Object Oriented Programming with C++ and Java

Course Designer and Acquisition Editor

Centre for Information Technology and Engineering

Manonmaniam Sundaranar University

Tirunelveli

CONTENTS

Lecture - 1

Introduction to Programming Languages

Objectives

In this lecture you will learn the following

- Introduction to Programming Languages
- Procedure Vs Object Oriented Programming
- Data Abstraction & Encapsulation

Coverage Plan

Lecture - 1

1.1 Snap Shot

There is an explosion in information technology in the last few years and this will continue in the new millennium, which has already dawned. The cause for the bounds and leaps in information technology is the introduction of powerful microprocessors and the Internet. The development in information technology has changed the boundaries and the world has shrunk in its size. Hence, this era of information technology will belong to the computer professionals who speak through the machine.

Information cannot be defined precisely. It is related to ideas and meaning that could be communicated, processed or converted into different forms. Moreover, the information may be presented in different ways in different languages. So it becomes necessary for one to understand the language otherwise the information becomes meaningless or it may be uninformative. A computer professional who speaks to the computer has to speak in specially designed languages that could be understood by the computer.

Software

As mentioned in the introduction, computers can understand only when the information is given in a language that the computer can understand. In order to make the man-machine communication easier the languages are developed. Generally languages use words and statements that are normally being used in communication. These types of languages are called the high level languages. The information given in a high level language can be easily understood by the user, while the machine can understand only when it is translated into binary forms like 0s and 1s.

The translated binary form is called the low level language. A user can even give information in the binary form, the machine language. But it is difficult to write entire set of instructions using machine level language and assembly language. The instructions may also be given in mnemonic codes, like ADD, MUL, STO, GO, MOV etc. This is called the assembly language.

Following table illustrates the instruction for adding two numbers

High level language	Machine language*	Assembly language**
10 input a, b	001xxxxxx	COP A
20 c = a + b	010xxxxxx	ADD B
30 write c	011xxxxxx	STO C
	100xxxxxx	STP

*In machine language the first three values indicate the type of instruction, while the last six values give the number or address. 001-> copy the value to memory location xxxxxxx

010 -> Add the number, 011-> store the value in memory, 100-> write the result.
** COP means copy, ADD means add, STO means store, STP means stop.

Using any language we can write software instructions. With the help of software user can write high-level program. A program is nothing but a sequence of instruction that will be executed one after other. Software can be classified into two types:

1. System software
2. Application software

System software: Directly interacts with the computer system. Operating system, compiler, interpreter are examples for this.

Application software: All the programs written by a user with the help of any software is called as application software. Eg. Balance sheet preparation for a company, monitoring rail way reservation process etc.,

Advantages of Computers

Computers have made a huge impact in the style of life. The question that can be raised is how does computers benefit the world. The main advantages of using computers are

- Speed
- Accuracy
- Diligence

Since computers work at a very fast rate, the speed plays a major role. They work at a very fast speed such that the average time taken by a computer is somewhat equivalent to one million mathematicians working a day. They seldom make any errors. Since they do not control the overall functioning, they are less flexible than humans. They have to be deliberately specified of what is going to be performed. If an unexpected situation takes place, it does not know what to do and it either produces error or simply come out of the task without completing it.

1.2 Introduction to Programming Languages

The shift in programming language is categorized as following:

- Monolithic Programming
- Procedural Programming
- Structural Programming
- Object Oriented Programming

Monolithic Programming (Assembly language and BASIC)

This programming consists only global data and sequential code. Program flow control is achieved through the use of jump and the program code is duplicated each time it is used. **Fig 1.1** No subroutine concept is used. Since this programming style is not supporting the concept of data abstraction it is very difficult to maintain or enhance the program code.

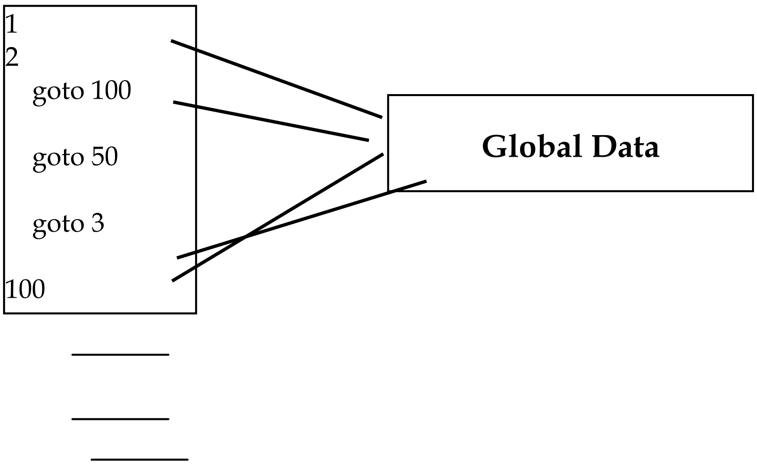


Fig. 1.1 Monolithic Programming

Procedural Programming (FORTRAN and COBOL)

Mainly comprises of algorithms. Programs were considered as important intermediate points between the problem and the computer in mid 1960s. Subprograms were originally seen as lab or saving devices but very quickly appreciated as a way to abstract program functions as shown in **Fig. 1.2**

The important features of Procedural Programming are

- Programs are organized in the form of subroutines and all data items are global
- Program controls are through jumps (goto's) and call subroutines
- Abstracted subroutines are used to avoid repetition
- Software application is minimized
- Difficult to maintain and enhance the program code

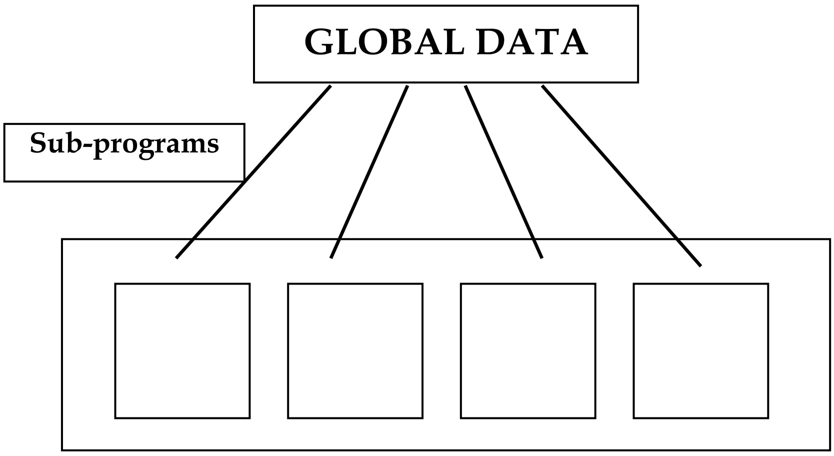


Fig 1.2 Procedural Programming

Structured programming (Pascal and C)

Structured programming is evolved as a mechanism to address the growing issues of programming in the large. Larger programming projects consist of large development teams, developing different parts of the same project independently. **Fig 1.3** Programs consist of multiple and in turn each module has a set of functions of related types.

- Structured programming is based upon the algorithm rather than data
- Programs are divided into individual modules that perform different task.
- Controls the scope of data
- Support modular programming
- Introduction of user defined data types

Technically, a structured language permits procedures and functions to be declared inside other procedures or functions, and therefore cannot formally be called a block-structured language. However, it is referred to as structured languages like ALGOL, Pascal and the likes.

Structured programming allows compartmentalization of code and data. This is a distinguishing feature of any structured language. It refers to the ability of a language to section off and hides all information and instructions necessary to perform a specific task from the rest of the program. Code can be compartmentalized in C++ using functions or code blocks. Functions are used to define and code separately, special tasks required in a program. This allows programs to be modular. Code block is a logically connected group of program statements that is treated like a unit

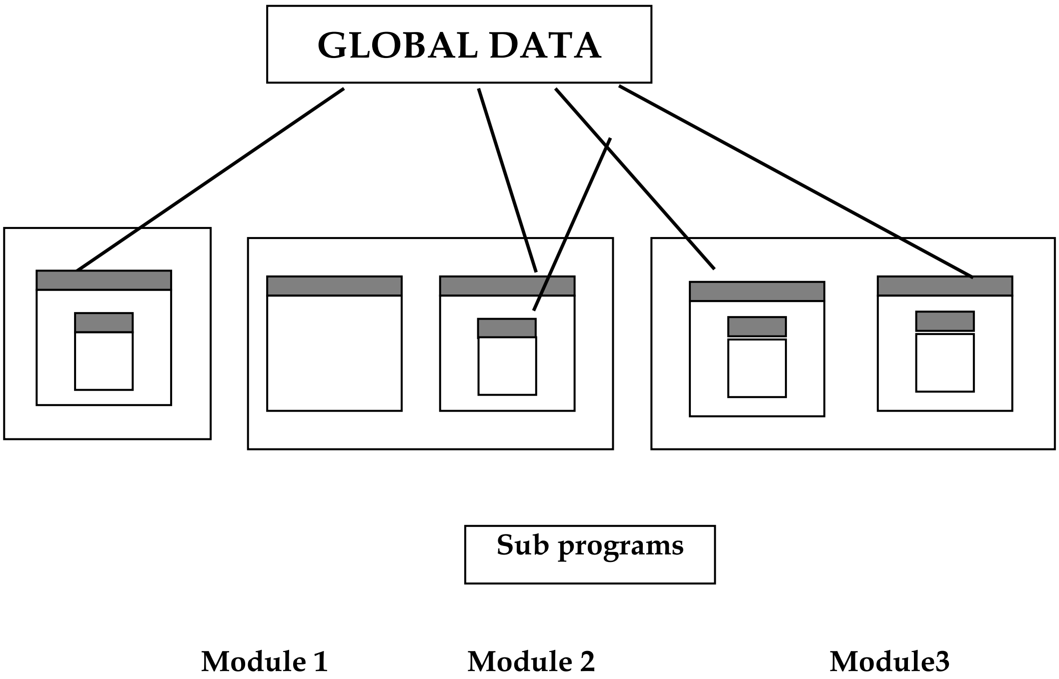


Fig 1.3 Structured Programming

Object Oriented Programming (C++, Smalltalk, Eiffel, Java etc.)

Approaches to programming have changed dramatically since the invention of computer, primarily to accommodate the complexity of the programs. Object-Oriented Programming is a new way of solving problems with computers. OOP is designed around the data being operated upon as opposed to the operations themselves.

The main objective of object-oriented programming is to eliminate some of the flaws encountered in the procedural approach. The object oriented programming has taken the best ideas of structured programming and combined them with several powerful concepts that encourage us to approach the task of programming in a new way. An **object** is a combination or collection of data and code designed to emulate a physical or abstract entity. Each object has its own identity and is distinguishable from other objects.

Definition of OOP:

Object oriented programming is a programming methodology that associates data structures with a set of operators, which act upon it.

OOP is a method of implementation in which programs are organized as co-operative collections of objects, each of which represents an instance of some class and whose classes are all members of a hierarchy of classes united through the property called inheritance.

Depending on the object features supported, the languages are classified into two categories:

- Object-Based Programming Languages
- Object-Oriented Programming Languages

Object-based programming languages support encapsulation object identity without supporting the important features of inheritance, polymorphism and message communications. Example ADA.

Object-based language = $\boxed{\textbf{Encapsulation}}$ + $\boxed{\textbf{Object Identity}}$

Object-Oriented Programming Language incorporate all the features of object-based programming languages along with inheritance and polymorphism.

Object-oriented programming language = $\boxed{\textbf{Object = based language + polymorphism + inheritance}}$

The topology of the Object Oriented Programming is shown in **Fig 1.4**. The modules represent the physical building blocks of these languages; a module is a collection of classes and object.

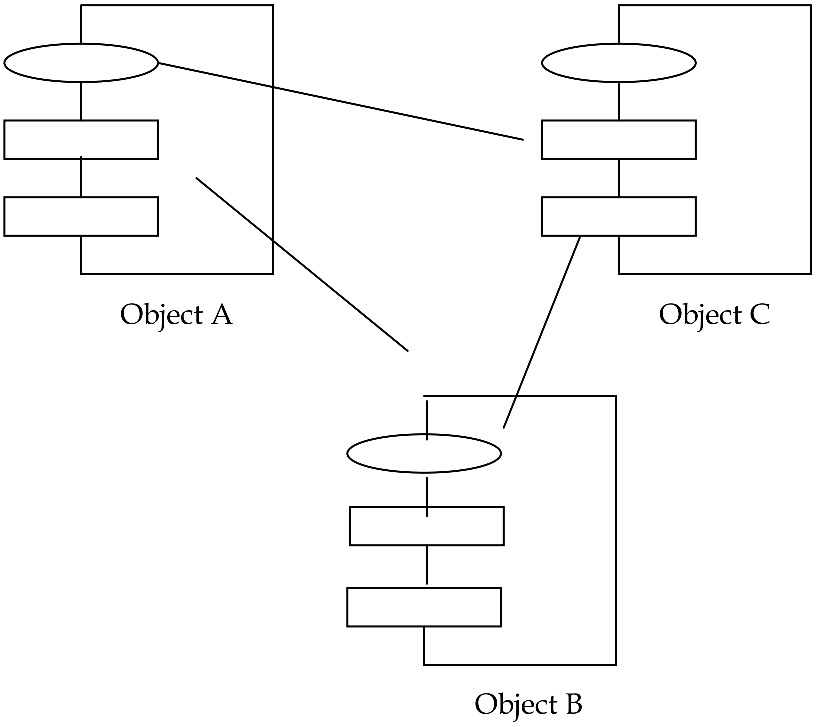


Fig 1.4 Object – Oriented Programming

Object oriented programming is a methodology that allows the association of data structures with operations similar to the way it is perceived in the human mind.

Features of Object-Oriented Programming

K Improvement of over the structured programming languages.

K Emphasis on data rather than algorithm

K Data abstraction is introduced in addition to procedural abstraction

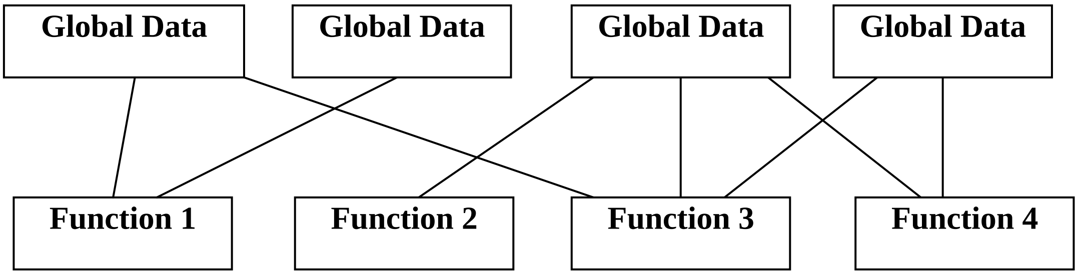
K Data and associated operations are unified into a single unit, thus the objects are grouped

K With common attributes, operations and semantics.

K Programs are designed around the data being operated, rather than operations themselves

1.3 Procedure versus Object-Oriented Programming

Program and data are the two basic elements of any programming language. Data plays an important role and it can exist without a program, but a program has no relevance without data. The conventional high-level languages stress on the algorithms used to solve a problem. Complex procedures have been simplified by structured programming. There are two paradigms that given how a program is constructed. The first way is called the process-oriented model. This approach characterizes a program as a series of linear steps. The process-oriented model can be taught of as code acting on data. Procedural languages are also called as Function oriented programming. (C language). The second approach is called object-oriented programming. It organizes a program around its data and a set of well-defined interfaces to that data. An object-oriented program can be characterized data controlling access to code.



Procedural Programming

Unlike Function oriented programming, Object oriented programming emphasizes on data rather than the algorithm. In OOP, data is compartmentalized or encapsulated with the associated functions and this compartment is called an object. In OO approach the problem is divide into objects, whereas in FOP the problem is divided into functions. OOP contains FOP and so OOP can be referred to as the super set of FOP.

OOP uses objects and not algorithms as its fundamental building blocks. Each object is an instance of some class. Classes allow the mechanism of data abstraction for creating new data types. Inheritance allows building of new classes from the existing class.

Unlike traditional languages OO languages allow localization of data and code and restrict other objects from referring to its local region. OOP is centered on the concepts of objects, encapsulation, abstract data types, inheritance, polymorphism, and message-based communication. An OO language views the data and its associated set of functions as an object and treats this combination as a single entity. Thus, an object is visualized as a combination of data and functions, which manipulate them. During the execution of a program, the objects interact with each other by sending messages and receiving responses.

1.4 Data abstraction and encapsulation

The wrapping up of data and methods into a single unit (called class) is known as **encapsulation.** Data encapsulation is the most striking feature of a class. The data is not accessible to the outside world and only those methods, which are wrapped in the class, can access it. These methods provide the interface between the object's data and the program. This insulation of the data from direct access by the program is called **data hiding**. Encapsulation makes it possible for objects to be treated like "black boxes" each performing a specific task without any concern for internal implementation.

Information in
Data and method
Information out

Abstraction refers to the act of representing essential features without including the background details or explanations. Classes use the concept of abstraction and are defined as a list of abstract attributes such as size, weight and cost and methods that operated on these attributes. They encapsulate all the essential properties of the objects that are to be created.

1.5 Short Summary

- In Procedural Programming Programs are organized in the form of subroutines and all data items are global.
- Structured programming allows compartmentalization of code and data. This is a distinguishing feature of any structured language.
- An **object** is a combination or collection of data and code designed to emulate a physical or abstract entity.
- OOP is a method of implementation in which programs are organized as co-operative collections of objects, each of which represents an instance of some class and whose classes are all members of a hierarchy of classes united through the property called inheritance.

1.6 Brain Storm

1. Discuss the features of Object oriented programming
2. List the pros and cons of object oriented programming over the structured programming.
3. What are the salient features of the object oriented design?
4. Illustrate the difference between system software and application software.
5. Give some examples for Object oriented programming languages.
6. How will you differentiate Data Abstraction and Encapsulation ?

Lecture - 2

Introduction to OOP

Objectives

In this lecture you will learn the following

- Knowing about Inheritance & Polymorphism
- Polymorphism
- Advantages of OOP's
- Introduction to C++

Coverage Plan

Lecture - 2

2.1 Snap Shot

2.2 Inheritance

2.3 Polymorphism

2.4 Advantages of OOP

2.5 Introduction to C++

2.6 Advantages of C++

2.7 Applications of C++

2.8 First C++ Program

2.9 Short Summary

2.10 Brain Storm

2.1 Snap Shot

In this lecture you will learn about Inheritance, Polymorphism, Advantages of OOP, Introduction to C++, and a sample C++ Program.

2.2 Inheritance

Inheritance is the process by which object of one class acquire the properties of objects of another class. Inheritance supports the concept of hierarchical classification. For example, the bird robin is a part of the class flying bird, which is again a part of the class bird. As Illustrated in Fig2.1 the principle behind this sort of division is that each derived class shares common characteristics with the class from which it is derived.

In OOP, the concept of inheritance provides the idea of reusability. This means that we can add additional features to an existing class without modifying it. This is possible by deriving a new class from the existing one. The new class will have the combined features of both the classes. Thus the real appeal and power of the inheritance mechanism is that it allows the programmer to reuse a class that is almost, but not exactly, what he wants, and to tailor the class in such a way that it does not introduce any undesirable side effects into the rest of the classes. In Java, the derived class is known as subclass.

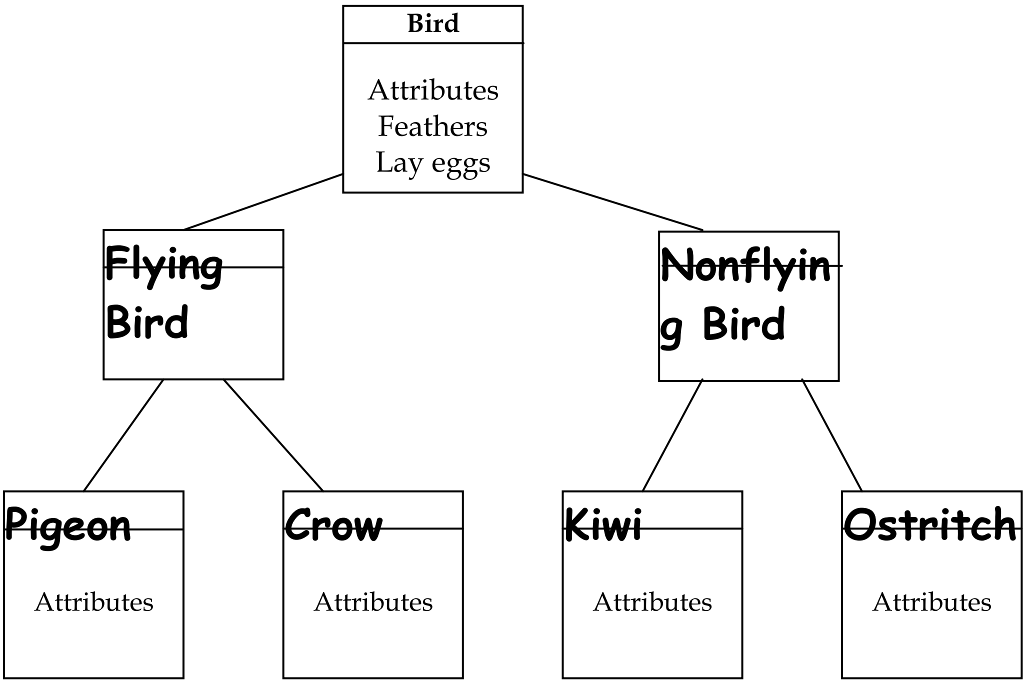


Fig 2.1 Property of Inheritance

Example:

// Sample program for inheritance
//source file: inheritance.cpp

```
class building
{
        int rooms;
        int floors;
        int areas;
        public:
                void set_rooms(int num);
                int get_rooms();
                void set_floors(int num);
                int get_floors();
                void set_area(int num);
                int get_area();
};
class house:public building    // class house is the inherited class of building
{
        int bedrooms;
        int baths;
        public:
                void set_bedrooms(int num);
                int get_bedrooms();
                void set_baths(int num);
                int get_baths();
};
void building::set_rooms(int num)    //function definition part
{
        rooms = num;
}
int building:: get_rooms()
{
        return rooms;
}
void building::set_floors(int num)
{
        floors = num;
}
int building::get_floor()
{
        return floors;
}
void building::set_area(int num)
{
        area = num;
}
int building::get_area()
{
        return area;
}
void house::set_bedrooms(int num)
{
        bedrooms = num;
}
int house::get_bedrooms()
{
        return bedrooms;
}
void house::set_baths(int num)
{
        baths = num;
}
int house::get_baths()
{
        return baths;
}
void main()
{
        house h1;
        h1.set_rooms(10);
        h1.set_floors(5);
        h1.set_bedrooms(10);
        h1.set_baths(5);
        h1.set_area(950);
                cout<<"number of rooms"<<h1.get_rooms();
                cout<<"number of floors"<<h1.get_floors();
                cout<<"number of bedrooms"<<h1.get_bedrooms();
                cout<<"number of baths"<<h1.get_baths();
                cout<<"area of a house"<<h1.get_area();
        return 0;
}
```

The above example clearly states the concept of inheritance. Since class house is the inherited class of building it can able to access the building class members(base class) and its member functions.

2.3 Polymorphism

Polymorphism is another important OOP concept. Polymorphism means the ability to take more than one form. For example, an operation may exhibit different behavior in different instances. The behaviour depends upon the types of data used in the operation. For example, consider the operation of addition. For two numbers, the operation will generate a sum. If the operands were strings, then the operation would produce a third string by concatenation. **Fig.** 2.2 illustrates that a single function name can be used to handle different number and different types of arguments. This is something similar to a particular word having several different meanings depending on the context.

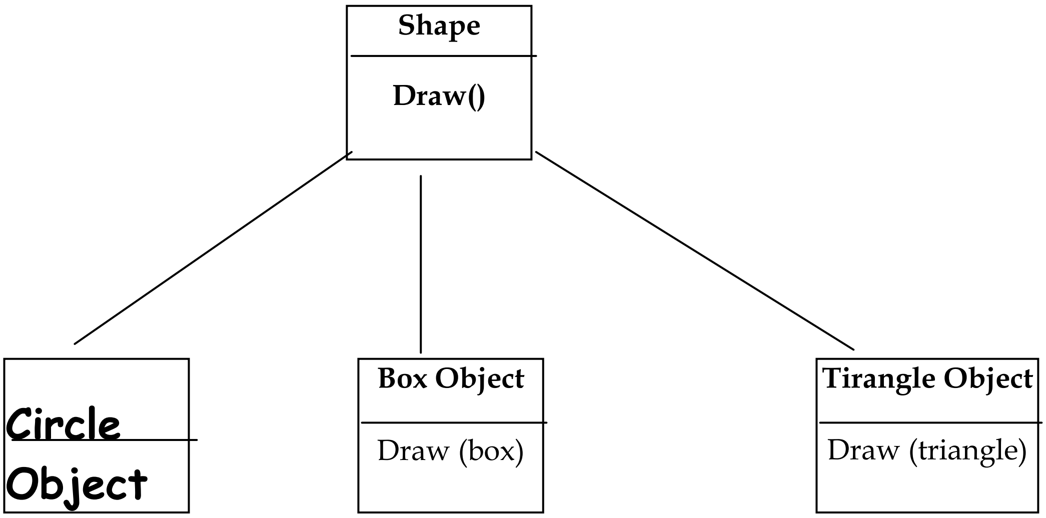


Fig 2.2 Polymorphism

Polymorphism plays an important role in allowing objects having different internal structures to share the same external interface. This means that a general class of operations may be accessed in the same manner even though specific actions associated with each operation may differ. Polymorphism is extensively used in implementing inheritance.

Example:

```
//sample program for polymorphism
//source file: poly.cpp
# include <iostream.h>
int abs(int I);
long abs (long l);
double abs(double d);
main()
```

```
{
        cout<<abs(-10)<<"\n";
        cout<<abs(10L)<<"\n";
        cout<<abs(-12.35)<<"\n";
return 0;
}
int abs(int i)
{
        cout<<"using int method";
        return i>0?i:-i;
}
long abs(long l)
{
        cout<<"using long method";
        return l>0?l:-l;
}
double abs(double d)
{
        cout<<"using double method";
        return d>0?d:-d;
}
```

This example proves the concept of polymorphism, i.e, the function abs() is same, but it acts differently in different places depending upon the argument we pass.

2.3 Advantages of OOP

- Through inheritance we can eliminate redundant code and extend the use of existing classes.

- We can build programs from the standard working modules that communicate with one another rather than having to start writing the code from scratch. This leads to saving of development time and higher productivity.

- The principle of data hiding helps the programmer to build secure programs that cannot be invaded by code in other parts of the program.

- It is possible to have multiple objects to coexist without any interference.

- It is easy to partition the work in a project based on objects.

- The data centered design approach enables us to capture more details of a model in an implementable form.

- Object-oriented systems can be easily upgraded from small to large systems.

- Message passing techniques for communication between objects make the interface descriptions with external systems much simpler.

- Software complexity can be easily managed.

- Dynamic binding increases flexibility by permitting the addition of a new class of objects without having to modify the existing code.

- Code reuse is possible in conventional languages as well, but Object Oriented languages greatly enhance the possibility of reuse.

- Object Orientation provides many other advantages in the production and maintenance of software; high degree of code sharing.

2.5 Introduction to C++

This lecture provides an overview of the key concepts embodied in C++. C++ is an object-oriented programming language, and its object-oriented features are highly interrelated. In several instances, this interrelation makes it difficult to describe one feature of C++ without implicitly involving several others. In many places, the object-oriented features of C++ are so intertwined that discussion of one feature implies prior knowledge of one or more other features. To address this problem, this lecture presents a quick overview of the most important aspects of C++.

The origin of C++

C++ is an expanded version of C. Bjarne Stroutstrup first invented the C++ extensions to C in 1980 at Bell Laboratories in Murray Hill, New Jersey. He initially called the new language as "C with Classes". However, in 1983 the name was changed to C++.

Although C++'s predecessor, C, is one of the most liked and widely used professional languages in the world, the invention of C++ was necessitated by one major programming factor: increasing complexity. Over the years, computer programs have become larger and more complex. Even though C is an excellent programming language, it too has its own limits. In C, once a program exceeds from

25,000 to 100,000 lines of code, it becomes so complex that it is difficult to grasp as a totality. The purpose of C++ is to allow this barrier to be broken. The essence of C++ is to allow the programmer to comprehend and manage larger, more complex programs. Most additions made by Stroustrup to C support object-oriented programming sometimes referred to as OOP.

Although C++ was initially designed to aid in the management of very large programs, it is no way limited to this use. In fact, the object-oriented attributes of C++ can be effectively applied to virtually any programming task. Also, because C++ shares C's efficiency, much high performance systems software is constructed using C++.

2.6 Advantages of C++

- **Data abstraction :** In OOP, the data abstraction is defined as a collection of data and methods (functions).

- **Data hiding :** In C++, the class construct allows to declare data and member functions, as a public, private and protected group. The implementation details of a class can be hidden. This is done by the data hiding principle.

- **Data encapsulation :** The internal data (the member data) of a class are first separated from the outside world(the defined class). They are then put along with the member functions in a capsule. In other words, encapsulation groups all the pieces of an object into one neat package. It avoids undesired side effects of the member data when it is defined out of the class and also protects the intentional misuse of important data. Classes efficiently manage the complexity of large programs through encapsulation.

- **Inheritance :** C++ allows a programmer to build hierarchy of classes. The derivation of classes is used for building hierarchy. The basic features of classes (parent or base classes) can be passed onto the derived classes (child classes). In practice, the inheritance principle reduces the amount of writing; as the derived classes do not have to be written again.

- **Polymorphism :** In OOP, polymorphism is defined as how to carry out different processing steps by a function having the same messages. Polymorphism treats objects of related classes in a generic manner.

2.7 Applications of C++

- Real time systems
- Simulation and modeling
- Object-oriented databases
- AI and Expert Systems
- Neural networks and parallel programming
- CAD/CIM system

2.8 First C++ Program

Welcome Program

```
// hello.cpp: displaying Hello World message
#include<iostream.h>           //preprocessor directive statement
void main()                    // function declarator
{                              //function block open brace
  cout<< “Welcome to C++";     // output statement
}                              //function block close brace
```

the output will be

```
Welcome to C++
```

Program Description

C++ programs must contain a function called main(), from which execution of program starts. The function main() is designated as the starting point of the program execution and the user defines it. It cannot be overloaded and its syntax type is implementation dependent. Therefore, the number of arguments and their data-type is dependent on the compiler. Let’s have a close at a very simple C++ program which displays Hello World on the screen: -

The header file iostream.h supports streams programming features by including pre-defined stream objects. The C++’s stream insertion operator, << sends the message “Welcome to C++” to the pre-defined console object, count, which in turn prints on the console.

The various components of the program hello.cpp are discussed in the following section:

Comment Line

The statement that starts with symbols **//** is treated as comment. Hence, the compiler ignores the complete line starting from the **//** character pair.

The character **.cpp** in hello. tells the compiler that it is a C++ program. (However, the extension is compiler dependent).

Preprocessor Directive

The preprocessor directive **#include<iostream.h>** includes all the statements of the header file **iostream.h.** It contains instructions and predefined constants that will be used in the program. It plays the same role as that of the **stdio.h** of C. The header file **iostream.h** contains declarations that are needed by the **cout** and **cin** objects. There are a number of such preprocessor directives provided by the C++ library, and they have to be included depending on the built-in functions used in the program. In effect, these directives are processed before any other executable statements in the source file of the program by the compiler.

Function Decelerator

The third line of the program is

```
void main()
```

The C++ program consists of a set of functions. Every C++ program must have one function with name main, from here the execution of the program begins. The name main is a special word (not a reserved word) and must not be invoked anywhere by the user. The names of the functions (except main) are coined by the programmer. A pair of parentheses, which may or may not contain arguments, follows the function name. In this case, there are no arguments, but still the parenthesis pair is mandatory. Every function is supposed to return a value, but the function in this example does not return any value. Such function names must be preceded by the reserved word void.

Compilation Process

The C++ program hello.cpp, can be entered into the system using any available text editor. Some of the most commonly available editors are Norton editor (ne), edline,edit,vi (most popular editor in UNIX environment). The program coded by the programmer is called the source code. This source code is supplied to the compiler for converting it into the machine code.

C++ programs make use of libraries. A library contains the object code of standard functions. The object codes of all functions used in the program have to be combined with the program written by the programmer. In addition, some start-up code is required to produce an executable version of the program. This process of combining all the required object codes and the start-up is called linking and the final product is called the executable code.

Most of the modern compilers support sophisticated features such as multiple window editing, mouse support, on-line help, project management support, etc. One such compiler is Borland C++. It can be invoked through command-line integrated development environment.

Command – Line Compilation

Most of the compilers support the command line compilation of a program. All the required arguments are passed to the compiler from the command line. For the purpose of discussion, consider the Borland C++ compiler. (however this process is implementation dependent.)

The command – line compiler is invoked by issuing the command:

```
tcc filename.cpp (in the case of Turbo C++)

bcc filename.cpp (in the case of Borland C++)
```

at the DOS prompt. It creates an object file filename.obj, and an executable file through the explicit issue of the linking command :

```
tlink filename1.obj filename2.obj <library name>
```

The library file can also be passed as a parameter to the linker for binding functions defined in it. To create the executable of hello.cpp, issue the command bcc hello.cpp at the MS-DOS prompt.

2.9 Short Summary

- Inheritance is the process by which object of one class acquire the properties of objects of another class.

- Polymorphism plays an important role in allowing objects having different internal structures to share the same external interface

- C++ is an expanded version of C. Bjarne Stroutstrup first invented the C++ extensions to C in 1980 at Bell Laboratories in Murray Hill, New Jersey.

- C++ programs must contain a function called main(), from which execution of program starts.

2.10 Brain Storm

1. Give an account of the concept of Inheritance.
2. Define: Polymorphism.
3. List some of the advantages of OOP
4. What are the applications and advantages of C++?
5. Write a C++ program that prints the message " This is my First CPP program".

Lecture - 3

Introduction to Java

Objectives

In this lecture you will learn the following

- Understanding the concept of Java
- Pros & Cons of Java
- Knowing the applications of Java

Coverage Plan

Lecture - 3

3.1 Snap Shot

In this Lecture you will be introduced in to the Introduction of Java, its Advantages, its Applications and finally with a Java Program.

3.2 Introduction to Java

The history of Java

Java was developed as software for interactive TV programs and VCRs. Users could interact with on-screen pictures, change the camera angle to their choice; choose the time to watch programs etc. A microprocessor chip was required to be embedded in the TV system and the software to run the programs.

James Gosling, engineer at the Sun Systems, thought of novel idea that would free manufacturers not to be dependent on a particular brand of the chip used in the TV. In January 1991, Bill Joy, co-founder of Sun micro system , along with James Gosling and Patrick Naughton, provided the specifications for the software to be used for these interactive TVs.

The highlights of the specification were that the software should be compatible with all the existing hardware at the same time occupy as little memory space as possible.

Between February 1991 and September 1992, an operating system was developed along those guidelines, which was called Green and a programming, language interpreter called the Oak. But later Oak was renamed as Java in 1995.

The primary motivation for Java was the need for a platform-independent (ie. Architecture-Neutral) language that could be used to create software to be embedded in various consumer electronic devices, such as microwave ovens and remote controls.

Gosling's team created a web browser, called the Hot Java using Java, that would download Java programs nesting on the web pages and use them to animate information on the pages.

What is Java?

Java is simple, object-oriented, distributed, interpreted, robust, secure, architecture-neutral, portable, high-performance, multithreaded and dynamic language.

3.3 Advantages of Java

- Java is a true object-oriented language. Almost everything in Java is an object. All program code and data reside within objects and classes. The object model in Java is simple and easy to extend.
- Portable: Java programs can be moved from one computer system to another anywhere and anytime.
- Platform-Independent
- Write once, run anywhere: As java programs are compiled into machine-independent byte codes, they run consistently on any java platform.
- Write robust and reliable programs
- Build an application on almost any platform and run that application on any other supported platform without having to recompiling your code.
- Distribute your applications over a network in a secure fashion

3.4 Applications of Java

- Java is designed as a distributed language for creating applications on network.
- It has the ability to share both the data and programs.
- Java applications are open and access remote objects on Internet as easily as they can do in a local system.
- It supports multithreading.
- Multithreading means handling multiple tasks simultaneously. We need not wait for the application to finish one task before beginning another.
- Java and Internet
- Java and World Wide Web
- WWW is an open-ended information retrieval system designed to be used in the Internet's distributed environment. Java can be used in distributed environment. Both Java and Web share the common technology.

3.5 First Java Program

```
//file name must be FirstJavaPgm.java
class FirstJavaPgm
{
        public static void main(String args[])
        {
                System.out.prinln("Welcome to the world to Java programming"):
        }
}
```

- Save the file as FirstJavaPgm.java
- Compile the program using javac (javac compiler JDK tool)
  C:\javac FirstJavaPgm.java
- Run the program using java (java interpreter JDK tool)
  C:\java FirstJavaPgm

The output is:

Welcome to the world of java programming

Description of the above program:

The first line "//file name must be FirstJavaPgm" is a comment line. The compiler ignores the line.
Comment line can be indicated by

- // Single line comment
- /* cl$_1$
  cl$_2$
  cl$_3$ **Multiline comment**
  cl$_n$
  */
- /** ………..*/ Documentation comment

The second line class FirstJavaPgm

Declares a class, which is an object-oriented construct. "FirstJavaPgm" is user defined class name. A java program may contain multiple class definitions. Classes are the primary and essential element of a Java program. These classes are used to map the objects of real world problems.

Opening Brace
Every class definition in Java begins with an opening brace "{" and ends with a matching closing brace "}" appearing in the last line in the example. This indicates the beginning and closing of any block.

The Main Line

public static void main(string args[ ])

- The main method must be declared as public, since it must be called by outside of its class

- The keyword static allows main() to be called without having to instantiate a particular instance of the class. This is necessary since main() is called by the Java interpreter before any objects are made.

- The key word void tells the compiler that main() does not return any value.

- The argument to main() is an array of string objects. Whether the command line arguments are used in this program or not, but they have to be there because they hold the arguments invoked on the command line.

The Output line

The only essential statement in the program is

"System.out.println("Welcome to the world of Java programming");

This is similar to the **printf()** statement of C. Since Java is a true Object oriented programming language every method is a part of an object. The **println** method is a member of the class out object, which is a static data member of System class. This line prints the string as

Welcome to the world of Java Programming

on the screen.

3.6 Short Summary

- Java is simple, object-oriented, distributed, interpreted, robust, secure, architecture-neutral, portable, high-performance, multithreaded and dynamic language.

- Java is a true object-oriented language , Portable , Platform Independent .

- Java is designed as a distributed language for creating applications on network.

- It has the ability to share both the data and programs.

3.7 Brain Storm

1. Explain the advantages, application of Java programming.

2. Name the compiler and interpreter used in Java?

3. Write a Java program that prints the message "This is my First Java Program".

Lecture - 4

Variables, Operators & Data types

Objectives

In this lecture you will learn the following

- Knowing the differences between C++ and Java
- Knowing about variables shows how to passing operators

Coverage Plan

Lecture - 4

4.1 Snap Shot

Operators to form expressions manipulate variables and constants. These are the basic elements of the C++ language, which can be combined to form comprehensive program components. In this topic, we will discuss the concepts of constants, variables and their types as they relate to C++ programming language.

4.2 Differences between C++ and Java

1. Java is a true object-oriented language while C++ is basically C with object-oriented extension.
2. Java does not support operator overloading where as C++ supports operator overloading
3. Java does not have template classes but C++ has templates
4. Java does not support multiple inheritance, it can be achieved using "interface" concept. C++ supports multiple inheritance.
5. No global variable declaration in Java, the variables and method declared in each class forms part of a class. C++ accepts global declaration of variables.
6. No pointer concept is used in Java. C++ has pointer concept
7. Java does not have destructor function it has been replaced by finalize ( ) function. C++ has destructor function.
8. No header file is used in Java. C++ uses header files to include all library files.

4.3 Basic Elements of C++

Character Set

Characters are used to form the words, numbers and expressions. The characters in C++ are grouped into the following categories: -

1 Character set alphabets from A.....Z,a....z
all decimal digits from 0....9characters, . ; : ?

‘ “ !| / \ ~ _ $ % #& ^

* - + < > () [ ] { }Spaces
blank
New Line endl

Keywords and Identifiers

C++ word is classified as either a keyword or an identifier. All identifiers have fixed meaning and these meanings cannot be changed. Keywords serve as the basic building blocks for program statements. The list of all keywords is listed below. All keywords must be written in lowercase.

auto	double	int	struct	break
else	long	switch case	enum	register
typedef	char	extern	return	union
const	float	short	unsigned	continue
for	signed	void	default	goto
sizeof	volatile do	if	static	while

Identifiers refer to the names of variables, functions and arrays. These are user - defined names and consist of a sequence of letters and digits.

Examples of identifiers: -

Valid identifiers	Invalid identifiers
Count	1count (digit, as a first letter is not allowed)
test23	hi! (Special characters except under score are not allowed)
high_bal	high balance (no spaces allowed)

Data Types

A data type defines a set of values that a variable can store along with a set of operations that can be performed on that variable. C++ has five basic built in data types: -.

Character (char)
Integer (int)
Floating point (float)
Double floating point (double)
Valueless (void)

Void has three uses: -.

To declare explicitly a function as returning no value.
To declare explicitly a function as having no parameters.
To create generic pointers.

C++ also supports several aggregate types including structures, unions, enumeration and user defined types.

Type Modifiers

Excepting type void, the basic data types may have various modifiers preceding them. A modifier is used to alter the meaning of the base type to fit the needs of various situations more precisely. The list of modifiers is shown here: -

signed
unsigned
long
short

Modifiers can be applied to character and integer base types. However, long can also be applied to double.

Type	Bit Width	Range
char	8	-128 to 127
int	16	-32768 to 32767
unsigned int	16	0 to 65535
signed int	16	-32768 to 32767
short int	16	-32768 to 32767
unsigned short int	16	0 to 65535
signed short int	16	-32768 to 32767
long int	32	-2147483648 to 2147483647
signed long int	32	-2147483648 to 21483647
float	32	3.4E-38 to 3.4E+38
double	64	1.7E-308 to 1.7E+308
long double	64	1.7E-308 to 1.7E+308

Constants

Constants in C++ refer to fixed values that do not change during the execution of a program. C++ supports several types of constants as shown below: -

Numeric Constants

1. Integer Constants
2. Floating point (real) Constants

Character Constants

1. Single Character Constants
2. String Constants

The following rules apply to all numeric constants: -, non-digit characters and blanks cannot be included within a constant. A constant can be preceded by a minus sign.

Integer Constants

An integer constant refers to a sequence of digits. There are three types of integers, namely, decimal, octal and hexadecimal.

Decimal integers consist of a set of digits from 0 through 9. The decimal integers can be positive or negative.

Example:

1234
0
+456
-4238

Octal integers consist of any combination of digits from 0 through 7, with a leading zero.

Example:

089
0456
0
0555

Hexadecimal integers consist of a sequence of digits 0 through 9 and alphabets from A (a) through F (f). These integers are preceded by 0x or 0X. The letters 'A' (a) through 'F ' (f) represent the integers 10 through 15.

Example:

0X3
0XBC
0x76f

The largest integer value that can be stored is machine dependent. It is 32767 on 16-bit machine and 2,147,483,647 on 32_bit machine. It is also possible to store large integer constants on these machines by appending modifiers such as U, L, UL to the constants.

Example:

23456U(u)	unsigned integer
987654321UL(ul)	unsigned long integer
928374L(l)	long integer

Floating-Point Constants

Floating-point constants are represented by numbers containing fractional parts like in 549.4545. Floating-point constants are also sometimes called as real number constants.

Example:

0.00098
3210.67
-56.890
+345.9

A real number may also be expressed in exponential notation. For example, the value 2345.78 may be written as 23.4578e2 in exponential notation. e2 means multiply by 100. The number 3e-2 would mean 3*1/100.

Some of the valid floating - point constants are: -

0.	(digit after the decimal point can be omitted)
.00045	(digit before the decimal point can be omitted)
7.31245e +3	(the notation e3 can also be written as e + 3)
2E-8	(e or E is allowed)

Some of the invalid floating - point constants are: -

12	(decimal point or an exponent must be present)
1,067.89	(comma is not allowed)
3.89e+1.5	(exponent must be an integer)
5E 12	(no embedded spaces are allowed)

Exponential notation is useful for representing numbers that are very large or very small in magnitude. For example, 45000000000 can be written as 45e9. Similarly, -0.000000454 is equivalent to -4.54e-7.

Single Character Constants

A character constant is a single character enclosed within a pair of single quotes.

Example:

'A'
'3'
'?'
';'
' '

Character constants have integer values called ASCII values. For example, the character constant '3' has an ASCII value of 83.Therefore, it is not the same as the integer value 3. Some of the character constants and their corresponding ASCII values are: -

<u>Constant</u>	<u>ASCII value</u>
'a'	97
'A'	65
'&'	38
';'	59

String Constants

A string constant is a sequence of characters enclosed within a pair of double quotes. The string constant may also include special characters, numbers and blank spaces.

Example:

```
" Hello!"
" I'm going for shopping today. Will you come?"
" 549, The Mall, Shimla."
" 4321-1234"
" O"
```

String constant "O" differs from the character constant 'O'. The string constant does not have an integer value like a character constant has. A string constant is always followed by a null character ('\0').

The following program accepts your name, address and phone number as string constants and displays them: -

```
#include <iostream.h>

void main(void)
{
   cout<<" Name:         Rajeev\n";
   cout<<" Address: 549, ABIDS, Hyderabad\n";
   cout<<" Contact:  040-211717\n";
}
```

The output is:

```
Name:     Rajeev
Address: 549, ABIDS, Hyderabad
Contact:  040-211717
```

Escape Sequences

Certain nonprinting characters, as well as the backslash (\) and the apostrophe ('), can be expressed in terms of escape sequences. An escape sequence always begins with a backward slash and is followed by one or more special characters. For example, a newline can be referred to as \n. Such escape sequences always represent single characters, even though they are written in terms of two or more characters. The commonly used escape sequences are listed below: -

Character	**Escape sequence**
bell	\a
backspace	\b
horizontal tab	\t
vertical tab	\v
newline	\n
carriage return	\r
quotation mark	\"
apostrophe	\'
backslash	\\
null	\0

Escape sequences are expressed in terms of character constants: -

```
'\t','\n','\"',        '\'','\0'
```

Particular interest is the escape sequence \0. This represents the null character, which is used to signify the end of a string. Null character is not equivalent to the character constant ' 0 '.

4.4 Variables

A variable is an identifier that is used to represent a single data item i.e. a numeric quantity or a character constant. The data item must be assigned to the variable at some point of time in the program. This data item can then be accessed later in the program simply by referring to the variable name. Unlike constants that remain unchanged during the execution of a program, a variable may take different values at different times.

All variables must be declared before they are used in the program. The general form of a declaration is shown as: -

```
Data-type var_list;
```

Data type must be a valid data type and var_list may consist of one or more identifier names with comma operators. The declaration tells the compiler what the variable names are and what type of data they hold. Some declarations are: -

```
int count;
short int num;
```

```
double balance;
float amount;
```

The words int, short int, double and float are keywords and cannot be used as variable names.

Assigning Values to Variables

Values can be assigned to variables using the assignment operator (=): -

```
var_name=expression;
```

An expression can be a single constant or a combination of variables, operators and constants. The statements:

```
p1=3250.00;.    (the expression is a single numeric constant)
p2=4570.00;
p3= p1+p2;      (the expression is a sum of price1 and price2)
```

are called assignment statements. Every statement must have a colon at the end. A statement implies that the value of the variable on the left of = is set to the value of the expression on the right. Therefore,

```
count=count+1;
```

implies that the 'new value' of count (left side) has been set to the 'old value' of count plus 1 (right side).

The variables can be assigned a value at the time of the declaration itself. This is called initialization. The general format is: -

Data-type var_name=constant;

Example:

```
int value=250;
char name="Gaurav";
double amount=76.80;
```

Program to show declarations, assignments and initialization of various types of variables: -

```
# include <iostream.h>
void main(void)
{
 /*.......Declarations....... */
float a, b;
double c, d;
unsigned u;
/*.......Initialization....... */
x = 19283;
int l = 9182736;
/* ....... Assignments.......*/
a= 2345.987;
b= 1928.283647;
d= c =2.20;
u= 54637;
/* .......Displaying.......*/
cout<<"x ="<<x;
cout<<"l ="<<l;
cout<<"a ="<<a;
cout<<"d ="<<d;
cout<<"u ="<<"b ="<<"c =\n"<<u<<b<<c;
}
```

The output will be:

```
: x = 19283    l = 9182736  a= 2345.987000  d= 1928.283647000000  u= 54637  b=
2.200000  c = 2.200000000000
```

4.5 Operators

C is very rich in built-in operators. An operator is a symbol that tells the compiler to perform specific mathematical or logical manipulations. There are three general classes of operators in C: -

Arithmetic Operators
Relational and Logical Operators
Assignment Operators

In addition, C has some special operators to perform particular tasks: -

Ternary(Conditional) Operator
Sizeof Operator
Comma Operator
& Operator
And -> Operators

Arithmetic Operators

The table below lists the arithmetic operators provided in C++. They can be applied to any built in data_type allowed by C++. The unary minus, in effect multiplies its single operand by -1. Therefore, a number preceded by a minus sign changes its sign.

Operator	*Action*	*Example*
-	Subtraction, also unary minus	a- b, -a
+	Addition	a + b
*	Multiplication	a * b
/	Division	a/b
%	Modulus Division	a%b
--	Decrement	a-- or --a
++	Increment	a++ or ++a

The modulus division (%) operation yeilds the remainder of an integer division. However, % cannot be used on type float or double.

The following code fragment illustrates the use: -

```
# include <iostream.h>
void main()
{
        int x, y;
        x=10; y=3;
        cout<<x / y;     /* displays 3, division of two integers*/
        cout<<x % y); /* displays 1, the remainder of the integer division */
         x=1;  y=2;
        cout<<x / y);  /* displays 0, integer division 1/2 yeilds 0 */
        cout<< x % y;  /* displays 1, the remainder of  integer division */
}
```

The output is:

```
3       10      1
```

Increment and Decrement Operators

C allows two very useful operators not usually found in other languages. These are increment and decrement operators, ++ and --. The operation ++ adds 1 to its single operand, and -- subtracts 1. Therefore, the following are equivalent operations: -

```
x = x+1;
```

is the same as

```
x++; or ++x;
```

and

```
x = x - 1;
```

is the same as

```
x--; or --x;
```

The increment and decrement operators may either precede or follow the operand. However, there is a difference when they are used in expressions. When an increment or decrement operator precedes its operand, C++ performs the increment or decrement operation prior to using the operand's value. If the operator follows its operand, C uses the value of the operand's value after incrementing or decrementing it.

Consider the following example:

```
x = 10;
y = ++x;
```

In this case, y is set to 11.

However, if the code had been written as

```
x = 10;
y = x++;
```

y would have been set to 10. In both the cases, x is set to 11; the difference is, when it happens.

Let's us see how different kinds of arithmetic operators can be used in a program:-

```
# include <iostream.h>
main()
{
     int a, b, c, d, x, y, p, q;
     a=20; b=35; c= a+b; d= b-a;
    cout<<"******Addition & Subraction*****";
    cout<<"a = "<<"b =\n"<< a<< b;
    cout<<" c = "<<"d = \n"<<c<<d;
    x = b / a;
    y = b % a;
    cout<<"******Division & Modulus******";
```

```
            cout<<"x =\n"<<x;
            cout<<"y = \n"<<y);
            p=++a;
            q= c + b--;
            cout<<"******Increment & Decrement******";
            cout<<"a = \n"<<a);
            cout<<"b =\n"<<b);
            cout<<"p = \n"<<p);
            cout<<"q = \n"<<q);
    }
```

The output is:

```
Addition & Subraction a=20  b = 35  c= 55  d = 15
Division & Modulus   x= 1  y= 15
Increment & Decrement a=  21  b= 34  p= 36  q=  71
```

Relational and Logical Operators

In Relational Operators, the word relational refers to the relationships that the values can have with one another. Depending on the relations, the values can be compared. This comparison results either in true or false.

Example:

10 > 20 (false)
10 < 20 (true)

Operators like >, < used to compare values are called relational operators. C supports six relational operators.

The operators and their meaning is listed in the table given below: -

Operator	Meaning
<	is less than
<=	is less than or equal to
>	is greater than
>=	is greater than or equal to
= =	is equal to
! =	is not equal

In Logical Operators, the word logical refers to the ways relationships can be connected together using the rules of formal logic. C++ has three logical operators: -

Operator	Meaning
&&	logical AND
| |	logical OR
!	logical NOT

exp1 && exp2 evaluates to true only if both the expressions are true
x= 10; y = 12;

x < 12 && y > 10 evaluates to true since the individual expression is true

x < 5 && y = = 12 evaluates to false since the first expression is false.

exp1 | | exp2 evaluates to true if either of the expression is true
x= 10; y= 12;

x > 12 | | y > 10 evaluates to true since at least one expression is true

x = = 20 | | y != 12 evaluates to false since neither expression is true.

! exp evaluates to true if its value is not equal to the expression

!x = 10 false since x is 10

!x = 12 true since x is not 12

&& and | | are used when we want to test more than one condition and make decisions.

Example:

a > b && c = 12

evaluates to true only if both expressions evaluate to true.
In C++, a true value is any value other than 0. A false value is 0.

Expressions using relational or logical operators return 0 for false and 1 for true. Therefore, the following code fragment is not only true, but also prints the integer value of true i.e. 1.

```
#include <iostream.h>
main()
{
        int x;
        x=100;
        cout<<(x = =100);
}
```

The output is : 1

Relational and Logical operators are lower in precedence than the Arithmetic operators. This means that when arithmetic expressions are used on either side of the relational operators, the arithmetic expression is evaluated first and then the results are compared. For example, 10 > 1+ 12 is evaluated only after evaluating 1+12 i.e., 13. Now, the resultant 10 > 13 is evaluated, which comes out to be false.

Assignment Operator

Assignment operator (=) is used to assign the result of an expression to a variable. It takes the form as: -

```
var_name = expression;
```

var_name is the name if the variable which will be assigned the value of expression by the assignment operator (=).

Example:

```
char name;
int x, y, z;
name = "Ashu";          /* the variable name is assigned the value,
Ashu */                x = 12;  /* x is assigned the value 12 */
y = 98;                 /* y is assigned the value 98 */
z = x + y               /* z is assigned the sum of x and y */
```

In addition to this, C++ has a set of 'shorthand' assignment operators of the form:

```
var_name oper= expression;
```

oper is a binary arithmetic operator. The operator oper= is called the 'shorthand' assignment operator. The assignment statement var_name oper= expression is equivalent to

var_name = var_name oper expression;

Example:

count = count + 1;

is equivalent to

count+ = 1;

The shorthand operator += means ' increment count by 1'.

More examples

Simple Assignment Operator	Shorthand Assignment Operator
x = x + 1;	x+ = 1;
x = x - 2;	x - = 2;
x = x * 12;	x * = 12;
x = x / y;	x / = y;
x = x % y;	x % = y;

Conditional (:?) Operator

C++ has a very powerful and convenient operator that can be used to construct conditional expressions. It is termed as conditional operator. Sometimes, it is also called ternary operator since it operates on three operands. The **:?** operator takes the form: -

exp1 ? exp2 : exp3

exp1, exp2, exp3 are expressions.

The ternary operator works like this: -
exp1 is evaluated. If it is true, exp2 is evaluated and becomes the value of the expression. If exp1 is false, then exp3 is evaluated and its value becomes the value of the expression.

Example:

```
x = 10;
y = x > 9 ? 100 : 200;
```

In this example, y will be assigned the value 100. If x had been less than or equal to 9, y would have received the value 200.

The same code, using the if-else statement is written as: -

```
x =10;
if ( x > 9)
{
        y = 100;
}
else
{
        y = 200;
}
```

Size of Compile-time Operator

Size of is a unary compile-time operator that returns the length of the operand, in bytes. The operand can be a variable, a constant or a data type.

Example:

```
#include <iostream.h>
main()
{
        char ch;
        cout<<sizeof ch;
        cout<<sizeof(int);
}
```

The output is:

```
        12
```

Note, to compute the size of a type, you must enclose the type name in parentheses. This is not necessary for variable names. Size of operator is normally used to determine the lengths of arrays and structures when their sizes are not known to the programmer. It is also used to allocate dynamic memory to variables during execution of the program.

Comma Operator

The comma operator is used to string together several expressions. Essentially, the comma causes a sequence of operations to be performed. When it is used on the right side of the assignment statement, the last expression of the comma-separated list, is the value assigned to the expression.

Example:

```
y= 10;
x = (y- = 5, 25 / y);
```

After execution, x will have the value 5 because y's original value is reduced by 5, and then that value is divided into 25, yielding 5 as the result.

Precedence of Operators

Table lists the precedence of all C++ operators.

Symbol	Description	Associatively
++	Post Increment	Left to Right
--	Post Decrement	Left to Right
( )	Function Call	Left to Right
[ ]	Array Element Reference	Left to Right
□□	Pointer to Structure Member	Left to Right
	Structure or Union Member	Left to Right
□□	Pre Increment	Right to Left
□□	Pre Decrement	Right to Left
□	Logical NOT	Right to Left
□	Bitwise NOT	Right to Left
□	Unary Plus	Right to Left
□	Unary Minus	Right to Left
□	Indirection (Pointer Reference)	Right to Left
□	Address	Right to Left
Sizeof()	Size of an Object	Right to Left
->*	Pointer to member	Left to Right
(type)	Cast Conversion	Right to Left
*	Multiplication	Left to Right
/	Division	Left to Right
%	Modulus Operator	Left to Right

+	Addition	Left to Right
-	Subtraction	Left to Right
<<	Left Shift	Left to Right
>>	Right Shift	Left to Right
<	Less Than	Left to Right
<=	Less Than equal To	Left to Right
>	Greater Than	Left to Right
>=	Greater Than equal to	Left to Right
==	Equality	Left to Right
!=	Inequality	Left to Right
&	Bitwise AND	Left to Right
^	Bitwise Exclusive OR	Left to Right
|	Bitwise OR	Left to Right
&&	Logical AND	Left to Right
| |	Logical OR	Left to Right
?	Conditional expression	Left to Right
,	Comma operator	Left to Right
=,*=,/=,% =.+=,- =,&=,^=,| =, <<=.>>=	Assignment operators	Right to Left

4.6 References

C++ contains a feature that is related to the pointer. This feature is called a reference. A reference is essentially an implicit pointer that acts as another name for an object.

Reference Parameters

One importance use for a reference is to allow you to create functions that automatically use call-by-reference parameter passing rather than c++'s default call-by-value.

As you know, in C, to create a call-by-reference you must explicitly pass the address of an argument to the function. For example, consider the following short program, which uses this approach in a function called neg(), which reverses the sign of the integer variable pointed to by its argument.

```
#include<iostream.h>
void neg(int *i);
main()
{
int x;
x=10;
cout << x<<"negated is ";
neg(&x);
cout <<x<<"\n";
 return 0;
}
void neg(int *i)
{
 i = -i;
}
```

In this program,neg() takes as a parameter a pointer to the integer whose sign it will reverse. Therefore, neg() is explicitly called with the address of x. Further, inside neg() the * operator must be used to access the variable pointed to by i. As you can automate this feature by using a reference parameter.

To create a reference parameter, precede the parameter's name with an &. Here is how neg() is declared using a reference:

```
Void neg(int &i);
```

This tells the compiler to make i into a reference parameter. Once this has been done, i essentially becomes another name for whatever argument neg() is called with. That is, i is an implicit pointer that automatically refers to the argument used in the call to neg(). Once i has been made into a reference, it is no longer necessary (or even legal) to apply the * operator. Instead, each time i is used, it is implicitly a reference to the argument's name with the & operator. Instead, the compiler does this automatically. Here is the reference version of the preceding program:

```
#include<iostream.h>
void neg(int &i);
main()
{
 int x;
 x=10;
 cout << x<< “negated is “;
 neg(x);
 cout << x<<”\n”;
 return 0;
}
void neg(int &i)
{
 i= -i;
}
```

to review: when you create a reference parameter, that parameter automatically refers to (implicitly points to) the argument used to call the function. Therefore, the statement

```
i=-1;
```

actually operates on x, not on a copy of x. There is no need to apply the & operator to an argument. Also, inside the function, the reference parameter is used directly without the need to apply the * operator.

It is important to understand that when you assign a value to a reference, you are actually assigning that value to the variable used in the call to the function.
Inside the function, it is not possible to change what the reference parameter is “pointing” to. That is, a statement like

```
i++;
```

Inside neg() increments the value of the variable used in the call. It does not cause i to point to some new location.

Here is another example. This program uses reference parameters to swap the values of the variables it is called with.

```
#include<iostream.h>
void swap(int &i,int &j);
main()
{
 int a,b,c,d;
a=1;
a=2;
a=3;
d=4;
cout << “a and b “ << a << “ “ << b << “\n”;
swap(a,b);
cout <<”a and b “ <<a << “ “  << b<<”\n”;
cout << “c and d” << c<< “ “ << d << “\n;
swap(c,d);
cout << “c and d:” << c<< “ “ << d << “\n”;
return 0;
}
void swap(int &i,int &j)
{
 int t;
 t=i;
 i=j;
 j=t;
}
```

this program displays the following:

```
a and b: 1 2
a and b: 2 1
c and d: 3 4
c and d: 4 3
```

Passing References to objects

When you pass by reference, no copy of the object is made. This means that no object used as a parameter is destroyed when the function terminates, and the parameter’s destructor is not called. For example, try this program:

```
#include<iostream.h>
class c1
{
 int id;
public:
int i;
c1(int i);
~c1();
void neg(c1 &o){0.i=-0.i;}
};
c1::c1(int num)
{
 cout <<”Constructing”<<num<<”\n”;
id=num;
}
c1::~c1()
{
 cout <<"Destructing "<<id<<"\n";
}
main()
{
 c1 o(1);
 o.i=10;
o.neg(o);
cout <<o.i<<"\n";
return 0;
}
```

here is the output of the program:

```
Constructing 1
-10
destructing 1
```

as you can see, only one call is made to c1's destructor function. Had o been passed by value, a second object would have been created inside neg(), and the destructor would have been called a second time when that object was destroyed at the time neg() terminated.

When passing parameters by reference, remember that changes to the object inside the function affect the calling object.

Returning References

A function may return a reference. This has the rather starting effect of allowing a function to be used on the left side of an assignment statement ! for example, consider this simple program:

```
#include<iostream.h>
char & replace(int i);
char s[80]="Hello There";
main()
{
 replace(5)='X';
cout <<s;
return 0;
}
char &replace(int i)
{
 return s[i];
}
```

this program replaces the space between hello and there with an X. That is, the program displays helloxthere. Take a look at how this is accomplished.

As shown, replace() is declared as returning a reference to a character array. As replace () is coded, it returns a reference to the element of s that is specified by its argument i. The reference returned by replace () is then used in main() to assign to that element the character X.

Restrictions to References

There are a number of restrictions that apply to references. You cannot reference another reference. Put differently, you cannot create a pointer to a reference. You cannot reference a bit-field.

A reference variable must be initialized when it is declared unless it is a member of a class, a function parameter, or a return value. Null references are prohibited.

4.7 Enum types

Enumerated types work when you know in advance a finite (usually short) list of values that a data type can take on. It is another user defined type which provides a way for attaching names to numbers, thereby increasing comprehensibility of the code. The **enum** keyword automatically enumerates a list of words by assigning them values 0,1,2,3,4, and so on.

The general form of enum is:

```
        enum variable name { list of constants separated by commas };
    where enum is a keyword
    variable name is the user defined variable name
    list indicates the fixed constant values
    eg.,
        enum days_of_week {sun, mon, tue, wed, thu, fri, sat};
```

Once we have specified the days of the week as shown we can define variable of this type.

Example:

```
// demonstration of enum data types
```

```
# include<iostream.h>
enum days_of_week {sun, mon, tue, wed, thu, fri, sat };
void main( )
{
        days_of_week day1, day2,day3;
        day1 = mon;
        day 2 = fri;
        int diff = day2 - day1;
        cout<<"days between = "<<diff<<endl;
        if (day1<day2)
                cout<< "day1 comes before day2\n";
}
```

the values listed inside braces of enum keyword are called members. Enumerated means that all the values are listed.

4.8 Anonymous Union

It is well known that a structure is a heterogeneous data type which allow to pack together different types of data values as a single unit. Union is also similar to a structure data type with a difference in the way the data is stored and retrieved. The union stores values of different types in a single location. A union will contain one of many different types of values.

The general form of union is

```
        union user_defined_name{
                member1;
                member 2;
                _________
                __________
                member n;
        };
```

the keyword union is used to declare the union data type. This is followed by a **user_defined_name** surrounded by braces which describes the member of the union.

Example:

```
        union sample{
                int x;
                char s;
                float t;
        } one, two;
```

where one and two are the union variables similar to data size of the sample.

It is possible to define a union data type without a user_defined_name or tag and this type of declaration is called an anonymous union.
The general form is

```
        union{
                member1;
                member2;
                __________;
                __________;
                member n;
};
```

The keyword union is used to declare the union data type. This is followed by braces which describes the member of the union.

```
        Eg.
                union{
                        int x;
                        float a;
                        char ch;
                };
```

Sample program:

```
        #include <iostream.h>
        void main(void)
        {
                union
                {
                        int x;
                        float y;
                        cout<<"enter the following information"<<endl;
                        cout<<"x(in integer)"<<endl;
                        cin>>x;
                        cout<<"y(in floating)"<<endl;
                        cin>>y;
                        cout<<"\n content of union"<<endl;
                        cout<<"x="<<x<<endl;
                        cout<<"y="<<y<<endl;
                }
```

the output is

```
                enter the following information
```

```
x(in integer)
10
y(in floating)
34.90
content of union
x=10
y=34.90
```

4.9 Short Summary

1. Variables are the quantities whose value may change while execution of program
2. Constant are the quantities whose value remains the same while processing
3. Operators are used to perform some mathematical calculations.
4. Operators can also be used to give true or false value as a result of its execution.
5. Enum data type is used to denote some fixed values.

4.10 Brain Storm

1. What do identifiers mean? How is a user defined identifier different form a standard reserved identifier?
2. What are the keywords or standard identifiers used in C++?
3. What are the different types of constants?
4. What is an operator?
5. List out the various types of operators used in C++.
6. List out the five arithmetic operators. What is the associatively rule involved in these operators?
7. What are meant by comparison and logical operators?
8. What is a unary operator?
9. Explain the syntax of Enum type.
10. What is the purpose of anonymous union?

Lecture - 5

Functions, Arguments & Overloading

Objectives

In this lecture you will learn the following

- Knowing types of C++ functions
- Shows overloading types

Coverage Plan

Lecture - 5

5.1 Snap Shot

In this lecture you will be introduced about Functions, its Prototypes, its Declaration, Function Overloading and about Operator Overloading.

5.2 C++ Functions and its Prototypes

A function groups a number of program statements into a unit and gives it a name. This unit can then be invoked from other parts of the program. It is used to reduce program size. The function's code is stored in only one place in memory, even though the function is executed many times in the course of program.

The main advantages of using a function are:

1. easy to write a correct small function
2. easy to read, write and debug a function
3. easier to maintain or modify such a function
4. small functions tend to be itself documenting and highly readable
5. it can be called any number of times in any place with different parameters

5.3 The function declaration

The general form of function declaration is

datatype function-name (datatype argument1, datatype aregument2 …datatype argument);

Here, datatype tells the function return data type. Function name is user defined function name. It can be given using the rules for naming variable. Inside parentheses argument or parameters are declared with its data type. Any number of arguments can be given.

Example:

```
void starline( );
```

This declaration tells the compiler that at some later point we plan to present a function called starline. The key word void specifies that the function has no return value, and the empty parentheses indicate that it takes no arguments. The function declaration is terminated with semicolon.

Function declarations are also called prototypes, since they provide a model or blueprint for the function. They tell the compiler, "a function that looks like this is coming up later in the program.

Function calling

The function is called or invoked as many time as required. To call the function we need to give function name, followed by parentheses. The syntax of the call is very similar to that of the declaration, except that the return type is not used. A semicolon terminates the call. Executing function statement causes the function to execute; the statements in the function definition are executed and then control returns to the statement following the function call.

Example: starline ( );

The Function Definition

The function definition contains the actual code for the function.

Example: function definition part of void starline( );

```
void starline( )                        //function declarator
{
        for (int j = 0 ; j<=45;j++)     //function body
                cout<<"*";
                cout<<endl;
}
```

The function body composed of the statements that make up the function, delimited by braces. The function declaration must use the same function name, have the same argument types in the same order, and have the same return type. The function declaration should not terminated with semicolon.

When the function is called the control is transferred to the first statement in the function body. The other statements in the function body are then executed, and when closing brace encountered, control returns to the calling program.

Sample Program:

```
// source code describes the function declaration, function calling
   and   function        //definition
// store the file as funct1.cpp
# include<iostream.h>
void main( )
{
        void display ( );               // function declaration
        display( );                     // function calling
}
void display ( )                        // function definition
{
        cout<< " this is a test program \n";
        cout<<" for demonstrating a function call \n";
        cout<< " in C++ \n";
    }
```

output of the above program

```
this is a test program
for demonstrating a function call
in C++
```

The main function invokes the display ( ) function. In C++ each function is almost a separate program. The control will be transferred from the main function or from a function to a called function. After it has executed all the statements in the function, the control switches back to the calling portion of the program.

Return statement

The keyword return is used to terminate function and return a value to its caller. The return statement may also be used to exit a function without returning a value. The return statement may or may not include an expression.

The general form of the return statement is.

```
return;
return ( expression);
```

The return can appear anywhere within the function body. A function can also have more than one return although it is good programming style for a function to have a single entrance and single exit.

Example:

```
return;
return (45);
return ( x + y);
return (++j);
return 0;
return i++ // correct but not good style
```

The return statement terminates the execution of the function and pass on the control back to the calling environment.

Example:

1. int sum (int a, int b)

```
{
        return(a+b); // return integer value.
}
float maximum (float a, float b)
{
        if (a>b)
                return a;
        else
                return b;
}// return floating point value.
```

Sample program:

```
//using multiple return statements in a function
// source program name: funct2.cpp
# include<iostream.h>
void main (void)
{
        float maximum( float, float, float);
        float x,y,z,max;
        cout<<" enter three numbers \n";
        cin>> x >> y >> z;
        max = maximum(x,y,z);
        cout << "maximum " << max;
}
float maximum (float a, float b, float c)
{
        if ( a>b)
        {
                if (a>c)
                        return (a);
                else
                        return  (c);
        }
        else
        {
                if ( b>c)
                        return ( b);
                else
                        return ( c);
        }
}// end of function
```

output of the above program is

```
enter three numbers
        4 5 6
maximum = 6
```

5.4 Types of functions

The user defined functions may be classified into three ways.

A function is invoked without passing any formal argument does not return any value to the calling portion.

A function is invoked with formal argument and does not return any value to the calling portion.

A function is invoked with formal argument and returns back a value to the calling environment

Type 1: There is no data communication between the calling portion of a program and a called function block. A calling environment invokes the function by not passing any formal argument and the function does not return back any value to the caller.

Example:

```
# include<iostream.h>
void main( void)
{
        void message ( void); // function declaration
        message( );           // function calling
}
void message (void)           // function definition
{
        __________
        _________
}
```

Type 2: The second type of user defined function passes some formal arguments to a function but the function does not return back any value to the caller. It is an one-way data communication between a calling portion of the program and the function block.

Example:

```
# include<iostream.h>
void main(void)
{
        void power(int, int);   //function declaration
        int x,y;
        ____________
        ____________
        power(x,y);
}
void power(int x, int y)                //function calling
{
        ___________
        ___________

// body of the function
// no value will be transferred back to the caller
}
```

Sample program:

```
// Passing formal arguments and no return statement
// source name: funct3.cpp
# include<iostream.h>
void main(void)
{
        void square ( int );     //function declaration
        int max;
        cout<< "enter a value for n \n";
        for ( int i = 0; i <= max; i++)
                square( i );
}
void square (int n)
{
        float value;
        value = n * n;
        cout<< "i = "<< n <<"square = "<< value<< endl;
}
```

output of the above program

```
enter value for n
        2
        i= 0 square = 0
        i= 1 square = 1
```

```
i= 2 square = 4
```

Type 3: The third type of user defined function passes some formal arguments to a calling portion of the program and the computed value, if any, is transferred back to the caller. Data communication between both the calling portion of a program and the function block.

Example:

```
void main (void)
{
        int output(int, int, char);        //function declaration
        omt x, y, temp;
        ______________
        ______________
        temp = output( x, y, ch);          // function calling
}
int output( int a, int b, char s)          //function definition
{
        int value;
        _________________
        _________________
        //body of the function
        return (something);
}
```

Sample program:

```
//program to find the factorial of a given number
// source file funct4.cpp
# include<iostream.h>
void main(void)
{
        long int fact(int);
        int x, n;
        cout<<"enter any integer number"<< endl;
        cin>>n;
        x = fact(n);
        cout<<" value = " <<n << "and its factorial =";
        cout << x << endl;
}
long int fact(int n)
{
        int value = 1;
        if (n  = = 1)
                return (value);
        else
```

```
		{
			for(int i= 1; i<= n; i++)
				value = value * i
				return(value);
		}
	}
output of the above program
	enter any integer number
		5
	value 5 and its factorial = 120
```

5.5 Actual and formal arguments

The arguments can be classified into two groups:

Actual argument
Formal argument

Actual argument: An actual argument is a variable or an expression contained in a function call that replaces the formal parameter which is a part of the function.

Example:

```
# include <iostream.h>
void main( )
{
	int x, y;
	void output(int x, int y);	//function	declaration
	______________
	______________
	output(x,y);		//x and y are the actual argument
}
```

Formal argument: Formal arguments are the parameters present in a function definition which may also be called as dummy arguments or the parametric variables. When the function is invoked, the formal parameters are replaced by the actual parameters.

Example:

```
# include<iostream.h>
void main(void)
{
	int x,y;
	void output(int x, int y);
	_______________
	________________
	output(x,y);
}
```

```
void output(int a, int b) // a and b are the formal or dummy
arguments
{
        // body of the function definition
}
```

Formal argument may be declared by the same name or by different names in calling a portion of the program or in a called function but the data types should be the same in both the blocks.

Local and Global variable

The variables in general may be classified as local and global variables.

Local variable:

Variables declared inside a particular block or function is known as local variables.

Example:

```
void funct (int i, int j)
{
        int k,n; //local variable
        ____________
        ____________
        // body of the function
}
```

The integer variables k and n are defined within a function block of the funct( ). All the variables to be used within a function block must be either defined at the beginning of the block itself or before using in a statement. Local variables are referred only the particular part of a block or a function. Same variable name may be given to different parts of a function or a block and each variable will be treated as a different entity.

Example:

```
funct1 (float a, float b)
{
        float x,y;
        x = 23.00;
        y = 10.9;
        ________
        ________
}
funct2 (int i, int j)
{
        float x,y;
        x = 56.00;
        y = 1.90;
        _________
        _________
}
```

Global variable:

Global variables are variables defined outside the main function block. These variables are referred by the same data type and by the same name through out the program in both the calling portion of a program and in the function block. Whenever some of the variables are treated as constants in both the main and the function block, it is advisable to use global variable.

Example:

```
int  x,y = 4;
void main(void)
{
        void funct1( );
        x = 10;
        ________
        funct1 ( );
}
 void funct1 ( )
{
        int sum;
        sum = x +y;
}
```

Sample program

```
//global and local variable declaration
# include<iostream.h>
int x;                              //global variable
int y = 5;                          //global variable
void main (void)
{
        x = 10;
        void sum (void);
        sum ( );
}
void sum ( )
{
        int sum;                              //local variable
        sum = x + y;
        cout<< "x = " <<x<<endl;
        cout<<"y="<<y<<endl;
```

```
            cout<<"sum = "<<sum << endl;
        }
```

output of the above program

```
x = 10
y = 5
sum = 15
```

Recursive function

A function which calls itself directly or indirectly again and again is known as the recursive function. Recursive functions are very useful while constructing the data structures like linked lists, double linked lists and trees. There is a distinct difference between normal and recursive functions. A normal function will be invoked by the main function whenever the function name is used. The recursive function will be invoked by itself directly or indirectly as long as the given condition is satisfied.
For example:

```
    # include<iostream.h>
    {
        void main(void)
        {
            void f1 ( ); // function declaration
            _______________
            _______________
            f1 ( ); // function calling
    }
        void f1( )  // function definition
        {
            _________________
            _________________
            f1 ( ); //function calls recursively
}
```

Sample program

```
# include<iostream.h>
void main( void)
{
    int sum(int);
    int n, temp;
    cout<<"Enter any integer number"<< endl;
    cin>>n;
```

```
        temp = sum(n);
        cout<<"value = "<<n<<"and its sum ="<< temp;
}
        int sum ( int n)
        {
                int sum (int); // local function declaration
                int value = 0;
                if (n= = 0)
                        return(value);
                else
                        value = n+sum(n-1);
                        return (value);
        }
```

out put of the above program is

```
        Enter an integer number
        4
        value = 4 and its sum = 10
```

5.6 Default arguments

One of the useful facilities available in C++ is the facility to define default argument values for the functions. In the function prototype declaration, the default value are given. Whenever a call is made to a function without specifying an argument, the program will automatically assign values to the parameters from the default function prototype declaration. Default arguments facilitate easy development and maintenance of programs.

Sample program:

```
// default argument declaration
# include <iostream.h>
void sum (int a, int b, int c = 6, int d = 10);    //default argument
                                                      initialization
void main( void)
{
        int a,b,c,d;
        cout<<"enter any two numbers\n";
        cin>>a>>b;
        sum(a,b);        //sum of default values
}
```

```
void sum(int a1, int a2, int a3, int a4)
{
        int temp;
        temp = a1+a2+a3+a4;
        cout<< "sum="<< temp;
}
```

output of the above program

```
enter any two numbers
        10 20
        sum = 46
```

5.7 Function Overloading

Overloading refers to the use of the same thing for different purposes. C++ also permits overloading of functions. This means that we can use the same function name to create functions that perform a variety of different argument tasks. This is known as function polymorphism in OOP.

Using the concept of function overloading, we can design a family of functions with one function name but with different argument lists. The function would perform different operations depending on the argument list in the function call. The correct function to be invoked is determined by checking the number and type of the arguments but on the function type. For example, an overloaded add( ) function handles different types of data as shown below :

```
// Declarations

int add(int a, int b);
int add(int a,int b, int c);
double add(double x,double y);
double add(int p, double q);
double add(double p,int q);

// Function calls

cout << add(5,10);
cout << add(15,10.0);
cout << add(12.5,7.5);
cout << add(5.10,15);
cout << add(0.75,5);
```

A function call first matches the prototype having the same number and type of arguments and then calls the appropriate function for execution. A best match must be unique. The function selection involves the following steps:

1. The compiler first tries to find an exact match in which the types of actual arguments are the same and use that function.

2. If an exact match is not found, the compiler uses the integral promotions to the actual arguments, such as,

   Char to int
   Float to double
   to find a match.

3. when either of them fails, the compiler tries to use the built-in conversions (the implicit assignment conversions) to the actual arguments and then uses the function whose match is unique. If the conversion is possible to have multiple matches, then the compiler will generate an error message. Suppose we use the following two functions :

   long square(long n);
   double square(double x)
   a function call such as square(10) will cause an error because int argument can be converted to either long or double, thereby creating an ambiguous situation as to which version of square () should be used.

4. if all of the steps fill, then the compiler will try the user-defined conversions in combination with integral promotions and built-in conversions to find a unique match. User-defined conversions are often used in handling class objects.

The following program illustrates function overloading

```
#include<iostream.h>

int volume(int);
double volume(double,int);
long volume(long,int,int);

main()
{
 cout <<  volume(10) << “\n”;
 cout << volume(2.5,8) << endl;
 cout << volume(100L,75,15);
}
int volume(int s)
{
        return (s*s*s);
}
double volume(double r,int h)
{
        return(3.14519*r*r*h);
}
long volume(long l,int b,int h)
{
        return (1*b*h);
}
```

The output of program would be

```
1000
157.2595
112500
```

overloading of the functions should be done with caution. We should not overload unrelated functions and should reserve function overloading for functions that perform closely related tasks. Sometimes, the default arguments may be used instead of overloading. This may reduce the number of functions to be defined.

Overloaded functions are extensively used for handling class objects.

5.8 Operator Overloading

Operator overloading is one of the many exciting features of C++ language. It is an important technique that has enhanced the power of extensibility of C++. We have stated more than once that C++ tries to make the user-defined data types behave in much the same way as the built-in types. For instance, C++ permits us to add two variables of user-defined types with the same syntax that is applied to the basic type. The mechanism of giving such special meaning to an operator is known as operator overloading.

Operator overloading provides a flexible option for the creation of new definitions for most of the C++ operators. We can almost create a new language of our own by the creative use of the function and operator overloading techniques. We can overload all the C++ operators except the following:

1. class member access operators (.,.*)
2. scope resolution operator (::)
3. size of operator (sizeof)
4. conditional operator (?:)

the excluded operators are very few when compared to the large number of operators, which qualify for the operator overloading definition.

Although the semantics of an operator can be extended, we cannot change its syntax, the grammatical rules that govern its use such as the number of operands, precedence and associatively.

For example, the multiplication operator will enjoy higher precedence than the addition operator. Remember, when an operator is overloaded, its original meaning is not lost. For instance, the operator +, which has been overloaded to add two vectors, can still be used to add two integers.

Defining Operator Overloading

To define an additional task to an operator, we must specify what it means in relation to the class to which the operator is applied. This is done with the help of a special function, called operator function, which describes the task. The general form of an operator function is:

Where return type is the type of value returned by the specified operation and op is the operator being overloaded. The op is preceded by the keyword operator . Operator op is the function name.

Operator functions must be either member functions or friend functions. A basic difference between them is that a friend function will have only one argument for unary operators and two for binary operators. This is because the object used to invoke the member function is passed implicitly and therefore is available for the member function. This is not the case with friend functions. Arguments may be passed either by value or by reference.

Operator functions are declared in the class using prototypes as follows:

```
Vector operator+(vector);
Vector operator –();
```

```
Friend vector operator+(vector,vector);
Friend vector operator-(vector);
Vector operator-(vector &a);
Int operator= =(vector);
Friend int operator = =(vector,vector)
```

Vector is a data type of class and may represent both magnitude and direction or a series of points called elements

The process of overloading involves the following steps:

1. First , create a class that defines the data types that is to be used in the overloading operation.

2. Declare the operator function operator op() in the public part of the class. It may be either a member function or a friend function.

3. Define the operator function to implement the required operations.

Overloaded operator functions can be invoked by expressions such as

Op x or x op

For unary operators and

x op y

for binary operators.

Op x(or x op) would be interpreted as

Operator op(x)

For friend functions, similarly, the expression x op y would be interpreted as either

x.operatro op(y)

in case of member functions, or

operator op(x,y)

in case of friend functions. When both the forms are declared, standard argument matching is applied to resolve any ambiguity.

Index class with operator overloading

```
#include<iostream.h>
class index
{
 private:
          int value;
public:
          index()
          {
          value=0;
          }
          int getindex()
          {
           return value;
          }
          void operator ++()
          {
           value=value+1;
          }
};
void main()
{
  index idx1,idx2;
cout <<"\n Index1 = "<< idex1.getindex();
cout <<"\n Index2 = "<< idx2.getindex();
++idx1;
idx2++;
idx2++;
cout <<"\nIndex1="<<idx1.getindex();
cout <<"\nIndex2="<<idx2.getindex();
}
```

Run

```
Index1=0
Index2=0
Index1=1
Index2=2
```

In main(), the statements

```
++idx1;
idx2++;
```

Invoke the overloaded function is ++. The word operator is a keyword and is preceded by the return type void. The operator to be overloaded is written immediately after the keyword operator. This declaration informs the compiler to invoke the overloaded function ++ whenever the unary increment operators prefixed or post fixed to an object of the index class.

The variables idx1 and idx2 are he objects of the class index. The index value is advanced by using statements such as ++idx1, idx++ .

Limitations of increment and decrement operators

The prefix notation causes a variable to be updated before its values is used in the expression, whereas the postfix notation causes it to be updated after its value is used. However, the statement

```
Idx1=++idx2;
```

Has exactly the same effect as the statement

```
Idx1=idx2++;
```

When ++ and – operators are overloaded, there is no distinction between the prefix and postfix overloaded operator function. This problem is circumvented in advanced implementations of C++, which provides additional syntax to express and distinguish between prefix and postfix overloaded operator functions. A new syntax to indicate postfix operator overloaded function is:

```
Operator ++(int)
```

The program index5.cpp illustrates the invocation of prefix and postfix operator functions.

```
#include<iostream.h>
class index
{
 private:
        int value;
public:
        index()
        {
        value=0;
        }
```

```
            int getindex()
            {
             return value;
            }
            index operator ++()
            {
              return index (++value);
            }
            index operator ++(int)
            {
             return index(value++);
            }
    };
    void main()
    {
      index idx1(2),idx2(2),idx3,idx4;
    cout <<"\n Index1 = "<< idx1.getindex();
    cout <<"\n Index2 = "<< idx2.getindex();
    idx3=idx1++;
    idx4=++idx2;
    idx2++;
    cout <<"\nIndex1="<<idx1.getindex();
    cout <<"\nIndex2="<<idx2.getindex();
    cout <<"\nIndex3="<<idx3.getindex();
    cout <<"\nIndex4="<<idx4.getindex();
    }
```

Run

```
Index1=2
Index2=2
Index1=3
Index2=2
Index1=3
Index1=3
```

In the postfix operator ++(int) function, first a nameless object with the old index value is created and then, the index value is updated to achieve the intended operation. The compiler will just make a call to this function for postfix operation, but the responsibility of achieving this test on the programmer.

Binary operator overloading

The syntax of binary operator overloading is

```
        Function returntype : primitive,void, or userdefined

Returntype operator operator symbol (arg)
{
        body of operator function
}
```

the binary overloaded operator function takes the first as an implicit operand and the second operand must be passed explicitly. This data members of the first object are accessed without using the dot operator whereas, the second argument members can be accessed using the dot operator if the argument is an object. Otherwise it can be accessed directly. Note that, the overloaded binary operator function is a member function defined in the first object's class.

5.9 Short Summary

- The function is called or invoked as many time as required. To call the function we need to give function name , followed by parentheses.

- The keyword return is used to terminate function and return a value to its caller.

- Variables declared inside a particular block or function are known as local variables.

- Function Overloading is nothing but same function name that perform a variety of different argument tasks.

- The process of overloading involves the following steps:

- First , create a class that defines the data types that is to be used in the overloading operation.

- Declare the operator function operator op() in the public part of the class. It may be either a member function or a friend function.

- Define the operator function to implement the required operations.

5.10 Brain Storm

1. Explain function and its type
2. What is the purpose of return statement in a function?
3. What is meant by the function arguments, function call and return values?
4. List out the rules normally governing the use of return statement.
5. What is a recursive function?
6. List out the merits and demerits of the functions?
7. How is a recursive function different from an ordinary function?

Lecture - 6

Classes & Objects

Objectives

In this lecture you will learn the following

- About Classes and Objects
- Access Specifiers
- Member Functions
- The 'this' keyword
- Static and non-static member function

Lecture - 6

6.1 Snap Shot

In this Lecture you will learn about Classes, Objects, Various Member Function, Friend Function and also you will know the use of the 'this' keyword.

6.2 Classes

In C++, the class forms the basis of object-oriented programming. Specifically, it is the class that is used to define the nature of an object. In fact, the class is C++'s basic unit of encapsulation. In this lecture, classes and objects are examined in detail.

Classes are created using the keyword class. A class declaration defines a new type that links code and data. This new type is then used to declare objects of that class. Thus, a class is a logical abstraction, but an object has physical existence. In other words, an object is an instance of a class.

A class declaration is similar syntactically to a structure. Here, is the entire general form of a class declaration that does not inherit any other class.

```
Class class-name
      {
            private data and functions
      access-specifier:
             data and functions
      access-specifier:
            data and functions
            .
            .
            .
      access-specifier:
            data and functions
      }object-list;
```

The object-list is optional. If present, it declares objects of the class.

6.3 Access specifier

Access-specifier is one of these three c++ keywords:

1. Public

2. Private
3. Protected

By default, functions and data declared within a class are private to that class and may be accessed only by other members of the class. However, by using public access specifier, you allow functions or data to be accessible to other parts of your program. Once an access specifier has been used, it remains in effect until either another access specifier is encountered or the end of the class declartion is reached. To switch back to private declarations, you can use the private access specifier. The protected access specifier is needed only when inheritance is involved.

Generally, a class specification has two parts:

A class declaration, which describes the data component in terms of data members, and the public interface, in terms of member functions

The class method definitions, which describes how certain class member functions are implemented

Roughly speaking, the class declaration provides a class overview, whereas the method definitions supply the details.

The class specifies the type and scope of its member. The keyword class indicates that the name, which follows (ClassName), is an abstract data type. The body of a class is enclosed within the curly braces followed by a semicolon – the end of a class specification. The body of a class contains declaration of variables and functions, collectively known as members. The variables declared inside a class grouped under two sections, private and public, which define the visibility of members.

The private members are accessible only to their own class's members. On the other hand, public members are not only accessible to their own members, but also from outside the class. The members in the beginning of class without any access specifier are private by default. Hence, the first use of the keyword private in a class is optional. A class, which is totally private, is hidden from the external world and will not serve any useful purpose.

The following declaration illustrates the specification of a class called student having roll_no and name as its data members:

```
Class student

{
```

```
int roll_no;
char name[20];
public :
void setdata(int roll_no_in,char *name_in)
{
        roll_no=roll_no_in;
        strcpy(name,name_in);
}
void outdata()
{
        cout << "Roll No = " << roll_no<<endl;
        cout << "Name = " << name << endl;
}
};
```

A class should be given some meaningful name, (for instance, student) reflecting the information it holds. The class name student becomes a new data type identifier, which satisfies the properties of abstraction; it can be used to define instances of class data type. The class student contains two data members and two member functions. The data members are private by default while both the member functions are specified as **public**. The member function setdata() can be used to assign values to the data members roll_no and name. The member function outdata() can be used for displaying the value of data members. The data members of the class student cannot be accessed by any other function except member functions of the student class. It is a general practice to declare data member as private and member function as public.

6.4 Class Objects

A class specification only declares the structure of objects and it must be instantiated in order to make use of the services provided by it. This process of creating objects (variables) of the class is called class instantiation. It is the definition of an object that actually creates objects in the program by setting aside memory space for its storage. Hence, a class is like a blueprint of a house and it indicates how the data and functions are used when the class is instantiated. The necessary resources are allocated only when a class is instantiated. The syntax for defining objects of a class is class classname objectname,...; . Note that the keyword class is optional.

Accessing a Class Members

Once an object of a class has been created, there must be a provision to access its members. This is achieved by using the member access operator, dot(.). the syntax for accessing members (data and functions) of a class is shown below

```
Objectname . datamember

Objectname . functionname(actual arguments)
```

If a member to be accessed is a function, then a pair of parentheses is to be added following the function name. The following statements access member functions of the object s1, which is an instance of the student class:

```
S1.setdata(10,”Rajkumar”);
S1.outdata();
```

The program student.cpp illustrates the declaration of the class student with the operations on its objects.

```
// student.cpp: member functions defined inside the body of the student class
#include <iostream.h>
#include<string.h>
class student
{
        private:
                int roll_no;            // roll number
                char name[20];// name of a student
        public :
                // initializing data members

                void setdata(int roll_no_in,char *name_in)
                {
                        roll_no = roll_no_in;
                        strcpy(name,name_in);
                }

                // display data members on the console screen

                void outdata()
                {
                        cout << “Roll No = “ << roll_no <<endl;
                        cout << “Name =” << name << endl;
                }
};
void main()
{
  student s1;                   // first object / variable of class student
  student s2;                   // second object / variable of class student
  s1.setdata(1,"Tejaswi");      // object s1 calls member setdata()
  s2.setdata(10,"Rajkumar");    // object s2 calls member setdata()
  cout << "Student details ..." << endl;
  s1.outdata();                 // object s1 calls member function outdata()
  s2.outdata();                 // object s2 calls member function outdata()
}
```

```
Run
Student details ...
Roll No = 1
Name = Tejaswi
Roll No = 10
Name = Rajkumar
```

In conventional programming languages, a function is invoked on a piece of data (function-driven communication), whereas in an OOPL (object-oriented programming language), a message is sent to an object (message –driven programming) i.e., conventional programming is based on function abstraction whereas, object oriented programming is based on data abstraction.

The object accessing its class members resembles a client-server model. A client seeks a service whereas; a server provides services requested by a client. In the above example, the class student resembles a server whereas, the objects of the class stduent resembles clients. They make calls to the server by sending messages. In the statements

```
S2.setdata(10,"Rajkumar");          // object s2 calls member function setdata
```

The object s2 sends the message setdata to the server with the parameters 10 and Rajkumar. As a server, the member function setdata() of the class student performs the operation of setting the data members according to the messages sent to it. Similarly, the statements

```
        S2.outdata()
```

Can be visualized as sending message (outdata) to objects s2's class to display object contents. Thus, by its very nature, OO computation resembles a client-server-computing model.

6.5 Inline Member Functions

The keyword inline is used as a function specifier only in function declarations. The inline specifier is a hint to the compiler that inline substitution of the function body is to be preferred to the usual function call implementation.

The advantage of using inline member functions are:

1. The size of the object code is considerably reduced.
2. It increases the execution speed
3. The inline member functions are compact functions calls.

The general form is:

```
class user_name
{
        //data variables;
        public:
                inline return_type function name (parameters);
                //other member functions;
}
```

Whenever a function is declared with inline specifier, the C++compiler merely replaces it with the code itself so the overhead of the transfer of control between the calling portion of a program and a function is reduced.

```
// demonstration of inline function
# include <iostream.h>
class sample
{
        int x;
        public:
                inline void getdata( );
                inline void display ( );
}
inline void sample :: getdata ( )
{
        cout<<" enter a number"<<endl;
        cin>>x;
}
inline void sample:: display ( )
{
        cout<<"entered number is = "<<x<<endl;
}
void main( )
{
        sample ob;
        ob.getdata ( );
        ob.display ( );
}
```

6.6 Friend functions

The main concept of the object oriented programming are data hiding and data encapsulation. Whenever data variables are declared in a private category of a class these members are restricted from accessing by non-member functions. To access a private data member by a non-member function is to change a private data member to a public group. When the private or protected data member is changed to a public category, it violates the whole concept of data hiding and data encapsulation. To solve this problem, a friend function can be declared to have access to these data members. Friend is a special mechanism for letting non-member functions access private data. A friend function may be declared or defined within the scope of a class definition. The keyword friend inform the compiler that it is not a member function of the class.

The general form is

```
        friend  return_type user_function_name(parameters);
                where friend is a keyword used as function modifier.
```

Example:

```
        class alpha
        {
                private:
                        int x;
                public:
                        void getdata( );
                        friend void display (alpha abc);
                        {
                                cout<<"value of x = "<<abc.x;
                                cout<< endl;
                        }
        };
        void sample :: getdata ( )
        {
                cout<<"enter value for x \n";
                cin>>x;
        }
        void main ( )
        {
                alpha a;
                a.getdata ( );
                cout<<"accessing private data by non-member function"<<endl;
                display(a);
        }
```

6.7 The 'This' keyword

The members functions of every object have access to a sort of magic pointer named this, which points to the object itself. Thus any member function can find out the address of the object of which it is a member.

Example:

```
        // demonstration of this key word
        # include<iostream.h>
        class sample
        {
                private:
                        char chararra[10];
                public:
                        void reveal( )
                                cout<<"my object's address is "<< this;
        };
        void main( )
        {
                sample s1,s2,s3;
                s1.reveal( );
                s2.reveal( );
                s3.reveal( );
                }
```

The main( ) program in this example creates three objects of type sample, then asks each object to print its address, using the reveal( ) member function. This function prints out the value of the this pointer.

Out put is

```
my object's address is 0x8f5effec
my object's address is ox8f5effec
my object's address is ox8f5effec
```

Accessing Member data with this

When you call a member function, it comes into existence with the value of this set of the address of the object for which it was called. The **this** pointer can be treated like any other pointer to an object and can thus be used to access the data in the object to points

```
// demonstration of accessing member function using this keyword
# include<iostream.h>
class example
{
  private:
     int value;
  public:
     inline void display( );
     {
        this-> value = 35;
        cout<<"contents of the value =" <<this->value;
        cout<<endl;
}
void main( )
{
  example obj1;
  obj1.display( );
}
the output is
  contents of the value = 20
```

6.8 Static class member

The static variables are automatically initialized to zero unless it has been initialized by some other value explicitly.
Static member of a class can be categorized into two types.

1. Static data member
2. Static member function

Static data member

Static data members are data objects that are common to all the objects of a class. They exist only once in all object of this class. The static members are used in information that is commonly accessible. This property of static data members may lead a person to think that static members are basically global variables. This is not true. Static members can be any one of the groups: public, private and protected, but not global data.

Example:

```
   // static data member
   # include<iostream.h>
   class example
   {
    private:
            static int counter;
    public:
            void display( );
 };
 int example::counter =100;
 void example::display ( )
 {
    int i;
 for(i =0; i<=10; ++I)
    counter = counter+1;
 }
 cout<<"sum of the counter  value ="<<counter
 cout<< endl;
 }void main( )
 {
    example obj1;
 int i;
 for(i = 0; i <=3; ++i)
 {
    cout<< "count="<<I+1<<endl;
    obj.display( );
 cout<<endl;
 }
 output is
    count  = 1
            sum of counter value  = 155
```

```
count = 2
           sum of counter value = 210
count = 3
           sum of counter value = 265
```

Static member function

The key word static is used to precede the member function to make a member function static. The static function is a member function of class and the static member function can manipulate only on static data member of the class. The static member function acts as global for member of its class without affecting the rest of the program. The purpose of static member is to reduce the need for global variables by providing alternatives that are local to a class. A static member function is not part of objects of a class. Static members of a global class have external linkage. A static member function does not have a **this** pointer so it can access nonstatic members of its class only by .(dot) or ->

The static member function cannot be a virtual function. It cannot be declared with the keyword const.

Example:

```
  //demonstration of static member function
  # include<iostream.h>
  class example
  {
     private:
        static int count;              //static data member declaration
     public:
        example( );
        static void display( );
};
//static data definition
int example::count = 0;
example::example( )
{
  ++count;
}
void example::display( )
{
  cout<<"counter value ="<< count << endl;
}
```

```
void main( )
{
      cout<<"before instantiation of the object"<< endl;
      example::display ( );
      example e1,e2;
      cout<<"after instatntiation of the object"<<endl;
      example::display ( );
}
output is
  before instantion of the object
counter value =0
  after instatiation of the object
counter = 2
```

Some Examples

A class encapsulates both data and functions manipulating them into a single unit. It can be further used as an abstract

```
<iostream.h>
const int MAX_ITEMS = 25;           // Maximum number of items that a bag can
hold
class bag
{
         private :
                  int contents[MAX_ITEMS];   // bag memory area
                  int itemcount              // number of items present in a bag
         public :
                  // sets itemcount to empty
                  void setempty()
                  {
                        itemcount=0;  // when you purchase a bag, it will be empty
                  }
                  void put(int item)         // puts item into bag
                  {
                           contents[itemcount++]=item;  // counter update
                  }
                  void show();
};

// display contents of a bag
```

```
void bag::show()
{
 for(int i=0;i<itemcount;i++)
  cout << contents[i] <<" ";
  cout << endl;
}

void main()
{
 int item;
 bag b1;
 b1.setempty();// set bag to empty
while (1)
{
  cout << "Enter item number to be put into the bag <0-no item>:";
  cin >> item;
  if (item == 0)              // items ends, break
           break;
 b1.put(item);
 cout << "Items in bag :";
 b1.show();
}
}
```

Run

```
Enter item number to be put into the bag <0-no item>: 1
Items in bag : 1
Enter item number to be put into the bag <0-no item>: 3
Items in bag : 1 3
Enter item number to be put into the bag <0-no item>: 2
Items in bag : 1 3 2
Enter item number to be put into the bag <0-no item>: 4
Items in bag : 1 3 2 4
Enter item number to be put into the bag <0-no item>: 0
```

In main(), the statement

```
 bag b1;
```

creates the object bag without initializing the itemcount to 0 automatically. However, it is performed by a call to the function setempty() as follows:

```
        bag.setempty();                    // set bag to empty
```

According to the philosophy of OOPs, when a new object such as bag is created, it will naturally be empty. To provide such a behavior in the above program, it is necessary to invoke the member function setempty explicitly. In reality, when a bag is purchased, it might contain some items placed inside the bag as gift items. Such a situation in C++ can be simulated by

```
                Bag b1 = 2;
```

It creates the object bag and initializes it with 2, indicating that the bag is sold with two gift items. It resembles the procedure of initialization of a built-in data type during creation, i.e., there must be a provision in C++ to initialize objects during creation itself.

It is therefore clear that OOPs must provide a support for initializing objects when they are created, and destroy them when they are no longer needed. Hence, a class in C++ may contain two special member functions dealing with the internal workings of a class. These functions are the constructors and the destructors. A constructor enables an object to initialize itself during creation and the destructor destroys the object when it is no longer required, by releasing all the resources allocated to it. These operations are called object initialization and clean up respectively.

6.9 Short Summary

- The concept of code reusage can be implemented with the help of Overloading functions
- Member variables and member functions are wrapped in a class
- Object is a run time entity
- Public, private and protected are the three access specifiers
- The members functions of every object have access to a sort of magic pointer named this, which points to the object itself
- Static member functions are acting as a global member functions

6.10 Brain Storm

1. What is function overloading?
2. List the merits and demerits of function overloading over the conventional functional usage.
3. What are the scope rules governing the function overloading?

4. What is meant by operator overloading?
5. List the C++ operators that can be overloaded for binary usage.
6. Describe how the data member of a class can be initialized in C++.
7. What is meant by 'this' operator?
8. What is a static class member?
9. What is a class?
10. Explain the following with respect to the object oriented programming: Private, protected, public
11. What is a friend function?
12. How is a heap memory allocated in C++?

Lecture - 7

Unions, Nested Classes, Constructors & Destracters

Objectives

In this lecture you will learn the following

- Knowing about Arrays
- Rules for writing constructors & Destructors

Lecture - 7

7.1 Snap Shot

C++ allows the user to create an array of any data type including user-defined data types. Thus, an array of variables of a class data type can also be defined, and such variables are called an array of objects. Union can be used as a user defined data type whose size is sufficient to contain one of its member. Like control structures classes can also be nested. The concept of constructors and Destructors are discussed.

7.2 Array Of Class Objects

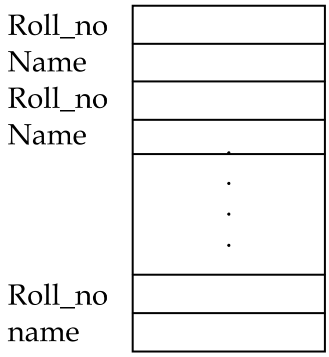
An array of objects is often used to handle a group of objects, which reside contiguously in the memory. Consider the following class specification:

```
class student
{
 Private :
  int roll_no;
  char name[20];

public :

 void setdata(int roll_no_in,char *name_in);
 void outdata();
};
```

The identifier student is a user-defined data type and can be used to create objects that relate to students of different courses. The following definition creates an array of objects of the student class:

```
student science[10];
student medical [5];
student engg[25];
```

The array science contains ten objects, namely science [0]....science[9] of type student class, the medical array contains 5 objects and the engg array contains 25 objects.
An array of objects is stored in the memory in the same way as a multidimensional array created at compile time. The representation of the objects is created; member functions are stored separately and shared by all the objects of student class.



An array of objects behaves similar to any other data-type array. The individual element of an array of objects is referenced by using its index, and member of an object is accesses using the dot operator. for instance, the statement.

```
Engg[i].setdata(10,"Rajkumar");
```

Sets the data members of the ith element of the array engg. Similarly, the statement

```
Engg[i].outdata();
```

Will display the data of the ith element of the array engg[i]. The program student1.cpp illustrates the use of the array of objects.

```
//student1.cpp: array of student data type

#include <iostream.h>
#include<string.h>
class stduent
{
 private:

  int roll_no;
 char name[20];

public:

 void setdata(int roll_no_in,char *name_in)
 {
  roll_no = roll_no_in;
 strcpy(name,name_in);
}
void outdata()
{
 cout << “Roll No = “ << roll_no <<endl;
 cout << “Name = “ <<name <<endl;
}
};

void main()
{
 int i,roll_no,count;
 char response,name[20];
 student s[10];
 count=0;
for(i=0;i<10;i++)
{
cout << “Initialize student object (y/n) :”;
 cin >> response;
 if ( response ==’y’ | | response ==’Y’)
 {
  cout << “Enter roll no. Of student :”;
 cin >> roll_no;
 cout << “Enter name of student :”;
s[i].setdata(roll_no,name);
count++;
}
else
 break;
}
cout << “student details ...”<<endl;
for(i=0;i<count;i++)
 s[i].outdata();
}
```

Run

```
Initialize student object (y/n) : y
Enter roll no. Of student : 1
Enter name of student : Rajkumar
Initialize student object (y/n) : y
Enter roll no. Of student : 2
Enter name of student : Tejaswi
Initialize student object (y/n) : y
Enter roll no. Of student : 3
Enter name of student : Savithri
```

```
Initialize student object (y/n) : n
Student details ...
Roll no = 1
Name = Rajkumar
Roll no = 2
Name = Tejaswi
Roll no = 3
Name = Savithri
```

In main(), the statement

```
Student s[10];
```

Creates an array of 10 possible objects of the student class. It should be clearly understood that an array of objects allow better organization of the program instead of having 10 different variables and each one of them is the object of the student class. Note that the subscripted notation used for object is similar to the manner in which arrays of other data types are usually handled. The statement

```
S[i].outdata();
```

Executes the outdata() member function in the student class for the ith object of the s array.

7.3 Unions and class

Union is a user defined data type whose size is sufficient to contain one of its members. At most, one of the members can be stored in a union at any time. A union is also used for declaring classes in C++. The members of a union are public by default.

A union allows to store its members only one at a time. A union may have member functions including constructors and destructors, but not virtual functions. A union may not have base class. An object of a class with a constructor or a destructor or a user defined assignment operator cannot be a member of a union. A union can have no static data member.
The general form is

```
union user_defined_name
{
private:
        //data;
```

```
                //methods;
        public:
                //methods;
        };
        user_defined_name object;
```

Example:

```
        union sample
        {
                public:
                        int a;
                        char name;
                        void display ( );
                        void sum ( );
        };
// sample program
 # include<iostream.h>

union sample
{
        private:
                int x;
                float y;
        public:
                void getinfo ( );
                void disinfo ( );
};
void sample :: getinfo ( )
{
        cout<<"value of x (in integer):";
        cin>> x;
        cout<<"value of y (in float):";
        cin>> y;
}
void sample:: disinfo( )
{
        cout<< endl;
        cout<<"x = "<<x<< endl;
        cout << "y= " << y << endl;
}
```

```
void main( void)
{
        sample obj;
        cout<<"enter the following information"<< endl;
        obj.getinfo( );
        cout<<"\ content of union"<< endl;
        obj.disinfo( );
}
```

the output is

```
enter the following information
value of x (in integer ) : 45
value of y( in float ) :  9.89

content of union
x = 45
y = 9.89
```

7.4 Nested classes

Including other class declarations inside a class can increase the power of abstraction. A class declared inside the declaration of another class is called nested class. Nested classes provide classes with non-global status. Host and nested classes follow the same access rules for members that exist between non-nested classes. Nested classes could be used to hide specialized classes and their instances within a host class.

The general form of nested class declaration is:

```
Class outer_class_name
        {
                private:
                        // data;
                public:
                        //methods;
                class inner_class_name
                {
                        private:
                                //data for inner class;
                        public:
                                //methods for inner class;
                }; // end of inner class declaration
        };  // end of outer class declaration
```

A member of a **class** may itself be a class. Such nesting enables building of very powerful data structures. The student class can be enhanced to accommodate the date of birth of a student. The new member data type date is a class by itself as shown below

```
 class student
 {
          private:
                   int rollno;
                   char name[25];
                   char branch [15];
                   int marks;
          public:
                   class date
                   {
                            int day;
                            int month;
                            int year;
                            date ( )
                            {
                            read ( );
                            }
                   }birthday;;
          student ( )
          {
                   ------------
                   ------------
          }
          student ( )
          {
                   ------------
                   ------------
          }
          read ( )
          {
                   cin>>rollno;
                   -----------------
                   birthday.read ( );
          }
                   ___________
          };
```

The embedded class date is declared within the enclosing class declaration. An object of type student can be defined as follows:

```
student s1;
```

The year in which the student s1 was born can be accessed as follows:

```
s1.birthday.year
```

A statement such as,

```
s1.date.day = 2;                //error
```

is invalid, because members of the nested class must be accessed using its object name.

The feature of nesting of classes is useful while implementing powerful data structures such as linked lists and trees. For instance, the stack data structure can be implemented having a node data member, which is an instance of another class (node class).

Member functions of a nested class have no special access to members of enclosing class. Member functions of an enclosing class have no special access to member of a nested class.

The following class declaration illustrates how the member functions of a nested class are accessed.

```
Class outer
{
        int a;
        void out_funt(int b);
        class inner
        {
                int x;
                void inner_funt(int y);
        };
};

outer ob1;                      // creating an object for outer class
outer :: inner :: ob2;          //creating an object for the inner class
ob1.outer_funt( );              //accessing an outer class member function
ob2.inner.funt( );              // accessing of inner class member function
```

When a class is defined as a member of another class, it contains only the scooping of outer class. The object of an outer class does not contain the object of the inner class.

```
//classes within classes demonstration
# include <iostream.h>
# include<string.h>

class student_info
{
private:
        char name[20];
        int rollno;
        char sex;
public:
        student_info(char *na, int rn, char sx);
        void display ( );
        class date
        {
                private:
                        int day;
                        int month;
                        int year;
                public:
                        date (int dy, int mh, int yr);
                        void show_date ( );
                class age_class
                {
                        private:
                                int age;
                        public:
                                age_class(int age_value);
                                void show_age ( );
                };
        };
};
student_info :: student_info(char *na, int rn, char sx)
{
strcpy(name, na);
rollno = rn;
sex = sx;
}
student_info :: date :: date ( int dy, int mh, int yr)
{
day = dy;
```

```
 month = mh;
 year = yr;
 }
 student_info : : date : : age_class : : age_class(int age_value)
 {
 age = age_value;
 }
 void student_info :: display ( )
 {
 cout <<"student's name, roll number, sex, date of birth, age \n";
 cout <<"_____________________________________________\n";
 cout << name << "                   "<<'\t';
 cout<<rollno<<"                              ";
 cout<<sex<<"              ";
}
void student_info ::date :: show_date ( )
{
 cout<<day<< '/ '<< month << ' /' <<year<<'\t';
}
void student_info::date::age_class:: show_age ( )
{
 cout<< '\t' << age << endl;
 cout <<" ______________________________________________"<< endl;
 }
 void main( )
          {
                    student_info obj1("Teresa", 78909,'f');
                    student_info::date obj2(25,3,67);
                    student_infor::date::age_class obj3(33);
                    obj1.display ( );
                    obj2.show_date ( );
                    obj3.show_age( );
          }
```

Out put of above program is

```
student's name, roll number, sex , date of birth, age
____________________________________________
Teresa          78909      f    25/3/67      33
____________________________________________
```

7.5 Constructors

A constructor is a special function for automatic initialization of an object. Whenever an object is created, the special member function, i.e., the constructor will be executed automatically. A constructor function is different from all other nonstatic member functions in a class because it is used to initialize the variables of whatever instance being created.

Rules for writing constructors:

1. A constructor name must be the same as that of its class name.
2. It is declared with no return type
3. It cannot be declared const or volatile but constructor can be invoked for a const and volatile object
4. It may not be static
5. It may not be virtual
6. It should have public or protected access within the class

The general form of constructor declaration is

```
Class user_name
{
        private:
                __________
                __________
        protected:
                __________
                __________
        public:
                user_name( );           // constructor
                ___________
                ___________
};
user_name :: user_name( )
{
__________
___________
}
```

Example:

```
Class student
{
        private:
                char name[20];
                int stcode;
                char address[20];
        public:
                student ( );    // constructor
                void get_data( );
                void put_data( );
}
student :: student( )           //constructor
{

}
```

Constructors and destructors can be explicitly called. A constructor is automatically invoked when an object begins to live. The constructor called before main( ) starts for execution. The constructors can be invoked whenever a temporary instance of a class needs to be created.

Example program:

```
//program generates Fibonacci numbers using constructors
# include<iostream.h>
class fibonacci
{
        private:
                long int f0,f1,fib;
        public:
                fibonacci ( )
                {
                        f0 = 0;
                        f1 = 1;
                        fib = f0 + f1;
                }
                void increment( )
                {
                        f0 = f1;
                        f1 = fib;
                        fib = f0 + f1;
                }
                void display ( )
                {
                        cout<<"fibonacci number = "<<fib<<'\t';
                }
        };
void main(void)
{
        fibonacci number;
        for (int i = 0; i<=10;++i)
        {
                number.display( );
                number.increment ( );
        }
}
```

7.6 Copy constructors

Copy constructors are always used when the compiler has to create a temporary object of a class object. The copy constructors are used in the following situations:

1. The initialization of an object by another object of the same class.
2. Return of objects as a function value.
3. Stating the object as by value parameters of a function

The general form is:

```
class_name :: class_name(class_name &ptr)
where class_name is user defined class name and ptr
is pointer to a class object class_name.
```

Normally, the copy constructors take an object of their own class as arguments an produce such an object. The copy constructors usually do not return a function value as a constructors cannot return any function values.

```
// program fibonacci numbers using copy constructors
 # include<iostream.h>
 class fibonacci
 {
        private:
                long int f0,f1,fib;
        public:
                fibonacci ( )
                {
                        f0 = 0;
                        f1 = 1;
                        fib = f0 + f1;
                }
                fibonacci ( fibonacci &ptr)
                {
                        f0 = ptr.f0;
                        f1 = ptr.f1;
                        fib = ptr.fib;
                }
                void increment( )
                {
                        f0 = f1;
                        f1 = fib;
                        fib = f0 + f1;
                }
                void display ( )
                {
                        cout<<"fibonacci number = "<<fib<<'\t';
                }
        };
void main(void)
{
        fibonacci number;
        for (int i = 0; i<=10;++i)
        {
                number.display( );
                number.increment ( );
        }
}
```

7.7 Default constructors

The default constructor is a special member function which is invoked by the C++ compiler without any argument for initializing the object class. C++ compiler automatically generates default constructors if it is not defined.

The general form is

```
class user_name
{
        private:
                ___________
                ___________
        protected:
                ___________
                ___________
        public:
                user_name ( ); //default constructor
                ___________
                ___________
}
user_name :: user_name ( )     //without any parameters
```

Example:

```
// program to demonstrate default constructor
# include<iostream.h>
class student
{
        private:
                char name[20];
                int rollno;
                char sex;
                float height;
                float weight;
        public:
                student( );
                void display( );
};
student :: student( )
{
        name[0] = '\0';
        rollno = 0;
        sex = '\0';
        height = 0;
        weight = 0;
}
void student:: display ( )
{
        cout <<"name          ="<< name<<endl;
        cout <<"rollnumber    ="<< rollnoname<<endl;
        cout <<"sex           ="<< sex<<endl;
        cout <<"height        ="<< height<<endl;
        cout <<"weight        ="<< weight<<endl;
}
void main( )
{
student a;
cout <<"demonstration of default constructor\n";
a.display( );
}
```

7.8 Parameterized Constructors

Constructors can be invoked with arguments, just as in the case of functions. The argument list can be specified within braces similar to the argument-list in the function. Constructors with arguments are called parameterized constructors. The distinguishing characteristic is that the name of the constructor functions have to be the same as that of its class name. Another constructor with arguments could have been provided with one integer value to initialize the data members.

Example

```
 class test
{
        //data
        test(int data1)  // constructor with parameter
        {
                //data
        }
        -----------
};
test t1(2);         // 2 is passed a  parameter
test t2 = 3;        // 3 is passed as a parameter
```

Since C++ allows function overloading, a constructor arguments can co-exist with another constructor without arguments.

```
// sample program
# include<iostream.h>
const int MAX_ITEMS = 25;
class bag
{
        private:
                int contents[MAX_ITEMS];
                int itemcount;
        public:
                bag ( )
                {
                        itemcount = 0;
                }
                bag (int item )
                {
                        contents[0] = item;
                        itemcount = 1;
                }
                void put(int item)
                {
                        contents[itemcount++] = item;
                }
                void show ( );
};
void bag :: show ( )
{
        if (itemcount)
                for(int i = 0; i<itemcount ; i++)
                cout<< contents[I] << " ";
        else
                cout<<"Nil";
                cout << endl;
}
void main( )
int item;
bag b1;
bag :: bag ( )
bag b2 = 4;
bag :: bag( int item)
cout<< "gifted bag1 initally has :";
b1.show ( );
cout<<"gifted bag2 initially has :";
b2.show ( );
while (1)
{
        cout<<"enter item number to be put into the bag <0-no item>";
        cin>> item;
        if (item == 0)
                break;
        b2.put(item);
        cout<<"items in bag 2:";
        b2.show ( ):
}
}
```

the output is

```
gifted bag1 initally has : Nil
gifted bag2 inially has: 4
enter item number to be put into the bag2<0-no item> 1
```

```
items in bag2 : 4 1
enter item number to be put into the bag 2<0-no item> 5
items in bag2 4 1 5
```

The bag class has two constructors. The first constructor does not have arguments. The next constructor has a single argument

The statement

```
        bag b1;
```

creates the object b1 and initializes its data member by invoking the no-argument constructor bag :: bag ( )

7.9 Destructors

A destructor is a function that automatically executes when an object is destroyed. A destructor function gets executed whenever an instance of the class to which it belongs goes out of existence. The primary use of the destructor function is to release space on the heap.

Rules for writing destructor function:

1. A destructor function name is the same as that of the class it belongs except that the first character of the name must be a tilde(~).
2. It is declared with no return type.
3. It cannot be declared static, const or volatile
4. It takes no arguments and therefore cannot be overloaded
5. It should have public access in the class declaration.

The general form is

```
 class user_name
{
        private:
                //data variables;
                //member functions;
        public:
                user_name ( ); //constructor
                ~user_name( );          //destructor
                //other member functions;
                };
```

Example:

```
                class student
                {
                        char name[20];
                        int age;
                        float avg;
                        int mark1,mark2, mark3;
                public:
                        student( );             //construcor
                        ~student( );            //destructor
                        void getdata ( );
                        void display ( );

                }
```

```
// demonstration of  constructor and destructor
# include<iostream.h>
# include<stdio.h>
class accout
{
        privat:
                float balance;
                float rate;
        public:
                account ( );
                ~account ( );
                void deposit ( );
                void compound ( );
                void getbalance ( );
                void menu ( );
};
account :: account ( )          //constructor
{
        cout << "enter the initial balance \n";
        cin>>balance;
        cout<<"interest rate"<< endl;
        cin>> rate;
}
account :: ~account ( )         // destructor
```

```
{
        cout<<"data base has been deleted"<< endl;
}
void account :: deposit ( )
{
        float amount;
        cout<<"enter the amount"<< endl;
        cin>>amount;
        balance = balance + amount;
}
void account :: withdraw ( )
{
        float amount;
        cout<<"how much to withdraw?\n";
        cin>>amount;
        if ( amount <= balance)
        {
                balance = balance - amount;
                cout<<"amount drawn = "<<amount<< endl;
                cout<<"current balance = "<<balance<<endl;
        }
        else
                cout<<0;
}
void account :: compound ( )
{
        float interest;
        interest = balance * rate;
        balance = balance + interest;
        cout<<"interest amount = " << interest<< endl;
        cout<<"total amount = << balance<<endl;
}
void account :: getbalance ( )
{
        cout<<"current balance : = "<<balance<<endl;
}
void account :: menu ( )
{
        cout <<" press  d -> deposit"<<endl;
        cout <<" press  w -> withdraw"<<endl;
        cout <<" press  c -> compound interest"<<endl;
        cout <<" press  g -> get balance"<<endl;
        cout <<" press  q -> quit"<<endl;
        cout <<" option, please ?\n";
}
void main (void)
{
        class account acct;
        char ch;
        while ((ch = getchar ( ) ) ! = 'q')
        {
                switch (ch)
                {
                        case 'd':
                                acct .deposit ( );
                                break;
                        case 'w':
                                acct . withdraw ( );
                                break;
                        case 'c':
                                acct . compound ( );
                                break;
                        case 'g':
                                acct . getbalance ( );
                                break;
                }
        }
}
```

7.10 Short Summary

- An array allocates memory contiguously
- Union can be used to get dissimilar data input
- Nested classes are used to give more information
- Constructors are used for automatic initialization
- Destructors are used to destroy the constructor

7.11 Brain Storm

1 What is an array of class object?

2 How the array of class objects is defined in C++?

3 In what way a union data type is useful for constructing a class object in C++?

4 Explain the syntactic rules of defining the union data type using a class object.

5 What is a nested class?

6 How a nested class is defined and declared in C++?

7 List the merits and demerits of declaring a nested class.

8 What is a constructor? What are the uses of declaring a constructor member function in a program?

9 What are the rules governing the declaration of a constructor?

10 When does a constructor member function is invoked in a class?

11 List the merits and demerits of copy constructors

12 What is a default constructor?

13 Explain parameterized constructors

14 When does a destructor member function is invoked in a class?

Lecture - 8

Inheritance

Objectives

In this lecture you will learn the following

- Inheritance
- Types of Inheritance - Example
- Overriding Member Function
- Calling the Base Method
- Single & Multiple Inheritance

Lecture - 8

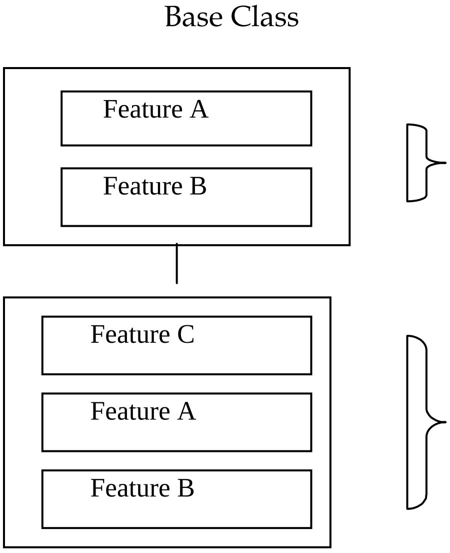
8.1 Snap Shot

This Lecture provides the different Forms and Types of Inheritance and also Explains Defining Derived Classes from Base Class, Ambiguity in Inheritance, Using Virtual Base Classes to avoid the ambiguity and Abstract Classes.

8.2 Inheritance

Inheritance is the process by which one object can acquire the properties of another object. This is important because it supports the concept of classification. If you think about it, most knowledge is made manageable by hierarchical classifications. For example, a Red Delicious apple is part of the classification apple, which in turn is part of the fruit class, which is under the larger class food. Without the use of classifications, each object would have to define explicitly all of its characteristics. However, through the use of classifications, an object need only define those qualities that make it unique within its class. It is the inheritance mechanism that makes it possible for one object to be a specific instance of a more general case. As you will see, inheritance is an important aspect of object-oriented programming.

Inheritance allows new classes to be built from older and less specialized classes instead of being rewritten from scratch. The technique of building new classes from the existing classes is called inheritance.



Inheritance, a prime feature of OOPs can be stated as the process of creating new classes (called derived classes), from the existing classes (called base classes). The derived class inherits all the capabilities of the base class and can add refinements and extensions of its own. The base class remains unchanged. The derivation of a new class from the existing class is represented in the above figure.

The main advantages of the inheritance are:

- Reusability of the code
- Reliability of the code
- Enhancement of the base class

When to Use Inheritance?

The following principles have to be followed to promote the use of inheritance in programming, which leads to code reuse, easy of code maintenance and extension:

- The most common use of inheritance and sub classing is for specialization, which is the most obvious and direct use of this rule. If two abstract concepts A and B are being considered, and the sentence A is a B makes sense, then it is probably correct in making A as a subclass of B. Examples car is a vehicle, triangle is a shape, etc.
- Another frequent use of inheritance is to guarantee that classes maintain a certain common interface; that is, they implement the same methods. The parent class can be a combination of implemented operations and operations that are to be implemented in the child classes. Often, there is no interface change between the super type and subtype – the child implements the behaviour described instead of its parent class. This feature has much significance with pure virtual function and will be discussed later.
- Using generalization technique, a subclass extends the behaviour of the super class to create a more general kind of object. This is often applicable when one is building on a base of existing classes that should not, or cannot be modified.
- While subclassing for generalization modifies or expands on the existing functionality of a class, subclassing for extension adds totally new abilities. Subclassing for extension can be distinguished from subclassing for generalization in derivation. Generalization must override at least one method from the parent, and the functionality is tied to that of the parent whereas extension simply adds new methods to those of the parent, and functionality is less strongly tied to the existing parent methods.
- In subclassing for limitation, the behavior of the subclass is more restricted than the behavior of the superclass. Like subclassing for generalization, subclassing for limitation occurs most frequently when a programmer is building on a base of existing classes that should not or cannot be modified.
- Subclassing for variance is useful when two or more classes have similar implementations, but there does not seem to be any hierarchical relationship is to factor out the common code into an abstract class, and derive the classes from these common ancestors.

- Subclassing by combination occurs when a subclass represents a combined feature from two or more parent classes.

8.3 Virtual Base Class

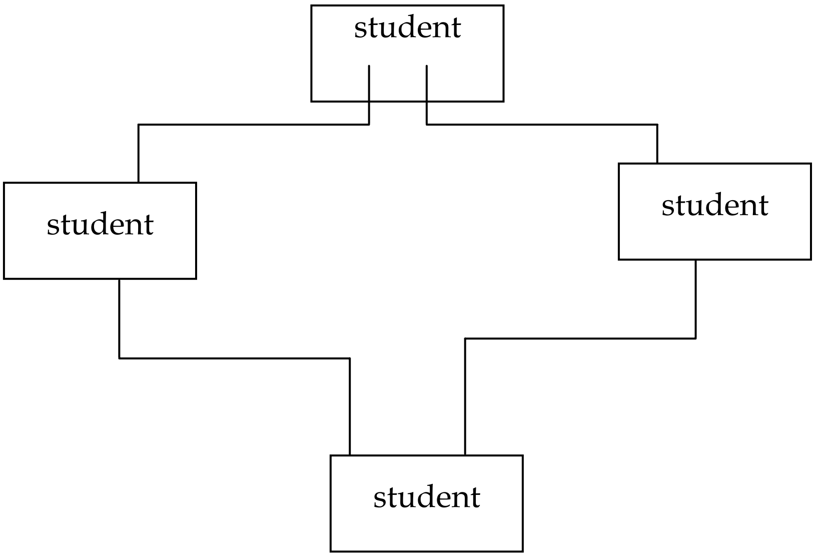
Consider a situation where all the three kinds of inheritance, namely multilevel, multiple and hierarchical inheritance are involved. This is illustrated in fig 8.11 . the 'child' has two direct base classes 'parent1' and 'parent2' which themselves have a common base class 'grandparent'. The 'child' inherits the traits of 'grandparent' via two separate paths. It can also inherit directly as shown by the broken line. The 'grandparent' is sometimes referred to as indirect base class.

Inheritance by the 'child' as shown in fig 8.11 might pose some problems. All the public and protected members of 'grandparent' are inherited into 'child' twice, first via 'parent1' and again via 'parent2'. This means, 'child' would have duplicate sets of the members inherited from 'grandparent'. This introduces ambiguity and should be avoided.

The duplication of inherited members due to these multiple paths can be avoided by making the common base class (ancestor class) as virtual base class while declaring the direct or intermediate base classes as shown below:

```
Class A                  //grandparent
{
        .......
        .......
};
class B1 : virtual public A          // parent1
{
        ...........
        ..........
};
class B2 : public virtual A          // parent2
{
        .............
        ..............
};
class C : public B1,public B2        //child
{
        ......          //only one copy of A
        ......          // will be inherited
};
```

when a class is made a virtual base class, C++ takes necessary care to see that only one copy of that class is inherited, regardless of how many inheritance paths exist between the virtual base class and a derived class. Note that the keywords virtual and public may be used in either order.



For example, consider again the student results processing system. Assume that the class sports derive the roll_number from the class student. Then, the inheritance relationship will be shown above. A program to implement the concept of virtual base class is illustrated in program.

```
#include<iostream.h>
class student
{
  protected:
           int roll_number;
  public:
          void get_number(int a)
          {
                 roll_number=a;
          }
          void put_number(void)
          {
                 cout << “Roll No :” << roll_number << “\n”;
          }
};
class test:virtual public student
{
  protected:
          float part1,part2;
 public:
```

```
            void get_marks(float x,float y)
            {
                        part1=x;
                        patr2=y;
            }
            void put_marks(void)
            {
                        cout << “Marks Obtained :” <<”\n”;
                        cout << “Part1 = “ << part1 << “\n”;
                        cout << “Part2 = “ << part2 << “\n”;
            }
};
class sports : public virtual student
{
  protected:
            float score;
  public :
            void get_score(float s)
            {
                        score=s;
            }
            void put_score(void)
            {
                        cout << “sports wt:” <<score << “\n\n”;
            }
};
class result : public test,public sports
{
 float total;
public:
            void display(void);
};
void result :: display(void)
{
            total = part1 +part2 +score;
            put_number();
            put_marks();
            put_score();
            cout << “Total Score :” << total << “\n”;
}
main()
{
  result studnet_1;
  student_1.get_number(678);
 student_1.get_marks(30.5,25.5);
student_1.get_score(7.0);
stduent_1.display();
}
```

The output would be

```
Roll No : 678
Marks Obtained :
Part1 = 30.5
Part2 = 25.5
Sport wt : 7
Total Score : 63
```

8.4 Container Class

C++ allows to declare an object of a class as a member of another class. When an object of a class is declared as a member of another class, if is called as a container class. Some of the examples for container classes are arrays, linked lists, stacks and queues.

The general syntax for the declaration of container class is,

```
Class user_defined_name_l
{
        ................
        ................
        ................
};
class user_defined_name_2
{
        .................
        ..................
};
class user_defined_name_n
{
        .................
        ..................
};

class derived_class
```

```
{
        user-defined-name-1 obj1;
        user_defined_name_2 obj2;
        ................
        ................
        user_defined_name_n objn;
};
```

for example, the following program segment illustrates how to declare the container class.

```
Class basic_info
{
        private:
        char name[20];
        public:
        void getdata();
        //end of class definition
};
class academic_fit
{

 private:
  int rank;
 public:
 void getdata();
}; // end of class definition

class  financial_assit
{
 private:
  basic_info bdata;
 academic_fit acd;
 float amount;
public:
 void getdata();
 void display();
}; //end of class definition

void main()
```

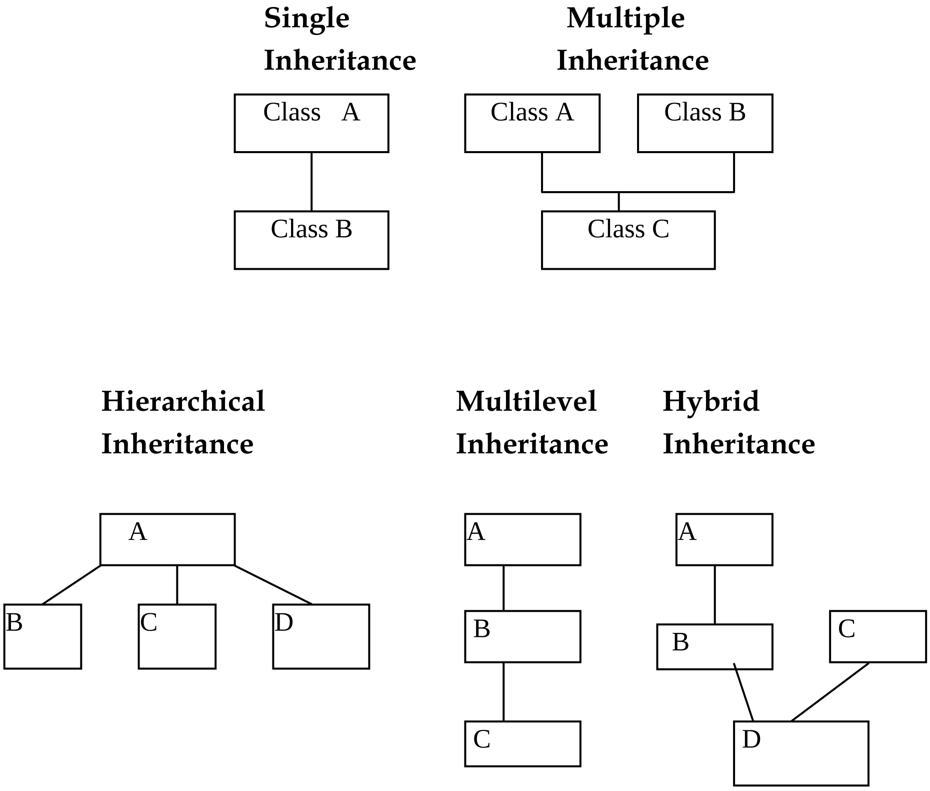
```
{
 financial_assit objf;
 ....................
 ...........................
}
```

8.5 Types of Inheritance

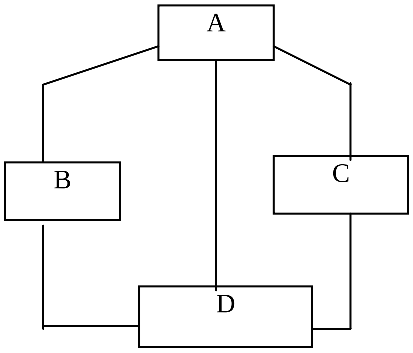
The derived class inherits some or all the features of the base class depending on the visibility mode and level of inheritance. Level of inheritance refers to the length of its (derived class) path from the root (top base class). A base class itself might have been derived from other classes in the hierarchy. Inheritance is classified into the following forms based on the levels of inheritance and interrelation among the classes involved in the inheritance process.

- Single Inheritance
- Multiple Inheritance
- Hierarchical inheritance
- Multilevel inheritance
- Hybrid Inheritance

The different forms of inheritance relationship are depicted in the following figure.



MultiPath Inheritance



Single Inheritance: Derivation of a class from only one base class is called Single Inheritance.

Multiple Inheritance: Derivation of a class from several base classes is called multiple inheritance.

Hierarchical Inheritance: Derivation of several classes from a single base class

Multilevel Inheritance: Derivation of a class from another derived class is called multilevel inheritance

Hybrid Inheritance: Derivation of a class involving more than one form of inheritance is known as hybrid inheritance.

Multipath inheritance: Derivation of a class from other derived classes, which are derived from the same base class is called multipath inheritance.

Class Revisited

C++ not only supports the access specifiers private and public, but also an important access specifier, protected, which is significant in class inheritance. As far as the access limit is concerned, within a class or from the objects of a class, protected access-limit is same as that of the private specifier. However, the protected specifier has a prominent role to play in inheritance. A class can use all the three visibility modes as illustrated below:

```
Class classname
{
        private:
                ......    // Visible to member functions within
                ......      // its class but not in derived class
        protected:
                ......      // Visible to member functions within its
                ......      //class and derived class
        public:
                ......      //visible to member functions within
                ......      //its class, derived classes and through objects
}
```

Defining the derived class:

The declaration of a singly derived class is as that same of an ordinary class. A derived class consists of the following components.

1. The Keyword class
2. The name of the derived class
3. A single colon
4. The type of derivation(private, public or protected)
5. The name of the base or parent class
6. The remainder of the class definition

The general syntax of the derived class declaration is as follows.

```
Class derived_class_name : private/public/protected base-class-name
{
        private:
                // Date members
                //Methods
        public:
                //Data members
                //Methods
        protected:
                //Data Members1
                //Methods
}
```

8.6 Single Inheritance

Single inheritance is the process of creating new classes from an existing base class. The existing class is known as the direct base class and the newly created class is called as a singly derived class.

Single inheritance is the ability of a derived class to inherit the member functions and variables of the existing base class.

Example

A program to read the derived class data members such as name, roll, sex, height, weight from the keyboard and display the contents of the class on the screen. This program demonstrates a single inheritance concept, which consists of a base class and a derived class.

```
#include <iostream.h>
#include <iomanip.h>
class basic_info
{
        private:
                char name[20];
                long int rollno;
                char sex;
        public:
                void getdata();
                void display();
};
class physical_fit: public basic_info
{
        private:
                float height;
                float weight;
        public:
                void getdata();
                void display();
};
void basic_info::getdata()
{
        cout<<"Enter a name ?\n";
        cout>>name;
        cout<<"Roll No.?\n";
        cin>>rollno;
        cout<<"Sex ?"\n";
        cin>>sex;
}
void basic_info::display()
{
        cout<<name<<"        ";
        cout<<rollno<<"        ";
        cout<<sex<<" ";
}
void physical_fit::getdata()
```

```
{
        basic_info::getdata();
        cout<<”Height ?\n”;
        cin>>height;
        cout<<”Weight ?\n”;
        cin>>weight;
}
void physical_fit::display()
{
        basic_info::display();
        cout<<setprecision(2);
        cout<<height<<”        “;
        cout<<weight<<’        “;
}
void main()
{
        physical_fit a;
        cout<<”Enter the following information \n”;
        a.getdata();
        cout<<”-----------------------------------------------\n";
        cout<<”Name           RollNo          Sex      Height Weight  \n”;
        cout<<”-----------------------------------------------\n“;
        a.display();
        cout<<endl;
        cout<<”-------------------------------------------------\n“;
}
```

Ambiguity in Single Inheritance

Whenever a data member and member function are defined with the same name in both the base and the derived classes, these names must be without ambiguity. The scope resolution operator (::) may be used to refer to any base member explicitly. This allows access to a name that has been redefined in the derived class.

For example, The following program segment illustrates how ambiguity occurs when the getdata() member function is accessed from the main() program.

```
class baseA
{
        public:
                void getdata()
                {
                 ....
```

```
                    ....
                    ....
                  }
};

class baseB
{
        public:
                void getdata()
                {
                  ....
                  ....
                  ....
                }
};

class derivedC:public baseA, public baseB
{
        public:
                void getdata()
                {
                  ....
                  ....
                  ....
                }
};
void main()
{
        derivedC obj;
        obj.getdata();
}
```

The members are ambiguous without scope operators. When the member function getdata() is accessed by the class object, naturally, the compiler cannot distinguish between the member function of the class baseA and the class baseB. Therefore it is essential to declare the scope operator explicitly to call a base class member as illustrated below:

```
obj.baseA::getdata();

obj.baseB::getdata();
```

8.7 Types of Derivation

Inheritance is a process of creating a new class from an existing class. While deriving the new classes, the access control specifier gives the total control over the data members and methods of the base classes. A derived class can be defined with one of the access specifiers, viz. Private, public and protected.

Public Inheritance

The most important type of access specifier is public. In a public derivation

- Each public member in the base class is public in the derived class
- Each protected member in the base class is protected in the derived class
- Each private member in the base class remains private in the base class

The general syntax of the public derivation is:

```
class  base_class_name
{
        ......
        ......
};
class derived_class_name:public base_class_name
{
        ..................
        ..................
};
```

For example, the following program segment illustrates how to access each member of the base class by the derived class members:

```
Class baseA
{
        private:
                int x;
        protected:
                int y;
        public:
                int z;
};
```

```
class derivedD:public baseA
{
        private:
                int w;
};
```

The class derivedD is derived from the base class baseA and the access specifier is public. The data members of the derived class derivedD is

```
int x
int y;
int z;
int w;
```

The following table shows the access specifier of the data member of the base class in the derived class:

Member in DerivedD	Access Status	How obtained
X	Not accessible	From class baseA
Y	Protected	From class baseA
Z	Public	From class baseA
W	Private	Added by class derivedD

Private Inheritance

In a private derivation,

- Each public member in the base class is private in the derived class.
- Each protected member in the base class is private in the derived class
- Each private member in the base class remains private in the base class and hence it is visible only in the based class

For eg,

```
        Class baseA
        {
                private:
                        int x;
                protected:
                int y;
        public:
                int z;
    };
```

```
class derivedD:private baseA
{
        private:
                int w;
};
```

The class derivedD is derived from the base class baseA and the access specifier is private. The data members of the derived class derivedD is

```
Int x,y,z,w;
```

The following table shows the access specifier of the data member of the base class in the derived class:

Member in DerivedD	Access Status	How obtained
X	Not accessible	From class baseA
Y	Private	From class baseA
Z	Private	From class baseA
W	Private	Added by class derivedD

Protected Inheritance

In a protected inheritance,

- Each public member in the base class is protected in the derived class.
- Each protected member in the base class is protected in the derived class
- Each private member in the base class remains private in the base class and hence it is visible only in the base class.

For eg,

```
        Class baseA
        {
                private:
                        int x;
                protected:
                int y;
        public:
                int z;
};
class derivedD:protected baseA
{
        private:
                int w;
};
```

The class derivedD is derived from the base class baseA and the access specifier is protected. The data members of the derived class derivedD is

```
int  x,y,z,w;
```

The following table shows the access specifier of the data member of the base class in

Member in DerivedD	Access Status	How obtained
X	Not accessible	From class baseA
Y	Protected	From class baseA
Z	Protected	From class baseA
W	Private	Added by class derivedD

8.8 Multiple Inheritance

In the original implementation of C++, a derived class could inherit from only one base class. Even with this restriction, the object-oriented paradigm is a flexible and powerful programming tool. The latest version of the C++ compiler implements the multiple inheritance. In this section, how a class can be derived from more than one base class is explained.

Multiple inheritance is the process of creating a new class from more than one base classes. The syntax for multiple inheritance is similar to that of single inheritance.

```
Class baseA
{
        ..................
        ..................
};
class baseB
{
        ................
        ................
};
class derivedC:public baseA, public baseB
{
        ..................
        ..................
        };
```

The class derivedC is derived from both classes baseA and baseB.

Multiple inheritance is a derived class declared to inherit properties of two or more parent classes(base classes). Multiple inheritance can combine the behaviour of multiple base classes in a single derived class. Multiple inheritance has many advantages over the single inheritance such as rich semantics and the ability to directly express complex structures. In C++, derived classes must be declared during the compilation of all possible combinations of derivations and the program can choose the appropriate class at run time and create object for the application.

In a single inheritance, a derived class has a single base class. In multiple inheritance, a derived class has multiple base classes. In a single inheritance hierarchy, a derived class typically represents a specialization of its base class. In a multiple inheritance hierarchy, a derived class typically represents a combination of its base classes. The rules of inheritance and access do not change from a single to a multiple inheritance hierarchy. A derived class inherits data members and methods from all its base classes, regardless of whether the inheritance links are private, protected or public.

Example Program:

A program to illustrate how a multiple inheritance can be declared and defined in a program. This program consists of two base classes and one derived class. The base class basic_info contains the data members:name, rollnumber, sex. An another base class academic_fit contains the data members:course,semester and rank. The derived class financial_assit contains the data member amount besides the data members of the base classes. The derived class has been declared as public inheritance. The member functions are used to get information of the derived class from the keyboard and display the contents of class objects on the screen.

```
#include <iostream.h>
#include <iomanip.h>
class basic_info
{
private:
        char name[20];
        long int rollno;
        char sex;
public:
        void getdata();
        void display();
};
class academic_fit
{
private:
        char course[20];
        char semester[10];
        int rank;
public:
        void getdata();
        void display();
};
class financial_assit: private basic_info,private academic_fit
{
private:
        float amount;
public:
        void getdata();
        void display();
};
void basic_info::getdata()
{
        cout<<"Enter a name? \n";
        cin>>name;
        cout<<"Roll no ?\n";
        cin>>rollno;
        cout<<"Sex ?\n";
        cin>>sex;
}
void basic_info::display()
{
        cout<<Name<<"       ";;
        cout<<rollno<<"       ";
        cout<<sex<<" ";
}
void academic_fit::getdata()
{
        cout<<"Course name (B.Tech/MCA/DCA etc) ?\n";
        cin>>course;
```

```
    cout<<”Semester (First/second etc) ? \n”;
    cin>>semester;
    cout<<”Rank of the student\n”;
    cin>>rank;
}
void academic_fit::display()
{
    cout<<course<<”       “;
    cout<<semester<<”   “;
    cout<<rank<<”         “;
}
void financial_assit::getdata()
{
    basic_info::getdata();
    academic_fit::getdata();
    cout<<”amount in rupees ? \n”;
    cin>>amount;
}
void financial_assit::display()

{
    basic_info::display();
    academic_fit::display();
    cout<<setprecision(2);
    cout<<amount<<”     “;
}
void main()
{
    financial_assit f;
    cout<<”Enter the following information for “;
    cout<<”Financial assistance\n”;
    f.getdata();
    cout<<endl;
    cout<<”Academic performance for financial Assistance\n”;
    cout<<”----------------------------------------------\n“;
    cout<<”Name  Rollno Sex  Course  Semester Rank Amount\n”;

    cout<<”------------------------------------------------\n“;
    f.display();
```

```
        cout<<endl;
        cout<<"--------------------------------------------\n";
}
```

Ambiguity in the Multiple inheritance

To avoid ambiguity between the derived class and one of the base classes or between the base class themselves, it is better to use the scope resolution operator::

Along with the data members and methods.

```
#include <iostream.h>
class A
{
        char ch;
  public:
        A(char c)
        {
                ch=c;
        }
        void show()
        {
                cout<<ch;
        }
};

class B
{
        char ch;
 public:
        B(char c)
        {
                ch=c;
        }
        void show()
        {
                cout<<ch;
        }
};

class C:public A, public B
{
        char ch;
public:
        C(char c1, char c2, char c3):A(c1),B(c2)
        {
                ch=c3;
        }
};

void main()
{
        C objc(‘a’,’b’,’c’);
  //    Objc.show();  Error: Field show is ambiguous in C
        cout<<”Objc.A::show()=”;
        objc.A::show();
        cout<<”Objc.B::show()=”;
        objc.B::show();
}
```

8.9 Short Summary

- Inheritance is the most powerful feature of Object oriented programming
- The main advantages are
- Reusability of the code
- To increase the reliability of the code
- To add some enhancement to the base class
- Single inheritance is the process of creating new classes from an existing base class.
- Multiple inheritance is the process of creating a new class from more than one base classes.

8.10 Brain Storm

1. Define: inheritance
2. What are the advantages of inheritance?
3. What is the difference between the base class and derived class?
4. What are the types of inheritance?
5. Explain the types of inheritance.
6. What is a container class?
7. List the pros and cons of a container class.

Lecture - 9

Polymorphism

Objectives

In this lecture you will learn the following

- Knowing briefly about Polymorphism
- Overloading Member Function
- Virtual Functions
- Restriction On Using Abstract Classes

Lecture - 9

9.1 Snap shot

This chapter focuses on the implementation of the concept of polymorphism using the keyword virtual. The emphasis is on how to realize the virtual functions; pure virtual functions; virtual base classes in C++. The early binding and the late binding of compilers are discussed.

9.2 Polymorphism

Object-oriented programming languages support polymorphism, which is characterized by the phrase "one interface, multiple methods." In simple terms, polymorphism is the attribute that allows one interface to be used with a general class of actions. The specific action selected is determined by the exact nature if the situation.

Polymorphism helps reduce complexity by allowing the same interface to be used to specify a general class of actions. It is the compiler's job to select the specific action (that is, method) as it applies to each situation. You, the programmer, don't need to make this selection manually. You need only remember and utilize the general interface.

The first object-oriented programming languages were interpreters, so compiled language. Therefore, in C++, both run-time and compile-time polymorphism is supported.

Polymorphism is another important OOP concept. Polymorphism means the ability to take more than one form. For example, an operation may exhibit different behavior in different instances. The behavior depends upon the types of data used in the operation. For example, consider the operation of addition. For two numbers, the operation will generate a sum. If the operands were strings, then the operation would produce a third string by concatenation. Fig 1.9 illustrates that a single function name can be used to handle different number and different types of arguments. This is something similar to a particular word having several different meanings depending on the context.

Polymorphism plays an important role in allowing objects having different internal structures to share the same external interface. This means that a general class of operations may be accessed in the same manner even though specific actions associated with each operation may differ. Polymorphism is extensively used in implementing inheritance.

9.3 Types of Polymorphism

There are two types of polymorphism

1. Function
2. Operator

Function Overloading

Function polymorphism is a neat C++ addition to C's capabilities. While default arguments let you call the same function using varying numbers of arguments, function polymorphism, also called function overloading , lets you use multiple functions sharing the same name. The word "polymorphism" means having many forms, so, function polymorphism lets a function have many forms. Similarly, the expression "function overloading " means you can attach more than one function to the same name, thus overloading the name. Both expressions boil down to the same thing, but we will usually use the expression function overloading ,it sounds harder working. You can use function overloading to design a family of functions that do essentially the same thing, but using different argument lists.

Overloaded functions are analogous to verbs having more than one meaning. For example, Miss Piggy can root at the ball park for the home team, and or she can root in the soil for truffles. The context (one hopes) tells you which meaning of roots is intended in each case. Similarly , C++ uses the context to decide which version of an overloaded function is intended.

The key to function overloading is a function's argument list, also called the function signature. If two functions use the same number and types of arguments in the same order, they have the same signature; the variable names don't matter. C++ enables you to define two functions by the same name provided that the functions have different signatures. The signature can differ in the number of arguments or in the type of arguments , or both. For example, you can define a set of print() functions with the following prototypes:

```
void print( const char * str , int width);       // #1
void print( double d, int width);               // #2
void print( long l, int width);                 // #3
void print( const char * str);                  // #4
```

when you then use a print() function, the compiler matches your use to the prototype that has the same signature:

```
print( "Pancakes", 15);                // use #1
print( "Syrup");                       // use #5
print(1999.0, 10);                     // use #2
print(1999,12);                  // use #4
print(1999L, 15);               // use #3
```

For example, print(“Pancakes”, 15) uses a string and an integer as arguments, and that matches prototype #1.

When you use overloaded functions, be sure you use the proper argument types in the function call. For example , consider the following statements.

```
unsigned int year = 3210;
print( year, 6);
```

Which prototype does the print() call match here? It doesn’t match any of them!. A lack of a matching prototype does not automatically rule out using one of the functions, for c++ will try to use standard type conversions to force a match. If, say, the only print() prototype were #2, the function call print(year, 6) would convert the year value to type double. But in the code above there are three prototypes that take a number as the first argument, providing three different choices for converting year. Faced with this ambiguous situation, c++ rejects the function call as an error.

Some signatures that appear different from each other can’t coexist. For example, consider these two prototypes:

```
double cube(double x);
double cube(double & x);
```

You might think this is a place you could use function overloading, for the function signatures appear to be different . But consider things from the compiler’s standpoint.
Suppose you have code like this:

```
cout <<cube (x);
```

The x argument matches both the double x prototype and the double &x prototype. Thus , the compiler has no way of knowing which function to use. Therefore, to avoid such confusion, when it checks function signatures, the compiler considers a reference to a type and the type itself to be the same signature.

The function matching process does discriminate between const and non-const varibles. Consider the following prototypes:

```
void dribble (char * bits);            //overloaded
void dribble ( const char *cbits);   //overloaded
void dabble (char * bits );            //not overloaded
void drivel (const char *bits);       //not overloaded
```

Here’s what various function calls would match:

```
const char p1[20] =  “How is the weather?”
char p2[20] = “How is business ?”
dribble (p1);    // dribble (const char *);
dribble (p2);   // dribble (char *);
dabble(p1) ;   // no match
dabble(p2);   // dabble (char *);
drivel (p1);   // drivel(const char *);
drivel(p2);   // drivel(const char *);
```

The dribble() function has two prototypes, one for const pointer and one for regular pointers, and the compiler selects one or the other depending on whether or not the actual arguments , but the drivel() function matches calls with either const or non-const arguments . The reason for this difference in behavior between drivel() and dabble() is that it’s valid to assign a non-const value to a const variable , but not vice versa.

Keep in mind that it’s the signature, not the function type, that enables function overloading . For example, the following two declarations are incompatible.

Therefore , C++ won’t permit you to overload grounk() in this fashion. You can have different return types, but only if the signatures also are different.

```
long grounk(int n, float m);
double grounk(float n, float m);
```

Operator Overloading

Operator Overloading is another example of C++ polymorphism. Operator overloading extends the overloading concept to operators, letting you assign multiple meanings to C++ operators. Actually, many C++ (and C) operators already are overloaded . For example, the * operator , when applied to an address , yields the value stored at that address. But applying * to two numbers yield the product of the values. C++ uses the number and type of operands to decide which action to take.

C++ lets you extend operator overloading to user-defined types, permitting you , say, to use the + symbol to add two objects. Again, the compiler will use the number and type of operands to determine which definition of addition to use. Overloaded operators often can make code look more natural. For example, a common computing task is adding two arrays. Usually , this winds up looking like the following for loop:

For (int i=0; i< 20; i++)
evening [i] = sam[i] + janet[i] // add element by element
But in C++ , you can define a class that represents arrays and that overloads the + operator so that you can do this:
evening = sam + janet; // add two array objects.

This simple addition notation conceals the mechanics and emphasizes what is essential, and that is another OOP goal.
To overload an operator, you use a special function form called an operator function. An operator function has the form:
operatorop(argument-list)

Where op is the symbol for the operator being overloaded. That is, operator+() overloads the + operator (op is +) and operator* () overloads the * operator (op is*) . The op has to be a valid C++ operator ; you can't just make up a new symbol. For example, you can't have an Operator @ ( ) function because C++ has no @ operator. But the operator[ ]() function would overload the [ ] operator because [ ] is the array-indexing operator. Suppose , for example, that you have a Salesperson class for which you define an operator+() member function to overload the + operator so that it adds sales figures of one salesperson object to another. Then, if district2, sid and sara all are objects of the Salesperson class , you can write this equation.
district2 = sid. Operator+(sara) ;

The function will then use the sid object implicity (because it invoked the method) and the sara object explicitly (because it is passed as an argument) to calculate the sum, which it then returns. Of course, the nice part is that you can use the nifty + operator notation instead of the clunky function notation.

C++ imposes some restrictions on operator overloading, but they are easier to understand after you have seen how overloading works.

Dynamic Binding

Binding refers to the linking of a procedure call to the code to be executed in response to the call. Dynamic binding means that the code associated with a given procedure call is not known until the time of the call at run-time. It is associated with polymorphism and inheritance. A function call associated with a polymorph reference depends on the dynamic type of that reference.

9.4 Overloading Member Functions

Aside from performing the special role of initialization, constructor functions are no different from other types of functions. This includes overloading. In fact, it is very common to find overloaded constructor functions. For example, consider the following program, which creates a class called date that holds a calendar date. Notice that the constructor is overloaded two ways.

```
#include<iostream.h>
#include<stdio.h>

class date
{
 int day.month,year;
public:
date(char *d);
date(int m,int d,int y);
void show_date();
};

date::date(char *d)
{

scanf(d,"%d%*c%d%*c%d",&month,&day,&year);
}

date::date(int m,int d,int y)
{
 day=d;
month=m;
year=y;
}

void date::show_date()
{
 cout << month <<"/" <<day;
 cout <<"/" << year <<"\n";
}

main()
{
 date ob1(12,4,96),ob2("10/22/97");
ob1.show_date();
ob2,show_date();
return 0;
}
```

in this program, you can initialize an object of type date, either by specifying the date using three digits to represent the month, day and year, or by using a string that contains the date in this general form:

mm/dd/yy

the most common reason to overload a constructor is to allow an object to be created by using the most appropriate and natural means for each circumstance. For example, in the following main(), the user is prompted for the date, which is input to arrays. This string can then be used directly to created. There is no need for it to be converted to any other form. However, if date() were not overloaded to accept the string form, you would have to manually convert it into three integers each time you created an object.

```
main()
{
 char s[80];
cout <<”Enter new date :”;
cin >> s;
date d(s);
d.show_date();
return 0;
}
```

in another situation, initializing an object of type date by using three integers may be more convenient. For example, if the date is generated by some sort of computational method, then creating a date object using date(iny,int,int) is the most natural and appropriate constructor to employ. The point here is that by overloading date’s and ease of use are especially important if you are creating class libraries that will be used by other programmers.

9.5 Overloading Non-member Function

Friend functions play a very important role in operator overloading by providing the flexibility denied by the member functions of a class. They allow overloading of stream operators (<< or >>) for stream computation on user defined data types. The only difference between a friend function and member function is that, the friend function requires the arguments to be explicitly passed to the function and processes them explicitly, whereas the member function considers the first argument implicitly. Friend functions can either be used with unary of binary operators. The syntax of operator overloading with friend functions is shown below

```
Friend return type operator symbol(arg1 [arg2])
{

        // body of operator friend function
}
```

Guidelines

It is essential to follow syntax and semantic rules of the language while extending the power of C++ using operator overloading. In fact, operator overloading feature opens up a vast vistas of opportunities for creative programmers. The following are some guidelines that needs to be kept in mind while overloading any operators to support user defined data types:

Retain meaning

Overloading operators must perform operations similar to those defined for primitive/basic datatypes. The operator + can be overloaded to perform subtraction; operator * can be overloaded to perform division operation. However, such definitions should be avoided to retain the initiative meaning of the operators. For example, the overloaded operator +() function operating on user-defined data-items must retain a meaning similar to addition. The operator + could perform the union operation on set data type, concatenation on string data type, etc.

Retain syntax

The syntactic characteristics and operator hierarchy cannot be changed by overloading. Therefore, overloaded operators must be used in the same way they for basic datatypes. For example, if c1 and c2 are the objects of complex class, the arithmetic assignment operator in the statement c1 +=c2;

Sets c1 to the sum of c1 and c2. The overloaded version of any operator should do something analogous to the standard definition of the language. The above statement should perform an operation similar to the statement
C1=c1+c2;

Use functions when Appropriate

An operator must not be overloaded if it does not perform the obvious operation. It should not demand the user effort in order to identify the actual operation performed by the operator. The main aim of overloading is to make the program code more readable. If the meaning of an operation to be performed by the overloaded operator is unpredictable or doubtful to the user, it is advisable to use a more descriptive and meaningful function name.

Avoid ambiguity

The existence of multiple data conversion routines performing the same operations, places the compiler in an ambiguous state. It does not know which one to select for conversion. For instance, existence of a one-argument constructor in the destination object's class and operator function also in the source object's class performing the same conversion function, confuses the compiler; it does not know which one to select and issues and error messages. Therefore, avoid defining multiple routines performing the same operation, which become ambiguous during compilation.

All operators cannot be overloaded

C++ supports a wide variety of operators, but all of them cannot be overloaded to operate in an analogous way on standard operators. These excluded operators are very few compared to the large number of operators, which qualify for overloading.

Operator Category	Operators
Member access:	(dot operator)
Scope resolution	: (global access)
Conditional	?: (conditional statement)
Pointer to member	*
Size of data type	sizeof(..)

An operator such as?: has an inherent meaning and it requires three arguments. C++ does not support the overloading of an operator, which operates on three operands. Hence, the conditional operator, which is the only ternary operator in the C++ language, cannot be overloaded.

Early Binding

Choosing a function in normal way, during compilation time is called as early binding or static binding. During compilation time the C++ compiler determines which function is used based on the parameters passed to the function or the function's return type. The compiler than substitutes the correct function for each invocation. Such compiler based substitution are called static linkage. By default C++ follows early binding.

9.6 Virtual Function

A virtual function is one that does not really exist but it appears real in some parts of a program.

Virtual functions are advanced features of the object oriented programming concept and they are not necessary for each C++ program.

The general form of virtual function is

```
Class user_defined_name
{
        private:
                ---------------
                ---------------
        public:
                virtual return_type function_name(argruments);
                virtual return_type function_name(argruments);
                virtual return_type function_name(argruments);
                ----------------
};
```

To make a member virtual, the key word virtual is used in the methods while it is declared in the class definition but not in the member function definition. The keyword virtual should be preceded by a return type of the function name. The compiler gets information from the keyboard virtual that it is a virtual functions and not a conventional function declaration.

Example:

```
        Class Sample
        {
                private:
                        int x;
                        int y;
                public:
                        virtual void display ( );
                        virtual int sum (  );
        };
```

1. The keyword virtual should not be repeated in the definition if the definition occurs outside the class declaration
2. A virtual function cannot be a static member because a virtual member is always a member of a particular object in a class rather than a member of the class as a whole.
3. A virtual function cannot have a constructor member function but it can have the destructor member functions
4. A destructor member function does not take any argument and no return type can be specified for it not even void
5. It is an error to redefine a virtual method with a change of return data type in the derived class with the same parameter types as those of a virtual method in the base class.
6. Only a member function of a class can be declared as virtual. It is an error to declare a non member function of a class virtual.

```
// virtual function demonstration
// array of pointers
# include<iostream.h>
class baseA
{
    public:
            virtual void display ( ){
            cout<< "one\n";
            }
};
class derivedB : public baseA
{
    public:
            virtual void display ( )
            {
                    cout<<"two\n";
            }
};
class derived C : public derivedB
{
    public:
            virtual void display ( )
            {
                    cout<<"three \n";
            }
};
void main( )
```

```
{
     baseA obja;
     derivedB objb;
     derivedC objc;
     baseA *ptr[3];
     ptr[0] = &obja;
     ptr[1] = &objb;
     ptr[2] = &objc;
     for(int i = 0; i <= 2; i++)
     ptr[ i ] -> display ( );
}
```

```
output of the above program is
one
two
three
```

Virtual functions with inline code substitution

Virtual functions can be declared as an inline code, being the run time binding of the computer. The inline code does not affect much of the programming efficiency. The compiler must get information about the functions, like from where they have to be invoked.

The general form is:

```
     Class base
     {
          private:
               //data;
          public:
               virtual inline return_type function_name(argruments);
               virtual inline return_type function_name(argruments);
     };
```

```
// demonstration of inline code with virtual function
# include<iostream.h>
class base
{
     private:
          int x;
          float y;
     public:
          virtual inline void getdata( );
```

```
                virtual inline void display( );
};
class derivedB : public base
{
        private:
                int rollno;
                char name[20];
        public:
                void getdata ( );
                void display ( );
};
void base :: getdata ( )
{
        cout <<"enter an integer \n";
        cin>> x;
        cout<<"enter a real number \n";
        cin<< y;
}
void base :: display( )
{
        cout<<" entered numbers are x = "<< x << "and  y = " << y;
        cout<< endl;
}
void derivedB :: getdata ( )
{
        cout <<"enter roll number of a student \n";
        cin >> rollno;
        cout<< "enter name of a student \n";
        cin>> name;
}
void derivedB :: display ( )
{
        cout << "roll number  student's name \n";
        cout << roll no << '\t' << name << endl;
}
void main ( )
{
        base *ptr;
        derivedB obj;
        ptr = &obj;
        ptr-> getdata ( );
        ptr -> display ( );
```

```
}
```

Output of the above program is

```
Enter roll number of a student
89001
enter name of a student
ganapathy

rollnumber            student's name
89001                 ganapathy
```

9.7 Pure Virtual Function

A pure virtual function is a type of function which has only a function declaration. It does not have the function definition. The following program illustrates how to declare a pure virtual function

```
// pure virtual function
# include<iostream.h>
class base
{
        private:
                int x;
                float y;
        public:
                virtual void getdata( );
                virtual void display( );
};
class derivedB : public base
{
        private:
                int rollno;
                char name[20];
        public:
                void getdata ( );
                void display ( );
};
void base :: getdata ( )
{
}
void base :: display( )
```

```
{
}
void derivedB :: getdata ( )
{
        cout <<"enter roll number of a student \n";
        cin >> rollno;
        cout<< "enter name of a student \n";
        cin>> name;
}
void derivedB :: display ( )
{
        cout << "roll number  student's name \n";
        cout << roll no << '\t' << name << endl;
}
void main ( )
{
        base *ptr;
        derivedB obj;
        ptr = &obj;
        ptr-> getdata ( );
        ptr -> display ( );
}
```

Output of the above program is

```
Enter roll number of a student
89001
enter name of a student
ganapathy
rollnumber     student's name
89001          ganapathy
```

9.8 Abstract Class

An abstract class is one that is not used to create objects. An abstract class is designed only to act as a base class (to be inherited by other classes). It is design concept in program development and provides a base upon which other classes may be built. In the previous example, the student class is an abstract class since it was not used to create any objects.

9.9 Restrictions on Using Abstract Classes

Abstract classes cannot be used for:

- Variables or member data
- Argument types
- Function return types
- Types of explicit conversions

Another restriction is that if the constructor for an abstract class calls a pure virtual function, either directly or indirectly, the result is undefined. However, constructors and destructors for abstract classes can call other member functions.

Pure virtual functions can be defined for abstract classes, but they can be called directly only by using the syntax:

abstract-class-name **::** function-name**( )**

This helps when designing class hierarchies whose base class(es) include pure virtual destructors, because base class destructors are always called in the process of destroying an object. Consider the following example:

```
#include <iostream.h>

// Declare an abstract base class with a pure virtual destructor.
class base
{
public:
   base() {}
   virtual ~base()=0;
};

// Provide a definition for destructor.
base::~base()
{
}
class derived:public base
{
public:
   derived() {}
   ~derived(){}
};
void main()
{
   derived *pDerived = new derived;

   delete pDerived;
}
```

When the object pointed to by pDerived is deleted, the destructor for class derived is called and then the destructor for class base is called. The empty implementation for the pure virtual function ensures that at least some implementation exists for the function.

Note In the preceding example, the pure virtual function base::~base is called implicitly from derived::~derived. It is also possible to call pure virtual functions explicitly using a fully qualified member-function name.

9.10 Short Summary

- Polymorphism means many form
- Static binding is possible in polymorphism
- Virtual function is one that does not exist really
- Virtual functions are advanced feature of OOP
- Late binding is nothing but dynamic binding
- Inline can also be given with virtual function

9.11 Brain Storm

1. What is polymorphism?
2. List the pros and cons of using polymorphism in OOP
3. Explain the concept of static binding.
4. What is a virtual function?
5. What are the syntactic rules to be observed while defining the virtual function?
6. Explain how the virtual base class is different from the conventional base classes of the OOP
7. Explain the syntactic rules for virtual base class in C++.
8. What is an abstract base class?
9. Explain dynamic binding.
10. What are the various techniques of defining pure virtual functions?

Lecture - 10

Java Architecture

Objectives

In this lecture you will learn the following

- Knowing the features of Java
- JDK
- Java Architecture

Lecture – 10

10.1 Snap Shot

This chapter gives you a detailed account on the features of java. Java is a general purpose, object-oriented programming language developed by Sun Microsystems of USA in 1991. Originally called as Oak by James Gosling, one of the inventors of the language, Java was designed for the development of software for consumer electronic devices like VCRs, TVs, toasters and such other electronic machines. The goal had a stron impact on the development team to make the language simple, portable and highly reliable. The java development kit is very much useful for developing application and applet programming.

10.2 Features of Java

Sun Microsystems officially describes Java with the following features:

- Compiled and interpreted
- Platform-Independent and Portable
- Object-Oriented
- Robust and Secure
- Distributed
- Familiar, Simple and Small
- Multithreaded and Interactive
- High Performance
- Dynamic and Extensible

The above mentioned features made Java the first application language of the World Wide Web.

Compiled and interpreted

Normally any language uses either compiler or interpreter to execute a program. But Java has both compiler and interpreter. So, Java is called as a two stage system. Java compiler first translates its source program into its byte code instructions. Byte codes are not machine instructions and therefore, in the second stage, Java interpreter generates machine code that can be directly executed by the machine. We can thus say that Java is compiled and interpreted.

Platform-Independent and Portable

Java programs can be easily moved from one computer to another, anywhere and anytime. Changes and upgrades in operating systems, processors and system resources will not force any changes in Java programs. This is the reason that Java has become popular in Internet. We can download a Java applet from a remote computer onto our local system via internet and execute it locally.

Java ensures portability in two ways. First, Java compiler generates bytecode instruction that can be implemented on any machine. Secondly, the size of the primitive data types are machine independent.

Object-Oriented

Java is a true object-oriented language. Almost everything in Java is an object. All the data and program codes reside within the classes and objects. Java comes with an extensive set of classes, arranged in packages, that we can use in our programs by inheritance.

Robust and Secure

Java is a robust language. It has strict compile time and run time checking for data types. It is designed as a garbage-collected language relieving the programmers virtually all memory management problems. Java also incorporates the concept of exception handling which captures series errors and eliminates any risk of crashing the system.

Security becomes an important issue for a language that is used for programming for internet. Java systems not only verify memory access but also ensure that no viruses are communicated with an applet. The absence of pointers in Java ensures that programs cannot gain access to memory locations without proper authorization.

Distributed

Java is designed as a distributed language for creating applications on networks. It has the ability to share both data and programs. Java applications can open and access remote objects on Internet as easily as they can do in a local system. This enables multiple programmers at multiple remote locations to collaborate and work together on a single project.

Simple, Small and Familiar

Java is a small and simple language. Java does not use pointers, preprocessor header files, goto statements and many others. It also eliminates operator overloading and multiple inheritance. Java uses many constructs of C and C++ and therefore, Java code "looks like a C++" code.

Multithreaded and Interactive

Multithreaded means handling multiple tasks simultaneously. Java supports multithreaded programs. This means that we need not wait for the application to finish one task before beginning another. For example, we can see any motion picture, and at the same time we can download any picture from remote computer. This feature greatly improves the interactive performance of graphical applications.

High performance

Java performance is impressive for an interpreted language, mainly due to the use of intermediate byte code. Java architecture is also designed to reduce overheads during runtime. Further, the incorporation of multithreading enhances the overall execution speed of Java program.

Dynamic and Extensible

Java is a dynamic language. Java is capable of dynamically linking in new class libraries, methods and object. Java can also determine the type of class through a query, making it possible to either dynamically link or abort the program, depending on the response.

Functions written in C and C++ can be used in Java. These methods are called as native methods. This facility enables the programmers to use the efficient functions available in these languages. Native methods are linked dynamically at run time.

10.3 Java Development Kit

Java Development Kit comes with a collection of tools that are used for developing and running Java programs. They are:

a. appletviewer
b. javac
c. java
d. javap
e. javah
f. javadoc
g. jdb

Appletviewer	Enables us to run java applets(without using java enabled web browser)

Javac	The Java compiler, which translates Java sourcecode to bytecode files that the interpreter can understand
Java	Java interpreter, which runs applets and applications by reading and interpreting bytecode files
Javadoc	Creates HTML-format documentation from Java source code files
Javah	Produces header files for use with native methods
Javap	Java disassdembler, which enables us to convert bytecode files into a program description
Jdb	Java debugger, which helps us to find errors in our programs

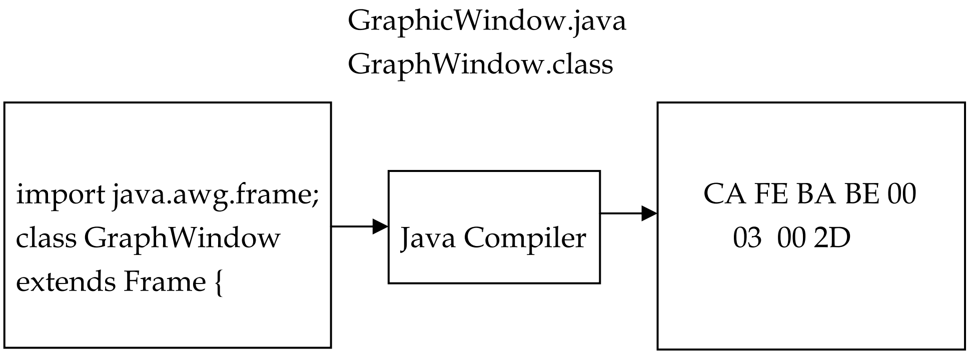
10.4 Java Architecture

Java's strength comes from its unique architecture. The designers of Java needed a language that was, above all, simple for the programmer to use. Yet in order to create reliable network applications, Java needed to be able to run securely over a network and, at the same time, work on a wide range of platform. Java fulfills all of these goals and more. The next few sections describe how Java works and how the features are that make Java a powerful network application development tool.

How Java Works

As with many other programming language, Java uses a compiler to convert human readable source code into executable programs. Traditional compilers produce code that can be executed by specific hardware . for example a Window 95 C++ compiler crates executable programs that work with intel x 86 compatible processors. In contrast, the Java compiler generates architecture independent byte codes. The byte codes can be executed by only a Java Virtual Machine (VM) which is an idealized Java architecture, usually implemented in software rather than hardware.

The compilation process is illustrated in Figure 1.4 java bytecode files are called class files because they contain a signal Java class. Classes will be described in detail in Chapter 3. For now, just think of a class as representing a group of related routines or an extended datatype. The vast majority of Java majority of Java programs will be composed of more than one class file.



```
import java.awg.frame;
class GraphWindow
extends Frame {
```

To execute java bytecodes, the VM uses a class loader to fetch the bytecodes from a disk or a network . Each class file is fed to a bytecode verifier that ensures the class is formatted correctly and will not corrupt memory when it is executed. The bytecode verification phase adds to the time it takes to load a class, but it actually allows the program to run faster because the class verification is performed only once, not continuously as the program runs.

The execution unit of the VM carries out the instructions specified in the bytecodes. The simplest execution unit is an interpreter, which is a program that reads the bytecodes. interprets their meaning, and then performs the associated function. Interpreters are generally much slower than native code compilers because they continuously need to look up the meaning of each bytecode during execution.

Fortunately, there is an elegant alternative to interpreting code, called just in time (JIT)

Compilation

The JIT compiler converts the bytecodes to native code instructions on the user's machine immediately before execution. Traditional native code compilers run on the developer's machine, are used by programmers, and produce nonprotable executables. JIT compilers run on the user's machine and are transparent to the user, the resulting native code instructions do not need to be ported because they are already at their destination. In the example both a Macintosh and Windows PC receive identical bytecodes, and each client performs a local JIT compilation.

Java-Enabled Browsers

A java-enabled Web browser contains its own NM. web documents with embedded java applets must specify the location of the main applet class file. The web browser then starts up the VM and passes the location of the applet class file to the class loader. Each class file knows the names of any additional class files that it requires. These additional class files may come from the network or from the client machine. this may require the class loader to make a number of additional class-loading operations before the applet starts. Note that supplemental classes are fetched only if they are actually going to be used or if they are necessary for the verification process of the applet.

After loading the class file, execution begins, and the applet is asked to draw itself in the browser window.

10.5 Short Summary

- Normally any language uses either compiler or interpreter to execute a program. But Java has both compiler and interpreter. So, Java is called as a two stage system.
- Java programs can be easily moved from one computer to another, anywhere and anytime.
- Java is designed as a distributed language for creating applications on networks.
- Functions written in C and C++ can be used in Java. These methods are called as native methods.
- Java Development Kit comes with a collection of tools that are used for developing and running Java programs.

10.6 Brain Storm

- Why is Java known as platform -neutral language?
- How is Java more secured than other language?
- List the features of Java.
- Describe the structure of a typical Java program.

Lecture - 11

Elements of Java

Objectives

In this lecture you will learn the following

- Knowing about Data types
- Understanding the concept of operators
- Shows how to create an array

Lecture - 11

11.1 Snap Shot

In this lecture you will learn about various Data types, Operators, Arrays, Control Structures and about Command line Argument.

11.2 Data types

Java is a strongly typed language

It is important to state at the outset that Java is a strongly typed language indeed, part of Java's safety and robustness comes from this fact.

1. Every variable has a type, every expression has a type, and every type is strictly defined.

2. Second, all assignments, whether explicit or via parameter passing in method calls, are checked for type compatibility. There are no automatic coercion or conversions of conflicting types as in some languages.

The Java compiler checks all expressions and parameters to ensure that the types are compatible. Any type mismatches are errors that must be forested before the compiler will finish compiling the class.

The Simple Types

Java defines eight simple (or elemental) types of data: **byte, short int, long, char, floats, Double,** and **Boolean.** These can be put in four groups.

- **Integers** This group includes Byte, short, int and long, which are for whole valued singed numbers.
- **Floating point numbers** This group includes float and double which represent numbers with fractional precision.
- **Characters** This group includes char, which represents symbols in a character set, like letters and numbers.
- **Boolean** This group includes boolean, which is a special type for representing true/false values.

You can use these types as-is, or to construct arrays your own class types. Thus, they form the basis for all other types of data that you can create.

The simple types represent single values not complex objects. Although Java is otherwise completely object oriented, the simple types are not. They are analogous to the simple types found in most other non-object-oriented language. The reason for this is efficiency. Making the simple types into objects would have degraded performance too much.

The simple types are defined to have an explicit range and mathematical behavior. Languages such as C and C++ allow the size of an integer to vary based upon the dictates of the execution environment. However, Java is different. Because of Java's portability requirement, all data types have a strictly defined range. For example, an **int** is always 32 bits, regardless of the particular platform. This allows programs to be written that are guaranteed to run *without porting* on any machine architecture. While environments, it is necessary in order to achieve portability.

Let's look at each type of data in turn.

Integers

Java defines four integer types. Byte, short, int, and long. All of these are signed, positive and negative values. Java does not support unsigned, positive only integers. Many other computer languages, including C/C++, support both singed and unsigned integers. However, Java's designers felt that unsigned integers were unnecessary. Specifically, they felt that the concept of unsigned was used mostly to specify the behavior of the high order bit, which defined the sign of an **int** when depressed as a number. Java manages the meaning of the high order bit differently, by adding a special "unsigned right shift" operator. Thus, the need for an unsigned integer type was eliminated.

The width of an integer type should not be though of as the amount of storage it consumes, but rather as the behavior it defines for variables and expressions of that type. The Java run time environment is free to use whatever size it wants, as long as the types behave as you devalued them. In fact, at least one implementation stores **bytes** and **shorts** as 32 bits (rather than 8 and 16 bit) values to improve performance, because that is the word size of most computers currently in use.

The width and ranges of these integer types vary widely, as shown in this table:

Name	***Width***	***Range***
long	64	-9,223,372,036,854,775,808 to 9,223,372,036,854,775,807
int	32	-2,147,483,648, to 2,147,483,647
short	16	-32,768, to 32,767
byte	8	- 128 to 127

Let's look at each type of integer.

Byte

The smallest integer type is **byte.** This is a signed 8 bit type that has a range from 128 to 127, Variables of types **byte** are especially useful when you're working with a stream of data from a network or file. They are also useful when you're other built in types.

Byte variables are declared by use of the **byte** keyword. For example, the following declares two **byte** variables called b and c:

byte b, c;

Short

short is a signed 16-bit type. It has a range from 32,768 to 32,767. It is probably the least used Java type, since it is defined as having its high byte first (*called big endian format). This type is mostly applicable to 16 bit computers, which are becoming increasingly scarce.

Here are some examples of short variable declarations:
short s;
short t;

int

The most commonly used integer type is **int**. It is a signed 32-bit type that has range from 2,147,483,648, to 2,147,483,647. In addition to other uses, variables of type in are commonly employed involving **bytes, shorts ints**, and literal numbers the entire expression is promoted to **int** before the calculation is done.

The **int** type is the most versatile and efficient type, and it should be used most of the time when you want to create a number for counting or indexing arrays or doing integer math. It may seem that using **short** or **byte** will save space, but there is no guarantee that Java won't promote those types to **int** internally anyway. Remember, type determines behavior, not size. (The only exception, is arrays, where byte, is guaranteed to use only one byte per array element, **short** will use two bytes, and int will use four).

Long

Long is a signed 64 bit type and is useful for those occasions where an **int** type is not large enough to hold the desired value. The range of a long is quite large. This makes it useful when bit, whole numbers are needed. For example, here is a program that computes the number of miles that lights will travel in a specified number of days.

```
        // Compute distance light travels using long variables.
        Class Light   {
        Public static void main (staring args [] ) {
        int lightspeed;
        long days;
        long seconds;
        long distance;
    // approximate speed of light in miles per second
    lightspeed  = 186000;
    days = 1000; // specify number of days here
    second  = days * 24 * 60 * 60;  // convert to seconds
    distance = lightspeed  * seconds ;   // compute distance
    System.out.print ("In"  + days );
    System.out.print ("days light will travel about") ;
    System.out.println (distance + "miles.");
   }
}
```

This program generates the following output:
In 1000 days light will travel about 16070400000000 miles.
Clearly the result could not have been held in an **int** variable.

Floating-Point Types

Floating point numbers, also known as real numbers are used when evaluating expression that require fractional precision. For example, calculation such as square root, or transcendentals such as sine and cosine, result in value whose precision requires a floating-point type. Java implements the standard (IEEE-754) set of floating point types an operator. There are two kinds of floating point types, float and double, which represent single and double precision numbers, respectively. Their width and ranges are shown here :

Name	Width in Bits	Range
Double	64	1.7e-308 t 1.7e+308
Float	32	3.4e-038 to 3.4e+038

Each of these floating point types is examined next.

Float

The type **float** specifies a **single precision** value that uses 32 - bits of storage. Single precision is faster on some processors and takes half as much space as double precision, but will become imprecise when the values are either very large or very small. Variables of type float are useful when you need a fractional component, but don't' require a large degree of precision. For example, float can be useful when representing dollars and cents.

```
Here are some example float variable declarations:
Float hightemp,         lowtemp;
```

Double

Double precision, as denoted by the double keyword, uses 64 bits to store a value. Double precision is actually faster than single precision on some modern processors that have been optimized for high speed mathematical calculations. All transcendental math functions, such as **sin**(), **cos**() , and **sqrt**(), return **double** values. When you need to maintain accuracy over many interative calculations, or are manipulating large values numbers, **double** is the best choice.

Here is a short program that uses double variables to compute the area of a circle:

```
// Compute the area of a circle .
class Area {
    public static void main (string args [ ] ) {
    double pi, r, a ;

    r = 10.8; // radius of circle
    pi = 3.1416;  // pi, approximately
    a  = pi * r * r; // compute area
    system.out.println ("Area of circle is" a) ;
         }
  }
```

Characters

In Java, the data type used to store characters is **char**. However, C/C++ programmers beware: **char** in Java is not the same as **char** in C or C++ . In C/C++ **char** is an integer type that is 8 bits wide. This is not the case in Java. Instead, Java uses Unicode to represent characerts. Unicode defines a fully international character set that can represent all of the characters found in all human languages. Thus, in Java **char** is a 16 bit type. The range of a **char** is 0 to 65,536. There are negative chars. The standard set of characters known as ASCII still ranges from 0 to 127 as always, and the extended 8 bit character set, ISO-Latin-1,ranges from 0 to 255. Here is a program that demonstrate **char** variables.

```
// Demonstrate char data type.
Class Chardemo {
     public static void main (string args [ ] ) {
     char  ch1,ch2;

     ch1=88; // code for X
     Ch2 = 'Y' ;
     System.out.print ("ch1 and ch2 :") ;
     System.out.print1n(ch1 + " " + ch2);
      }
}
```

This program displays the following output;

```
ch1 and ch2 : X Y
```

Notice that ch1 is assigned the value 88, which is the ASCII (and Unicode) value that corresponds to the letter X. As mentioned, the ASCII character set occupies that first 127 values in the Unicode character set. For this reason, all the "old tricks" that you have used with characters in the past will work in Java, too.

Even though **chars** are not integers, in many cases you can operate on them as if they were integers. This allows you to add two characters together, or to increment the value of a character variable. For example, consider the following program.

```
// char variables behave like integers.
class chardemo2
public static void main (string args [ ]) {
char ch1
ch1= 'X';
System.out.print1n ("ch1 contains"  +ch1);
     ch1++;   //  increment ch1
     System.out.print1n("ch1 is now" + ch1);
          }
     }
```

The output generated by this program is shown here.

```
ch1 contains X
ch1 is now Y
```

In the program, **ch1** is first given the value X. Next, **ch1** is incremented. This results in **ch1** containing Y, the next character in the ASCII (and Unicode) sequence.

Boolean

Java has a simple type, called **boolean**, for logical values. It can have only one of two possible values, true or false. This is the type returned by all relational operations, such as **a<b. Boolean** is also the type by the conditional expressions that given the control statements such as **if** and **for**.

```
// Demostrate boolean values.
class booltest {
public static void main (string args [ ] )       {
boolean b;
b=false;
System.out.print1n("b is + b ) ;
b=true;
System.out.print1n ("b is"  + b);

// a boolean value can control the if statement
if (b) system.out.print1n("This is executed.");

b=false;
if(b) system.out.print1n("This is not executed.");
// outcome of a relational operator is a boolean value
System.out.print1n ("10>9 is" +(10>9));
        }
  }
```

The output generated by this program is shown here:

```
B  is false
B is  true
This is executed.
10> 9 is true
```

There are three interesting things to notice about this program. First, as you can see, when a **Boolean** value is output by **print1n ( ),** "true" or "false" is displayed. Second, the value of a **Boolean** variable is sufficient, by itself, to control the **if** statement. There is no need to write an **if** statement like this:

```
If(b= =true) ….
```

Third, the outcome of a relational operator, such as <, is a boolean value. This is why expression 10>9 displays the value "true". Further, the extra set of parentheses around 10>9 is necessary because the + operator has a higher precedence than the>.

Integer Literals

Integers are probably the most commonly used type in the typical program. Any whole number value is an integer literal. Examples are 1,2,3, and 42. These are all decimal values, meaning they are describing a base 10 numbers. However, to specify a long literal, you will need to explicitly tell the compiler that the literal value is of type long. You do this by appending an upper-or lowercase L to the literal. For example, 0x7fffffffffffffffL or 9223372036854775807L is the largest **long.**

Floating-Point Literals

Floating-point numbers represented decimal values with a fractional component. For example 2.0,3.14and etc. represent valid standard-notation floating-point numbers. Scientific notation uses a standard-notation, floating-point number plus a suffix that specifies power of 10 by which the number is to be multiplied. The exponent is indicated by an E or e followed by a decimal number.

Boolean Literals

Boolean Literals are simple. There are only two logical values that a boolean value can have, true and false. The true literal in Java does not equal 1, nor does the false literal equal 0. In Java, they can only be assigned to variables declared as boolean, or used in expressions with boolean operators.

Character Literals

Characters in Java are indices into the Unicode character set. They are 16 bit values that can be converted into integers and manipulated with the integer operators, such as the addition and subtraction operators. Literal characters can be directly entered inside the quotes. All of the visible ASCII characters can be directly entered inside the quotes, such as 'a', 'z', and '@'.

The following tables shows the character escape sequence.

Escape Sequence	**Description**
\ddd	Octal character (ddd)
\uxxxx	Hexadecimal UNICODE character (xxxx)

\′	Single quote
\″	Double quote
\ \	Backslash
\ r	Carriage return
\n	New line (also known as line feed)
\f	Form feed
\t	Tab
\b	Backspace

Character Escape Sequence

String Literals

String literals in Java are specified like they are in most other languages by enclosing a sequence of characters between a pair of double quotes. Examples of string literals are

```
“Hello world”
“two\nlines”
“\ “this is in quotes\””
```

The escape sequences and octal/hexadecimal notations that were defined for character literals work the same way inside of string literals. One important thing to note about Java strings is that they must begin and end on the same line. There is no line continuation escape sequence as there is in other languages.

Variables

The variable is the basic unit of storage in Java program. A variable is defined by the combination of an identifier, a type, and an optional initializer. In addition, all variables have a scope, which defines their visibility, and lifetime. These elements are examined next.

Declaring a Variable

In Java, all variables must be declared before they can be used. The basic form of a variable declaration is shown here:

```
Type identifier [=value] [identifier [=vaule] ….];
```

The type is one of Java’s atomic types, or name of a class or interface. The identifier is the name of the variable. You can initialize the variable by specifying an equal sign and a value keep in mind that the initialization expression must result in a value of the same (or compatible) type as that specified for the variable. To declare more than one variable of the specified type use a comma - separated list.

Here are several example of variable declarations of various types. Note that some include an initialization.

```
int a,  b,  c;                 // declares three ints, a,  b, and c.
int d =3, e, f = 5            // declares three more ints, initializing d and f
byte z =22;                   // initializes z.
double pi = 3.14159;  // declare an approximation of pi
char x ='x';                 //  the variable x has the value 'x'.
```

The identifiers that you choose have nothing intrinsic in their names that indicates their type. Many readers will remember when FORTRAN predefined all identifiers from I through N to be of type INTEGER while all other identifiers were REAL . Java allows any properly formed identifier to have any declared type.

11.3 Operators

Arithmetic Operators

Arithmetic operators are used in mathematical expression the same way that they are used in algebra. The following table lists the arithmetic operators:

Operator	Result
+	Addition
-	Subtraction (also unary minus)
*	Multiplication
/	Division
%	Modulus
++	Increment
+=	Addition assignment
-=	Subtraction assignment
* =	Multiplication assignment
/ =	Division assignment
% =	Modulus assignment
- -	Decrement

The operands of arithmetic operators must be of a numeric type. you cannot use them on **boolean** types, but you can use them on **char** types, since the char type in Java is, essentially , a subset of **int.**

```
// Demonstrate the basic arithmetic operators.
class basicMath  {
public static void main (string args [ ] ) {
/ / arithmetic using integers
System.out.print1n (“Integer Arithmetic”)
int a = 1 + 1;
int b = a * 3;
int c = b  / 4;
int d = c – a;
int e = -d ;

System.out.println ( “a= “ +a);
System.out.println (“b =”  + b);
System.out.println (“c=”    + c);
System.out.println (“d =”   + d);
System.out.println (“ e =”   + e);

// arithmetic using doubles
System. out. println (“\no Floating Point Arithmetic”);
double da = 1 + 1;
double db = da * 3;
double dc = db / 4;
double  dd = dc – a;
double de  = - dd;

System.out.println (“da=” +da);
System.out.println (“db=” +db);
System.out.println (“dc=” +dc);
System.out.println (“dd=” + dd);
System.out.println (“de =” +de);
```

When you run this program, you will see the following output.

```
Integer Arithmetic
a= 2
b= 6
c=1
d = -1
```

```
e = 1

Floating point Arithmetic
da = 2
db = 6
dc = 1.5
dd = -0.5
de = 0.5
```

Arithmetic Assignment Operators

Java provides special operators that can be used to combine an arithmetic operation with an assignment. As you probably know, statements like the following are quite common in programming.

```
a = a + 4 ;
```

In java you can rewrite this statements as shown here:

```
a  + = 4;
```

This version uses the + = assignment operator. Both statements perform the same action they increase the value of **a** by 4.

Here is another example,

```
a = a % 2;
```

which can be expressed as

```
a % = 2;
```

In this case, the % = obtains the remainder of a/2 and puts that result back into a. There are assignment operators for all the arithmetic, binary operation. Thus, any statement of the forms. The assignment operators provide two benefits. First, they save you a bit of typing, because they are "shorthand" for their equivalent long forms. Second , they are implemented more efficiently by the Java run time system than are their equivalent long forms. For these reasons, you will often see the assignment operators used in professionally written Java programs.

These operators are unique in that they can appear both in **postfix** form, where they follow the operand as just shown, and **prefix** form, where they precede the operand. In the foregoing examples there is no difference between the **prefix** and **postfix** forms.

In **postfix** form, the previous value is obtained for use in the expression, and then the operand is modified. For example:

```
x = 42 ;
y = + + x ;
```

In this case, **y** is set to 43 as you would expect, because the increment occurs before **x** is assigned to **y**. Thus, the line **y = + + x**; is the equivalent of these two statement.

```
x = x + 1;
y = x;
```

however, when written like this,

```
x=42;
y=x++;
```

The value of **x** is obtained before the increment operator is executed, so the value of **y** is 42. Of course, in both cases **x** is set to 43. Here, the line y=x++; is the equivalent of these two statements:

```
y=x;
x=x+1;
```

The following program demonstrates the increment operator.

```
// Demonstrate ++.
class IncDec {
public static void main(String a[ ]);
        int a=1;
        int b=2;
        int c,d;
        c = ++b;
        d = a++;
        c++;
        System.out.println("a = " +a);
        System.out.println("b = " +b);
        System.out.println("c = " +c);
        System.out.println("d = " +d);
        }
}
```

The output of this program follows

```
a=2
```

b=3
c =4
d=1

The Bitwise Operators

Java defines several bitwise operators which can be applied to the integer types, long, int, short, char, and byte. These operators act upon the individual bits of their operands. They are summarized in the following table:

Operator	Result
~	Bitwise unary NOT
&	Bitwise AND
|	Bitwise OR
^	Bitwise exclusive OR
>>	Shift right
>>>	Shift zero fill
<<	Shift left
&=	Bitwise AND assignment
|=	Bitwise OR assignment
^=	Bitwise exclusive OR assignment
>>=	Shift right assignment
>>>=	Shift right zero fill assignment
<<=	Shift left assignment

The Bitwise Logical Operators

The bitwise logical operators are &, |, ^ and ~. The following table shows the outcome of each operation. In the discussion that follows, keep in mind that the bitwise operators are applied to each individual bit within each operand.

A	B	A|B	A&B	A^B	~A
0	0	0	0	0	1
1	0	1	0	1	0
0	1	1	0	1	1
1	1	1	1	0	0

The Bitwise NOT

Also called the bitwise complement, the unary NOT operator, ~, inverts all of the bits of its operand. for example, the number 42, which has the following bit pattern;

```
     00101010
becomes
     11010101
```

after the NOT operator is applied.

The Bitwise AND

The AND operator, &, produces a 1 bit if both operands are also 1. A zero is produced in all other cases. here is an example:

```
     00101010     42
    &00001111     15
     -------------
     00001010     10
```

The Bitwise OR

The OR operator, 1, combines bits such that if either of the bits in the operands is a 1, then the resultant bit is a 1, as shown here:

```
     00101010     42
    | 00001111     15
  -----------------
     00101111     47
```

The Bitwise XOR

The XOR operator, ^, combines bits such that if exactly one operand is 1, then the result is 1. Otherwise, the result is zero. The following example shows the effect of the ^. This example also demonstrates a useful attribute of the XOR operation. Notice how the bit pattern of 42 is inverted whenever the second operand has a 1 bit. Whenever the second operand has a 0 bit, the first operand is unchanged. you will find this property useful when performing some types of bit manipulations.

```
     00101010     42
    ^00001111     15
     ---------------
     00100101     37
```

Relational Operators

The relational Operators determine the relationship that one operand has to the other. Specifically, they determine equality and ordering. The relational operators are shown here:

Operator	Result
==	Equal to
!=	Not Equal to
>	Greater than
<	Less than
>=	Greater than or equal to
<=	Less than or equal to

The outcome of these operations is a **Boolean** value. The relational operators are most frequently used in the expressions that control the **if** statement and the various loop statements.

Assignment Operators

The assignment operator is the single equal sign, =. The assignment operator works in java much as it does in any other computer language. It has this general form:

```
var = expression;
```

here, the type of var must be compatible with the type of expression.

The assignment operator does have one interesting attribute that you may not be familiar with: it allows you to create a chain of assignments. for example, consider this fragment;

```
int x, y, z;
x=y=z=100;  //set x, y, and z to 100
```

The ? operator

Java includes a special ternary operator that can replace certain types of if-then-else statements. this operator is the ?, and it works in java much like it does in C and C++. It can seem somewhat confusing at first, but the ? can be used very effectively once mastered. The ? has this general form:

```
expression1? expression1: expression2
```

Here,expression1 can be any expression that evaluates to a boolean value. if expression1 is true, then expression2 evaluated; otherwise, expression3 is evaluated. The result of the ? operation is that of the expression evaluated. Both expression2 and expression3 are required to return the same type, which can't be void.

Here is an example of the way that the? is employed:

```
ratio = denom == 0 ? 0 : num/denom;
```

11.4 Control Structures

Control Statement

Java supports two selection statements **if** and **Switch**. These statements allow you to control the flow of your program's execution based upon conditions knows only during run time. If your background in programming does not include C/C++ you will be pleasantly surprised by the power and flexibility contained in these two statements.

The if statement is Java's conditional branch statement. It can be used to route program execution through two different paths. Here is the general form of the **if** statement .

```
if (condition ) statement1;
 else statment2;
```

Here, each statement may be a single statement or a compound statement enclosed in curly braces (that is, a block). The condition is any expression the returns a **Boolean** value. The **else** clause is optional.

The **if** works like this. If the condition is true then statment1 is executed. Otherwise, statement2 (if it exists) is executed. In no case will both statements be executed. For example, consider the following.

```
int a, b;
/ / ….
if(a < b)  a= 0;
else b= 0;
```

Here if **a** is less than **b**, then **a** is set to zero. Otherwise, **b** is set to zero. In no case are they both set to zero.

Nested ifs

A nested **if** is an if statement that is the target of another **if** or **else**. Nested **if**s are very common in programming. When you nest **if**s, the main thing to remember is that an else statement always refers to the nearest if statement that is within the same block as the else and that is not already associated with an else. Here is an example.

```
     if (i ==10 ) {
       if (j< 20) a=b;
      if (k>100) c=d; / / this if is
        else a =c;         / / associated with this else
    }
    else  a = d ;          / / this else refers to if  (i == 10)
```

As the comments indicate, the final else is not associated with **if (j<20,)** because it is not in the same block (even though it is the nearest **if** without an else.) Rather, the final **else** is associated with **if (i==10).** The inner else refers to **if (k>100,)** because it is the closes **if** within the same block.

The if-else-if Ladder

A common programming construct that is based upon a sequence of nested **if**s is the if else if ladder. It looks like this:

```
     if(condition)
       statement;
       else if (condition)
        statement;
      else if (condition);
        statement;
 .
 .
 .
else.
       statement.
```

This **if** statements are executed from the top down. As soon as one of the conditions controlling the if is **true**, the statement associated with that **if** is executed, and the rest of the ladder is bypassed. If none of the conditions is true, then the final else statement will be executed. The final else acts as a default condition; that is if all other conditional tests fail, then the last **else** statement is performed. If there is not final **else** and all other conditions are **false**, then no action will take place.

Switch

The switch statement is Java's multi way branch statement. It provides an easy way to dispatch execution to different parts of your code based on the value of an expression. As such, it often provides a better alternative than a large series of if-else-if statements. Here is the general form of a switch statement:

```
        switch (expression) {
                case value1:
                                / / statement sequence
                break;
                case value2:
                                /  / statement sequence
                break;
            .
            .
            .
                case valueN:
                                / /  statement sequence
                break;
                default:
                                /  / default statement sequence
}
```

The expression must be of type byte, short, int, or char, each of the values specified in the case statements must be of a type compatible with the expression. Each case value must be a unique literal (that is, it must be a constant, not a variable). Duplicate case values are not allowed.

Iteration Statements

Java's iteration statements are for while, and do while. These statements create what we commonly call loops. As you probably know, a loop repeatedly executes the same set of instructions until a termination condition is met. Java has a loop to fit any programming need.

While

The while lope is Java's most fundamental looping statement. It repeats a statement or block while its controlling expression is true. Here is its general form.

```
        while (condition)  {

                / /  body of loop
        }
```

The condition can be any boolean expression. The body of the loop will be executed as long as the conditional expression is true. When condition become false, control passes to the next line of code immediately following the loop. The curly braces are unnecessary if only a single statement is begin repeated.

Here is a while loop that counts down from 10, printing exactly ten lines of "tick".

```
/ / Demonstrate the while loop.
        class while{
                public static void main (String args [ ] ) {
                int n = 10;
                while (n > 0) {
                System.out.println ("tick"  + n);
                n - -;
                 }
        }
}
```

when you run this program, it will "tick" ten times:

```
        tick 10
        tick 9
        tick 8
        tick 7
        tick 6
        tick 5
        tick 4
        tick 3
        tick 2
        tick 1
```

Do-While

As you just saw, if the conditional expression controlling a **while** loop is initially false, then the body of the loop will not be executed at all. However, sometimes it is desirable to execute the body of a **while** loop at least once, even if the conditional expression is false to begin with. In other words, there are times when you would like to test the termination expression at the end of the loop rather than at the beginning. Fortunately, Java supplies a loop that does just that the **do-while**. The **do-while** loop always executes its body at least once, because its conditional expression is at the bottom of the loop. Its general form is

```
 do {
          / / body of loop
 } while (condition);
```

Each iteration of the **do-whole-loop** first executes the body of the loop and then evaluates the conditional expression. If this expression is true, the loop will repeat. Otherwise the loop terminates. As with all of Java's loop's condition must be a **Boolean** expression.

Here is a reworked version of "tick" program that demonstrates the **do-while-loop**. It generates the same output as before.

```
/ / Demostrate the do-while loop.
 class Dowhile {
    public static void main (String args [ ] ) {
         int n = 10;

         do {
                 System.out.println("tick" +n );
                 n - - ;
         }while (n>0);
    }
}
```

The loop in the preceding program, while technically correct, can be written more efficiently as follows.

```
    do  {
           System.out.println ("tick")  + n);
     } while ( - - n > 0);
```

In this example, the expression **(- -n >0)** combines the decrement of n and the test for zero into one expression. Here is how it works. First, the **- - n** statement executes, decrementing **n** and returning the new value of **n** . This value is then compared with zero. If it is greater than zero, the loop continues otherwise it terminates.

for

for loop is a powerful and versatile construct. Here is the general form of the for statement.

```
for (initialization; condition; iteration){
```

```
            / / body
}
```

If only one statement is being repeated, there is no need for the curly braces. The **for** loop operates as follows. When the loop first starts, the initialization portion of the loop is executed. Generally, this is an expression that sets the value of the loop control variable, which acts as a counter that controls the loop. It is important to understand that the initialization expression is only executed once.

Nest, condition is evaluated. This must be a boolean expression. It usually tests the loop control variable against a target value. If this expression is true, then the body of the loop is executed. If it is flaps, the loop terminates. Next the integration portion of the loop is executed. The loop then iterates, first evaluating the conditional expression, then executing the body of the loop, and then executing the iteration expression with each pass. This process repeats until the controlling expression is false.

Here is a version of the "**tick**" program that uses a **for** loop.

```
/ / demonstrate the for loop.
class ForTick   {
        public static void main (String atgs [ ] 0 {
                int n ;
                for (n=10; n>0; n - - )
                System.out.println ("tick" + n)
        }
}
```

11.5 Arrays

An array is a group of like-types variables that are referred to by a common name. Arrays of any type can be created and may have one or more dimensions. A specific element in an array is accessed by its index. Arrays offer a convenient means of grouping related information.

One Dimensional Arrays

A one dimensional array is, essentially, a list of like typed variables. To create an array, you first must create an array variable of the desired type. The general form of a one dimensional array declaration is

```
type var-name [ ];
```

Here, *type* declares the base type of the array. The base type determines the data type of each element that comprises the array. Thus, the base type for the array determines what type of data the array will hold. For example, the following declares an array named **month days** with the type "array of int".

int month_days [ ] ;

Although this declaration established the fact that **month_days** is an array variable, no array actually exists. In fact, the value of month days is set to **null**, which represents an array with no value. To link **month_days** with an actual, physical array of integers, you must allocate one using new and assign it to **month_days**. New is a special operator that allocates memory.

The general form of new as it applies to one-dimensional arrays appears as follows:

```
        array –var =new type [size];
```

Here is a program that creates an array of the numbr of days in each month.

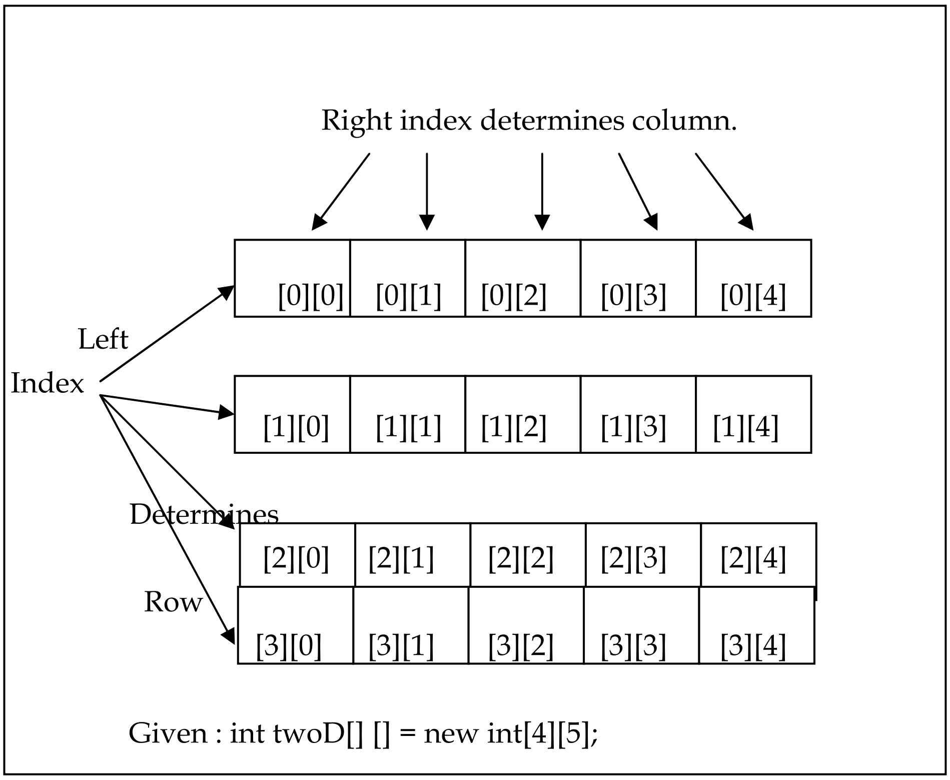
```
public static void main ( String args [ ]) {
        int month_days [ ] ;
        month_days = new int [ 5];
        month_days [0] = 31;
        month_days [1] = 28;
        month_days [2] = 31;
        month_days [3] = 30;
        month_days [4] = 31;
        month_days [5] = 30;
        System.out.println("April has"+month_day(3)+"days.");
        }
}
```

Multidimensional Arrays

In Java multidimensional arrays are actually arrays of arrays. These as you might expect, look and act like regular multidimensional arrays. However, as you will see, there are a couple of subtle difference. To declare a multidimensional array variable, specify each additional index using another set of square brackets. For example, the following declares a two dimensional array variable called **twoD.**

int twoD [ ] [ ] = new int [4] [5];

This allocates a 4 by 5 array and assigns it to **twoD.** Internally this matrix is implemented as an array of arrays of **int**. Conceptually, this array will look like the one shown in figure



Given : int twoD[] [] = new int[4][5];

Two-dimensional array

The following program numbers each element in the array from left to right, top to bottom, and then displays these values:

```
/ / Demonstrate a two dimensional array.
        class TwoDArray {
                public static void main (String args [ ] )    {
                        int two D [ ] = new int [4] [5] ;
                        int i,j, k = 0;
                        for (i=0; i<4; i+ +)
                                for (j=0; j<5; j + +) {
                                        twoD[i] [j] = K;
                                        K + +;
                        }

                        for (i=0; i<4; i+ +) {
                                for (j=0; j<5; j ++ )
                                system.out.print (twoD[i] [j] + "  ") ;
                                system.out.println ( );
                }
            }
    }
```

This program generates the following output.

```
0 1 2 3 4
5 6 7 8 9
10 11 12  13 14
15 16 17 18 19
```

When you allocate memory for a multidimensional array, you need only specify the memory for first (leftmost) dimension. You can allocate the remaining dimensions separately. For example, this following code allocates memory for the first dimension of **twoD** when it is declared. It allocates the second dimension manually.

```
int twoD [ ]  [ ] = new int [4] [ ];
twoD[0] = new int [5];
twoD[1] = new int [5];
twoD[2] = new int [5];
twoD[3] = new int [5];
```

Alternative Array Declaration Syntax

There is a second form that may be used to declare an array.

```
type [ ] var-name;
```

Here, the square brackets follow the type specifier, and not the name of the array variable. For example, the following two declarations are equivalent.

```
   int al[ ] = new int [3];
   int [ ] a2 = new int [ 3];
```

The following declarations are also equivalent;

```
   char twod1[ ] [ ] = new char [3] [4] ;
   char [ ] [ ] twod2 = new char [3] [4];
```

This alternative declaration form is included mostly as a convenience.

11.6 Using Command Line Arguments

Sometimes you will want to pass information into a program when you run it. This is accomplished by passing command line arguments to **main ( ).** A command-line argument is the information that directly follows the program's name on the command line when it is executed. To access the command-line arguments inside a Java program is quite easy, they are stored as **strings** in the **string** array passed to **main ( )** . For example, the following program displays all of the command line arguments that it is called with:

```
/ / Display all command-line arguments.
        class CommandLine   {
          public static void main (String args [ ] ) {
             for (int  i=o; i<args.length; i + +)
                 System.out.println("args[" + i  +" ]:" +
                 args [I]) ;

        }
}
```

Try executing this program, as shown here

```
   java CommandLine this is a test 100 –1
```

When you do, you will see the following output.

```
args[0] : this
args[1] : is
args[2] : a
args [3] : test
args[4] : 100
args [5] : -1
```

11.7 Short Summary

- Java defines eight simple (or elemental) types of data: **byte, short int, long, char, floats, Double,** and **Boolean**

- A variable is defined by the combination of an identifier, a type, and an optional initializer.

- An array is a group of like-types variables that are referred to by a common name.

- All command-line arguments are passing as strings.

11.8 Brain Storm

1. What are the various data types involved in Java Explain Briefly.
2. How variables can be declared?
3. Explain all the types of Operators?
4. With an example, Explain One Dimensional & Multi Dimensional
5. What is the use of Command line argument?

Lecture - 12

Classes & Objects

Objectives

In this lecture you will learn the following

- Understanding the concept of object & classes
- Assigning object reference variables
- Knowing about the 'this' keyword

Lecture - 12

12.1 Snap Shot

In this lecture we introduce the concept of classes and objects. It attributes and methods represent a class.

12.2 Class fundamentals

The classes created in the primarily exist simply to encapsulate the **main()** method, which has been used to demonstrate the basics of the Java syntax. Perhaps the most important thing to understand about a class is that it defines a new data type. Once defined, this new type can be used to create objects of that type. Thus, a class is a *template* for an object, and an object is an *instance* of a class. Because an object is an instance of a class, you will often see the two words *object* and *instance* used interchangeably.

The General form of a class

A class is declared by use of the **class** keyword. The classes that have been used up to this point are actually very limited examples of its complete form. Classes can get much more complex. The general form of a class definition is shown here:

```
class classname {
type instance-variable1;
type instance-variable2;
//…
type instance-variableN;

type methodname1(parameter-list) {
        // body of method
}
type methodname2(parameter-list) {
        //body of method
}
        //…
type methodnameN(parameter-list) {
        //Body of method
}
}
```

The data, or variables, defined within a class are instance variables. The code is contained within methods. Collectively, the methods, and variables defined within a class are called *members* of the class. In most classes, the instance variables are acted upon and accessed by the methods defined for that class. In most classes, the instance variables are acted upon and accessed by the methods defined for that class. Thus, it is the methods that determine how a class' data can be used.

Variables defined within a class instance variables because each instance of the class contains its own copy of these variables. Thus, the data for one object is separate and unique from the data for another. We will come back to this point shortly, but it is an important concept to learn early.

All methods have the same general form as main(), which we have been using thus far. However, most methods will not be specified as static or public.

A simple class

Let's begin our study of the class with a simple example. Here is a class called **box** that defines three instance variables: **width, height, and depth.** Currently, **Box** does not contain any methods

```
Class box {
        double width, heigth, depth;
}
```

12.3 Declaring Objects

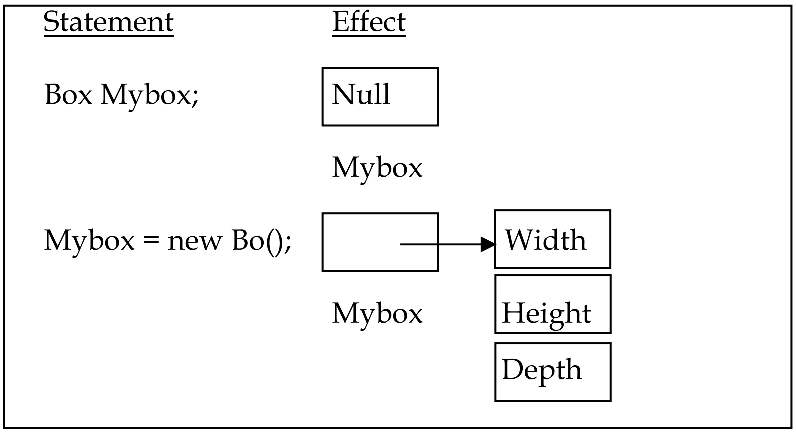
As just explained, when you create a class, you are creating a new data type. You can use this type to declare objects of that type. However, obtaining objects of a class is a two step process. First you must declare a variable of the class type. This variable does not define an object. Instead, it is simply a variable that can refer to an object. Second, you must acquire an actual, physical copy of the object and assign it to that variable. You can do this using the new operator. The **new** operator dynamically allocates (that is allocates at run time) memory for an objects and return a reference to it. This reference is more or less, the address in memory of the object allocated by **new**. This reference is then stored in the variable. Thus in Java all class objects must be dynamically allocated. Let's look at the details of this procedure.

```
        Box mybox = new box ( ) ;
```

This statement combines the two steps just described. It can be rewritten like this to show each step more clearly;

```
Box mybox ;  / / declare reference to object
Mybox = new Box  } ( ) ; allocate a Box object
```

The effect of there two lines of code is depicted in the following figure



Declaring an object of type Box

A closer look at new

As just explained the new operator dynamically allocates memory for an object. It has this general form:

class-var = new classname();

Here, **class-var** is a variable of the class type being created. The **classname** is the name of the class that is being instantiated. The class name followed by parentheses specifies the constructor for the class. A constructor defines what occurs when an object of a class is created. Constructors are an important part of all classes and have many significant attributes. Most real world classes explicitly constructor is specified, then Java will automatically supply a default constructor. This is the case with box. For now , we will use the default constructor. Soon, you will see how to define your own constructor.

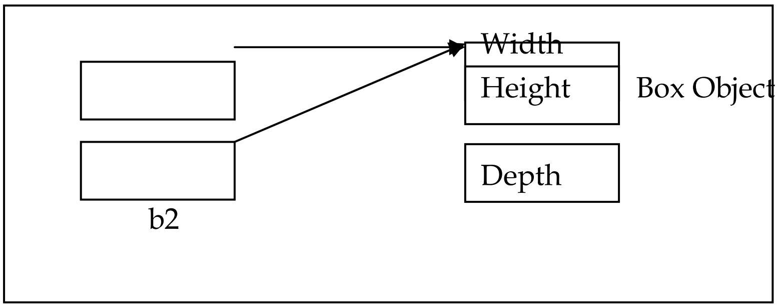
Assigning object reference variables

Object reference variables act differently than you might expect when an assignment takes place./ for example, what do you think the following fragment does?

```
Box b1 = new Box();
Box b2 = b1;
```

You might think that b2 is being assigned a reference to a copy of the object referred to by b1. That is you might think that b1 and b2 refer to separate and distinct objects. However, this would be wrong. Instead, after this fragment executes, b1 and b2 will both refer to the same object. The assignment of b1 to b2 did not allocate any memory or copy any part of the original object. It simply makes b2 refer to the same object as does b1. Thus any changes made to the object through b2 will affect the object to which b1 is referring, since they are the same object.

This situation is depicted here:



Although b1 and b2 both refer to the same object, they are not linked in any other way. For example, a subsequent assignment to b1 will simply unhook b1 from the original object without affecting the object or affecting b2 for example

```
Box b1 = new Box();
Box b2 = b1;
// …
b1=null;
```

here, b1 has been set to null, but b2 still points to the original object.

Introducing methods

Classes usually consist of two things: instance variables and methods. The topic of methods is a large one because Jana gives them so much power and flexibility. In fact, much of the next chapter is developed to methods. However, there are some fundamentals that you need to learn now so that you can begin to add methods to your classes.

This is the general form of a method

```
        Type name(parameter-list){
//body of method
}
```

12.4 Constructors

A constructor initializes an object immediately upon creation. It has the same name as the class in which it resides and is syntactically similar to a method. Once defined, the constructor is automatically called immediately after the object is created, before the new operator completes. Constructors look a little strange because they have no return type, not even void. This is because the implicit return type of a class' constructor is the class type itself. It is the constructors' job to initialize the internal state of an object so that the code creating an instance will have a fully initialized, usable object immediately.

Before moving on, let's reexamine the new operator. As you know, when you allocate an object, you use the following general form:

```
Class-var = new classname();
```

Now you can understand why the parentheses are needed after the class name. What is actually happening is that the constructor for the class is being called thus, in the line

```
Box mybox1= new Box();
```

New Box() is calling the Box() constructor. When you do not explicitly define a constructor for a class, then Java creates a default constructor for the class. This is why the preceding line of code worked in earlier versions of box that did not define a constructor.

12.5 Parameterized Constructors

While the box() constructor in the preceding example does initializes a box object, it is not very useful- all boxes have the same dimensions. What is needed is a way to construct box objects of various dimensions. The easy solution is to add parameters to the constructor. As you can probably guess, this makes them much more useful. For example, the following version of box defines a parameterized constructor, which sets the dimensions of a box as specified by those parameters. Pay special attention to how box objects are created.

```
class box {
double width, height, depth;
box(double w, double h, double d)
{
 width = w;
height = h;
depth = d;
}
```

12.6 The 'this' keyword

Sometimes a method will need to refer to the object that invoked it. To allow this, java defines the this keyword. This can be used inside any method to refer to the current object. that is always a reference to the object on which the method was invoked. You can use this anywhere a reference to an object of the current class' type is permitted.

To better understand what this refers to, consider the following version of box():

```
// a redundant use of this.
Box(double w, double h, double d) {
This.width;
This.height;
This.depth;
}
```

This version of box() operates exactly like earlier version. The use of this is redundant, but perfectly correct. Inside box(), this will always refer to the invoking object. While it is redundant in this case, this is useful in other contexts.

12.7 Short Summary

- A **class** is a template for an object, and an **object** is an instance of a class.
- The new operator dynamically allocates (that is allocates at run time) memory for an objects and return a reference to it.
- A constructor initializes an object immediately upon creation.

12.8 Brain Storm

1. What is Class, Explain with one Example?
2. What is object?
3. How to declare an object, Explain with one Example?
4. What is the difference between Class and Objects?
5. What is the use of 'this' keyword?

Lecture - 13

Inheritance in Java

Objectives

In this lecture you will learn the following

- ✈ About Inheritance
- ✈ Super and Final Key word

Lecture - 13

13. 1 Snap shot

In this lecture you will learn about Basics of Inheritance, 'Super' keyword and about 'Final' keyword.

13.2 Inheritance Basics

To inherit a class, you simply incorporate the definition of one class into another by using the extends keyword. To see how, let's begin with a short example. The following program creates a superclass called A and a subclass called B. Notice how the keyword extends is used to create a subclass of A

```
// A simple example of inheritance

// Create a superclass
class A {
          int i, j ;

     void showij( ) {
                    System.out.println("i and j : " + i  + " " + j ) ;
          }
}

// Create a subclass by extending class A
class B extends A {
          int k;

          void showk( ) {
                    System.out.println("i+j+k: " + ( i+j+k));
          }
}

class SimpleInheritance {
          public static void main(String args[ ] ) {
                    A superOb = new A( );
                    B subOb = new B ( );

     // The supeclass may be used by itself
     superOb.i = 10;
     superOb.j = 20;
     System.out println("Contents of superOb: ");
     superOb.showij( );
```

```
    System.out.println( );

    /* The subclass has access to all public members of its superclass. */
    subOb.i = 7;
    subOb.j = 8;
    subOb.k = 9;
    System.out.println("Contents of subOb: ");
    subOb.showij( );
    subOb.showk( );
    System.out.println( );

    System.out.println("Sum of i,j and k in subOb:");
    SubOb.sum ( );
        }
}
```

The output from this program is shown here

```
    Contents of superOb:
    i and j : 10 20

    Contents of subOb;
    i and j: 7 8
    Sum of I,j and k in subOb;
    i+j+k: 24
```

As you can see, the subclass B includes all of the members of its superclass, A. This is why subOb can access i an j and call showij( ). Also, inside sum ( ),i and j call showij( ). Also, inside sum ( ),i and j can be referred to directly, as if they were part of B.

Even though A is a superclass for B, it is also a completely independent, stand-alone class. Being a superclass for a subclass does not mean that the superclass cannot be used by itself. Further, subclass can be a superclass for another subclass.
The general form of a class declaration that inherits a superclass is shown here:

```
class subclass-name extends superclass-name{
//body of class
}
```

You can only specify one superclass for any subclass that you create. Java does not support the inheritance of multiple superclasses into a single subclass. (This differs from C++, in which you can inherit multiple base class.) . Create a hierarchy of inheritance in which a subclass becomes a superclass of another subclass. However, no class can be a superclass of itself.

13.3 Using 'super' keyword

In the preceding examples, classes derived from **Box** were not implemented as efficiently or as robustly as they could have been. For example, the constructor for **BoxWeight** explicitly initializes the **width, height and depth** fields of **Box( ).** Not only does this duplicate code found in its superclass, which is inefficient, but it implies that a subclass must be granted access to these members. However, there will be times when you will want to create a superclass that keeps the details of its implementation to itself (that is, that keeps its data members private). In this case, there would be no encapsulation is a primary attribute of OOP, it is not surprising that Java provides a solution to this problem. Whenever a subclass needs to refer to its immediate supercalss, it can do so by use of the keyword **super.**

super has two general forms. The first calls the superclass' constructor. The second is used to access a member of the superclass that has been hidden by a member of a subclass. Each use is examined here.

Using super to Call Superclass Constructors

A subclass can call a constructor method defined by its superclass by use of the following form of super

```
super(parameter – list);
```

Here *parameter-list* specifies any parameters needed by the constructor in the superclass. *super( )* must always be the first statement executed inside a subclass' constructor.

To see how *super ( )* is used, consider this improved version of the **BoxWeight( )** class:

```
//BoxWeight now uses super to initialize its Box attributes.
class BoxWeight extends Box {
        double weight; // weight of box

   // initialize width, height and depth using super ( )
   BoxWeight (double w, double h, double d, double m) {
                super (w,h,d); // call superclass constructor
                weight = m;
   }
}
```

Here **BoxWeight( )** calls **super ( )** with the parameters **w,h** and **d**. This causes the **Box( )** constructor to be called, which initializes **width, height** and **depth** using these values. **BoxWeight** no longer initializes these values itself. It only needs to initialize the value unique to it: **weight**. This leaves **Box** free to make these values **private** if desired.

In the preceding example, **super( )** was called with three arguments. Since constructors can be overloaded, **super( )** can be called using any form defined by the superclass. The constructor executed will be the one that matches the arguments. for example here is a complete implementation of **BoxWeight** that provides constructors for the various ways that a box can be constructed. In each case, super( ) is called using the appropriate arguments. For example, here is a complete implementation of **BoxWeight** that provides construction for the various ways that a box can be constructed. In each case, **super( )** is called using the appropriate arguments. Notice that width, height and depth have been made private within Box.

```
     // A complete implementation of BoxWeight.
     class Box {
          private double width;
  private double height;
          private double depth;

     // construct clone of an object
     Box (Box Ob) { // pass object to constructor
          width = Ob.width;
          height = Ob.height;
          depth = Ob.depth;
     }

     // constructor used when all dimensions specified
     Box(double w, double h, double d) {
          width = w;
          height = h;
          depth = d;
     }

     //constructor used when no dimensions specified
     Box( ) {
          width =-1; //use –1 to indicate
          height =-1; // an uninitialized
          depth = -1; //box
```

```
        }

        // constructor used when cube is created
        Box(double len) {
                width = height = depth = len;
        }

        //compute and return volume
        double volume( )  {
                return width *height * depth;
        }
}

// BoxWeight now full implements all  constructors
class BoxWeight extends Box {
        double weight; // weight of box

        // construct clone of an object
        BoxWeight (BoxWeight Ob) { // pass object to constructor
                super(Ob);
                weight = Ob.weight;
        }

        // constructor when all parameters are specified
        BoxWeight(double w, double h, double d, double m) {
                super(w,h,d); // call superclass constructor
                weight = m;
        }

        // default constructor
        BoxWeight ( ) {
                super( );
                weight = -1;
        }

        //constructor used when cube is created
        BoxWeight (double len, double m) {
                super(len);
                weight = m;
        }
}
```

```
class DemoSuper {
        public static void main(String args[ ]) {
                BoxWeight mybox1 = new BoxWeight(10, 20, 15, 34.3);
                BoxWeight mybox2 = new BoxWeight(2, 3, 4, 0.076);
                BoxWeight mybox3 = new BoxWeight ( ); // default
                BoxWeight mycube = new BoxWeight (3, 2);
                BoxWeight myclone = new BoxWeight(mybox1);
                double vol;

                vol = mybox1.volume( ) ;
                system.out.println("Volume of mybox1 is "+ vol);
                system.out.println("Weight of mybox1 is " + mybox1.weight);
                system.out.println( ) ;

                vol = mybox2.volume( ) ;
                system.out.println("Volume of mybox2 is "+ vol);
                system.out.println("Weight of mybox2 is " + mybox2.weight);
                system.out.println( ) ;

                vol = mybox3.volume( ) ;
                system.out.println("Volume of mybox3 is "+ vol);
                system.out.println("Weight of mybox3 is " + mybox3.weight);
                system.out.println( ) ;

                vol = myclone.volume( ) ;
                system.out.println("Volume of myclone is "+ vol);
                system.out.println("Weight of myclone is " + myclone.weight);
                system.out.println( ) ;

                vol = mycube.volume( ) ;
                system.out.println("Volume of mycube is "+ vol);
                system.out.println("Weight of mycube is " + mycube.weight);
                system.out.println( ) ;
        }
}
```

This program generates the following output:

```
Volume of mybox1 is 3000.0
Weight of mybox1 is 34.3
```

```
Volume of mybox2 is 24.0
Weight of mybox2 is 0.076

Volume of mybox3 is –1.0
Weight of mybox3 is –1.0

Volume of myclone is  3000.0
Weight of myclone is 34.3

Volume of mycube is 27.0
Weight of mybox1 is 2.0
```

Pay special attention to this constructor in BoxWeight( ) ;

```
// construct clone of an object
BoxWeight(BoxWeight Ob) { // pass object to constructor
      super(Ob);
      weight = Ob.weight;
}
```

Notice that **super( )** is called with an object of type **BoxWeight** – not of type Box. This still invokes the constructor **Box**(Box ob). As mentioned earlier, a superclass variable can be used to reference any object derived form that class. Thus, we are able to pass a **BoxWeight** object to the Box constructor. Of course, Box only a knowledge of its own members.

Let's review the key concepts behind **super( )**. When a subclass calls s**uper( ),** it is calling the constructor of its immediate superclass. Thus, **super( )** always refers to the superclass immediately above the calling class. This is true even in a multileveled hierarchy. Also, **super( )** must always be the first statement executed inside a subclass constructor.

A Second Use for super

The second form of super acts somewhat like this, except that it always refers to the superclass of the subclass in which it is used. This usage has the following general form:

```
super.member
```

Here member can be either a method or an instance variable.

This second form of super is most applicable to situations in which member names of a subclass hide members by the same name in the superclass. Consider this simple class hierarchy:

```
// Using super to overcome name hiding.
class A {
        int i;
}

//Create a subclass by extending class A.
class B extends A {
        int i; // this i hides the i in A

        B(int a, int b) {
                super.i = a; // i in A
                i = b; // i in B
        }

        void show( ) {
                System .out.println("i in superclass: " = super.i);

                System.out.println("i in subclass: " + i);
        }
}

class UseSuper {
        public static void  main(String args[ ] ) {
                B subOb =  new B(1,2);

                subOb.show( );
        }
}
```

The program displays the following:

```
i in superclass: 1
i in subclass: 2
```

Although the instance variable i in B hides the i in A, super allows access to the i defined in the superclass. **Super** can also be used to call methods that are hidden by a **subclass**.

13.4 Using 'Final' keyword

A variable can be declared as final. Doing so prevents its contents from being modified. This means that you must initialize a final variable when its is declared. For example

```
final int FILE_NEW = 1;
final int FILE_OPEN = 2;
final int FILE_SAVE = 3;
final int FILE_SAVEAS = 4;
final int FILE_QUIT = 5;
```

Subsequent parts of your program can now use FILE_OPEN etc as if they were constants, without fear that a value has been changed.

It is a common coding convention to choose all uppercase identifiers for final variables. Variables declared as final do not occupy memory on a per-instance basis. Thus a final variable is essentially a constant.

The keyword final can also be applied to methods, but its meaning is substantially different than when it is applied to variables. This second usage of final is described in the next chapter, when inheritance is described.

Using final with Inheritance

The keyword final has three uses. First, it can be used to create the equivalent of a named constant. The other two uses of final apply to inheritance.

Using final to Prevent Overriding

While method overriding is one of Java's most powerful features, there will be times when you will want to prevent it from occurring. To disallow a method from being overridden, specify final as a modifier at the start of its declaration. Methods declared as final cannot be overridden. The following fragment illustrates final:

```
class A{
        final void meth( ) {
                System.out.println("This is a final method.");
        }
}

class B extends A {
        void meth( ) { // ERROR! Can't override.
```

```
                    System.out.println((“Illegal!”);
          }
}
```

Because meth( ) is declared as final, it cannot be overridden in B. If you attempt to do so, a compile-time error will result.

Methods declared as final can sometimes provide a performance enhancement. The compiler is free to inline calls to them because it “knows” they will not be overridden by a subclass. When a small final function is called often the Java compiler can copy the byte code for the subroutine directly inline with the compiled code of the calling method, thus eliminating the costly overhead associated with a method call. Inlining is only an option with final methods. Normally, Java resolves calls to methods dynamically at run time. This is called late binding. However since final methods cannot be overridden a call to one can be resolved at compile time. This is called entry binding.

Using final to Prevent Inheritance

Sometimes you will want to prevent a class from being inherited. To do this, procedure the class declaration with final. Declaring a class a final implicitly declares all of its methods a final, too. As you might expect, it is illegal to declare a class as both abstract and final since an abstract class is incomplete by itself and relies upon its subclasses to provide complete implementations.

Here is an example of a final class:

```
final class A {
          // …
}

// The following class is illegal.
class B extends A { // ERROR! Can’t subclass A
          // …
}
```

As the comments imply, it is illegal for B to inherit A since A is declared as final.

13.5 Short Summary

- Java does not support the inheritance of multiple super classes into a single subclass.

- Whenever a subclass needs to refer to its immediate supercalss, it can do so by use of the keyword super.

- A super class variable can be used to reference any object derived form that class.

- Sometimes you will want to prevent a class from being inherited. To do this, procedure the class declaration with final

13.6 Brain Storm

1. What is the use of Inheritance in Java ?

2. What is the need of ‘Super’ keyword?

3. Explain briefly about the ‘Final’ keyword.

Lecture - 14

Polymorphism in Java

Objectives

In this lecture you will learn the following

- Know about Polymorphism in Java
- Dynamic Method Dispatch
- Abstract Classes

Lecture - 14

14.1 Snap shot

This lectures covers the Polymorphism in Java , need of Dynamic method dispatch and Using Abstract Classes.

14.2 Polymorphism in Java

Polymorphism (from the Greek , meaning "many forms ") is a feature that allows one interface to be used for a general class of actions. The specific action is determined by the exact nature of the situation. Consider a stack (which is a last-in, first-out list). You might have a program that requires three types of stacks . One stack is used for integer values, one for floating-point values, and one for characters. The algorithm that implements each stack is the same, even though the data being stored differs. In a non-object-oriented language, you would be required to create three different sets of stack routines, with each set using different names. However , because of polymorphism, in Java you can specify a general set of stack routines that all share the same names.

More generally, the concept of polymorphism is often expressed by the phrase " one interface , multiple methods." This means that it is possible to design a generic interface to a group of related activities. This helps reduce complexity by allowing the same interface to be used to specify a general class of action. It is the compiler's job to select the specific action (that is, method) as it applies to each situation. You, the programmer , do not need to make this selection manually . You need only remember and utilize the general interface.

Extending the dog analogy, a dog's sense of smell is polymorphic. If the dog smells a cat, it will bark and run after it. If the dog smells its food, it will salivate and run to its bowl. The same sense if smell is at work in both situations. The difference is what is being smelled, that is, the type of data being operated upon by the dog's nose! This same general concept can be implemented in Java as it applies to methods within a Java Program.

14.3 Dynamic method dispatch

Indeed, if there were nothing more to method overriding than a name space convention, then it would be, at best, an interesting curiosity, but of little real value. However, this is not the case. Method overriding forms the basis for one of Java's most powerful concepts: dynamic method dispatch.

Dynamic method dispatch is the mechanism by which a call to an overridden function is resolved at run time, rather than compile time. Dynamic method dispatch is important because this is how java implements run-time polymorphism.

Let's begin by restating an important principle: a super class reference variable can refer to a subclass object. Java uses this fact to resolve calls to overridden methods at run time. Here is how. When an overridden method is called through a superclass reference, Java determines which version of that method to execute based upon the type of the object being referred to at the time the call occurs. Thus, this determination is made at run time. When different types of objects are referred to, different versions of an overridden method will be called. In other words, it is the type of the object being referred to that determines which version of an overridden method will be executed. Therefore, it a superclass contains a method that is overridden by a subclass, then when different types of objects are referred to through a superclass reference variable, different versions of the method are executed.

Here is an example that illustrates dynamic method dispatch:

```
// Dynamic method dispatch
class A {
 void callme() {
System.out.println("Inside A's callme method");
}
}
class B extends A {
// override callme()
void callme()
{
System.out.println("Inside B's  callme method");
}
}
class C extends A {
// Override callme()
System.out.println("Inside C's callme method");
}
}
class Dispatch {
public static void main(String args[]){
A a = new  A();   // objects of type A
B b = new B();    // Objects of type B
C c = new C();   //  objects of type C
A r; //Obtain a reference of type A
r=a;
r.callme();
r=b;
r.callme();
r=c;
r.callme();
}
}
```

The output from the program is shown here:

```
Inside A's callme method
Inside B's callme method
Inside C's Callme method
```

This program creates one superclass called A and two subclasses of it, called B and C. Subclasses B and C override callme() declared in A. Inside the main() method, objects of type A, B,and C are declared. Also, a reference of type A, called r, is declared. The program then assigns a reference to each type of objects to r and uses that reference to invoke callme() As the output shows the version of callme() executed is determined by the type of object being referred to at the time of the call. Had it been determined by the type of the reference variable, r, you would see three calls to A's callme() method.

14.4 Why Overridden Methods?

As stated earlier, overridden method allow Java to support run-time polymorphism. Polymorphism is essential to object oriented programming for one reason: it allows a general class to specify methods that will be common to all of its derivatives, while allowing subclasses to define the specific implementation of some or all of those methods. Overridden methods are another way that Java implements the "one interface, multiple methods" aspect of polymorphism.

Part of the key to successfully applying polymorphism is understanding that the super classes and subclasses form a hierarchy which moves from lesser to greater specialization. Used correctly, the superclass provides all elements that a subclass can own. This allows the subclass the flexibility to define its won methods, yet still enforces consistent interface. Thus, by combining inheritance with overridden methods, a superclass can define the general form of the methods that will be used by all of its subclasses..

Dynamic, run time polymorphism is one of the most powerful mechanisms that object-oriented design brings to bear on code reuse and robustness. The ability of existing code libraries to call methods on instances of new classes without recompiling while maintaining a clean abstract interface is a profoundly powerful tool.

14.5 Applying Method Overriding

Let's look at a more practical example that uses methods overriding. This following program creates a superclass called **Figure** that stores the dimensions of various two-dimensional objects. It also defines a method called **area** ( ) that computes the area of an object. The program derives two subclasses from **Figure**. The first is rectangle and the second it **Triangle.** Each of these subclasses overrides **area** ( ) so that it returns the area of a rectangle a triangle, respectively.

```
// using run time polymorphism.
        Class figure {
        Double dim 1 ;
        Double dim2;
Figure (double a, double b ) {
        Dim1 = a;
        Dim2 = b;
        }
        double area  ( ) {
         System.out.println ("Area for figure is undefined,");
        return 0;
            }
        }
        class Rectangle extends Figure  {
        rectangle (double a , double b ) {
        super ( a, b) ;
        }

/ / override area for rectangle
        double area  ( ) {
        System.out.println( "Inside Area for Rectangle.");
        return dim1 * dim2;
          }
        }

        class triangle extends figure {
        Triangle (double a , double b ) {
        super (a,   b);
```

```
/ / override area for right triangle
          double area ( ) {
          System.out.println("inside area for triangle.");
          Return dim1*dim2/2;
            }
          }
          class Find Areas {
          public static void main (string args [ ]) {
          Figure f = new Figure (10,10);
          rectangle r = new rectangle ( 9, 5 );
          triangle T = new Triangle (10, 8 );
          Figure figref;

          figref = r;
          System.out.println ("Area is" + figref.area ( ) );

          figref = t;
          System.out.println.("Area is" + figref.area ( ) );

          figref = f;
          System.out.println ("Area is" + figref.area  ( )  );
```

The output from the program is shown here:

```
Inside Area for Rectangle.
Area is 45
Inside Area for Triangle.
Are is 40
Area for Figure is undefined.
Area is 0
```

Through the dual mechanisms of inheritance and run time polymorphism, it is possible to define one consistent interface that is used by sever al different, yet related types of objects. In this case, if an object is derived from figure, then its area can be obtained by called area ( ). The interface to this operation is the same no matter what type of figure is being used.

14.6 Using Abstract Classes

There are situation in which you will want to define a superclass that declares the structure of a given abstraction without providing a complete implementation of

every method. That is sometimes you will want to create a superclass that only defines a generalized form that will be shared by all of its subclasses, leaving it to each subclass to fill in the details. Such a class determines the nature of the methods that the subclasses must implement. One way this situation can occur is when a superclass is unable to create a meaningful implementation for a method. This is the case with the class figure used in the preceding example. The definition of area ( ) is simply a placeholder. It will not compute and display the area of any type of object.
As you will see as you create your own class libraries, it is not uncommon for a method to have no meaningful definition in the context of its superclass. You can handle this situation two ways. One way, as shown in the previous example, is to simply have it report a warning message. While this approach can be useful in certain situations such as debugging it is not usually appropriate. You may have methods which must be overridden by the subclass in order for the subclass to have any meaning. Consider the class Triangle. It has no meaning if **area ( )** is not defined. In this case, you want some way to ensure that a subclass does, indeed, override all necessary methods. Java's solution to this problem is the abstract method.

You can require that certain methods be overridden by subclasses by specifying the **abstract** type modifier. These methods are sometimes referred to a *subclasses responsibility* because they have no implementation specified in the superclasses . Thus, a subclass must override them-it cannot simply use the version defined in the superclass. To declare an abstract methods, use this general form.

abstract type name (parameter-list);
As you can see, no method body is present.
Any class that contains one or more abstract methods must also be declared abstract. To declare a class abstract, you simply use the **abstract** keyword in front of the **class** keyword at the beginning of the class declaration. There can be no objects of an abstract class. That is, an abstract class cannot be directly instantiated with the new operator. Such objects would be useless, because an abstract class is not fully defined. Also, you cannot declare abstract constructors, or abstract static methods. Any subclass of an abstract class must either implement all of the abstract methods in the superclass, or be itself declared **abstract**.

Here is simple example of class with an abstract method, followed by a class which implements that method;

```
/ / A simple demonstration of abstract.
        abstract class A {
        abstract void callme ( ) ;
 / / concrete methods are still allowed in abstract classes
        void callmetoo ( ) {
          System.out.println (“This is a concrete method.”);
         }
        }
        class B extends A {
        void callme ( ) {
          system.out.println (“B’ s implementation of callme.”);
         }
        }
        class AbstractDemo {
          public static void main (string args [ ] ) {
        B  b = new  B ( ) ;
        b. callme ( );
        b.callmetoo ( );
          }
        }
```

Notice that no objects of class **A** are declared in the program. As mentioned, it is not possible to instantiated an abstract class. One other point, class **A** implements a concrete method called **callmetoo ( ).** This is perfectly acceptable. Abstract classes can include as much implementation as they see fit.

Although abstract classes cannot be used to instantiate objects, they can be used to create object references, because Java’s approach to run_time polymorphism is implemented through the use of superclass references. Thus, it must be possible to create a reference to an abstract class so that it can be used to point to a subclass object. You will see this feature put to use in the next example..

Using an abstract class, you can improve the **Figure** class shown earlier. Since there is no meaningful concept of area for an underlined two-dimensional **Figure,** the following version of the program devalues **area ( )** as abstract inside **Figure** . This, of course, means that all classes derived from **Figure** must override **area** ( ).

```
/ / using abstract methods an classes.
        abstract class figure {
        double dim1;
        double dim2;

Figure (double a, double b) {
dim = a;
dim2 = b;
}
```

```
 / / area is now an abstract method
        abstract double area ( ) ;
        }
        class Rectangle extends Figure {
        Rectangle ( double a, double b) {
        super (a,  b ) ;

        }

/ / overide area for rectangle
        double are ( ) {
           system.out.println ( “Inside Area for Rectangle.”);
        return dim1 * dim 2;
          }
        }

        class Traingle extends Figure  {
        Traingle  ( double a, double b) {
          super (a,    b );
        }

/ / override area for right triangle
        double area ( ) {
        System.out.println( “Inside Area for Triangle.”);.
        retrun.dim1 * dim2 / 2 ;
        }
        }

        class AbstractAreas {
        public static void main (String args [ ] ) {

/ / Figure f = new Figure ( 10,10 ); / / illegal now
        Rectangle r = new Rectangle ( 9,5 );
        Triangle t = new Triangle ( 10, 8 );

        Figure figref;  / /  this is ok, no object is created
        figref = r;
        System.out.println (“Area is”  + figref.area ( ) );

        figref = t;
        System.out.println (“Area is   + figref.area ( ) );
          }
```

```
}
```

As the comment inside **main**( ) indicates, it is no longer possible to declare objects of type figure, since it is now abstract. And, all subclasses of **Figure** must override **area ( ).** To prove this to yourself, try creating a subclass that does not override **area ( )** . You will receive a compile- time error..

Although it is not possible to create an object of type **Figure**, you can create a reference variable for type **Figure.** The variable **Figref** is declared as a reference to Figure, which means that it can be used to refer to an object of any class derived from **Figure**. As explained, it is through superclass reference variables that overridden methods are resolved at run time.

14.7 Short Summary

- More generally, the concept of polymorphism is often expressed by the phrase " one interface , multiple methods."
- Dynamic method dispatch is the mechanism by which a call to an overridden function is resolved at run time, rather than compile time.
- There are situation in which you will want to define a superclass that declares the structure of a given abstraction without providing a complete implementation of every method.

14.8 Brain Storm

1. Define the term Polymorphism in Java.
2. What is meant by Dynamic Method Dispatch?
3. How the Abstraction Classes used in Java?

Lecture - 15

Interface in Java Inner Classes

Objectives

In this lecture you will learn the following

- Knowing Interfaces
- Applying Interfaces

Lecture - 15

15.1 Snap Shot

In this lecture you will learn about the Interfaces, Defining an Interface, Implementing an Interface, what are the variables used in interface and how to extend an Interface.

15.2 Interfaces

Using the keyword interface, you can fully abstract a class' interface from its implementations. That is using **interface** , you can specify what a class must do, but not how it does it. Interfaces are syntactically similar to classes, but they lack instance variables and their methods are declared without any body. In practice, this means that you can define interfaces which don't make assumptions about how they are implemented. Once it is defined, any number of classes can implement an **interface**. Also one class can implement any number of interfaces.

To implement an interface, a class must create the complete set of methods defined by the interface. However, each class is free to determine the details of its own implementation. By providing the **interface** keyword, Java allows you to fully utilize the "one interface, multiple methods" aspect of polymorphism.

Interfaces are designed to support dynamic method resolution at run time. Normally, in order for a method to be called from one class to another, both classes need to be present at compile time so the Java compiler can check to ensure that the method signatures are compatible. This requirement by itself makes for a static and non extensible classing environment. Inevitably in a system like this functionality gets pushed up higher and higher in the class hierarchy so that the mechanisms will be available to more and more subclasses. Interfaces are designed to avoid this problem. They disconnect the definition of a method or set of methods from the inheritance hierarchy. Since interfaces are in a different hierarchy form classes, it is possible for classes that are unrelated in terms of the class hierarchy to implement the same interface. This is where the real power of interfaces is realized.

15.3 Defining an Interface

An interface is defined much like a class. This is the general form of an interface:

```
access interface name {
```

```
                return-type method-name1(parameter-list);
                return-type method-name2(parameter-list);
                type final-varname1 = value;
                type final-varname2 = value;
                // …
                return-type method-nameN(parameter-list);
                type final-varnameN = value;
        }
```

Here, access is either **public** or not used. When no access specifier is included, then default access results and the interface is only available to other members of the package in which it is declared. When it is declared as **public**, the interface can be used by any other code. *name* is the name of the interface and can be any valid identifier. Notice that the methods which are declared have nobodies. They end with a semicolon after the parameter list. They are essentially abstract methods; there can be no default implementation of any method specified within an interface. Each class that includes an interface must implement all of the methods.

Variables can be declared inside of interface declarations. They are implicitly **final** and **static**, meaning the implementing class cannot change them. They must also be initialized with a constant value. All methods and variables are implicitly **public** if the interface, itself, is declared as **public.**

Here is an example of an interface definition. It declares a simple interface which contains one method called **callback( )** that takes a single integer parameter.

```
interface Callback {
  void  callback (int param );
}
```

15.4 Implementing Interfaces

Once an **interface** has been defined, one or more classes can implement that interface. To implement an interface include the **implements** clause in a class definition, and then create the methods defined by the interface. The general form of a class that includes the implements clause looks like this:

```
  access class classname [extends superclass]
                    [implements interface [interface …] ] {
         / /  class-body
}
```

Here, access is either **public** or not used. If a class implements more than one interface, the interfaces are separated with a comma. If a class implements two interfaces that declare the same methods, then the same method will be used by clients of either interface. The methods that implement an interface must be declared **public.** Also, the type signature of the implementing method must match exactly the type signature specified in the **interface** definition.

Here is small example class that implements the **Callback** interface shown earlier.

```
   Class Client implements Callback {
  / / Implement Callback's interface
public void callback(  int   p ) {

         system.out.println ("callback called with" + p)

     }
}
```

Notice that **callback ( )** is declared using the **public** access specifier.

It is both permissible and common for classes that implement interfaces to define additional members of their own. For example, the following version of **Client** implements **callback ( )** and adds the method **nonIface Meth ( ).**

```
    class Client implements Callback {
/ / Implement Callback's interface
         public void callback (int  p ) {
           System.out.println ("callback called with" + p );
         }

         void nonifaceMeth ( ) {
          System.out.println ("Classes that implement interfaces" +
                               "may also define other members, too." );
                 }
         }
```

Accessing Implementations through Interface References

You can declare variables as object references that use an interface rather than a class type. Any instance of any class that implements the declared interface can be stored in such a variable. When you call a method through one of these references, the correct version will be called based on the actual instance of the interface being referred to. This is one of the key features of interface. The method to be executed is looked up dynamically at runtime, allowing classes to be created later than the code which calls methods on them. The calling code can dispatch through an interface without having to know anything about the "callee".

The following example calls the **callback ( )** method via an interface reference variable:

```
class TestIface  {
   public static void main (String args [ ]) {
       Callback c = new Client  ( ) ;
c.callback (42);
  }
}
```

The output of this program is shown here:

```
callback called with 42
```

Notice that variable c is declared to be of the interface type **callback,** yet it was assigned an instance of client. Although c can be used to access the **callback ( )** methods, it cannot access any other members of the client class. An interface reference variable only has knowledge of the methods declared by its **interface** declaration. Thus, c could not be used to access **nonIfaceMeth ( )** since it is defined by client but no **Callback.**

While the preceding example shows, mechanically, how an interface reference variable can access an implementation object, it does not demonstrate the polymorphic power of such a reference. To sample this usage, first create the second implementation of Callback, shown here:

```
   / / Another implementation of Callback
         class AnotherClient implements Callback {

  // Implement Callback's interface
         public void callback(int p) {
         System.out.println ("Another version of callback");
         System.out.println ("p squared is "+(p*p));
         }
         }
         Now, try the following class:
         class TestIface2 (
         public static void main(String args []) {
         Callback  c= new Client ( ) ;
         AnotherClient ob = new anotherClient ( );

         c.callback (42);

         c = ob; / / c now refers to AnotherClient object
         c.callback (42) ;
           }
         }
```

The output from this program is shown here:

```
 Callback called with 42
Another version of callback
p squared is 1764
```

As you can see the versions of **callback ( )** that is called is determined by the type of object that c refers to at run time. While this is a very simple example, you will see another, more practical one shortly.

15.5 Partial Implementations

If a class includes an interface but does not fully implement the methods defined by that interface then that class must be declared as **abstract**. For example:

```
         abstract class Incomplete implements Callback {
               int a, b;
           void show ( ) {
                 System.out.println (a+ " "    +b);
        }
          / / ….
}
```

Here, the class **Incomplete** does not implement **callback ( )** and must be declared as abstract. Any class that inherits **incomplete** must implement **callback ( )** or be declared **abstract** itself.

15.6 Applying Interfaces

To understand the power of interfaces, let's look at a more practical example. In earlier chapters you developed a class called **Stack** that implemented a simple fixed size stack. However there are many ways to implement a stack. For example, the stack can be of a fixed size or it can be " growable". The stack can also be held in an array, a linked list, a binary tree, and so on. No matter how the stack is implemented, the interface to the stack remains the same. That is, the methods **push ( )** and **pop ( )** define the interface to the stack independently of the details of the implementation. Because the interface to a stack is separate from its implementation, it is easy to define a stack interface, leaving it to each implementation to define the specifies. Let's look at two examples.

Frist, here is the interface that defines an integer stack. Put this in a file called **IntStack Java .** This interface will be used by both stack implementations.

```
 / /  Define an integer stack interface.
        interface Intstack {
          void push (int item); / / store an item
        int pop ( ) ; retrieve an item


        }
```

The following program creates a class called **FixedStack** that implements a fixed length version of an integer Stack.

```
 / / An implementation of IntStack that uses fixed storage.
         class FixedStack implements Intstack {
           private int stck [ ];
          private int tos ;

/ / allocate and initialize stack
          FixedStack (int size) {
        stck = new int (size);
        tos = -1;


        }

/ / Push an item onto the stack
        public void push ( int item)  {
        if (tos = =stck.length –1) / / use length member
          System.out.println (“Stack is full.”);
        else
```

```
     Stck [+ +tos] = item;
     }
/ / Pop an item from the stack
     public int pop ( ) {
     if (tos < 0) {
     System.out.println ("Stack underflow.");
     retrun 0;
     }
     else
      return stck [tos - -];
      }
     }

     class IFTest {
     public static void main(String args [ ]) {
       FixedStack mystack1 = new FixedStack (5) ;
       FixedStack mystack2 = new Fixedstack (8);

 / / push some numbers onto the stack
      for (int i=0; i<5; i++ ) mystack1. push(I;
      for (int I=0;I<8; I++) mystack2. push (I );

  / / pop those numbers off the stack
       System.out.println ("Stack in mystack1:");
       for (int I=0; I<5; I++);
        system.out.println(mystacl1.pop ( ) );

        system.out.println ("Stack in mystack2:");
        for (int I=0; I<8;I+ +)
          System.out.println(mystack2.pop ( ));
       }
     }
```

Following is another implementation of **IntStack** that creates a dynamic stack by use of the same **interface** definition. In this implementation, each stack is constructed with an initial length. If this initial length is exceeded, then the stack is increased in size. Each time more room is needed, the size of the stack is doubled.

```
/ /Implement a “growable” stack.
        class DynStack implemts Intstack {
        private int stck [ ];
        private int tos ;

/ / allocate and initialize stack
        DynStack (int size) {
        stck=new int [size];
        tos = -1;
        }
/ / Push an item onto the stack
        public void push (int item ) {

/ / if stack is full, allocate a large stack
        if (tos= = stack.length –1) {
        int temp [ ]=new int [stack.length * 2]; / / double size
        for (int i=0; i<stck.length; i+ +) temp[i] = stck [ i ];
        stck = temp;
        stck[+ +tos] =item;

        }
        else
        stck [+ +tos] = item;
        }

/ / Pop an item from the stack’
        public int pop( ) {
        if (tos<0)  {
        System.out.println(“Stack underflow.”);
        return 0;
        }
        else
          return stck [tos - -];
              }
        }


        class IFTest2 {
          public static void main (string args [ ]) {
              DynStack mystack1 = new DynStack (5);
              DynStack mystack 2 = new DynStack (8);

/ / these loops cause each stack to grow
        for (int i=0; i<12; i+ +) mystack1 push (i);
        for (int i=0; i<20; i++) mystack2 .push (i);

         System.out.println ("Stack in mystack1:");
        for (int i= 0; i<12; i ++ )
             system.out.println(mystack1.pop( )  ) ;
        System.out.println ("Stack in mystack2:");
        for (int i=0;i<20; i++)
              System.out.println(mystack2.pop ( ));

           }
        }
```

The following class uses both the **FixedStack** and **DynStack** implementation. It does so through an interface reference. This means that calls to **push ( )** and **pop( )** are resolved at run time rather than at compile time.

```
   /* Create an iterface variable and
       access stacks through it.
*/
class IFTest 3
```

```
          {
      public static void main (String args [ ] ) {
         IntStack mustack; / / create an interface reference variable
        DynStack ds = new FixedStack (5);
        FixedStack fs  =new FixedStack ( 8);

         mystack= ds;  / / load dynamic stack
         / / push some numbers onto the stack
        for (int i=0; i<12;i ++ ) mystack.push (i);

        mystack = fs; / / load fixed stack
        for (int i =0; i<8; i+ +) mystack.push (i);


          mystack = ds
          System.out.println ("Values in dynamic stack:")
          for (int i=0; i<12; i++
             system.out.println (mystack.pop ( ));

        mystack = fs;
System.out.println ("Values in fixed stack:");
for (int i=0; i<8; i++)
  System.out.println(mystack.pop ( )) ;
}
}
```

In this program**, mystack** is a reference to the **IntStack** interface. Thus, when it refers to **ds**, it uses the versions of **push ( )** and **pop ( )** defined by the **DynStack** implementation. When it refers to **fs**, it uses the versions of **push ( )** and **pop ( )** defined by **FixedStack.** As explained, these determinations are made at run time. Accessing multiple implementations of an interface through an interface reference variable is the most powerful way that Java achieves run-time polymorphism.

15.7 Variables in Interfaces

You can use interfaces to import shared constants into multiple classes by simply declaring an interface that contains variables which are initialized to the desired values. When you include that interface in a class (that is, when you "implement" the interface,) all of those variable names will be in scope as constants. This is similar to using a header file in C/C + +

to create a large number of **# defined** constants or **const** declaration. If an interface contains no methods, then any class that includes such an interface doesn't actually implement anything. It is as if that class were importing the constant variables into the class name space as **final**, variables. The next example uses this technique to implement an automated "decision maker".

```
   import java.util.Random;

interface SharedConstants {

 int NO = 0;
int YES = 1;
int MAYBE = 2;
int LATER =3;
int SOON =4;
int NEVER =5 ;

}

class Question implements SharedConstants {
   Random rand = new Random ( );
 int ask ( ) {
int prob =(int ) (100 * rand. nextDouble( ) );
if (prob < 30 )
 return NO;                    / /30%
else if (prob < 60 )
  return YES;                 / / 30%
else if (prob <75)
 return LATER     / / 15%
else if (prob <98)
  return SOON                 / / 13%
else
   return NEVER             / / 2%
   }
}

class AskMe implements sharedConstants {
 static void answer (int result ) {
 switch (result ) {
   case NO:
          System.out.println (“No”);
                break;
         case YES:
           System.out.println (“yes”);
            break;
    case MAYBE:
       System.out.println (“Maybe”);
         break;
      case LATER:
            System.out.println (“later”);
          break;
        case SOON:
           System.out.println (“soon”);
           break;
        case NEVER:
        Syste.out.println (“Nnever”);
            break;
    }

}
        public static void main (String args [ ] )
          Question q= new Question ( );
           answer (q.ask ( ));
           answer (q.ask ( ) );
           answer ( q.ask( ) );
            answer ( q.ask ( ) ) ;
  }
```

```
}
```

Notice that this program makes use of one of Java's standard classes: **Random.** This class provides pseudorandom numbers. It contains several methods which allow you to obtain random numbers in the form required by your program . In this example, the method **nextDouble ( )** is used. It return random numbers in the range 0.0 to 1.0

In this sample program, the two classes, Question and **AskMe**, both implement the **Shared Constants** interface where **NO, YES, MAYBE, SOON, LATER**, and **NEVER** are defined. Inside each class, the code refers to these constants as if each class had defined or inherited them directly. Here is the output of a sample run of this program. Note that the result are different each time it is run.

```
Later
Soon
No
Yes
```

15.8 Interfaces Can Be Extended

One interface can inherit another by use of the keyword extends. The syntax is the same as for inheriting classes. When a class implements an interface that inherits another interface, it must provide implementations for all methods defined within the interface inheritance chain. Following is an example:

```
    / / One interface can extend another.
  interface A {
    void meth1 ( );
     void meth2 ( );
   }
/ / B now includes meth1 ( ) and meth2 ( ) - - it adds meth3 ( ).
interface B extends A {
    void meth3 ( );
}

/ / This class must implement all of A and B
class MyClass implements B {
  public void meth1 ( ) {
      System.out.println ("Implement meth1 ( )." ) ;
}


 public void meth2 ( ) {
      system.out.println ("Implement meth2 ( ));
}


 public void meth3 ( ) {
      System.out.println ("Implement meth3 ( ) .") ;
 }


}
 class IFExtend {
        public static void main (String arg [ ]) {
           Myclass ob = new Myclass ( ) ;


   ob.meth1( );
    ob.meth2( );
     ob.meth3( );
  }
}
```

As an experiment you might want to try removing the implementation **formeth1** (in **MyClass**. This will cause a compile-time error. As stated earlier, any class the implements an interface must implement all methods defined by that interface, including any that are inherited from other interfaces.

Although the examples we've included in this book do not make frequent use for packages or interfaces, both of these tools are an important part of the Java programming environment. Virtually all real programs and applets that you write in Java will be contained within packages. A number will probably implement interfaces as well. It is important, therefore, that you be comfortable with their usage.

15.9 Short Summary

- Interfaces are designed to support dynamic method resolution at run time.

- If a class includes an interface but does not fully implement the methods defined by that interface then that class must be declared as abstract.

- When a class implements an interface that inherits another interface, it must provide implementations for all methods defined within the interface inheritance chain.

15.10 Brain Storm

1. What is the meaning of Interface?
2. Define the term Interface?
3. How to Implement an Interface?
4. How to use Interfaces for Multiple Classes?
5. How can you extend and Interface?

Lecture - 16

Garbage Collection

Objectives

In this lecture you will learn the following

- Garbage Collection
- Finalize Method

Lecture - 16

16.1 Snap Shot

In this lecture you will learn about the meaning of Garbage Collection and the use of Finalize Method.

16.2 Garbage Collection

Since objects are dynamically allocated by using the **new** operator, you might be wondering how such objects are destroyed and their memory released for later reallocation. In some languages, such as C++, dynamically allocated objects must be manually released by use of a **delete** operator. Java takes a different approach; it handles reallocation for you automatically. The technique that accomplishes this is called *garbage collection*. It works like this: when no references to an object exist, that object is assumed to be no longer needed, and the memory occupied by the object can be reclaimed. there is no explicit need to destroy objects as in C++. Garbage collection only occurs sporadically during the execution of your program. It will not occur simply because one or more objects exist that are no longer used. Furthermore, different Java run-time implementations will take varying approaches to garbage collections, but for the most part, you should not have to think about it while writing your programs.

16.3 The Finalize() Method

Sometimes an object will need to perform some action when it is destroyed. For example, if an object is holding some non-Java resource such as a file handle or window character font, then you might want to make sure these resources are freed before an object is destroyed. To handle such situations, Java provides a mechanism called finalization. By using finalization, you can define specific actions that will occur when an object is just about to be reclaimed by the garbage collector.

To add a finalizer to a class, you simply define the **finalize()** method. The Java runtime calls that method whenever it is about to recycle an object of that class. Inside the **finalize()** method you will specify those actions that must be performed before an object is destroyed. The garbage collector runs periodically, checking for objects that are no longer referenced by any running state or indirectly through other referenced objects. Right before an asset is freed, the Java run time calls the **finalize()** method on the object.

The **finalize()** method has this general form:

```
Protected void finalize()
{
 // Finalization code here
```

```
        }
```

Here, the keyword protected is a specifier that prevents access to **finalize()** by code defined outside its class.

It is important to understand that **finalize()** is only called just prior to garbage collection. It is not called when an object goes out-of-scope, for example. This means that you cannot know when-or even if – **finalize()** will be executed. Therefore, your program should provide other means of releasing system resources, etc., used by the object. It must not rely on **finalize()** for normal program operation.

16.4 Short Summary

- In Java the Dynamically Allocated Memories are released Automatically and is known as Garbage Collection.

- An Object will need to perform some action when it is destroyed, for this Java provides a mechanism called Finalization.

16.5 Brain Storm

1. What is the Meaning of Garbage Collection and how it is different from C++?

2. What is the use of Finalize Method?

Lecture - 17

Packages & Class Libraries

Objectives

In this lecture you will learn the following

- Knowing about Java Packages & Class Libraries
- Overview Access Specifiers

Lecture - 17

17.1 Snap Shot

In this lecture you will learn about the meaning of Packages, Java Class Libraries and about User Defined Packages.

17.2 Packages

Packages and interfaces are two of the basic components of java program. In general, a java source file can contain any of the following four internal parts:

A single package statement
Any number of import statements
A single public class declarations
Any number of classes private to the package

Only one of these- the single public class declaration- has been used in the examples so far. The name of each example class was taken from the same name space. This means that a unique name had to be used for each class to avoid name collisions.

After a while , without some way to manage the name space, you could run out of convenient, descriptive names for individual classes. You also need some way to be assured that the name you choose for a class will be reasonably unique and not collide with class names chosen by other programmers. Thankfully, Java provides a mechanism for partitioning the class name space into more manageable chunks.This mechanism is the package. The package is both a naming and a visibility control mechanism. You can define classes inside a package that are not accessible by code outside that package.

You can also define class members that are only exposed to other members of the same package. This allows your classes to have intimate knowledge of each other, but not expose that knowledge to the rest of the world.

17.3 Defining a package

To create a package is quite easy: simply include a **package** command as the first statement in a Java source file. Any classes declared within that file will belong to the specified package. The **package** statement defines a name space in which classes are stored. If you omit the **package** statement, the class names are out into the default package, which has no name. While the default package is fine for short, sample programs, it is inadequate for real applications. Most of the time, you will define a package for your code.

This is the general form of the **package** statement:

Package *pkg;*

Here, *pkg* is the name of the package. For example, the following statement creates a package called **Mypackage**.
package **Mypackage**;

Java uses file system directories to store packages. For example, the .class files for any classes you declare to be part of **Mypackage** must be stored in a directory called **Mypackage**. Remember that case is significant, and the directory name must match the package name exactly.

More than one file can include the same **package** statement. The **package** statement simply specifies to which package the classes defined in a file belong. It does not exclude other classes in other files from being part of that same package. Most real-world packages are spread across many files.

You can create a hierarchy of packages. To do so, simply separate each package name from the one above it by use of a period. The general form of a multileveled package statement is shown here:
package pkg1[.pkg2[pkg3]];

A package hierarchy must be reflected in the file system of your java development system. For example, a package declared as package java.awt.image;

Needs to be stored in **java/awt/image,java\awt\image,** or **java:awt:image** on your UNIX, windows, or Macintosh file system, respectively. Be sure to choose your package names carefully. You cannot rename a package without renaming the directory in which the classes are stored.

17.4 The Java class libraries

The built-in methods: **println()** and print() are members of the **System** class, which is a class predefined by Java that is automatically included in your programs. In the larger view, the Java environment relies on several built-in class libraries that contain many built-in methods that provide support for such things as I/O, string handling, networking and graphics. The standard classes also provide support for windowed output. Thus, Java as a totality is a combination of the Java language itself, plus its standard classes. As you will see, the class libraries provide much of the functionality that comes with Java. Indeed, part of becoming a Java programmer is learning to use the standard Java classes.

17.5 Overview Access Specifiers User defined Package Java.lang

We already know that access to a private member of a class is granted only to other members of that class. Packages add another dimension to access control. Java provides many levels of protection to allow fine-grained control over the visibility of variables and methods within classes, subclasses, and packages.

Classes and packages are both means of encapsulating and containing the name space and scope of variables and methods. Packages act as containers for classes and other subordinate packages. Classes act as containers for data and code. The class is java's smallest unit of abstraction. Because of the interplay between classes and packages, Java addresses four categories of visibility for class members

Subclasses in the package
Non-subclasses in the package
Subclasses in different package
Classes that are neither in the same package nor subclasses

The three access specifiers, private, public, and protected, provide a variety of ways to produce the many levels of access required by these categories.

While java's access control mechanism may seem complicated, we can simplify it as follows. Anything declared public can be accessed from anywhere. Anything declared private cannot be seen outside of its class. When a member does not have an explicit access specification, it is visible to subclasses as well as to other classes in the same package. This is the default access. If you want to allow an element to be seen outside your current package, but only to classes that subclass your class directly, then declare that element protected.

	Private	No Modifier	Protected	Public
Same class	Yes	Yes	Yes	Yes
Same Package				
Sub class	No	yes	Yes	Yes
Same Package				
Non-subclass	No	Yes	Yes	Yes
Different				

Package Subclass	No	No	Yes	Yes
Different Package Non-subclass	No	No	No	Yes

A class has only two possible access levels: default and public. When a class is declared as public, it is accessible by any other code. If a class has default access, then it can only be accessed by other code within its same package.

All of the standard java classes included with java are stored in a package called java. The basic language functions are stored in a package inside of the java package called java.lang. normally, you have to import every package or class that you want to use, but since java is useless without much of the functionality in java.lang, it is implicitly imported by the complier for all programs. This is equivalent to the following line being at the top of all of your programs:

```
Import java.lang.*;
```

Package

Java 2 adds a class called package that encapsulates version data associated with a package. Package version information is becoming more important because of the proliferation of packages and because a java program may need to know what version of a package is available.

17.6 Short Summary

- Packages and interfaces are two of the basic components of java program.
- The Package statement defines a name space in which classes are stored.
- Anything declared public can be accessed from anywhere. Anything declared private cannot be seen outside of its class.

17.7 Brain Storm

1. What is the meaning of Packages?
2. How to Define Package?
3. What are the Build-in Java Class Libraries?
4. What is User Defined Packages?

Lecture - 18

Built-in Classes

Objectives

In this lecture you will learn the following

- Knowing briefly about types of classes
- Collections of classes

Lecture - 18

18.1 Snap Shot

In this lecture you will learn about the various in-build functions such as String Buffer Classes, Math Classes, Java.util , Vector, Hashtable and about Collection .

18.2 String & String Buffer Classes

String is probably the most commonly used class in java's class library. The obvious reason for this is that strings are a very important part of programming.

The first thing to understand about strings is that every string you create is actually an object of type **String**. Even **string** constants are actually **string** objects. For example, in the statement System.out.println("this is a string, too");

the string "This is a String, too" is a String constant. Fortunately, Java handles String constants in the same way that other computer languages handle "normal" strings, so you don't have to worry about this.

The second thing to understand about strings is that objects of type String are immutable; once a String object is created, its contents cannot be altered. While this may seem like a serious restriction, it is not , for two reasons:

- If you need to change a string, you can always create a new one that contains the modifications.

- Java defines a peer class of string, called String buffer , which allows strings to be altered, so all of the normal string manipulations are still available in Java.

Strings can be constructed a variety of ways. The easiest is to use a statement like this:

```
String mystring = " this is a test"
```

Once you have created a **string** object, you can use it anywhere that a string is allowed. For example, this statement displays **mystring**:

```
System.out.println(mystring);
```

Java defines one operator for String objects: + . It is used to concatenate two strings. For example, this statement

String **myString** = "I"+"like" +"Java."

Results in **myString** containing "I like Java"

The following program demonstrates the preceding concepts:

```
          Demonstrating Systems
          class StringDemo {
           public static void main(String args[ ])
          {
          String s1 = "First String";
          String s2 = "Second String";
          String s3 = s1+ " and " + s2;
          System.out.println(s1);
          System.out.println(s2);
          System.out.println(s3);
          }
}
```

The output produced by this program is shown here:

```
First string
Second String
First string and Second String
```

The String class contains several methods that you can use. Here are a few. You can test two strings for equality by using **equals().** You can obtain the length of a string by calling the **length()** method. You can obtain the character at a specified index within a string by calling **charAt().** The general forms of these three methods are shown here:

```
          Boolean equals(String object)
int length()
char charAt(int index)
```

Here is a program that demonstrates these methods:

```
//Demonstrating some String methods.
class  StringDemo2 {
          public static void main(String a[])
          {
          String s1 = "First string";
          String s2 = "Second string";
          String s3 = s1;
          System.out.println("Length of s1 : " + s1.length());
          System.out.println("char at index 3 in s1 :" +s2.charAt(3));
          if(s1.equals(s2))
                    System.out.println("s1 == s2");
          else
                    System.out.println("s1 != s2");
          if(s1.equals(s3))
                    System.out.println("s1 == s3");
          else
                    System.out.println(“s1 != s3”);
          }
}
```

The output produced by this program is shown here:

```
Length of s1 : 12
Char at index 3 in s1 : s
s1 !=  s2
s1 == s3
```

The string constructors

The String class supports several constructors. To create an empty String, you call the default constructor. For example,

```
          String s = new String();
```

will create an instance of string with no characters in it.

Frequently, you will want to create strings that have initial values. The string class provides a variety of constructors to handle this. To create a string initialized by an array of characters, use the constructor chown here:

```
          String(char c[ ]) (Char Chars [ ])
```

Here is an example

```
          Char chars[]={‘a’,’b’,’c’};
          String s = new String(c);
```

This constructors initializes s with the string “abc”.

You can specify a sub range of a character array as an initializer using the following constructor:

```
String(char chars[],int startIndex, int numChars)
```

Here, startIndex specifies the index at which the subrange begins, and numChars specifies the number of characters to use. Here is an example:

```
          char chars[]= { ‘a’, ‘b’,’c’,’d’,’e’,’f’,};
          String s = new String(chars,2,3);
```

This initializes s with the characters cde.

You can construct a string object that contains the same character sequence as another **string** object using this constructor:

```
          String(String s)
```

Here, s is a string object. Consider this example:

```
        //Construct one string from another.
        class MakeString {
        public static void main(String a[ ]) {
        char c[] = { 'J','a','v','a'};
        String s1 = new String(c);
        String s2 = new String(s1);
        System.out.println(s1);
        System.out.println(s2);
        }
}
```

The output from this program is as follows:

```
Java
Java
```

As you can see, s1 and s2 contains the same string.

Special String Operations

Because strings are a common and important part of programming, java has added special support for several string operations within the syntax of the language. These operations include the automatic creation of new string instances from string literals, concatenation of multiple string objects by use of the + operator, and the conversion of other data types to a string representation. There are explicit methods available to perform all of these functions, but java does them automatically as a convenience for the programmer and to add clarity.

String Buffer Constructors

StirngBuffer defines these three constructors:

```
StringBuffer()
StringBuffer(int size)
StringBuffer(String str)
```

The default constructor (the one with no parameters) reserves room for 16 characters without deallocation. The second version accepts an integer arguments that explicitly sets the size of the buffer. The third version accepts a String argument that characters without reallocation. StringBuffer allocates room for 16 additional characters when no specific buffer length is requested, because reallocation is a costly allocating room for a few extra characters, StringBuffer reduces the number of reallocations that take place.

Length() and capacity()

The current length of a **StringBuffer** can be found via the **length()** method, while the total allocated capacity can be found through the **capacity()** method. They have the following general forms:

```
int length()
int capacity()
//StringBuffer length vs. capacity.
class StringBufferDemo
{
public static void main(string args[]){

StringBuffer sb = new StringBuffer("Hello");
 System.out.println("buffer ="  +sb);
System.out.println("length ="  +sb.length());
System.out.println("Capacity = " +sb.capacity());
}
}
```

Here is the output of this program, which shows how StringBuffer reserves extra space for additional manipulations:

```
buffer = Hello
length = 5
capacity = 21
```

Since **sb** is initialized with the string "Hello" when it is created, its length is 5. Its capacity is 21 because room for 16 additional characters is automatically added.

18.3 Math Classes

The **Math** class contains all the floating- point functions that are used for geometry and trigonometry, as well as several general-purpose methods. Math defines two double constants: E and PI

Transcendental Functions

Method	***Description***
Static double sin(double arg)	Returns the sine of the angle specified by arg in radians.
Static double asin(double arg)	Returns the angle whose sine is specified by arg.

Static double cos(double arg)	Returns the cosine of the angle specified by arg in radians.
Static double tan(double arg)	Returns the sine of the tangent specified by arg in radians.
Static double acos(double arg)	Returns the angle whose cosine is specified by arg.
Static double atan(double arg)	Retuns the angle whose tangent is specified by arg.
Static double atan2 (double x, double y)	Returns the angle whose tangent is x/y. Exponential Functions

Exponential Functions

Math defines the following exponential methods.

Method	Description
Static double exp(double arg)	Returns e to the arg.
Static double log(double arg)	Returns the natural logarithm of arg.
Static double pow (double y, double x)	Returns y raised to the x; for example Pow(2.0,3.0) returns 8.0.
Static double squrt(double arg)	Returns the square root of arg.

Rounding Functions

The Math class defines several methods that provide various types of rounding operations . They are shown in Table .

Method	Description
Static int abs(int arg)	Returns the absolute value of arg.
Static long abs(float arg)	Returns the absolute value of arg.
Static float abs(float arg)	Returns the absolute value of arg.
Static double abs(double a)	Returns the absolute value of arg.
Static double ceil(double a)	Returns the smallest whole number greater than or equal to arg.
Static double floor(double a)	Returns the largest whole number less than or equal to arg.
Static int max(int x, int y)	Returns the maximum of x and y.
Static long max(long x, long y)	Returns the maximum of x and y.
Static float max(float x, float y)	Returns the maximum of x and y.
Static double max	Returns the maximum of x and y.

(double x, double y)	
Static int min(int x, int y)	Returns the minimum of x and y.
Static long min(long x, long y)	Returns the minimum of x and y.
Static float min(float x, float y)	Returns the minimum of x and y.
Static double min (double x, double y)	Returns the minimum of x and y.
Static double rint(double arg)	Returns the integer nearest in value to arg.
Static int round(float arg)	Returns arg rounded up to the nearest int.
Static long round(double arg)	Returns arg rounded up to the nearest long.

18.4 Java.util

The **java.util** package contains some of the most exciting enhancements added by java 2: collections. A collection is a group of objects. The addition of collections caused fundamental alternation in the structure and architecture of many elements in **java.util**. It also expanded the domain of tasks to which the package can be applied. Collections are a state-of-the-art technology that merits close attention by all java programmers.

In addition to collection, **java.util** contains a wide assortment of classes and interfaces that support a broad range of functionality. These classes and interfaces are used throughout the core java packages and, of course, are also available for use in programs that you write. Their applications include generating pseudorandom numbers, manipulating date and time, observing events, manipulating sets of bits, and tokenizing strings. Because of its many features, **java.util** is one of Java's most widely used packages.

The **java.util** classes are listed here. The ones added by Java 2 are so labled.

AbstractCollection
AbstractList
AbstractMap
AbstractSequentialList
AbstractSet
ArrayList
Arrays
Bitset
Calendar
Collections

EventObject
GregorianCalendar
HashMap
HashSet
Hashtable
LinkedList
ListResourceBundle
Locale
Observable
Properties

PropertyResourceBundle
Random
ResourceBundle
SimpleTimeZone
Stack
StringTokenizer
TimeZone
TreeMap
TreeSet
Vector

Date	PropertyPermission	WeakHashMap
Dictionary		

Java.util defines the following interfaces. Notice that most were added by java 2.

Collection	**List**	**Observer**
Comparator	ListIterator	Set
Enumeration	Map	SortedMap
EventListener	Map.Entry	SortedSet
Iterator		

The **ResourceBundle, ListResourceBundle**, and **PropertyResourceBundle** classes aid in the internationalization of large programs with many locale-specific resources. These classes are not examined here.

18.5 Enumeration

The **enumeration** interface defines the methods by which you can enumerate the elements in a collection of objects. Iterator has superceded this legacy interface. Although not deprecated, **Enumeration** is considered obsolete for new code. However, it is used by several methods defined by the legacy classes, is used by several other API classes, and is currently in wide spread use in application code.

Enumeration specifies the following two methods:

```
Boolean hasMoreElements()
Object nextElement()
```

When implemented, hasMoreElements() must return true while there are still more elements to extract, and false when all the elements have been enumerated. nextElement() returns the next objects in the enumeration as a generic Object reference. That is, each call to nextElement() obtains the next object in the enumeration. The calling routine must cast that object into the object type held in the Enumeration.

18.6 Vector

Vector implements a dynamic array. It is similar to ArrayList, but with two differences: **Vector** is synchronized, and it contains many legacy methods that are not part of the collections framework. With the release of java2, vector was reengineered to extend AbstractList and implement the list interface, so it now is fully compatible with collections.

Here are the vector constructors.

```
Vector()
Vector(int size)
Vector(int size, int incr)
Vector(Collection c)
```

The first form creates a default vector, which has an initial size of 10. The second form creates a vector whose initial capacity is specified by size. The third form creates a vector whose initial capacity is specifies the number of elements to allocate each time that a vector is resized upward. The fourth form creates a vector that contains the elements of collection c. This constructor was added by Java2.

Vector defines these protected data members:

```
int capacityIncrement;
int elementCount;
Object elementData [ ];
```

The increment value is stored in capacity increment. The number of elements currently in the vector is stored in elementCount. The array that holds the vector is stored in elementData.

In addition to the collection methods defined by List, Vector defines several legacy methods, which are shown in Table .

Because Vector implements List you can use a vector just like you use an ArrayList instance. You can also manipulate one using its legacy methods. For example, after you instantiated a Vector, you can add an element to it by calling **addElement ( ).** To obtain the element at a specific location, call **elementsAt ( ).** To obtain the first element in the vector, call **firstElement ( ).** To retrieve the last element, call **lastElement ( )**. You can obtain the index of an element by using **indesOf( )** and **lastInsdexOf ( ).** To remove an element, **call removeElement ( )** or **removeElement( ).**

Method	Description
final void addElement(Object element)	The object specified by element is added to the vector.
final int capacity()	Returns the capacity of the vector.

Object clone()	Returns a duplicate of the invoking vector.
final boolean contains(Object element)	Returns true if element is contained by the vector, and returns false if it is not.
final void copyInto(Object array[])	The elements contained in the invoking vector are copied into the array specified by array.
final Object elementAt(int index)	Returns the element at the location specified by index.
final Enumeration elements()	Returns an enumeration of the elements in the vector.
final void ensureCapacity(int size)	Sets the minimum capacity of the vector to size.
Final void insertElementAt (Object element, int index)	Adds element to the vector at the location specified by index.

18.7 Hashtable

Hashtable was part of the original **java.util** and is a concrete implementation of a dictionary. However, Java2 reengineered **Hashtable** so that it also implements the map interface. Thus, Hashtable is now integrated into the collections framework. It is similar to **HashMap**, but is synchronized.

Like HashMap, Hashtable stores key/value pairs in a hash table. When using a Hashtable, you specify an object that is used as a key, and the value that you want linked to that key. The key is then hashed, and the resulting hash code is used as the index at which the value is stored within the table.
A hash table can only store objects that override the **hashCode()** and **equals()** methods that are defined by object. The **hashCode()** method must compute and return the hash code for the object. Of course, **equals()** compares two objects. Fortunately, many of Java's built-in classes already implement the **hashCode()** method. For example, the most common type of Hashtable uses a String object as the key. String implements both **hashCode()** and **equals()**.

The Hashtable constructors are shown here:

```
Hashtable()
Hashtable(int size)
Hashtable(int size, float fillratio)
```

Hashtable(Map m)

The first version is the default constructor. The second version creates a hash table that has an initial size specified by size. The third version creates a hash table that has an initial size specified by size and a fill ratio specified by fillratio. This ratio must be between 0.0 and 1.0. and it determines how full the hash table is expected. If you do not specify a fill ratio, then 0.75 is used. Finally, the fourth version creates a hash table that is initialized with the elements in m. The capacity of the hash table is set to twice the number of elements in m. The default load factor of 0.75 is used. The fourth constructor was added by Java 2.

In addition to the methods defined by the Map interface, which Hashtable now implements, Hashtable defines the legacy methods

Method	**Description**
void clear()	Resets and empties the hash table.
Object clone()	Returns a duplicate of the invoking object.
boolean contains(Object value)	Returns true if some value equal to value exists within the Hashtable . Returns false if the value isn't found.
boolean containsKey(Object key)	Returns true if some key equal to key exists within the hashtable. Returns false if the key isn’t found.

18.8 Collection

The Java collections framework standardizes the way in which groups of objects are handled by your programs. In the past, Java provided ad hoc classes such as Dictionary, vector,stack and properties to store and manipulate groups of objects. Although these classes were quite useful, they lacked a central, unifying theme. Thus, the way that you used vector was different from the way that you used properties, for example. Also, the previous, ad hoc approach was not designed to be easily extensible or adaptable. Collections are an answer to these problems.

The collections framework was designed to meet several goals. First, the framework had to be high-performance. The implementations for the fundamental collections are highly efficient. You seldom, if ever, need to code one of these “data engines” manually. Second, the framework had to allow different types of collections to work in a similar manner and with a high degree of interoperability. Third, extending and /or adapting a collection had to be easy. Toward this end, the entire collections framework is designed around a set of standard interfaces. Several standard implementations of these interfaces are provided that you may use as-is. You may also implement your own collection class easier. Finally, Mechanisms were added that allow the integration of standard arrays into collections framework.

Algorithms are another important part of the collection mechanism. Algorithms operate on collections and are defined as static methods within the collections class. Thus, they are available for all collections. Each collection class need not implement its own versions. The algorithms provide a standard means of manipulating collections.

Another item created by the collections framework is the Iterator interface. An Iterator gives you a general-purpose, standardized way of accessing the elements within a collection, one at a time. Thus , an iterator provides a means of enumerating the contents of a collection. Because each collection implements Iterator, the elements of any collection class can be accessed through the methods defined by Iterator. Thus, with only small changes, the code that cycles through a set can also be used to cycle through a list, for example.

In addition to collection, the framework defines several map interfaces and classes. Map store key/value pairs. Although maps are not "collections" in the proper use of the term, they are fully integrated with collections. In the language of the collections framework, you stored in a collection . Thus, you can process the contents of a map as a collection. If you choose.

The collection mechanism was retrofitted to some of the original classes defined by java.util so that they too could be integrated into the new system. It is important to understand that although the addition of collections has altered the architecture of many of the original utility classes, it has not caused the deprecation of any. Collections simply provide a better way of doing several things.

One last thing: if you are familiar with c++, then you will find it helpful to know that the java collections technology is similar in sprit to the standard template library defined by c++. What c++ calls a container, java calls a collection.

18.9 Short Summary

- Once you have created a string object, you can use it anywhere that a string is allowed.
- The math class contains all the floating- point functions that are used for geometry and trigonometry, as well as several general-purpose methods.

- The java.util package contains some of the most exciting enhancements added by Java 2: collections.
- Hashtable was part of the original java.util and is a concrete implementation of a dictionary
- The Java collections framework standardizes the way in which groups of objects are handled by your programs.

18.10 Brain Storm

1. What is the use of String Class?
2. Explain the Math Class.
3. What is Enumerator?
4. Explain the Java.util class.
5. How the Vector Implements the Dynamic Array?
6. What is the use of Hashtable?
7. Explain the Collection Object.

Lecture - 19

Exception Handling

Objectives

In this lecture you will learn the following

- Knowing all types of exception
- Throw , Try and Catch Blocks
- The Finally Clause

Lecture - 19

19.1 Snap Shot

Error conditions that are not expected to occur under normal conditions are called exceptions. When an exception occurs, bad things happen. For example, a negative number is passed to a function that computes the square root of a number. The function expects all numbers it receives to be positive real numbers. It receives a negative number instead, and an exception occurs. Sometimes programs die right then and there; other times they do more insidious things, such as passing incorrect pointers that eventually access protected areas of the system.

Although there is no definitive way to handle exceptions in C++, any bleary-eyed C++ programmers will be happy to know that exceptions are a fundamental part of the Java programming language. In Java if you call a method that could throw an exception, you must check to see if any of the possible exceptions occurred and handle them. Additionally, the Java compiler checks for exception handling and will tell you if you have not handled the exceptions for a particular method.

There are problem that are beyond program control, and therefore also beyond the programmer's control. these include problems such as running out of memory. other nonprogrammatic problems are the network being down or a hardware failure.

There are two main classes of problems in java: errors and exceptions. errors are caused by problems in java itself and are generally of too detailed a nature for the program itself to solve. when an error is encountered, java generates an error message to the screen and aborts the program.

19.2 Exception Handling Fundamentals

A Java exception is an object that describes an exceptional (that is, error) condition that has occurred in a piece of code. When an exceptional condition arises, an object representing that exception is created and thrown in the method that caused the error. That methods may choose to handle the exception itself, or pass it on. Either way, at some point the exception is caught and processed. Exceptions can be generated by the Java run-time system, or they can be manually generated by your code. Exceptions thrown by Java relate to fundamental errors that violate the rules of the Java language or the constraints of the Java execution environment . Manually generated exceptions are typically used to report some error condition to the caller of a method.

Java exception handling is managed via five keywords; try catch, throw, throws, and finally. Briefly here is how they work. Program statements that you want to monitor for exceptions are contained within a try block. If an exception occurs within the try block, it is thrown. Your code can catch this exception (using catch) and handle it in some rational manner. System-generated exceptions are automatically thrown by the Java run-time system. To manually throw an exception, use the keyword throw. Any exception that is thrown out of a method must be specified as such by a throws clause. Any code that absolutely must be executed before a method returns is put in a finally block.

This is the general form of an exception-handling block:

```
try{

/ / block of code to monitor for errors
        }
        catch (ExceptionType1 exOb){

/  / exception handler for Exception Type1
        }
        catch (ExceptionType2 exOb) {

/ / exception handler for Exception Type2
        }

/ / …
        finally{

/ / block of code to be executed before try block ends.
}
```

Here Exception type is the type of exception that has occurred. The remainder of this chapter describes how to apply this framework.

19.3 Exception Types

All exception types are subclasses of the built in class Throwable. Thus, Throwable is at the top of the exception class hierarchy. Immediately below Throwable are two subclasses that partition exceptions into two distinct branches. One branch is headed by Exception. This class is used is used for exceptional conditions that user programs should catch. This is also the class that you will subclass to create your own custom exception types. There is an important subclass of Exception ,called RuntimeException. Exceptions of this type are automatically defined for the programs that you write and include things such as division by zero and invalid array indexing.

The other branch is topped by Error, which defines exceptions that are not expected to be caught under normal circumstances by your program. Exceptions of type Error are used by the Java run-time system to indicate errors having to do with the run-time environment, itself. Stack overflow is an example of such an error. This chapter will not be dealing with exceptions of type Error, because these are typically created in response to catastrophic failures that cannot usually be handled by your program.

Uncaught Exceptions

Before you learn how to handle exceptions in your program, it is useful to see what happens when you don't handle them. This small program includes an expression that intentionally causes a divide by zero error.

```
 Class Exc0 {
            public  static void main(string   args[]) {
               int d= 0;
               int a =42 / d;
             }
}
```

When the java run time system detects the attempt to divide by zero, it constructs a new exception object and then throws this exception. This causes the execution of EXC0 to stop, because once an exception has been thrown, it must be caught by an exception handler and dealt with immediately. In this example, we haven't supplied any exception handlers of our own, so the exception is caught by the default handler provided by the Java run-time system. Any exception that is not caught by your program will ultimately be processed by the default handler. The default handler displays a string describing the exception, prints a stack trace from the point at which the exception occurred, and terminated the program.

Here is the output generated when this example is executed by the standard Java JDK run-time interpreter:

```
java.lang.ArithmeticException:/ by zero
     at Exc0.main(Exc0.java:4)
```

Notice how the class name, Exc0; the method name, main; the filename, Exc0.java; and the line number , 4, are all included in the simple stack trace.

Also, notice that the type of the exception thrown is a subclass of Exception called Arithmetic Exception, which more specifically describes what type of error happened. As disculled later in this chapter Jave suppplies several built in exception types that match the various sorts of run-time errors that can be generated .

The stack trace will always show the sequence of method invocations that led up to the error. For example, here is another version of the preceding program that introduces the same error but in a method separate from main();

```
    class Excl{
        static void subroutine() {
             int = 0;
            int a =10/d;
    }
    public static void main (string args[]) {
   Excl.subroutine();
}
}
```

The resulting stack trace from the default exception handler shows how the entire call stack is displayed;

```
java.lang.ArithmeticException : / by zero
      at Excl.subrooutine(Excl.java:4)
      at Excl.main(Excl.java:7)
```

As you can see, the bottom of the stack is main's line 7, which is the call to subroutine(), which caused the exception at line4. The call stack is quite useful for debugging, because it pinpoints the precise sequence of steps that led to the error.

Using try and catch

Although the default exception handler provided by the Java run-time system is useful for debugging, you will usually want to handle an exception yourself. Doing so provides two benefits. First, it allows you to fix the error. Second, it prevents the program from automatically terminating. Most users would be confused if your program stopped running and printed a stack trace whenever an error occurred! Fortunately it is quite easy to prevent this.

To guard against and handle a run-time error, simply enclose the code that you want to monitor inside a try block. Immediately following the try block, include a catch cluse that specifies the exception type that you wish to catch. To illustrate how easily this can be done, the following program includes a try block and a catch clause which processes the ArithmeticException generated by the division-by-zero error.

```
class Exc2 {
  public static void main (String args[]) {
    int d,a;
  try { // monitor a block of code.
      d=0;
       a=42/d;
    system.out.println("This will not be printed.");
  }  catch (ArithmeticException e) { // catch divide-by-zero error
       System.out.println("Division by zero.");
}
System.out.println("After catch statement");
}
```

This program generates the following output:

```
   Division by Zero
   After catch statement.
```

Notice that the call println ( ) inside the try block is never executed. Once an exception is thrown, program control transfers out of the **try** block into the **catch** block.

Put differently, catch is not "called" so execution never "returns" to the **try** block from a **catch**. "Thus, the line this will not be printed". is not displayed. Once the catch statements has executed, program control continues with the next line in the program following the entire try/catch mechanism.

A **try** and its **catch** statement form a unit. The scope for the catch clause is restricted to those statements specified by the immediately preceding try statement. A **catch** statement cannot catch an exception thrown by another **try** statement (except in the case of nested try statements, described shortly.) The statements that are protected by **try** must be surrounded by curly braces. (That is they must be within a block). You cannot use try on a single statement.

The goal of most well constructed **catch** clauses should be to resolve the exceptional condition and then continue on as if the error has never happened . For example, in the next program each interaction of the **for** loop obtains two random integers. Those two integers are divided by each other, and the result is used to divide the value 12345. The final result is put into **a**. If either division operation causes a divide by zero error , it is caught, the value of a is set to zero, and the program continues.

Exception Methods

Most Java exception handling is performed using the try, catch,throw and finally methods. Ofcourse, these methods can be extended if some unusual circumstance requires it.

Java uses the try, catch, and throw keywords to do actual exception handling. They are conceptually similar to a switch statement; think of try like the switch statement interms of exactly identifying the condition to be tested.

Catch is used to specify the action that should be taken for a particular type of exception. It is similar to the case part of a switchstatement. There can be several catch statements in a row to deal with each of the exceptions that may be generated in the block specified by the try statement.

Throw

Understanding exception handling in Java requires that you learn some new terminology. The first concept you need to grasp is that of throwing an exception, Java's name for causing an exception to be generated. For example say a method was written to read a file. If the methodcould not read the file because the file did not exist, this would generate an IOExecption. In Java terminology, it is said that the method threw an IOException.

Think of it as a horse throwing a shoe: You must stop everything before real damage is done.

Catch

The next term to learn in Java exception handling is catch. An exception catch is code that realizes the exception has occurred and deals with it appropriately. In java terms, you say a thrown exception gets caught.

In the case of the IOException thrown because of the nonexistence file mentioned in the previous section, the catch statement writes an error message to the screen stating that the specified file does not exist. It then allows the user to try entering a different filename if the first was incorrect, or it may exit. In Java terminology, the IOException was caught.

Try

Try is the java exception handling term that means a Java program is going to try to execute a block of code that might generate(throw) an exception. The try is way of telling the compiler that some attempt will be made to deal with at least some of the exceptions generated by the block of code.

Finally

The **finally** statement is used to specify the action to take if none of the previous catch statements specifically deals with the situation. It is similar to the default part of a switch statement, finally is the big net that catches everything that falls out of the exception handling statement.
An Example with **try, catch**, and finally

The following example is the standard structure for Java exception handling, incorporating **try, catch**, and **finally;**

```
try{
    statement that generates an exception
}
catch(ExceptionType1 e) {
 process exception type 1
}
 catch(ExceptionType2 e) {
 process exception type2
}
finally {
  process all other exception types
}
```

Using **try** in exception handling **try** is used to inform **Java** that a block of code may generate an exception and that some processing of that exception will be done immediately following the try. The syntax of **try** is

```
 try statement;
  or

try {
  statement (s)}
```

The statement try begins the try construct and is followed by a statement or block containing the code that might generate an exception. this code could consist of several statements, one or more of which may generate an exception.

If any one statement generates an exception, the remaining statements in the block are skipped an execution continues with the first statement following the try construct, which must be a catch or finally statement. This is an important point to remember. It is an easy way to determine whilch block of code should be skipped if an error occurs. Here is an example:

```
public class Mymain {
  public static void main (String args[]) {
 int[] myArray =new int[10];
  try{
System.output.println("Before valid array assignment) ;
Myarray[0]=1;
System.output.println("Before valid array assignment);
MyArray[100]=1;
System.output.println("After array exception");
}
}
}
```

In this example the array MyArray is created with a length of 10. This is followed by a **try** statement that contains several statements. The **First, Third**, and **Fifth** statements simply write trace messages to the screen. The **Second** statement contains a standard assignment statement that assigns the value 1 to array element 0. The **Third** statement also assigns an array, but attempts to assign a value of 1 to element 100 of the array. Because the array is only 10 in size, this generates an ArrayIndexOutBounds exception.

In tracing the execution of the block of code following the **try** statement, the first three statements are executed normally. The Fourth statement, the invalid assignment, will strt to execute and then generate an exception, which causes execution to continue at the end of the block, skipping the Fifth statement.

A Compilation error will result if you attempt to compile this code as it stands because any try statement must be followed immediately by one or more **catch** or **finally** statements. No othertype of statement is allowed after the end of the **try** statement and before the first **catch** or **finally** statement.

19.4 Throw, Try and Catch Blocks

To respond to an exception, the call to the method that produces it must be placed within a try block. A try block is a block of code beginning with the try keyword followed by a left and a right curly brace. Every try block is associated with one or more catch blocks. Here is a try block:

```
try
  {
  // method calls go here
  }
```

If a method is to catch exceptions thrown by the methods it calls, the calls must be placed within a try block. If an exception is thrown, it is handled in a catch block. Different catch blocks handle different types of exceptions. This is a **try** block and a **catch** block set up to handle exceptions of type Exception:

```
try
  {
  // method calls go here
  }
catch( Exception e )
  {
  // handle exceptions here
  }
```

When any method in the try block throws any type of exception, execution of the try block ceases. Program control passes immediately to the associated catch block. If the catch block can handle the given exception type, it takes over. If it cannot handle the exception, the exception is passed to the method's caller. In an application, this process goes on until a catch block catches the exception or the exception reaches the **main()** method uncaught and causes the application to terminate.

An Exceptional Example

Because all Java methods are class members, the passingGrade() method is incorporated in the gradeTest application class. Because **main()** calls **passingGrade(), main()** must be able to catch any exceptions passingGrade() might throw. To do this, main() places the call to passingGrade() in a try block. Because the throws clause lists type Exception, the catch block catches the Exception class.

```
//The gradeTest application.
      import Java.io.* ;
      import Java.lang.Exception ;
      public class gradeTest {

        public static void main( String[] args ) {

          try
            {
            // the second call to passingGrade throws
            // an excption so the third call never
            // gets executed

            System.out.println( passingGrade( 60,  80 ) ) ;
            System.out.println( passingGrade( 75,   0 ) ) ;
            System.out.println( passingGrade( 90, 100 ) ) ;
            }
          catch( Exception e )
            {
            System.out.println( "Caught exception --" +
                        e.getMessage() ) ;
            }
        }

        static boolean passingGrade( int correct, int total )
                          throws Exception {

          boolean returnCode = false ;

          if( correct > total ) {
            throw new Exception( "Invalid values" ) ;
          }
        if ( (float)correct / (float)total > 0.70 ) {
            returnCode = true ;
        }

        return returnCode ;
    }

}
```

The second call to **passingGrade()** fails in this case, because the method checks to see whether the number of correct responses is less than the total responses. When **passingGrade()** throws the exception, control passes to the **main()** method. In this example, the catch block in **main()** catches the exception and prints Caught exception -- Invalid values.

Multiple catch Blocks

In some cases, a method may have to catch different types of exceptions. Java supports multiple catch blocks. Each catch block must specify a different type of exception:

```
try
   {
   // method calls go here
   }
catch( SomeExceptionClass e )
   {
   // handle SomeExceptionClass exceptions here
   }
catch( SomeOtherExceptionClass e )
   {
   // handle SomeOtherExceptionClass exceptions here
   }
```

When an exception is thrown in the try block, it is caught by the first catch block of the appropriate type. Only one catch block in a given set will be executed. Notice that the catch block looks a lot like a method declaration. The exception caught in a catch block is a local reference to the actual exception object. You can use this exception object to help determine what caused the exception to be thrown in the first place.

A method that ignores exceptions thrown by the method it calls.

```
import Java.io.* ;
import Java.lang.Exception ;
```

```
public class MultiThrow {

   public static void main( String[] args ) {

      try
         {
         foo() ;
         }
      catch( Exception e )
         {
         System.out.println( "Caught exception " +
                        e.getMessage() ) ;
         }

   }

   static void foo() throws Exception {

      bar() ;

   }

   static void bar() throws Exception {

      throw new Exception( "Who cares" ) ;

   }

}
```

In the example main() calls foo() which calls bar(). Because bar() throws an exception and doesn't catch it, foo() has the opportunity to catch it. The foo() method has no catch block, so it cannot catch the exception. In this case, the exception propagates up the call stack to foo()'s caller, main().

// A method that catches and re throws an exception.

```
import java.io.* ;
import java.lang.Exception ;
public class MultiThrow {

   public static void main( String[] args ) {
      try
         {
         foo() ;
         }
      catch( Exception e )
         {
         System.out.println( "Caught exception " +
                        e.getMessage() ) ;
         }

   }

   static void foo() throws Exception {

      try
         {
         bar() ;
         }
      catch( Exception e )
         {
         System.out.println( "Re throw exception -- " +
                        e.getMessage() ) ;
         throw e ;
         }   }

   static void bar() throws Exception {

      throw new Exception( "Who cares" ) ;

   }
}
```

The foo() method calls bar(). The bar() method throws an exception and foo() catches it. In this example, foo() simply rethrows the exception, which is ultimately caught in the application's main() method. In a real application, foo() could do some processing and then rethrow the exception. This arrangement allows both foo() and main() to handle the exception.

The Throwable Class

All exceptions in Java are sub classed from the class **Throwable**. If you want to create your own exception classes, you must subclass Throwable. Most Java programs do not have to subclass their own exception classes.

Following is the public portion of the class definition of Throwable:

```
public class Throwable {

    public Throwable() ;
    public Throwable(String message) ;
    public String getMessage()
    public String toString() ;
    public void printStackTrace() ;
    public void printStackTrace(java.io.PrintStream s) ;
    private native void printStackTrace0(java.io.PrintStream s);
    public native Throwable fillInStackTrace();
}
```

The constructor takes a string that describes the exception. Later, when an exception is thrown, you can call the getMessage() method to get the error string that was reported.

The methods of the Java API and the language itself also throw exceptions. These exceptions can be broken into two classes: Exception and Error.

Both the Exception and Error classes are derived from Throwable. Exception and its subclasses are used to indicate conditions that may be recoverable. Error and its subclasses indicate conditions that are generally not recoverable and should cause your applet to terminate.

The various packages included in the Java Developers Kit throw different kinds of Exception and Error exceptions, as described in the following sections.

java.lang Exceptions

The java.lang package contains much of the core Java language. The exceptions subclassed from RuntimeException do not have to be declared in a method's throws clause. These exceptions are considered normal and nearly any method can throw them.

The java.lang exceptions.

Exception	Cause
ArithmeticException	Arithmetic error condition (for example, divide by zero).
ArrayIndexOutOfBoundsException	Array index is less than zero or greater than the actual size of the array.
ArrayStoreException	Object type mismatch between the array and the object to be stored in the array.
ClassCastException	Cast of object to inappropriate type.
ClassNotFoundException	Unable to load the requested class.

CloneNotSupportedException	Object does not implement the cloneable interface.
Exception	Root class of the exception hierarchy.
IllegalAccessException	Class is not accessible.
IllegalArgumentException	Method receives an illegal argument.
IllegalMonitorStateException	Improper monitor state (thread synchronization).
IllegalThreadStateException	The thread is in an improper state for the requested operation.
IndexOutOfBoundsException	Index is out of bounds.
InstantiationException	Attempt to create an instance of the abstract class.
InterruptedException	Thread interrupted.
NegativeArraySizeException	Array size is less than zero.
NoSuchMethodException	Unable to resolve method.
NullPointerException	Attempt to access a null object member.
NumberFormatException	Unable to convert the string to a number.
RuntimeException	Base class for many java.lang exceptions.
SecurityException	Security settings do not allow the operation.
StringIndexOutOfBoundsException	Index is negative or greater than the size of the string.

The java.lang errors.

Error	**Cause**
AbstractMethodError	Attempt to call an abstract method.
ClassCircularityError	This error is no longer used.
ClassFormatError	Invalid binary class format.
Error	Root class of the error hierarchy.
IllegalAccessError	Attempt to access an inaccessible object.
IncompatibleClassChangeError	Improper use of a class.
InstantiationError	Attempt to instantiate an abstract class.
InternalError	Error in the interpreter.
LinkageError	Error in class dependencies.
NoClassDefFoundError	Unable to find the class definition.

NoSuchFieldError	Unable to find the requested field.
NoSuchMethodError	Unable to find the requested method.
OutOfMemoryError	Out of memory.
StackOverflowError	Stack overflow.
ThreadDeath	Indicates that the thread will terminate. May be caught to perform cleanup. (If caught, must be rethrown.)
UnknownError	Unknown virtual machine error.
UnsatisfiedLinkError	Unresolved links in the loaded class.
VerifyError	Unable to verify bytecode.
VirtualMachineError	Root class for virtual machine errors.

java.io Exceptions

The classes in java.io throw a variety of exceptions. Any classes that work with I/O are good candidates to throw recoverable exceptions. For example, activities such as opening files or writing to files are likely to fail from time to time. The classes of the **java.io** package do not throw errors at all.

The java.io exceptions

Exception	**Cause**
IOException	Root class for I/O exceptions.
EOFException	End of file.
FileNotFoundException	Unable to locate the file.
InterruptedIOException	I/O operation was interrupted. Contains a bytesTransferred member that indicates how many bytes were transferred before the operation was interrupted.
UTFDataFormatException	Malformed UTF-8 string.

java.awt Exceptions

The AWT classes have members that throw one error and one exception:

- AWTException (exception in AWT)

- AWTError (error in AWT)

java.util Exceptions

The classes of java.util throw the following exceptions:

- EmptyStackException (no objects on stack)
- NoSuchElementException (no more objects in collection)

Built-In Exceptions

In the example you see how the automatic exceptions in Java work. This application creates a method and forces it to divide by zero. The method does not have to explicitly throw an exception because the division operator throws an exception when required.

```
// An example of a built-in exception.
    import java.io.* ;
    import java.lang.Exception ;

    public class DivideBy0 {
      public static void main( String[] args ) {

      int a = 2 ;
      int b = 3 ;
      int c = 5 ;
      int d = 0 ;
      int e = 1 ;
      int f = 3 ;

      try
        {
        System.out.println( a+"/"+b+" = "+div( a, b ) ) ;
        System.out.println( c+"/"+d+" = "+div( c, d ) ) ;
        System.out.println( e+"/"+f+" = "+div( e, f ) ) ;
        }
      catch( Exception except )
        {
        System.out.println( "Caught exception " +
                      except.getMessage() ) ;
        }
      }
    static int div( int a, int b ) {

        return (a/b) ;

    }

}
The output of this application is shown here:
2/3 = 0
Caught exception / by zero
```

The first call to **div()** works fine. The second call fails because of the divide-by-zero error. Even though the application did not specify it, an exception was thrown-and caught. So you can use arithmetic in your code without writing code that explicitly checks bounds.

19.5 The finally Clause

Finally, for **finally**. Suppose there is some action that you absolutely must do, no matter what happens. Usually, this is to free some external resource after acquiring it, to close a file after opening it, or something similar. To be sure that "no matter what" includes exceptions as well, you use a clause of the try statement designed for exactly this sort of thing, finally:

```
SomeFileClass  f = new SomeFileClass();

if (f.open("/a/file/name/path")) {

   try {

      someReallyExceptionalMethod();

   } finally {

      f.close();

   }

}
```

This use of finally behaves very much like the following

```
SomeFileClass  f = new SomeFileClass();
```

```
if (f.open("/a/file/name/path")) {

   try {

      someReallyExceptionalMethod();

   } catch (Throwable t) {

      f.close();

      throw t;

   }

}
```

except that finally can also be used to clean up not only after exceptions but after return, break, and continue statements as well. Here's a complex demonstration:

```
public class  MyFinalExceptionalClass extends ContextClass {

   public static void  main(String args[ ]) {
      int  mysteriousState = getContext();
      while (true) {
         System.out.print("Who ");
         try {
            System.out.print("is ");
            if (mysteriousState == 1)
               return;
            System.out.print("that ");
            if (mysteriousState == 2)
               break;
            System.out.print("strange ");
            if (mysteriousState == 3)
               continue;
            System.out.print("but kindly ");
```

```
          if (mysteriousState == 4)
             throw new UncaughtException();
          System.out.print(“not at all “);
       } finally {
          System.out.print(“amusing man?\n”);
       }
       System.out.print(“I’d like to meet the man “);
    }
    System.out.print(“Please tell me.\n”);
 }
}
```

Here is the output produced depending on the value of mysteriousState:

1. Who is amusing man?

2. Who is that amusing man?

   Please tell me

3. Who is that strange amusing man?

   ...

4. Who is that strange but kindly amusing man?

5. Who is that strange but kindly not at all amusing man?

   I’d like to meet the man Who is that strange...?

   ...

The exception handling mechanism in Java allows your methods to report errors in a manner that cannot be ignored. Every exception that is thrown must be caught or the application terminates. Exceptions are actually class objects derived from the Throwable class. Therefore, exceptions combine data and methods; an exception object generally contains a string explaining what the error is.

19.6 Short Summary

- A Java exception is an object that describes an exceptional (that is, error) condition that has occurred in a piece of code.

- Java exception handling is managed via five keywords; try catch, throw, throws, and finally.

- An exception catch is code that realizes the exception has occurred and deals with it appropriately.

- Try is the java exception handling term that means a Java program is going to try to execute a block of code that might generate(throw) an exception.

- All exceptions in Java are sub classed from the class Throwable

19.7 Brain Storm

1. What is an exception?
2. How do we define a try block?
3. How do we define a catch block?
4. Is it essential to catch all type of exceptions?
5. How many catch blocks can be use with one try block?
6. What is the finally block ?When and how it is used?
7. Define an exception call "NoMatchException".That thrown when a string is not equal to "India".

Lecture - 20

IO Stream

Objectives

In this lecture you will learn the following

- Knowing types of streams
- Files
- Filters
- Object Serialization

Lecture - 20

20.1 Snap Shot -I/O Streams

In fact, aside from print() and println(), none of the I/O methods have been used significantly. The reason is simple: most real applications of Java are not test-based, console programs. Rather, they are graphically oriented applets that rely upon Java's Abstract window toolkit for interaction with the user. Although test-based programs are excellent as teaching examples, they do not constitute an important use for Java in the real world. Also, Java's support for console I /O is just not very important to Java programming.

The preceding paragraph notwithstanding, Java's does provide strong, flexible support for I/O as it relates to files and networks. Java's I/O system is cohesive and consistent. In fact, once you understand its fundamentals, the rest of the I/O system is easy to master.

20.2 Streams

Java programs perform I/O through streams. A stream is an abstraction that either produces or consumes information. A stream is linked to a physical device by the Java I/O system. All streams behave in the same manner, even if the actual physical devices to which they are linked differ. Thus, the same I/O classes and methods can be applied to any type of device. This means that an input stream can abstract many different kinds of input: from a disk file, a keyboard, or a network socket. Likewise, an output stream may refer to the console, a disk file, or a network connection. Streams are a clean way to deal with input/output without having every part of your code understand the difference between a keyboard and a network, for example, . Java implements streams within class hierarchies defined in the java.io package.

20.3 Byte Streams and Character Streams

Java 2 defines two types of streams, byte and character. Byte streams provide a convenient means for handling input and output of bytes. Bytes steams are used, for example, when reading or writing binary data. Character streams provide a convenient means for handling input and output of characters. They use Unicode, and, a therefore, can be internationalized. Also, in some cases, character streams are more efficient that byte streams.

The original version of Java (Java 1.0) did not include character streams and, thus, all I/O was byte oriented. Character streams were added by Java 1.1, and certain byte-oriented classes and methods were deprecated. That is why older code that doesn't use character streams should be updated to take advantage of them, where appropriate.

One other point at the lowest level, all I/O is till byte oriented. The character –based streams simply provide a convenient and efficient means for handling character.

An overview of both byte-oriented streams and character oriented streams is presented in the following section.

The Byte Stream Classes

Byte streams are defined by using two class hierarchies. At the top are two abstract classes. **Inputstream** and **Outputstream**. Each of these abstract classes has several concrete subclasses, that handle the difference between various devices, such as disk files, network connections, and even memory buffers. The byte stream classes are shown in Table 12-1 . A few of these classes are discussed later in this section. Others are described in part 2 . Remember to use the stream classes, you must import **jave.io.**

The abstract classes **Inputstream** and **OutputStream** define several key methods that the other stream classes implement. Two of the most important are read ( ) and write ( ), which, respectively, read and write bytes of data. Both method are declared as abstract inside **InoutStream** and **OutputStream.** They are overridden by derived stream classes.

The Character Stream Classes

Character stream are defined by using two class hierarchies. At the top are two abstract classes, Reader and Writer. These abstract classes handle Unicode character streams. Java has server al concrete subclasses of each of these. The character stream classes are shown in Table.

The abstract classes Reader and Writer define several key methods that the other stream classes implement . Two of the most important methods are read ( ) and write ( ) which read and write characters of data, respectively. These methods are overridden by derived stream classes.

Stream Class	**Meaning**
BufferedInputstream	Buffered input stream
BufferedOutputStream	Buffered output stream
ByteArrayInputStream	Input steam that reads from a byte array

ByteArrayOutputStream	Output stream that writes to a byte array
DataInputStream	An input stream that contains methods foe reading the Java standard data types.
FileInputStream	Input stream that reads from a file
FileOutputStream	Output stream that writes to a file.
FileInputStream	implements **InputStream**
FilterOutputStream	Implements **OutputStream**
InputStream	Abstract class that describes stream input
OutputStream	Abstract class that describes stream output
PipedInputStream	Input pipe
PipedOutputStream,	Output pipe
PrintStream	Output stream that contains **print ( )** and **println ( )**
PushbackInputStream	Input stream that supports one-byte "unget" which returns a byte to the input stream.
RandomAccessFile	Supports random access file I/O
SequenceInputStream	Input stream that is a combination of two or more input Streams that till be read sequentially, one after the other.

Stream Class	**Meaning**
BufferedReader	Buffered input character stream
BufferedWriter	Buffered output character stream
CharArrayReader	Input steam that reads from a character array
CharArraryWriter	output stream that writes to a character array
FileReader	Input stream that writes to a file
FilterReader	Filtered reader
FilterWriter	Filtered writer
InputStreamReader	Input stream that translates to bytes characters.
LineNumberReadir	Input stream that counts lines.
OutputStreamWriter	Output steam that translates character to bytes.
PipedReader	Input pipe.
PipedWriter	Output pipe.

PrintWriter	Output stream that contains **print ( )** and **println( )**
PushbackReader	Input stream that allows characters to be returned to the Input stream.
Reader	Abstract class that describes character stream input
StriingReader	Input stream that reads from a string
StringWriter	Output stream that writes to a string
Writer	Abstract class that describes character stream output.

20.4 Files

Although most of the classes defined by **java.io** operate on streams, the **File** class does not. It deals directly with files and the file system. That is, the File class does not specify how information is retrieved from or stored in files; it describes the properties of a file itself. A File objects is used to obtain or manipulate the information associated with a disk file, such as the permissions, time, date, and directory path, and to navigate subdirectory hierarchies.

Files are a primary source and destination for data within many programs. Although there are severe restrictions on their use within applets for security reasons, files are still a central resource for storing persistent and shared information. A directory in Java is treated simply as a File with one additional property – a list of filenames that a can be examined by the **list( )** method

The following constructors can be used to create **File** objects

```
File( String directory Path)
File( String directoryPath, String filename)
File(FiledirObj,String filename)
```

Here directoryPath is the path name of the file, filename is the name of the file, and dirObj is a **File** object that specifies a directory.

The following example creates **three** files**:f1,f2** and **f3**. The first **File** object is constructed with a directory path as the only argument. The **second** includes two arguments - the path and the filename. The third includes the file path assigned to **f1** and a filename **f3** refers to the same file as **f2.**

```
File f1 = new File (“/”);
File f2 = new File (“/” , “autoexec.bat”);
File f3 = new File (f1,”autoexc.bat”);
```

File defines many methods that obtain the standard properties of a **File** object. For example, **getName ( )** returns the name of the file, **getParent ( )** returns the name of the parent directory, and **exists ( )** returns **true** if the file exists, **false** if it does not. The **File** class, however, is not symmetrical. By this, we mean that there are many methods that allow you to examine the properties of a simple file object, but no corresponding function exists to change those attributes. The following example demonstrates several of the **File** methods.

```
/ / Demonstrate File.
import java.io.File;
Class FileDemo {
   Static void p (String  s)    {
              System.out.println (s);
}
 public static void main (String args [ ] )  {
    File f1 – new File (“/java/COPYRIGHT”);
    p(“File Name: “ + f1.getName ( ) );
    p(“Path: “+ f1.getPath ( ) );
    p (“Abs Path: “ + f1 getAbsolutePath ( ) );
    p (“Parent: “ + f1.getAbsolutePath ( ) );
    p(f1.exists ( ) ? “exists” : does not exist” );
    p (f1.canWrite ( ) ? “is writeable” : is not writeable”);
    p (f1.canRead ( ) ? “is readable : “is not readable”);
    p (“is” + (f1.isDirectory ( ) ? “ “ : “not” + “a directory”);
    p (f1.isFile ( ) ? “is normal file” : “might be a named pipe”);
    p (f1.isAbsoulte ( ) ? “is absolute” : “is not abosulte” );
    p (“File last modified:” + f1.lastModified ( ));
    p (“File size: “ + f1 length ( ) + “Bytes” );
```

When you run this program, you will see something similar to the following:

```
File name : COPYRIGHT
Path :/ java/COPYRIGHT
Abs path :/java/COPYRIGHT
Parent: /java
exixts
is writeable
is readable
is not a directory
is normal file
is absolute
File last modified : 812465204000
File size : 695 Bytes
```

Most of the **File** methods are self explanatory. **isFile ( )** and is **Absoulte (**) are not. **isFile ( )** return true if called on a file and false if called on a directory. Also, **isFile ( )** returns false for some special files, such as device drivers and named pipes, so this method can a be used to make sure the file will behave as a file. The **isAbsoulte ( )** method returns true if the file has an absoulte path and **false** if its path is relative.

File also includes two useful utility methods. The first is **rename To( ),** shown here:

```
boolean renameTo  (File newName)
```

Here, the filename specified by new Name becomes the name of the invoking **File** object. It will return **true** upon success and **false** if the file cannot be renamed (if you either attempt to rename a file so that it move from one directory to another or use an existing filename, for example).

The second utility method is **delete( )** which deletes the disk file represented by the path of the invoking **File** object. It is shown here:

```
boolean delete( )
```

You can also use **delete ( )** to delete a directory if the directory is empty. **delete(**) returns if it deletes the file and **false** if the file cannot be removed.

Java 2 adds some new method to **File** that you might find helpful in certain situations. Some of the most interesting are should here:

Method	**Description**
void deleteOnExit ( )	Removes the file associated with the invoking object. When the Java Virtual Machine terminates
boolean isHidden ( )	Returns **true** if the invoking file is hidden. Return **false** Otherwise.
boolean setLastModified (long Millisec)	Sets the time stamp on the invoking file to that specified by millisec, which is the number of milliseconds from January 1, 1970, Coordinated Universal Time (UTC).
Boolean setReadOnly ( )	Sets the invoking file to read only.

Also, because **File** now supports the **Comparable** interface, the methods **compareTo( )**is Also supported.

20.5 Filtered Byte Streams

Filtered streams are simply wrappers around underlying input or output streams that transparently provide some extended level of functionality. These streams are typically accessed by methods that are expecting a generic streams, which is a superclass of the translation. The filtered byte streams are **FilterInputStream** and **FilterOutputStream**. Their constructors are shown here:

FilterOutputStream(OutputStream os)
FilterInputStream(InputStream is)

The method provided in these classes are identical to those in **InputStream** and **OutputStream**

20.6 Serialization

Serialization is the process of writing the state of an object to a byte stream. This is useful when you want to save the state of your program to a persistent storage area, such as a file. At a later time, you may restore these objects by using the process of deserialization.

Serialization is also needed to implement Remote Method Invocation(RMI).RMI allows a Java object on one machine to invoke a method of a Java object on a different machine. An object may be supplied as an argument to that remote method. The sending machine serializes the object and transmits it. The receiving machine deserializes it.

Assume that an object to be serialized has references to other object, which in turn, have references to still more objects. This set of objects and the relationships among them form a directed graph. There may also be circular references within this object graph. That is object X may contain a references to themselves. The object serialization and deserialization facilities have been designed to work correctly in these scenarios. If you attempt to serialize of an object at the top of an object graph, all of the other referenced objects are recursively located and serialized. Similarly, during the process of deserialization all of these objects and their references are correctly restored.

An overview of the interfaces and classes that support serialization follows.

Serializable

Only an object that implements the serializable interface can be saved and restored by the serialization facilities. The Serialization interface defines no members. It is simply used to indicate that a class may be serialized. If a class is serializable all of its subclasses are also serializable.

Variables that are declared as **transient** are not saved by the serialization facilities. Also **static** variables are not saved.

Externalizable

The Java facilities for serialization and deserialization have been designed so that much of the work to save and restore the state of an object occurs automatically. However there are cases in which the programmer may need to have control over these processes. For example, it may be desirable to used compression or encryption techniques. The **Externalizable** interface is designed for these situations.

The **Externalizable** interface defines these two methods:

```
void readExteranal (ObjectInput in Stream)
  throws IOException, ClassNotFountExpecption

void writeExternal (ObjectOutput outStream)
                  throws IOException
```

In these methods, inStream is the byte steam from which the object is to be read, and outStream is the byte stream to which the object is to be written.

ObjectOutput

The **ObjectOutput** interface extends the **DataOutput** interface and supports object serialization. It defines the methods shown in Table 17-5. Note especially the **writeObject ( )** method. This is called to serialize an object. All of these methods will throw an **IOException** on error conditions

Method	Description
void close ( )	Closes the invoking stream. Further write attempts will generate an **IOException.**

void flush ( )	Finalized the output state so that any buffers are cleared That is, it flushes the output buffers.
void write (byte buffer [ ])	Writes an array of bytes to the invoking stream.
void write (byte buffer[ ], int offset, int num Bytes)	Writes a subrange of numBytes bytes from the array buffer, beginning at buffer [offset].
void write (int b)	Writes a single byte to the invoking stream. The Byte written is the low-order byte of b.
void writeObject (Object obj)	Writes object obj to the invoking stream.

ObjectOutput Stream

The ObjectOutputStream **calss** extends the OutputStream **class** and implements the ObjectOutput interface. It is responsible for writing to a stream. The constructor of this **class** is

ObjectOutputStream (OutputStream outStream) throws IOException

The argument outStream is the output stream to which serialized object will be written. The most commonly used methods in this class are shown in Table 17-6. They will throw an **IOException** on error conditions. Java 2 adds an inner class to **ObjectOutputStream** called **Putfield**. It facilitated the writing of persistent fields and its use is beyond the scope of this book.

Method	Description
void close()	Closes the invoking stream. Further write attempts will generate an **IOException**
void flush()	Finalizes the output state so that any buffers are closed. That is, it flushes the output buffers.

void write(byte buffer[])	Writes an array of bytes to the invoking stream.
void write(byte buffer[],int offset, int numbytes)	Writes a subrange of numBytes bytes from the array buffer, beginning at buffer[offset].
void write(int b)	Writes a single **byte** to the invoking stream. The byte written is the low-order byte of b.
void writeBoolean(boolean b)	Writes a **boolean** to the invoking stream.
void writeByte(int b)	Writes a **byte** to the invoking stream. The byte written is the low-order byte of b.
void writeBytes(String s)	Writes the bytes repersenting str to the invoking stream.
void writeChar(int c)	Writes a **char** to the invoking stream.
void writeChars(String s)	Writes the charcters in str to the invoking stream.

ObjectInput

The ObjectInput interface extends the **DataInput** interface and defines the methods. It support object serialization. Note especially the **readObject()** method. This is called to deserialize an object. All of these methods will throw an **IOException** on error conditions.

Method	Description
int avilable()	Returns the number of bytes that are now available in the input buffer.
void close()	Closes the invoking stream. Further read attempts will generate an **IOException.**

int read()	Returns an integer representation of the next avialable byte of input. –1 is returned when the end of the file is encountered.
long skip(long numBytes)	Ignores, numBytes bytes in the invoking stream, returning the number of bytes actually ignored.

ObjectInputStream

The **ObjectInputStream** class extends the **InputStream** class and implements the **ObjectInput** interface . **ObjectInputStream** is responsible for reading objects from a stream. The constructor of this class is ObjectInputStream(InputStream inStream) throws IOException , StreamCorruptedException.

The argument inStream is the input stream from which serialized objects should be read. The most commonly used methods in this class are shown in Table. They will throw an **IOException** on error conditions. Java 2 adds an inner class to **ObjectInputStream** called **GetField**. It facilitates the reading of persistent fields and its use is beyond the scope of this book. Also, the method readLine() was deprecated by Java 2 and should no longer be used.

Method	**Description**
int available()	Returns the number of bytes that are now available in the input buffer.
void close()	Closes the invoking stream. Further read attempts will generate an IOException.
int read()	Returns an integer representation of the next available byte of input. -1 is returned when the end of the file is encountered.
Boolean readBoolean()	Reads and returns a **boolean** from the invoking stream.

A Serialization Example

The following program illustrates how to use object serialization and deserialization. It begins by instantiating an object of class **MyClass**. This object has three instance variables that are of types **String, int,** and **double**. This is the information we want to save and restore.

A **FileOutputStrem** is created that refers to a file named "serial" , and an **ObjectOutputStream** is created for that file stream. The **writeObject()** method of **ObjectOutputStream** is then used to serialize our object. The object output stream is flushed and closed.

A **FileInputStream** is then created that refers to the file named "serial" ,and an **ObjectInputStream** is created for that file stream. The **readObject()** method of **ObjectInputStream** is then used to deserialize our object. The object input stream is then closed.

Note that **MyClass** is defined to implement the **Serializable** interface. If this is not done, a **NotSerializableException** is thrown. Try experimenting with this program by declaring some of the **MyClass** instance variables to be **transient**. That data is then not saved during serialization.

```
Import java.io.*;
public class SerializationDemo{
public static void main( String args[]){|
//Object serialization
try{
MyClass Object1 = new MyClass("Hello",-7,2.7e10);
System.out.println("object1: " +object1);
FileOutputStream fos = new FileOutputStream("serial");
objectOutputSteam oos = new ObjectOutputStream(fos);
oos.writeObject(object1);
oos.flush();
oos.close();
}
catch(Exception e) {
System.out.println("Exception during serialization:"+e);
System.exit(0);
}
//object deserialization
try{
MyClass object2;
FileInputStream fis = new FileInputStream("serial");
ObjectInputStream ois = new ObjectInputStream(fis);
Object 2 = (MyClass)ois.readobject();
ois.close();
System.out.println("object2:"+object2);
}
catch(Exception e){
System.out.println ("Exception during deserialization:" +e);
System.exit(0);
}
}
}

class MyClass implements Serializabele{
String s;
int I;
double d;
public MyClass(String s, int i,double d){
this.s = s;
this.i = i;
this.d= d;
}
public String toString(){
return "s=" +s "; i ="+ i +"; d=" +d;
}
}
```

This program demonstrates that the instance variables of **object1** and **object2** are identical. The output is shown here:

```
Object 1: s=Hello; I=-7; d=2.7E10
Object2: s=Hello; I=-7; d=2 .7E10
```

20.7 Short Summary

- A stream is an abstraction that either produces or consumes information.
- A file objects is used to obtain or manipulate the information associated with a disk file, such as the permissions, time, date, and directory path, and to navigate subdirectory hierarchies.
- Serialization is the process of writing the state of an object to a byte stream.
- The Java facilities for serialization and deserialization have been designed so that much of the work to save and restore the state of an object occurs automatically.

20.8 Brain Storm

1. What is meant by Streams?
2. What are the two types of Streams? Explain Briefly.
3. What is the use of File Class?
4. Explain the process Called Serialization.

Lecture - 21

Applets & Applications

Objectives

In this lecture you will learn the following

- Java Applets
- Java Applications

Lecture - 21

21.1 Snap Shot

In this lecture you will learn about Java Applets and Java Applications.

21.2 Types of Java programs

Java language is mainly used for internet programming. The purpose of programming has completely changed because of internet. Formerly, a program was written to input a few numbers and process them in a very complex way and output the results. In data processing a program is written to read one or more data files and produce a report or a ticket, now, these are the days of advertisements. Programs are written for presentation of certain ideas or company profile, advertisement or notifications. Using HTML we can create static pages can be created in Java. All graphical user input interface windows can be designed in Java. Sound, animation and all Mutlimedia elements dance in the small screen. A small intelligent dynamic program that can also be used for writing ordinary application programs.

Java Programs

Application programs and Applet Programs

There are two types of Java programs. Application programs and applet programs, Java applets are used in Internet applications. It may present your company profile, notification advertisement or anything. The Java applet will be placed on the web by embedding into a HTML file. Any internet user can click your applet and run your applet on your web.

21.3 Types of Java programs

Before concluding this chapter let use see how Java programs look like we have already seen that it looks almost like a C++ program.

- Java is case sensitive. It differentiates between lower and upper case letters. For example pay, Pay and PAY are considered as three different variable names in Java.
- Every Java statement must end with a semi colon. In one line we normally write only one statement and it ends with a semi colon.
- Compound Sentences can be formed with braces{}.

Let us first see a small Java program, which prints the message "Radiant welcomes you to Java world".

```
class welcomes
{
public static void main(String a[])
{
System.out.println("welcomes you to java world");
}
}
```

as we have already mentioned, Java is an object oriented programming language. So, every program defines a class. Here we have defined a class named welcomes.

21.4 Java Applets

In this section let us see how to create a Java applet. Any Java applet can be embedded into a HTML file and a dynamic web page can be designed. When we develop an applet, we must import two important Java groups. They are

Java.awt
Java.applet

Java.awt is called the Java abstract windows tool kit. Java.applet is the applet group. We must import all the classes in these two groups as follows:

```
Import java.awt.*;
Import java.applet.*;
```

In the above statements the asterix symbol(*) is used as wild card.

Let us write a small applet to display the message,

Welcomes you to the java world

The names of the class is welcome applet , which is a subclass of applet class the following is the java code for the applet.

```
Import java.awt.*;
Import java.applet.*;
public class welcomeapplet extends Applet
{
public void paint(Graphics g)
{
g.drawString("welcomes you to the java world " , 20,20);
}
}
```

Let us see how different this is from out first program class welcomes. The import statements have already been explained. The statement, public class welcomesapplet extends Applet declares that welcomesApplet is a new class which is a subclass of Applet class. This means that welcomesApllet ia an applet. For this class we define a method called paint which has only one statement. The statement g.drawString("welcomes you to the java world",20,20) tells us that the message welcomes you to the java world will be printed at the position 20,20 Graphics is a class which is avialable in java and drawString() is a behaviour of Graphics class.

When a java applet is ready we must first compile it using the compiler javac. For our program we complie using the command. Javac welcomes applet .java.

When the compilation is over, the class file is ready and so it can be embedded in an HTML file.

Before embedded in a HTML. File, we can verify the output the output using a program called applet viewer. We embed the applet in a HTML page and then open it with an internet browser. For example we can embed the above applet welcomes applet in a HTML page as follows.

```
<html>
<head>
<title>
Embedding an applet in a HTML document
</title>
<body>
<applet code= welcomesapplet.class width=300 height=100>
</applet>
</body>
</html>
```

when we open this HTML document using Internet explorer we get the window with welcome message.

21.5 Java Applets and Applications

Java can be used two types of programs: applications and applets. An application is a program that runs on your computer, under the operating system of that computer. That is, an application created by java is more or less like one created using C or C++. When used to create applications, java is not much different from any other computer language. Rather, it is Java's ability to create applets that makes it important. An applet is an application designed to be transmitted over the Internet and executed by a Java-compatible Web browser. An applet is actually a tiny Java program, dynamically downloaded across the network, just like an image, sound file, or video clip. The important difference is that an applet is an intelligent program, not just an animation or media file. In other words, an applet is a program that can react to user input and dynamically change-not just run the same animation or sound over and over.

An exiting as applets are, they would be nothing more than wishful thinking if java were not do address the two fundamental problems associated with them: security and portability. Before continuing, let's define what these two terms mean relative to the Internet.

21.6 Short Summary

- Java language is mainly used for internet programming.
- An applet is an application designed to be transmitted over the Internet and executed by a Java-compatible Web browser.
- Any Java applet can be embedded into a HTML file and a dynamic web page can be designed.

21.7 Brain Storm

1. What are the two types of Programs?
2. Explain the Java Application Program.
3. Explain the Java Applet Program.

=

Lecture - 22

Multithreading & Multitasking

Objectives

In this lecture you will learn the following

- ❖ Multi Threading
- ❖ Multi Tasking

Lecture - 22

22.1 Snap Shot

In this lecture you will learn about the concept of Multi Threading, Multi Tasking and its differences

22.2 Using Multithreading

If you are like most programmers, having multithreaded support built into the language will be new to you. The key to utilizing this support effectively is to think concurrently rather than serially. For example, when you have two subsystems within a program that can execute concurrently, make them individual threads. With the careful use for multithreading, you can create very efficient programs. A word of caution is in order, however, if you create too many threads, you can actually degrade the performance of your program rather than enhance it. Remember that some overhead is associated with context switching. If you create too many threads, more CPU time will be spent changing contexts than executing your program!

Java provides built-in support for multithreaded programming. A multithread program contains two or more parts that can run concurrently. Each part of such a program is called a thread, and each thread defines a separate path of execution. Thus, multithreading is a specialized form of multitasking.

22.3 Multi Tasking

You are almost certainly acquainted with multitasking, because it it's supported by virtually all modern operating systems. However, there are two distinct types of multitasking: process-based and thread-based. It is important to understand the difference between the two. For most readers, process-based multitasking is the more familiar form. A process is, in essence, a program that is executing. Thus process-based multitasking is the feature that allows your computer to run two or more programs concurrently. For example, process-based multitasking enables you to run the Java compiler at the same time that you are using a text editor. In process-based multitasking, a program is the smallest unit of code that can be dispatched by the scheduler.

In a thread-based multitasking environment, the thread is the smallest unit of dispatchable code. This means that a single program can perform two or more tasks simultaneously. For instance, a text editor can format text at the same time that it is printing, as long as these two actions are being performed by two separate threads.

Thus, process-based multitasking deals with the "big picture", and the thread-based multitasking handles the details.

Multitasking threads require less overheads than multitasking processes. Processes are heavy weight tasks that require their own separate address space. Inter-process communication is expensive and limited. Context switching from one process to another is also costly. Threads, on the other hand, are lightweight. They share the same address space and cooperatively share the same heavyweight process. Inter thread communication is inexpensive, and context switching from one thread to the next is low cost. While Java programs make use of process-based multitasking environments, process-based multitasking is not under the control of Java. However, multithreaded multitasking is.

Multithreading enables you to write very efficient programs that make maximum use of the CPU, because idle time can be kept to a minimum. This is especially important for the interactive, networked environment in which Java operates, because idle time is common. For example, the transmission rate of data over a network is much slower than the rate at which the computer can process it. Even local file system resources are read and written at a much slower pace than they can be processed by the CPU. And, of course, user input is much slower than the computer. In a traditional, single-threaded environment your program has to wait for each of these tasks to finish before it can proceed to the next one - even though the CPU is sitting idle most of the time. Multithreading lets you gain access to this idle time and put it to good use.

If you have programmed for operating systems such as Windows 98 or Windows NT, then you are already familiar with multithreaded programming. However, the fact that Java manages threads makes multithreading especially convenient, because many of the details are handled for you.

22.4 Multithreading Vs Multitasking

A thread is the smallest unit of execution that the system can schedule to run; a path of execution through a process. Each thread consists of a stack, an instruction pointer, the CPU state , and an entry in the system's scheduler list. A thread may be blocked , scheduled to execute, or executing.

Threads communicate by sending messages to each other, and they compete for ownership of various semaphores, which govern the allocation of computing resources between the individual threads. The threads ask the system for an instruction to carry out. If no instruction is ready, the thread is suspended until it has something to do. If an instruction is ready, the thread performs the task and makes another request to the system for work.

Older operating systems achieve multitasking by creating multiple processes, which creates a great deal of overhead. In a multithreaded environment, a process is broken into independent executable tasks (threads). These threads then collectively perform all the work that a single program could execute, allowing applications to perform many tasks simultaneously. The separate threads complete their tasks in the background and allow continued operation of the primary assignment. The challenge is to break the application up into discrete tasks that can become threads.

An ice cream parlor is an example of a multithreaded process. As demand increases, more counter help is added. Each additional person shares the floor space and the equipment (the ice cream , the cones and dishes , the scoops, the cash register.) In an environment that is not multithreaded , each additional person would have their own equipment and floor space. At some point, even though shared resources are being used, it may make sense to add a whole new environment to service the additional demand. Hence, a new ice cream parlor opens up one mile away. In IS terms, a larger server machine is added to the environment.

Tightly coupled processes that execute concurrently require programmers to push problem abstraction further than they have in the past. A thread of execution is a new conceptual unit that performs the work in the system by moving from one instruction or statement (thread) to the next, executing each in turn.

The greatest adjustment to multitasking may be in user's work habits. Users are accustomed to taking a break or starting at the screen after issuing a command. Under multithreading, users need to adjust to the idea that they don't have to wait after issuing a command- they can switch to another task.

22.5 Short Summary

- A multithread program contains two or more parts that can run concurrently.
- Processes are heavy weight tasks that require their own separate address space.
- In a multithreaded environment, a process is broken into independent executable tasks (threads).

22.6 Brain Storm

1. What is Multi Threading?

2. What is Multi Tasking?

3. What is the difference between process-based and thread-based Multi tasking?

4. What is the difference between Multi Tasking & Multi Threading?

Lecture - 23

Working with Threads

Objectives

In this lecture you will learn the following

- Knowing Models in Threads
- Creating Thread

Lecture - 23

23.1 Snap Shot

In this lecture you will learn about Java Threading Model , Creating a Thread and about Extending Thread.

23.2 Thread

When a Java program starts up, one thread begins running immediately. This is usually called the main thread of your program, because it is the one that is executed when your program begins. The main thread is important for two reasons.

- It is the thread from which other "child" threads will be spawned.
- It must be the last thread to finish execution. When the main thread strops, your program terminates.

Although the main thread is created automatically when your program is started, it can be controlled through a **Thread** object. To do so, you must obtain a reference to it by calling the method current Thread ( ), which is a **public static** member of **Thread**. Its general form is shown here:

```
static Thread currentThread ( )
```

This method returns a reference to the thread in which it is called. Once you have a reference to the main thread, you can control it just like any other thread.

Let's begin by reviewing the following example:

```
 / / Controlling the main Thread.
    class CurrentThreadDemo {
      public static void main (String args [ ]  ) {
         Thread t = Thread. currentThread ( );

System.out.println ("Current thread :" + t);

    / /  change the name of the thread
      t.setName ("My Thread");
    System.out.println ("After name change:" + t);
     try  {
       for (int n = 5; n> 0; n - -) {
         System.out.println (n) ;
       Thread.sleep (1000);
    }
```

```
        }  catch (InterruptedException,e)  {
               System.out.println ("Main thread interrrupted");
                 }
           }
        }
```

In this program, a reference, to the current thread (the main thread, in this case) is obtained by calling **currentThread ( ),** and this reference is stored in the local variable t. Next, the program dispalys information about the thread. The program then calls s**etName( )** to change the internal name of the thread. Information about the thread is then redisplayed. Next, a loop counts down from five, pausing one second between each line. The pause is accomplished by the **sleep( )** method. The argument to **sleep( )** specifies the delay period in milliseconds. Notice the try/catch block around this loop. The **sleep ( )** method in **Thread** might throw an **Interrupted Exception**. This would happen if some other thread wanted to interrupt this sleeping one. This example just prints a message if it gets interrupted. In a real program, you would need to handle this differently. Here is the output generated by this program.

```
Current thread:  Thread [main, 5 main]

 After name change: Thread [ My Thread, 5,main]
     5
     4
     3
     2
     1
```

Notice the output produced when it is used as an argument to **println( ).** This displays, in order the name of the thread, its priority and the name of its group. By default, the name of the main thread is **main**. Its priority is 5, which is the default value, and **main** is also the name of the group of threads to which this thread belongs. A thread group is a data structure that controls the state of a collection of threads as a whole. This process is managed by the particular run-time environment and is not discussed in detail here. After the name of the thread is changed, **t** is again output. This time, the new name of the thread is displayed.

Let's look more closely at the methods defined by **Thread** that are used in the program. The **sleep( )** method causes the thread from which it is called to suspend execution for the specified period of milliseconds. Its general form is shown here:

```
static void sleep(long milliseconds) throws InterruptedException
```

The number of millisecond to suspend is specified in milliseconds. This methods may throw an **Interrupted Exception**.

The **sleep( )** method has a second form, shown next, which allows you to specify the period in terms of millisecond and nanoseconds.

static void sleep (long millliseconds, int nanoseconds) throws Interrupted Exception.

This second form is useful only in environment that allow timing periods as short as nanoseconds.

As the preceding program shows, you can set the name of a thread by using **setName ( )**. You can obtain the name of a thread by calling **getName( )** (but note that this procedure is not shown in the program..) These methods are members of the **Thread** class and are declared like this:

```
final void setName (String threadName)
final String getName ( )
```

Here, threadName specifies the name of the thread.

23.3 The Java Thread Model

The Java a run time system depends on threads for many things, and all the class libraries are designed with multithreading in mind. In fact, Java uses threads to enable the entire environment to be asynchronous. This helps reduce inefficiency by preventing the waste of CPU cycles.

The value of a multithreaded environment is best understood in constrict to its counterpart. Single threaded systems used an approach called an event loop with polling. In this models a single thread of control run in an infinite loop, polling a single event queue to decide what to do next. Once this polling mechanism returns with, say a signal that a network file is ready to be read, then the event loop dispatches control to the appropriate event handler. Until this event handler returns, nothing else can happen in the system . This wastes CPU time. It can also result in one part of a program general, in a singled threaded environment, when a thread blocks (that is, suspends execution) because it is waiting for some resource, the entire program stops running.

The benefit of Java's multithreading is that the main loop/polling mechanism is eliminated. One thread can pause without stopping other parts of your program. For example , the idle time created when a thread reads data from a network or waits for user input can be utilized elsewhere. Multithreading allows animation loops to sleep for a second between each frame without causing the whole system to pause. When a thread blocks in a Java program, only the single thread that is blocked pauses. All other threads continue to run.

23.4 Creating a Thread

In the most general sense, you create a thread by instantiating an object of type **Thread**. Java defines two ways in which this can be accomplished:

- You can implement the **Runnable** interface.
- You can extend the **Thread** class, itself.

The following two sections look at each method, in turn.

Implementing Runnable

The easiest way to create a thread is to create a class that implements the **Runnable** interface. **Runnable** abstracts a unit of executable code. You can construct a thread on any object that implements **Runnable**. To implement **Runnable**, a class need only implement a single method called run ( ) , which is declared like this:

```
public void run ( )
```

Inside **run( )** you will define the code that constitutes the new thread . It is important to understand that **run( )** can call other methods, use other classes , and declare variables, just like the main thread can. The only difference is that **run( )** establishes the entry point for another, concurrent thread of execution within your program. This thread will end when **run( )** returns.

After you create a class that implements **Runnable**, you will instantiated an object of type **Thread** from within that class. **Thread** defines several constructors. The one that we will use is shown here:

Thread (RunnableOb,String *threadName*)

In this constructor, threadOb is an instance of a class that implements the **Runnable** interface. This defines where execution of the thread will begin. The name of the new thread is specified by *threadName.*

After the new thread is created, it will not start running until you call its start ( ) method, which is declared within **Thread**. In essence, **start**. ( ) .executes a call to **run ( )**

The **start( )** method is shown here:

```
void start ( )
```

Here is an example that creates a new thread and starts it running.

```
      / / Create a second thread.
          class NewThread implements Runnable  {
               Thread t ;
     NewThread ( ) {
      / / Create a new, second thread
       t = new Thread (this, "Demo Thread");
       System.out.println ("Child thread:" + t);
      t. start ( ); / / Start the thread
}
   / / This is the entry point for the second thread.
   public void run ( ) {
     try {
        for (int i=  5; i> 0; i - - ) {
            System.out.println ("Child Thread:" + i);
                 Thread.sleep (500) ;
 }
 } catch (InterruptedException  e)  {
      System.out.println ("Chiled intrrupted".);
}
       System.out.println ("Exiting child thread.");
   }
}

class ThreadDemo {
public static void main(String args [ ] ) {
 new NewThread ( ); / / create a new thread

     try {
        for( int i=  5 ; i> 0;i - - ) {
       system.out.println("Main Thread:" + i);
          Thread.sleep (1000);
      }
}catch (InterrruptedExpection   e ) {
 System.out.println ("Main thread interrupted.");
      }

System.out.println ("Main thread exiting.");
```

```
  }
}
```

Inside **NewThread's** Constructor, a new **Thread** object is created by the following statement.

```
t= new thread (this, "Demo Thread");
```

Passing **this** as the first argument indicates that you want the new thread to call the **run ( )** method on **this** object. Next, **start ( )** is called, which starts the thread of execution beginning at the **run( )** method. This causes the child thread's for loop to being . After calling **start ( )**, **NewThread's** constructor returns to **main( ).** When the main thread resumes, it enters its **for** loop. Both threads continue running sharing the CPU until their loops finish. The output produces by this program is as follows:

```
Child thread: thread[demo thread ,5 ,main]
Main Thread: 5
Child  Thread : 5
Child Thread : 4
Main Thread : 4
Child Thread : 3
Child Thread : 2
Main thread : 3
Child Thread : 1
Exiting child thread.
Main thread: 2
Main Thread :1
Main thread exiting.
```

As mentioned earlier, in a multithreaded program, the main thread must be the last thread to finish running . If the main thread finished before a child thread has completed, then the Java run-time system may "hang". The preceding program ensures that the main thread finished last because the main thread sleeps for 1,000 milliseconds between iteration's but the child thread sleeps for only 500 millisecond. This causes the child thread to terminate earlier than the main thread. Shortly, you will see a better way to ensure that the main thread finishes last.

23.5 Extending Thread

The second way to create a thread is to create a new class that extends **Thread**, and then to create an instance of that class. That extending class must override the **run ( )** method, which is the entry point for the new thread. It must also call **start ( )** to begin execution of the new thread. Here is the preceding program rewritten to extend **Thread**.

```
1.  Create a second thread by extending thread
2.  Class NewThread extends Thread {
3.  NewThread(){
      / / Create a new, second thread
      super (“Demo Thread”);
       System.out.println (“Child thread:” + this );
        start ( ); / / start the thread

          }

      / / This is the entry point for the second thread.
       public void run ( ) {
        try {
          for (int i=5; i > 0; i - -) {
            System.out.println (“Child Thread:” + i);
                  Thread.sleep (500);
 }
        }catch (InterrruptedException  e ) {
           System.out.println (“Child interrupted.”);
              }
         System.out.println (“Exiting child thread,”);
        }
}

class extendThread  {
  public static void main (String args [ ] ) {
        new NewThread ( ); / / create a new thread

      try {
      for (int i =5; i> 0; i - -){
           System.out.println (“Main Thread:”  + i);
             Thread.sleep (1000);
          }
     }catch (InterruptedException  e ) {
      System.out.println (“Main thread interrupted.”);
 }
System.out.println (“Main thread exiting .”);
  }
}
```

This program generates the same output as the preceding version. As you can see, the child thread is created by instantiating an object of **NewThread**, which is derived from **Thread**.

Notice the call to **super ( )** inside **NewThread**. This invokes the following form of the **Thread** constructor:
public Thread (String threadName)

Here, threadName specifies the name of the thread.

23.6 The Thread Class and the Runnable Interface

Java's multithreading system is built upon the **Thread** class, its methods and its companion interfaces, **Runnable, Thread** encapsulated a thread of execution. Since you can't directly refer to the ethereal state of a running thread, you will deal with it through its proxy, the **Thread** instance that spawned it. To create a new thread, your program will either extend **Thread** implement the Runnable interface.

Method	Meaning
getName	Obtain a thread's name
getPriority	Obtain a thread's priority.
isAlive	Determine if a thread is still running.
join	Wait for a thread to terminate
run	Entry point for the thread
sleep	Suspend a thread for a period of time
start	Start a thread by calling its run method.

Thus far, all the examples in this book have used a single thread of execution. The remainder of this chapter explains how to use **Thread** and **Runnable** to create and manage threads, beginning with the one thread that all Java programs have: the main thread.

23.7 Override

In a class hierarchy, when a method in a subclass has the same name and type signature as a method in its superclass , then the method in the subclass, subclass is said to override the method in the superclass. When an overridden method is called from within a subclass, it will always refer to the version of that method defined by the superclass will be hidden consider the following:

```
// method overriding
class A
 {
int i,j;
A(int a, int b) {
I=a;
J=b;
}
//display i and j
void show() {
System.out.println(“i and j: “ +i “ “ + j);
}
}
class B extends A{
 int k;
B(int a, int b, int c) {
super(a,b);
k=c;
}
// display k – this overrides show() in a
void show() {
System.out.println(“k: “+k);
}
}
class override {
public static void main(String a[]) {
B subOb = new b(1,2,3);
SubOb.show();
}
}
```

The output of this program is

```
K: 3
```

23.8 Short Summary

- The sleep ( ) method causes the thread from which it is called to suspend execution for the specified period of milliseconds.

- The easiest way to create a thread is to create a class that implements the Runnable interface.

- In a class hierarchy, when a method in a subclass has the same name and type signature as a method in its superclass , then the method in the subclass, subclass is said to override the method in the superclass.

23.9 Brain Storm

1. What is threading?
2. What are the methods we used for creating thread?
3. Explain about Java Thread Model?
4. What is Override?

Lecture - 24

Thread States & Priorities

Objectives

In this lecture you will learn the following

- Thread States
- Thread Priorities

Lecture - 24

24.1 Snap Shot

In this lecture you will learn about Thread States, Thread Priorities and about Rules of Context Switch.

24.2 Threads States

Sometimes, suspending execution of a thread is useful. For example, separate thread can be used to display the time of day. If the user doesn't want a clock, then its thread can be suspended. Whatever the case, suspending a thread is a simple matter. Once suspended, restarting the thread is also a simple matter.

The mechanisms to suspend, stop, and resume threads difference between Java 2 and earlier versions. Although you should use the Java2 approach for all new code, you still need to understand how these operations were accomplished for earlier Java environment. For example you may need to update or maintain older, legacy code. You also need to understand why a change was made for Java2. For these reasons, the next section describes the original way that the execution of a thread was controlled, followed by a section that describes the new approach required for Java2.

Prior to Java 2, a program used **suspend ( )** and **resume ( ),** which are methods defined by **Thread,** to pause and restart the execution of a thread. They have the form shown below:

```
final void suspend ( )
final void resume ( )
```

The following program demonstrates these methods.

```
/ / Using suspend  ( ) and resume ( ).
class NewThread implements Runnable {
Straing name; / / name of thread
Thread t;

NewThread(String threadname)  {
name = threadname;
t = new Thread (this , name);
```

```
            System.out.println (“New thread:”  + t );
            t.start ( ); / / Start the thread
      }

            / / This is the entry point for thread
            public void run ( ) {
               try {
            for (int i = 15; i>0; i - -) {
              System .out.println(name + “:” + i);
            Thread.sleep (200);
      }
          } catch (InterruptedException e ) {
            System.out.println (name + “interrupted.”);
            }

            System.out.println (name + “exiting.”);
            }
      }

      class SuspendResume  {
        public static void main ( String args [ ]) {
      NewThread ob1= new NewThread (“One”);
      NewThread ob2 = new NewThread (“Two”);
      try {
        Thread.sleep (1000);
        obl.t.suspend ( );
        System.out.println (“Suspending thread One”);
      Thread.sleep (1000);
      obl.t.resume ( );
      System.out.println(“Resuming thread one”);
      ob2.t.suspend( );
```

```
System.out.println(“Suspending thread Two”);
Thread.sleep (1000);
ob2.t.resume ( ) ;
System.out.println (“Resuming thread Tow);
}catch (InterruptedException  e)   {
System.out.println(“Main thread Interrupted”);
}
/  / wait for thread to finish
try    {
         System.out.println (“Waiting for threads to finish.”);
          ob1.t.join ( );
           ob2.t.join( );
          } catch (InterruptedException e ) {
              System.out.println (“Main thread Interrupted”);
 }
           System.out.println (“Main thread exiting.”);
         }

}
```

Sample output from this program is shown here:

```
New thread: Thread [One,5,main]
One: 15
New thread: Thread[Tow,5 main]
Two: 15
One : 14
Two : 14
One : 13
Two: 13
One : 12
```

```
          Two : 12
          One :  11
          Two :  11
Suspending thread One
          Two:  10
          Two :  9
          Two :  8
          Two : 7
          Two : 6
Resuming thread One
Suspending thread Two
          One :  10
          One :   9
          One :  8
          One :  7
          One :  6

Resuming thread Two
Waiting for threads to finish
          Two : 5
          One:  5
          Two : 4
          One:  4
          Two : 3
          One :  3
          Two :  2
          Two                      :                                    2
          Two :  1
          One:  1
Two exiting.
One exiting.
```

Main thread exiting.

The **Thread** class also defines a method called **stop( )** that stops a thread . Its signature is shown here:

```
        void stop ( )
```

Once a thread has been stopped, it cannot be restarted using **resume ( ).**

Suspending resuming and stopping threads using java 2

While the **suspend ( ), resume ( ), and stop( )** methods defined by **Thread** seem to be a perfectly reasonable and convenient approach to managing the execution of threads, they must not be used for new Java programs. Here's why. The **suspend( )** method of the **Thread** class is deprecated in Java 2. This was done because **suspend( )** can sometimes cause serious system failures. Assume that a thread has obtained locks on critical data structures. If that thread us suspended at that point, those locks are not relinquished. Other threads that may bewailing for those resources can be deadlocked.

The **resume( )** methods is also deprecated. It does not cause problems, but cannot be used without the **suspend( )** method as its counterpart.

The **stop( )** methods of the **Thread** class, too, is deprecated in Java 2 This was done because this methods can sometimes cause serious system failures. Assume that a thread is writing to a critically important data structure and has completed only part of its changes. If thread is stopped at that point, that data structure might be left in a corrupted state.

Because you can't use the **suspend( ) resume( ),** or **stop( )** methods in Java 2 to control a thread, you might be thinking that no way exists to pause, restart, or terminate a thread. But, fortunately, this is not true. Instead, a threads must be designed so that the **run( )** method periodically checks to determine whether that thread should suspend, resume, or stop its own execution. Typically, this is accomplished by establishing a flag variable that indicates the execution state of the thread. As long as this flag is set to "running", the **run( )** method must continue to let the thread execute. If this variable is set to "suspend", the thread must pause. If it is set to "stop", the thread must terminate. Of course, a variety, of ways exist in which to write such code, but the central theme will be the same for all programs.

The following example illustrates how the **wait ( )** and **notify( )** methods that are inherited from **Object** can be used to control the execution of a thread, This example is similar to the program in the previous section. However, the deprecated methods calls have been removed. Let us consider the operation of this program.

The **NewThread** class contains a **boolean** instance variable named **supendFlag**, which is used to control the execution of the thread. It is initialized to false by the constuctor. The **run( )** method contains a **synchronized** statement block that checks **suspendFlag**. IF that variable is **true**, the **wait ( )** method is invoked to suspend the execution of the thread. The **mysuspend ( )** method sets **suspendFlag** to **true**. The **myresume ( )** method set **suspendFlag** to **false** and invokes **notify ( )** to wake up the thread. Finally, the **main ( )** method has been modified to invoke the my suspend() and ,**myresume ( )** methods.

```
// Suspending and resuming a thread for java 2
class NewThread implements Runnable
{
String n;
Thread t;
boolean s;
NewThread(String tn) {
n= threadname;
t=new Thread(this, n);
System.out.println("New thread: " +t);
s=false;
t.start();
}
//This is the entry point for thread.
public void run() {
try{
for(int i= 15; i> 0; i--){
System.out.println(name " ";"+i);
Thread.sleep(200);
synchronized(this){
while(s) {
wait();
}
}
}
}catch(InterruptedException e)
System.out.println(name+” interrupted.”);
}
System.out.println(name+” exiting.”);
}
void mysuspend(){
s=true;
}
synchornized void myresume() {
s=false;
notify();
}
}
class SuspendResume {
public static void main(String a[])
 NewThread o1 = new NewThread(“one”);
 NewThread o2 = new NewThread(“Two”);
try {
Thread.sleep(1000);
o1.mysuspend();
System.out.println(“Suspending thread One”);
Thread.sleep(1000);
o1.myresume();
System.out.println(“Resuming thread One”);
o2.mysuspend();
```

```
System.out.println("Suspending thread Two");
Thread.sleep(1000);
o2.myresume();
System.out.println("Resuming thread Two");
} catch(InterruptedException e) {
System.out.println("Main thread Interrupted");
}
// wait fot thread to finish
try {
System.out.println("waiting for threads to finish");
o1.t.join();
o2.t.join();
}catch(InterruptedException e) {
System.out.println("Main thread interrupted");
}
System.out.println("Main thread Exiting");
}
}
```

Although this mechanism isn't as "clean" as the previous way, nevertheless, it is the way required to ensure that run-time errors don't occur. It is the approach that must be used for all new code.

24.3 Thread Priorities

Java assigns to each thread a priority that determines how that thread should be treated with respect to the others. Thread priorities are integers that specified the relative priority of one thread to another. As an absolute value, a priority is meaningless; a higher-priority thread doesn't run any faster than a lower-priority thread if it is the only thread running. Instead, a thread's priority is used to decide when to switch from one running thread to the next. This is called a context switch. The rules that determine when a content switch takes place are simple:

The Thread scheduler to decide when each thread should be allowed to run uses thread priorities. In theory, higher priority threads get more CPU time than lower priority threads. In practice, the mount of CPU time that a thread gets often depends on several factors besides its priority. (For example, how an operating system implements multitasking can affect the relative availability of CPU time.) A higher-priority thread can also preempt a lower priority one. For instance, when a lower priority thread is running and a higher priority thread resumes (from sleeping or waiting on I/O, for example,) it will preempt the lower priority thread.

In theory, threads of equal priority should get equal; access to the CPU. But you need to be careful. Remember, Java is designed to work in a wide range of environments. Some of those environments implements multitasking fundamentally differently than others. For safety, threads that share the same priority should yield control once in a while. This ensures that all threads have a chance to run under a non preemptive operating system. In practice, even in non preemptive environments, blocking situation, such as waiting for I/O . When this happens, the booked thread is suspended and other threads can run. But, if you want smooth multithreaded execution, you are better off not relying on this. Also, some types of tasks are CPU intensive . Such threads, dominate the CPU. For these types of threads, you want to yield control occasionally, so that other threads can run.

To set a thread's priority, use the **setPriority ( )** methods, which is a member of Thread. This is its general form.

```
final void setPriority (int level)
```

Here ,level specifies the new priority setting for the calling thread. The value of level must be within the range **MIN_PRIORITY** and **MAX_PRIORITY**. Currently, these values are 1 and 10 respectively. To return a thread to default priority, specify NORM_PRIORTY, which is currently 5. These priorities are defined as final variables within **Thread.**

You can obtain the current priority setting by calling the **getPriority ( )** method of **Thread**, shown here:

```
final int getPriority ( )
```

Implementations of Java may have radically different behavior when it comes to scheduling . The Windows 95/98/NT version works, more or less, as you would expect. However other versions may work quite differently. Most of the inconsistencies arias when you have threads that are relying on preemptive behavior, instead of cooperatively giving up CPU time. The safest way to obtain predictable, cross platform behavior with Java is to use threads that voluntarily give up control of the CPU.

The following example demonstrates two threads at different priorities, which do not run on a preemptive platform in the same way as they run on a nonpreemptive platform. One thread is set two levels above the normal priority, as defined by **Thread. NORM_PRIORITY**, and the other is set to two levels below. The threads are started and allowed to run for ten seconds. Each thread executes a loop, counting the number of interruptions. After ten seconds, the main thread stops both threads. The number of times that each thread made it through the loop is then displayed.

```
//Demonstrate thread priorities.
class clicker implements runnable {
int  c=0;
thread t;
private volatitle boolean running=true;
public clicker(int p) {
t=new thread(this);
t.setPriority(p);
}
public void run()
{
while(running) {
c++;
}
}
public void stop()
running=false;
public void start()
{ t.start();
}
}
class HiLoPri {
```

```
public static void main(String a[]) {
Thread.currentThread().setPriority(Thread.MAX_PRIORITY);
clicker h = new clicker(Thread.NORM_PRIORITY+ 2);
clicker l = new clicker(Thread.NORM_PRIORITY-2);
l.start();
h.strat();
try {
Thread.sleep(10000);
}
catch(InterruptedException e) {
System.out.println("Main thread interrupted.");
}
l.stop();
h.stop();
// wait for child threads to terminate.
try {
h.t.join();
l.t.join();
}catch(InterruptedException e)
{
System.out.println("InterruptedException caught.");
}
System.out.println("Low-priority thread " + l.c);
System.out.println("high-priority thread " + h.c);
}
}
```

The output of this program, shown as follows when run under windows 98, indicates that the threads did context switch , even though neither voluntarily yielded the CPU nor blocked for I/O. The higher-priority thread got approximately 90 percent of the CPU time.

```
Low-priority threads: 4408112
High-priority thread: 589626904
```

Of course, the exact output produced by this program depends on the speed of your CPU and the number of other tasks running in the system. When this same program is run under a nonpreemtive system, different results will be obtained, one other note about the preceding program. Notice that running is preceded by the keyword volatile. although volatile is examined more carefully , it is used here to ensure that the value of running is examined each time the following loop iterates:

```
while(running) {
c++;
}
```

without the use of volatile, Java is free to optimize the loop in such way that the value of running is held in a register of the CPU and not necessarily reexamined with each iteration. The use of volatile prevents this optimization, telling Java that running may change in ways not directly apparent in the immediate code.

24.5 Short Summary

- The mechanisms to suspend, stop, and resume threads difference between Java 2 and Java 1.

- The **Thread** class also defines a method called **stop( )** that stops a thread .

- The **resume( )** methods is also deprecated. It does not cause problems but cannot be used without the **suspend ( )** method as its counterpart.

- The **stop( )** methods of the Thread class, too, is deprecated in Java 2 This was done because this methods can sometimes cause serious system failures.

- The thread scheduler to decide when each thread should be allowed to run uses thread priorities.

24.6 Brain Storm

1. What is suspend() method?
2. What is the main difference between suspend() and stop() methods?
3. What is resume() method?
4. What is called thread priority?
5. Why we use Thread Priority?

Lecture - 25

Synchronization

Objectives

In this lecture you will learn the following

- ❖ Synchronization
- ❖ Inter-thread Communication
- ❖ Deadlock

Lecture - 25

25.1 Snap Shot

In this lecture you will learn about Synchronization , Using the Synchronized Method , Statement , Inter thread Communication and about Dead Lock.

25.2 Synchronization

When two or more threads need access to a shared resource, they need some way to ensure that the resource will be used by only one thread at a time. The process by which this is achieved is called synchronization. As you will see, Java proceeds unique, language-level support for it.

Key to synchronization is the concept of the monitor (also called a semaphore.) A monitor is an object that is used as a mutually exclusive lock, or mute. Only one thread can own monitor at a given time. When a thread acquires a lock, it is said to have entered the monitor. All other threads attempting to enter the locked monitor will be suspended until the first thread exits the monitor. These other threads are said to be waiting for the monitor. A thread that owns a monitor can reenter the same monitor if it so desires.

If you have worked with synchronization when using other languages, such a C or C ++ , you know that it can be a bit tricky to use. This is because most languages do not, themselves, support synchronization. Instead, to synchronize threads, your programs need to utilize operating system primitives. Fortunately, because Java implements synchronization through language elements, most of the complexity associated with synchronization has been eliminated.

You can synchronize your code in either of two ways. Both involve the use of the **synchronized** keyword, and both are examined here.

25.3 Using Synchronized Methods

Synchronization is easy in Java because all objects have their own implicit monitor associated with them. To enter an object's monitor, just call a method that has been modified with the synchronized keyword. While a thread is inside a synchronized method, all other threads that try to call it (or any other synchronized method) on the same instance have to wait. To exit the monitor and relinquish control of the object to the next waiting thread, the owner of the monitor simply returns from the synchronized method.

To understand the need for synchronization, let's begin with a simple example that does not use it but should. The following program has three simple classes. The first one**, Callme**, has a single method named **call ( )** . The **call ( )** method takes a String parameter called **msg.** This method tries to print the **msg** string inside of square brackets. The interesting thing to notice is that after **call ( )** prints the opening bracket and the **msg** string, it calls **Thread.Sleep (1000),**which pauses the current thread for one.

The constructor of the next class, Caller, takes a reference to an instance of the **Callme** class and a String, which are stored in **target** and **msg**, respectively. The constructor also creates a new thread that will call this objects **run ( )** method. The thread is started immediately. The **run ( )** method of **Caller** calls the **Call ( )** methods on the target instance of **Callme** , passing in the **msg** string . Finally, the Synch class starts by creating a single instance of **Callme**, and three instances of **Caller**, each with a unique message string . The same instance of **Callme** is passed to each **Caller**.

```
// This program is not synchronized.
class Callme {
void call(String m) {
System.out.println("["+m);
try {
Thread.sleep(1000);
}catch(InterruptedException e){
System.out.println("Interrupted");
}
System.out.println("]");
}
}
class Caller implements Runnable {
String msg;
Callme target;
Thread t;
public Caller(Callme t1, String s) {
tt=t1;
t=new Thread(this);
t.start();
}
public void run() {
tt.call(m);
}
}
class Synch {
public static void main(String a[]) {
Callme tt = new Callme();
Caller o1 = new Caller(tt,"hello");
Caller o2 = new Caller(tt,"Synchronized");
Caller o3 =  new Caller(tt, "World");
// Wait for threads to end
try {
o1.t.join();
o2.t.join();
o3.t.join();
}catch(InterruptedException e) {
System.out.println("Interrrupted");
}
}
}
```

Here is the output produced by this program:

```
Hello[Synchronized[World]]
]
]
```

As you can see, by calling **sleep( ),**the **call( )** method allows exception to switch to another thread. This results in the mixed-up output of the three message strings. In this program, nothing exists to stop all three threads from calling the same method, on the same object, at the same time. This is known as a race condition, because the three threads are racing each other to complete the method. This example used **sleep( )** to make the effects repeatable and obvious. In most situations, a race condition is more subtle and less predictable, because you can't be sure when the context switch will occur. This can cause a program to run right one time and wrong the next.

To fix the preceding program, you must serialize access to **call( )**. That is , you must restrict its access to only one thread at a time. To do this, you simply need to precede **call( )**'s definition with the keyword **synchronized** as shown here:

```
  class Callme {
            synchronized void call (String msg){
…
```

This prevents other threads from entering **call ( )** while another thread is using it. After synchronized has been added to **call( ),** the output of the program is as follows:

```
[Hello]
[Synchronized]
[World]
```

Any time that you have a method, or group of methods, that manipulates the internal state of an object in a multithreaded situation, you should use the **synchronized** keyword to guard the state from race conditions. Remember once a thread enters any synchronized method on an instance, no other thread can enter any other synchronized methods on the same instance. However, nonsynchronized methods on that instance will continue to be callable.

25.4 The Synchronized Statement

While creating **synchronized** methods within classes that you create is an easy an effective means of achieving synchronization, it will not work in all cases. To understand why, consider the following. Imagine that you want to synchronize access to objects of a class that was not designed for multithreaded access. That is, the class does not use **synchronized** methods. Further, this class was not created by you but by a third party, and you do not have access to the source code. Thus, you can't add **synchronized** to the appropriate methods within the class. How can access to an object of this class be synchronized? Fortunately, the solution to this program is quite easy. You simply put calls to the methods defined by this class inside a **synchronized** block.

This is the general form of the **synchronized** statement.

```
        synchronized (object)  {
/ / statements to be synchronized
}
```

Here object is a reference to the object being synchronized. If you want to synchronize only a single statement, then the curly braces are not needed. A synchronized block ensures that a call to a method that is a member of object occurs only after the current thread has successfully entered object's monitor.

Here is an alternative version of the preceding example, using a synchronized block within the **run( )** method.

```
// This program uses a synchronized block.
Class Callme{
void call (string msg){
System.out.print("["+msg);
try{
Thread.sleep(1000);
}catch (InterruptedException e){
System.out.println("Interrupted");
system.out.println(“]”);
}
}

class Caller implements Runnable{
String msg;
Callme target;
Thread t;
Public Caller(Callme targ , String s)
Target = targ;
msg  = s;
t = new Thread (this);
t.start();
}
//synchronize calls to call()
public void run(){
synchronized(target){
//synchronized block
target.call(msg);
}
}
}
class Synch1{
public static void main(String args[]){
Callme target = new Callme();
Caller ob1 = new Caller(target,”hello”);
Caller ob2 = new Caller (target, “Synchronized”);
Caller ob3 = new Caller (targer,”World”);
//wait for threads to end
try{
ob1.t.join();
ob2.t.join();
ob3.t.join();
}
catch(InterruptedException e){
System .out.println(“Interrupted”);
}
}
}
```

Here, the **call()** method is not modified by synchronized. Instead, the **synchronized** statement is used inside **Caller’s run()** method. This causes the same correct output as the preceding example, because each thread waits for the prior one to finish before proceeding.

25.5 Inter thread Communication

The preceding examples unconditionally blocked other threads from asynchronous access to certain methods. This use of the implicit monitors in Java objects is powerful, but you can achieve a more subtle level of control through inter process communication. As you will see this is especially easy in Java.

As discuses earlier , multithreading replaces event loop programming by dividing your tasks into discrete and logical units. Threads also provide a secondary benefit. They do away with polling. Polling is usually implemented by a loop that is used to check some condition repeatedly. Once the condition is true, appropriate action is taken. This wastes CPU time . For example, consider the classic queuing problem,. Where one thread is producing some data and another is consuming it. To make the problem more interesting, suppose that the producer has to wait until the consumer is finished before it generates more data. In a polling system, the consumer would waste many CPU cycles while it waited for the producer to produce . Once the producer was finished, it would start polling wasting more CPU cycles waiting for the consumer to finish, and so on. Clearly this situation is undesirable.

To avoid poling, Java includes an elegant inter process communication mechanism via the **wait( )**, **notify( )** and notify **All( )** methods. These methods are implemented as **final** methods in **Object**, so all classes have them. All three methods can be called only from within a **synchronized** method. Although conceptually advanced from a computer science perspective, the rules for using these methods are actually quite simple:

- **wait( )** tells the calling thread to give up the monitor and go to sleep until some other thread enters the same monitor and calls notify( )

- **notify( )** wakes up the first thread that called **wait( )** on the same object

- **notifyAll( )** wakes up all the threads that called **wait( )** on the same object. The highest priority thread will run first

These methods are declared within **Object,** as shown here

```
final void wait( ) throws InterruptedException
final void notify( )
final void notifyAll( )
```

Additional forms of **wait( )** exist that allows you to specify a period of time to wait.

The following sample program incorrectly implements a simple form of the producer/customer problem. It consists of four classes: **Q**, the queue that you're trying to synchronize; **Producer**, the threaded object that is producing queue entries; **Consumer**, the threaded object that is consuming queue entries; and **PC** the tiny class that creates the single **Q**, **Procedure** and **Consumer**.

```
// An incorrect implemention of a producer and consumer
class Q {
        int n;
        synchronized int get( ) {
                System.out.println("Got: " +n);
                return n;
        }

        synchronized void put(int n){
                this.n = n;
                System.out.println("Put: " + n);
        }
}

class Producer implements Runnable {
        Q q;

        Producer (Q,q) {
                this.q= q;
                new Thread(thos,"Producer").start( ) ;
        }

        public void run( ) {
                int i = 0;

                while(true) {
                        q.put(i++);
                }
        }
}

class Consumer implement Runnable {
        Q,q;
        Consumer(Q,q) {
                this.q= q;
                new Thread(this,”Consumer”).start( ) ;
        }

        public void run( ) {
                while(true) {
                        q.get( );
                }
        }
}

class PC {
        public static void main(String args[ ] ) {
                Q,q =  new Q( );
                new Producer(q);
                new Consumer(q);

                System.out.println(“Press Control-c to stop.”);
        }
}
```

Although the **put( )** and **get( )** methods on **Q** are synchronized, nothing stops the producer from overrunning the consumer, nor will anything stop the consumer from consuming the same queue value twice. Thus you get the erroneous output shown here:

```
Put :1
Got :1
Got :1
Got :1
Got :1
Got :1
Put :2
Put :3
Put :4
Put :5
Put :6
Put :7
Got :7
```

As you can see, after the producer put 1, the consumer started and got the same 1 five times in a row. Then the procedure resumed and produced 2 through 7 without letting the consumer have a chance to consume them.

The proper way to write this program in Java is to use **wait( )** and **notify( )** to signal in both directions as shown here:

```
//A correct implemetation of  a procedure and  consumer
class Q {
        int n;
        boolean valueSet = false;

    synchronized int get( ) {
                if(!valueSet)
                        try {
                                wait( );

          } catch(InterruptedException e) {
                        System.out.println("InterruptedException caught");
                }

                System.out.println("Got: "+n);
                valueSet = false;
                notify( ) ;
    return n;
        }

        synchronized void put(int n){
                if(!valueSet)
                        try {
                                wait( );

                        }   catch(InterruptedException e) {
                        System.out.println("InterruptedException  caught");
                        }

                        this.n = n;
                        valueSet = true;
                        System.out.println("Put: "+n);
                        notify( ) ;
        }
}
class Producer implements Runnable {
```

```
          Q q;
      Producer (Q,q) {
                    this.q= q;
                    new Thread(this,"Producer").start( ) ;
          }

          public void run( ) {
                    int i = 0;

                    while(true) {
                              q.put(i++);
                    }
          }
}
class Consumer implement Runnable {
          Q,q;

          Consumer(Q,q) {
                    this.q= q;
                    new Thread(this,"Consumer").start( ) ;
          }
          public void run( ) {
                    while(true) {
                              q.get( );
                    }
          }
}

class PCFixed {
          public static void main(String args[ ] ) {
                    Q,q =  new Q( );
                    new Producer(q);
                    new Consumer(q);
                    System.out.println("Press Control-c to stop.");
          }
}
```

Inside **get( )**, **wait( )** is called. This causes its execution to suspend until the Producer notifies you that some data is ready. When this happens, execution inside **get( )** resumes. After the data has been obtained, **get( )** calls **notify( ).** This tells Product that it is okay to put more data in the queue. Inside **put( )**, **wait( )** suspends execution until the Consumer has removed the item form the queue. When execution resumes, the next item of data is put in the queue, and **notify( )** is called. This tells the **Consumer** that it should now remove it.

Here is some output from this program, which shows the clean synchronous behavior.

```
put : 1
Got : 1
put : 2
Got : 2
put : 3
Got : 3
put: 4
Got :4
put:  5
Got: 5
```

25.6 Deadlock

A special type of error that you need to avoid that relates specifically to multitasking is deadlock, which occurs when two threads have a circular dependency on a pair of synchronized objects. For example, suppose one thread enters the monitor on object X and another thread enters the monitor on object as T. If the thread in tries to call any synchronized methods on Y, it will block as expected. However, if the thread in Y in turn tries to call any synchronized method on X, the thread waits forever, because to access X. it would have to release its own lock on Y so that the first thread could complete. Deadlock is a difficult error to debug for two reasons:

- In general, it occurs only rarely when the two threads time-slice in just the right way.
- It may involve more than two threads and synchronized objects. (That is, deadlock can occur through a more convoluted sequence of events than just described.)

To understand deadlock fully, it is useful to see it in action. The next example creates two classes, **A** and **B**, with methods **foo( )** and **bar ( )** respectively, which pause briefly before trying to call a method in the other class. The main class, named Deadlock, creates an **A** and a **B** instance, and then starts a second thread to set up the deadlock condition. The **foo( )** and **bar ( )** methods use **sleep ( )** as a way to force the deadlock condition to occur.

```
/ / An example of deadlock.
class A {
  synchronized void foo (B b) {
        String name = Thread.currenThread ( ).getName( );

system.out.prinln (name + “entered A.foo”);

 try {
      Thrad.sleep (1000);
    }catch (Exception  e) {
       System.out.println (“A Interupted”);
 }

System.out.println(name + “trying to call B last  ( )” );
b.last ( );

}

synchronized void last ( )  {
    system.out.println(“Inside A. last”);
  }
}



class B {
      synchronized void bar (A  a)   {
           String name = Thread.currentThread ( ). getName( );
            System.out.println (name + “entered B.bar”);

             try   {
                   Thread.sleep(1000);
            }    catch (Exception   e)   {
                    System.out.println (“B  interrupted”);
             }

           system.out.println (name + “trying to call A.last ( )”  );
            a.last ( );
}

            synchronized void last ( ) {
                    System.out.println(“Inside A.last”);
        }
}

class Deadlock implements Runnable  {
       A    a = new A ( ) ;
        B    b = new B (  );

Deadlock ( )  {
   Thread.currentThread ( ).setName (“MainThread”);
     Thread t = new Thread (this,  “RachingThread”) ;
  t.start  ( );

a.foo (b); / / get lock on a in this thread.
system.out.println (“Back in main thread”);
}

       public void run( ) {
            b.bar(a); / / get lock on b in other thread.
             System.out.println (“Back in other thread”);
}

public static void main(String args [ ] ) {
     new Deadlock ( );
       }
}
```

when you run this program, you will see the output shown here:

```
             MainThread entered A.foo

             RachinThread entered B.bar

             MainThread trying to call B. last ( )

     RacingThread trying to call  A. last ( )
```

Because the program has deadlocked, you need to press CTRL-C to end the program. You can see a full thread and monitor cache dump by pressing CTRL-BREAK on a PC (or CTRL-\on Solaris).You will see that **Racing Thread** owns the monitor on b, while it is waiting for the monitor on a . At the same time, **Main Thread** own a and is waiting to get **b**. This program will never complete. As this example illustrates, if your multithreaded program locks up occasionally, deadlock is one of the first conditions that you should check for.

25.7 Short Summary

- When two or more threads need access to a shared resource, they need some way to ensure that the resource will be used by only one thread at a time. The process by which this is achieved is called **synchronization.**

- A special type of error that you need to avoid that relates specifically to multitasking is deadlock, which occurs when two threads have a circular dependency on a pair of synchronized objects.

25.8 Brain Storm

1. What is Synchronization?
2. Explain Synchronization Method.
3. What is the need of Inter thread Communication?
4. What is Deadlock?

Lecture - 26

AWT GUI Components

Objectives

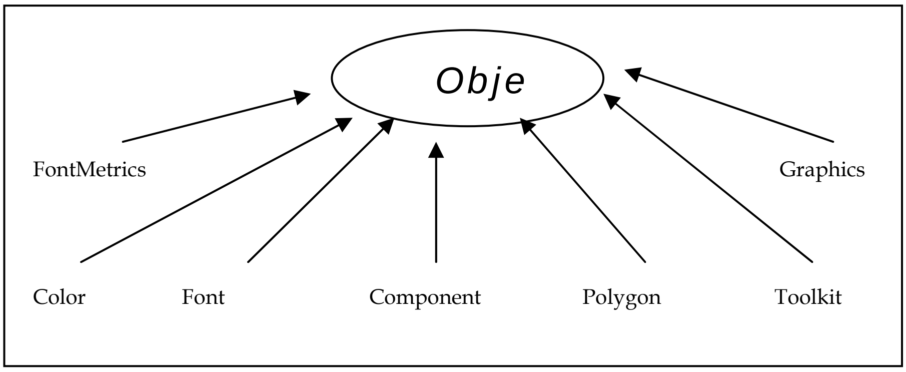
In this lecture you will learn the following

- Knowing about AWT
- About GUI Components
- Java.awt Packages
- Event Handling

Lecture - 26

26.1 Snap Shot

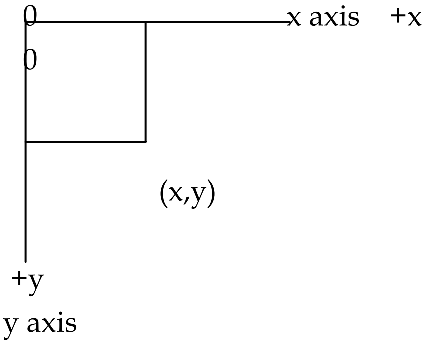
We now begin in-depth treatment of Java's Abstract Windowing Toolkit (AWT) - the classes that comprise the java.awt package. The figure shows a portion of the java.awt class his hierarchy that includes the classes covered in this chapter. Each class in the figure inherits directly form class Object. Class Color contains methods and constants for manipulating colors. Class Font contains methods and constants for manipulating fonts. Class FontMetrics contains methods for obtaining font information. Class Polygon contains methods for creating polygons. Class Graphics contains methods for drawing strings lines rectangles and other shapes. Class Toolkit provides methods for getting graphical information.



26.2 Java Coordinating System

By default, the upper-left corner of the screen has the coordinates (0,0). A coordinate pair is composes of an x coordinate and a y coordinate.

The x coordinate is the horizontal distance moving right from the upper-left corner. The y coordinate is vertical distance moving down from the upper-left corner.



26.3 Graphics Contexts & Graphics Objects

A graphics context enables drawing on the screen in Java. A Graphics object manages a graphics context by controlling how information is drawn. Graphics objects contain methods for drawing, font manipulation, color manipulation etc. Every applet we have seen in the text that performs drawing on the screen has used the Graphics object **g** to manage the applet's graphics context.

A Graphics class is an abstract class. That is Graphics object cannot be instantiated. Class Component is the super class for many of the classes in the AWT. Component method paint takes a Graphics object as an argument. This object is passed into the paint methods by the system when a paint operation occurs for a Component. The header for the paint method is :

```
public void paint(Graphics g)
```

The Graphics object g receives a reference to an object of the system's derived Graphics class. When an applet is initially executed the paint method is automatically called. If paint method is to be called once again, a call is made to the Component class repaint method. The repaint method requests a call to the Component class update method as soon as possible to clear the Component's background of any previous drawing, then update calls paint directly.

The general form of repaint and update is

```
public void repaint( )
public void update (Graphics g)
```

Both methods are public and have a void return type. The update method takes a Graphics object as an argument which is supplied automatically by method repaint.

Graphics methods for drawing string and characters and bytes. Method drawstring draws a String. The method takes three arguments - String to be drawn, an x coordinate, and a y coordinate. The String is drawn in the current color and font. The current color is the color in which text is drawn. The point(x,y) corresponds to the lower left corner of the string.

The general form is

```
public abstract void drawString ( String string, int x, int y)
        string = string to be drawn
        int x =    x coordinate
        int y =   y coordinate
```

Method drawChars draws a series of characters. The method takes five arguments. The first argument is an array of characters. The second argument specifies the subscript in the array of the first character to be drawn. The third argument specifies the number of character to be drawn. The last two arguments specify the coordinates where drawing is to begin. The point (x, y) corresponds to the lower-left corner of the first character drawn.

The general form is

```
        public void drawChars(char chars[ ], int offset, int number, int x, int y)
                chars[ ] = array to be drawn
                offset     = starting subscript (index)
                number  = number of elements to draw
                x and y  =  x and y coordianates
```

Method drawBytes draws a series of byes. Like the drawChars method, the drawBytes method takes five arguments. The first argument is an array of bytes. The second argument specifies the subscript in the array of the first byte to be drawn. The third argument specifies the number of elements to be drawn. The last two arguments specify the coordinates where drawing is to begin. The point(x,y) corresponds to the lower-left corner of the bytes drawn

The general form is

```
        public void drawBytes ( byte bytes [ ], int offset, int number, int x, int y)
                bytes [ ]    =    array of bytes
                offset       =    starting subscript
                number            =     number of elements to draw
                x and y           =     x and y coordianate
```

```
// Demonstrating drawString, drawChars and drawBytes
import java. applet.Applet;
import java.awt.Graphics;
public class DrawSCB extends Applet
{
        private String s = "using drawstring";
        private char c [ ] = { 'c', 'h', 'a','r', 's', ' ' ,' 8'}
        private byte b[ ] =  { 'b','y','t','e', 1,2,3}

        pubic void paint (Graphics g)
        {
                g.drawstring(s,100,25);
                g.drawChars(c,2,3,100,50);
                g.drawBytes(b,0,5,100,50);
        }
}
```

In the program the drawstring method displays "using drawstring" at location 100,25 The statement

```
        g.drawString(c,2,3,100,50);
```

displays "ars" at location of (100,50). The second argument , 2, specifies that drawing is to begin with subscript 2 of the character array c. The third argument, 3, specifes that three elements will be drawn. Method drawBytes displays a set of byte sat location (100,75).

26.4 Overview of the Java.awt Package

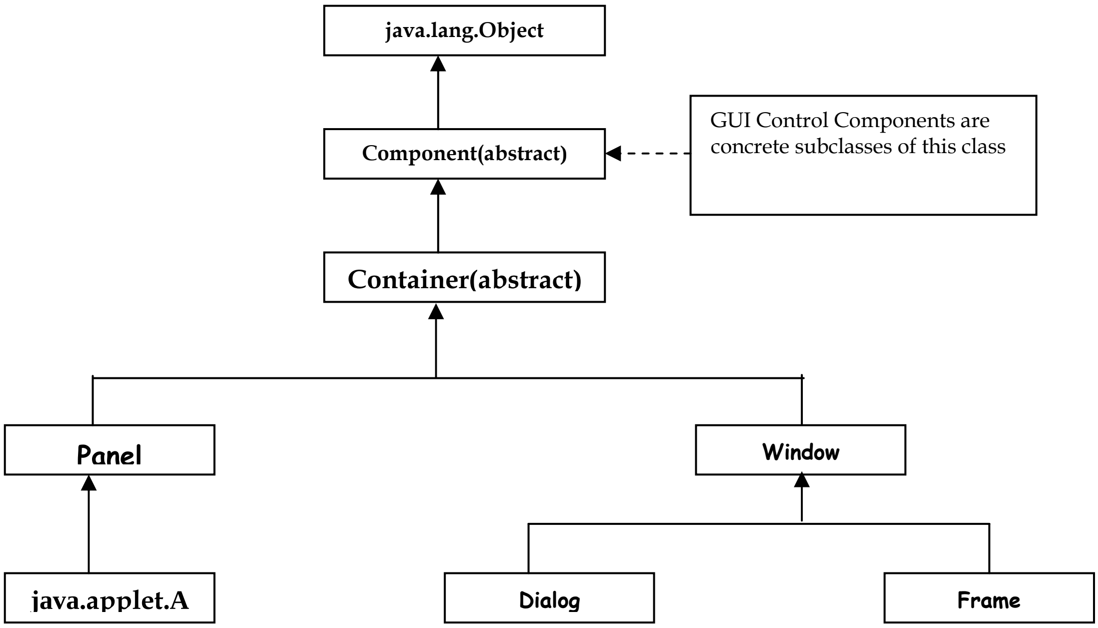
The Java Foundation classes (JFC) provides two frameworks for building GUI based applications. The Abstract Windowing Toolkit (AWT) relies on the underlying windowing system on a specific platform to present its GUI components. The other GUI toolkit in the JFC is called Swing, implements a new set of lightweight GUI components that are written in Java and have a pluggable look and feel .

The package java.awt provides the primary facilities of the AWT:

- Managing the layout of the components within the container object
- Support for event handling that is essential for user interaction in GUI based systems.
- Rendering graphics in GUI components using color, fonts, images and polygons

Components and containers

The partial class hierarchy is shown in the figure below. The figure depicts the principal container classes that provides the underlying functionality for building GUI based applications.



Partial inheritance Hierarchy of Components and Container in AWT

Component	The Superclass of all non-menu-related components that provides basic support for handling of events, changing of component size, controlling of fonts and colors and drawing of components and their controls.
Container	A container is a component that can accommodate other components and also containers. Containers provide the support for building complex hierarchical graphical user interface
Panel	A panel is a container ideal for packing other component and panels to build component hierarchies
Applet	An applet is a specialized panel that can be used to develop programs that run in a web browser
Window	The window class represents a top-level window that has no title, menus or borders
Frame	A frame is an optionally user resizable and movable top-level window that can have a title bar, an icon, and menus
Dialog	The dialog class defines an independent optionally user resizable window that can only have a title bar and border. A Dialog window can be model, meaning that all input is directed to this window until it is dismissed

26.5 Component Class

All non-menu-related elements that comprise a graphical user interface are derived from the abstract class Component. The Component class specifies a large assortment of methods for handling events, changing window bounds, controlling fonts and colors and drawing components and their contents. A Component uses the visual properties (font face, background color, foreground color, etc.) of its parent container, unless set explicitly for the component.

The following utility methods are provided by component and its sub class:

```
Dimension getSize ( )
void setSize (int width, int height)
void setSize (Dimension d)
```

The getSize( ) method can be used to get the size of a component in pixels. The return object is of type Dimension, which has two public data members width, and height. The setSize( ) methods can be used to set the size of a component in pixels.

```
point getLocation ( )
void setLocation ( int x, int y)
void setLocation (Point p)
```

The getLocation ( ) method returns the coordinates of the top-left corner of the component. The return object is of type point, which has two public data members x and y. The setLocation methods can be used to move the component to the specified locations

```
Rectangle getBounds( )
void setBounds(int x, int y, int width, int height)
void setBounds (Rectangle r)
```

The getBounds ( ) method can be used to get the bounds of a component(both in size and location). The return object is of type Rectangle, which has four public data members: x, y, width and height. The setBounds ( ) methods can be used to set the bounds of a component

```
void setForeground(Color c)
void setBackground(Color c)
```

The setForground( ) method can be used to set the foreground color of a component. The setBackground ( ) method can be used to set the background color of a component. Normally the background colors is used to fill the area occupied by the component and text is rendered in the component's area using the foreground color.

If foreground and background colors are not explicitly specified for a component the corresponding values from this component's immediate container are used.

```
Font getFont ( )
void setFont( Font f)
```

The getFont( ) method returns the font used for rendering the text in a component. The setFont ( ) method can be used to set a particular font.

```
void setEnabled(boolean b)
```

if the argument of this method is true, the component acts as normal, i.e., it is enabled and can respond to user input and generate events. If the arguments is false, then the component appears grayed out and does not respond to external stimuli. All components are initially enabled

```
void setVisible(boolean b)
```

A component is either shown on the screen or hidden, depending on the argument to this method being either true or false respectively. It influences the visibility of the child components except for window, Frame, and Dialog classes, whose instance must explicitly be made visible by this method.

Container Class

The abstract class Container is a subclass of the abstract class Component. It defines methods for nesting components in a container. A container is a component that can accommodate other components and thereby other containers since a container is also a component by virtue of inheritance.

Containers provide the functionality for building complex, hierarchical graphical user interfaces. They define a component hierarchy in contrast to the inheritance hierarchy defined by classes. Container provide overloaded method add ( ) to include components in a container. A container uses a layout manager to position its components.

Panel Class

The panel class is a concrete subclass of the Container class. It provides an intermediate level for GUI organization. A panel is recursively nested, concrete container that is not a top-level window. It also does not have a title, menus or borders. It is therefore ideal for packing other components and panels to build component hierarchy archies using the inherited add ( ) method

Applet Class

The Applet class belongs to java.applet package. It is a subclass of the Panel class and thus inherits its functionality. An applet is a specialized panel used to develop programs than run in a web browser.

Frame Class

The Frame class is a subclass of the window class. It is used to create what we usually mean by a GUI application window. A frame is an optionally user resizable and movable top-level window that can have a title-bar, an icon and menus. A Frame object is usually starting with GUI application and serves as the root of the component hierarchy. A Frame object can contain several panels which in turn can hold other GUI control components and other nested panels.

The Frame class defines two constructors

```
    Frame ( )
    Frame(String title)
```

Dialog Class

The dialog is a subclass of the window class. The class defines an optionally user resizable and movable top-level window with a title bar. Dialog box will not have menu bar and an icon. A dialog window can direct all input to its window until it is dismissed.
The Dialog class contains several constructors

```
        Dialog(Frame parent)
        Dialog(Frame parent, boolean modal)
        Dialog(Frame parent, String title)
        Dialog(Frame parent, Stirng title, boolean model)
```

All constructors create an initially invisible dialog box.

```
//create a child frame window
import java.awt.*.;
import java.awt.event.*;
import java.applet.*;
/* <applet code = "Applet Frame" width = 300 height = 50>
</applet>
*/
//create a subclass of Frame
class SampleFrame extends Frame
{
        SampleFrame(Strint title)
{
        super(title);
        // create an object to handle window events
        MyWindowAdapter adapter = new MyWindowAdapter (this);
                // register it to receive those events.
                addWindowListener(adapter);
                }
public void paint ( Graphics g)
{
        g.drawstring("this is in frame window",10,40);
}
}
class MyWindowAdapter extends WindowAdapter
{
        SampleFrame sampleFrame;
public MyWindowAdapter(SampleFrame sampleFrame)
{
        this.sampleFrame = sampleFrame;
}
public void windowClosing(WindowEvent we)
{
        sampleFrame.setVisible(false);
}
}
// creat frame window
public class AppletFrame extends Applet
{
        Frame f;
        public void init( )
        {
                f = new SampleFrame("A frame window");
                f .setSize(250,250);
                f.setVisible(true):
        }
        public void start( )
        {
                f.setVisible(true);
        }
        public void stop ( )
        {
                f.setVisible(false);
        }
        public void paint (Graphics g)
        {
                g.drawstring("this is an applet window" 10, 20);
        }
}
```

26.6 GUI Control Components

Graphical user interface control components are the primary elements of a GUI that enable interaction with the user. They are all concrete subclasses of Component. GUI control components for constructing menus are derived from the abstract class MenuComponent.

The following three steps are essential in making use of a GUI control component.

- A GUI control component is created by calling the appropriate constructor

- Button guiComponent = new Button("OK");
- The GUI control component is added to a container using a layout manager. This usually involves invoking the overloaded method add ( ) on a container with the GUI contral as the argument:
- guiFrame.add(guiComponent);
- Listeners are registered with the GUI component so that they can receive events when these occur. GUI components gernerate particular events response to user actions

Button	A button with a textual label, designed to invoke an action when pushed, called a push button
Canvas	A generic component for drawing and designing new GUI component
Checkbox	A checkbox with a textual label that can be toggled on and off. Checkboxes can be grouped to represent radio buttons
Choice	A component that provides a pop-up menu choices. Only the current choice is visible in the Choice component
Label	A lebel is a component that displays a single line read-only, non-selectable text.
List	A component that defines a scrollable list of text items
Scroll bar	A slider to denote a position or a value
Text Field	A component that implements a single line of optionally editable text.
Text Area	A component that implements multiple lines of optionally editable text.

Running Example

Each GUI control component is described below. The following generic HTML file can be modified to run the examples in a web browser or using an applet viewer. Only appropriate class for the applet needs to be changes.

```
<!-- HTML file to run an applet-- >
<!-- Change Applet class name as appropriate -- >
<title > GUI Control  Component</title>
<hr>
<applet code = "AppletClassName.class" width = 200 height = 200></applet>
```

26.7 Event handling

The event class is central to the Java window event-generation and handling mechanism. Event objects are generated by a user who interacts with a java window program or applet and by the Java runtime system. User-generated events occur when users make selections on a menu or press a key. Events generated by the runtime system include errors and exceptions. They are handled predefined event-handling methods that are defined by the component class and its subclasses, these methods are overridden to perform custom event processing.

The event class defines numerous constants to identify the events that are defined for the AWT classes. It is important that you review these constants to become familiar with the types of events that may need to be handled by your programs. You'll become aware of the common event handling performed for the various window components by working through the example programs.

The classes that represent events are at the core of Java's event handling mechanism. Thus, we begin our study of event handling with a tour of the event classes. As you will see, they provide a consistent, easy-to-use means of encapsulating events.

At the root of the java event class hieraracy is EventObject, which is in java.util. It is the superclass for all events.Its one constructor is shown here:

EventObject(Object src)
Here,src is the object that generates this event.

EventObject contains two methods :getSource() and toString().The getSource()

Method returns the source of the event.Its general form is shown here:

Object getSource()
As expected,toString() returns the string equivalent of the event.
The class AWTEvent, defined within the java.awt package, is a subclass of EventObject. It is the superclass(either directly or indirectly)of all AWT-based events used by the delegation event model. Its getID() method can be used to determine the type of the event. The signature of this method is shown here:
int getID()

- EventObject is a superclass of all events.
- AWTEvent is a superclass of all AWT events that are handled by the delegation event model.

The package java.awt.event defines several types of events that are generated by various user interface elements. The Most important event class are given in the below table.

Event Class	Description
ActionEvent	Generated when a button is pressed, a list item is Double-clicked, or a menu item is selected.
AdjustmentEvent	Generated when a scroll bar is manipulated.
ComponentEvent	Generated when a component is hidden, moved, resized, or becomes visible.
ContainerEvent	Generated when a component is added to or removed from a container.
FocusEvent	Generated when a component gains or loses keyboard focus.
InputEvent	Abstract super class for all component input event classes
ItemEvent	Generated when a checkbox or list item is clicked; also occurs when a choice selection is made or a checkable menu item is selected or deselected.
KeyEvent	Generated when input is received from the keyboard.
MouseEvent	Generated when the mouse is dragged, moved, clicked, pressed, or released; also generated when the mouse enters or exits a component.
TextEvent	Generated when the value of a text area or text area or text field is changed.
WindowEvent	Generated when a window is acivated, close, deacivated, deiconified, iconified, opened, or quit.

Table: Main Event Classes in java.awt.event

Updates to the spreadsheet applet are driven by the occurrence of user events, such as keypresses and mouse button clicks. As discussed events are usually handled with either the action() method or the handleEvent() method. The action() method of the components class is the preferred way to handle a limited number of events. The handleEvent method of the Event class is the perferred way to handle multiple or complex series of events.

When an applet has only a few possible events that deal with the mouse or keypresses, you can call more direct methods that eliminate the necessary of nested if or case statements to check event type. These methods, like the action() method, are members of the component class.

26.8 Short Summary

- A Graphics object manages a graphics context by controlling how information is drawn.

- The Abstract Windowing Toolkit (AWT) relies on the underlying windowing system on a specific platform to present its GUI components.

- The package java.awt provides the primary facilities of the AWT Managing the layout of the components within the container object Support for event handling that is essential for user interaction in GUI based systems. Rendering graphics in GUI components using color, fonts, images and polygons

- All non-menu-related elements that comprise a graphical user interface are derived from the abstract class Component.

26.9 Brian Storm

1. What is AWT?
2. Explain the features of GUI.
3. What are the subclasses of Container class?
4. Explain each of the method with your own examples.

Lecture - 27

Components

Objectives

In this lecture you will learn the following

- Component
- Container
- Events
- Layouts
- Painting and Updating
- Understanding Layout Managers

Lecture - 27

27.1 Snap Shot

In this lecture you will learn about component, container, events, layout and painting and updating and understanding layout managers.

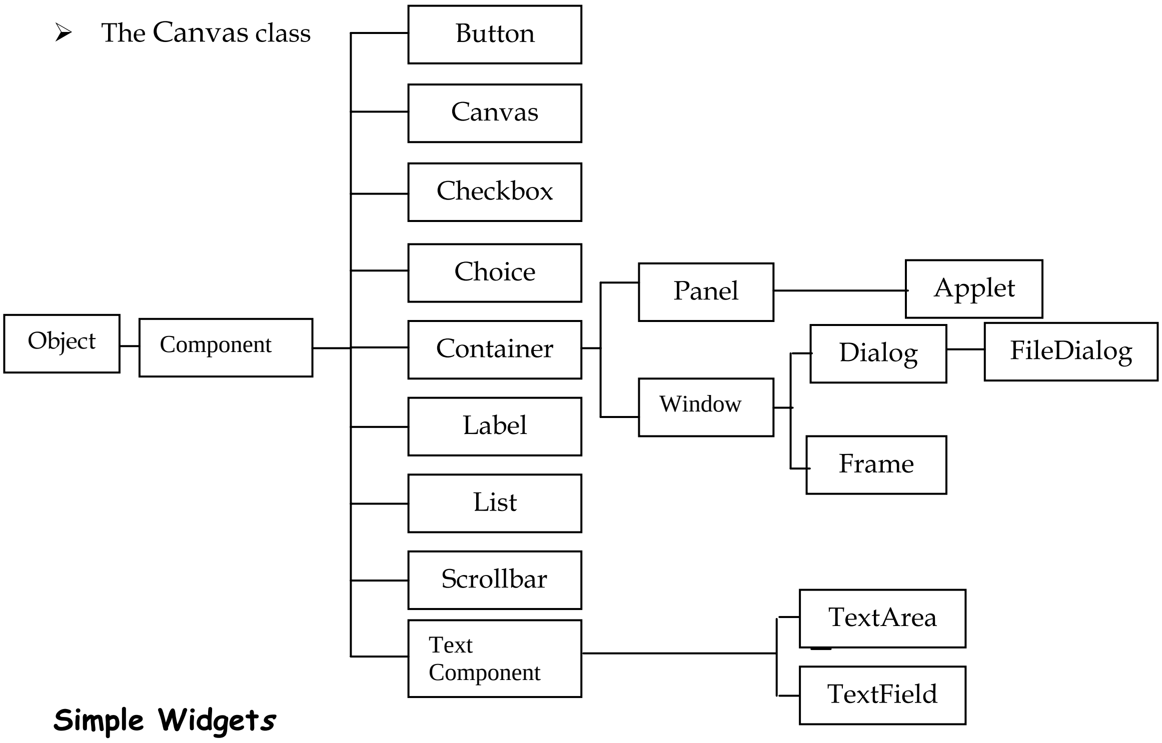
27.2 Component

The AWT component classes are all derived from a common base class: the Component class. The Component class is an abstract class. This class defines the elements common to all GUI components. The Component class is derived from the Object class. The Component class also implements the ImageObserver interface.

The Component class provides a unified interface to all the graphic components of the AWT. Figure shows all the AWT widgets derived from the Component class.

The components in this hierarchy can be divided into the following functional groups:

- Simple widgets (buttons, checkboxes, and so on)
- Text controls
- The Canvas class



Simple Widgets

The Java AWT encapsulates many of the controls common to most GUIs. Specifically, these are the Button, Checkbox, Choice, Label, List, and Scrollbar classes. Figure 16.2 shows an applet that displays the AWT simple widgets.

The following code shows the Simple applet, which contains the simple AWT widgets. The applet's init() method creates an example of each simple widget and adds it to the applet.

```
import java.awt.*;

public class Simple extends java.applet.Applet {

   public void init() {

      Button     button     = new Button( "Quit" ) ;
      Checkbox   checkbox   = new Checkbox( "Test" ) ;
```

Both the Choice and List objects include addItem() methods. These methods allow you to fill the control with the items you specify. Unlike some GUIs, the AWT Choice and List controls do not sort the items they contain. They are displayed in the order in which you add them.

```
      Choice     choice     = new Choice() ;
       // fill the Choice
      choice.addItem( "Clinton" ) ;
      choice.addItem( "Dole" ) ;
      choice.addItem( "Perot" ) ;
      choice.addItem( "Browne" ) ;
      choice.addItem( "Nader" ) ;

     Label label = new Label( "This is a label" );

      List list = new List( 5, false ) ;
      // fill the List
      list.addItem( "Clinton" ) ;
      list.addItem( "Dole" ) ;
      list.addItem( "Perot" ) ;
      list.addItem( "Browne" ) ;
      list.addItem( "Nader" ) ;

Scrollbar scrollbar = new Scrollbar(scrollbar.HORIZONTAL);
```

To display these controls, you must add them to the applet's layout using the add() method:

```
      // add the controls to the default layout
```

```
        add( button ) ;
        add( checkbox ) ;
        add( choice ) ;
        add( label ) ;
        add( list ) ;
        add( scrollbar ) ;

    }

}
```

This applet displays the simple widgets. To be truly useful, an applet should do more than just display the controls-the applet must also be interactive. This brings up the topic of *event handling*. The DemoFrame applet presented later in this chapter demonstrates event handling.

At the top of the AWT Hierarchy is the **Component** class. **Component** is an abstract class that encapsulates all of the attributes of a visual component.All user interface elements that are displayed on the screen and that interact with the user are subclass of Component. It defines over a hundred public methods that are responsible for managing events,such as mouse and keyboard input,positioning and sizing the window, and repainting.

A Component object is responsible for remembering the current foreground and background colors and the currently selected text font.

Controls are component that allow a user to interact with your application in various ways.

Control Fundamentals

The AWT supports the following types of controls

- Labels
- Push buttons
- Check boxes
- Choice Lists
- Lists
- Scroll bars
- Text editing

These controls are subclasses of **Component.**

Adding and Removing Controls

To include a control in a window, you must add it to the window. To do this, you must first create an instance of the desired control and then add it to a window by calling **add().** Which does Container define. The add() method has several forms.

```
Component add(Component compobj)
```

Here, *compobj* is an instance of the control that you want to add. A reference to *compobj* is returned. Once a control has been added, it will automatically be visible whenever ,its parent window is displayed. Suppose, you will want to remove a control from a window when the control is no longer needed. To do this, call remove().This method is also defined by **Container.**

It has this general form:

```
Void remove(Component obj)
```

Here, *obj* is a reference to the control you want to remove.You can remove all controls by calling removeAll().

The Component Event Class

A Component Event is generated when the size position, or visibility of a component is changed. There are four types of component events. The ComponentEvent class defines integer constants that can be used to identify them. The constants and the meanings are shown here.

COMPONENT_HIDDEN	The component was hidden
COMMPONENT_MOVED	The component was moved.
COMPONENT_RESIZED	The component was resized.
COMPONENT_SHOWN	The component became visible.

ComponentEvent has this constructor:

ComponentEvent(Component *src*, int *type*)

Here, *src* is a reference to the object that generated this event.The type of the event is specified by *type*.

ComponentEvent is the superclass either directly or indirectly of **ContainerEvent, FocusEvent, KeyEvent, MouseEvent, and WindowEvent.**

The **getComponent()** method returns the component that generated the event.It is shown here:

```
Component getComponent()
```

27.3 Container

The Container class is a subclass of Component. It has additional methods that allow other Component objects to be nested within it. Other Container objects can be stored inside of a Container(since they are themselves instances of

Component).This makes for a multileveled containment system. A Container is responsible for laying out any components that is contains.

The subclasses of the Container is

- Panel
- Window

The ContainerEvent Class:

A **ContainerEvent** is generated when a component is added to or removed from a container. There are two types of container events. The **ContainerEvent** class defines int constants that can be used to identify them. **COMPONENT_ADDED** and **COMPONENT_REMOVED**. They indicate that a component has been added to or removed from the container.

ContainerEvent is a subclass of **ComponentEvent** and has this constructor:

ContainerEvent(Component *src*,int *type*,Component *comp*)

Here, *src* is a reference to the container that generated this event. The type of the event is specified by *type*, and the component that has been added to or removed from the container is *comp*. You can obtain a reference to the container that generated this event by using the **getContainer()** method, shown here:

Container getContainer()

The **getChild()** Method returns a reference to the component that was added to or removed from the container. Its general form is shown here:

Component getChild()

27.4 Events

In the delegation model, an event is an object that describes a state change in a source. It can be generated as a consequence of a person interacting with the elements in a graphical user interface. Some of the activities that causes events to be generated are pressing a button, entering a character via the keyboard, selecting an item in a list, and clicking the mouse. Many other user operations could also be cited as examples.

Events may also occur that are not directly caused by interactions with a user interface. For example, an event may be generated when a timer expires, a counter exceeds a value, a software or hardware failure occurs, or an operation is completed. You are free to define events that are appropriate for your application.

27.5 Layouts

Layout Classes

The AWT contains a group of classes designed to handle placement of controls in Container objects. These are the layout classes. All layout classes are derived directly from Object. These classes all implement the LayoutManager interface.

The AWT implements the following layout classes:

- FlowLayout
- BorderLayout
- CardLayout
- GridLayout
- GridBagLayout

These classes provide a flexible, platform-independent means of arranging Component objects in your Container objects. It is possible that these five classes provide all the flexibility your applets and applications need. If you have specific needs, you can implement your own layout class by deriving it from Object and implementing the LayoutManager interface.

27.6 Painting and updating

The paint() method is called each time your applet's output must be redrawn. This situation can occur for several reasons. For example, the window in the which the applet is running may be overwritten by another window and then uncovered, or the applet window may be minimized and then restored. Paint() is also called when the applet begins execution. Whatever the cause, whenever the applet must redraw its output, paint() is called. The paint() method has one parameter of type Graphics. This parameter will contain the graphics context, which describes the graphics environment in which the applet is running. This context is used whenever output to the applet is required.

The method paint is used to paint or repaint the screen. It is automatically called by repaint or can be called explicitly by the applet. The applet calls paint when the browser requires a repaint, such as when an obscured applet is brought to the front of the screen again.

Paint has a fixed format. It always must be named paint, have a return type of void, and be declared public. However, unlike init and start, it does have an argument, of type graphics. This is a predefined type in java that contains many of the methods for writing graphics to the screen. Here is an example of a paint method:

```
        public void paint(Graphics g) {
        g.drawstring(“counter = “ + counter, 10,10);
}
```

This method writes the value of counter to the screen each time it is invoked. Combine this with the start and repaint methods. The start method increments counter and then calls repaint. The screen displays the new value of counter every time it is updated. The compiler does not require a paint method. It overrides a default method provided by java.applet.Applet. However, if you do not be able to write anything to the screen.

After you have initialized the spreadsheet, you must display it. This is usually done by means of the apit b( ) method, which draw all the objects added during the initialization phase. As discussed in "Step 4. Sizing spreadsheet, the object should sized according to the dimensions of the frame or a key constraint such as the font size. You should also paint the objects in the orders they are to be added to the frame.

The spreadsheet applet uses many interesting techniques to place objects precisely on the frame. As shown in Listing 20.8, centering the title horizontally in a in the title area involves five steps.

1. Obtaining the dimension of the frame
2. Determining the pixel of the title from the width of the frame
3. subtracting the length of the title form difference in font size
4. Adding in a factor that accounts for differences in font size.
5. Dividing the result 2

The value you obtain by doing this is the x coordinates for the beginning o the title. You also need a y coordinate for the title. Because the height of the title area is relative to the font size used, the font size is used to help determine this coordinate.

Next, the paint ( ) method draws the input area by simply filling a rectangle with a color based on the value of the inputColor object. The starting location for the upper left corner of the input area is determined by moving vertically down the height of one cell. The width of the input area is set to the width of the frame, and the height is set to the height of one cell. Finally the paint ( ) method draws the individual cells,. To crater them, use a series of draws, the find adds the horizontal lines for rows using the draw3DRect method of the Graphics class . As the 3D lines are drawn to the frame, blue numerals representing the row numbers are added as Appropriate. The second draw adds the vertical lines for columns, again using the draw3DRect method. As the 3D lines are drawn to the frame, red letters representing the column letter are added as appropriate. Finally data is added to the cells by a painting the values associated with a cell precisely within the rectangles created by the previous draws The last section of code in the paint ( ) method, although only a few lines, is important to the spread sheet., the line of code using the draw3DRect( ) method draws a 3D line above the input area and on the left side of the frame. The next to last line calls the paint ( ) methods of the inputArea class, which ensures that the data associated with the currently selected cell is painted to the input area.

Painting the frame

```
Public synchronized void paint(Graphics g){

int i,j;

int cx,cy;

char 1[] = new char[1];

Dimension d = sixe();

//draws the title on the frame

g.setFont(titleFont);

i=g.getFontMetrics().stringWidth(title);

g.drawString((title = =null)? "Spreadsheet":title,(d.width-i +(fontSize *3/2))/2,(fontSize *3/2));

//draws the input area on the frame

g.setColor(inputColor);

g.fillRect(0,cellHeight, d.width, cellHeight);
g.setFont(titleFont);
for(i=0;i<rows+1;i ++)
{ cy=(i+2)*cellHeight;
g.draw3DRrect(0,cy,d.width,2,true);
if(I<rows) {
g.setColor(Color.blue);
g.drawString(“”+(I+1),fontSize,cy+(fontSize * 3/2));
}
}
g.setColor(Color.red);
for(i=0;I<columns;i++)
{
cx=i*cellWidth;
g.setColor(getBackgroung());
g.draw3DRect(cx + rowLabelWidth,2*cellHeight,1,d.height,true);
if(I<columns) {
g.setColor(Color.red);
i0]=(char)((int)’A’+i);
g.drawString(new String(i),cx+rowLabelWidth+(cellWidth/2),d.height – 3);
}
}
for(i=0;i<rows;i++)
{for(j=0;i<columns;j++)
{ cx=(j*cellWidth)+2+rowLabelWidth;
cy=(i+1)*cellHeight)+2+titleHeight;
if(cells[I][j] != null) {
cells[I][j].paint(g,cx,cy);
}
}
}
g.setColor(getBackground());
g.draw3DRect(0,titleHeight,d.width,d.height-titleHeight,false);
inputArea.paint(g,1,titleHeight+1);
}
```

When changes occur in the applet, you must repaint the applet’s frame using the repaint ( ) method of the applet class, if you recall earlier discussions on repainting the applet’s frame, you probably know that the repaint ( ) method calls the update ( ) method of the applet class, which in turn clarets the screen and calls the paint ( ) methods. The paint method then draws in the applet’s frame. Clearing and then drawing the frame produces a noticeable, flicker. To reduce it, usually it’s best to override the update ( ) method of the applet class. You do this by defining an update( ) method in your applet that does not clear the frame at all before painting it and, if possible clears only the parts of the screen that have changed. Listing 20.9 shows the update ( ) method for the spreadsheet applet. This method uses a boolean value Called full update to determine whether to repaint the whole frame or just a portion of it. When only a portion must be repainted, the appropriate section is redrawn. The particle update occurs when the used selects a cell and the applet places the data associated with the cell in the input area and redraws the cell with a white background. When the entire applet frame needs to be up dated, the update( ) method calls the paint method directly without clearing the frame first.

Updating the frame

```
Public void update(Graphics g) {
If(! FullUpdate) {
Int cx,cy;
g.setFont(titleFont);
for(int i=0;i<rows;i++)
{ for(int j=0;j<columns;j++)
  { if (cells[i][j].needRedisplay) {
            cx=(j * cellWidth)+(fontSize*2) + rowLabelWidth;
            cy=((i+1) cellHeight)+(fontSize*2)+titleHeight;
cells[I][j].paint(g.cx,cy);
}
}
}
}
else {
paint(g);
fullUpdate = false;
}
}
```

27.7 Understanding Layout Managers

All of the component that we have shown so far have been positioned by the default layout manager. As we mentioned at the beginning of this chapter , a layout manager automatically arranges your controls within a window by using some type of algorithm. If you have programmed for other GUI environments, such as Windows, then you are accustomed to laying out your controls by hand. While it is possible to lay out Java controls by hand too, you generally won't want to, for two main reasons. First, it is very tedious to manually layout a large number of components. Second, sometimes the width and height information is not yet available when you need to arrange some control, because the native toolkit component haven't been realized. This is a chicken-and-egg situation. it is pretty confusing to figure out when it is okay to use the size of a given component to position it relative to another.

Each Container object has a layout manager associated with it. A layout manager is an instance of any class that implements the LayoutManager interface. The layout manager is set by the setLayout() method. If no call to setLayout() is made, then the default layout manager is used. Whenever a container is resized ,the layout manager is used to position each of the components within it.

The setLayout() method has the following general form:

```
void setLayout(LayoutManager layoutObj)
```

Here, **layoutObj** is a reference to the desired layout manager. If you wish to disable the layout manager and position components manually, pass null for **layoutObj**. If you do this, you will need to determine the shape and position of each component manually, using the setBounds() method defined by Component. Normally, you will want to use a layout manager.

Each layout manager keeps track of a list of components that are stored by their names. The layout manager is notified each time you add a component to a container. Whenever the container needs to be resized, the layout manager is consulted via its minimumlayoutSize() and preferredLayoutSize() methods.Each component that is being managed by a layout manager contains the getPreferredSize() and getMinimumSize() methods. These return the preferred and minimum size required to display each component. The layout manager will honor these requests if at all possible, while maintaining the integrity of the layout policy. You may override these methods for controls that you subclass. Default value are provided otherwise.

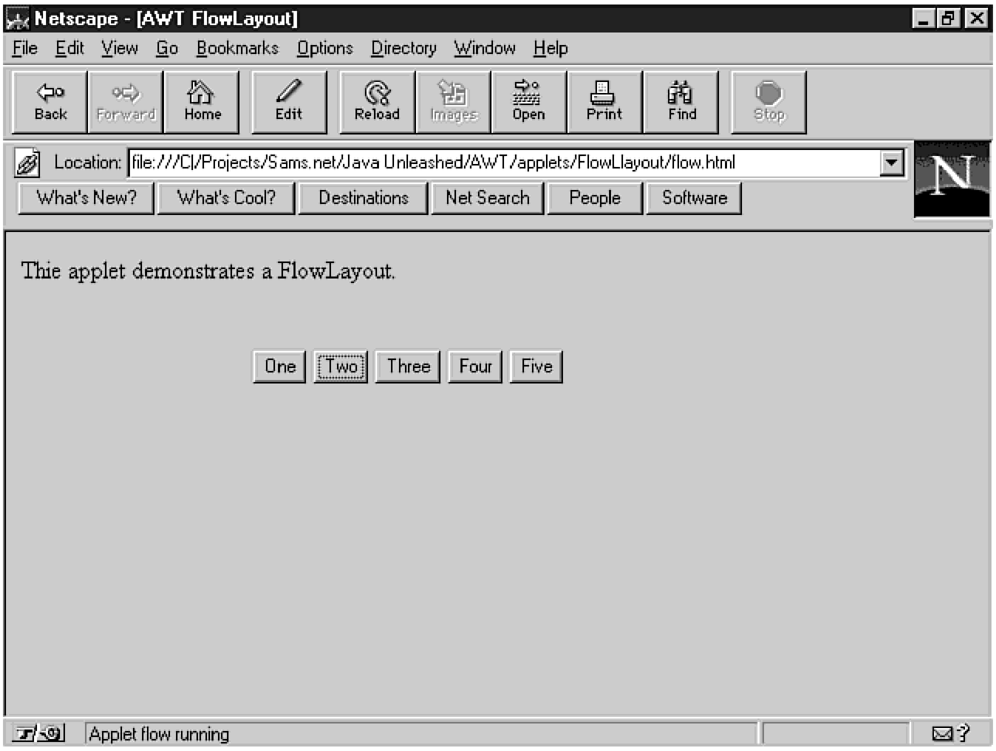
Java has several predefined LayoutManager classes. You can use layout manager that best fits your application.

The FlowLayout Class

The FlowLayout class allows you to lay out controls in rows. Controls are placed in rows as long as there is room. After a row has been filled, subsequent controls are placed in the next row. Figure 16.7 shows the flow layout applet.

The flow applet creates a simple FlowLayout:

```
import java.awt.*;

public class flow extends java.applet.Applet {
   public void init() {
      setLayout( new FlowLayout() ) ;
      add( new Button( "One"   ) ) ;
      add( new Button( "Two"   ) ) ;
      add( new Button( "Three" ) ) ;
      add( new Button( "Four"  ) ) ;
      add( new Button( "Five"  ) ) ;
   }
}
```


Netscape - [AWT FlowLayout]
File Edit View Go Bookmarks Options Directory Window Help
Back Forward Home Edit Reload Images Open Print Find Stop
What's New? What's Cool? Destinations Net Search People Software
One Two Three Four Five

Location: file:///C|/Projects/Sams.net/Java Unleashed/AWT/applets/FlowLlayout/flow.html

This applet demonstrates a FlowLayout.

Applet flow running

In this applet, the FlowLayout constructor is called with no parameters. There are two other overloaded constructors for the FlowLayout class. These allow you to fine tune the FlowLayout to meet your particular needs.

The first constructor takes one parameter. By passing FlowLayout.LEFT, FlowLayout.CENTER, or FlowLayout.RIGHT, you specify the alignment for the controls. The default alignment (when you don't specify one) is FlowLayout.CENTER. Therefore, to align the buttons on the left in the flow applet, you replace the call to setLayout() with this call:
setLayout(new FlowLayout(FlowLayout.LEFT)) ;

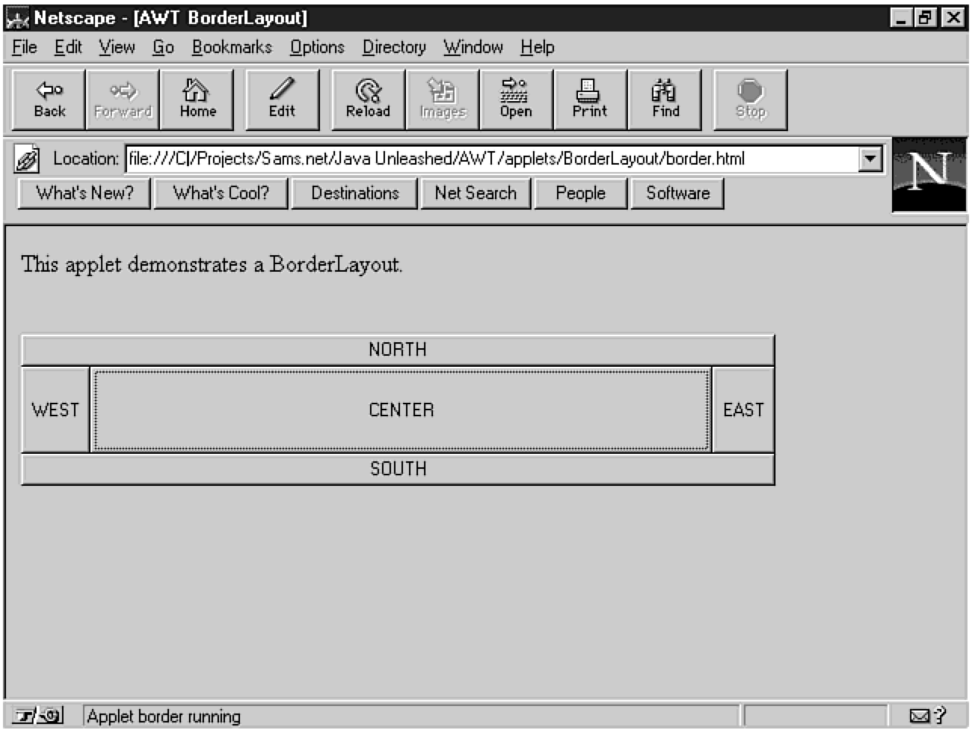
Layouts also give you control over the amount of space between controls. The FlowLayout() method fills the first row and then each subsequent row as necessary. If the layout requires more than one row, you can specify the vertical spacing as well. The constructor that does this takes an alignment parameter followed by two parameters specifying the spacing between controls. To make the flow applet place its controls centered with ten pixels of horizontal gap and five pixels of vertical gap, use the following code:

setLayout(new FlowLayout(FlowLayout.CENTER, 10, 5)) ;

The BorderLayout Class

The AWT BorderLayout class places controls so that they fill their Container object. The controls are placed according to a geographic position that you specify. Controls can be placed on the north, south, east, and west edges of the Container. You can also place a control in the center of the Container. The centered control is then expanded to fill the remaining space. Figure shows the border applet with five controls. The border applet creates a simple BorderLayout:

```
import java.awt.*;

public class border extends java.applet.Applet {
   public void init() {
      setLayout( new BorderLayout() ) ;
      add( "North",  new Button( "NORTH"  ) ) ;
      add( "South",  new Button( "SOUTH"  ) ) ;
      add( "East",   new Button( "EAST"   ) ) ;
      add( "West",   new Button( "WEST"   ) ) ;
      add( "Center", new Button( "CENTER" ) ) ;
   }
}
```


Netscape - [AWT BorderLayout]
File Edit View Go Bookmarks Options Directory Window Help
Back Forward Home Edit Reload Images Open Print Find Stop
Location: file:///C|/Projects/Sams.net/Java Unleashed/AWT/applets/BorderLayout/border.html
What's New? What's Cool? Destinations Net Search People Software
NORTH
WEST
CENTER
EAST
SOUTH
Applet border running

This applet demonstrates a BorderLayout.

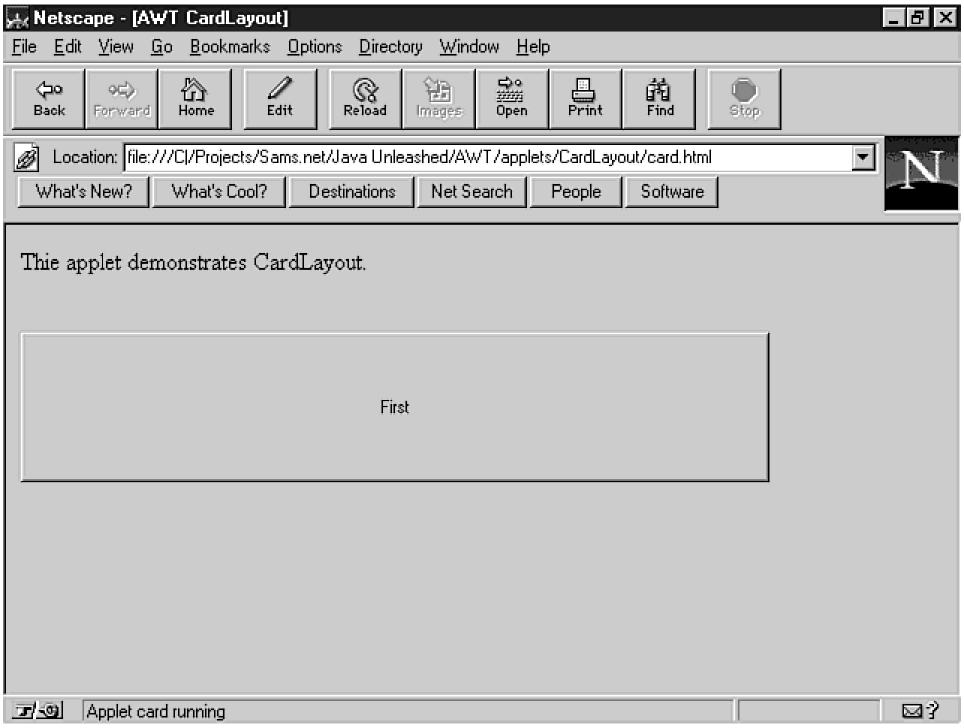
When you create a BorderLayout, you can specify vertical and horizontal gap values as you can with the setLayout() method (described in the flow applet).

The CardLayout Class

The AWT CardLayout class is unique. Rather than placing multiple controls in a Container object, this layout displays the controls one at a time (much like the familiar deck of cards in the ubiquitous Solitaire game). The controls that are displayed may, in fact, be composite controls. Therefore, you can present entirely different sets of controls to the user in a manner similar to the tabbed dialog boxes that Microsoft Windows uses. Figure shows an applet with five buttons laid out in a CardLayout fashion.

The card applet creates a CardLayout with five buttons:

```
import java.awt.*;

public class card extends java.applet.Applet {

   CardLayout layout ;

   public void init() {
      layout = new CardLayout() ;
      setLayout( layout ) ;

      add( new Button( "First"  ) ) ;
      add( new Button( "Second" ) ) ;
      add( new Button( "Third"  ) ) ;
      add( new Button( "Fourth" ) ) ;
      add( new Button( "Fifth"  ) ) ;
   }

   public boolean action( Event evt, Object arg ) {

      if ( evt.target instanceof Button ) {
         layout.next(this) ;
         return true ;
         }
      return false ;
   }
}
```


Netscape - [AWT CardLayout]
File Edit View Go Bookmarks Options Directory Window Help
Back Forward Home Edit Reload Images Open Print Find Stop
What's New? What's Cool? Destinations Net Search People Software
First

Location: file:///C|/Projects/Sams.net/Java Unleashed/AWT/applets/CardLayout/card.html

Thie applet demonstrates CardLayout.

Applet card running

The card applet places five Button objects in a CardLayout. When any button is pressed, the action() method calls the CardLayout's next() method to display the next card in order. This layout also allows you to label the various controls that are added.

The add() method takes an optional String parameter that labels the controls you add. The following call adds a Button labeled my button with the label *Push Me*:

add("my button", new Button("Push Me") ;

Once the controls have labels, you can display them without having to show them in order, without calling the layout's next() method. To display my button, simply call the layout's show() method:
show(this, "my button") ;

The show() method displays a specified control; the next() method displays the next control in order. The CardLayout class provides the following functions to navigate the controls in the layout:

- first(Container)
- last(Container)
- next(Container)
- previous(Container)
- show(Container, String)

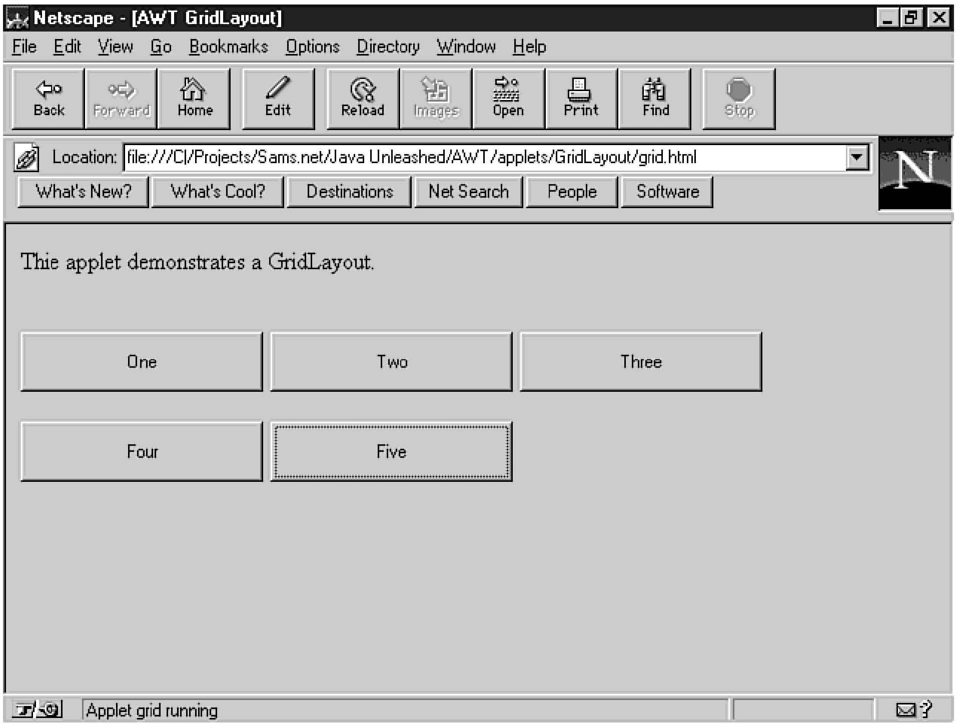
All these navigational functions take a reference to a Container object as a parameter. The show() method takes a String containing the label given to the control when it was added.

The GridLayout Class

As its name suggests, the GridLayout class places controls in the Container in a grid. It is important to note that this is a *regular* grid-all the grid cells are the same size. The applet in Figure shows a GridLayout.

The grid applet defines a grid with two rows and three columns. The add() method adds each control starting with row 1, column 1 followed by row 1, column 2 and so on.

```
import java.awt.*;

public class grid extends java.applet.Applet {
   public void init() {
      setLayout( new GridLayout( 2, 3 ) ) ;

      add( new Button( "One"   ) ) ;
      add( new Button( "Two"   ) ) ;
      add( new Button( "Three" ) ) ;
      add( new Button( "Four"  ) ) ;
      add( new Button( "Five"  ) ) ;

   }
}
```


Netscape - [AWT GridLayout]
File Edit View Go Bookmarks Options Directory Window Help
Back Forward Home Edit Reload Images Open Print Find Stop
Location: file:///C|/Projects/Sams.net/Java Unleashed/AWT/applets/GridLayout/grid.html
What's New? What's Cool? Destinations Net Search People Software
One
Two
Three
Four
Five
Applet grid running

Thie applet demonstrates a GridLayout.

You can also create a GridLayout with vertical and horizontal gap values by using the setLayout() method as you do with the FlowLayout and BorderLayout classes.

The GridBagLayout Class

The GridBagLayout class is complex enough to fill an entire chapter by itself. This class was added to the AWT very late in the Java beta. Therefore, many early acceptors of Java didn't use this layout at all. Some early books omit it completely.

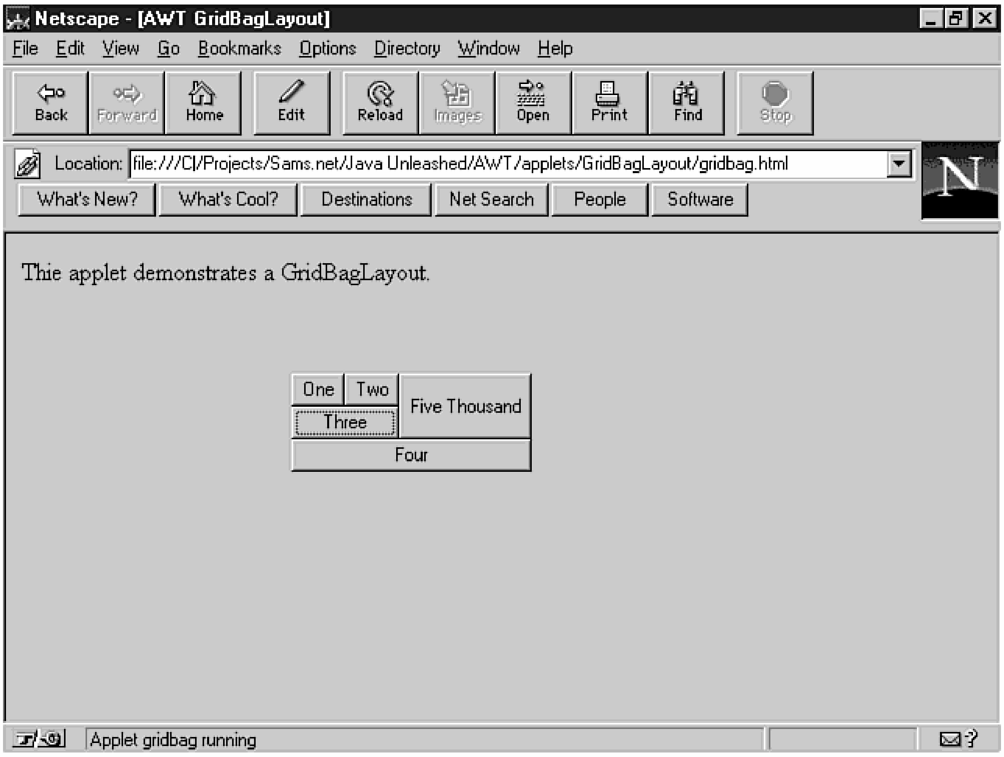
Of all the layouts offered by the AWT, GridBagLayout is the most versatile. Despite its funny name, once you learn how to use GridBagLayout, it will become an indispensable part of your Java toolkit.

Like GridLayout, GridBagLayout places controls in a Container in a grid. The difference is that in a GridBagLayout, controls can span any number of grid cells vertically, horizontally, or both. Controls can be placed in any grid cell. Cells can be of differing sizes as well.

The gridbag applet displays five buttons in a GridBagLayout arrangement:

```
import java.awt.*;

public class gridbag extends java.applet.Applet {

   public void init() {
        Button b1 = new Button( "One"   ) ;
        Button b2 = new Button( "Two"   ) ;
        Button b3 = new Button( "Three" ) ;
        Button b4 = new Button( "Four"  ) ;
        Button b5 = new Button( "Five Thousand"  ) ;
        GridBagLayout gridbag = new GridBagLayout();
        setLayout( gridbag ) ;
        {
    GridBagConstraints c = new GridBagConstraints();
        c.fill      = GridBagConstraints.BOTH ;
        c.gridx     = 1 ;
        c.gridy     = 1 ;
        gridbag.setConstraints(b1, c);
        add( b1 ) ;
        }
        {
       GridBagConstraints c = new GridBagConstraints();

        c.anchor    = GridBagConstraints.WEST ;
        c.gridx     = 2 ;
        c.gridheight = 2 ;
        gridbag.setConstraints(b2, c);
        add( b2 ) ;
        }
        {
      GridBagConstraints c = new GridBagConstraints();

        c.fill      = GridBagConstraints.BOTH ;
        c.gridx     = 1 ;
        c.gridy     = 2 ;
        c.gridwidth  = 2 ;
        gridbag.setConstraints(b3, c);
        add( b3 ) ;
        }
        {
      GridBagConstraints c = new GridBagConstraints();
        c.fill      = GridBagConstraints.BOTH ;
        c.gridx     = 1 ;
        c.gridy     = 3 ;
        c.gridwidth  = 3 ;
        gridbag.setConstraints(b4, c);
        add( b4 ) ;
```

```
          }
          {
        GridBagConstraints c = new GridBagConstraints();
          c.fill      = GridBagConstraints.VERTICAL ;
          c.gridx     = 3 ;
          c.gridy     = 1 ;
          c.gridheight = 2 ;
          gridbag.setConstraints(b5, c);
          add( b5 ) ;
          }
        }
      }
```



file:///C|/Projects/Sams.net/Java Unleashed/AWT/applets/GridBagLayout/gridbag.html

This applet demonstrates a GridBagLayout.

Applet gridbag running

The key to using GridBagLayout is the GridBagConstraints class. This class is used to encapsulate information about each control that is added to the layout. Setting the class data members determines how the controls will be placed.

To use a GridBagLayout, you must create a GridBagConstraints object. Then set the data members of the GridBagConstraints object to appropriately lay out the given control. Next, call the GridBagLayout's setConstraints() method to associate a GridBagConstraints object with a control. Finally, add the control.

The following GridBagConstraints public data members determine how your controls are placed:

- The anchor member specifies how a control is displayed if it is smaller than the grid cell in which it is placed. This member can be set to CENTER, NORTH, SOUTH, EAST, WEST, NORTHEAST, SOUTHEAST, NORTHWEST, or SOUTHWEST.

- The fill member lets a control grow to fill its allotted grid cells if the cells are larger than the control's default size. The choices for this member are BOTH, HORIZONTAL, VERTICAL, and NONE.
- The gridheight and gridweight members determine how many grid cells a control takes up.

- The gridx and gridy members specify the row and column (in grid coordinates) at which to place the control.

- The ipadx and ipady members specify the vertical and horizontal gap (or padding) for components.

- The weightx and weighty members specify how *excess* space is assigned to the various components if the container in which they are embedded is resized.

- The Insets member is a class that specifies the *margins* of a Container that has a GridBagLayout.

By using the GridBagLayout and GridBagConstraints classes, you can produce layouts to meet nearly all your needs. If these are not flexible enough for you, there is always the option of creating your own LayoutManager. You create your own LayoutManager by creating a class (subclassed from Object) that implements the LayoutManager interface.

27.8 Summary

- Component is an abstract class that encapsulates all of the attributes of a visual component.

- Container is a subclass of component

- A Layout manager automatically positions components within a container thus, the appearance of a window is determined by a combination of the controls that it contains and the layout manager used to position them

27.10 Brain Storm

1. Which one is the base class of component class?
2. What are the derived class of container class?
3. What is Painting?
4. Why we use repaint() method?
5. Explain about Layouts?
6. How we can declare the different types of Layout in applet?

Lecture - 28

EventListener

Objectives

In this lecture you will learn the following

- Event class
- EventListener
- Interfaces of EventListener

Lecture - 28

28.1 Snap Shot

In this lecture you will learn about Events , Event handling methods, EventListener method, and We also know about Examples of EventListener.

28.2 Event Class

GUI Event Basics AWT Event Flow, Event and Listener types

The classes that represent events are at the core of Java's event handling mechanism. Thus, we begin our study of event handling with a tour of the event classes. As you will see, they provide a consistent, easy-to-use means of encapsulating events.

At the root of the Java event class hierarchy is EventObject, which is in java.util. It is the superclass for all events. Its one constructor is shown here:

EventObject(Object src)

Here, src is the object that generates this event.

EventObject contains two methods: **getSource()** and **toString().**The **getSource()** Method returns the source of the event. Its general form is shown here:

Object getSource()

As expected, **toString()** returns the string equivalent of the event.

The class AWTEvent, defined within the java.awt package, is a subclass of **EventObject**. It is the superclass(either directly or indirectly)of all AWT-based events used by the delegation event model. Its **getID()** method can be used to determine the type of the event. The signature of this method is shown here:

int getID()

- EventObject is a superclass of all events.
- AWTEvent is a superclass of all AWT events that are handled by the delegation event model.

The package java.awt.event defines several types of events that are generated by various user interface elements. The Most important event classes are given in the below table.

Event Class	Description
ActionEvent	Generated when a button is pressed, a list item is Double-clicked, or a menu item is selected.
AdjustmentEvent	Generated when a scroll bar is manipulated.
ComponentEvent	Generated when a component is hidden, moved, resized, or becomes visible.
ContainerEvent	Generated when a component is added to or removed from a container.
FocusEvent	Generated when a component gains or loses keyboard focus.
InputEvent	Abstract super class for all components input event classes
ItemEvent	Generated when a checkbox or list item is clicked; also occurs when a choice selection is made or a checkable menu item is selected or deselected.
KeyEvent	Generated when input is received from the keyboard.
MouseEvent	Generated when the mouse is dragged, moved, clicked, pressed, or released; also generated when the mouse enters or exits a component.
TextEvent	Generated when the value of a text area or text area or text field is changed.
WindowEvent	Generated when a window is activated, closed, deactivated, deiconified, iconified, opened, or quit.

Table: *Main Event Classes in java.awt.event*

28.3. Event Sources

A source is an object that generates an event. This occurs when the internal state of that object changes in some way. Sources may generate more than one type of event.

A source must register listeners in order for the listeners to receive notifications about a specific type of event. Each type of event has its own registration method.

Here is the general form:

Public void addTypeListener(TypeListener el)

Here, Type is the name of the event and el is a reference to the event listener. For example, the method that registers a keyboard event listeners is called addKeyListener(). The method that registers a mouse motion listener is called addMouseMotionListener(). When an event occurs, all registered listeners are notified and receive a copy of the event object. This is known as multicasting the event. In all cases, notifications are sent only to listeners to receive them.

Some sources may allow only one listener to register. The general form of such a method is this:

Public void addTypeListner(TypeListener el)
Throws java.util.TooManyListnersException

Here, Type is the name of the event and el is a reference to the event listener. When such an event occurs, the registered listener is notified. This is known as unicasting the event.

A source must also provide a method that allows a listener to unregistered an interest in a specific type of event. The general form of such a method is this:

Public void removeTypeListener(TypeListener el)

Here, Type is the name of the event and el is a reference to the eventlistener. For example, to remove a keyboard listener, you would call removeKeyListener().
The methods that add or remove listeners are provided by the source that generates events. For example, the Component class provides methods to add and remove keyboard and mouse event listeners.

Some type of Event classes

The ActionEvent Class

An ActionEvent is generated when a button is pressed, a list item is double clicked, or a menu Item is selected. The ActionEvent class defines four integer constants that can be used to Identify any modifiers associated with an action event:

ALT_MASK, CTRL_MASK,META_MASK,and SHIFT_MASK.In addition,

there is an integer constant, ACTION_PERFORMED, which can be used to

identify action events.

ActionEvent has these two constructors:

ActionEvent(Object src, int type,String cmd)

ActionEvent(Object src, int type, String cmd, int modifiers)

Here, src is a reference to the object that generated this event. The type of the event is specified by Type, and its command string is cmd. The argument modifiers indicates which modifiers keys(ALT,CTRL,META, and/or SHIFT)were pressed when the event was generated.

You can obtain the command name for the invoking **ActionEvent** Object by using the **getActionCommand()** method, shown here:

String getActionCommand()

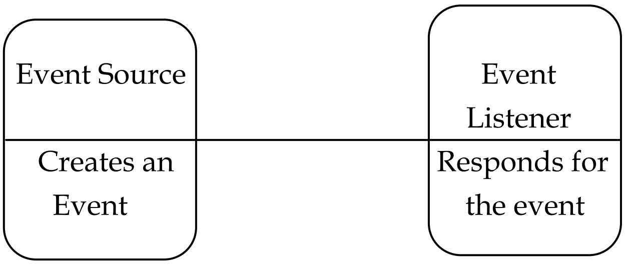
For example, when a button is pressed, an action event is generated that has a command name equal to the label on that button.

The **getModifiers()** method returns a value that indicates which modifier keys(ALT,CTRL,META,and/or SHIFT) were pressed when the event was generated.Its form is shown here:

int getModifiers()

28.4 Event Listeners

In the program, a Java object can respond to an event. The object is called the event Listener. The object that creates the event is called Event Source.



A listener is an object that is notified when an event occurs. It has two major requirements. First, it must have been registered with one or more sources to receive notifications about specific types of events. Second, it must implement methods to receive and process these notifications.

The methods that receive and process events are defined in a set of interfaces found in java.awt.event. For example, the MouseMotionListener interface defines two methods to receive notifications when the mouse is dragged or moved .Any object may receive and process one or both of these events if it provides an implementation of this interface.

28.5. Using EventListener Examples

1. The ActionListener Interface

This interface defines the actionPerformed() method that is invoked when an action event occurs. Its general form is shown here:

void actionPerformed (ActionEvent ae)

2. The Adjustment Listener interface

This interface defines the adjustementValueChanged() method that is invoked when an adjustment event occurs. Its general form is shown here:

void adjustmentValue Changed (AdjustementEvent ae)

3. The ComponentListener Interface

This interface defines four methods that are invoked when a component is resized, moved shown ,or hidden.. Their general forms are shown here:

Void componentResized (componentEvent ce)

Void componentMoved (componentEvent ce)

Void componentShown (componentEvent ce)

Void componentHidden (componentEvent ce)

4. The ContainerListener interface

This interface contains two methods. When a component is added to a container, componentAdded ( ) is invoked. When a component is removed from a container, componentRemoved ( ) is invoked. Their general forms are shown here:

```
        Void componentAdded (ContainerEvent ce )
        Void componentRemoved (ContainerEvent ce)
```

5. This FocusListener Interface

This interface defines two methods, when a component obtains keyboard focus, focusGained( ) is invoked . When a component loses keyboard focus, focusLost( ) is called. Their general forms are shown here:

```
Void focusGained (FacusEvent fe)
        Void foucusLost(FocusEvent fe)
```

6. The ItemListener Interface

This interface defines the itemStateChanged ( ) method that is invoked when the state of an item changes. It general form is shown here:

```
        Void itemStateChanged (ItemEvent ie)
```

7. The KeyListener Interface

This interface defines three methods. The KeyPressed ( ) and KeyReleased ( ) methods are invoked when a key is pressed and released respectively. The KeyTyped ( ) methods is invoked when a character has been entered.

For example, if a user presses and releases the a key, three events are generated in sequence, key pressed, typed and released. If a user presses and releases the Home key, two key events are generated in sequence key pressed and released.

The general forms of these methods are shown here:

```
        void keyPressed (keyEvent ke)
        void keyReleased (keyEvent ke)
        void keyTyped (keyEvent ke)
```

8. The mouse Listener Interface

This interface defines five methods. If the mouse is pressed and released at the same point, mouseCliked( ) is invoked. when the mouse enters a component the mouseEntered ( ) methods is called. When it leaves, mouseExited ( ) is called. The mousePressed ( ) and mouseRelased ( ) methods are invoked when the mouse is pressed and released respectively.

The general forms of these methods are shown here:

```
void mouseClicked(MouseEvent me)
void mouseEntered (MouseEvent me)
void mouseExited(MouseEvent me)
void mousePressed(MouseEvent me)
void mouseReleased (mouseEvent me)
```

9. The MouseMotionListener Interface

This interface defines two methods. The mouseDragged ( ) method is called multiple times as the mouse is dragged. The mouseMoved() method is called multiple times as the mouse is moved. their general forms are shown here:

```
void mouseDragged(MouseEvent me)
void mouseMoved(MouseEvent me)
```

10. The TextListener Interface

This interface defines the textChanged() method that is invoked when a change occurs in a text area or textfield. Its general form is shown here:

```
void textChanged(TextEvent te)
```

11. The WindowListener Interface

This interface defines seven methods. The windowActivated() and windowDeactivated() methods are invoked when a window is activated or deactivated, respectively. If a window is iconified, the windowIconified() method is called. when a window is deiconified, the windowDeiconified() method is called. When a window is opened or closed, the windowOpened() or windowClosed() methods are called, respectively. The windowClosing() method is called when a window is being closed. The general forms of these methods are

```
void windowActivated(WindowEvent we)
void windowClosed(WindowEvent we)
void windowClosing(WindowEvent we)
void windowDeactivates(WindowEvent we)
void windowDeiconified(WindowEvent we)
void windowIconified(WindowEvent we)
void windowOpened(WindowEvent we)
```

programming using the delegation event model is actually quite easy. Just follow these two steps:

- ✈ Implement the appropriate interface in the listener so that it will receive the type of event desired.
- ✈ Implement code to register and unregistered the listener as a recipient for the event notifications.

Remember that a source may generate several types of events. Each event must be registered separately. Also, an object may register to several types of events, but it must implement all of the interfaces that are required to receive these events.

28.6 Short Summary

- ♣ When a program is running on a machine the user can interrupt through the input devices. These interruptions are called events.
- ♣ In Java each events is considered as an object of a class.
- ♣ In the delegation model, an event is an object that describes a state change in a source.
- ♣ A source is an object that generates an event.
- ♣ A listener is an object that is notified when an event occurs.

28.7 Brain Storm

1. What is Event class?
2. In java.awt package what are all the important event classes are placed?
3. What are the interfaces we used for EventListener?

Lecture - 29

Controls & Events

Objectives

In this lecture you will learn the following

- ❖ Using Widgets
- ❖ Making Windows
- ❖ Processing Mouse Events
- ❖ Handling Keyboard Events
- ❖ Canvas

Lecture - 29

29.1 Snap Shot

In this lecture you will learn about various classes such as Button, Checkbox, List, Choice and about mouse & keyboard events and about canvas.

29.2 Examples for using Widgets (Button, Checkbox, List and Choice)

The button class demonstrates a push-button that can only have a textual label.

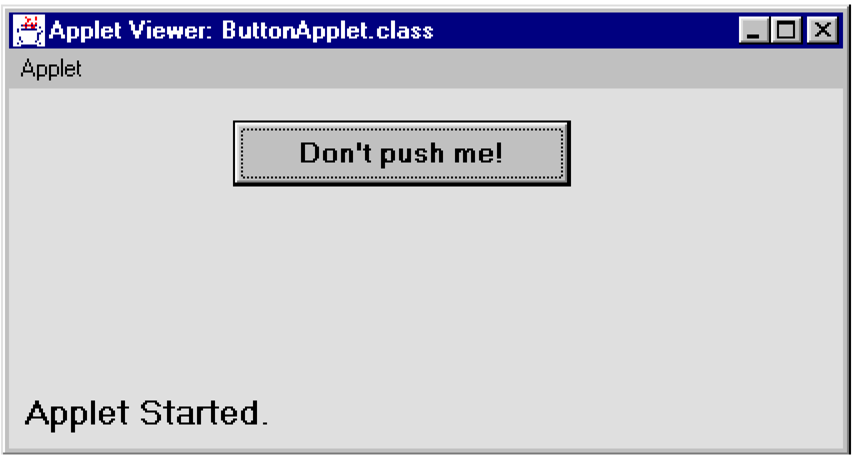
```
Button ( )
Button(String label)
```

These constructors are used for creating buttons.

```
String getLabel( )
void setLabel(String label)
```

These methods can be used to get and set the textual label.

```
// Illustrating button

import java.awt.*;
import jave.applet.*;
public class ButtonApplet extends Applet
{
        public void init ( )
        {
                Button button = new Button("Don't push me");
                add(button);
        }
}
```



Checkbox and Checkbox Group

The Checkbox class implements a GUI checkbox with a textual label. A Checkbox object can be in one of the two states:

- true: meaning that it is checked
- false: meaning that it is unchecked

The Checkbox class has the following constructors

```
        Checkbox( )
        Checkbox(String label)
        Checkbox(Stiring label, boolean state)
        Checkbox(String label, boolean state, CheckboxGroup group)
```

If the state is not explicitly specified in the appropriate constructor the initial state is unchecked. The state of the checkbox can be toggled by clicking on the checkbox.

A checkbox can be incorporated in a CheckboxGroup to implement radio buttons. Unless the CheckboxGroup is specified in the appropriate constructor, the checkbox is not part of any CheckboxGroup.

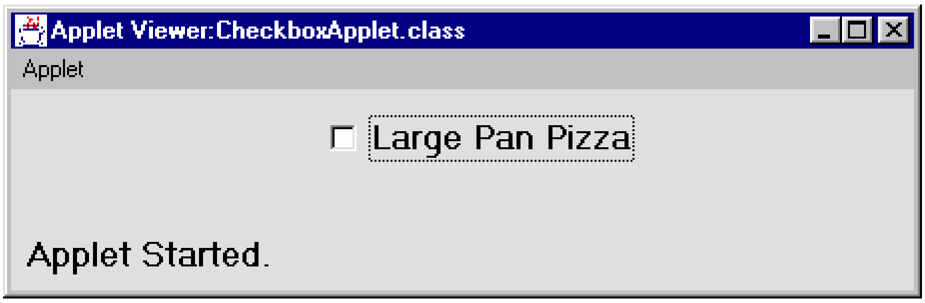
```
// Illustrating Checkbox

import java.awt.*;
import java.applet.*;
public class CheckboxApplet extends Applet
{
        public void init ( )
        {
                Checkbox option = new Checkbox("Large pan Pizza");
                option.setState(true);
                add(option);
        }
}
```

The following methods

boolean getState ( ) to read and change the state of a checkbox
void setState(boolean state)

String getLabel ( ) → to read and change the texual label of a check void
setLabel (String label) ↗

CheckboxGroup getCheckboxGroup( )
Void setCheckboxGroup(CheckboxGroup g)

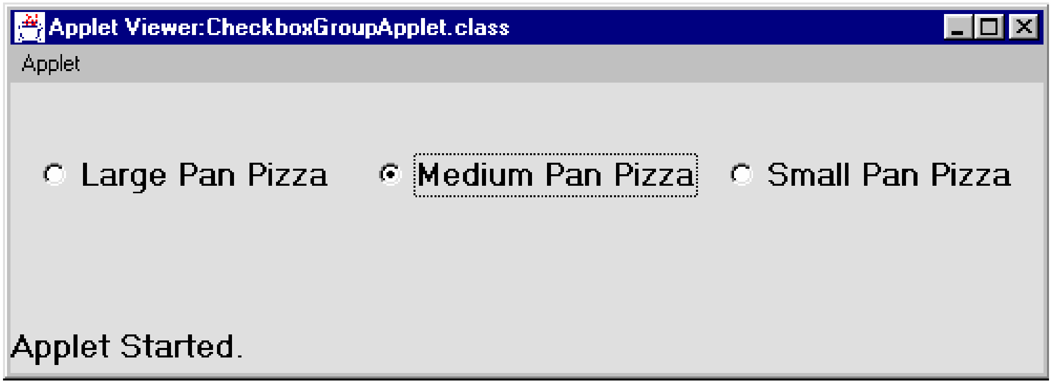


The class java.awt.CheckboxGroup can be used to control the behaviour of a group of checkboxes. Such a group only allows a single selection. Clicking on a different check box in a group automatically unchecks the previous check box. Such mutually exclusive check boxes are often called radio buttons.
The following methods return the currently selected check box and set a particular check box as the current selection in a CheckboxGroup:

```
        CheckboxGroup getSelectedCheckbox( )
        void setSelectedChekbox(Checkbox box)
```

CheckboxGroup object does not have a graphical representation, and its not a subclass of Component. The CheckboxGroup is just a class to implement mutual exclusion among a set of checkboxes.
// Illustration of Radio buttons

```
import java. awt.*;
import java.applet.*;
public class CheckboxGroupApplet extends Applet
{
        public void init ( )
        {
CheckboxGroup pizzaGroup = new CheckboxGroup ( );
CheckboxGroup cbLarge = new CheckboxGroup("Large Pan Pizza", pizzaGroup, false);
CheckboxGroup bMedium=new CheckboxGroup("MediumPan Pizza",true,pizzaGroup);
CheckboxGroup cbSmall = new CheckboxGroup("Small Pan Pizza", false);
CbSmall.setCheckboxGroup(pizzaGroup);
add(cbLarge);
add(cbMedium);
add(cbSmall);
}
}
```



Applet Started.

Choice

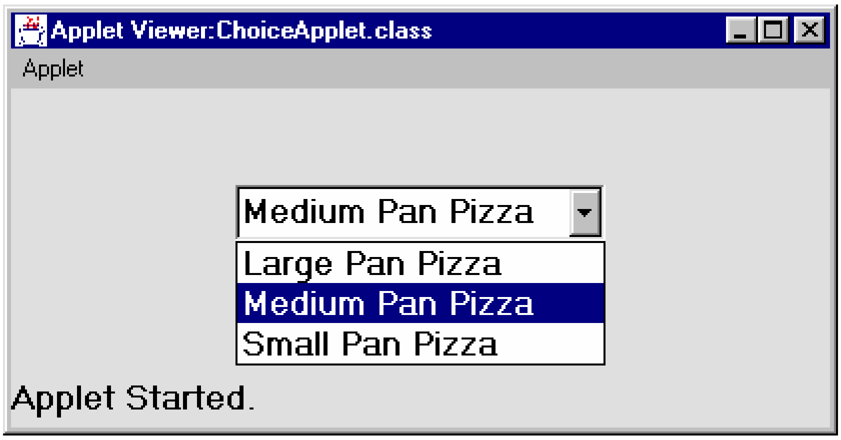
The Choice class implements a pop-up menu of choices. Only the current choice is visible in a Choice component. The choice can be changed by popping up the list of choices by clicking on the menu and selecting another item on the choice list.

Constructing a pop-up menu of choices involves the following steps:
Creating choice object using the single default constructor provided.
Adding the items using the add( ) method.
void add(String item)

Int getItemCount	Returns the number of items in the pop-up menu
String getItem(int index	Returns the item at a particular index in the pop-up menu. Start index is 0.
Int getSelectedIndex( )	Returns the index of the currently selected item in the pop-up menu
String getSelectedItem( )	Returns the index of the currently selected item in the pop-up menu.
void select(int pos)	Makes the item at the given position in the pop-up menu the current choice.
void select(String str)	Makes the item with the argument string in the pop-up menu the current choice

The Choice class also defines methods for inserting and removing item from the pop-up menus

```
// Illustrating choice
import java.awt.*;
import java.applet.*;
public class ChoiceApplet extends Applet
{
        public void init ( )
        {
                Choice pizzaChoice = new Choice ( );
                pizzaChoice.add("Large Pan Pizza");
                pizzaChoice.add("Medium Pan Pizza");
                pizzaChoice.add("Small Pan Pizza");
        }
  }
```



Label

A label is a component that displays a single line of read-only, non-selectable text. It does not generate any special events. The label class defines three constructors

```
Label ( )
Label (String text)
Label (String text, int alignment)
```

The alignment of the label in a container can be specified by the following constants of the Label class. The default alignment is left.

```
public static final int LEFT
public static final int CENTER
public static final int  RIGHT
```

The Label class defines accessor methods for reading the current text and changing the text in a label:

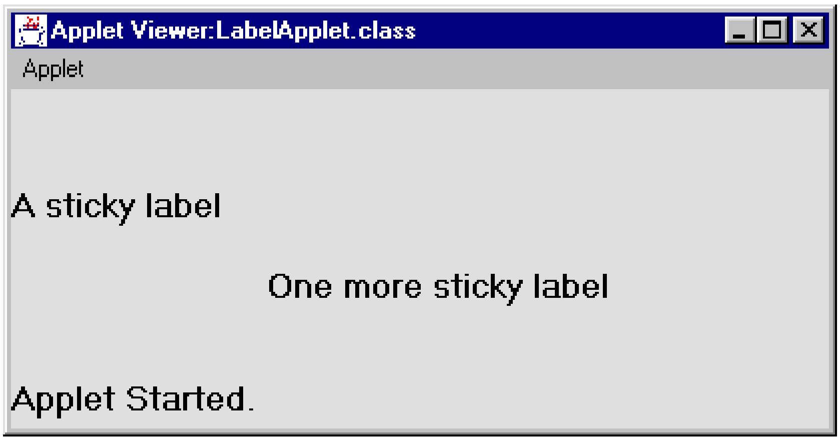
```
String getText ( )
void setText(String text)
```

There are also accessor methods for reading the current alignment and setting a particular alignment for a label:

```
int getAlignment ( )
void setAlignment( int alignment)
```

Example demonstrates two Labels, one with default(left) alignment and another with centered.

```
// Illustrating Labels

import java.awt.*;
import java.applet.*;
public class LabelApplet extends Applet
{
        public void init ( )
        {
                setLayout(new GridLayout(2,1);
                add(new Label("A sticky label");
                add(new Label("One more sticky label", label.CENTER);
}
}
```



```
A sticky label
                    One more sticky label
Applet Started.
```

List

The **List** class implements a scrollable list of text items. Since the list is scrollable the number of items that can be visible in the list box is defined as the number of rows in the list. The list can be of course have any number of text items and a scroll bar appears when necessary to scroll the list.

A List object can be created using one of the following constructors, with options for specifying the number of rows and multiple selection mode

```
List ( )
List (int rows)
List (int rows, boolean multipleMode)
```

Constructing a **list** involves the following steps.

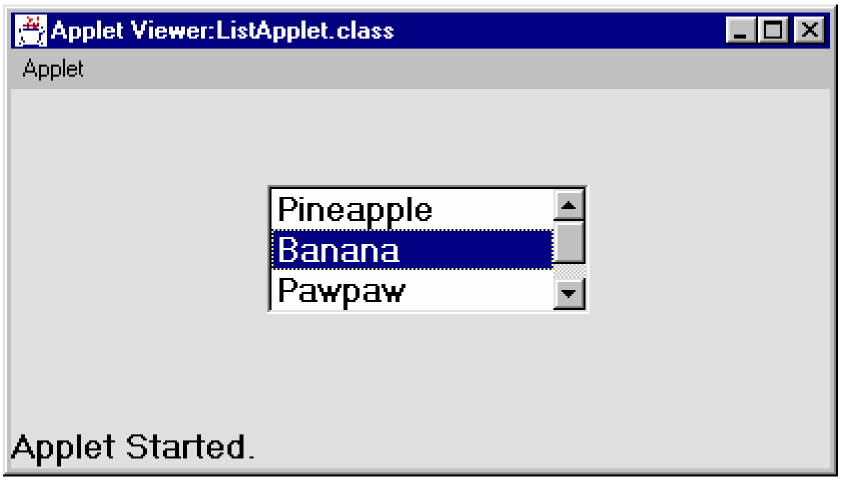
1. Creating a List object, optionally specifying the number of rows and multiple selection mode.
2. Adding the items using the add( ) method. The items are strings.
    a. void add(String item)
    b. void add(String item, int index)

Various accessor methods are defined for scrollbar lists:

Int getRows ( )	Returns the number of rows in the list
Boolean isMultipleMode( )	Returns true if multiple selections are allowed in the list
Int getItemCount( )	Returns the number of items in the list
String getItem(int index)	Returns the item at a particular index in the list
String [ ] getitems ( )	Returns the items in the list
String getSelectedItem ( )	Returns the selected item if one is selected, otherwise returns the null value
String[ ] getSelectedItems( )	Returns the selected items if any, otherwise returns the null value
int getSelectedIndex( )	Returns the index of the selected item if one is selected, otherwise returns the value -1
void select(int index)	Selects the item at a given index in the list
void select(String str)	Selects the items that matches the argument string in the list
void deselect(int index)	Deselects the item at the given index in the list.

The List class defines methods for changing, inserting and removing items.

```
// Illustrating List
import java.awt.*;
import java.applet.*;
public class ListApplet extends Applet
{
        public void init ( )
        {
                String [ ] fruit = {"Mango", "Pineapple","Bannana", "Pawpaw");
                List fruitList = new List(fruit.length -1, true);
                for(int i = 0; i<fruit.length; i++){
                fruitList.add(fruit[i]);
                }
                add(fruitList);
        }
}
```



Scroll Bar

A document window in a text processor usually has two **scroll bars**. A **vertical scrollbar** to scroll up and down. A **horizontal scroll** bar similarly scrolls the document left and right. A scroll bar thus indicates the relative position of the visible contents in relation to the whole document. This is one typical use of scroll bars. A scroll bar can also be used as a controller to specify a value from a given interval.

The Scroll bar provides two constants to indicate orientation

public static final HORIZONTAL
public static final VERTICAL

Three constructors provide various ways to create scroll bars:

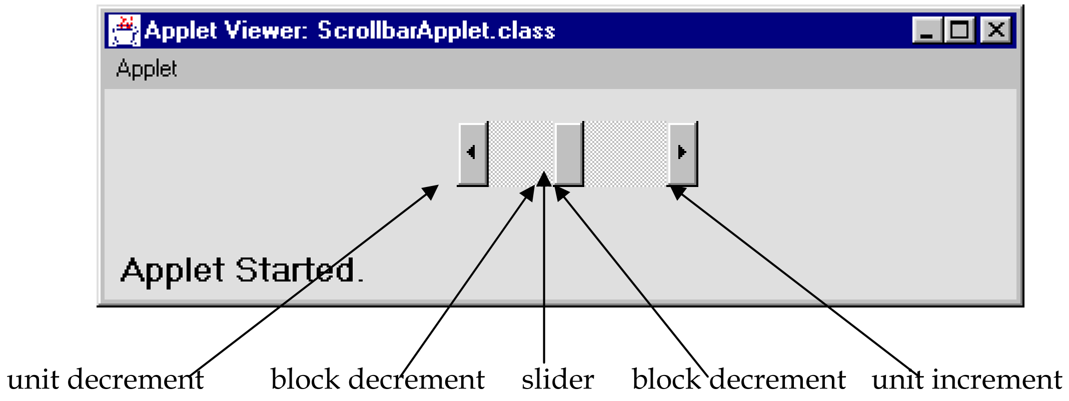
Scrollbar( )
Scrollbar(int orientation)
Scrollbar(int orientation, int value, int visible, int minimum, int maximum)

The visible argument determines the visible width of the slider. The arguments minimum and maximum specify the interval represented by the scroll bar.
The Scroll bars defines an assortment of accessor methods.

int getValues ( )	Returns the current value of the scroll bar
void setValue(int newValue)	Sets the value of the scroll bar to the argument value
int getMinimum( )	Returns the minimum value of the scroll

	bar
void setMinimum(int minimum)	Sets the minimum value for the scroll bar
int getMaximum( )	Returns the maximum value of the scroll bar
void setMaximum(int maximum)	Sets the maximum value for the scroll bar
int getVisibleAmount( )	Returns the visible amount of the scroll bar
void setVisibleAmount(int newAmount)	Sets the visible amount of the scroll bar that is the range of values represented by the width of the scroll bar's slider
int getUnitIncrement( )	Returns the unit increment for the scroll bar
void setUnitIncrement(int v)	Sets the unit increment for the scroll bar
int getBlockIncrement	Gets the block increment for the scrollbar
void setBlockIncrement(int v)	Sets the block increment for the scroll bar

```
// Illustrating Scroll bar

import java.awt.*;
import java.applet.*;
public class ScrollbarApplet extends Applet
{
        public void init( )
        {
                Scrollbar bar = new Scrollbar( );
                Scrollbar(Scrollbar.HORIZONTAL,0,10,-50,100);
                add(bar);
        }
}
```



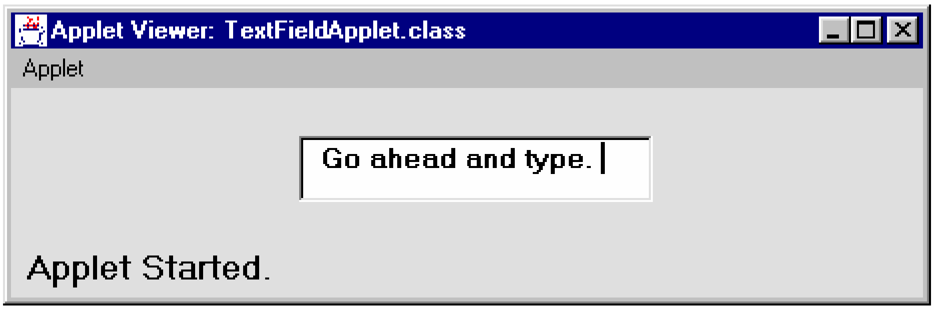
Text field & Text area

The class **TextComponent** provides the functionality for selecting and editing the text. Its **two** subclasses **TextField** and **TextArea** inherit the functionality to implement a **single line of text** or **multiple lines of text** respectively. The text in the component can be read-only or editable.

The TextField class implements a single line of optionally editable text. The size of the text field is measured in columns. Some initial text and a preferred size can be specified when a text field is created.

```
        TextField( )
        TextField(String text)
        TextField(int columns)
        TextField(String text, int columns)
```

```
// Illustrating TextField
import java.awt.*;
import java.applet.*;
public class TextFieldApplet extends Applet
{
        public void init ( );
        {
        TextField entryField = new TextField(18);
        entryField.setFont(new Font("Serif",Font.PLAIN, 12);
        entryField.setText("Go ahead and type");
        add(entryField);
        }
}
```



Go ahead and type.

Applet Started.

The **TextArea** class implements multiple lines of optionally editable text. These lines are separated by '\n'(newline)character. The size of the text area is measured in **columns** and **rows**. When creating text areas, the intial text and the pereferred size of the component can be specified.

Constructors:

```
TextArea( )
TextArea(String text)
TextArea(int rows, int columns)
TextArea(String text, int rows, int columns)
TextArea(String text, int rows, int columns, int scrollbars)
```

Constants:

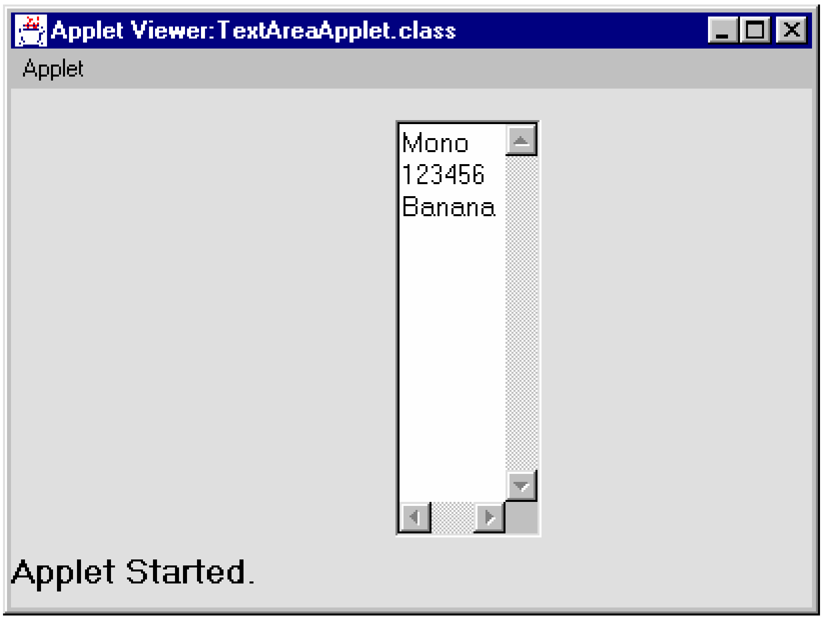
```
public static final int SCROLLBARS_BOTH
public static final int SCROLLBARS_VERTICAL_ONLY
public static final int SCROLLBARS_HORIZONTAL_ONLY
public static final int SCROLLBARS_NONE
```

Both the TextField and TextArea provides the following methods

```
int getColumns( )
void setColumns(int columns)
```

The text component class does not provide any public constructors and is therefore uninstantiable.

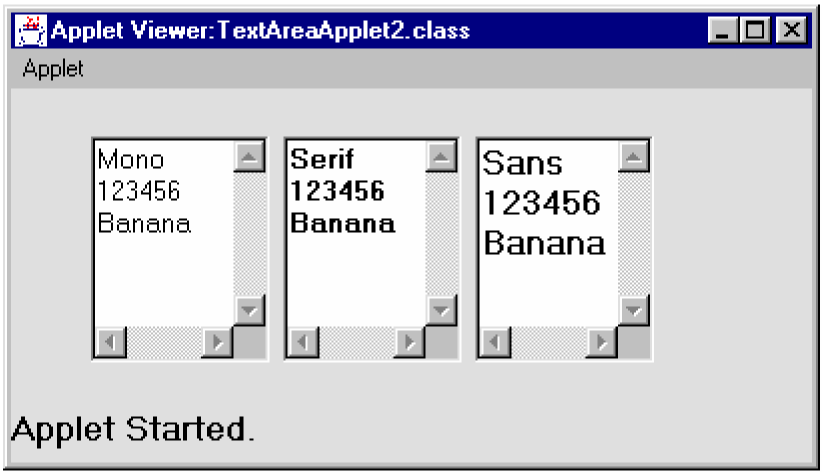
```
// Illustrating TextArea
import java.awt.*;
import java.applet.*;
public class TextAreaApplet extends Applet
{
        public void init ( )
        {
                TextArea display = new TextArea(10,6);
                display.setFont(new("Monospaced",Font.PLAIN, 12);
                display.setText("Mono\n123456\nBanana\n");
                display.setEditable(false);
                add(display);
        }
}
```



```
Mono
123456
Banana
```

Applet Started.

```
// illustrating Text Lines and Variable Pitch Fonts

import java.awt.*;
import java.applet.*;
public class TextAreaAppletTwo extends Applet
{
        public void init ( )
        {
                TextArea display1 = new TextArea(4,6);
                display1.setFont(new("Monospaced",Font.PLAIN, 18);
                display1.setText("Mono\n123456\nBanana\n");

                TextArea display2 = new TextArea(4,6);
                display2.setFont(new("Monospaced",Font.PLAIN, 18);
                display2.setText("Mono\n123456\nBanana\n");

                TextArea display3 = new TextArea(4,6);
                display3.setFont(new("Monospaced",Font.PLAIN, 18);
                display3.setText("Mono\n123456\nBanana\n");

                add(display1);
                add(display2);
                add(display3);
        }
}
```



Mono
123456
Banana

Serif
123456
Banana

Sans
123456
Banana

Applet Started.

29.3 Making Windows

Although creating applets is the most common use for Java's AWT, it is possible to create stand alone AWT based applications, too. To do this simply create an instance of the window or windows you need inside **main( ).** For example, the following program creates a frame window that respond to mouse clicks and keystrokes.

```
/ / Create an AWT based aplication.

Import java.awt.*;
Import java.awt.event.*;
 Import java.applet.*;

/ / Create a frame window.

public class Appwindow extends Frame   {
   String Keymsg  = * *;
   String mousemsg  = * * ;
    int mouseX=30, mouseY=30;

 public Appwindow ( ) {
   addKeyListener (new MyKeyAdapter (this ) ;
   addMouseListener (new MyMouseAdapter (this );
   addWindowListener (new MywindowAdapter ( ) );

}

  public void paint (Graphics g )  {
    g.drawString(keymsg,10,40 );
    g.drawString (mouseG, mouseX,mouseY);

}

/ / Create the window.
Public static viod main   (String args [ ] );  {
  AppWindow appwin = new Appwindow  ( );

Appwin.setSize (new Dimension (300,200) );
Appwin.setTitle ("AnAWT-Based Appocation");
Appwin.setVisible (true );

}
}

class MyKeyAdapter extends KeyAdapter  {
  appWindowappWindow ;
  public MyKeyAdapter (AppWindow appWindow ) {
this.appWindow = appWindow:
 }
 public void keyTyped (KeyEvent Ke) {
appWindow.keymsg + =key.get.KetChar ( );
 appWindow.repaint ( );
  };
}

class MyMouseAdapter extends MouseAdapter {
AppWindow appWindow;
  Public MyMouseAdapter (AppWindow appWindow ) {
 This.appWindow = appWindow;

}

public void mousePressed (MouseEvent  me ) {
appWindow.mouseX = me.getX ( );
appWindow,mouseY =me.getY ( );
appWindow.mousemsg = "Mouse Down at" + appWinxdow.mouseX  +
                                    "," + appWindow.mouseY;
appWindow.repaint ( );
 }
}
```

```
class MyWindowsAdapter extends  WindowAdapter  {
public void windowClosing  (WindowEvent we ) {
System.exit (0);

  }
}
```

Once created a frame, window takes on a life of its own. Notice that **main()** ends with the call to **appwin.setVisble (true).** However, the program keeps running until you close the window. In essence, when creating a windowed application, you will use **main( )** to launch it top level window. After that, your program will function as a GUI-based application not like the console-based programs used earlier.

29.4 Handling keyboard Events

Method	Description	Called with values
keyDown()	Called if a key is pressed	keyDown(Event e, int x)
keyUp()	Called if a key is released	keyUp(Event e, int x)

The KeyEvent Class

A **KeyEvent** is generated when keyboard input occurs. There are three types of key events, which are identified by these integer constants : KEY_PRESSED, KEY_RELEASSED ,and KEY_TYPED. The first two events are generated, when any key is pressed or released. The last event occurs only when a character is generated.

There are many other integer constants that are defined by **KeyEvent**. For example,VK_0 through VK_9 and VK_A through VK_Z define the ASCII equivalent of the nubers and letters.Here are some others:

VK_ENTER VK_ESCAPE VK_CANCEL VK_UP

VK_DOWN VK_LEFT VK_RIGHT VK_PAGE_DOWN

VK_PAGE_UP VK_SHIFT VK_ALT VK_CONTROL

The VK Constants specify vitrual key codes and are independent of any modifiers,such as control,shift ,or alt.

KeyEvent is a subclass of InputEvent and has these **two** constructors :

KeyEvent(Component src,int type,long when,int modifiers,int code)

KeyEvent(Component src,int type,long when,int modifiers,int code,char ch)

Here, **src** is a reference to the component that generated the event. The type of the event is specified by type. The system time at which the key was pressed is passed in when. The modifiers argument indicates which modifiers were pressed when this key event occurred. The virtual key code, such as VK_UP,VK_A ,and so forth ,is passed in code. The character equivalent (if one exists) is passed in ch.If no valid character exists, then ch contains CHAR_UNDEFINED. For KEY_TYPED events, code will contain VK_UNDEFINED.

The KeyEvent class defines several methods,but the most commonly used ones are getKeyChar(),which returns the character thet was entered, and getKeyCode() ,which returns the key code.Their general forms are shown here:

Char getKeyChar()

Int getKeyCode()

If no valid character is available, then getKeyChar() returns CHAR_UNDEFINED. When a KEY_TYPED event occurs, getKeyCode() returns VK_UNDEFINED.

29.5 Processing Mouse Events

Method	Description	Called with values
mouseDown()	Called if the mouse button is down	mouseDown(Event e,int x, in y)
mouseDrag()	Called if the mouse while a Button is pressed	mouseDrag(Event e, int x, int y)
mouseEnter()	Called when the mouse enters the component	mouseEnter(Event e, int x, int y)
mouseExit()	Called when the mouse exits the component	mouseExit(Event e, int x, int y)
mouseMove()	Called if the mouse moves while no buttons are pressed	mouseMove(Event e, int x, int y)
mouseUp()	Called if the mouse button is up	mouseUp(Event e, int x, int y)

The Action Event Class

An **ActionEvent** is generated when a button is pressed, a list item is double-clicked, or a menu item is selected. The **ActionEvent** class defines **four** integer constants that can be used to identify any modifiers associated with an action event:

ALT_MASK,CTRL_MASK,META_MASK,and SHIFT_MASK.In addition,there is an integer constant,ACTION_PERFORMED,which can be used to identify action events.

ActionEvent has these two constructors:

ActionEvent(Object src,int type,String cmd)

ActionEvent(Object src,int type,String cmd,int modifiers)

Here, **src** is a reference to the object that generated this event. The type of the event is specified by Type, and its command string is cmd. The argument modifiers indicates which modifiers keys(ALT, CTRL, META, and/or SHIFT) were pressed when the event was generated.

You can obtain the command name for the invoking ActionEvent Object by using the getActionCommand() method,shown here:

String getActionCommand()

For example,when a button is pressed, an action event is generated that has a command name equal to the label on that button.

The getModifiers() method returns a value that indicates which modifier keys(ALT,CTRL,META,and/or SHIFT) were pressed when the event was generated.Its form is shown here:

Int getModifiers()

29.6 The Mouse Event Class

There are several types of mouse events. The **MouseEvent** class defines the following integer contants that can be used to identify them:

MOUSE_CLICKED	The user clicked the mouse
MOUSE_DRAGGED	The user dragged the mouse
MOUSE_ENTERED	The mouse entered a component
MOUSE_EXITED	The mouse exited from a component.
MOUSE_MOVED	The mouse moved.
MOUSE_PRESSED	The mouse was pressed.
MOUSE_RELEASED	The mouse was released.

Mouse Event is a subclass of InputEvent and has this constructor:

MouseEvent(Component src,int type, long, when, int modifiers, int x, int y, int clicks boolean triggersPopup)

Here **src** is a reference to the competent that generated the event. The type of the event is specified by type. The system time at which the mouse event occurred is passed in when. The modifiers argument indicates which modifiers were pressed when a mouse event occurred. The coordinated of the mouse are passed in x and y. The click count is passed in clicks. The triggersPopup flag indicated if this event causes a pop-up menu to appear on this platform.

The most commonly used methods in this class are getX ( ) and get Y ( ). These return the X and y corrdinates of the mouse when the event occurred. Their form are shown here:

```
Int get X ( )
 Int  get Y ( )
```

Alternatively, you can use the getPoint ( ) method to obtain the coordinated of the mouse . It is shown here:

```
Point getPoint  ( )
```

It returns a Point object that contains the X,Y coordinated in its integer members. X and y. The translatePoint( ) method changes the location of the event . Its form is shown here:

```
Void translatePoint int x,int y)
```

Here the arguments x and y are added to the coordinated of the event.

The **getClikCount( )** method obtains the number of mouse clicks for this event. Its signature is shown here:

```
Int getClickCount ( )
```

The is **PopupTrigger( )** methods tests if this event causes a pop-up menu to appear on this platform. Its form is shown here:

```
Bollean is PopupTrigger ( ).
```

The package java.awt.event defines several types of events that are generated by various user interface elements. The Most important event class are given in the below table.

Event Class	**Description**
ActionEvent	Generated when a button is pressed, a list item is Double-clicked, or a menu item is selected.
AdjustmentEvent	Generated when a scroll bar is manipulated.

ComponentEvent	Generated when a component is hidden, moved, resized, o or becomes visible.
ContainerEvent	Generated when a component is added to or removed from a container.
FocusEvent	Generated when a component gains or loses keyboard focus.
InputEvent	Abstract super class for all component input event classes
ItemEvent	Generated when a checkbox or list item is clicked;also occurs when a choice selection is made or a checkable menu item is selected or deselected.
KeyEvent	Generated when input is received from the keyboard.
MouseEvent	Generated when the mouse is dragged, moved, clicked, pressed, or released;also generated when the mouse enters or exits a component.
TextEvent	Generated when the value of a text area or text area or text field is changed.
WindowEvent	Generated when a window is acivated, closed, deacivated, deiconified, iconified,opened,or quit.

Table . Main Event Classes in java.awt.event

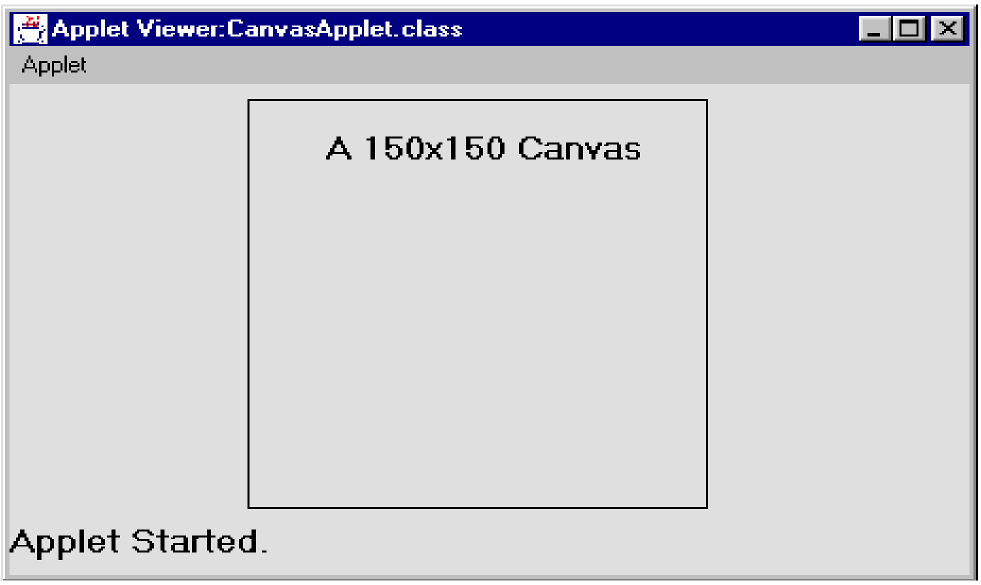
29.7 Canvas

A Canvas class provides the ability to construct generic GUI components. The class does not have any default graphical representation, or any event handlers of its own. It inherits these capabilities from its super class Component. The Canvas class is usually subclassed to construct customized GUI components consisting of drawing or images, and can handle user input events relating to mouse and keyboard actions. Its **paint( )** method is usually overridden to render graphics in the component

```
// Illustrating Canvas
import java.awt.*;
import java.applet.*;
public class CanvasApplet extends Applet
```

```
{
          public void init( )
          {
                    DrawingRegion region = new DrawingRegion ( );
                    add(region);
          }
}
class DrawingRegion extends Canvas
{
                    public DrawingRegion
                    {
                              setSize(150,150);
                    }
                    public void paint (Graphics g)
                    {
                    g.drawRect(0,0,149,149);       //draw border around region

                    g.drawstring("A 150 x 150 Canvas", 20,20);   //draw string
          }
          }
```

The above code draws a rectangular region on a canvas.



29.8 Short Summary

- The class java.awt.CheckboxGroup can be used to control the behavior of a group of checkboxes.

- The Choice class implements a pop-up menu of choices.
- A document window in a text processor usually has two scroll bars. A vertical scrollbar to scroll up and down. A horizontal scroll bar similarly scrolls the document let and right.
- There are three types of key events, which are identified by these integer constants: KEY_PRESSED, KEY_RELEASSED, and KEY_TYPED.
- The Text Area class implements multiple lines of optionally editable text.
- A canvas class provides the ability to construct generic GUI components.

29.9 Brain Storm

1. How can we add the button control in applet?
2. What is difference between choice and checkbox controls?
3. How can we create a window?
4. What is difference between key_pressed and key_Releashed events?
5. What are events we used for Mouse and keyboard devices?

Lecture - 30

Applet versus Application

Objectives

In this lecture you will learn the following

- About Features of Applet
- Difference between applet and application

Lecture - 30

30.1 Snap Shot

Although Java is a general-purpose programming language suitable for a large variety of tasks. An applet is a Java program that executes on a World Wide Web page.

30.2 Introduction and Features of Applet

Applet Basics

All applets are subclasses of Applet. Thus all applets must import java applet. Applets must also import java.awt. recall that AWT stands for the Abstract Windows Toolkit. Since all applets run in a windows, it is necessary to include support for that window. Applets are not executed by the console based java runtime interpreter. Rather they are executed by either a Web browser or an applet viewer. The figures shown in this chapter were created with the standard applet viewer, called applet viewer, provided by the JDk. But you can use any applet viewer or browser you like.

Execution of an applet does not begin at main ( ). Actually few applets even have main ( ) methods. Instead, execution of an applet us started and controlled with an entirely different mechanism, which will be explained shortly. Output to your applet's window is not performed by Syste.out.println ( ). Rather, it is handled with various AWT methods, such as drawstring( ) which outputs a string to a specified X ,Y location . input is also handled differently that in an application.

Once an applet has been compiled it is included in an HTML file using the APPLET tag. The applet will be executed by a java enabled web browser when it encounters the APPLET tag within the HTML file. To view and test an applet more conveniently, simply include a comment at the head of your java source code file that contains the APPLET tag. This way your code is documented with the necessary HTML statements needed by you applet, and you can test the compiled applet by starting the applet viewer with your java source code file specified as the target. Here is an example of such a comment:

```
  /*
 <applet code="MyApplet" width =200 height = 60>
 </applet>
*/
```

This comment contains an APPLET tag that will run an applet called MyApplet in a window that is 200 pixel wide and 60 pixels high. Since the inclusion of an APPLET command makes testing applets easier, all of the applets shown in this book will contain the appropriate APPLET tag embedded in a comment.

Applet Fundamentals

All of the preceding examples in this book have been java application. However, application constitute only one class of java programs. The other type of program is the applet. As mentioned in Chapter 1, applets are small applications that are accessed on an internet server, transported over the internet, automatically installed, and run as part of a Web document. after an applet arrives on the client, it has limited access to resources, so that it can produce an arbitrary multimedia user interface and run complex computations without introducing the risk of ciruses or branching data integrity.

Many of the issues connected with the creation and use of applets are found in part II when the applet package is examined. However, the fundamentals connected to the cretion of an applet are presented here, because applets not structured in the same way as the programs that the have been used thus far.

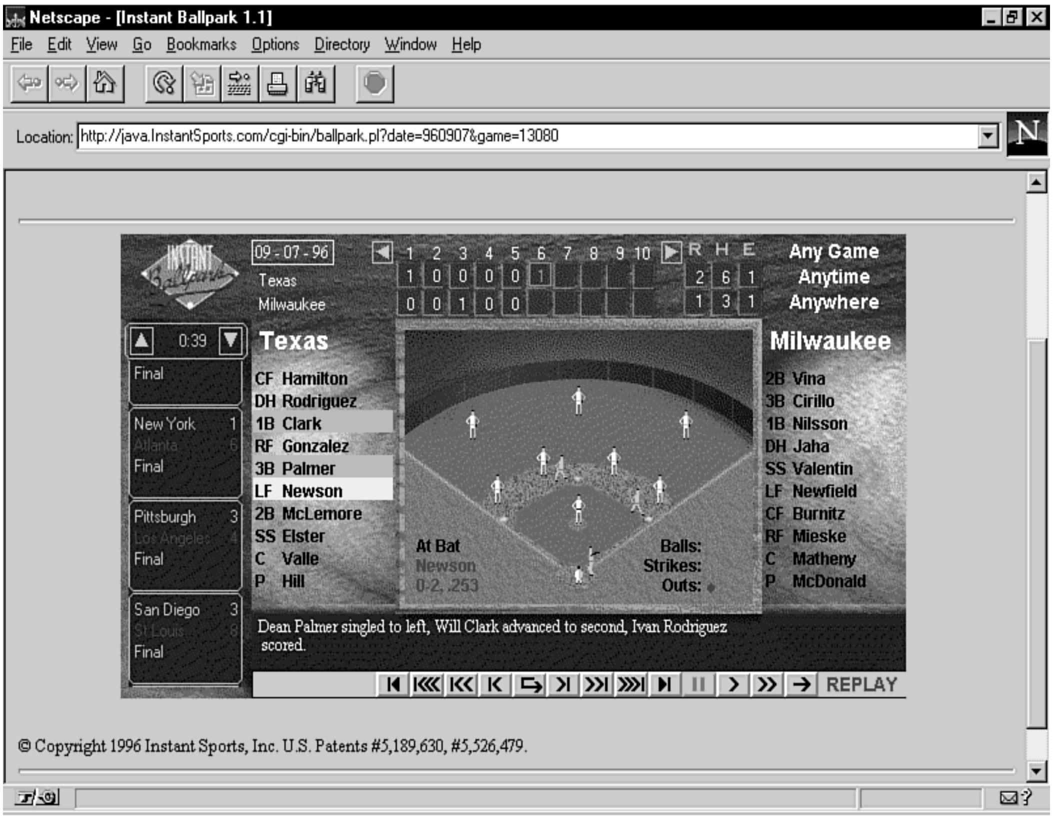
Using Applet

Today, applets are being used to accomplish far more than demonstrative goals. There are working examples of applets on Web sites throughout the Internet-a check of the AltaVista search engine finds more than 4,200 Web pages that have applets embedded on them.

The current uses of applets include the following:

- Tickertape-style news and sports headline updates
- Animated graphics
- Video games
- Student tests
- Image maps that respond to mouse movement
- Advanced text displays
- Database reports

Figure shows a noteworthy example of an applet: the Instant Ballpark program from Instant Sports.



Location: http://java.InstantSports.com/cgi-bin/ballpark.pl?date=960907&game=13080

	1	2	3	4	5	6	7	8	9	10	R	H	E
Texas	1	0	0	0	0	1					2	6	1
Milwaukee	0	0	1	0	0						1	3	1

Texas		Milwaukee	
CF	Hamilton	2B	Vina
DH	Rodriguez	3B	Cirillo
1B	Clark	1B	Nilsson
RF	Gonzalez	DH	Jaha
3B	Palmer	SS	Valentin
LF	Newson	LF	Newfield
2B	McLemore	CF	Burnitz
SS	Elster	RF	Mieske
C	Valle	C	Matheny
P	Hill	P	McDonald

Dean Palmer singled to left, Will Clark advanced to second, Ivan Rodriguez scored.

© Copyright 1996 Instant Sports, Inc. U.S. Patents #5,189,630, #5,526,479.

Instant Ballpark takes real-time data from live baseball games and updates its display to reflect what's happening in the game. Players run the bases, the ball goes to the place it was hit, and sound effects are used for strike calls, crowd noise, and other elements. The program, which was unique enough to qualify for a U.S. patent, is reminiscent of the old-time baseball tradition of presenting the play-by-play for road games by moving metal figures on the side of a building. In addition to the live coverage, Instant Ballpark can be used to review the play-by-play action of past games.

The applet shows one of the advantages of a Web program over a Web page. With HTML and some kind of gateway programming language such as Perl, a Web page can offer textual updates to a game in progress. However, Instant Ballpark offers a visual presentation of a live game in addition to text, and the applet can respond immediately to user input. Java can be used to provide information to Web users in a more compelling way, which is often the reason site providers are offering applets.

Viewing Applets

As you know, applets are displayed as a part of a Web page. A special HTML tag, <APPLET>, is used to attach a Java applet to an HTML page. Running an applet requires the use of a Web browser or other software that serves the function of a browser, such as the applet viewer program that ships with the Java Developers Kit from JavaSoft.

The browser acts as the operating system for applets-you cannot run an applet as a standalone program in the same way you can run an executable file.

At the time of this writing, there are three widely available Web browsers that can run Java applets:

- Netscape Navigator version 2.02 or higher
- Microsoft Internet Explorer 3.0
- JavaSoft HotJava 1.0 pre-beta 1

These programs load applets from a Web page and run them remotely on the Web user's computer. This arrangement raises security issues that must be handled by the Java language itself and by Java-enabled browsers.

30.3 Differences between Applets and Applications.

Traditionally, the word applet has come to mean any small application. In java, an applet is any java program that is launched from a web document; that is, from an HTML file. Java applications, on the other hand, are programs that run from a command line, independent of a web browser. The size or complexity of a java applet has no limit. In fact, java applets are in some ways more powerful than java applications. However, with the internet, where communication speed is limited and download times are long, most java applets are small by necessity.

The technical differences between applets and applications stem from the context in which they run. A java application runs in the simplest possible environment-its only input from the outside world is a list of command-line parameters. On the other hand, a java applet receives a lot of information from the web browser. It needs to know when it is initialized, when and where to draw itself in the browser window, and when it is activated or deactivated. As a consequence of these two very different execution environments, applets and applications have different minimum requirements.

The decision to write a program as an applet versus an application depends on the context of the program and its delivery mechanism. because java applets are always presented in the context of a web browser's graphical user interface(GUI), java applications are perferred over applets when graphical displays are unnecessary. for example, an HTTP server written in java needs no graphical display; it requries only file and network access.

The convenience of web protocols for applet distribution makes applets the preferred program type for Internet applications, although applications can easily be used to perform many of the same tasks. with java, writing Internet-based software, either as applets or applications, is extremely easy. non-networked systems and systems with small amounts of memory are much more likely to be written as java applications than as java applets.

Difference between Java applets and application

	Java Applet	**Java Application**
User graphics	Inherently graphical	Optional
Memory requirements	Java application requirements plus web browser requirements	Minimal java application requirements
Distribution	Linked via HTML and transported via HTTP	Loaded from the file system or by a custom class loading process
Environmental input	Browser client location and size; parameters embedded in the host HTML document	command-line parameters
Method expected by the virtual Machine	init- initialization method start-startup method stop pause/ deactive method destroy-termination method paint-drawing method	Main - startup method
Typical applications	public-access order-entry systems for the web, online multimedia persentations, web page animation	Network server, multimedia kiosks, developer tools, appliance and consumer electronics control and navigation.

You should consider one other major factor when deciding applet or application. If you are using features of newer java versions, you need to wait until browsers support the capabilities. With an application, you can provide the Java Runtime Environment. However, within an applet, you can only use the capabilities a browser offers. In an Internet environment, you can expect users to still be using older browser versions, which do not support to be more control over software versions, you can know what versions are available and develop accordingly. Also, you may want to consider using Sun's Java Plug in product which can automatically update the Java version of browsers when a new version becomes available.

30.4 Summary

Java security is important because it makes exciting new things possible with very little risk. Early security holes caused by implementation bugs are being closed, and technology is being fielded that permits the strict security policy to be carefully and selectively relaxed. Resources that can be used to destroy or steal data are protected, and researchers are examining ways to prevent applets from using other resources to cause annoyance or inconvenience.

Application developers can design their own security policies and supply parts of the third layer of the Java security model to implement those policies in their applications.

The Java security architecture is sound. Early weaknesses and bugs are not a surprise, and the process that has exposed those flaws has also helped remove them.

30.5 Brain Storm

1. What is applet?
2. Explain the difference between Java applets and applications.
3. Explain the features of applet.

Lecture - 31

Applet Life Cycle

Objectives

In this lecture you will learn the following

- Applet Life cycle
- Features of Applet
- Package java.applet
- Applet Capabilities
- Security and Restrictions
- Applet implementation

Lecture - 31

31.1 Snap Shot

In this session we deal about applet life cycle, Package java.applet, Features of applet, capability, Restriction and Implementation of applet.

31.2 Applet Life Cycle

All but the most trivial applets override a set of methods that provides the basic mechanism by which the browser or applet viewer interfaces to the applet and controls its execution. Four of these method – init() , start(), stop(), and destroy() – are defined by Applet. Another , paint(), is defined by the AWT Component Class . Default implementations for all of these methods are provided. Applets do not need to override those methods they do not use. However, only very simple applets will not need to define all of them. These five methods can be assembled into the skeleton shown here:

```
// An Applet Skeleton.
import java.awt.*;
import java.applet.*;
/*
<applet code = "AppletSkel " width = 300 height =100>
</applet>
*/
public class AppletSkel extends Applet{
//Called first.
Public void init(){
// initialization
}
/* Called second, after init() . Also called whenever the applet is restarted .*/
public void start(){
//start or resume execution
}
//Called when the applet is stopped.
Public void stop(){
//suspends execution
}
/* Called when applet is terminated. This is the last method executed .*/
public void destroy () {
// perform shutdown activites
}
//Called when an applet's window must be restored.
Public void paint (Graphics g){
//redisplay contents of window
          }
          }
```

Although this skeleton does not do anything , it can be compiled and run. When run, it generates the following window when viewed with an applet viewer.

Applet Initialization and Termination

It is important to understand the order in which the various methods shown in the skeleton are called . When an applet begins, the AWT calls the following methods , in this sequence:

1. init()
2. start()
3. paint()

When an applet is terminated , the following sequence of method calls takes place.

1. stop()
2. destroy()

Let's look more closely at these methods.

init()

The init() method is the first method to be called . This is where you should initialize variables. This method is called only once during the run time of your applet.

start()

The start() method is called after init() . It is also called to restart an applet after it has been stopped. Whereas init() is called once- the first time an applet is loaded – start() is called each time an applet's HTML document is displayed onscreen. So, if a user leaves a web page and comes back, the applet resumes execution at start().'

paint()

The paint() method is called each time your applet's output must be redrawn. This situation can occur for several reasons. For example , the window in which the applet is running may be overwritten by another window and then uncovered. Or the applet window may be minimized and then restored. Paint() is also called when the applet begins execution. Whatever the cause, whenever the applet must redraw its output, paint() is called. The paint() method has one parameter of type Graphics. This parameter will contain the graphics context, which describes the graphics environment in which the applet is running. This context is used whenever output to the applet is required.

stop()

The stop() method is called when a web browser leaves the HTML document containing the applet – when it goes to another page, for example. When stop() is called, the applet is probably Running . You should use stop() to suspend threads that don't need to run when the applet is not visible. You can restart them when start() is called if the user returns to the page.

destroy()

The destroy() method is called when the environment determines that your applet needs to be removed completely from memory. At this point, you should free up any resources the applet may be using. The stop() method is always called before destroy().

Overriding update()

In some situations, your applet may need to override another method defined by the AWT , called update(). This method is called when your applet has requested that a portion of its window be redrawn. The default version of update() first fills an applet with the default background color and then calls paint(). If you fill the background using a different color in paint(), the user will experience a flash of the default background each time update() is called- that is , whenever the window is repainted. One way to avoid this problem is to override the update() method so that it performs all necessary display activities. Then have paint() simply call update(). Thus, for some applications, the applet skeleton will override paint() and update(), as shown here.

```
public void update(Graphics g){
//redisplay your window, here.
}
public void paint(Graphics g){
update(g);
```

}

For the examples in this book, we will override update() only when needed.

31.3 Features in Applet

Java 1.1 has been out since early 1997 . If you are moving to Java 1.2 from java 1.0 several features will appear new and different. Here is summary of the new features of java 1.1

Internationalization:

New classes and methods have been added to make writing programs for international users even easier. This includes support for non –English characters and text sorting, as well as a variety of time and date standards.

Security:

Several enhancements, including digital signing, have been incorporated into a new and improved security manager. Also security for native method calls has been enhanced.

Performance enhancements:

A new AWT event model and complete rewrite of the native code for AWT has boosted GUI performance dramatically. Also much of the compiler and interpreter coded has been rewritten.

Network and I/O enhancements:

New classes provide extra network a functionality, as well as greater customization, buttered input and output.

Object reflection.:

A special API for getting privileged information about a specific object has been added. This is especially useful for debuggers and other VM- enhancement programs.

The JDBC and javaBenas capabilities previously mentioned are also new to java 1.1

Java 1.2 adds even more enhancements some improving capabilities added inn Java with other completely new. The following summarizer the features new to Java.

Enhancements to security, javaBeans, reflection, and performance:

Numerous java a packages received enhancements in java 1.2 The security architecture incorporates policy-based access control to enhance permission management. Java Beans adds drag-and-drop support, while reflection includes the ability to bypass security, specifically when using object reflection. Also, the performance of various pieces of the java libraries was improved, for instance, faster memory allocation, reduced memory usage for loaded classes, and just in time compilers.

Java Foundation Classes :

The java Foundation Classes (JFC) encompass a broad range of enhancements. There is now support for assertive technologies with Accessibility API, a java 2-D API enhanced graphics and imaging, and Swing for a new set of GUI a component, in +-addition to the AWT components.

Collections:

The Collection API makes working with groups of objects much easier. Prior to java 1.2 you basically used the earlier Vector and hashtable classes, as well as the enumeration interface. Here, you can work with things like balanced trees, circular linked lists, and simplified array sorting.

In addition to these changes, a whole host of changes to methods, classes and packages have provided extra functionality. These changes are documented throughout this book.

31. 4 Package java.applet

A big reason for java's runaway success is that it's a highly efficient and easy-to-learn language for distributed software components. java applets are nothing more or less than distributed software components. even so, the standard class framework contains little that explicitly deals with those instrumental applets.

The java.applet package looks very barren compared to the other packages. its sole contents are one class and three interfaces. Class java.applet.Applet is the main repositry for methods supporting applet functionality.

The methods it makes available can be grouped into the following categories.

- Applet initialization, restarting, and freezing.
- Embedded HTML applet parameter support
- High-level image loading
- High-level audio loading and playing for applets and applications
- Origins querying (getDocumentBase() and getCodeBase())
- Simple status displaying(showStatus(String))

31. 5 Applet Capability

Much of the power of the World Wide Web stems from its platform dependence that is it presents information in a way that can be viewed on almost every type of machine and operating system. It doesn't matter whether you use a PC, Macintosh, or Unix workstation the Web is architecture neutral which is why so many people have access to it.

Unfortunately, being so widely accepted also has its drawbacks. It is difficult to extend the Web protocol without leaving many Web users behind.. For instance Web content developers are constantly trying to extend the capability of the Web by integrating new types of media, like 3 d worlds and animation, but these developers then face the prospect of excluding people without those viewing capabilities, which limits their audience.

The existing Web standards permit seamless integration of graphics with text. Other forms of media ,a such as sound, video, and animation are accessible via the Web but they ar enot smoothly connected with normal Web content. For example it is easy to create a link to a sound file in a HTML document the Web browser will either play the sound or download it to a file when a the user clicks on the link. However, there is no browser independent way to create background music for a document or give audio feedback when a button is pressed. This is just one of the many creative limitations that have frustrated Web developers over the past few years.

Until now, the solution to this extensibility problem has been to create a proprietary protocol, and then try to seal the solution to as many users on as many platforms as possible. This is a hard sell and has had limited success. As a result, Web pages tend to cater to the lowest common denominator therefore, the content has not reached its full potential in many instances.

A good example of this is Adobe's Portable Document Format . This is a cross platform solution for creating robust documents and distributing them on the internet. PDF provides support for documents far richer than simple HTML allowing groups like the internal Revenue Service to ship tax forms across the Web. Adobe provides the viewer for free but tries to make money on the tools that create PDF document. PDF's main limited is that you need to download a special programs from Adobe to view the files.

31.6 Security and Restriction

Java applets are programs that run on a Web user's machine. Anything that can execute code is a potential security risk because of the damaging things that can occur. Viruses can damage a computer's file system and reproduce onto other disks. Even Microsoft Word has been a security risk because of Word Basic-an executable programming language that can be used in conjunction with Word documents.

Security is one of the primary concerns of Java's developers, and they have implemented safeguards at several levels. Some of these safeguards affect the language as a whole: The removal of pointers, the verification of bytecodes, and other language issues have been discussed elsewhere in this book.

Some of Java's functionality is not possible when programming applets because of security concerns. The following safeguards are in place:

- Applets cannot read or write files on the Web user's disk. If information must be saved to disk during an applet's execution, the storage of information must be done on the disk from which the Web page is served.
- Applets cannot make a network connection to a computer other than the one from which the Web page is served.
- Pop-up windows opened by applets are identified clearly as Java windows. A Java cup icon and text such as Untrusted Applet Window appear in the window. These elements are added to prevent a window opened by Java from pretending to be something else, such as a Windows dialog box requesting a user's name and password.
- Applets cannot use dynamic or shared libraries from any other programming language. Java can make use of programs written in languages such as Visual C++ by using a native statement from within Java. However, applets cannot make use of this feature because there's no way to adequately verify the security of the non-Java code being executed.
- Applets cannot run any programs on the Web user's system.

As you can see, Java applets are more limited in functionality than standalone Java applications. The loss is a tradeoff for the security that must be in place for the language to run remotely on users' computers.

Many users and applet developers understand the need for security, but wish it weren't so strict and inflexible. They say that users should be able to disable or weaken Java's security if they want to. To a large degree, they're right, and you can expect Java applications to have more flexibility in the future.

Java security isn't all or nothing. The application can grant or deny access to applets based on a wide variety of criteria: the name of the applet, where it came from, the type of resource it's trying to access, even the particular resource. An application can choose to let applets read some files, but not others, for example.

Flexible, selective schemes involve a lot of extra complexity, and with complexity comes the potential for mistakes. In addition, there are subtle, difficult questions surrounding flexible security schemes. For example, employees may have very different ideas about acceptable security than their employer does-how much control should be given to the users and how much to the site security administrator? Faced with numerous tough questions like that, the people behind Java decided to be careful at first. They are starting with an extremely conservative security model, which will become less rigid as time goes on. This is probably a good strategy because a big security scare early in Java's lifetime would have really dampened enthusiasm for the language.

The other problem is that there isn't yet a good criterion for deciding which applets should be trusted and which should not. The best solution is probably to trust applets based on who wrote them, but that's difficult to verify.

What Are the Dangers?

To really understand the Java security model-why it's important, how it works, and how to work with it-you should have a good idea about the kinds of security attacks that are possible and which system resources can be used to mount such attacks. Java takes care to protect these resources from untrusted code. If you are using a Java application that allows you to configure applet security, or if you are writing a Java application that loads classes from the Net, it helps to understand just what doors you might be opening when you give an applet access to a particular resource.

The Kinds of Attacks

There are several different kinds of security attacks that can be mounted on a computer system. Some of them are surprising to people who are new to computer security issues, but they are very real and can be devastating under the right circumstances. Table 1 lists some common types of attacks.

Type of Attack	Description
Theft of information	Nearly every computer contains some information that the owner or primary user of the machine would like to keep private.
Destruction of information	In addition to data that is private, most of the data on typical computers has some value, and losing it would be costly.
Theft of resources	Computers contain more than just data. They have valuable, finite resources that cost money: disk space and a CPU are the best examples. A Java applet on a Web page could quietly begin doing some extensive computation in the background, periodically sending intermediate results back to a central server, thus stealing some of your CPU cycles to perform part of someone else's large project. This would slow down your machine, wasting another valuable resource: your time.
Denial of service	Similar to theft of resources, denial-of-service attacks involve using as much as possible of a finite resource, not because the attacker really needs the resource, but simply to prevent someone else from being able to use it. Some computers (like mail servers) are extremely important to the day-to-day operations of businesses, and attackers can cause a lot of damage simply by keeping those machines so busy with worthless tasks that they can't do their real jobs.
Masquerade	By pretending to be from another source, a malicious program can persuade a user to reveal valuable information voluntarily.
Deception	If a malicious program were successful in interposing itself between the application and some important data source, the attacker could alter data-or substitute completely different data-before giving it to the application or the user. The user would take the data and act on it, assuming it to be valid.

Table 1. Some common types of attacks against computers.

In addition to these common attacks, Java applets can try another kind of attack. Because applets are fetched to your machine and run locally, they can try to assume your identity and do things while pretending to be you. For example, machines behind corporate firewalls often trust each other more than they trust machines on the wider Internet, so once an applet has started running on your machine behind the firewall, it may try to access other machines, exploiting that trust. Another example is mail forging: once on your machine, an applet may attempt to send threatening or offensive mail which appears to be from you. Of course, Internet mail can be forged from other machines besides your own, but doing it from your own machine makes it a little more convincing.

How Does Java Security Work?

Now that you understand why security features are important and what kinds of threats exist, it's time to learn how Java's security features work and how they protect against those threats.

The Java security model is composed of three layers, each dependent on those beneath it. The following sections cover each of the layers, describing how the security systems works.

The Three Layers of Security

The first line of defense against untrusted programs in a Java application is a part of the basic design of the language: Java is a safe language. When programming language theorists use the word safety, they aren't talking about protection against malicious programs. Rather, they mean protection against incorrect programs. Java achieves this in several ways:

- Array references are checked at runtime to ensure that they are within the bounds of the array. This check prevents incorrect programs from running off the end of an array into storage that doesn't belong to the program or that contains values of the wrong type.
- Casts are carefully controlled so that they can't be used to violate the language's rules, and implicit type conversions are kept to a minimum.
- Memory management is automatic. This arrangement prevents "memory leaks" (when unused storage is never reclaimed) and "dangling pointers" (when valid storage is freed prematurely).
- The language does not allow programmers to manipulate pointers directly (although they are used extensively behind the scenes). This feature prevents many invalid uses of pointers, some of which could be used to circumvent the preceding restrictions.

All these qualities make Java a "safe" language. Put another way, they ensure that code written in Java actually does what it appears to do, or fails. The surprising things that can happen in C (such as continuing to read data past the end of an array as though it were valid) cannot happen. In a safe language, the behavior of a particular program with a particular input should be entirely predictable-no surprises.

The second layer of Java security involves careful verification of Java class files-including the virtual machine bytecodes that represent the compiled versions of methods-as they are loaded into the virtual machine. This verification ensures that a garbled class file won't cause an error within the Java interpreter itself, but it also ensures that the basic language safety is not violated. The rules about proper language behavior that were written into the language specification are good, but it's also important to make sure that those rules aren't broken. Checking everything in the compiler isn't good enough, because it's possible for someone to write a completely new compiler that omits those checks. For that reason, the Java library carefully checks and verifies the bytecodes of every class that is loaded into the virtual machine to make sure that those bytecodes obey the rules. Some of the rules, such as bounds checking on references to array elements, are actually implemented in the virtual machine, so no real checks are necessary. Other rules, however, must be checked carefully. One particularly important rule that is verified rigorously is that objects must be true to their type-an object that is created as a particular type must never be able to masquerade as an object of some incompatible type. Otherwise, there would be a serious loophole through which explicit security checks could be bypassed.

This verification process doesn't mean that Java code can't be compiled to native machine code. As long as the validation is performed on the bytecodes first, a native compiled version of a class is still secure. "Just-in-time" (JIT) compilers run within the Java virtual machine, compiling bytecodes to native code as classes are loaded, just after the bytecode verification stage. This compilation step doesn't usually take much time, and the resulting code runs much faster.

The third and final layer of the Java security model is the implementation of the Java class library. Classes in the library provide Java applications with their only means of access to sensitive system resources, such as files and network connections. Those classes are written so that they always perform security checks before granting access.

This third layer is the portion of the security system that an application can control-not by changing the library implementation, but by supplying the objects that actually make the decisions about whether to grant each request for access. Those objects-the security manager and the class loaders-are the core of an application's security policy, and you'll read more about them (including how to implement them) a little later in this chapter.

Protected System Resources

The first two layers of the Java security model are primarily concerned with protecting the security model itself. It's the third layer, the library implementation, in which explicit measures are taken to protect against the kinds of attacks listed in Table 2. Those resources fall into six categories, as listed in Table 2.

Resource	Description
Local file access	The capability to read or write files and directories. These capabilities can be used to steal or destroy information, as well as to deny service by destroying important system files or writing a huge file that fills the remaining space on your disk. Applets can also use local file access to deceive you by writing an official-looking file somewhere that you will find later and believe to be trustworthy.
System access	The capability to execute programs on the local machine, plus access to system properties. These capabilities can be used for theft or destruction of information or denial of service in much the same way that direct file access can: by executing commands that manipulate your files. Additionally, system properties may contain information that you view as private or that can help an attacker break into your system using other means.
Network access	The capability to create network connections, both actively (by connecting to some machine) and passively (by listening and accepting incoming connections). Applets that actively create connections may be trying to usurp the user's identity, exploiting the trust that other machines place in him or her. Applets that try to listen for incoming connections may be taking over the job of a system service (such as a Web server).
Thread manipulation	The capability to start, stop, suspend, resume, or destroy threads and thread groups, as well as other sorts of thread manipulation such as adjusting priorities, setting names, and changing the daemon status. Without restrictions on such capabilities, applets can destroy work by shutting down or disabling other components of the applications within which they run, or do so to other applets. Rogue applets can also mount denial of service attacks by raising their own priority while lowering the priorities of other threads (including the system threads that may be able to control the errant applets).
Factory object creation	The capability to create factory objects that find and load extension classes from the network or other sources. An untrustworthy factory object can garble user data,

	transparently substitute incoming data from a completely different source, or even steal outgoing data-without the user of the application realizing what's happening. See "Further Reading," later in this chapter, for pointers to more information about factory objects.
Window creation	The capability to create new top-level windows. New top-level windows may appear to be under the control of a local, trusted application rather than an applet, and they can prompt unwary users for important information such as passwords. The Java security system permits applications to forbid applets from creating new windows, and it also permits tagging applet-owned windows with a special warning for users.

Table 2. Resources checked by Java security.

The third layer of the security model isn't just concerned with protecting system resources; it also provides protection for some Java runtime resources, to protect the integrity of the security model itself.

Example: Reading a File

Let's look at an example to see how the security model works in practice. This example concentrates on what happens in the third layer, for two reasons: The lower two layers sometimes deal with some rather esoteric issues of type theory, and they are not within the programmer's control. The top layer, on the other hand, is relatively straightforward and can be controlled by Java application programmers.

Suppose that the Snark applet has been loaded onto your system and wants to read one of your files-say, diary.doc. To open the file for reading, Snark must use one of the core Java classes-in particular, FileInputStream or RandomAccessFile in the java.io package. Because those core classes are a part of the security model, before they allow reading from that particular file, they ask the system security manager whether it's okay. Those two classes make the request in their constructors; FileInputStream uses code like this:

```
// Gain access to the system security manager.
SecurityManager security = System.getSecurityManager();
if (security != null) {
  // See if reading is allowed.  If not, the security manager will
  // throw a SecurityException.  The variable "name" is a String
  // containing the file name.
  security.checkRead(name);
}
// If there is no security manager, anything goes!
```

The security manager is found using one of the static methods in the System class. If there is no security manager, everything is allowed; if there is a security manager, it is queried to see whether this access is permitted. If everything is fine, the SecurityManager.checkRead() method returns; otherwise, it throws a SecurityException. Because this code appears in a constructor, and because the exception isn't caught, the constructor never completes, and the FileInputStream object can't be created.

The SecurityManager class, an abstract class from which all application security managers are derived, contains several native methods that can be used to inspect the current state of the Java virtual machine. In particular, the execution stack-the methods in the process of executing when the security manager is queried-can be examined in detail. The security manager can thus tell which classes are involved in the current request, and it can decide whether all those classes can be trusted with the resource being requested.

In the Snark example, the security manager examines the execution stack and sees several classes, including Snark. That means something to us, but it probably doesn't mean a lot to the security manager. In particular, the security manager has probably never heard of a class called Snark, and presumably it doesn't even know that Snark is an applet. Yet that's the really important piece of information: if one of the classes currently on the execution stack is part of an applet or some other untrusted, dynamically loaded program, then granting the request could be dangerous.

At this point, the security manager gets some help. For each class on the execution stack, it can determine which class loader is responsible for that class. Class loaders are special objects that load Java bytecode files into the virtual machine. One of their responsibilities is to keep track of where each class came from and other information that can be relevant to the application security policy. When the security manager consults Snark's class loader, the security manager learns (among other things) that Snark was loaded from the network. At last, the security manager knows enough to decide that Snark's request should be rejected.

An Applet's View of Java Security

Applets and other untrusted (or partially trusted) classes, such as "servlets" in Java-based Web servers, or protocol handlers and content type handlers in HotJava, run within the confines of the application security policy. Such "unprivileged" classes are the kind that most Java programmers will be writing, so it's important to understand what the Java security facilities look like from the point of view of ordinary code.

Security violations are signaled when the security manager throws a SecurityException. It is certainly possible to catch that SecurityException and ignore it, or try a different strategy, so an attempt to access a secured resource doesn't have to mean the end of your applet. By trying different things and catching the exception, applets can build a picture of what they are and are not allowed to do. It's even possible to call the security manager's access checking methods directly, so that you can find out whether a certain resource is accessible before actually attempting to access it.

31.7 Implementing Applet

Java applet source code is written in the same way as java application source code with a text editor. The difference is that java applets do not have a main method. Instead, they have several other methods that are called by the VM when requested by the browser. Here is the source code for the simple FilledBox applet:

```
import java.awt.*;
import java.applet.Applet;
/* filled box displays a filled, colored box in the browser window */
public class FilledBox extends Applet {

color b;

public void init()
{
String s;
s= getParameter(“color”);
b=Color.gray;
if (s != null)
{
if(s.equals(“red”)) b=Color.red;
if(s.equals(“white”)) b=Color.white;
if(s.equals(“blue”))b=Color.blue;
}
}
public void paint(Graphics g) {
g.setColor(b);
g.fillRect(0,0,size().width, size().height);
}
}
```

It's a little more complicated than the java application example, but that is because it does more. you will recall that a main method is required by all java applications; it is conspicuously absent in this applet. In fact, java applets do not have any required methods at all. However, there are five methods that the VM may call when requested by the web browser

public void init() initializes the applet. Called only once.

public void start() Called when the browser is ready to start executing the initialized applet. Can be called multiple times if user keeps leaving and returning to the web page. also called when browser deiconified.

public void stop() Called when the browser wishes to stop executing the applet. Called whenever the user leaves the web page. also called when browser iconified.

public void destroy() Called when the browser clears the applet out of memory.

public void paint(Graphics g) Called whenever the browser needs to redraw the applet

If the applet does not implement any of these methods, the applet will have no functionality for the specific method not implemented. In the example, init and paint are implemented. The init function obtains the desired box color from a parameter in the host document. The paint method draws the filled box in the browser window.

Save this java applet source as **FilledBox. java**

using Javac

The java compiler works the same on applets as it does on java applications

javac FilledBox.java

Here are a few tips that may help you get start. First, applet classes must always be declared public or they will not get compiled. Also, remember that java is case sensitive FilledBox java is not the same as filledBox.Java and will not be compiled.

If the java code is acceptable to the compiler, they only message you will see is about a deprecated API:

```
note: FilledBox. java uses a deprecated API. recompile with
"-deprecation" for details.
1 warning.
```

for now, ignore the warning. As long as there were no error messages, the file FilledBox. class will be created. If there were error messages, you need to go back and fix your code. there are many different types of error messages that the compiler may generate when given a source file. the simplest to fix are syntax errors, such as a missing semicolon or closing brace. other messages will highlight incorrect use of variable types, invalid expressions, or violation access restrictions. Getting your source code to compile is only the first part of the debugging process; error free compilation does not guarantee that your program will do what you want. But don't worry about debugging just yet this example is simple enough that is should run without any problems.

Before you can run your applet, your must create an HTML document to host it.

Creating and HTML File

Now that you know a little about HTML it is easy to create a simple HTML file to host your applet

```
<HTML>
  < HEAD>
<TITLE >Sample HTML Document With Filled Box </TITLE>
</HEAD>
<BODY>
<H1> FilledBox  Demo </H1>
<P>
<APPLET CODE ="FilledBox.class" WIDTH =50 HEIGHT =50>
<PARAM NAME =color VALUE=  "blue">
</APPLET>
</BODY>
</HTML>
```

you can create this file by simply typing it into a text editor. Save the file as Filled Box.html. HTML files can be named anything you like, although it is common practice to name them after the applets they host.

Using appletviewer

The appletviewer utility is used to display the applet as it would be seen by the browser without displaying any of the HTML document itself. In the case of FilledBox.html, appletviewer will display a filled box in its own window:

```
appletviewer FilledBox.html
```

For comparison you can open the file FilledBox.html using a Java-enabled Web browser. Both the applet and the text are displayed.

If there is more than one applet in a page, appletviewer will open a separate window for each applet a Web browser will show them in their respective locations within the same Web page. One rather nice feature of appletviewer is that it can load classes from across the network, not just from files. Just give appletviewer the URL of the HTML document containing one or more applets, and it will load the applets as if they were on your local disk. Note, however, that the securityManager for the appletviewer may expose your system to greater risks from network loaded applets than would a Web browser like Netscape Navigator.

Appletviewer makes it possible to distribute and run java applets without the aid of a Web browser, so the choice between writing applets versus applications becomes less critical. Most applets are easy to convert into applications and vice versa. The key to this convertibility is to avoid placing a lot of code directly in the main, init, start, stop and destroy methods, and use called to generic methods instead.

31. 8 Short Summary

Java's security model is possibly the least understood aspect of the Java system. Because it's unusual for a language environment to have security facilities, some people have been bothered by the danger; at the same time, because the security restrictions prevent some useful things as well as harmful things, some people have wondered whether security is really necessary.

Java security is important because it makes exciting new things possible with very little risk. Early security holes caused by implementation bugs are being closed, and technology is being fielded that permits the strict security policy to be carefully and selectively relaxed. Resources that can be used to destroy or steal data are protected, and researchers are examining ways to prevent applets from using other resources to cause annoyance or inconvenience.

Application developers can design their own security policies and supply parts of the third layer of the Java security model to implement those policies in their applications.

The Java security architecture is sound. Early weaknesses and bugs are not a surprise, and the process that has exposed those flaws has also helped remove them.

31.9 Brain Storm

1. What do you mean by a secured data?
2. Why Java is called highly secured Language?
3. What are the layers of security?
4. List some of the resources that are checked by the Java Security.
5. Name some common attacks to the system.