Java Programming Notes

Contributor: Abhishek Sharma

[Founder at TutorialsDuniya.com]

Computer Science Notes

Unit-1

1. Introduction to Object Oriented Programming

The Object Oriented Programming (OOP) is one of the most interesting innovations in the Software Development. It addresses problems commonly known as "**Software Crisis**". Many software failed in the past. The term 'Software Crisis' is described in terms of failure in the software

- Exceeding Software Budget
- Software is not meeting the client's requirements.
- Bugs in the software.

OOP is a programming paradigm with deals with the concepts of the objects to build programs and software applications. IT is molded around the real world objects. Every object has well-defined ***identity, attributes, and behavior***. The features of the object oriented programming are similar to the real world features like ***Inheritance, abstraction, encapsulation, and polymorphism***.

2. Need of OOP

Whenever a new programming language is designed some trade-offs are made, such as

- ease-of-use verses power
- safety verses efficiency
- Rigidity versus extensibility

Prior to the c language, there were many languages:

- FORTRAN, which is efficient for scientific applications, but is not good for ***system code.***
- BASIC, which is easy to learn, but is not powerful, it is lack of ***structure.***
- Assembly Language, which is highly efficient, but is not ***easy to learn***.

These languages are designed to work with GOTO statement. As a result programs written using these languages tend to produce "**spaghetti code**", which is impossible to understand.

Dennis Ritchie, during his implementation of UNIX operating system, he developed C language, which similar to an older language called **BCPL**, which is developed by Martin Richards. The BCPL was influenced by a language called **B**, invented by Ken Thomson. C was formally standardized in December 1989, when the American National Standards Institute (ANSI) standard for C was adopted.

During the late 1970s and early 1980s, C became the dominant computer programming language, and it is still widely used today. The C language is ***structured***, ***efficient*** and ***high level language***. Since C is a successful and useful language, you might ask why ***a need for something else existed***. The answer is ***complexity***. As the program complexity is increasing it demanded the better way to manage the complexity.

When computers were first programming was done by manually **toggling** in the binary machine instructions by use of the front panel. As long as programs were just a few hundred instructions long, this approach worked. As programs grew, **assembly language** was invented so that a programmer could deal with larger, increasingly complex programs by using symbolic representations of the machine instructions. As programs continued to grow, **high-level languages** were introduced that gave the programmer more tools with which to handle complexity.

The 1960s gave birth to ***structured programming***. This is the method of programming championed by languages such as C. The use of structured languages enabled programmers to write, for the first time, moderately complex programs fairly easily. However, even with structured programming methods, once a project reaches a certain size, its complexity exceeds what a programmer can manage. To solve this problem, a new way to program was invented, called ***object-oriented programming (OOP).*** OOP is a programming methodology that helps organize complex programs through the use of **inheritance**, **encapsulation**, and **polymorphism**.

3. The OOP Principles

OOP languages follow certain principles such as, class, object, abstraction, encapsulation, inheritance and polymorphism.

3.1 Classes

A class is defined as the blueprint for an object. It serves as a plan or template. An object is not created just by defining the class, but it has to be created explicitly. It is also defined as new data type **contains data and methods.**

3.2 Objects

Objects are defined as the instances of a class. For example chairs and tables are all instances of Furniture. The objects have unique Identity, State and Behavior.

3.3 Abstraction

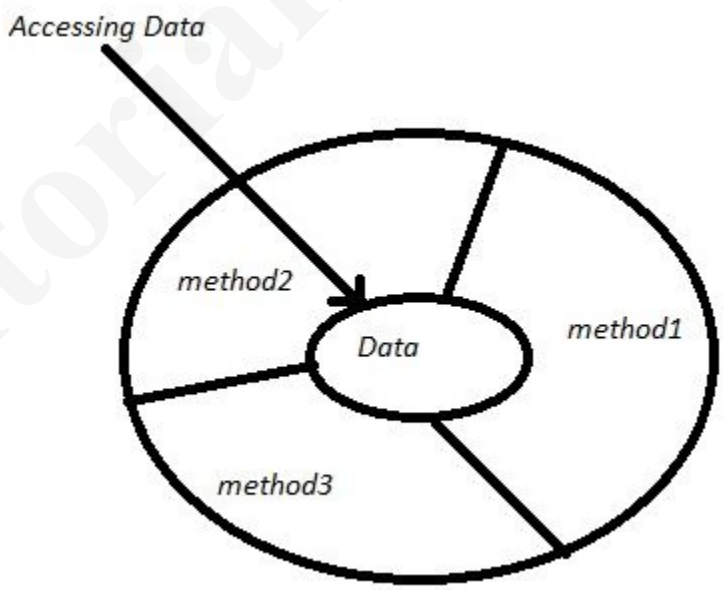
We can group the following items as Animals, Furniture and Electronic Devices.

Elephant	Television
CD Player	Chair
Table	Tiger

Here just by focusing on the generic characteristics of the items we can group the items into their classes rather than specific characteristics. This is called ***"abstraction***".

3.4 Encapsulation

- ***Encapsulation*** is the mechanism that binds ***methods*** and ***data*** together into a single unit.
- This hides the data from the outside world and keeps both safe from outside interference and misuse.
- It puts some restriction on outside code from directly accessing data.
- Encapsulation is also known as "***Data Hiding***".
- The Data can be accessed by the methods of the same class. Access to the code and data inside the wrapper is tightly controlled through a well-defined interface.
- In Java, the basis of encapsulation is the **class**.
- A *class* defines the structure and behavior (Data and Method) that will be shared by a set of **objects**. Each object of a given class contains the structure and behavior defined by the class.
- The objects are sometimes referred to as *instances of a class.* Thus, a class is a **logical** construct; an object has **physical** reality.
- When you create a class, you will specify the code and data that constitute that class. Collectively, these elements are called ***members*** of the class.
- Specifically, the data defined by the class are referred to as ***member variables*** **or** ***instance variables***.
- The code that operates on that data is referred to as ***member methods*** **or just** ***methods***. The members can be **public or private**. When a member is made public any code outside the class can access them. If the members are declared as private, then only the members of that class can access its members.



3.5 Inheritance

- *Inheritance* is the process by which one class acquires the properties or characteristics from another class.
- Here we have two types of classes: ***base*** class and ***derived*** class.
- The class from which the properties or characteristics are acquired is called "***Base Class***". The class that acquires the properties is called "***Derived Class***".
- The base class is also called super class or parent class. The derived class is also called sub class or child class.
- The main use of Inheritance is code reusability.
- The keyword "**extends**" is used for inheriting the properties.

3.6 Polymorphism

- *Polymorphism simply means many forms* (from Greek, meaning "many forms").
- It can be defined as same thing being used in different forms.
- It has two forms: ***compile-time*** polymorphism and ***run-time*** polymorphism.
- We know that binding is the process of linking function call to the function definition. If the linking is done at compile-time, then it is called compile-time binding. If it is done at the run time it is called run-time binding.
- The compile-time binding is also called as "static binding". The run-time binding is also called as "dynamic binding".
- The compile-time binding is implemented using method ***overloading***. The run-time binding is implemented using method ***overriding***.

4. Procedural language Vs OOP

All computer programs consist of two elements: methods and data. Furthermore, a program can be conceptually organized around its methods or around its data. That is, some programs are written around "***what is happening***" and others are written around "***who is being affected.***" These are the two paradigms that govern how a program is constructed. The first way is called the ***process-oriented model or procedural language***. This approach characterizes a program as a series of linear steps (that is, code). The process-oriented model can be thought of as ***code acting on data.*** Procedural languages such as C employ this model to considerable success.

To manage increasing complexity, the second approach, called ***object-oriented programming,*** was designed. Object-oriented programming organizes a program around its data (that is, objects) and a set of well-defined interfaces to that data. An object-oriented program can be characterized as ***data controlling access to code***.

Procedural language	Object Oriented Programming language
Separates data from functions that operate on them	Encapsulates the data and methods in a class
Not suitable for defining abstract types	Not suitable for defining abstract types
Debugging is difficult	Debugging is easier
Difficult to implement the change	Easier manage and implement the change

Not suitable for large programs and applications	suitable for large programs and applications
Analysis and design is not so easy	Analysis and design is easy
Faster	Slower
Less flexible	Highly flexible
Less reusable	More reusable
Only procedures and data are there	Inheritance, Encapsulation and polymorphism are the key features.
Uses top-Down approach	Uses Bottom-Up approach
Examples: C, Basic, FORTRAN	Example: C++,Java

5. Applications of OOP

There are mainly 4 types of applications that can be created using java programming:

1) Standalone Application

It is also known as desktop application or window-based application. An application that we need to install on every machine such as media player, antivirus etc. AWT and Swing are used in java for creating standalone applications.

2) Web Application

An application that runs on the server side and creates dynamic page, is called web application. Currently, servlet, jsp, struts, etc. technologies are used for creating web applications in java.

3) Enterprise Application

An application that is distributed in nature, such as banking applications etc. It has the advantage of high level security, load balancing and clustering. In java, EJB is used for creating enterprise applications.

4) Mobile Application

An application created for mobile devices is called Mobile Applications. Currently Android and Java ME are used for creating mobile applications.

In addition to the above applications, there are other applications that use OOP concepts as follow:

5. Neural Networks –The neural system of the human body is simulated into the machine, so that optimizations are carried out according to it.

6. Real Times –These Systems work bases on the time. Examples airline, rockets, etc;

7. **Expert System** –Expert's knowledge is integrated with system, that system will respond as the expert in the particular domain.

8. Database management systems – Data is constructed into tables and collection of table is called database. Java is used to process the data that is available in the database, such as insertion, deletion, display etc;.

6. History of JAVA

1. Brief history of Java
2. Java Version History

Java history is interesting to know. The history of java starts from Green Team. Java Team members (also known as **Green Team**), initiated a revolutionary task to develop a language for digital devices such as set-top boxes, televisions etc. For the green team members, it was an advance concept at that time. But, it was suited for internet programming. Later, Java technology as incorporated by Netscape. Currently, Java is used in internet programming, mobile devices, games, e-business solutions etc. There are given the major points that describe the history of java.

1) **James Gosling**, **Mike Sheridan**, and **Patrick Naughton** initiated the Java language project in June 1991. The small team of sun engineers called **Green Team**.

2) Originally designed for small, embedded systems in electronic appliances like set-top boxes.

3) Firstly, it was called **"Greentalk"** by James Gosling and file extension was **".gt"**

4) After that, it was called **Oak** and was developed as a part of the Green project. This name was inspired by the Oak Tree that stood outside Golsling's Office.

Why Oak name for java language?

5) **Why Oak?** Oak is a symbol of strength and chosen as a national tree of many countries like U.S.A., France, Germany, Romania etc. During 1991 to 1995 many people around the world contributed to the growth of the Oak, by adding the new features. **Bill Joy, Arthur Van Hoff**, **Jonathan Payne, Frank Yellin, and Tim Lindholm** were key contributors to the original prototype.

6) In 1995, Oak was renamed as **"Java"** because it was already a trademark by Oak Technologies.

Why Java name for java language?

7) **Why they chosen java name for java language?** The team gathered to choose a new name. The suggested words were "dynamic", "revolutionary", "Silk", "jolt", "DNA" etc. They wanted something that reflected the essence of the technology: revolutionary, dynamic, lively, cool, unique, and easy to spell and fun to say. According to James Gosling "Java was one of the top choices along with **Silk**". Since java was so unique, most of the team members preferred java.

8) Java is an island of Indonesia where **first coffee** was produced (called java coffee). Java coffee was consumed in large quantities by the GreenTeam.

9) Notice that Java is just a name not an acronym.

10) Originally developed by James Gosling at Sun Microsystems (which is now a subsidiary of Oracle Corporation) and released in 1995.

11) In 1995, Time magazine called **Java one of the Ten Best Products of 1995**.

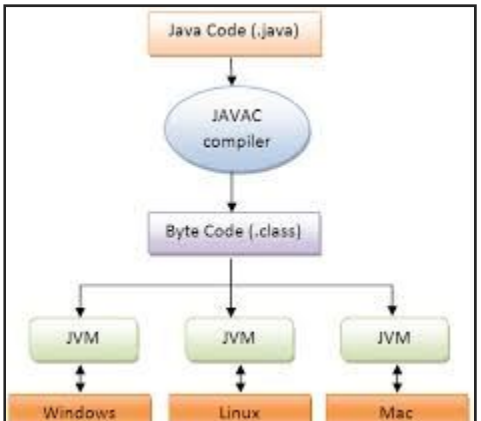
12) JDK 1.0 released in(January 23, 1996).

Java Version History

There are many java versions that has been released. Current stable release of Java is Java SE 8.

1. JDK Alpha and Beta (1995)
2. JDK 1.0 (23rd Jan, 1996)
3. JDK 1.1 (19th Feb, 1997)
4. J2SE 1.2 (8th Dec, 1998)
5. J2SE 1.3 (8th May, 2000)
6. J2SE 1.4 (6th Feb, 2002)
7. J2SE 1. 5 (30th Sep, 2004)
8. Java SE 1.6 (11th Dec, 2006)
9. Java SE 1.7 (28th July, 2011)
10. Java SE 1.8 (18th March, 2014)

7. Java Virtual Machine



The key that allows Java to solve both the **security and the portability** problems is the ***byte code***. The output of Java Compiler is not directly executable. Rather, it contains highly optimized set of instructions. This set of instructions is called, "**byte code**". This byte code is designed to be executed by Java Virtual Machine (JVM). The JVM also called as the ***Interpreter*** for byte code.

JVM also helps to solve many problems associated with web-based programs.

Translating a Java program into byte code makes it much easier to run a program in a wide variety of environments (or platforms) because only the JVM needs to be implemented for each platform. Once the run-time package exists for a given system, any Java program can run on it. Remember, although the details of the JVM will differ from platform to platform, all understand the same Java ***Byte Code***. Thus, the execution of byte code by the JVM is the easiest way to create truly portable programs.

The fact that a Java program is executed by the JVM also helps to make it ***secure***. Because the JVM is in control, it can contain the program and prevent it from generating side effects outside of the system.

In general, when a program is compiled to an intermediate form and then interpreted by a virtual machine, it ***runs slower than*** it would run if compiled to executable code. However, with Java, the differential between the two is not so great. Because byte code has been highly optimized, the use of byte code enables the JVM to execute programs much faster than you might expect

To give on-the-fly performance, the Sun began to design HotSpot Technology for Compiler, which is called, Just-In-Time compiler. The JIT, Compiler also produces output immediately after compilation.

8. Features of Java

There is given many features of java. They are also known as java buzzwords. The Java Features given below are simple and easy to understand.

- Simple
- Secure
- Portable
- Object-oriented
- Robust
- Multithreaded
- Architecture-neutral
- Interpreted
- High performance
- Distributed
- Dynamic

Simple

Java was designed to be easy for the professional programmer to learn and use effectively. According to Sun, Java language is simple because: syntax is based on C++ (so easier for programmers to learn it after C++). Removed many confusing and/or rarely-used features e.g., explicit pointers, operator overloading etc. No need to remove unreferenced objects because

there is Automatic Garbage Collection in java.

Secure

Once the byte code generated, the code can be transmitted to other computer in the world without knowing the internal details of the source code.

Portable

The byte code can be easily carried from one machine to other machine.

Object Oriented

Everything in Java is an Object. The object model in Java is simple and easy to extend, while primitive types, such as integers, are kept as high-performance non-objects.

Robust

The multi-plat-formed environment of the Web places extraordinary demands on a program, because the program must execute reliably in a variety of systems. Thus, the ability to create robust programs was given a high priority in the design of Java. Java also frees from having worry about many errors. Java is Robust in terms of **memory management and mishandled exceptions**. Java provides automatic memory management and also provides well defined exception handling mechanism.

Multithreaded

Java was designed to meet the real-world requirement of creating interactive, networked programs. To accomplish this, Java supports multithreaded programming, which allows you to write programs that do many things **simultaneously**.

Architecture-neutral

The Java designers made several hard decisions in the Java language and the Java Virtual Machine in an attempt to alter this situation. Their goal was "**write once; run anywhere, any time, forever.**" To a great extent, this goal was accomplished.

Interpreted and High Performance

Java enables the creation of cross-platform programs by compiling into an intermediate representation called Java byte code. This code can be executed on any system that implements the Java Virtual Machine. Most previous attempts at cross-platform solutions have done so at the expense of performance. As explained earlier, the Java byte code was carefully designed so that it would be easy to translate directly into native machine code for very high performance by using a just-in-time compiler.

Distributed

Java is designed for the distributed environment of the Internet because it handles TCP/IP protocols. In fact, accessing a resource using a URL is not much different from accessing a file. Java also supports ***Remote Method Invocation (RMI)***. This feature enables a program to invoke methods across a network.

Dynamic

Java programs carry with them substantial amounts of run-time type information that is used to verify and resolve accesses to objects at run time. This makes it possible to dynamically link code in a safe and expedient manner.

9. Program Structures

Simple Java Program

Example.java

```
class Example
{
        public static void main(String args[])
        {
                System.out.println("Hello World");
        }
}
```

Entering the Program

We can use any text editor such as "**notepad**' or "**dos text editor**". The source code is typed and is saved with "**.java**" as extension. The source code contains one or more class definitions. The program name will be same as class name in which main function is written. This in not compulsory, but by convention this is used. ***The source file is officially called as compilation unit***. We can even uses our choice of interest name for the program. If we use a different name than the class name, then compilation is done with program name, and running is done with class file name. To avoid this confusion and organize the programs well, it is suggested to put the same name for the program and class name, but not compulsory.

Compiling the Program

To compile the program, first execute the compiler, "**javac**", specifying the name of the source file on the command line, as shown bellow:

c:\>javac Example.java

The javac compiler creates the file called "**Example.class**", that contains the byte code version of the source code. This byte code is the intermediate representation of the source code that

contains the instructions that the Java Virtual Machine (JVM) will execute. Thus the output of the javac is not the directly executable code.

To actually run the program, we must use Java interpreter, called "java". This is interpreter the "**Example.class**" file given as input.
When the program is run with java interpreter, the following output is produced:

```
Hello World
```

Description of the every line of the program

The first line contains the keyword **class** and **class name**, which actually the basic unit for encapsulation, in which data and methods are declared.

Second line contains "{" which indicates the beginning of the class.

Third line contains the
public static void main(String args[])

where public is access specifier, when a member of a class is made public it can be accessed by the outside code also. The main function is the beginning of from where execution starts. Java is case-sensitive. "**Main**" is different from the "**main**". In main there is one parameter, String args, which is used to read the command line arguments.

Fourth line contains the "{", which is the beginning of the main function.

Fifth line contains the statement

System.out.println("Hello World");

Here "**System**" is the predefined class, that provides access to the system, and **out** is the output stream that is used to connect to the console. The **println()**, is used to display string passed to it. This can even display other information to.

10. Installation of JDK 1.6

Installing the JDK Software

If you do not already have the JDK software installed or if JAVA_HOME is not set, the Java CAPS installation will not be successful. The following tasks provide the information you need to install JDK software and set JAVA_HOME on UNIX or Windows systems.

The following list provides the Java CAPS JDK requirements by platform.

Solaris

JDK5: At least release 1.5.0_14

JDK6: At least release 1.6.0_03

IBM AIX

JDK5: The latest 1.5 release supported by IBM AIX

Linux (Red Hat and SUSE)

JDK5: At least release 1.5.0_14

JDK6: At least release 1.6.0_03

Macintosh

JDK5: The latest 1.5 release supported by Apple

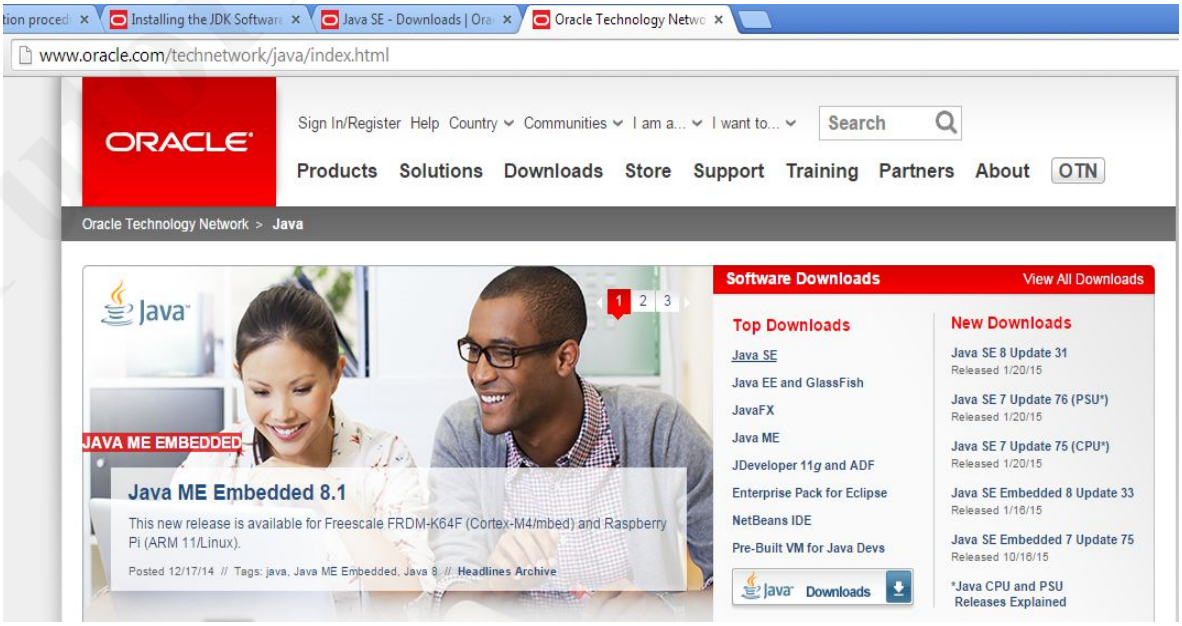
Microsoft Windows

JDK5: At least release 1.5.0_14

JDK6: At least release 1.6.0_03

To Install the JDK Software and Set `JAVA_HOME` on a Windows System

1. Install the JDK software.
   a. Go to `http://java.sun.com/javase/downloads/index.jsp`.

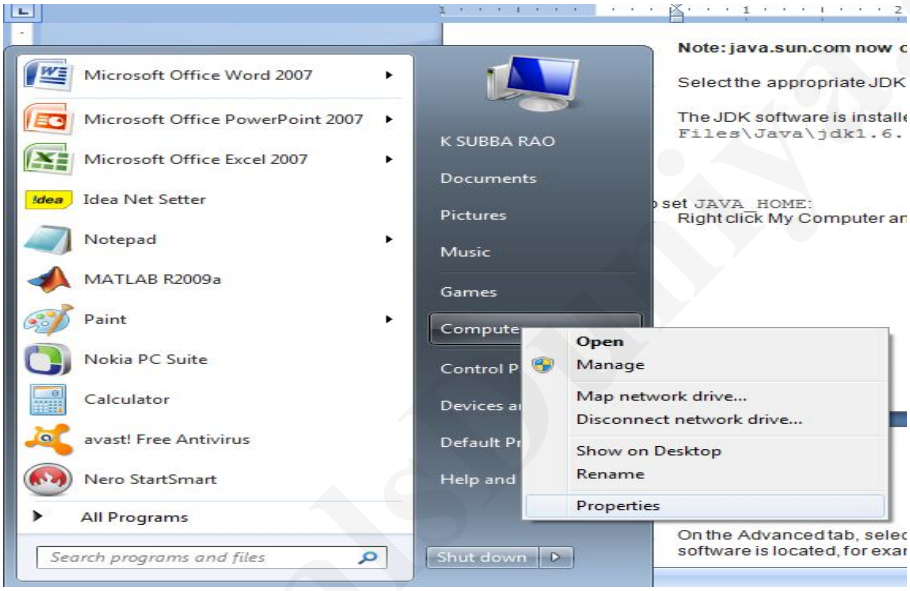


Note: java.sun.com now owned by oracle corporation

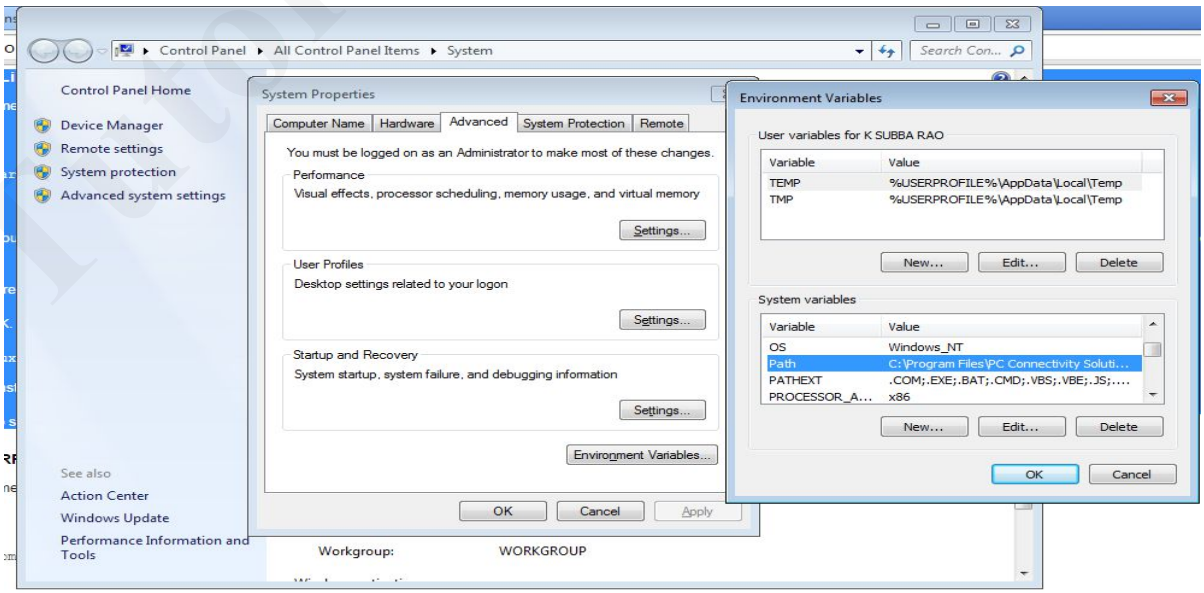
b. Select the appropriate JDK software and click Download.

The JDK software is installed on your computer, for example, at `C:\Program Files\Java\jdk1.6.0_02`. You can move the JDK software to another location if desired.

2. To set `JAVA_HOME`:
   a. Right click My Computer and select Properties.



   b. On the Advanced tab, select Environment Variables, and then edit `JAVA_HOME` to point to where the JDK software is located, for example, `C:\Program Files\Java\jdk1.6.0_02`.



TutorialsDuniya.com

Download FREE Computer Science Notes, Programs, Projects, Books PDF for any university student of BCA, MCA, B.Sc, B.Tech CSE, M.Sc, M.Tech at https://www.tutorialsduniya.com

- Algorithms Notes
- Artificial Intelligence
- Android Programming
- C & C++ Programming
- Combinatorial Optimization
- Computer Graphics
- Computer Networks
- Computer System Architecture
- DBMS & SQL Notes
- Data Analysis & Visualization
- Data Mining
- Data Science
- Data Structures
- Deep Learning
- Digital Image Processing
- Discrete Mathematics
- Information Security
- Internet Technologies
- Java Programming
- JavaScript & jQuery
- Machine Learning
- Microprocessor
- Operating System
- Operational Research
- PHP Notes
- Python Programming
- R Programming
- Software Engineering
- System Programming
- Theory of Computation
- Unix Network Programming
- Web Design & Development

Please Share these Notes with your Friends as well

Installation of the 32-bit JDK on Linux Platforms

This procedure installs the Java Development Kit (JDK) for 32-bit Linux, using an archive binary file (`.tar.gz`).

These instructions use the following file:

```
jdk-8uversion-linux-i586.tar.gz
```

1. Download the file.

   Before the file can be downloaded, you must accept the license agreement. The archive binary can be installed by anyone (not only root users), in any location that you can write to. However, only the root user can install the JDK into the system location.

2. Change directory to the location where you would like the JDK to be installed, then move the `.tar.gz` archive binary to the current directory.

3. Unpack the tarball and install the JDK.

4. `% tar zxvf jdk-8uversion-linux-i586.tar.gz`

   The Java Development Kit files are installed in a directory called `jdk1.8.0_version` in the current directory.

5. Delete the `.tar.gz` file if you want to save disk space.

UNIT – 2

Topics to be covered:

Java Basics: Data types, variables, identifiers, Keywords, Literals, Operators, Expressions, Precedence Rules and Associativity, Type Conversion and Casting, Flow of Control: Branching, Conditional, Loops, **Classes** and objects, creating objects, methods, constructors, constructor-overloading, **Cleaning up unused objects and garbage collection**, overloading methods and constructors, Class variables and methods, Static keyword, this keyword, Arrays and command line arguments

Data Types

Java is strongly typed language. The safety and robustness of the Java language is in fact provided by its strict type. There are two reasons for this: First, every variable and expression must be defined using any one of the type. Second, the parameters to the method also should have some type and also verified for type compatibility. ***Java language 8 primitive data types:***

The primitive data types are: char, byte, short, int, long, float, double, boolean. These are again grouped into 4 groups.

1. **Integer Group**: The integer group contains byte, short, int, long. These data types will need different sizes of the memory. These are assigned positive and negative values. The width and ranges of these values are as follow:

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

byte:

The smallest integer type is **byte**. This is a signed 8-bit type that has a range from –128 to 127.Variables of type **byte** are especially useful when you're working with a stream of data froma network or file. They are also useful when you're working with raw binary data that maynot be directly compatible with Java's other built-in types.Byte variables are declared by use of the **byte** keyword.

For example, the followingdeclares 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. 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 a rangefrom –2,147,483,648 to 2,147,483,647. In addition to other uses, variables of type **int** arecommonly employed to control loops and to index arrays. We can store byte and short values in an int.

Example
int x=12;

long:
long is a signed 64-bit type and is useful for those occasions where an **int** type is not largeenough to hold the desired value. The range of a **long** is quite large. This makes it usefulwhen big, whole numbers are needed.
Example
long x=123456;

2. **Floating-Point Group**

Floating-point numbers, also known as ***real***numbers, are used when evaluating expressionsthat require fractional precision. These are used with operations such as square root, cosine, and sine etc. There are two types of Floating-Point numbers: float and double. The float type represents single precision and double represents double precision. Their width and ranges are as follows:

Name	Width in Bits	Approximate Range
double	64	4.9e–324 to 1.8e+308
float	32	1.4e–045 to 3.4e+038

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. Variables of type **float** are usefulwhen you need a fractional component, but don't require a large degree of precision.

Example:
float height, price;

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 the math functions, such as **sin( )**, **cos( )**, and **sqrt( )**, return **double** values.

Example:
double area,pi;

Example program to calculate the area of a circle

```
import java.io.*;
class Circle
{
public static void main(String args[])
    {
        double r,area,pi;
        r=12.3;
        pi=3.14;
        area=pi*r*r;
        System.out.println("The Area of the Circle is:"+area);
    }
}
```

3. Characters Group

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 8 bits wide. This is *not* thecase in Java. Instead, Java uses Unicode to represent characters. *Unicode* defines a fullyinternational character set that can represent all of the characters found in all humanlanguages.

Java **char** is a 16-bit type. The range of a **char** is 0 to 65,536. There are no negative **char**s. The standard set of characters known as ASCII still ranges from 0 to 127 as always, and theextended 8-bit character set, ISO-Latin-1, ranges from 0 to 255.

Here is a program that demonstrates 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.println(ch1 + " " + ch2);
        }
}
```

4. Booleans

Java has a primitive type, called **boolean**, for logical values. It can have only one of twopossible values, **true** or **false**. This is the type returned by all relational operators, as in thecase of **a < b**.

Here is a program that demonstrates the boolean type:

```
// Demonstrate boolean values.
class BoolTest
 {
public static void main(String args[])
{
        boolean b;
        b = false;
        System.out.println("b is " + b);
        b = true;
        System.out.println("b is " + b);
        // a boolean value can control the if statement
        if(b) System.out.println("This is executed.");
        b = false;
        if(b) System.out.println("This is not executed.");
        // outcome of a relational operator is a boolean value
        System.out.println("10 > 9 is " + (10 > 9));
        }
}
```

Identifiers

Identifiers are used for **class names, method names, and variable names**. An identifier may be any descriptive sequence of uppercase and lowercase letters, numbers, or the underscore and dollar-sign characters. They must not begin with a number, lest they be confused with anumeric literal. Again, Java is case-sensitive, so **VALUE** is a different identifier than **Value**.Some examples of valid identifiers are:

Rules for Naming Identifier:

1. ***The first character of an identifier must be a letter, or dollar($) sign.***
2. ***The subsequent characters can be letters, an underscore, dollar sign or digit.***
3. ***White spaces are not allowed within identifiers.***
4. ***Identifiers are case sensitive so VALUE is a different identifier than Value***

Average	Height	A1	Area_Circle

Invalid Identifiers are as follow:

2types	Area-circle	Not/ok

Naming Convention for Identifiers

- **Class or Interface-** These begin with a capital letter. The first alphabet of every internal word is capitalized. Ex: class **Myclass;**
- **Variable or Method –**These start with lower case letters. The first alphabet of every internal word is capitalized. Ex:- int **totalPay;**

- **Constants-** These are in upper case. Underscore is used to separate the internal word. Ex:-final **double PI=3.14**;
- **Package –** These consist of all lower-case letters. Ex:- **import java.io.*;**

Variables

The variable is the basic unit of storage in a Java program. A variable is defined by thecombination of an identifier, a type, and an optional initializer. In addition, all variables havea scope, which defines their visibility, and a lifetime.

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/literal*][, *identifier* [= *value/literal*] ...] ;**

Here the type is any primitive data types, or class name. The identifier is the name of the variable. We can initialize the variable by specifying the equal sign and value.
Example

```
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; // declares an approximation of pi.
char x = 'x';      //the variable x ahs the value 'x'
```

Dynamic Initialization of the variable
We can also assign a value to the variable dynamically as follow:

```
int x=12;
int y=13;
float z=Math.sqrt(x+y);
```

The Scope and Lifetime of Variables

- ✓ Java allows, to declare a variable within any block.
- ✓ A block begins with opening curly brace and ended with end curly brace.
- ✓ Thus, each time we start new block, we create new scope.
- ✓ A scope determines what objects are visible to parts of your program. It also determines the life time of the objects.
- ✓ Many programming languages define two scopes: **Local and Global**
- ✓ As a general rule a variable defined within one scope, is not visible to code defined outside of the scope.
- ✓ Scopes can be also nested. The variable defined in **outer scope** are visible to the **inner scopes**, but reverse is not possible.

Example code

```
void function1()
{//outer block
        int a;
//here a,b,c are visible to the inner scope
int a=10;
if(a==10)
{// inner block
        int b=a*20;
        int c=a+30;
}//end of inner block
b=20*2;
// b is not known here, which declared in inner scope
}//end of the outer block
```

Literals

A literal is a value that can be passed to a variable or constant in a program. Literals can be numeric, boolean, character, string notation or null. A constant value can be created using a literal representation of it. Here are some literals:

Interger literal	Character literal	Floating point literal	byteliteral
int x=25;	char ch=88;	flaot f=12.34	byte b=12;

Comments

In java we have three types of comments: single line comment, Multiple line comment, and document type comment.
Single line comment is represented with // (two forward slashes), Multiple comment lines represented with /*…….*/ (slash and star), and the document comment is represented with /**…….*/.

Separators

In Java, there are a few characters that are used as separators. The most commonly usedseparator in Java is the semicolon. As you have seen, it is used to terminate statements.The separators are shown in the following table:

Symbol	Name	Purpose
( )	Parentheses	Used to contain lists of parameters in method definition and invocation. Also used for defining precedence in expressions, containing expressions in control statements, and surrounding cast types.
{ }	Braces	Used to contain the values of automatically initialized arrays. Also used to define a block of code, for classes, methods, and local scopes.
[ ]	Brackets	Used to declare array types. Also used when dereferencing array values.
;	Semicolon	Terminates statements.
,	Comma	Separates consecutive identifiers in a variable declaration. Also used to chain statements together inside a **for** statement.
.	Period	Used to separate package names from subpackages and classes. Also used to separate a variable or method from a reference variable.

The Java Keywords

There are **50 keywords** currently defined in the Java language (see Table bellow). These keywords,combined with the syntax of the operators and separators, form the foundation of the Javalanguage. These keywords cannot be used as names for a variable, class, or method.

abstract	continue	for	new	switch
assert	default	goto	package	synchronized
boolean	do	if	private	this
break	double	implements	protected	throw
byte	else	import	public	throws
case	enum	instanceof	return	transient
catch	extends	int	short	try
char	final	interface	static	void
class	finally	long	strictfp	volatile
const	float	native	super	while

Operators

An operator performs an operation on one or more operands. Java provides a rich operator environment. An operator that performs an operation on one operand is called ***unary operator.*** An operator that performs an operation on two operands is called ***binary operator***.

Most of its operators can be divided into thefollowing ***four groups:arithmetic, bitwise, relational, and logical***. Java also defines someadditional operators that handle certain special situations. In Java under ***Binary operator*** we have Arithmetic, relational, Shift, bitwise and assignment operators. And under **Unary operators** we have ++, - - , !(Boolean not), ~(bitwise not) operators.

Arithmetic Operators

Arithmetic operators are used in mathematical expressions in the same way that they areused 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 the arithmetic operations are of numeric type. The Boolean operands are not allowed to perform arithmetic operations. The basic arithmetic operators are: addition, subtraction, multiplication, and division.

Example program to perform all the arithmetic operations

Arith.java

```
import java.io.*;
class Arith
{
	public static void main(String args[])
	{
		int a,b,c,d;
		a=5;
		b=6;
		//arithmetic addition
		c=a+b;
		System.out.println("The Sum is :"+c);
		//aritmetic subtraction
		d=a-b;
		System.out.println("The Subtractio is :"+d);
		//arithmetic division
		c=a/b;
		System.out.println("The Dision is :"+c);
		//arithmetic multiplication
		d=a*b;
		System.out.println("The multiplication is :"+d);
	}
```

}

The Modulus Operator

The modulus operator, **%**, returns the remainder of a division operation. It can be applied tofloating-point types as well as integer types. The following demonstrates the %

Modulus.java

```
// Demonstrate the % operator.
class Modulus
{
public static void main(String args[])
      {
            int x = 42;
            double y = 42.25;
            System.out.println("x mod 10 = " + x % 10);
            System.out.println("y mod 10 = " + y % 10);
      }
}
```

Arithmetic Compound Assignment Operators

Java provides special operators that can be used to combine an arithmetic operation withan assignment. As you probably know, statements like the following are quite common inprogramming:
a = a + 4;

In Java, you can rewrite this statement as shown here:

a += 4;

This version uses the **+=** *compound 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;

Increment and Decrement

The ++ and the – – are Java's increment and decrement operators. The increment operator increases its operand by one. The decrement operator decreases its operand by one. For example, this statement:

x = x + 1;

can be rewritten like this by use of the increment operator:

x++;

Similarly, this statement:
x = x - 1;

is equivalent to
x--;

Note: If we write increment/decrement operator after the operand such expression is called post increment decrement expression, if written before operand such expression is called pre increment/decrement expression

The following program demonstrates the increment and decrement operator.

IncDec.java

```
// Demonstrate ++ and --
class IncDec
{
public static void main(String args[])
   {
      int a = 1;
      int b = 2;
      int c;
      int d;
      c = ++b; //pre increment
      d = a--; //post decrement
      c++;  //post increment
      d--; //post decrement
      System.out.println("a = " + a);
      System.out.println("b = " + b);
      System.out.println("c = " + c);
      System.out.println("d = " + d);
   }
}
```

The Bitwise Operators

Java defines several *bitwise operators* that can be applied to the integer types, **long**, **int**, **short**,**char**, and **byte**. These operators act upon the individual bits of their operands. They aresummarized in the following table:

Operator	Result
~	Bitwise unary NOT
&	Bitwise AND
|	Bitwise OR
^	Bitwise exclusive OR
>>	Shift right
>>>	Shift right 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

These operators are again classified into 3 categories: **Logical operators, Shift operators, and Relational operator**

The Bitwise Logical Operators

The bitwise logical operators are **&**, **|**, **^**, and **~**. The following table shows the outcome ofeach operation. 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, |, 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 wherever the second operand has a 1 bit. Wherever 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
```

Using the Bitwise Logical Operators

The following program demonstrates the bitwise logical operators:

BitLogic.java

```
// Demonstrate the bitwise logical operators.
class BitLogic
{
public static void main(String args[])
{

int a = 3; // 0 + 2 + 1 or 0011 in binary
int b = 6; // 4 + 2 + 0 or 0110 in binary
int c = a | b;
int d = a & b;
int e = a ^ b;
int f = (~a & b) | (a & ~b);
int g = ~a & 0x0f;
System.out.println(" a|b = " +c);
System.out.println(" a&b = " +d);
System.out.println(" a^b = " +e);
System.out.println("~a&b|a&~b = " +f);
System.out.println(" ~a = " + g);
}
}
```

Shift Operators: (left shift and right shift)

The Left Shift

The left shift operator, <<, shifts all of the bits in a value to the left a specified number of times. It has this general form:

value << *num*

Here, ***num***specifies the number of positions to left-shift the value in ***value***. That is, the **<<**moves all of the bits in the specified value to the left by the number of bit positions specifiedby *num*.

The Right Shift

The right shift operator, **>>**, shifts all of the bits in a value to the right a specified number oftimes. Its general form is shown here:

value >> *num*

Here, ***num***specifies the number of positions to left-shift the value in ***value***. That is, the >>moves all of the bits in the specified value to the right by the number of bit positions specifiedby *num*.

ShiftBits.java

```
class ShiftBits
{
	public static void main(String args[])
	{
		byte b=6;
		int c,d;
		//left shift
		c=b<<2;
		//right shift
		d=b>>3;
		System.out.println("The left shift result is :"+c);
		System.out.println("The right shift result is :"+d);
	}
}
```

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:

TutorialsDuniya.com

Download FREE Computer Science Notes, Programs, Projects, Books PDF for any university student of BCA, MCA, B.Sc, B.Tech CSE, M.Sc, M.Tech at https://www.tutorialsduniya.com

- Algorithms Notes
- Artificial Intelligence
- Android Programming
- C & C++ Programming
- Combinatorial Optimization
- Computer Graphics
- Computer Networks
- Computer System Architecture
- DBMS & SQL Notes
- Data Analysis & Visualization
- Data Mining
- Data Science
- Data Structures
- Deep Learning
- Digital Image Processing
- Discrete Mathematics
- Information Security
- Internet Technologies
- Java Programming
- JavaScript & jQuery
- Machine Learning
- Microprocessor
- Operating System
- Operational Research
- PHP Notes
- Python Programming
- R Programming
- Software Engineering
- System Programming
- Theory of Computation
- Unix Network Programming
- Web Design & Development

Please Share these Notes with your Friends as well

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.

Any type in Java, including **integers**, **floating-point** numbers, **characters**, and **Booleans**can be compared using the equality test, ==, and the inequality test, **!**=. Notice that in Javaequality is denoted with two equal signs, not one. (Remember: a single equal sign is theassignment operator.) Only numeric types can be compared using the ordering operators.That is, only integer, floating-point, and character operands may be compared to see whichis **greater or less than the other**.

Short-Circuit Logical Operators (|| and &&)

Java provides two interesting Boolean operators not found in many other computer languages.These are secondary versions of the Boolean AND and OR operators, and are known as*short-circuit* logical operators.

When we use || operator if left hand side expression is true, then the result will be true, no matter what is the result of right hand side expression. In the case of && if the left hand side expression results true, then only the right hand side expression is evaluated.
Example 1: (expr1 || expr2) Example2: (expr1 && expr2)

The Assignment Operator

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* (three-way) *operator***that can replace certain types of if-then-elsestatements. This operator is the **?**. It can seem somewhat confusing at first, but the **?** can beused very effectively once mastered. The **?** has this general form:

expression1* ? *expression2* : *expression3

Here, *expression1* can be any expression that evaluates to a **boolean** value. If *expression1* is**true**, then *expression2* is evaluated; otherwise, *expression3* is evaluated. The result of the **?**operation is that of the expression evaluated. Both *expression2* and *expression3* are requiredto return the same type, which can't be **void**.Here is an example of the way that the **?** is employed:

Test.java

```
class Test
{
public static void main(String args[])
{
        int x=4,y=6;
        int res= (x>y)?x:y;
        System.out.println("The result is :"+res);
}
}
```

Expressions:

An expression is a combination of operators and/or operands. Expressions are used to create Objects, Arrays, and Assigning values to variables and so on. The expression may contain identifiers, literals, separators, and operators.

Example:-

```
        int m=2,n=3,o=4;
        int y=m*n*o;
```

Operator Precedence Rules and Associativity

The precedence rules are used to determine the priority in the case there are two operators with different precedence. The Associativity rules are used to determine the order of evaluation, in case two operators are having the same precedence.Associativity is two types: Left to Right and Right to Left.

Table shows the order of precedence for Java operators, from highest to lowest. Notice that the first row shows items that you may not normally think of as operators: **parentheses, square brackets, and the dot operator**. Technically, these are called ***separators***, but they act like operators in an expression. Parentheses are used to alter the precedence of an operation. As you know from the previous chapter, the square brackets provide array indexing. The dot operator is used to dereference objects.

Highest			
( )	[ ]	.	
++	– –	~	!
*	/	%	
+	–		
>>	>>>	<<	
>	>=	<	<=
==	!=		
&			
^			
|			
&&			
||			
?:			
=	op=		
Lowest			

Note: The operators =,?:, ++, and – are having Right to Left Associativity. The remaining Operators are having Left to Right Associativity.

Using Parentheses

Parenthesesraise the precedence of the operations that are inside them. This is often necessaryto obtain the result you desire. For example, consider the following expression:

a >> b + 3

This expression first adds 3 to **b** and then shifts **a** right by that result. That is, this expression can be rewritten using redundant parentheses like this:

a >> (b + 3)

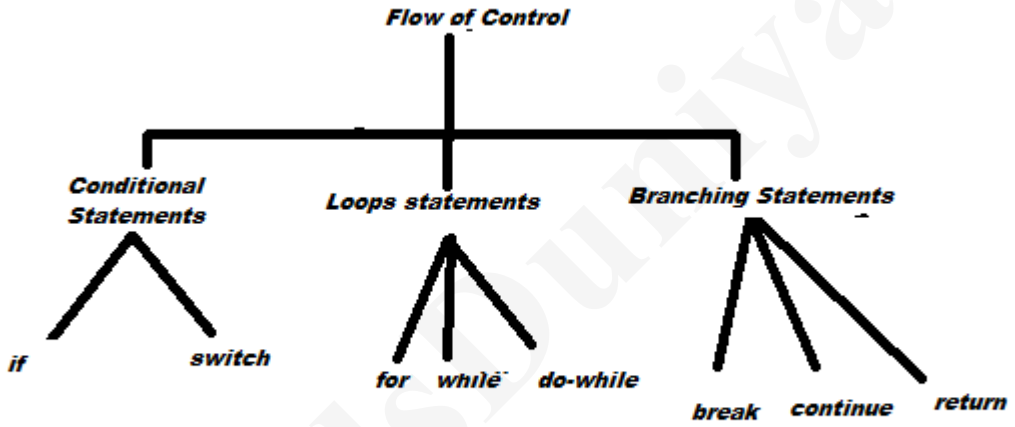
However, if you want to first shift **a** right by **b** positions and then add 3 to that result,you will need to parenthesize the expression like this:

(a >> b) + 3

Flow of Control (Control Statements)

The control statements are used to control the flow of execution and branch based on the status of a program. The control statements in Java are categorized into 3 categories:

i. **Conditional Statement (Selection Statements/Decision Making Statements)**
ii. **Loops (Iteration Statements)**
iii. **Branching Statements (Jump Statements)**



I. **Conditional Statement (Selection Statements/Decision Making Statements)**: These include**if** and **switch**. These statements allow the program to choose different paths of execution based on the outcome of the conditional expression.

if statement: This is the Java's conditional branch statement. This is used to route the execution through two different paths. The general form ofif statement will be as follow:

```
if (conditional expression)
{
        statement1
}
else
{
        statement2
}
```

Here the statements inside the block can be single statement or multiple statements. The conditional expression is any expression that returns the **Boolean** value. The else clause is optional.The if works as follows: if the conditional expression is true, then statement1 will be executed. Otherwise statement2 will be executed.

Example:
Write a java program to find whether the given number is even or odd?

EvenOdd.java

```
import java.io.*;
classs EvenOdd
{
	public static void main(String args[])
	{
		int n;
		System.out.println("Enter the value of n");
		DataInputStream dis=new DataInputStream(System.in);
		n=Integer.parseInt(dis.readLine());
		if(n%2==0)
		{
			System.out.println(n+" is the Even Number");
		}
		else
		{
			System.out.println(n+"is the ODD Number");
		}
	}
}
```

Nested if: The nested if statement is an if statement, that contains another if and else inside it. The nested if are very common in programming. When we nest ifs, the else always associated with the nearest if.
The general form of the nested if will be as follow:

```
if(conditional expresion1)
{
	if(conditional expression2)
	{
		statements1;
	}
	else
	{
		satement2;
	}
}
else
{
	statement3;
```

}

Example program:

Write a java Program to test whether a given number is positive or negative.

Positive.java

```
import java.io.*;
class Positive
{
public static void main(String args[])  throws IOException
{
      int n;
      DataInputStream dis=new DataInputStream(System.in);
      n=Integer.parseInt(dis.readLine());
      if(n>-1)
          {
            if(n>0)
                  System.out.println(n+ " is positive no");
             }
             else
                  System.out.println(n+ " is Negative no");

}
}
```

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*;**

The **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 isbypassed. If none of the conditions is true, then the final **else** statement will be executed.

Example Program:

Write a Java Program to test whether a given character is Vowel or Consonant?

Vowel.java

```
import java.io.*;
class Vowel
{
```

```
public static void main(String args[])  throws IOException
{
 char ch;
          ch=(char)System.in.read();
      if(ch=='a')
            System.out.println("Vowel");
      else if(ch=='e')
            System.out.println("Vowel");

      else if(ch=='i')
            System.out.println("Vowel");
      else if(ch=='o')
            System.out.println("Vowel");
      else if(ch=='u')
            System.out.println("Vowel");
      else

            System.out.println("consonant");



}
}
```

The Switch statement

The **switch** statement is Java's multiway 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.

The **switch** statement works like this: The value of the expression is compared with eachof the literal values in the **case** statements. If a match is found, the code sequence followingthat **case** statement is executed. If none of the constants matches the value of the expression,then the **default** statement is executed. However, the **default** statement is optional. If no **case**matches and no **default** is present, then no further action is taken.

The **break** statement is used inside the **switch** to terminate a statement sequence. Whena **break** statement is encountered, execution branches to the first line of code that follows theentire **switch** statement. This has the effect of "jumping out" of the **switch**.

Write a Java Program to test whether a given character is Vowel or Consonant?(Using Switch)

SwitchTest.java

```
import java.io.*;
class SwitchTest
{
public static void main(String args[])  throws IOException
{
      char ch;
      ch=(char)System.in.read();
      switch(ch)
      {
       //test for small letters
              case 'a': System.out.println("vowel");
                 break;
           case 'e': System.out.println("vowel");
                 break;
           case 'i': System.out.println("vowel");
                 break;
           case 'o': System.out.println("vowel");
                 break;
           case 'u': System.out.println("vowel");
                 break;
          //test for capital letters
                case 'A': System.out.println("vowel");
                 break;
           default: System.out.println("Consonant");
      }
}
```

```
}
```

The **break** statement is optional. If you omit the **break**, execution will continue on into thenext **case**. It is sometimes desirable to have multiple **case**s without **break** statements betweenthem. For example, consider the following program.

```
class Switch
 {
public static void main(String args[])
{
        int month = 4;
        String season;
        switch (month)
        {
                case 12:
                case 1:
                case 2:season = "Winter";
                        break;
                case 3:
                case 4:
                case 5:season = "Spring";
                        break;
                case 6:
                case 7:
                case 8:season = "Summer";
                        break;
                case 9:
                case 10:
                case 11:season = "Autumn";
                        break;
                default:season = "Bogus Month";
}
System.out.println("April is in the " + season + ".");
}
}
```

Nested switch Statements

You can use a **switch** as part of the statement sequence of an outer **switch**. This is called a*nested* **switch**. Since a **switch** statement defines its own block, no conflicts arise between the**case** constants in the inner **switch** and those in the outer **switch**. For example, the followingfragment is perfectly valid:

```
switch(expression)   //outer switch
{
```

```
        case 1: switch(expression)  // inner switch
                {
                        case 4: //statement sequence
                                break;
                        case 5://statement sequence
                                break;
                }  //end of inner switch
                break;
        case 2: //statement sequence
                break;
        default: //statement sequence
} //end of outer switch
```

There are three important features of the switch statement to note:

- The **switch** differs from the **if** in that **switch** can only test for equality, whereas **if** can evaluate any type of Boolean expression. That is, the **switch** looks only for a match between the value of the expression and one of its **case** constants.

- No two **case** constants in the same **switch** can have identical values. Of course, a **switch** statement and an enclosing outer **switch** can have **case** constants in common.

- A**switch** statement is usually more efficient than a set of nested **if**s.

II. Loops (Iteration Statements)

Java's iteration statements are **for**, **while**, and **do-while**. These statements create what wecommonly call *loops*. As you probably know, a loop repeatedly executes the same set ofinstructions until a termination condition is met.

i. while

The '***while***' loops is used to repeatedly execute a block of statements based on a condition. The condition is evaluated before the iteration starts. A ***'for'*** loop is useful, when we know the number of iterations to be executed in advance. If we want to execute the loop for indefinite number of times, a while loop may be better choice. For example, if you execute a ***query to fetch data from database,*** you will not know the exact number of records returned by the query.

Syntax:

```
while(condition)
{
        // body of loop
        increment or decrement statement
```

}

The ***condition***can be any Boolean expression. The body of the loop will be executed as longas the conditional expression is true. When ***condition***becomes false, control passes to thenext line of code immediately following the loop. The curly braces are unnecessary if onlya single statement is being repeated.

Example program:
Write a java program to add all the number from 1 to 10.

WhileTest.java

```
import java.io.*;
class WhileTest
{
	public static void main(String args[])
	{
		int i=1,sum=0;
		while(i<=10)
		{
		 sum=sum+i;
		 i++;
		}
		System.out.println("The sum is :"+sum);
	}
}
```

ii. do-while statement

However, sometimes it is desirable to execute the body of a 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 terminationexpression at the end of the loop rather than at the beginning.

Fortunately, Java supplies aloop that does just that: the **do-while**. The **do-while** loop always executes its body at leastonce, 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-while** loop first executes the body of the loop and then evaluatesthe conditional expression. If this expression is true, the loop will repeat. Otherwise, the loopterminates. As with all of Java's loops, *condition* must be a Boolean expression.

Example program:
Write a java program to add all the number from 1 to 10. (using do-while)

WhileTest.java

```
import java.io.*;
class WhileTest
{
	public static void main(String args[])
	{
		int i=1,sum=0;
do
		{
		 sum=sum+i;
		 i++;
		}while(i<=10);
		System.out.println("The sum is :"+sum);
	}
}
```

Note 1: Here the final value of the i will be 11. Because the body is executed first, then the condition is verified at the end.

Note 2: The **do-while** loop is especially useful when you process a **menu** selection, because you will usually want the body of a menu loop to execute at least once.

Example program: Write a Java Program to perform various operations like addition, subtraction, and multiplication based on the number entered by the user. And Also Display the Menu.

DoWhile.java

```
import java.io.*;
class DoWhile
{
	public static void main(String args[])  throws IOException
	{
		int n,sum=0,i=0;
		DataInputStream dis=new DataInputStream(System.in);

		do
		{
		System.out.println("Enter your choice");
		System.out.println("1 Addition");
		System.out.println("2 Subtraction");
		System.out.println("3 Multiplicaton");
		n=Integer.parseInt(dis.readLine());
		System.out.println("Enter two Numbers");
		int a=Integer.parseInt(dis.readLine());
		int b =Integer.parseInt(dis.readLine());
		int c;
		switch(n)
		{
```

```
                case 1: c=a+b;
                        System.out.println("The addition is :"+c);
                        break;
                case 2: c=a-b;
                        System.out.println("The addition is :"+c);
                        break;
                case 3: c=a*b;
                        System.out.println("The addition is :"+c);
                        break;
                default:System.out.println("Enter Correct Number");

            }

            } while(n<=3);


        }
}
```

iii. for statement

The for loop groups the following three common parts into one statement: ***Initialization, condition and Increment/ Decrement.***

Syntax:

```
for(initialization; condition; iteration)
{
// body of the loop
}
```

The **for** loop operates as follows. When the loop first starts, the ***initialization***portion ofthe 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 thatthe initialization expression is only executed once. Next, ***condition***is evaluated. This must bea Booleanexpression. It usually tests the loop control variable against a target value. If thisexpression is true, then the body of the loop is executed. If it is false, the loop terminates. Third, ***Increment/Decrement*** is used to increment /decrement the loop control variable value by one.

Example program: same program using the for loop

ForTest.java

```
import java.io.*;
class ForTest
{
      public static void main(String args[])
      {
            int i,sum=0;
           for(i=1;i<=10;i++)
```

```
            {
          sum=sum+i;
 }
            System.out.println("The sum is :"+sum);
      }
}
```

There are some important things about the for loop

1. The initialization of the loop controlling variables can be done inside for loop.
   Example:
   for(int i=1;i<=10;i++)
2. We can write any Boolean expression in the place of the condition for second part of the loop.
   Example: where b is a Boolean data type
   boolean b=false;
   for(int i=1; !b;i++)
   {
   //body of the loop
   b=true;
   }
   This loop executes until the b is set to the true;

3. We can also run the loop infinitely, just by leaving all the three parts empty.
   Example:

```
for( ;  ;)
{
        //body of the loop
}
```

For each version of the for loop:

A for loop also provides another version, which is called **Enhanced Version** of the for loop. The general form of the for loop will be as follow:

```
for(type itr_var:collection)
{
        //body of the loop
}
```

Here, type is the type of the iterative variable of that receives the elements from collection, one at a time, from beginning to the end. The collection is created sung the array.

Example program:
Write a java program to add all the elements in an array?

ForEach.java

```
import java.io.*;
class ForEach
```

TutorialsDuniya.com

Download FREE Computer Science Notes, Programs, Projects, Books PDF for any university student of BCA, MCA, B.Sc, B.Tech CSE, M.Sc, M.Tech at https://www.tutorialsduniya.com

- Algorithms Notes
- Artificial Intelligence
- Android Programming
- C & C++ Programming
- Combinatorial Optimization
- Computer Graphics
- Computer Networks
- Computer System Architecture
- DBMS & SQL Notes
- Data Analysis & Visualization
- Data Mining
- Data Science
- Data Structures
- Deep Learning
- Digital Image Processing
- Discrete Mathematics
- Information Security
- Internet Technologies
- Java Programming
- JavaScript & jQuery
- Machine Learning
- Microprocessor
- Operating System
- Operational Research
- PHP Notes
- Python Programming
- R Programming
- Software Engineering
- System Programming
- Theory of Computation
- Unix Network Programming
- Web Design & Development

Please Share these Notes with your Friends as well

```
{
        public static void main(String args[])
    {
        int i, a[], sum=0;
                a=new int[10];
        a={12,13,14,15,16};
          for(int x:a)
           {
         sum=sum+x;
}
           System.out.println("The sum is :"+sum);
    }
}
```

III. Branching Statements (The Jump Statements)

Java supports three jump statements: **break**, **continue**, and **return**. These statements transfer control to another part of your program.

i. break statement

In Java, the **break** statement has three uses. First, as you have seen, it terminates a statement sequence in a **switch** statement. Second, it can be used to exit a loop. Third, it can be used as "civilized" form of goto statement.

Using break to Exit a Loop

By using **break**, you can force immediate termination of a loop, bypassing the conditionalexpression and any remaining code in the body of the loop. When a **break** statement isencountered inside a loop, the loop is terminated and program control resumes at the nextstatement following the loop. Here is a simple example:

// Using break to exit a loop.

```
class BreakLoop
{
public static void main(String args[])
{
        for(int i=0; i<100; i++)
        {
                if(i == 10) break; // terminate loop if i is 10
                System.out.println("i: " + i);
        }
System.out.println("Loop complete.");
}
}
```

Using break as a Form of Goto

In addition to its uses with the **switch** statement and loops, the **break** statement can also beemployed by itself to provide a "civilized" form of the goto statement. For example, the goto can be useful when you areexiting from a deeply nested set of loops. To handle such situations, Java defines an expandedform of the **break** statement. By using this form of **break**, you can, for example, break out ofone or more blocks of code.

The general form of the labeled **break** statement is shown here:

break *label*;

Most often, *label* is the name of a label that identifies a block of code. This can be a stand-aloneblock of code but it can also be a block that is the target of another statement. When this form of**break** executes, control is transferred out of the named block. The labeled block must enclosethe **break** statement, but it does not need to be the immediately enclosing bloc.

To name a block, put a label at the start of it. A *label* is any valid Java identifier followedby a colon. Once you have labeled a block, you can then use this label as the target of a**break** statement.

Example code:

```
class Break
{
public static void main(String args[])
{
boolean t = true;
first: {
second: {
third: {
        System.out.println("Before the break.");
        if(t) break second; // break out of second block
        System.out.println("This won't execute");
}
System.out.println("This won't execute");
}
System.out.println("This is after second block.");
}
}
}
```

Running this program generates the following output:

```
Before the break.
```

This is after second block.

ii. continue statement

It is used to stop the current iteration and go back to continue with next iteration. Sometimes it is useful to force an early iteration of a loop. That is, you might want to continuerunning the loop but stop processing the remainder of the code in its body for this particulariteration.In **while** and **do-while** loops, a **continue** statementcauses control to be transferred directly to the conditional expression that controls the loop.In a **for** loop, control goes first to the iteration portion of the **for** statement and then to theconditional expression. For all three loops, any intermediate code is bypassed.Here is an example program that uses **continue** to cause two numbers to be printed oneach line:

// Demonstrate continue.

```
class Continue
{
public static void main(String args[])
{
for(int i=1; i<=10; i++)
{

                if (i%5 == 0) continue;
                System.out.print(i + " ,");}
}
}
```

Here all the numbers from 1 to 10 except 5 are printed. as 1,2,3,4,6,7,8,9,10.

iii. return statement

The last control statement is **return**. The **return** statement is used to explicitly return from a method. That is, it causes program control to transfer back to the **caller of the method**. As such, it is categorized as a jump statement.

Example code

```
class Test
{
        p s v main(String args[])      // caller of the method
        {
                int a=3,b=4;
                int x=method(a,b);//function call
                System.out.println("The sum is :"+x);
        }
        int method(int x,int y)     // called method
        {
```

```
            return (x+y);
        }
}
```

After computing the result the control is transferred to the caller method, that main in this case.

Type Conversion and casting

There are two types of conversion. They are ***Implicit Conversion, and Explicit Conversion***.

Implicit Conversion

In the case of the implicit conversion type conversion is automatically performed by java when the types are compatible. For example, the **int** can be assigned to **long**. The **byte** cane assigned to **short**. However, all the types are compatible, thus all the type conversions are implicitly allowed.For example, double is not compatible with byte.

Conditions for automatic conversion

1. The two types must be compatible
2. The destination type must be larger than the source type

When automatic type conversion takes place the ***widening conversion*** takes place. For example,

```
int a;    //needs 32 bits
byte b=45;  //needs the 8 bits
a=b;  // here 8 bits data is placed in 32 bit storage. Thus widening takes place.
```

Explicit Conversion

Fortunately, it is still possible obtain the conversion between the incompatible types. This is called explicit type conversion. Java provides a special keyword "**cast**" to facilitate explicit conversion. For example, sometimes we want to assign int to byte, this will not be performed automatically, because byte is **smaller** than int. This kind of conversion is sometimes called "***narrowing conversion***".Since, you are explicitly making the value narrow.The general form of the cast will be as follow:

```
destination_variable=(target type) value;
```

Here the target type specifies the destination type to which the value has to be converted.

Example

```
int a=1234;
byte b=(byte) a;
```

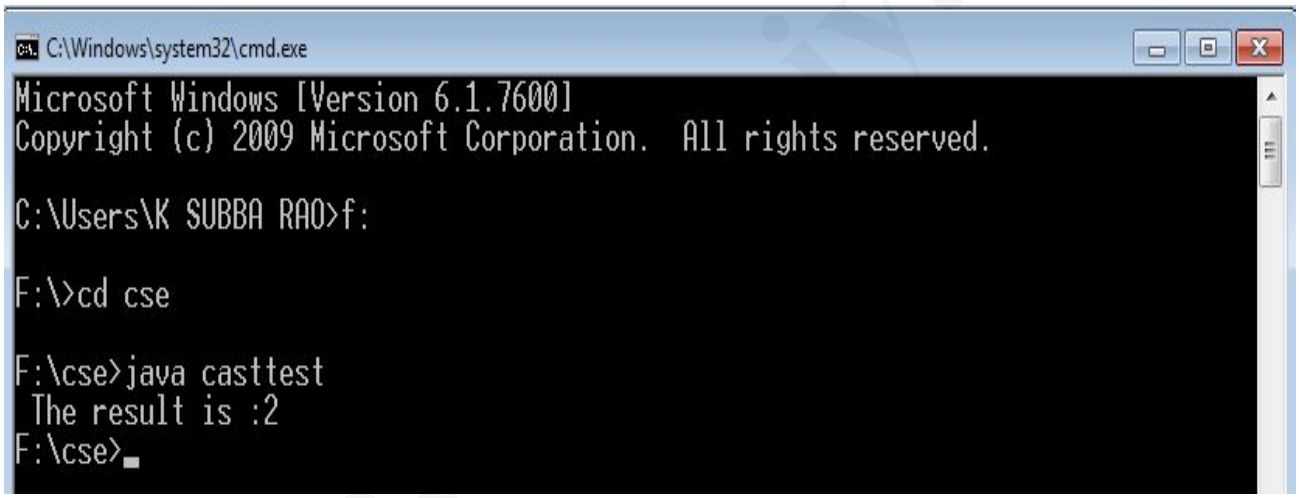
The above code converts the int to byte. If the integer value is larger than the byte, then it will be reduced to modulo byte's range.

casttest.java

```
import java.io.*;
class casttest
```

```
{
        public static void main(String args[])
        {
                int a=258;
                byte b;
                b=(byte) a;

                System.out.print(" The result is :" +b);
        }
}
```

output:



Adifferent type of conversion will occur when a floating-point value is assigned to aninteger type: ***truncation***. As you know, integers do not have fractional components. Thus,when a floating-point value is assigned to an integer type, the fractional component is lost.For example, if the value 1.23 is assigned to an integer, the resulting value will simply be 1.The 0.23 will have been truncated.

Automatic Type Promotion in Expressions

The expression contains the three things: operator, operand and literals (constant). In an expression, sometimes the sub expression value exceeds the either operand.

For example,examine the following expression:

```
byte a = 40;
byte b = 50;
byte c = 100;
int d = a * b / c;
```

The result of the intermediate term **a * b** easily exceeds the range of either of its **byte**operands. To handle this kind of problem, Java automatically promotes each **byte**, **short**,or **char** operand to **int** when evaluating an expression. This means that the subexpression **a * b**is performed using integers—not bytes. Thus, 2,000, the result of the intermediate expression,**50 * 40**, is legal even though **a** and **b** are both specified as type **byte**.

As useful as the automatic promotions are, they can cause confusing compile-time errors.For example, this seemingly correct code causes a problem:

```
byte b = 50;
b = b * 2; // Error! Cannot assign an int to a byte!
```

The code is attempting to store 50 * 2, a perfectly valid **byte** value, back into a **byte** variable.However, because the operands were automatically promoted to **int** when the expression wasevaluated, the result has also been promoted to **int**. Thus, the result of the expression is nowof type **int**, which cannot be assigned to a **byte** without the use of a cast.

In cases where you understand the consequences of overflow, you should use an explicit cast, such as

```
byte b = 50;
b = (byte)(b * 2);
```

which yields the correct value of 100.

The Type Promotion Rules

Java defines several *type promotion* rules that apply to expressions. They are as follows:

- First,all **byte**, **short**, and **char** values are promoted to **int**, as just described.
- Then, if one operandis a **long**, the whole expression is promoted to **long**.
- If one operand is a **float,** the entireexpression is promoted to **float**.
- If any of the operands is **double**, the result is **double**.

Introduction to Arrays

An Array is a collection of elements that share the same type and name. The elements from the array can be accessed by the index. To create an array, we must first create the array variable of the desired type. The general form of the One Dimensional array is as follows:

type var_name[];

Here type declares the base type of the array. This base type determine what type of elements that the array will hold.

Example:

int month_days[];

Here type is int, the variable name is month_days. All the elements in the month are integers. Since, the base type is int.

In fact, the value of **month_days** is set to **null**, which represents an arraywith no value. To link **month_days** with an actual, physical array of integers, you must allocateone 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 appearsas follows:

***array-var* = new *type*[*size*];**

Here, *type* specifies the type of data being allocated, *size* specifies the number of elements inthe array, and *array-var* is the array variable that is linked to the array. That is, to use **new** toallocate an array, you must specify the type and number of elements to allocate. The elementsin the array allocated by **new willautomatically be initialized to zero**. This example allocatesa 12-element array of integers and links them to **month_days**.

month_days = new int[10];

month_days

Element	0	0	0	0	0	0	0	0	0	0
Index	0	1	2	3	4	5	6	7	8	9

After this statement executes, **month_days** will refer to an array of 12 integers. Further, all elements in the array will be initialized to zero.

Once you have allocated an array, you can access a specific element in the array byspecifying its index within square brackets. All array indexes start at zero. For example,this statement assigns the value 28 to the second element of **month_days**.

month_days[1] = 28;

Element	0	**28**	0	0	0	0	0	0	0	0
Index	0	**1**	2	3	4	5	6	7	8	9

The next line displays the value stored at index 3.
System.out.println(month_days[3]);

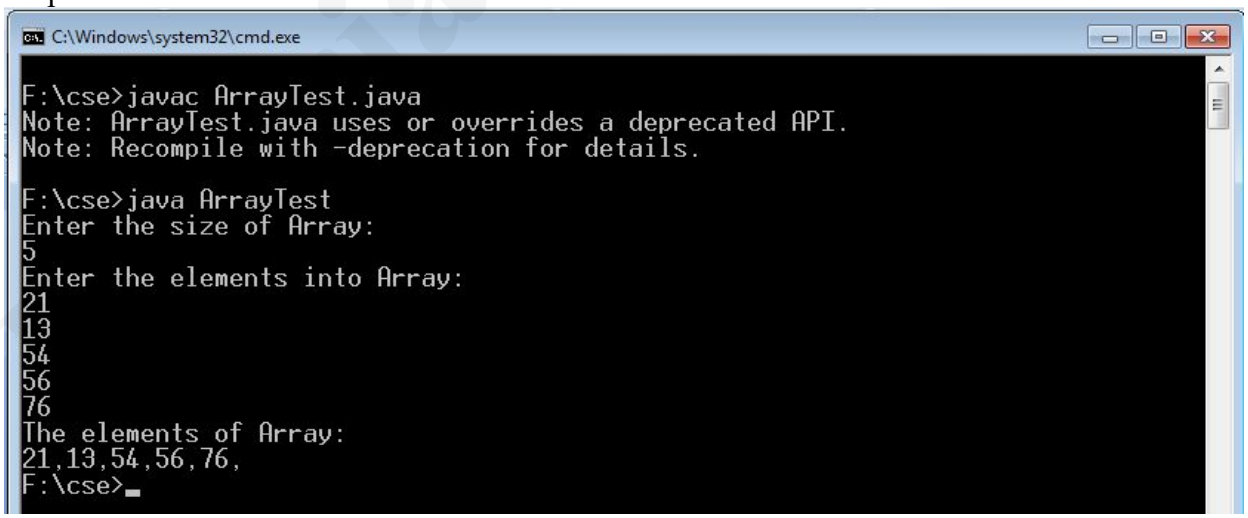
Example Program: Write a Java Program to read elements into array and display them?

ArrayTest.java

```
import java.io.*;
class ArrayTest
```

```
{
        public static void main(String args[]) throws IOException
        {

                DataInputStream dis=new DataInputStream(System.in);
                int a[]; //declaring array variable
                int n, i; //size of the array
                System.out.println("Enter the size of Array:");
                n=Integer.parseInt(dis.readLine());
                a=new int[n]; //allocating memory to array a and all the elements are set zero

                //read the elements into array
                System.out.println("Enter the elements into Array:");
                for(i=0;i<n;i++)
                {
                        a[i]=Integer.parseInt(dis.readLine());
                }
                //displaying the elements
                System.out.println("The elements of Array:");
                for(i=0;i<n;i++)
                {
                        System.out.print(a[i]+",");
                }
        }
}
```

Output



Multidimensional Arrays

In Java, *multidimensional arrays* are actually arrays of arrays. These, as you might expect, lookand act like regular multidimensional arrays. However, as you will see, there are a coupleof

subtle differences. To declare a multidimensional array variable, specify each additionalindex using another set of square brackets. For example, the following declares a twodimensional

array variable called **twoD**.
int twoD[][] = new int[4][4];

This allocates a 4 by 4 array and assigns it to **twoD**. Internally this matrix is implemented as an *array* of *arrays* of **int**.

	Right Index Determines the Columns			
Left index determines the Rows	[0,0]	[0,1]	[0,2]	[0,3]
	[1,0]	[1,1]	[1,2]	[1,3]
	[2,0]	[2,1]	[2,2]	[2,3]
	[3,0]	[3,1]	[3,2]	[3,3]

Example Program for Matrix Addition

```
import java.io.*;
class AddMatrix
{
  public static void main(String args[]) throws IOException
  {
    int m, n, c, d;
    DataInputStream dis=new DataInputStream(System.in);

    System.out.println("Enter the number of rows and columns of matrix");
    m = Integer.parseInt(dis.readLine());
    n  = Integer.parseInt(dis.readLine());

    int first[][] = new int[m][n];
    int second[][] = new int[m][n];
    int sum[][] = new int[m][n];

    System.out.println("Enter the elements of first matrix");

    for (  c = 0 ; c < m ; c++ )
      for ( d = 0 ; d < n ; d++ )
        first[c][d] = Integer.parseInt(dis.readLine());

    System.out.println("Enter the elements of second matrix");

    for ( c = 0 ; c < m ; c++ )
      for ( d = 0 ; d < n ; d++ )
        second[c][d] = Integer.parseInt(dis.readLine());
```

```
    for ( c = 0 ; c < m ; c++ )
      for ( d = 0 ; d < n ; d++ )
        sum[c][d] = first[c][d] + second[c][d];  //replace '+' with '-' to subtract matrices

    System.out.println("Sum of entered matrices:-");

    for ( c = 0 ; c < m ; c++ )
    {
      for ( d = 0 ; d < n ; d++ )
        System.out.print(sum[c][d]+"\t");

      System.out.println();
    }
  }
}
```

Example program for Matrix Multiplication

```
import java.io.*;

class MulMatrix
{
  public static void main(String args[]) throws IOException
  {
    int m, n, p, q, sum = 0, c, d, k;

DataInputStream dis = new DataInputStream(System.in);
    System.out.println("Enter the number of rows and columns of first matrix");
    m = Integer.parseInt(dis.readLine());
    n = Integer.parseInt(dis.readLine());

    int first[][] = new int[m][n];

    System.out.println("Enter the elements of first matrix");

    for ( c = 0 ; c < m ; c++ )
      for ( d = 0 ; d < n ; d++ )
        first[c][d] = Integer.parseInt(dis.readLine());

    System.out.println("Enter the number of rows and columns of second matrix");
    p = Integer.parseInt(dis.readLine());
    q = Integer.parseInt(dis.readLine());

    if ( n != p )
      System.out.println("Matrices with entered orders can't be multiplied with each other.");
    else
```

```
    {
      int second[][] = new int[p][q];
      int multiply[][] = new int[m][q];

      System.out.println("Enter the elements of second matrix");

      for ( c = 0 ; c < p ; c++ )
        for ( d = 0 ; d < q ; d++ )
          second[c][d] = Integer.parseInt(dis.readLine());

      for ( c = 0 ; c < m ; c++ )
      {
        for ( d = 0 ; d < q ; d++ )
        {
          for ( k = 0 ; k < p ; k++ )
          {
            sum = sum + first[c][k]*second[k][d];
          }

          multiply[c][d] = sum;
          sum = 0;
        }
      }

      System.out.println("Product of entered matrices:-");

      for ( c = 0 ; c < m ; c++ )
      {
        for ( d = 0 ; d < q ; d++ )
          System.out.print(multiply[c][d]+"\t");

        System.out.print("\n");
      }
    }
  }
}
```

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 offers convenience when declaring several arrays at the same time. For example,

int[] nums, nums2, nums3; // create three arrays

creates three array variables of type **int**. It is the same as writing

int nums[], nums2[], nums3[]; // create three arrays

The alternative declaration form is also useful when specifying an array as a return type for a method. Both forms are used.

Command Line arguments

The java command-line argument is an argument i.e. passed at the time of running the java program.The arguments passed from the console can be received in the java program and it can be used as an input.

So, it provides a convenient way to check the behavior of the program for the different values. You can pass **N** (1,2,3 and so on) numbers of arguments from the command prompt. The command line arguments can be accessed easily, because they are stored as Strings in String array passed to the args in the main method. The first command line argument is stored in args[0], the second argument is stored in args[1], the third argument is stored in args[2], and so on.

Example program:

Write a Java program to read all the command line arguments?

```
import java.io.*;
class CommnadLine
{
        public static void main(String args[])
        {
```

```
        for(int i=0;i<args.length();i++)
        {
                System.out.println(args[i]+" ");
        }
        }
}
```

Here the String class has a method length(), which is used to find the length of the string. This length can be used to read all the arguments from the command line.

Introduction to Strings

String is probably the most commonly used class in Java'sclass library. The obvious reason for this is that strings are a very important part ofprogramming.

- 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 thestatement

System.out.println("This is a String, too");

the string "This is a String, too" is a **String** constant.

- 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 aserious restriction, it is not, for two reasons:
  - If you need to change a string, you can always create a new one that containsthe modifications.
  - Java defines a peer class of **String**, called **StringBuffer**, which allows strings to be altered, so all of the normal string manipulations are still available in Java

Strings can be created in a many ways. The easiest is to use a statement like this:

1. String initialization
   String myString = "this is a test";
2. Reading from input device
   DataInputStream dis=new DataInputStream(System.in);
   String st=dis.readLine();

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 **String** class contains several methods that you can use.
Here are a few.

1. **equals( ) -** used to test whether two strings are equal or not
2. **length() –**used to find the length of the string
3. **charAt(i) -** used to retrieve the character from the string at the index i.
4. **compareTo(String)** –returns 0, if the string lexicographically equals to the argument, returns greater than 0 if the argument is lexicographically greater than this string, returns less than 0 otherwise.
5. **indexOf(char)** –returns the index of first occurrence of the character.
6. **lastIndesOf(char)**- returns the last index of the character passed to it.
7. **concat(String)** -Concatenates the string with the specified argument

Example :**int n=myString.length(); //gives the length of the string**

Part-II

Introduction to classes

Fundamentals of the class

A class is a group of objects that has common properties. It is a **template** or blueprint from which objects are created. The objects are the **instances** of class. Because an object is an instanceof a class, you will often see the two words *object* and *instance* used interchangeably.

A class is used to define new type of the data. Once defined, this new type can be used to create objects of its type. The class is the logical entity and the object is the logical and physical entity.

The general form of the class

Aclass is declared by use of the **class** keyword. The classes that have been used up to thispoint are actually very limited examples of its complete form. Classes can (and usually do)get much more complex. A simplified general form of a **class** definition is shown here:

class *classname*
{
    type instance-variable1;
    type instance-variable2;
    ……
    ……
    type instance-variableN;

TutorialsDuniya.com

Download FREE Computer Science Notes, Programs, Projects, Books PDF for any university student of BCA, MCA, B.Sc, B.Tech CSE, M.Sc, M.Tech at https://www.tutorialsduniya.com

- Algorithms Notes
- Artificial Intelligence
- Android Programming
- C & C++ Programming
- Combinatorial Optimization
- Computer Graphics
- Computer Networks
- Computer System Architecture
- DBMS & SQL Notes
- Data Analysis & Visualization
- Data Mining
- Data Science
- Data Structures
- Deep Learning
- Digital Image Processing
- Discrete Mathematics
- Information Security
- Internet Technologies
- Java Programming
- JavaScript & jQuery
- Machine Learning
- Microprocessor
- Operating System
- Operational Research
- PHP Notes
- Python Programming
- R Programming
- Software Engineering
- System Programming
- Theory of Computation
- Unix Network Programming
- Web Design & Development

Please Share these Notes with your Friends as well

```
        type method1(parameterlist)
        {
                //body of the method1
        }
        type method2(parameterlist)
        {
                //body of the method2
        }
        ……..
        ……..
        type methodN(parameterlist)
        {
                //body of the methodN
        }
}
```

- The data or variables, defined within the class are called, ***instance variable***.
- The methods also contain the code.
- The methods and instance variable collectively called as ***members***.
- Variable declared within the methods are called ***local variables***.

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
{       //instance variables
        double width;
        double height;
        double depth;
}
```

As stated, a class defines new data type. The new data type in this example is, Box. This defines the template, but does not actually create object.

Creating the Object

There are three steps when creating an object from a class:

- **Declaration:** A variable declaration with a variable name with an object type.
- **Instantiation:** The 'new' key word is used to create the object.
- **Initialization:** The 'new' keyword is followed by a call to a constructor. This call initializes the new object.

Step 1:

Box b;

Effect: b | null |

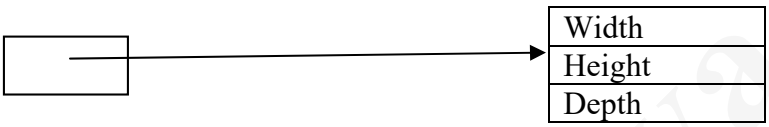
Declares the class variable. Here the class variable contains the value **null**. An attempt to access the object at this point will lead to Compile-Time error.

Step 2:

Box b=new Box();

Here new is the keyword used to create the object. The object name is b. The **new** operator allocates the memory for the object, that means for all instance variables inside the object, memory is allocated.

Effect: b



Box Object

Step 3:

There are many ways to initialize the object. The object contains the instance variable. The variable can be assigned values with reference of the object.

```
b.width=12.34;
b.height=3.4;
b.depth=4.5;
```

Here is a complete program that uses the **Box** class:

BoxDemo.java

```
class Box
{
        double width;
        double height;
        double depth;
}
// This class declares an object of type Box.


class BoxDemo
{
public static void main(String args[])
{
        //declaring the object (Step 1) and instantiating (Step 2) object
        Box mybox = new Box();
        double vol;
        // assign values to mybox's instance variables  (Step 3)
        mybox.width = 10;
        mybox.height = 20;
```

```
        mybox.depth = 15;
        // compute volume of box
        vol = mybox.width * mybox.height * mybox.depth;
        System.out.println("Volume is " + vol);
}
}
```

When you compile this program, youwill find that two **.class** files have been created, one for **Box** and one for **BoxDemo**. The Javacompiler automatically puts each class into its own **.class** file. It is not necessary for both the**Box** and the **BoxDemo** class to actually be in the same source file.

To run this program, you must execute **BoxDemo.class**. When you do, you will see the following output:

Volume is 3000.0

As stated earlier, each object has its own copies of the instance variables. This means thatif you have two **Box** objects, each has its own copy of **depth**, **width**, and **height**. It is importantto understand that changes to the instance variables of one object have no effect on the instancevariables of another. For example, the following program declares two **Box** objects:

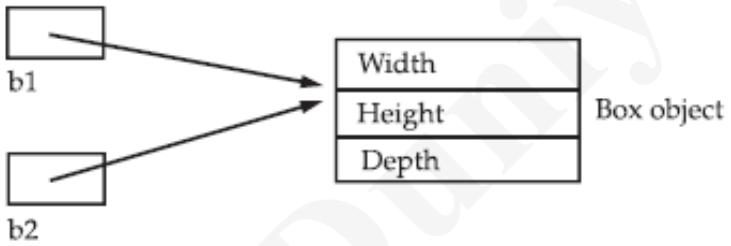
```
// This program declares two Box objects.
class Box
{
       double width;
       double height;
       double depth;
}
class BoxDemo2
{
public static void main(String args[])
{
       Box mybox1 = new Box();
       Box mybox2 = new Box();
       double vol;
// assign values to mybox1's instance variables
       mybox1.width = 10;
       mybox1.height = 20;
       mybox1.depth = 15;
/* assign different values to mybox2's
instance variables */
       mybox2.width = 3;
       mybox2.height = 6;
       mybox2.depth = 9;
// compute volume of first box
       vol = mybox1.width * mybox1.height * mybox1.depth;
       System.out.println("Volume is " + vol);
// compute volume of second box
       vol = mybox2.width * mybox2.height * mybox2.depth;
       System.out.println("Volume is " + vol);
}
}
```

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 theoriginal object. It simply makes **b2** refer to the same object as does **b1**. Thus, any changesmade to the object through **b2** will affect the object to which **b1** is referring, since they are thesame object.
This situation is depicted here:



Introduction to Methods

Classes usually consist of two things: ***instance variables and methods***. The topic of methods is a large one because Java gives them so much power and flexibility. In fact, much of the next chapter is devoted to methods.

This is the general form of a method:

```
type name(parameter-list)
{
        // body of method
}
```

Here, *type* specifies the type of data returned by the method. This can be any valid type,including class types that you create. If the method does not return a value, its return typemust be **void**. The name of the method is specified by *name.* This can be any legal identifierother than those already used by other items within the current scope. The *parameter-list* is asequence of type and identifier pairs separated by commas. Parameters are essentially variablesthat receive the value of the ***arguments***passed to the method when it is called. If the methodhas no parameters, then the parameter list will be empty.

Methods that have a return type other than **void** return a value to the calling routine usingthe following form of the **return** statement:

return *value*;

Here, *value* is the value returned.

Adding a method to the Box class

Box.java

```
class Box
{
      double width, height, double depth;
// display volume of a box
void volume()
      {
      System.out.print("Volume is ");
      System.out.println(width * height * depth);
      }
}
```

Here the method name is "volume()". This methods contains some code fragment for computing the volume and displaying. This method cane be accessed using the object as in the following code:

BoxDemo3.java

```
class BoxDemo3
{
public static void main(String args[])
{
Box mybox1 = new Box();
// assign values to mybox1's instance variables
mybox1.width = 10;
mybox1.height = 20;
mybox1.depth = 15;
/* assign different values to mybox2's

// display volume of first box
mybox1.volume();
// display volume of second box
}
}
```

Returning a Value

A method can also return the value of specified type. In this case the type of the method should be clearly mentioned. The method after computing the task returns the value to the **caller** of the method.

BoxDemo3.java

```
Box
{
      double width, height, depth;

double volume()
      {
            return (width*height*depth);
```

```
        }
}

class BoxDemo3
{
public static void main(String args[])
{
        Box mybox1 = new Box();
        // assign values to mybox1's instance variables
        mybox1.width = 10;
        mybox1.height = 20;
        mybox1.depth = 15;
        double vol;
/* assign different values to mybox2's
//calling the method
vol= mybox1.volume();
System.out.println("the Volume is:"+vol);
}
}
```

Adding a method that takes the parameters

We can also pass arguments to the method through the object. The parameters separated with comma operator. The values of the actual parameters are copied to the formal parameters in the method. The computation is carried with formal arguments, the result is returned to the caller of the method, if the type is mentioned.

```
double volume(double w,double h,double d)
        {
                width=w;
                height=h;
                depth=d;
                return (width*height*depth);
        }
```

Constructors

It can be tedious to initialize all of the variables in a class each time an instance is created. Even if we use some method to initialize the variable, it would be better this initialization is done at the time of the object creation.

A***constructor***initializes an object immediately upon creation. It has the same name as theclass in which it resides and is syntactically similar to a method. Once defined, the constructoris 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 isbecause the implicit return type of a class' constructor is the class type itself. It is the constructor'sjob to initialize the internal state of an object so that the code creating an instance will havea fully initialized, usable object immediately.

Example Program:

```
class Box
{
      double width;
      double height;
      double depth;
// This is the constructor for Box.
      Box()
      {
            System.out.println("Constructing Box");
            width = 10;
            height = 10;
            depth = 10;
      }
// compute and return volume
      double volume()
      {
            return width * height * depth;
      }
}
class BoxDemo6 {
public static void main(String args[]) {
// declare, allocate, and initialize Box objects
Box mybox1 = new Box();
double vol;
// get volume of first box
vol = mybox1.volume();
System.out.println("Volume is " + vol);
// get volume of second box
}}
```

Parameterized Constructors

While the **Box( )** constructor in the preceding example does initialize a **Box** object, it is notvery 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.

```
/* Here, Box uses a parameterized constructor to
initialize the dimensions of a box.
*/
class Box
{
      double width;
      double height;
      double depth;
// This is the constructor for Box.
Box(double w, double h, double d)
      {
      width = w;
```

```
        height = h;
        depth = d;
        }
// compute and return volume
double volume() {
return width * height * depth;
}
}

class BoxDemo7 {
public static void main(String args[]) {
// declare, allocate, and initialize Box objects
Box mybox1 = new Box(10, 20, 15);
Box mybox2 = new Box(3, 6, 9);
double vol;
// get volume of first box
vol = mybox1.volume();
System.out.println("Volume is " + vol);
// get volume of second box
vol = mybox2.volume();
System.out.println("Volume is " + vol);
}
}
```

Constructor-overloading

In Java it is possible to define two or more class constructors that share the same name, as long as their parameter declarations are different. This is called constructor overloading.

When an overloaded constructor is invoked, Java uses the type and/or number ofarguments as its guide to determine which version of the overloaded constructor to actuallycall. Thus, overloaded constructors must differ in the type and/or number of their parameters.

Example: All the constructors names will be same, but their parameter list is different.

OverloadCons.java

```
class Box {
double width;
double height;
double 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;
}
}

class OverloadCons
{
public static void main(String args[])
{
// create boxes using the various constructors
Box mybox1 = new Box(10, 20, 15);
Box mybox2 = new Box();
Box mycube = new Box(7);
double vol;
// get volume of first box
vol = mybox1.volume();
System.out.println("Volume of mybox1 is " + vol);
// get volume of second box
vol = mybox2.volume();
System.out.println("Volume of mybox2 is " + vol);
// get volume of cube
vol = mycube.volume();
System.out.println("Volume of mycube is " + vol);
}
}
```

The output produced by this program is shown here:

```
Volume of mybox1 is 3000.0
Volume of mybox2 is -1.0
Volume of mycube is 343.0
```

Garbage Collection

Since objects are dynamically allocated by using the **new** operator, you might be wonderinghow such objects are destroyed and their memory released for later reallocation. In somelanguages, such as C++, dynamically allocated objects must be manually released by use ofa **delete** operator. Java takes a different approach; it handles deallocation for you automatically.The technique that accomplishes this is called ***garbage collection***.Java has its own set of algorithms to do this as follow.

There are Two Techniques: **1) Reference Counter 2) Mark and Sweep**. In the Reference Counter technique, when an object is created along with it a reference counter is maintained in the memory. When the object is referenced, the reference counter is incremented by one. If the control flow is moved from that object to some other object, then the counter value is decremented by one. When the counter reaches to zero (0), then it's memory is reclaimed.

In the Mark and Sweep technique, all the objects that are in use are ***marked*** and are called live objects and are moved to one end of the memory. This process we call it as compaction. The memory occupied by remaining objects is reclaimed. After these objects are deleted from the memory, the live objects are placed in side by side location in the memory. This is called copying.

It works like this: when noreferences to an object exist, that object is assumed to be no longer needed, and the memoryoccupied by the object can be reclaimed. There is no explicit need to destroy objects as in C++.Garbage collection only occurs during the execution of your program. The main job of this is to release memory for the purpose of reallocation. Furthermore,different Java run-time implementations will take varying approaches to garbage collection

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 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 candefine specific actions that will occur when an object is just about to be reclaimed by thegarbage collector.

To add a finalizer to a class, you simply define the **finalize( )** method. The Java run time 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 byany running state or indirectly through other referenced objects

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.

Overloading methods

In Java it is possible to define **two or more methods** within the same class that share thesame name, as long as their parameter declarations are different. When this is the case, themethods are said to be *overloaded,* and the process is referred to as ***method overloading***. Methodoverloading is one of the ways that Java supports **polymorphism**.

When an overloaded method is invoked, Java uses the type and/or number ofarguments as its guide to determine which **version** of the overloaded method to actuallycall. Thus, overloaded methods must differ in the type and/or number of their parameters.While overloaded methods may have different return types, the return type alone isinsufficient to distinguish two versions of a method. When Java encounters a call to anoverloaded method, it simply executes the version of the method whose parameters matchthe arguments used in the call.

Here is a simple example that illustrates method overloading:

```
// Demonstrate method overloading.

class OverloadDemo
{
        void test()
        {
                System.out.println("No parameters");
        }
// Overload test for one integer parameter.
        void test(int a)
        {
                System.out.println("a: " + a);
        }
}

class Overload
{
public static void main(String args[])
{
        OverloadDemo ob = new OverloadDemo();
        double result;
// call all versions of test()
        ob.test();
        ob.test(10);
}
}
```

The test() method is overloaded two times, first version takes no arguments, second version takes one argument. When an overloaded method is invoked, Java looks for a match between arguments of the methods.Method overloading supports **polymorphism** because it is one way that Java implementsthe “one interface, multiple methods” paradigm.

static keyword

There will be times when you will want to define a class member that will be usedindependently of any object of that class. Normally, a class member must be accessed onlyin conjunction with an object of its class. However, it is possible to create a member that canbe used by itself, without reference to a specific instance.

To create such a member, precedeits declaration with the keyword **static**. When a member is declared **static**, it can be accessedbefore any objects of its class are created, and without reference to any object. You can declareboth methods and variables to be **static**. The most common example of a **static** member is**main( )**. **main( )** is declared as **static** because it must be called before any objects exist.

Instance variables declared as **static** are, essentially, global variables. When objects of its class are declared, no copy of a **static** variable is made. Instead, all instances of the classshare the same **static** variable.These variables are also called as "***Class Variable***".

Methods declared as **static** have several restrictions:

- They can only call other **static** methods.
- They must only access **static** data.
- They cannot refer to **this** or **super** in any way.

If you need to do computation in order to initialize your **static** variables, you can declare a**static** block that gets executed exactly once, when the class is first loaded. The followingexample shows a class that has a **static** method, some **static** variables, and a **static** initializationblock:

```
class UseStatic
        {
                //static variable
                static int a = 3;
                static int b;
                //static method
                static void meth(int x)
                 {
                System.out.println("x = " + x);
                System.out.println("a = " + a);
                System.out.println("b = " + b);
                }
                //static block
                static
                {
                System.out.println("Static block initialized.");
                b = a * 4;
                }
public static void main(String args[])
{
        //static methods are called without object
        meth(42);
}
}
```

Note: Static blocks are executed first, even than the static methods.

static variables **a** and **b**, as well as to the local variable **x**.Here is the output of the program:

```
Static block initialized.

x = 42
```

a = 3
b = 12

The this Keyword

Sometimes a method will need to refer to the object that invoked it. To allow this, Java definesthe **this** keyword. **this** can be used inside any method to refer to the ***current object***. That is,**this** 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 = w;
                this.height = h;
                this.depth = d;
        }
```

Note: This is mainly used to hide the local variables from the instance variable.

Example:

```
class  Box
{
        //instance variable
        double width, height, depth;
        Box(double width, double height, double depth)
        {
                //local variables are assigned, but not the instance variable
                width=width;
                height=height;
                depth=depth;
        }
}
```

To avoid the confusion, this keyword is used to refer to the instance variables, as follows:

```
class  Box
{
        //instance variable
        double width, height, depth;
        Box(double width, double height, double depth)
        {
                //the instance variable are assigned through the this keyword.
                this.width=width;
                this.height=height;
                this.depth=depth;
        }
}
```

********** End of the 2nd Unit**********

Unit-3

Topics:

Inheritance –Types of Inheritance, using the extends keyword, overriding method, super keyword, final keyword, abstract class.
Interfaces, Packages: Interface, extending interface, Interface Vs Abstract class, Creating package, using the package, access protection.
Exceptions and Assertions: Exception Handling techniques, User defined Exception, exception encapsulation and enrichment, Assertions.

Inheritance Vs Aggregation

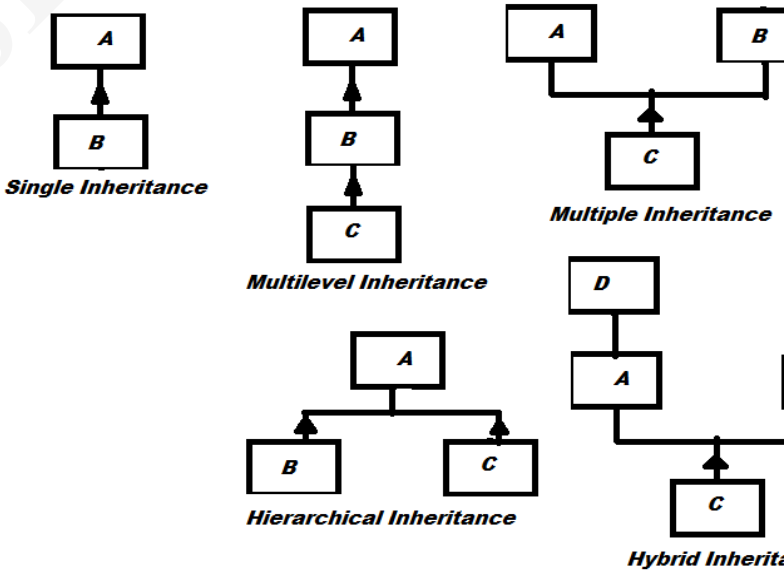
Inheritanceis the process by which one class acquires the properties of another class. Thisis important because it supports the concept of hierarchical classification. The class that is inherited is called ***superclass***. The class that is inheriting the properties is called ***subclass***. Therefore the subclass is the specialized version of the superclass. The subclass inherits all the instance variable and methods, and adds its own code.

Aggregation is the process of making an object combining number of other objects. The behavior of the bigger object is defined by the behavior of its component objects. For example, cars contain number of other components such as engine, clutches, breaks, starter etc.

Types of Inheritance

There are five different types of Inheritances:

i. Single Inheritance
ii. Multiple Inheritance
iii. Multilevel Inheritance
iv. Hierarchical Inheritance
v. Hybrid Inheritance



Single Inheritance:
In this One class Acquires the properties from another object and adds its own code to it.
Multiple Inheritance:
In this one class acquires the properties from two or more classes at a time and adds its own code to it.
Multilevel Inheritance:
In this one class acquires the properties from another class, which in turn has acquired the properties from another class. Hence, in this there are many levels in the process of Inheritance.
Hierarchical Inheritance:
In this two or more classes will acquire the properties from only one same class and add their own code.

TutorialsDuniya.com

Download FREE Computer Science Notes, Programs, Projects, Books PDF for any university student of BCA, MCA, B.Sc, B.Tech CSE, M.Sc, M.Tech at https://www.tutorialsduniya.com

- Algorithms Notes
- Artificial Intelligence
- Android Programming
- C & C++ Programming
- Combinatorial Optimization
- Computer Graphics
- Computer Networks
- Computer System Architecture
- DBMS & SQL Notes
- Data Analysis & Visualization
- Data Mining
- Data Science
- Data Structures
- Deep Learning
- Digital Image Processing
- Discrete Mathematics
- Information Security
- Internet Technologies
- Java Programming
- JavaScript & jQuery
- Machine Learning
- Microprocessor
- Operating System
- Operational Research
- PHP Notes
- Python Programming
- R Programming
- Software Engineering
- System Programming
- Theory of Computation
- Unix Network Programming
- Web Design & Development

Please Share these Notes with your Friends as well

Note: Java Supports Single and Multilevel Inheritances between classes and Multiple Inheritance among the Interfaces.

Deriving Classes Using "extends" Keyword

In Java, Properties from one class to another class are acquired using the keyword "***extends***".

The general for of inheritance will be as follow:

```
class subclass_name extends superclass_name
{
        //body of the subclass
}
```

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. `SimpleInheritance.java`

```
// 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("k: " + k);
}
void sum()
        {
        System.out.println("i+j+k: " + (i+j+k));
        }
}
class SimpleInheritance
{
public static void main(String args[])
{       // subclass object creation, which acquires all the properties of the superclass
        B b = new B();
/* The subclass has access to all public members of its superclass. */
        b.i = 7;b.j = 8;b.k = 9;
```

```
        System.out.println("Sum of i, j and k in b:");
        b.sum();
}
}
```

Overriding Method

When a method in the subclass has the same name and type signature as a method in its super class, the method in the sub class is said to "override" the method in the superclass. When a overridden method is called by the subclass it always refers to the method of the subclass. The super class version of the method is hidden.

Example program

OverrideTest.java

```
import java.io.*;
class A
{
        int i;
        void dooverride(int x)
        {
                i=x;
                System.out.println("Value of i is:"+i);
        }
}
class B extends A
{
        void dooverride(int k)
        {
                i=2*k;
                System.out.println("Value of i is:"+i);
        }
}
class OverrideTest
{       public static void main(String args[])
        {
                B b=new B();
                b.dooverride(12);
        }
}
```

Output:

```
E:\ksr>javac OverrideTest.java

E:\ksr>java  OverrideTest
Value of i is:24

E:\ksr>
```

private member access and inheritance

Although a subclass includes all of the members of its **superclass**, it cannot access thosemembers of the **superclass** that have been declared as **private**. The private members of the class are accessed by the members of that class only, but are not accessed by the code outside that class in which they are declared.For example, consider thefollowing simple class hierarchy:

/* In a class hierarchy, private members remainprivate to their class.This program contains an error and will notcompile.*/

```
// Create a superclass.
class A
{
        int i; // public by default
        private int j; // private to A
        void setij(int x, int y)
        {
                i = x;
                j = y;
        }
}
// A's j is not accessible here.
class B extends A
 {      int total;
        void sum()
        {
                total = i + j; // ERROR, j is not accessible here
        }
}
class Access
{
public static void main(String args[])
{
        B b = new B();   // creating the object b
        b.setij(10, 12);
        b.sum();   // Error, private members are not accessed
        System.out.println("Total is " + subOb.total);
}
}
```

Note:
This program will not compile because the reference to **j** inside the **sum( )** method of **B**causes an access violation. Since **j** is declared as **private**, it is only accessible by other membersof its own class. Subclasses have no access to it.

A Superclass Reference Variable Can be assigned a Subclass Object

Areference variable of a superclass can be assigned a reference to any subclass derived fromthat superclass. You will find this aspect of inheritance quite useful in a variety of situations.when a reference to a subclass object is assigned to a superclass reference variable, you will have accessonly to those parts of the object defined by the superclass.
For example,

```
class Box
{
        double width,height,depth;
}
class Boxweight extends Box
{
        double weight;
        Boxweight(double x,double y,double z,double z)
        {
                width=x; height=y; depth=z; weight=a;
        }
        void volume()
        {
                System.out.println("The volume is :"+(width*height*depth));
        }
}
//main class
class BoxDemo
{
        //creating superclass object
        public static void main(String args[])
        {
        Box b=new Box();
        //creating the subclass object
        Boxweight bw=new Boxweight(2,3,4,5);
        bw.volume();
        //assigning the subclass object to the superclass object
        b=bw;    // b has been created with its own data, in which weight is not a member
        System.out.println ("The weight is :"+b.weight);
        // here it raises error, super class does not contain the volume() method
        }
}
```

"*super*" keyword

There are three uses of the ***super*** keyword.

i. super keyword is used to call the superclass ***constructor***
ii. super keyword is used to access the super class ***methods***
iii. super keyword is used to access the super class ***instance variables***.

super(arg_list);

Here, the arg_list, is the list of the arguments in the superclass constructor. This must be the first statement inside the subclass constructor. For example,

```
// BoxWeight now uses super to initialize its Box attributes.
class Box
{
        double width,height,depth;
        //superclass constructor
        Box(double x,double y,double z)
        {
                width=x;height=y;depth=z;
        }
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;
        } }
```

Using the super to access the super class members (methods or instance variable)

The second form of **super** acts somewhat like **this**, except that it always refers to the superclassof 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 ofa subclass hide members by the same name in the superclass. Consider this simple classhierarchy:

```
// 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
```

```
                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();
}
}
```

"final" keyword

The "***final***" keyword is used for the following purposes.

- to declare constants
- to prevent method overriding
- to prevent inheritance.

When a variable is declared as final through the program its value should not be changed by the program statement. If any modification is done on the final variable, that can lead to error, while compiling the program.

Example program: demonstrating (1) and (2)

FinalTest.java

```
import java.io.*;
class A
{
        final int MAX=100;   // (1) constant declaration
final   void disp()   //(2) prevents overriding
        {
                // MAX++ or MAX-- operations are illegal
        System.out.println("The super class disp method MAX is:"+MAX);

        }
}
class B extends A
{
        void disp()
        {
        System.out.println("The SUB class disp method :");
```

```
{
        public static void main(String args[])
        {
                B b=new B();
                b.disp();
        }
}
```

Output:

```
E:\ksr>javac FinalTest.java
FinalTest.java:14: disp() in B cannot override disp() in A; overridden method is
 final
      void disp()
           ^
1 error
```

Explanation:

In the above program, the super class method is declared as final, and hence the sup class can not override this. So we should not redefine the same method in the sub class. If we do so, it leads to an error. If we do the same program with out disp() method in the sub class, it will produce the following out put.

Example Program: removing the disp() method in the sub class

```
import java.io.*;
class A
{
        final int MAX=100;   // (1) constant declaration
final   void disp()   //(2) prevents oerriding
        {
                // MAX++ or MAX-- operations are illegal
        System.out.println("The super class disp method MAX is:"+MAX);

        }
}
class B extends A
{       /*void disp()    multiple comments
        {
        System.out.println("The SUB class disp method :");
        }*/
}
class FinalTest
{       public static void main(String args[])
        {       B b=new B();
                b.disp();
        }
}
```

Output:

```
E:\ksr>javac FinalTest.java
```

Example program: Demonstrating "final" to Prevent Inheritance

Sometimes you will want to prevent a class from being inherited. To do this, precede theclass declaration with **final**. Declaring a class as **final** implicitly declares all of its methodsas **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 providecomplete 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**.

Using Abstract Classes

Sometimes it may be need by the superclass to define the structure of the every method without implementing it. The subclass can fill or implement the method according to its requirements. This kind of situation can come into picture whenever the superclass unable to implement the meaningful implementation of the method. For example, if we want to find the area of the Figure given, which can be Circle, Rectangle, and Traingle. The **class Figure**defines the method area(), when subclass implements its code, it implements its own version of the method. The Java's solution to this problem is **abstract method.**

To declare anabstract method, use this general form:

```
abstract type name(parameter-list);
```

As you can see, no method body is present.

To declare a class abstract, you simply use the **abstract** keyword in front of the **class** keywordat 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 wouldbe useless, because an abstract class is not fully defined.

Example Program:

FigureDemo.java

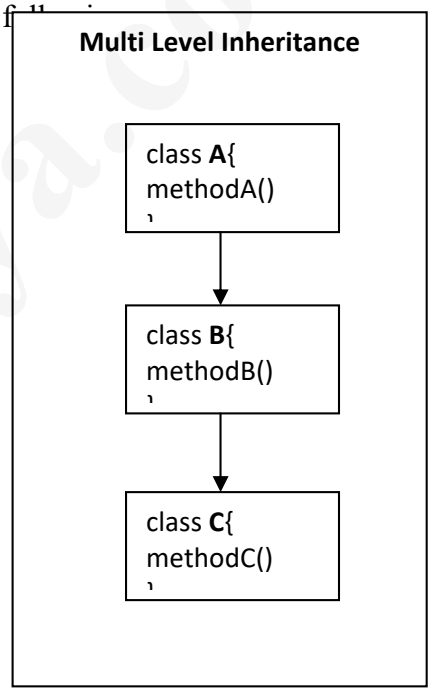
```
        abstract void area();
}
//extending the super class
class Circle extends Figure
{       double r;
        Circle(double x)
        {
                r=x;
        }
        //CIrcle implementing its own method
        void area()
        {
                a=3.14*r*r;     //a is the meber of super class
                System.out.println("The area of the traingle is :"+a);
        }
}
//Extending the super class
class Triangle extends Figure
{       double l,b;
        Triangle(double x,double y)
        {
                l=x;
                b=y;
        }
        //Triangle implementing its own method
        void area()
        {
                a=l*b;  //a is the meber of super class
                System.out.println("The area of the traingle is :"+a);
        }
}
class FigureDemo
{
        public static void main(String args[])
        {
        Circle c=new Circle(4.5);
                c.area();
        Triangle t=new Triangle(2,3);
                t.area();
        }
}
```

Output:

```
E:\ksr>javac FigureDemo.java
E:\ksr>java  FigureDemo
```

Creating a Multilevel Hierarchy

Up to this point, we have been using simple class hierarchies that consist of only a superclassand a subclass. However, you can build hierarchies that contain as many layers of inheritanceas you like. As mentioned, it is perfectly acceptable to use a subclass as a superclass of another.For example, given three classes called **A**, **B**, and **C**, **C** can be a subclass of **B**, which is asubclass of **A**. When this type of situation occurs, each subclass inherits all of the traitsfound in all of its superclasses. In this case, **C** inherits all aspects of **B** and **A**. To see howa multilevel hierarchy can be useful, consider the f



Example program

```
class A
{
        void methodA()
        {
                System.out.println("This is the method of A");
        }
}
//class b extends the super class A
class B extends A
{
        void methodB()
        {
                System.out.println("This is the method of B");
        }
}

//class C extends the super class B
class C extends B
{
        void methodC()
        {
                System.out.println("This is the method of C");
        }
}
class Hierachy
{
        public static void main(String args[])
        {
                C cobj=new C();
                c.methodA();
                c.methodB();
                c.methodC();
        }
}
```

Dynamic Method Dispatch

Dynamic method dispatch is the mechanismby which a call to an overridden method 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 superclass reference variable can referto a subclass object. Java uses this fact to resolve calls to overridden methods at run time. Hereis how. When an overridden method is called through a superclass reference, Java determineswhich version of that method to execute based upon the type of the object being referredto at the time the call occurs. Thus, this determination is made at run time.

Here is an example that illustrates dynamic method dispatch:

```
// Dynamic Method Dispatch or run-time polymorphism
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()
void callme()
        {
                System.out.println("Inside C's callme method");
        }
}
class Dispatch
{
        public static void main(String args[])
        {
                A a = new A(); // object of type A
                B b = new B(); // object of type B
                C c = new C(); // object of type C
                A r; // obtain a reference of type A
                r = a; // r refers to an A object
```

```
            r = c; // r refers to a C object
            r.callme(); // calls C's version of callme
        }
}
```

The output from the program is shown here:

```
Inside A's callme method
Inside B's callme method
Inside C's callme method
```

Packages and Interfaces

Introduction to Packages

One of the main features of the Object Oriented Programming language is the ability to ***reuse the code*** that is already created. One way for achieving this is by extending the classes and implementing the interfaces. Java provides a mechanism to partition the classes into smaller ***chunks***. This mechanism is the ***Package***. The ***Package is container of classes***. The class is the container of the data and code. The package also addresses the problem of name ***space collision*** that occurs when we use same name for multiple classes. Java provides convenient way to keep the same name for classes as long as their subdirectories are different.

Benefits of packages:

1. The classes in the packages can be easily reused.
2. In the packages, classes are unique compared with other classes in the other packages.
3. Packages provide a way to hide classes thus prevent classes from accessing by other programs.
4. Packages also provide way to separate "design" from "coding".

Categories of Packages

The Java Packages are categorized into two categories (i) Java API Packages (ii) User Defined Packages.

1. **Java API Packages** –Java API provides large number of classes grouped into different packages according to the functionality. Most of the time we use the package available with Java API.

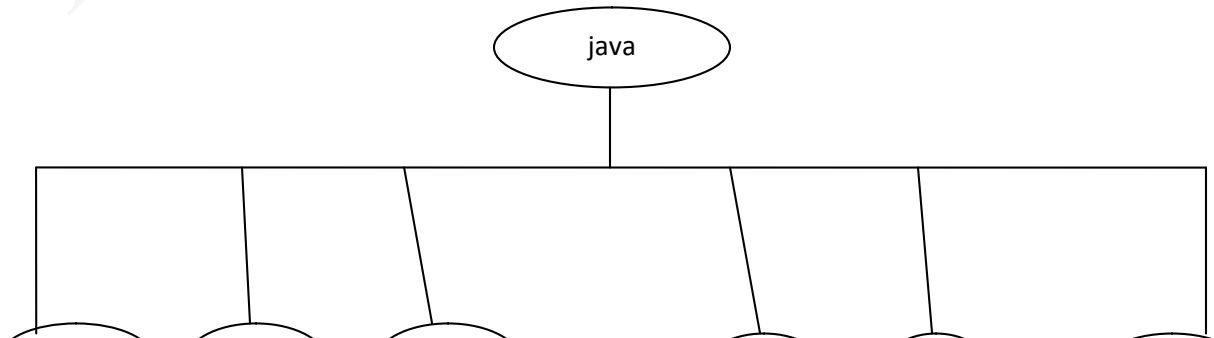


Fig 1 Some of the Java API packages

Table 1 Java System Packages and their Classes

Sl No	Package Name	Contents
1	java.lang	Contains language support classes. The java compiler automatically uses this package. This includes the classes such as primitive data types, String, StringBuffer, StringBuilde etc;
2	java.util	Contains the language utility classes such asa Vectors, Hash Table , Date, StringTokenizer etc;
3	java.io	Contains the classes that support input and output classes.
4	java.awt	Contains the classes for implementing the graphical user interfaces
5	Java.net	Contains the classes for networking
6	Java.applet	Contains the classes for creating and implementing the applets.

Using the System Packages

Packages are organized in hierarchical structure, that means a package can contain another package, which in turn contains several classes for performing different tasks. There are two ways to access the classes of the packages.

i. **fully qualified class name**- this is done specifying package name containing the class and appending the class to it with dot (.) operator.
Example: java.util.StringTokenizer(); Here, "java.util" is the package and "StringTokenizer()" is the class in that package. This is used when we want to refer to only one class in a package.

ii. **import** statement –this is used when we want to use a class or many classes in many places in our program. **Example: (1) import java.util.*;**
(2) import java.util.StringTokenizer;

Naming Conventions

Packages can be named using the Java Standard naming rules. Packages begin with "**lower case**" letters. It is easy for the user to distinguish it from the class names. All the class Name by convention begin with "**upper case**" letters. Example:

double d= java.lang.Math.sqrt(3);

Here, "java.lang" is the package, and "Math" is the class name, and "sqrt()" is the method name.

2. User Define Packages

To create a package is quite easy: simply include a **package** command in the first line of the source file. A class specified in that file belongs to that package. The package statement defines a name space in which classes are stored. If you omit the package statement, the class names are put in the default package, which has no name.

The general form of the package statement will be as followed: **package pkg;**
Here, "pkg" is the package name. Example: **package MyPackage;**

Java Uses file system directories to store packages. For example, the ".**class**" files for any

2. The directory name must match with package name.

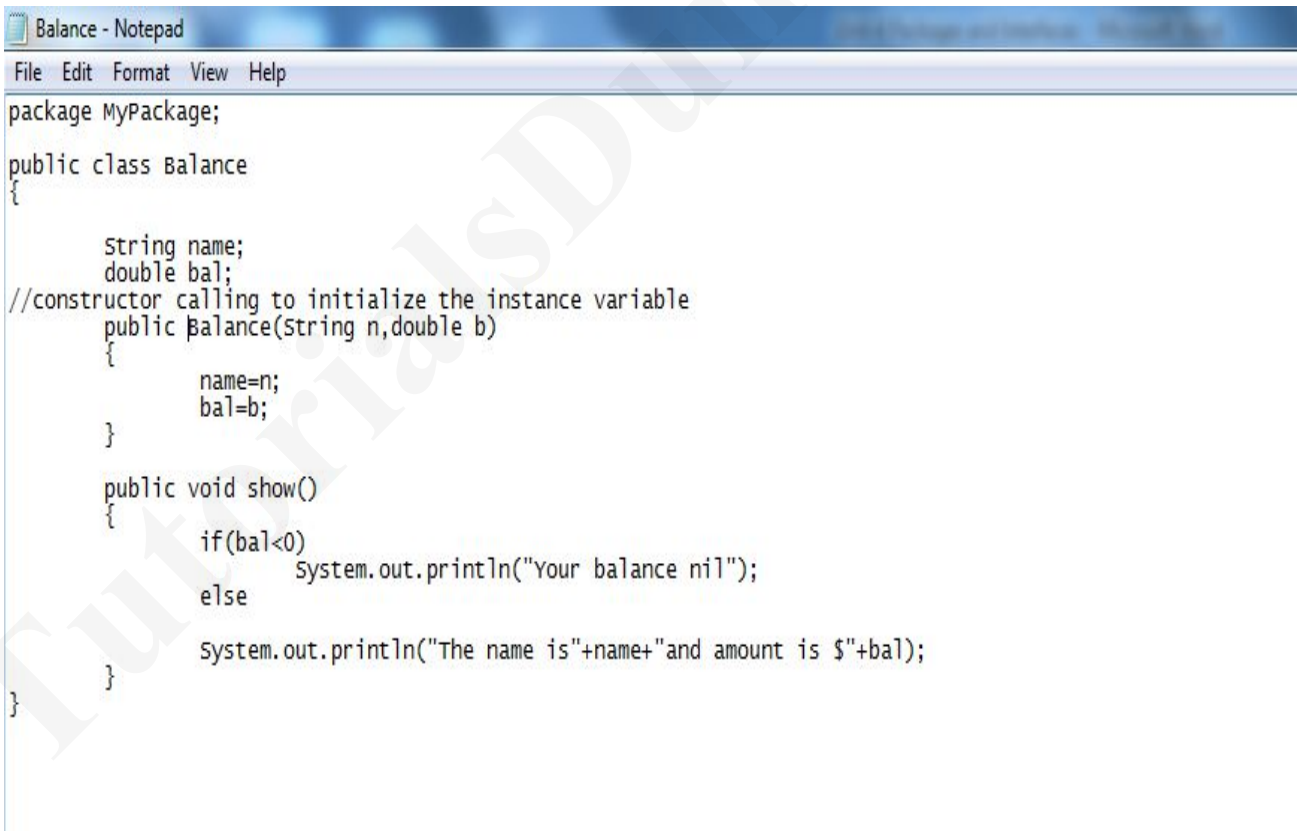
We can create hierarchy of packages. To do so, simply separate package name from the above one with it by use of the dot (.) operator. The general form of multileveled package statement is shown here: **package pkg1.pkg2.pkg3;**

Example: package iicse.asection.java

Finding the Packages and CLASSPATH

Packages are mirrored by the directories. This raises an import question: how does Java-Run time system look for the packages that you create? The answer has three parts: (1) By default, Java-Run time system uses the current working directory as its starting point. Thus your package is in a subdirectory of your directory, it will be found. (2) You can specify a directory path by setting the CLASSPATH environment variable. (3) You can use - **classpath** option with **java** and **iavac**to specify the path for the classes.

A Short Example Package: **Balance.java**



Open a notepad or text editor, write the above code and save it as "Balance.java" in the folder name "Mypackage. Here, the package name and subdirectory(folder) name must be same. This subdirectory(Folder) must be part of your main current directory (current Folder) in which you write the main class PackTest1.java. That means the "**PackTest1.java**" source file and "**MyPackage**" will be in the same directory (Folder). The "Balance.java" file must be compiled to generate the "**.class**" file in the same directory(folder), which will be accessed in another program by importing

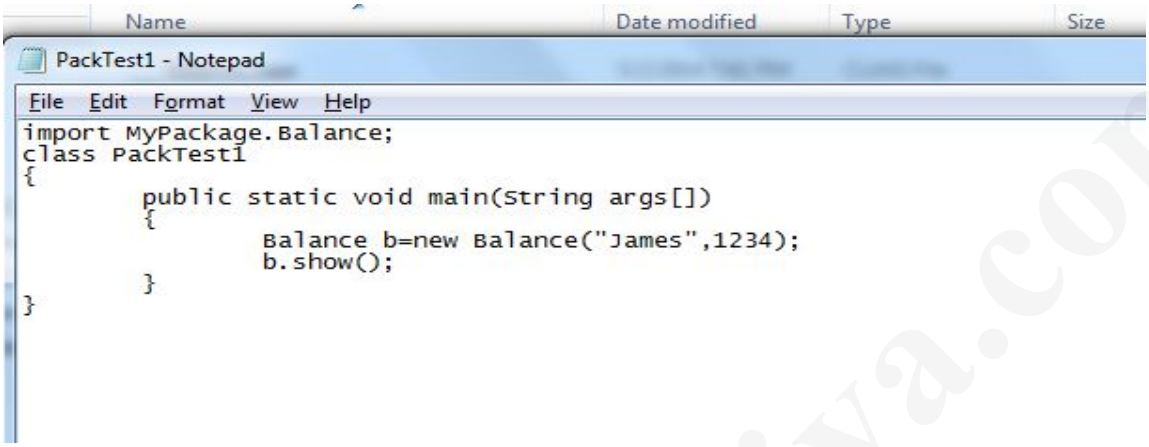
TutorialsDuniya.com

Download FREE Computer Science Notes, Programs, Projects, Books PDF for any university student of BCA, MCA, B.Sc, B.Tech CSE, M.Sc, M.Tech at https://www.tutorialsduniya.com

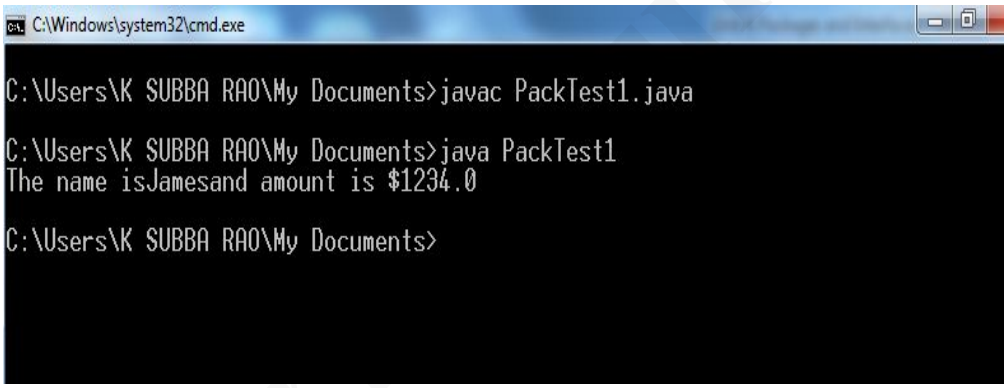
- Algorithms Notes
- Artificial Intelligence
- Android Programming
- C & C++ Programming
- Combinatorial Optimization
- Computer Graphics
- Computer Networks
- Computer System Architecture
- DBMS & SQL Notes
- Data Analysis & Visualization
- Data Mining
- Data Science
- Data Structures
- Deep Learning
- Digital Image Processing
- Discrete Mathematics
- Information Security
- Internet Technologies
- Java Programming
- JavaScript & jQuery
- Machine Learning
- Microprocessor
- Operating System
- Operational Research
- PHP Notes
- Python Programming
- R Programming
- Software Engineering
- System Programming
- Theory of Computation
- Unix Network Programming
- Web Design & Development

Please Share these Notes with your Friends as well

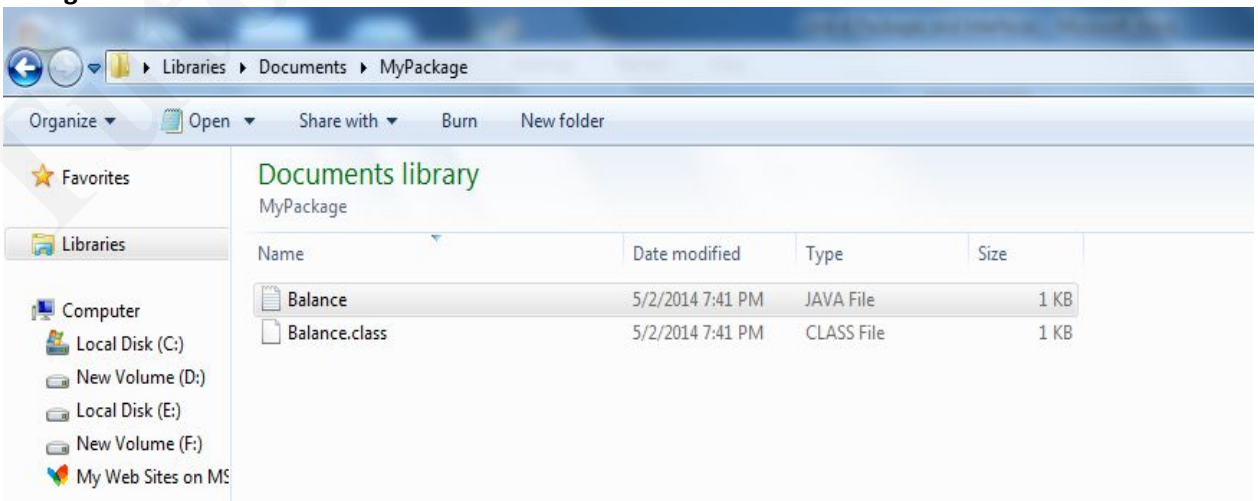
PackTest1.java



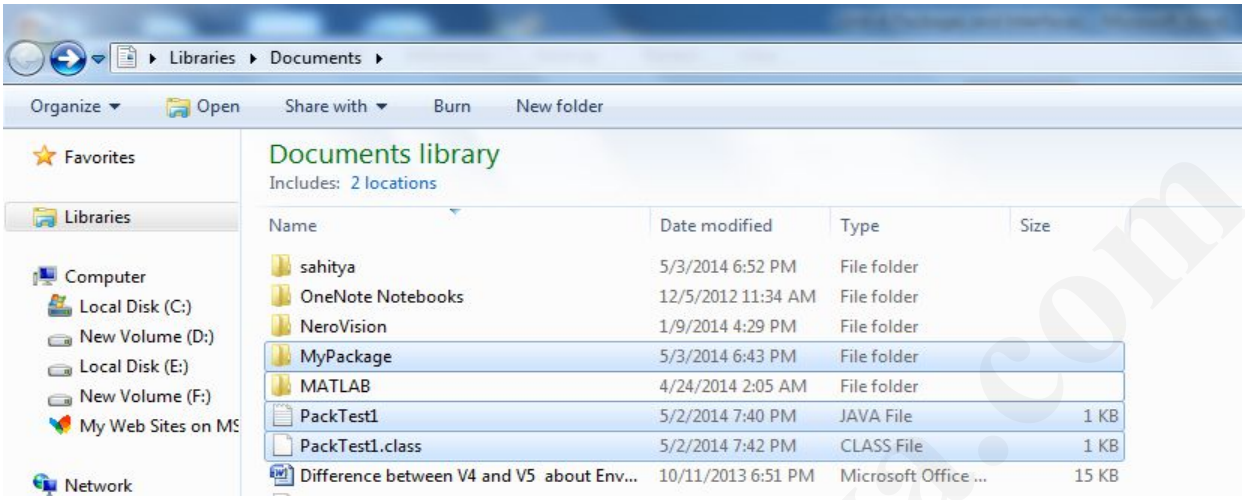
Output



Saving Procedure will be as follows:



The Subdirectory "MyPackage" and the "PackTest 1"are in the same folder



Package and Member Accessing

The visibility of an element is specified by the access specifiers: public, private, and protected and also the package in which it resides. The visibility of an element is determined by its visibility within class, and visibility within the package.

- If any members explicitly declared as "**public**", they are visible everywhere, including in different classes and packages.
- "**private**" members are accessed by only other members of its class. A private member is unaffected by its membership in a package.
- A member specified as "**protected**" is accessed within its package and to all subclasses.

Class Member Access:

Sl No	Class member	Private Member	Default Member	Protected Member	Public Member
1	Visible within same class	YES	YES	YES	Yes
2	Visible within the same package by subclasses	No	YES	YES	YES
3	Visible within same package by non-subclass	No	YES	YES	YES
4	Visible within different packages by subclasses	NO	NO	YES	YES
5	Visible within different packages by Non-subclasses	NO	NO	NO	YES

Adding a class to a Package:

A.java

```
package p1;

public class A
{
                // body of the class
}
```

Here, the package p1 contains one public class by the name A. Suppose if we want to add another class B to this package. This can be done as follows:

1. Define the class and make it public
2. Place the package statement in the begin of class definition
   ```
   package p1;
   public class B
   {
   //body of the class B
   }
   ```
3. store this as "**B.java**" file under the directory p1.
4. Compile "**B.java**" file by switching to the subdirectory. This will create "**B.class**" file and place it in the directory **p1**.
   Now the package p1 will contain both A.class and B.class files

Introduction to Interfaces

Java supports the concept of Inheritance, that is acquiring the properties from one class to other class. The class that acquire properties is called **"subclass"**, and the class from which it acquire is called "**superclass**". Here, one class can acquire properties from other class using the following statement:

```
class A extends  B
{
---------
---------
}
```

But, Java does not allow to acquire properties from more than one class, which we call it as "**multiple inheritance**". We know that large number of real-life applications require the use of multiple inheritance. Java provides an alternative approach known as "***interface***" to support the concept of multiple inheritance.

Defining Interface

An interface is basically a kind of class. Like classes, interfaces contain the methods and variables but with major difference. The difference is that interface define only abstract methods and final fields. This means that interface do not specify any code to implement these methods and data fields contain only constants. Therefore, it is the responsibility of the class that implements an interface to define the code for these methods.

The syntax of defining an interface is very similar to that of class. The general form of an

```
        interface  Intyerface_name
{
        //variables inside interface are by default final, publicand static
        type id=value;
        //by default abstract methods
        return_type method_name(paprameter_list);
        return_type method_name(paprameter_list);
        return_type method_name(paprameter_list);

}
```
// Here, ***interface*** is the keyword and the Calculator is name for interface. The variables are declared as follows:

Note: 1) ***variables inside interface are by default final, public and static***
2) by default methods are public and abstract
These methods must be implemented **any class that want to acquire the properties.**

Here is an example of an interface definition that contain two variable and one method

```
interface Calculator
{
        //variables inside interface are by default final, public and static
        double PI=3.14;
        //by default abstract methods
        int add(int a,int b);
        int sub(int a,int b);
        int mul(int a,int b);
        int div(int a,int b);
        double area(int r);
}
```

Implementing the interface

Interfaces are used as "**superclasses**" whose properties are inherited by the classes. It is therefore necessary to create a class that inherits the given interface. This is done as follows:

```
        class Class_Name     implements     Interface_Name
        {
                //Body of the class
        }
```

Here, Class_Name class, implements the interface "Interface_Name". A more general form of implementation may look like this:

```
{
//        body of the class
}
```

This shows that a class can extend another class while implementing interfaces. When a class implements more than one interface they are separated by a comma.

Example program using interface

InterfaceTest.Java

```
interface Calculator
{
        //variables inside interface are by default final, publi and static
        double PI=3.14;
        //by default abstract methods
        int add(int a,int b);
        int sub(int a,int b);
        int mul(int a,int b);
        int div(int a,int b);
        double area(int r);
}
class NormCal implements Calculator
{
        public int add(int x,int y)
        {
        return(x+y);
        }
        public int sub(int x,int y)
        {
        return(x-y);
        }
        public int mul(int x,int y)
        {
        return(x*y);
        }
        public int div(int x,int y)
        {
        return(x/y);
        }
        public double area(int r)
        {
                return(PI*r*r);
        }
        public static void main(String args[])
        {
        NormCal nc=new NormCal();
        System.out.println("The sum is:"+nc.add(2,3));
        System.out.println("The sum is:"+nc.sub(2,3));
```

```
    }
}
```

OutPut:

```
E:\ksr>javac NormCal.java
E:\ksr>java  NormCal
The sum is:5
The sum is:-1
The sum is:6
The sum is:2
area of Circle is:78.5
```

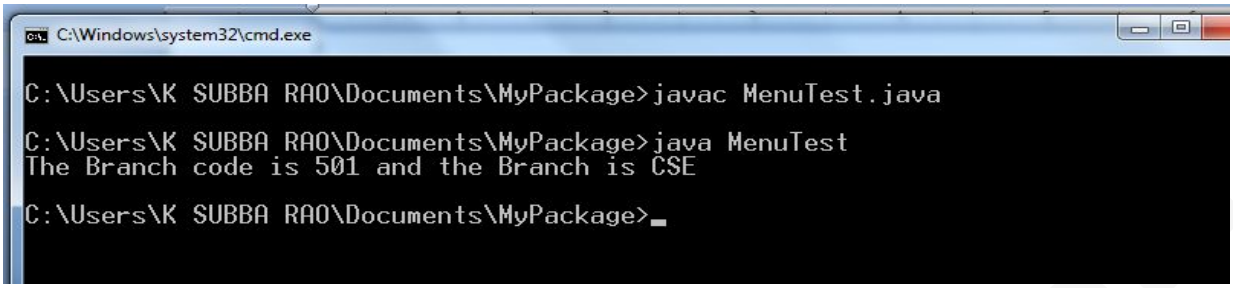
Extending the Interfaces

Like classes interfaces also can be extended. That is, an interface can be subinterfaced from other interface. The new subinterface will inherit all the members from the superinterface in the manner similar to the subclass. This is achieved using the keyword extends as shown here:

```
interface Interface_Name1 extends Interface_Name2
{
    //Body of the Interface_name1
}
```

For example, **MenuTest.java**

```
interface Const
{
    static final int code=501;
    static final String branch="CSE";
}
interface Item extends Const
{
    void display();
}
class Menu implements Item
{
    public void display()
    {
        System.out.println("The Branch code is "+code+" and the Branch is "+branch);
    }
}
class MenuTest{
    public static void main(String args[])
    {
        Menu m=new Menu(); // this contains the interfaces Item and Const
        m.display();
    }
```



Interfaces Vs Abstract classes

Sl	Interface	Abstract
1	Multiple Inheritance possible	Multiple Inheritance not possible
2	***implements*** keyword is used	***extends*** keyword is used
3	By default all the methods are public, and abstract. No need to tag as public and abstract	Methods have to be tagged as public and abstract.
4	All methods of interface need to be overridden	Only abstract methods need to be overridden
5	All variable declared in interface are By default public, final and static	Variable if required, need to be declared in interface as public, final and static
6	Methods cannot be static	Non-abstract methods can be static

Exceptions and Assertions:

Introduction to Exception

An Exception is an abnormal condition that arises in a code sequence at run time. In other words, an exception is run-time error. 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 method may choose to handle the exception itself, or pass it on. Either way, at some point, the exception is *caught* and processed.

Exception handling Techniques

Exception handling is managed by Java by **via five keywords:**

- **try**
- **catch**
- **throw**
- **throws**
- **finally**.

2. 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.
3. System-generated exceptions are **automatically** thrown by the Java run-time system. To **manually** throw an exception, use the keyword **throw**.
4. Any exception that is thrown out of a method must be specified as such by a **throws** clause.
5. Any code that absolutely must beexecuted after a **try** block completes is put in a **finally** block.

Syntax:

```
try
{
        // block of code to monitor for errors
}
catch (ExceptionType1 exOb)
{
        // exception handler for ExceptionType1
}
catch (ExceptionType2 exOb)
{
        // exception handler for ExceptionType2
}
// ...
finally
{
        // block of code to be executed after try block ends
}
```

Exception Types

All exception types are subclasses of the built-in class **Throwable**. Thus, **Throwable** is at thetop of the exception class hierarchy. Immediately below (Fig 2) **Throwable** are two subclasses thatpartition exceptions into two distinct branches. One branch is headed by **Exception**. This classis used for exceptional conditions that user programs should catch. This is also the class thatyou will subclass to create your own custom exception types. There is an important subclassof **Exception**, called **RuntimeException**. Exceptions of this type are **automatically** defined forthe programs that you write and include things such as division by zero and invalid arrayindexing.

The other branch is topped by **Error**, which defines exceptions that are not expected tobe caught under normal circumstances by your program. Exceptions of type **Error** are usedby 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 withexceptions of type **Error**, because these are typically created in response to catastrophic failuresthat cannot usually be handled by your program.

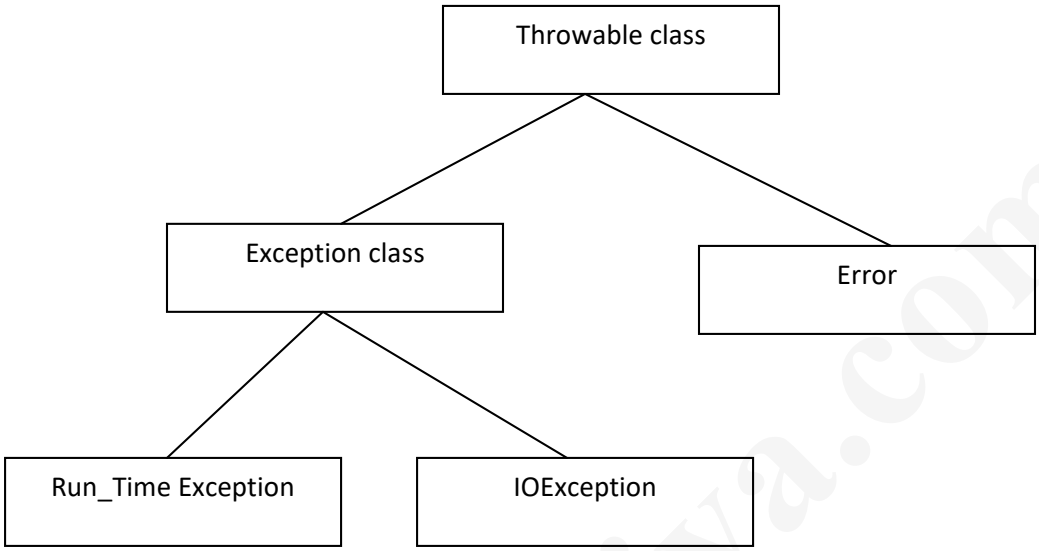


Fig 2: Exception Types

The Java Built-in Exceptions are broadly classified into two categories: Checked and Unchecked exceptions.The Checked Exceptions are those for which the compiler checks to see whether they have been handled in your program or not. Unchecked or Run_Time Exceptions are not checked by the compiler. These Unchecked Exceptions are handled by the Java Run_Time System automatically.

Checked Exceptions	Unchecked Exceptions
ClassNotFoundException	Arithmetic Exception
NoSuchFieldException	ArrayIndexOutOfBoundsException
NoSuchMethodException	NullPointerException
InerruptedException	ClassCastException
IOException	BufferOverFlowException
IllegalAccessException	BufferUnderFlowException

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 anew exception object and then ***throws***this exception. This causes the execution of **Exc0** to

In the above program we have not provided any exception handler, in this context the exception is caught by the **default handler**. The **default handler** displays string describing the exception.

Here is the exception generated when this example is executed:

```
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 typeof exception thrown is a subclass of **Exception** called **ArithmeticException**, which morespecifically describes what type of error happened.

Using try and catch

Although the default exception handler provided by the Java run-time system is useful fordebugging, you will usually want to handle an exception yourself. Doing so provides **twobenefits**.

- First, it allows you to fix the error.
- Second, it prevents the program from automatically terminating.

Most users would be confused (to say the least) 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 wantto monitor inside a **try** block. Immediately following the **try** block, include a **catch** clausethat specifies the exception type that you wish to catch. To illustrate how easily this can bedone, the following program includes a **try** block and a **catch** clause that processes the**ArithmeticException** generated by the division-by-zero error:

Exc2.java

```
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.");
```

Note: println() statement inside the "**try**" will never execute, because once the exception is raised the control is transferred to the "**catch**" block. Here is catch is not called, hence it will not return the control back to the "try" block. The "try" and "catch" will form like a unit. A catch statement cannot handle the exception thrown by another "try" block.

Displaying a Description of an Exception

Throwable overrides the **toString( )** method (defined by **Object**) so that it returns a stringcontaining a description of the exception. You can display this description in a **println( )**statement by simply passing the exception as an argument. For example, the **catch** blockin the preceding program can be rewritten like this:

```
catch (ArithmeticException e)
        {
                System.out.println("Exception: " + e);
                a = 0; // set a to zero and continue
        }
```

When this version is substituted in the program, and the program is run, each divide-by-zero error displays the following message:

Exception: java.lang.ArithmeticException: / by zero

Multiple catch Clauses

In some cases, more than one exception could be raised by a single piece of code. To handlethis type of situation, you can specify two or more **catch** clauses, each catching a differenttype of exception. When an exception is thrown, each **catch** statement is inspected in order,and the first one whose type matches that of the exception is executed. After one **catch**statement executes, the others are bypassed, and execution continues after the **try**/**catch**block. The following example traps two different exception types:

```
// Demonstrate multiple catch statements.
class MultiCatch
{
public static void main(String args[])
{
try
{
        int a = args.length;
        System.out.println("a = " + a);
        int b = 42 / a;
        int c[] = { 1 };
        c[42] = 99;
}
catch(ArithmeticException e)
{
        System.out.println("Divide by 0: " + e);
}
 catch(ArrayIndexOutOfBoundsException e)
```

```
System.out.println("After try/catch blocks.");
}
}
```

Nested try Statements

The **try** statement can be nested. That is, a **try** statement can be inside the block of another **try**.Each time a **try** statement is entered, the context of that exception is pushed on the stack. If aninner **try** statement does not have a **catch** handler for a particular exception, the stack is unwound and the next **try** statement's **catch** handlers are inspected for a match. This continuesuntil one of the **catch** statements succeeds, or until all of the nested **try** statements are exhausted.If no **catch** statement matches, then the Java run-time system will handle the exception.

throw:

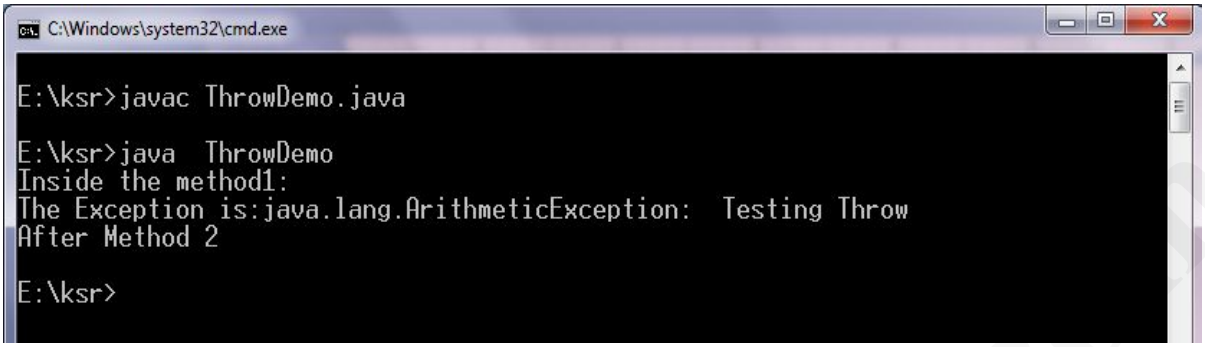
So far, you have only been catching exceptions that are thrown by the Java run-time system**implicitly**. However, it is possible for your program to throw an exception **explicitly**, using the **throw**statement. The general form of **throw** is shown here:

throw*ThrowableInstance***;**

Here, *ThrowableInstance* must be an object of type **Throwable** or a subclass of **Throwable**.
Example Program:

ThrowDemo.java

```
class ThrowDemo
{
        public static void main(String args[])
        {
         method1();
        }
        static void method1()
        {
                System.out.println("Inside the method1:");
                try
                {
                        method2();
                }
                catch(Exception e)
                {
                        System.out.println("The Exception is:"+e);
                }
                System.out.println("After Method 2");
        }
        static void method2()
        {
                throw new ArithmeticException(" Testing Throw");
```



throws:

If a method is capable of causing an exception that it does not handle, it must specify thisbehavior so that callers of the method can guard themselves against that exception. You dothis by including a **throws** clause in the method's declaration. A**throws** clause lists the typesof exceptions that a method might throw. This is necessary for all exceptions, except those oftype **Error** or **RuntimeException**, or any of their subclasses. All other exceptions that a methodcan throw must be declared in the **throws** clause. If they are not, a compile-time error will result.

This is the general form of a method declaration that includes a **throws** clause:

```
type method-name(parameter-list) throws exception-list
{
        // body of method
}
```

Here, *exception-list* is a comma-separated list of the exceptions that a method can throw.

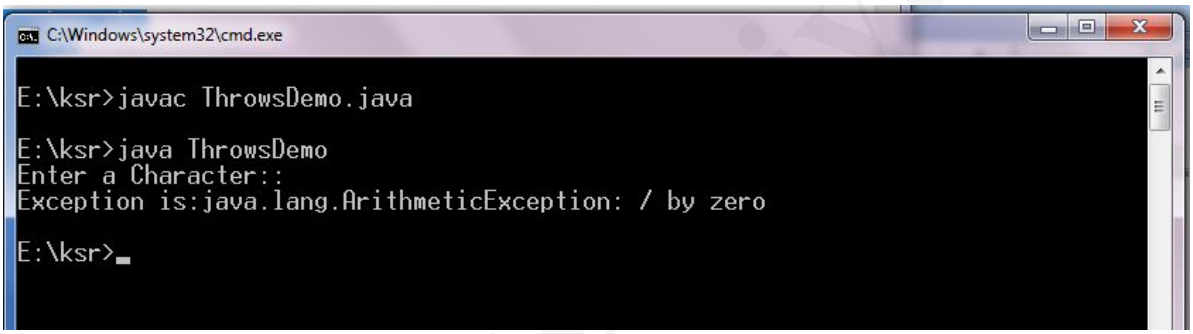
Example Program:

ThrowsDemo.java

```
import java.io.*;
class ThrowsDemo
{
public static char prompt(String str) throws IOException,ArithmeticException
{
//called method throws two exceptions
System.out.println(str+":");
int x=20,y=0;
int z=x/y;
return (char) System.in.read();
}
public static void main(String args[])
{
char ch;
try
```

```
                //it is the responsibility of the caller to handle it
}
catch(IOExceptione)
{
                System.out.println("Exception is:"+e);
}
catch(ArithmeticExceptionae)
{
                System.out.println("Exception is:"+ae);
}
}
}
```

Output:



finally:

The finally block is always is executed. It is always used to perform house keeping operations such as releasing the resources, closing the files that already opened etc,. That means the statement that put inside the finally block are executed compulsorily.It is always followed by the try-catch statements.

Syntax:

```
try
{
        // statements
        }
catch(Exception e)      {
        //Handlers
}
finally
{
        //statements
}
```

Exception encapsulation and enrichment

The process of wrapping the caught exception in a different exception is called "Exception Encapsulation". The Throwable super class has added one parameter in its constructor for the wrapped exception and a "getCause()" method to return the wrapped exception. Wrapping is also used to hide the details of implementation.

```
            throw new ArithmeticException();
        }
        catch(AritmeticExceptions ae)
        {
        //wrapping exception
            throw new ExcepDemo("Testing User Exception",ae);
        }
```

Disadvantages of wrapping:

- It leads to the long Stack traces.
- It is a difficult task to figure out where the exception is

Solution:

The possible solution is Exception enrichment. Here we don't wrap exception but we add some information to the already thrown exception and rethrow it.

Example program:

```
class ExcepDempo extends Exception
{
        String message;
        ExcepDemo(String msg)
        {
            message=msg;
        }
        public void addInformation(String msg)
        {
            message=message+msg;
        }
}
class ExcepEnrich
{
        static void testException()
        {
            try{
                throw new ExcepDemo("Testing user Exceptio:");
            }
            catch(ExcepDemo e)
            {
                e.addInformation("Example Exception");
            }
        }
        public static void main(String args[])
        {
            try
        {
            testException();
        }
        catch(Exception e)
        {
```

}

Assertions:

Assertions are added after java 1.4 to always createreliable programs that are Correct and robust programs. The assertions are boolean expressions.Conditions such as positve number or negative number are examples.

Syntax:

```
assert expression1;
        or
assert expression1:expression2;
```

Where **assert** is the keyword. Expression 1 is the boolean expression, expression2 is the string that describes the Exceptions.

Note: assertions must be explicitly enabled. The –ea option is used to enable the exception and –da is used to disable the exception.

AI.java

```
class AI
{
        static void check(int i)
        {
                assert i>0:" I must be positive:";
                System.out.prinln("Your I value if fine");
        }
        pulbic static void main(String args[])
        {
                check(Integer.parseInt(args[0]);
        }
}
```

Compiling the program:

c:>javac AI.java

Running the Program:

C:>java –ea AI

----------*End of 3rd Unit***----------**

Unit 4
Multithreading

Topics:
MultiThreading: java.lang.Thread, the main thread, Creation of new thread, Thread priority, Multithreading, Synchronization, Suspending and resuming threads, Communication between threads, Input/Output: reading and writing data, java.io.package.

Introduction

The program in execution is called "**Process**". The program can be structured as set of individual units that can run in parallel. These units can be called as "**Threads**". Multithreading is actually a kind of multitasking. The multitasking is either process-based or thread base. Process-based multitasking is nothing but, execution of more than one program concurrently. Thread-based multitasking allows more than one thread within the program run simultaneously or concurrently.The process is a Heavy Weight Process. The Thread is a Light Weight Process. The context-switching of CPU from one process to other requires more overhead as it different address spaces are involved in it. On the other hand, context-switching is less overhead because of all the threads within the same program. The objective of all forms of the Multitasking including the multithreading is to utilize the idle time of the processor. From here onwards, thread-based multitasking is called "**Multithreading**".

Difference between Multiprocessingand Multithreading

Process-Based Multitasking	Thread-Based Multitasking
This deals with "Big Picture"	This deals with Details
These are Heavyweight tasks	These are Lightweight tasks
Inter-process communication is expensive and limited	Inter-Thread communication is inexpensive.
Context switching from one process to another is costly in terms of memory	Context switching is low cost in terms of memory, because they run on the same address space
This is not under the control of Java	This is controlled by Java

Multithreading in Java

Every program that we have been writing has at least one thread, that is, the "**main**" thread. Whenever a program starts executing, the JVM is responsible for creating the main thread and calling "main()" method. Along with this main thread, some other threads are also running to carryout the tasks such as "**finalization**" and "**garbage collection**". The thread can either die naturally or be forced to die.

- Thread dies naturally when it exits from the "run()" method.
- Thread can be forced to die by calling "interrupt()" method.

java.lang.Thread package

Creation of Thread in java is very simple task. There is a class called "Thread", which belongs to the "java.lang.Thread" package. This package contains one interface also called "Runnable". Both these contain a common method called "**run()**" which is the heart of the thread. The run() methods would have the following syntax:

Syntax:

```
public void run()
{
        //statement for implementing the thread.
}
```

The methods of the Thread class are as follow:

1. **public void run():** is used to perform action for a thread.
2. **public void start():** starts the execution of the thread.JVM calls the run() method on the thread.
3. **public void sleep(long miliseconds):** Causes the currently executing thread to sleep (temporarily cease execution) for the specified number of milliseconds.
4. **public void join():** waits for a thread to die.
5. **public void join(long miliseconds):** waits for a thread to die for the specified miliseconds.
6. **public int getPriority():** returns the priority of the thread.
7. **public int setPriority(int priority):** changes the priority of the thread.
8. **public String getName():** returns the name of the thread.
9. **public void setName(String name):** changes the name of the thread.
10. **public Thread currentThread():** returns the reference of currently executing thread.
11. **public int getId():** returns the id of the thread.
12. **public Thread.State getState():** returns the state of the thread.
13. **public boolean isAlive():** tests if the thread is alive.
14. **public void yield():** causes the currently executing thread object to temporarily pause and allow other threads to execute.
15. **public void suspend():** is used to suspend the thread(depricated).
16. **public void resume():** is used to resume the suspended thread(depricated).
17. **public void stop():** is used to stop the thread(depricated).
18. **public boolean isDaemon():** tests if the thread is a daemon thread.

19. **public void setDaemon(boolean b):** marks the thread as daemon or user thread.
20. **public void interrupt():** interrupts the thread.
21. **public boolean isInterrupted():** tests if the thread has been interrupted.
22. **public static boolean interrupted():** tests if the current thread has been interrupted.

Thread Constructors:

- Thread ()-without arguments, default constructor
- Thread(String str)- Thread contains name given as argument

The Main Thread

Every java program has a thread called "main" thread. When the program execution starts, the JVM creates "main" Thread and calls the "main()" method from within that thread. Along with this JVM also creates other threads for the purpose of the Housekeeping task such as "garbage" collection. The "main" thread Spawns the other Threads. These spawned threads are called "Child Threads". The main thread is always is the last thread to finish execution. We, as Programmer can also take control of the main thread, using the method "**currentThread()**". The main thread can be controlled by this method. We can also change the name of the Thread using the method "**setName(String name)**".

Example Program:

MainThread.java

```
class MainThread
{
        public static void main(String args[])
        {
        Thread t=Thread.currentThread();
        System.out.println("Name of the Thread is:"+t);
        t.setName("KSR");
        System.out.println("Name of the Thread is:"+t);
        }
}
```

Output:

```
E:\ksr>javac MainThread.java

E:\ksr>java  MainThread
Name of the Thread is:Thread[main,5,main]
Name of the Thread is:Thread[KSR,5,main]

E:\ksr>_
```

TutorialsDuniya.com

Download FREE Computer Science Notes, Programs, Projects, Books PDF for any university student of BCA, MCA, B.Sc, B.Tech CSE, M.Sc, M.Tech at https://www.tutorialsduniya.com

- Algorithms Notes
- Artificial Intelligence
- Android Programming
- C & C++ Programming
- Combinatorial Optimization
- Computer Graphics
- Computer Networks
- Computer System Architecture
- DBMS & SQL Notes
- Data Analysis & Visualization
- Data Mining
- Data Science
- Data Structures
- Deep Learning
- Digital Image Processing
- Discrete Mathematics
- Information Security
- Internet Technologies
- Java Programming
- JavaScript & jQuery
- Machine Learning
- Microprocessor
- Operating System
- Operational Research
- PHP Notes
- Python Programming
- R Programming
- Software Engineering
- System Programming
- Theory of Computation
- Unix Network Programming
- Web Design & Development

Please Share these Notes with your Friends as well

Creation of Threads

Creating the threads in the Java is simple. The threads can be implemented in the form of object that contains a method "**run()**". The "**run()**" method is the heart and soul of any thread. It makes up the entire body of the thread and is the only method in which the thread behavior can be implemented. There are two ways to create thread.

1. Declare a class that **extends** the **Thread** class and override the **run()** method.
2. Declare a class that implements the **Runnable** interface which contains the **run()** method

1. Creating Thread using The Thread Class

We can make our thread by extending the Thread class of java.lang.Thread class. This gives us access to all the methods of the Thread. It includes the following steps:

I. Declare the class as Extending the Thread class.
II. Override the "**run()**" method that is responsible for running the thread.
III. Create a thread and call the "**start()**" method to instantiate the Thread Execution.

Declaring the class

TheThread class can be declared as follows:

```
class MyThread extends Thread
{
        ----------------------
        ----------------------
        ----------------------
        ----------------------
}
```

Overriding the method run()

The run() is the method of the Thread. We can override this as follows:

```
public void run()
{
        ---------------
        ---------------
        ---------------
}
```

Starting the new Thread

To actually to create and run an instance of the thread class, we must write the following:

```
MyThread a=new MyThread();  // creating the Thread
a.start();                  // Starting the Thread
```

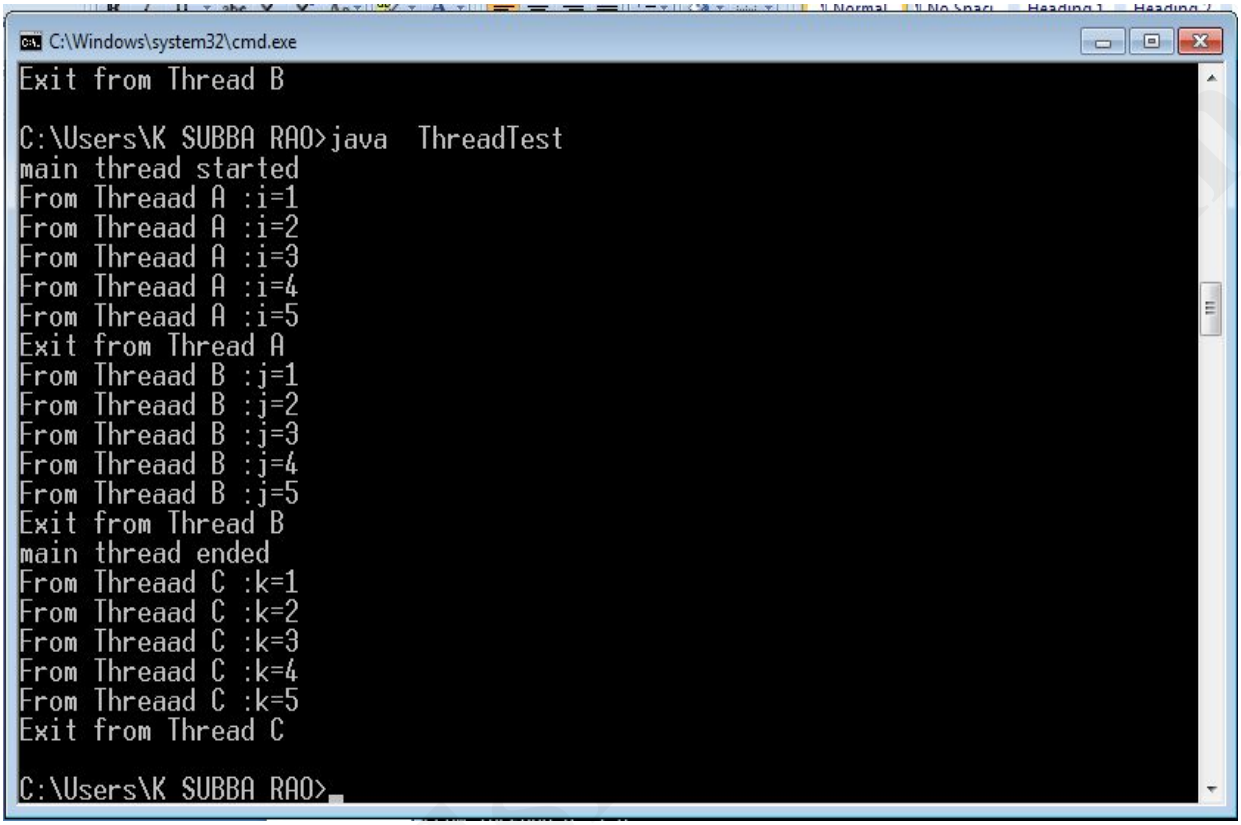
Example program:

ThreadTest.java

```
import java.io.*;
import java.lang.*;

class A extends Thread
```

```
{
	public void run()
	{
		for(int i=1;i<=5;i++)
		{
			System.out.println("From Threaad A :i="+i);
		}
		System.out.println("Exit from Thread A");
	}
}

class B extends Thread
{
	public void run()
	{
		for(int j=1;j<=5;j++)
		{
			System.out.println("From Threaad B :j="+j);
		}
		System.out.println("Exit from Thread B");
	}
}
class C extends Thread
{
	public void run()
	{
		for(int k=1;k<=5;k++)
		{
			System.out.println("From Threaad C :k="+k);
		}
		System.out.println("Exit from Thread C");
	}
}
class ThreadTest
{
	public static void main(String args[])
	{
		System.out.println("main thread started");
		A a=new A();
		a.start();
		B b=new B();
		b.start();
		C c=new C();
		c.start();
	System.out.println("main thread ended");
	}
}
```

output: First Run



2. creating the Thread using Runnable Interface

The Runnable interface contains the run() method that is required for implementing the threads in our program. To do this we must perform the following steps:

I. Declare a class as implementing the **Runnable** interface
II. Implement the **run()** method
III. Create a **Thread** by defining an object that is instantiated from this "runnable" class as the target of the thread
IV. Call the thread's **start()** method to run the thread.

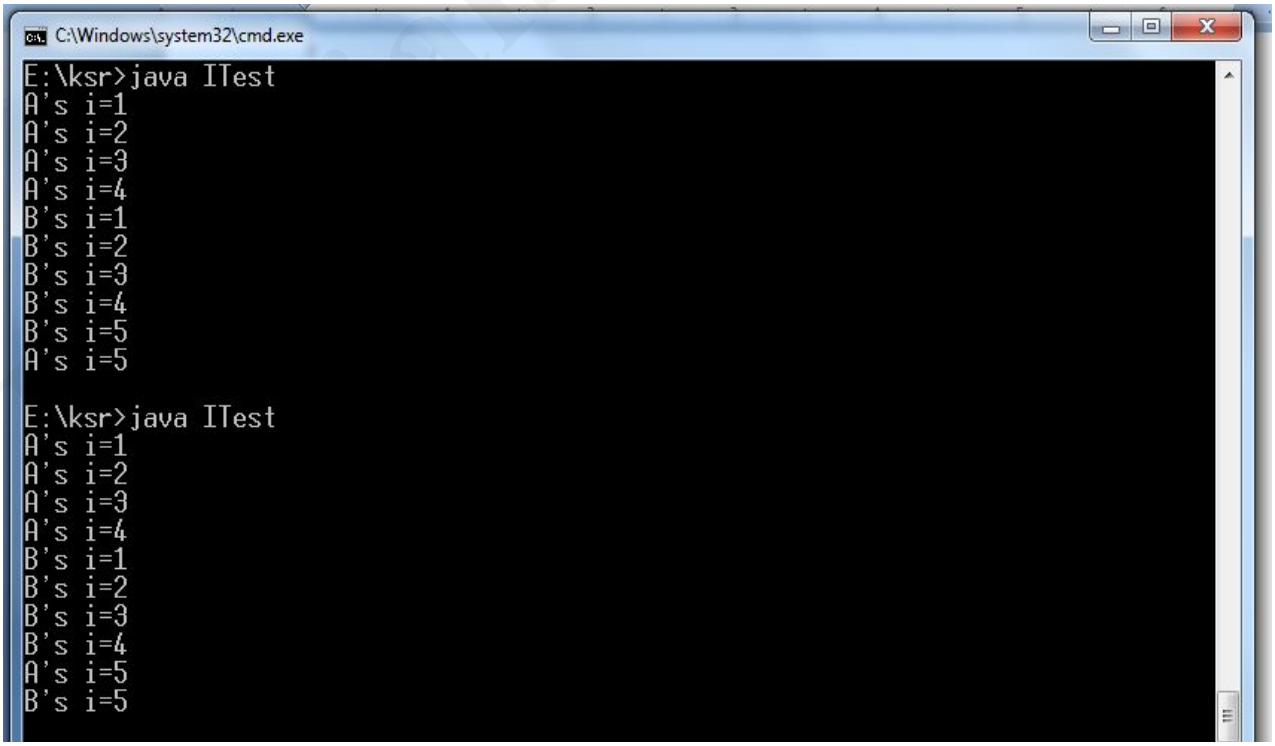
Example program:

Runnable.java

```
class A implements Runnable
{
        public void run()
        {
        for(int i=1;i<=5;i++)
        {
                System.out.println("A's i="+i);
        }
        }
}
```

```
class B implements Runnable
{
        public void run()
        {
        for(int i=1;i<=5;i++)
        {
                System.out.println("B's i="+i);
        }
        }
}
class ITest
{
        public static void main(String args[])
        {
                A a=new A();
                Thread t1=new Thread(a);
                t1.start();
                B b=new B();
                Thread t2=new Thread(b);
                t2.start();
        }

}
```

Output: Threads A and B execution by running the above program two times. (You may see a different sequence of Output, every time you run this program)



Advantage of the Multithreading

- It enables you to write very efficient programs that maximizes the CPU utilization and reduces the idle time.
- Most I/O devices such as network ports, disk drives or the keyboard are much slower than CPU
- A program will spend much of it time just send and receive the information to or from the devices, which in turn wastes the CPU valuable time.
- By using the multithreading, your program can perform another task during this idle time.
- For example, while one part of the program is sending a file over the internet, another part can read the input from the keyboard, while other part can buffer the next block to send.
- It is possible to run two or more threads in multiprocessor or multi core systems simultaneously.

Thread States

A thread can be in one of the several states. In general terms, a thread can ***running***. It can be ***ready*** to run as soon as it gets the CPU time.A running thread can be ***suspended***, which is a temporary halt to its execution. It can later be ***resumed***. A thread can be ***blocked*** when waiting for the resource. A thread can be ***terminated***.

Single Threaded Program

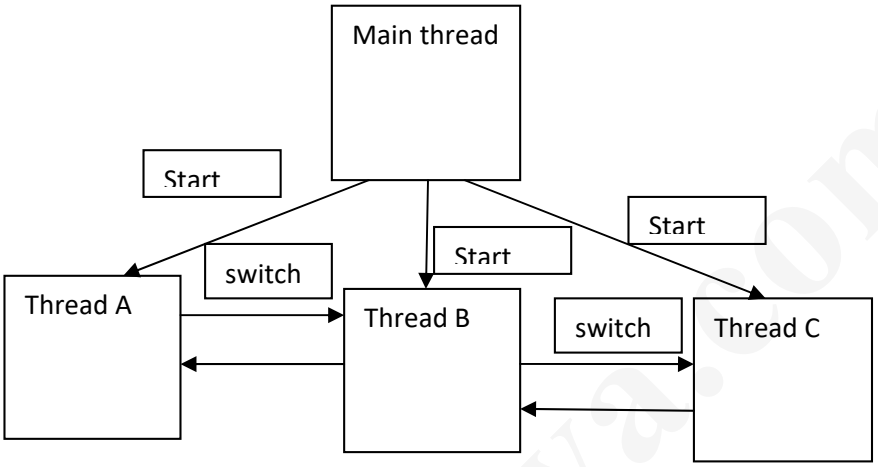
A Thread is similar to simple program that contains single flow of control. It has beginning, body, and ending. The statements in the body are executed in sequence.

For example:

```
class ABC                    //Begining

        {
                ----------      //body
                ----------
        }                    //ending
```

Multithreaded Program

- A unique property of the java is that it supports the multithreading. Java enables us the multiple flows of control in developing the program.
- Each separate flow of control is thought as tiny program known as "**thread**" that runs in parallel with other threads.
- In the following example when the main thread is executing, it may call thread A, as the Thread A is in execution again a call is mad for Thread B. Now the processor is switched from Thread A to Thread B. After the task is finished the flow of control comes back to the Thread A.
- The ability of the language that supports multiple threads is called "***Concurrency***". Since threads in the java are small sub programs of the main program and share the same address space, they are called "***light weight processes***".



The Main Thread

When a Java program starts up, one thread begins running immediately. This is usuallycalled the *main thread* of your program, because it is the one that is executed when yourprogram begins. The main thread is important for two reasons:

- It is the thread from which other "child" threads will be spawned.
- Often, it must be the last thread to finish execution because it performs variousshutdown actions.

Although the main thread is created automatically when your program is started, it canbe controlled through a **Thread** object. To do so, you must obtain a reference to it by callingthe method **currentThread( )**, which is a **public static** member of **Thread**. Its general form isshown here:

static Thread.currentThread( )

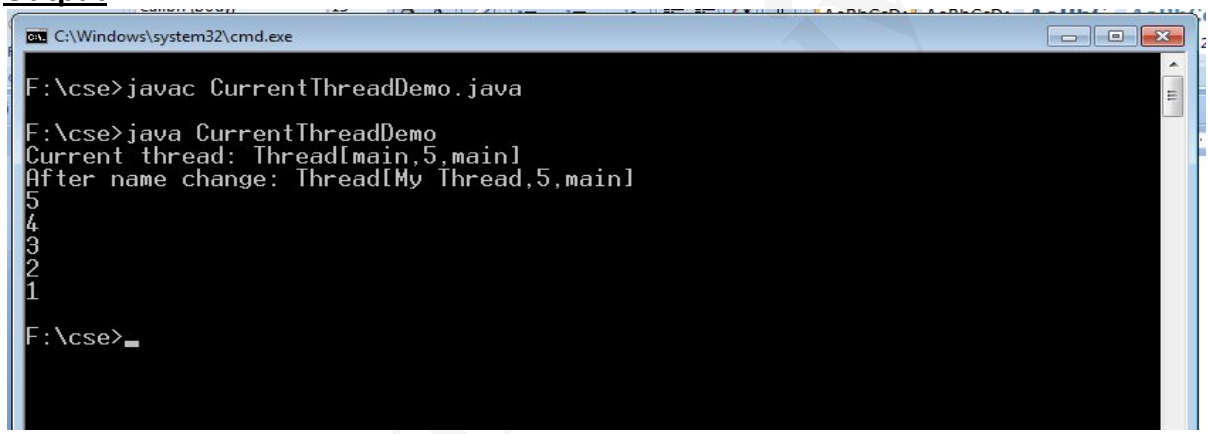
This method returns a reference to the thread in which it is called. Once you have a reference to the main thread, you can control it just like any other thread.
Let's begin by reviewing the following example:

CurrentThreadDemo.java

```
// 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 interrupted");
        }
}
}
```

Output



Creating Multiple Threads

So far, you have been using only two threads: the main thread and one child thread. However, your program can spawn as many threads as it needs. For example, the following program creates three child threads:

// **Create multiple threads.**

```
class NewThread implements Runnable
{
String 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 = 5; i > 0; i--)
                {
                        System.out.println(name + ": " + i);
                        Thread.sleep(1000);
                }
        } catch (InterruptedException e)
        {
                System.out.println(name + "Interrupted");
        }
System.out.println(name + " exiting.");
}//end of run  method
}//end of NewThread
class MultiThreadDemo
{
public static void main(String args[])
{
new NewThread("One"); // start threads
new NewThread("Two");
new NewThread("Three");
try
{
        // wait for other threads to end
        Thread.sleep(10000);
}
 catch (InterruptedException e)
{
        System.out.println("Main thread Interrupted");
}
System.out.println("Main thread exiting.");
}
}
```

When a Thread is ended

It is often very important to know which thread is ended. This helps to prevent the main from terminating before the child Thread is terminating. To address this problem "Thread" class provides two methods: **1) Thread.isAlive() 2) Thread.join()**.

The general form of the **"isAlive()"** method is as follows:

final boolean isAlive();

This method returns the either "**TRUE**" or "**FALSE**" . It returns "TRUE" if the thread is alive, returns "FALSE" otherwise.

While **isAlive( )** is occasionally useful, the method that you will more commonly use towait for a thread to finish is called **join( )**, shown here:

final void join( ) throws InterruptedException

This method waits until the thread on which it is called terminates. Its name comes from theconcept of the calling thread waiting until the specified thread *joins* it. Additional forms of**join( )** allow you to specify a maximum amount of time that you want to wait for the specifiedthread to terminate.

Example Program:

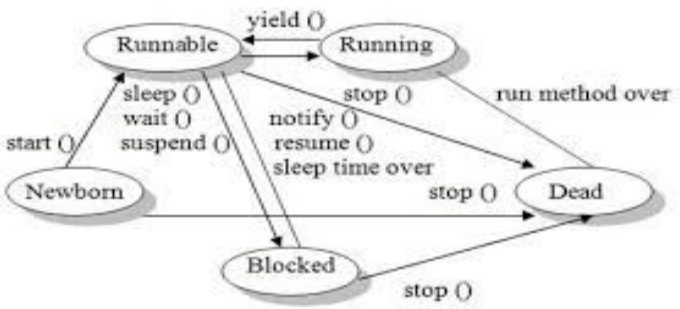
```
// Using join() to wait for threads to finish.
class NewThread implements Runnable
{
        String name; // name of thread
        Thread t;
        NewThread(String threadname)
        {
                name = threadname;
                t = new Thread( name);
                System.out.println("New thread: " + t.getName());
                t.start(); // Start the thread
        }
// This is the entry point for thread.
public void run()
{
try
 {
        for(int i = 5; i > 0; i--)
        {
                System.out.println(name + ": " + i);
                Thread.sleep(1000);
        }
}
 catch (InterruptedException e)
        {
                System.out.println(name + " interrupted.");
        }
System.out.println(name + " exiting.");
}
}
```

```
class DemoJoin
{
public static void main(String args[])
{
        NewThread ob1 = new NewThread("One");
        NewThread ob2 = new NewThread("Two");
        NewThread ob3 = new NewThread("Three");
        System.out.println("Thread One is alive: "+ ob1.t.isAlive());
        System.out.println("Thread Two is alive: "+ ob2.t.isAlive());
        System.out.println("Thread Three is alive: "+ ob3.t.isAlive());
        // wait for threads to finish
        try {
                System.out.println("Waiting for threads to finish.");
                ob1.t.join();
                ob2.t.join();
                ob3.t.join();
        }
         catch (InterruptedException e)
        {
                System.out.println("Main thread Interrupted");
        }
        System.out.println("Thread One is alive: "+ ob1.t.isAlive());
        System.out.println("Thread Two is alive: "+ ob2.t.isAlive());
        System.out.println("Thread Three is alive: "+ ob3.t.isAlive());
        System.out.println("Main thread exiting.");
}
}
```

Life Cycle of a Thread

During the life time of the thread, there are many states it can enter. They include the following:

- Newborn state
- Runnable State
- Running State
- Blocked state
- Dead state



Life-cycle of a thread.

A thread can always in any one of the five states. It can move from one state to other via variety of ways as shown in the fig.

Newborn State: When we create a thread it is said to be in the new born state. At this state we can do the following:

- schedule it for running using the start() method.
- Kill it using stop() method.

Runnable State: A runnable state means that a thread is ready for execution and waiting for the availability of the processor. That is the thread has joined the queue of the threads for execution. If all the threads have equal priority, then they are given time slots for execution in the round rabin fashion, first-come, first-serve manner. The thread that relinquishes the control will join the queue at the end and again waits for its turn. This is known as time slicing.

Running state: Running state means that the processor has given its time to the thread for it execution. The thread runs until it relinquishes the control or it is preempted by the other higher priority thread. As shown in the fig. a running thread can be preempted using the suspen(), or wait(), or sleep() methods.

Blocked state: A thread is said to be in the blocked state when it is prevented from entering into runnable state and subsequently the running state.

Dead state: Every thread has a life cycle. A running thread ends its life when it has completed execution. It is a natural death. However we also can kill the thread by sending the stop() message to it at any time.

The Thread Priorities

Thread priorities are used by the thread ***scheduler*** to decide when and which thread should beallowed to run. In theory, **higher-priority** threads get more CPU time than **lower-priority**threads. In practice, the amount 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**(stop) alower-priority one. For instance, when a lower-priority thread is running and a higher-prioritythread **resumes** (from sleeping or waiting on I/O, for example), it will preempt the lowerprioritythread.

To set a thread's priority, use the **setPriority( )** method, 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 bewithin the range **MIN_PRIORITY** and **MAX_PRIORITY**. Currently, these values are 1 and10, respectively. To return a thread to default priority, specify **NORM_PRIORITY**, which iscurrently 5. These priorities are defined as **static final** variables within **Thread**.

You can obtain the current priority setting by calling the **getPriority( )** method of **Thread**,shown here:

final int getPriority( )

Example Program:

//setting the priorities for thethread

```
        class PThread1 extends Thread
        {
                public void run()
                {
                        System.out.println(" Child 1 is started");
                }
        }
        class PThread2 extends Thread
        {
                public void run()
                {
                        System.out.println(" Child 2 is started");
                }
        }
        class PThread3 extends Thread
        {
                public void run()
                {
                        System.out.println(" Child 3 is started");
                }
        }

class PTest
{
   public static void main(String args[])
   {
   //setting the priorities to the thread using the setPriority() method
        PThread1 pt1=new PThread1();
                pt1.setPriority(1);
        PThread2 pt2=new PThread2();
                pt2.setPriority(9);
        PThread3 pt3=new PThread3();
                pt3.setPriority(6);
        pt1.start();
        pt2.start();
        pt3.start();
        //getting the priority
        System.out.println("The pt1 thread priority is :"+pt1.getPriority());
   }
}
```

Synchronization

When two or more threads need access to a shared resource, they need some way to ensurethat the resource will be used by only one thread at a time. The process by which this isachieved is called ***synchronization***.

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 ***mutex***. Only one thread can *own* amonitor 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 firstthread ***exits***the monitor. These other threads are said to be ***waiting***for the monitor. A threadthat owns a monitor can reenter the same monitor if it so desires.

Let us try to understand the problem without synchronization. Here, in the following example to threads are accessing the same resource (object) to print the Table. The Table class contains one method, printTable(int), which actually prints the table. We are creating two Threads, Thread1 and Thread2, which are using the same instance of the Table Resource (object), to print the table. When one thread is using the resource, no other thread is allowed to access the same resource Table to print the table.

Example without the synchronization:

```
Class Table
{

        void printTable(int n)
        {//method not synchronized
                for(int i=1;i<=5;i++)
            {
                System.out.println(n*i);
                try{
                        Thread.sleep(400);
                }
                catch(InterruptedException ie)
                {
                        System.out.println("The Exception is :"+ie);
                    }
                }
          } //end of the printTable() method
}

class MyThread1 extends Thread
{
Table t;
        MyThread1(Table mt)
        {
                t=mt;
        }
        public void run()
        {
```

```
            t.printTable(5);
        }
} //end of the Thread1

class MyThread2 extends Thread
{
Table t;
        MyThread2(Table mt)
        {
            t=mt;
        }
        public void run()
        {
            t.printTable(100);
        }
} //end of Thread2

class TestSynchronization1
{
public static void main(String args[])
{
Table obj = new Table();//only one object
MyThread1 t1=new MyThread1(obj);
MyThread2 t2=new MyThread2(obj);
t1.start();
t2.start();
}
}
```

The output for the above program will be as follow:

```
Output: 5
        100
        10
        200
        15
        300
        20
        400
        25
        500
```

In the above output, it can be observed that both the threads are simultaneously accessing the Table object to print the table. Thread1 prints one line and goes to sleep, 400 milliseconds, and Thread1 prints its task.

Using the Java synchronized method

If you declare any method as synchronized, it is known as synchronized method.Synchronized method is used to lock an object for any shared resource.When a thread invokes a synchronized

method, it automatically acquires the lock for that object and releases it when the thread completes its task.

The general form of the synchronized method is:

synchronized type method_name(para_list)

{

//body of the method

}

where synchronized is the keyword, method contains the type, and method_name represents the name of the method, and para_list indicate the list of the parameters.

Example using the synchronized method

```
Class Table
{

        synchronized void printTable(int n)
        {//method not synchronized
                for(int i=1;i<=5;i++)
            {
                System.out.println(n*i);
                try{
                        Thread.sleep(400);
                }
                catch(InterruptedException ie)
                {
                        System.out.println("The Exception is :"+ie);
                    }
            }
          } //end of the printTable() method
}

class MyThread1 extends Thread
{
Table t;
        MyThread1(Table t)
        {
                this.t=t;
        }
        public void run()
        {
                t.printTable(5);
        }
 } //end of the Thread1

class MyThread2 extends Thread
{
Table t;
        MyThread2(Table t)
        {
```

```
            this.t=t;
        }
        public void run()
        {
            t.printTable(100);
        }
} //end of Thread2

class TestSynchronization1
{
public static void main(String args[])
{
Table obj = new Table();//only one object
MyThread1 t1=new MyThread1(obj);
MyThread2 t2=new MyThread2(obj);
t1.start();
t2.start();
}
}
```

```
Output: 5
        10
        15
        20
        25
        100
        200
        300
        400
        500
```

In the above output it can be observed that when Thread1 is accessing the Table object, Thread2 is not allowed to access it. Thread1 preempts the Thread2 from accessing the printTable() method.

Note:

1. This way of communications between the threads competing for same resource is called **implicit communication.**
2. This has one disadvantage due to polling. The polling wastes the CPU time. To save the CPU time, it is preferred to go to the **inter-thread communication.**

Inter-Thread Communication

If two or more Threads are communicating with each other, it is called "inter thread" communication. Using the synchronized method, two or more threads can communicate indirectly. Through, synchronized method, each thread always competes for the resource. This

way of competing is called **polling**. The polling wastes the much of the CPU valuable time. The better solution to this problem is, just notify other threads for the resource, when the current thread has finished its task, meanwhile other threads will be doing some useful work.. This is **explicit communication** between the threads.

Java addresses this polling problem, using via **wait()**, **notify()**, and **notifyAll()** methods. These methods are implemented as **final**methods in **Object**, so all classes have them. All three methods can be called only fromwithin a **synchronized** context.

- **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 a thread that called **wait( )** on the same object.
- **notifyAll( )** wakes up all the threads that called **wait( )** on the same object. One of the threads will be granted access.

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 allow you to specify a period of time to wait.

Although **wait( )** normally waits until **notify( )** or**notifyAll( )** is called, there is a possibility that in very rare cases the waiting thread could beawakened due to a *spurious wakeup*. In this case, a waiting thread resumes without **notify( )**or **notifyAll( )** having been called. (In essence, the thread resumes for no apparent reason.)Because of this remote possibility, Sun recommends that calls to **wait( )** should take placewithin a loop that checks the condition on which the thread is waiting. The followingexample shows this technique.

Example program for producer and consumer problem

```
class Q
{
        int n;
        boolean valueSet = false;//flag
        synchronized int get()
        {
        while(!valueSet)
        try {
                wait();
        }
         catch(InterruptedException e)
        {
                System.out.println("InterruptedException caught");
        }
        System.out.println("Got: " + n);
```

```
        valueSet = false;
        notify();
        return n;
        } //end of the get() method
        synchronized void put(int n)
        {
        while(valueSet)
        try {
                wait();
        }
        catch(InterruptedException e)
        {
                System.out.println("InterruptedException caught");
        }
        this.n = n;
        valueSet = true;
        System.out.println("Put: " + n);
        notify();
        }//end of the put method
} //end of the class Q

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++);
        }
}
}//end of Producer
class Consumer implements Runnable
{
        Q q;
Consumer(Q q)
{
this.q = q;
new Thread(this, "Consumer").start();
}
```

```
public void run()
{
        while(true)
        {
                q.get();
        }
}
}//end of Consumer


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.");
}
}
```

Suspending, Blocking and StoppingThreads

Whenever we want stop a thread we can stop from running using "**stop()**" method of thread class. It's general form will be as follows:

T**hread.stop();**

This method causes a thread to move from **running** to **dead** state. A thread will also move to dead state automatically when it reaches the end of its method.

Blocking Thread

A thread can be temporarily suspended or blocked from entering into the runnable and running state by using the following methods:

sleep() —blocked for specified time
suspend() ----blocked until further orders
wait() --blocked until certain condition occurs

These methods cause the thread to go into the blocked state. The thread will return to the runnable state when the specified time is elapsed in the case of **sleep()**, the **resume()** method is invoked in the case of **suspend()**, and the **notify()** method is called in the case of **wait()**.

Example program:

The following program demonstrates these methods:

// Using suspend() and resume().

```
class NewThread implements Runnable
{
        String name; // name of thread
        Thread t;
        NewThread(String threadname)
```

TutorialsDuniya.com

Download FREE Computer Science Notes, Programs, Projects, Books PDF for any university student of BCA, MCA, B.Sc, B.Tech CSE, M.Sc, M.Tech at https://www.tutorialsduniya.com

- Algorithms Notes
- Artificial Intelligence
- Android Programming
- C & C++ Programming
- Combinatorial Optimization
- Computer Graphics
- Computer Networks
- Computer System Architecture
- DBMS & SQL Notes
- Data Analysis & Visualization
- Data Mining
- Data Science
- Data Structures
- Deep Learning
- Digital Image Processing
- Discrete Mathematics
- Information Security
- Internet Technologies
- Java Programming
- JavaScript & jQuery
- Machine Learning
- Microprocessor
- Operating System
- Operational Research
- PHP Notes
- Python Programming
- R Programming
- Software Engineering
- System Programming
- Theory of Computation
- Unix Network Programming
- Web Design & Development

Please Share these Notes with your Friends as well

```
        {
        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);
        ob1.t.suspend();
        System.out.println("Suspending thread One");
        Thread.sleep(1000);
        ob1.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 Two");
}
 catch (InterruptedException e)
{
```

```
            System.out.println("Main thread Interrupted");
}
// wait for threads 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.");
}
}
```

Thread Exceptions

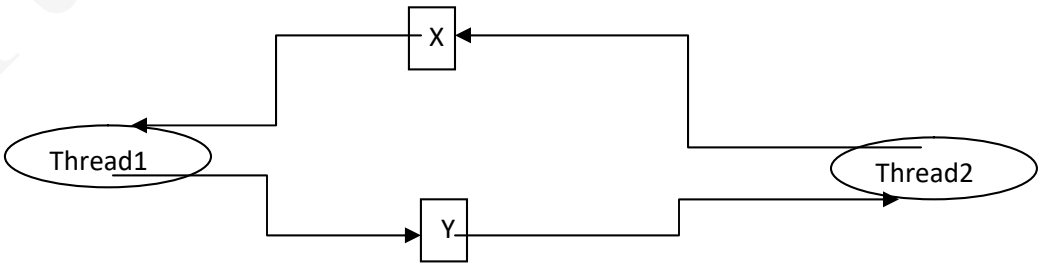
Note that a call to the sleep() method is always enclosed in try/ catch block. This is necessary because the sleep() method throws an exception, which should be caught. If we fail to catch the exception the program will not compile.

it general form will be as follows:

```
try
{
            Thread.sleep(1000);
}
cathc(Exception e)
{           --------
            ---------
}
```

Deadlock

Deadlock in java is a part of multithreading. Deadlock can occur in a situation when a thread is waiting for an object lock, that is acquired by another thread and second thread is waiting for an object lock that is acquired by first thread. Since, both threads are waiting for each other to release the lock, the condition is called **deadlock.**



Here, in the above figure, the resource X is held by Thread1, and at the same time the Thread1 is trying to access the resource which is held by the Thread2. This is causing the circular dependency between two Threads. This is called, Deadlock.

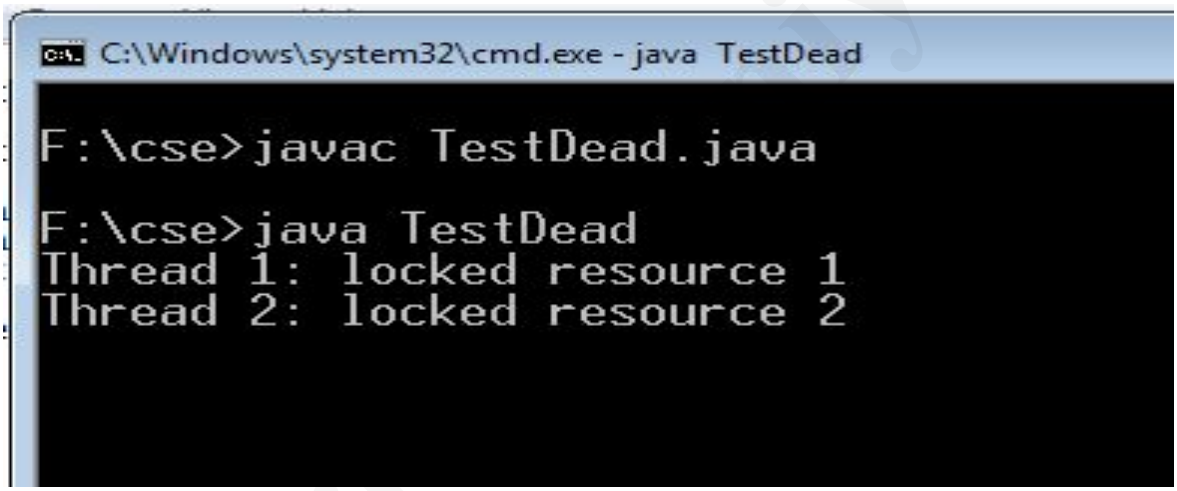
Example program:

TestDead.java

```
public class TestDead
{
 public static void main(String[] args)
{
   final String resource1 = "John Gardner";
   final String resource2 = "James Gosling";
   // t1 tries to lock resource1 then resource2

   Thread t1 = new Thread()
         {
         public void run()
         {  //locking the resource
      synchronized (resource1)
         {
         System.out.println("Thread 1: locked resource 1");

         try {
                  Thread.sleep(100);
           }
        catch (Exception e)
                  {
                           System.out.println(e);
                  }

         synchronized (resource2)
            {
          System.out.println("Thread 1: locked resource 2");
         }
      }
    } //end of run()
  };  //end of t1

  // t2 tries to lock resource2 then resource1
  Thread t2 = new Thread()
  {
   public void run()
   {
     synchronized (resource2)
     {
      System.out.println("Thread 2: locked resource 2");

      try { Thread.sleep(100);} catch (Exception e) {}

      synchronized (resource1)
```

```
        {
          System.out.println("Thread 2: locked resource 1");
        }
      }
     }//end of run()
   };  //end of t2


   t1.start();
   t2.start();
  }
}
```

Output:



Input/Output: reading and writing data, java.io.package

There are two predefined packages in java that contain classes to perform I/O operations. These are **java.io.***, and **java.nio.***. The **java.io.*** used to perform reading and writing to console and reading and writing to the Files. The **java.nio.***, contains all the classes of java.io.*, and aslo contain the classes to perform advanced operations such as buffering, memory mapping, character encoding and decoding etc;. The java.io.* package provides separate classes for reading and writing data , these are **byte streams** and **character streams**.

Stream

A stream can be defined as a sequence or flow of data. There are two kinds of Streams.

- **InPutStream:** The InputStream is used to read data from a source.
- **OutPutStream:** the OutputStream is used for writing data to a destination.

Byte Streams

Java byte streams are used to perform input and output of 8-bit bytes. Though there are many classes related to byte streams but the most frequently used classes are , **FileInputStream** and **FileOutputStream**. Following is an example which makes use of these two classes to copy an input file into an output file:

CopyFile.java

```
import java.io.*;
publicclassCopyFile{
publicstaticvoid main(String args[])throwsIOException
{
FileInputStreamin=null;
FileOutputStreamout=null;

try{
in=newFileInputStream("input.txt");
out=newFileOutputStream("output.txt");
int c;
while((c =in.read())!=-1)
        {
        out.write(c);
}
} //end of try
finally{
if(in!=null)
        {
```

```
in.close();
}
if(out!=null)
        {
out.close();
}
}//end of finally
}//end of main
} //end of class
```

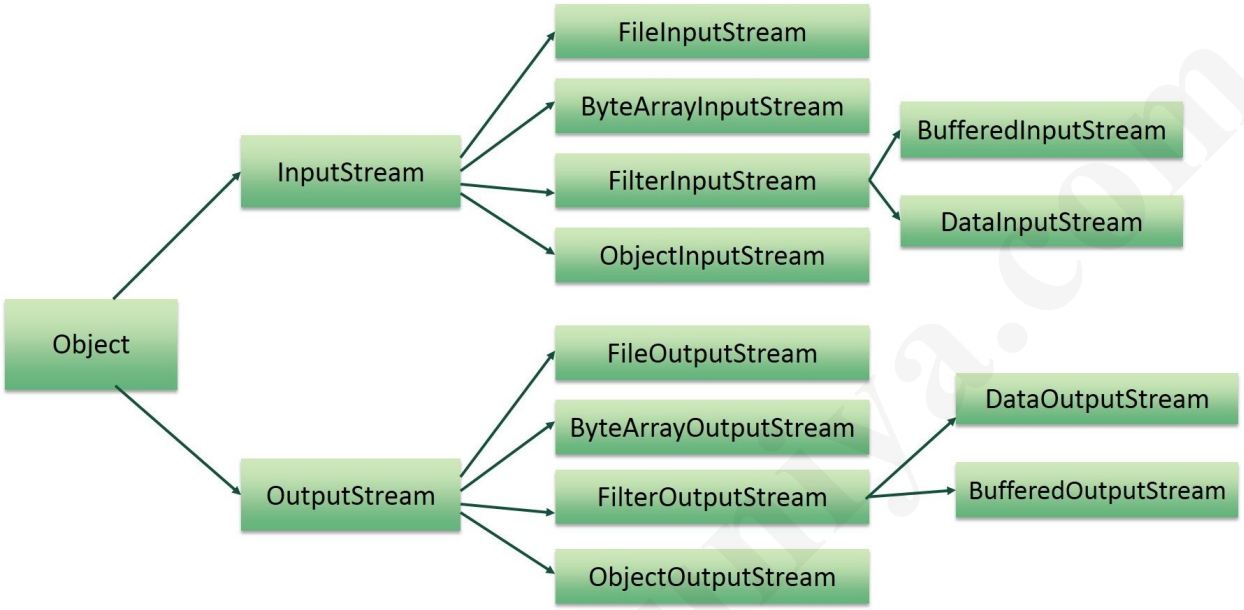
List of methods in **java.io.InputStream class** are as follow**:**

Method	**Description**
int available throws IOException	Returns the number of available bytes that can be read from input stream
void close() throws IOException	Closes the input stream
Void mark(int readlimit)	Makes the mark at the current position in the input stream. readlimit defines after how many lines this mark is nullified
Void markSupported()	Mark() method works if this method returns true
Abstract int read()	Used to read next byte from the input stream. It returns the byte, other wise -1 if EOF is encountered
Int read(byte []b)	Reads the byte and stores them in byte array b if return the true, otherwise -1 if EOF is encountered.
Int read(byte []b, int off, int len)	Reads the byte and stores them in byte array b upto the length from the offset off in b.
Void reset()	Resets the current pointer to the position set by the mark
Long skip(long n)	Skips the specified number of bytes from the input stream

List of methods in **java.io.OutputStream class** are as follow**:**

Method	**Description**
Void close()	Closes the output stream
Void flush()	Flushes the output stream
Void write(byte [] b)	Writes the contents of the byte array b to the output stream
Void write(byte [] b, int off, int len)	Writes the specified number of bytes (len) to the output stream starting from the offset off in b
Abstract void write(int b)	Abstract method to write to the output stream

Here is a hierarchy of classes to deal with Input and Output streams.



java.io.File class

The **Java.io.File** class is an abstract representation of file and directory pathnames. Following are the important points about File:

- Instances may or may not denote an actual file-system object such as a file or a directory. If it does denote such an object then that object resides in a partition. A partition is an operating system-specific portion of storage for a file system.
- A file system may implement restrictions to certain operations on the actual file-system object, such as reading, writing, and executing. These restrictions are collectively known as access permissions.
- Instances of the File class are immutable; that is, once created, the abstract pathname represented b

Methods of java.io.File class are as follow:

Method	Description
boolean canExecute()	This method tests whether the application can execute the file denoted by this abstract pathname.
boolean canRead()	This method tests whether the application can read the file denoted by this abstract pathname.
boolean canWrite()	This method tests whether the application can modify the file denoted by this abstract pathname.
boolean exists()	This method tests whether the file or directory denoted by this abstract pathname exists.

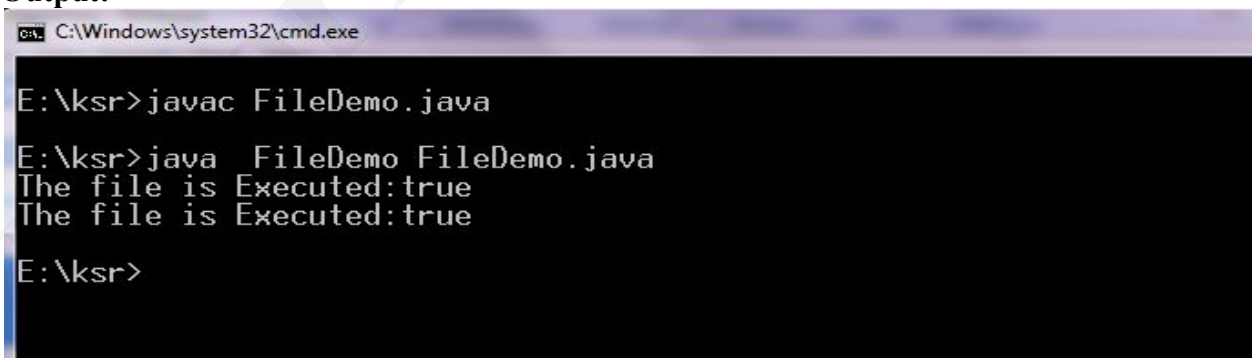
String getName()	This method returns the name of the file or directory denoted by this abstract pathname.
String getParent()	This method returns the pathname string of this abstract pathname's parent, or null if this pathname does not name a parent directory.
String getPath()	This method converts this abstract pathname into a pathname string.
String[] list()	This method returns an array of strings naming the files and directories in the directory denoted by this abstract pathname.
boolean isDirectory()	This method tests whether the file denoted by this abstract pathname is a directory.
boolean isFile()	This method tests whether the file denoted by this abstract pathname is a normal file.

Example program:

FileDemo.java

```
import java.io.File;
class FileDemo
{
public static void main(String args[])
{
        File f=new File(args[0]);
        System.out.println("The file is Executed:"+f.isFile());
        System.out.println("The file is Executed:"+f.canRead());
}
}
```

Output:



Reading and Writing using the Byte Stream

FIleInputStream and FileOutputStream classes are used to read and write respectively. The Constructor will be as follow:

FileInputStream fis=new FileInputStream("FileDemo.java");

The list of methods of **FileInputStream** are as follow:

SN	Methods with Description
1	**public void close() throws IOException{}** This method closes the file output stream. Releases any system resources associated with the file. Throws an IOException.
2	**protected void finalize()throws IOException {}** This method cleans up the connection to the file. Ensures that the close method of this file output stream is called when there are no more references to this stream. Throws an IOException.
3	**public int read(int r)throws IOException{}** This method reads the specified byte of data from the InputStream. Returns an int. Returns the next byte of data and -1 will be returned if it's end of file.
4	**public int read(byte[] r) throws IOException{}** This method reads r.length bytes from the input stream into an array. Returns the total number of bytes read. If end of file -1 will be returned.
5	**public int available() throws IOException{}** Gives the number of bytes that can be read from this file input stream. Returns an int.

Methods of FileOutputStream are as follow:

FileOutputStream is used to create a file and write data into it. The stream would create a file, if it doesn't already exist, before opening it for output.
The Constructor will be as follow:

FileOutputStream fos=new FileOutputStream("FileDemo1.java");

List of Methods of FileOutputStream are as Follow:

SN	Methods with Description
1	**public void close() throws IOException{}** This method closes the file output stream. Releases any system resources associated with the file. Throws an IOException
2	**protected void finalize()throws IOException {}** This method cleans up the connection to the file. Ensures that the close method of this file output stream is called when there are no more references to this stream. Throws an IOException.
3	**public void write(int w)throws IOException{}** This methods writes the specified byte to the output stream.
4	**public void write(byte[] w)** Writes w.length bytes from the mentioned byte array to the OutputStream.

Example Program:

```
import java.io.*;


publicclass fileStreamTest{


publicstaticvoid main(String args[]){


try{
byte bWrite []={11,21,3,40,5};
OutputStream os =newFileOutputStream("test.txt");
for(int x=0; x < bWrite.length ; x++){
        os.write( bWrite[x]);// writes the bytes
}
      os.close();


InputStreamis=newFileInputStream("test.txt");
int size =is.available();
```

```
for(int i=0; i< size; i++){
System.out.print((char)is.read()+"  ");
}
is.close();
}catch(IOException e){
System.out.print("Exception");
}
}
}
```

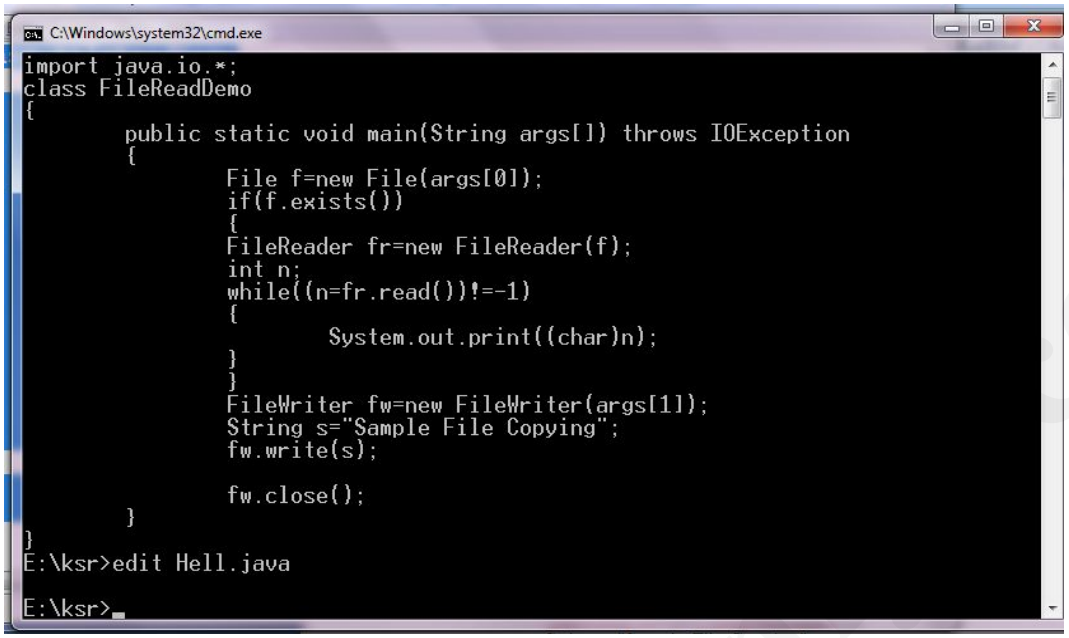
Reading and Writing Using Character Streams

Files can be read and written using character streams. The **FileReader** class used for reading contents of a file, and the **FileWriter** is used to write the contents to the file.

Example program:FileReadDemo.java

```
import java.io.*;
class FileReadDemo
{
        public static void main(String args[]) throws IOException
        {
                File f=new File(args[0]);
                if(f.exists())
                {
                FileReader fr=new FileReader(f);
                int n;
                while((n=fr.read())!=-1)
                {
                        System.out.print((char)n);
                }
                }
                FileWriter fw=new FileWriter(args[1]);
                String s="Sample File Copying";
                fw.write(s);
                fw.close();
        }
}
```

Output:
The "Hell.java" contains String contained in the String variable s.



Reading and Writing Using the Console (Scanner class)

The java.util package contains one particular class called Scanner class, which is used to read and write. A Snap shot of the Scanner class is as follows:

ScannerDemo.java

```
import java.util.*;
class ScannerDemo
{
	public static void main(String args[])
	{
		Scanner s=new Scanner(System.in);
		String name;
		int age;
		System.out.println("Enter your name:");
		name=s.nextLine();
		System.out.println("enter your age;");
		age=s.nextInt();
		System.out.println("Your name is:"+name);
		System.out.println("Your Age is:"+age);
	}
}
```

Unit 5
Applets and Event Handling

Topics:
Applets: Applet class, Applet structure, An example of Applet, Applet life Cycle, Event Delagation Model, Java.awt.event description, Sources of Events, Event Listeners, Adapter class, inner class

Part-1: Applet Programming

Introduction to Applet

Applets are small Java programs that are used in Internet computing. The Applets can be easily transported over the internet from one computer to another computer and can be run using "**appletviewer**" or java enabled "**Web Browser**". An Applet like any application program can do many things for us. It can perform arithmetic operations, display graphics, animations, text, accept the user input and play interactive games.

Applets are created in the following situations:

1. When we need something to be included dynamically in the web page.
2. When we require some flash output
3. When we want to create a program and make it available on the internet.

Types of Applet

There are two types of Applets.

- The first type is created is based on the **Applet** class of java.applet package. These applets use the **Abstract Window Toolkit(AWT)**for designing the graphical user interface.
- The second type of type of the Applets are based on the **Swing** class **JApplet.** The swing Applets use the swing classes to create Graphical User Interface.

The JApplet inherits the properties from the Applet, so all the features of the Applet are available in the JApplet.

Applet Basics

- All applets are subclasses (either directly or indirectly) of **Applet**. Applets are not stand-alone programs. Instead, they run within either a web browser or an applet viewer, which is provided by the JDK.
- Execution of an applet does not begin at **main( )**.
- Output to your applet's window is not performed by **System.out.println( )**. Rather, in non-Swing applets, output is handled with various AWT methods, such as **drawString( )**, which outputs a string to a specified X,Y location
- To use an applet, it is specified in an HTMLfile. One way to do this is by using the APPLET tag.**(HTML stands for Hyper Text Markup Language)**

- The applet will be executed by a Java-enabled web browser when it encounters the APPLET tag within the HTMLfile.
- To just test the applet it can be executed using the appletviewer. The applet tag must be included as comment lines in the java source program as follow:
  /*<applet code="applet_name" width=400 height=400 ></applet> */
- To view the applet using the HTML file, it can be included in the HTML file with <applet. Tag as follw:

Filename.HTML

```
<html>
<head><title> The name of the Web Page</title>
</head>
<body>
<applet code="applet_name"   width=400 height=400 ></applet>
</body>
</html>
```

Note: Here, the **<applet>** is the name of the tag, and "code" ,"width" and "height" are called attributes of the Tag, applet_name, 400, 400 are called values respectively.

The Applet class

The Applet class defines several methods that support for execution of the applets, such as starting and stopping. It also provides methods to load and display images. It also provides methods for loading and playing the audio clips. The **Applet** extends the AWT class **Panel**. The **Panel** extends **Container** class, which in turn extends from the **Component.**The applet will be executed by a Java-enabled web browser when it encounters the APPLET tag within the HTMLfile.

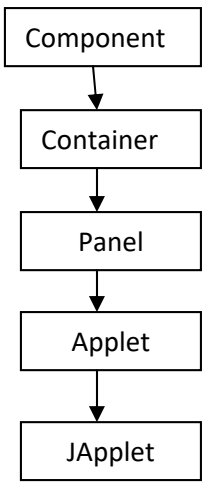


Fig 1: Hierarchy of Applet class

methods of Applet class:

destroy()
Called by the browser or applet viewer to inform this applet that it is being reclaimed and that it should destroy any resources that it has allocated.
getAppletContext()
Determines this applet's context, which allows the applet to query and affect the environment in which it runs.
getAppletInfo()
Returns information about this applet.
getAudioClip(URL)
Returns the AudioClip object specified by the URL argument.
getAudioClip(URL, String)
Returns the AudioClip object specified by the URL and name arguments.
getCodeBase()
Gets the base URL.
getDocumentBase()
Gets the document URL.
getImage(URL)
Returns an Image object that can then be painted on the screen.
getImage(URL, String)
Returns an Image object that can then be painted on the screen.
getLocale()
Gets the Locale for the applet, if it has been set.
getParameter(String)
Returns the value of the named parameter in the HTML tag.
getParameterInfo()
Returns information about the parameters than are understood by this applet.
init()
Called by the browser or applet viewer to inform this applet that it has been loaded into the system.
isActive()
Determines if this applet is active.
play(URL)
Plays the audio clip at the specified absolute URL.
play(URL, String)
Plays the audio clip given the URL and a specifier that is relative to it.
resize(Dimension)
Requests that this applet be resized.
resize(int, int)
Requests that this applet be resized.
setStub(AppletStub)
Sets this applet's stub.
showStatus(String)
Requests that the argument string be displayed in the "status window".

start()
Called by the browser or applet viewer to inform this applet that it should start its execution.

stop()
Called by the browser or applet viewer to inform this applet that it should stop its execution.

Applet Architecture

An applet is a window-based program. As such, its architecture is different from the console-basedprograms. The key concepts are as follow:

- **First, applets are event driven**. An applet waits until an event occurs. The run-timesystem notifies the applet about an event by calling an event handler that has been providedby the applet. Once this happens, the applet must take appropriate action and then quicklyreturn.
- **Second, the user initiates interaction with an applet**. These interactions are sent to the applet as events to which the applet must respond. For example, when the user clicks the mouse inside the applet's window, a mouse-clicked event is generated. If the user presses a key while the applet's window has input focus, a keypress event is generated

An Applet Skelton –An Example of Applet

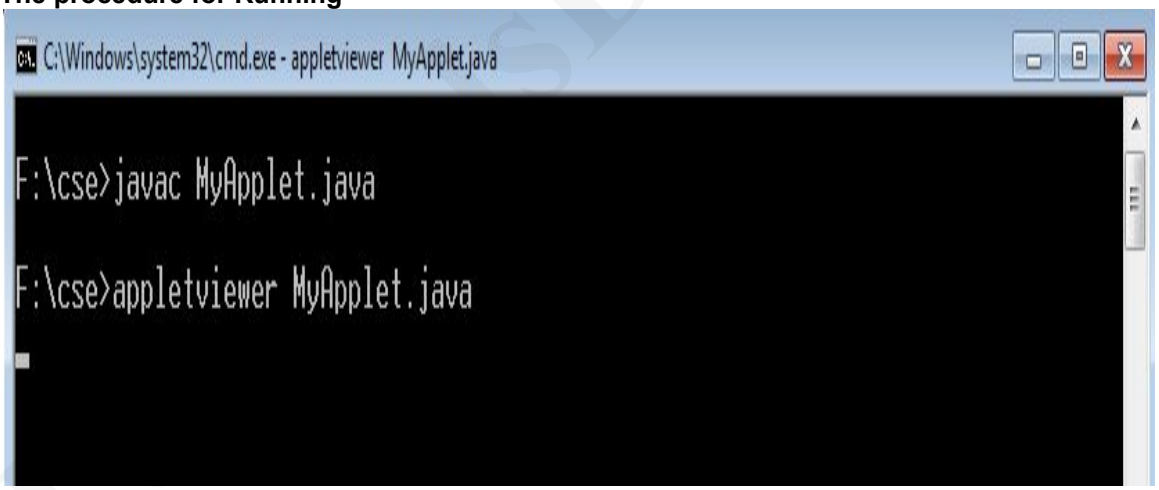
Most of the applets override a set of methods of the Applet. Four of these methods, **init( )**, **start( )**, **stop( )**, and **destroy( )**, apply to all applets and are defined by **Applet**. Default implementations for all of these methods are provided. Applets do not need to override those methods they do not use.

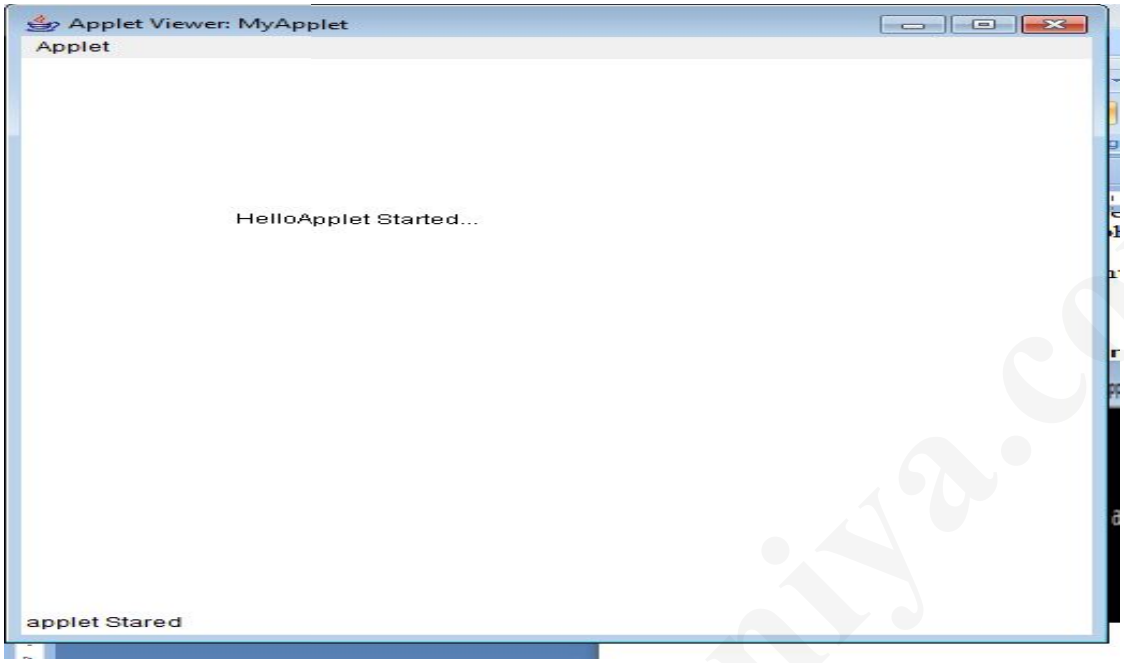
AWT-based applets will also override the **paint( )** method, which is defined by the AWT **Component** class. This method is called when the applet's output must be redisplayed. These five methods can be assembled into the skeleton shown here:

```
// 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 activities
}
// Called when an applet's window must be restored.
public void paint(Graphics g)
{
        // redisplay contents of window
}
}
```

The procedure for Running





Life Cycle of an Applet

It is important to understand the order in which the various methods shown in the skeletonare called. When an applet begins, the following methods are called, 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.

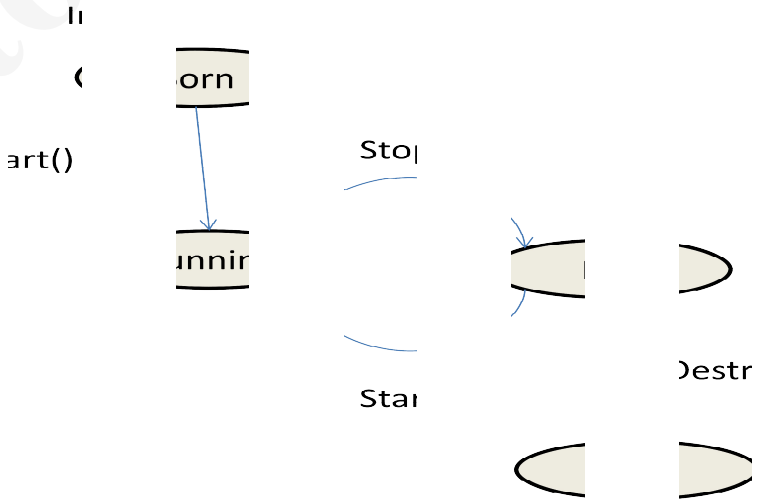


Fig 2: The Life Cycle of the Applet

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 beenstopped. Whereas **init( )** is called once—the first time an applet is loaded—**start( )** is calledeach time an applet's HTML document is displayed onscreen. So, if a user leaves a web pageand comes back, the applet resumes execution at **start( )**.

paint( )

The **paint( )** method is called each time your applet's output must be redrawn. This situationcan occur for several reasons. For example, the window in which the applet is running maybe overwritten by another window and then uncovered. Or the applet window may beminimized 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 usedwhenever output to the applet is required.

stop( )

The **stop( )** method is called when a web browser leaves the HTML document containing theapplet—when it goes to another page, for example. When **stop( )** is called, the applet isprobably running. You should use **stop( )** to suspend threads that don't need to run when theapplet 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 tobe removed completely from memory. At this point, you should free up any resources theapplet may be using. The **stop( )** method is always called before **destroy( )**.

Requesting the repaint() method

One of the important architectural constraints that have been imposed on an applet is that it must quickly return control to the AWT run-time system. It cannot create a loop inside paint( ). This would prevent control from passing back to the AWT. Whenever your applet needs to update theinformation displayed in its window, it simply calls ***repaint( )***. The ***repaint( )*** method is defined by the AWT that causes AWT run-time system to execute a call to your applet's ***update()*** method, which in turn calls ***paint()***. The AWT will then execute a call to

TutorialsDuniya.com

Download FREE Computer Science Notes, Programs, Projects, Books PDF for any university student of BCA, MCA, B.Sc, B.Tech CSE, M.Sc, M.Tech at https://www.tutorialsduniya.com

- Algorithms Notes
- Artificial Intelligence
- Android Programming
- C & C++ Programming
- Combinatorial Optimization
- Computer Graphics
- Computer Networks
- Computer System Architecture
- DBMS & SQL Notes
- Data Analysis & Visualization
- Data Mining
- Data Science
- Data Structures
- Deep Learning
- Digital Image Processing
- Discrete Mathematics
- Information Security
- Internet Technologies
- Java Programming
- JavaScript & jQuery
- Machine Learning
- Microprocessor
- Operating System
- Operational Research
- PHP Notes
- Python Programming
- R Programming
- Software Engineering
- System Programming
- Theory of Computation
- Unix Network Programming
- Web Design & Development

Please Share these Notes with your Friends as well

paint( ) that will display the stored information. The repaint( ) method has four forms. The simplest version of repaint( ) is:

1. **void repaint ( )**

This causes the entire window to be repainted. Other versions that will cause repaint are:

2. **void repaint(int left, int top, int width, int height)**

If your system is slow or busy, update( ) might not be called immediately. If multiple calls have been made to AWT within a short period of time, then update( ) is not called very frequently. This can be a problem in many situations in which a consistent update time is necessary. One solution to this problem is to use the following forms of **repaint( ):**

3. **void** repaint (long **maxDelay)**
4. **void** repaint (long **maxDelay, int x, int y, int width, int height)**

Where "maxDelay" is the number milliseconds should be elapsed before updat() method is called.

Using the Status Window

In addition to displaying information in its window, an applet can also output a messageto the status window of the browser or applet viewer on which it is running. To do so, call**showStatus( )** with the string that you want displayed. The status window is a good placeto give the user feedback about what is occurring in the applet, suggest options, or possiblyreport some types of errors. The status window also makes an excellent debugging aid,because it gives you an easy way to output information about your applet.

Example program

StatusWindow.java

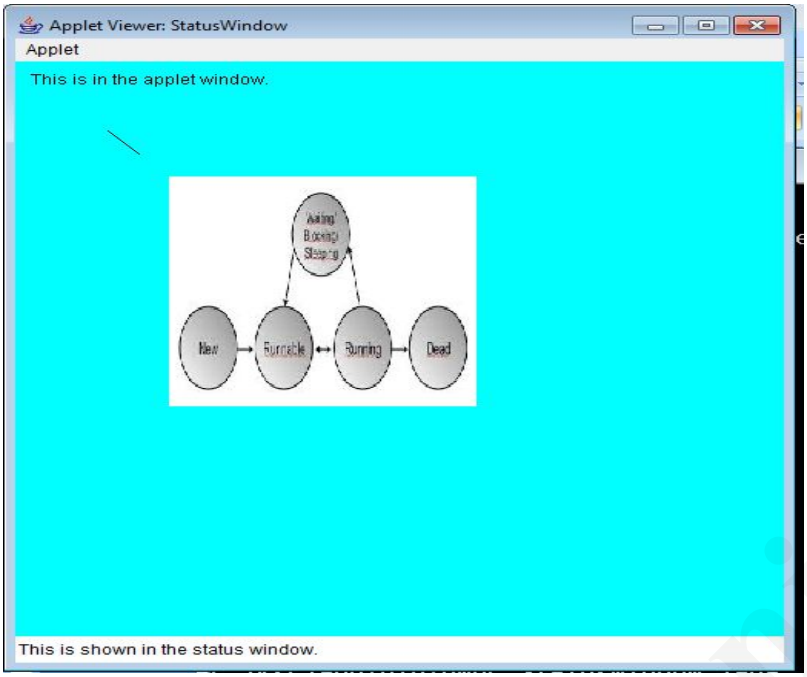
```
// Using the Status Window.
import java.awt.*;
```

```
import java.applet.*;
/*
<applet code="StatusWindow" width=500 height=500>
</applet>
*/
public class StatusWindow extends Applet
{
        Image img;
        public void init()
        {
        setBackground(Color.cyan);
        //initializing the image object
        img=getImage(getCodeBase(),"Life.jpg");
        }

        public void paint(Graphics g)
        {
        g.drawString("This is in the applet window.", 10, 20);
        showStatus("This is shown in the status window.");
        g.drawLine(60,60,80,80);
        //drawing the image on applet
        g.drawImage(img,100,100,200,200,this);
        }
}
```

Running applet:

1. javac StatusWindow.java
2. appletviewer StatusWindow.java

Output:



Passing parameters to Applet

We can supply user defined parameters to an applet using the **<param>** Tag of HTML. Each <param> Tag has the attributes such as "name" , "value" to which actual values are assigned. Using this tag we can change the text to be displayed through applet. We write the <param> Tag as follow:

<param name="name1" value="Hello Applet" > </param>

Passing parameters to an Applet is something similar to passing parameters to main() method using command line arguments. To set up and handle parameters, we need to do two things:

1. Include appropriate <param> Tag in the HTML file
2. Provide the code in the applet to take these parameters.

Example Program:

ParaPassing.java

```
import java.applet.*;
import java.awt.*;
public class ParaPassing extends Applet
{
      String str;
      public void init()
      {
        str=getParameter("name1");
        if(str==null)
```

```
            str="Java";
          str="Hello  "+str;
      }
      public void paint(Graphics g)
      {
            g.drawString(str,50,50);
      }
}
```

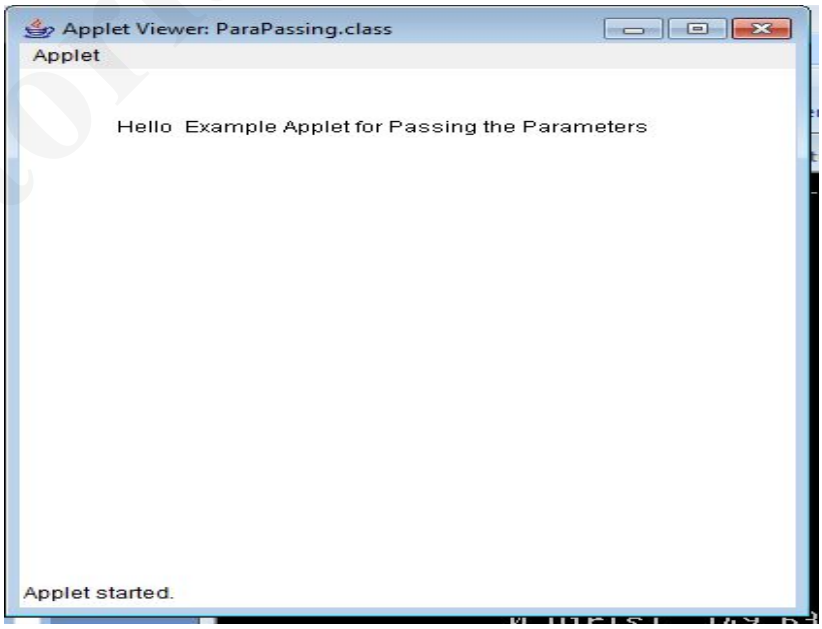
para.html

```
<html>
<head><title> Passing Parameters</title></head>
<body bgcolor=pink >
<param name="name1" value="Example Applet for Passing the Parameters" >
<applet code="ParaPassing.class"  width=400 height=400 >
</param>
</applet>
</body>
</html>
```

Running the Program:

1. Compile the "ParaPassing.java" using the "javac" command, which generates the "ParaPassing.class " file
2. Use "ParaPassing.class" file to code attribute of the <applet> Tag and save it as "para.html"
3. Give "para.html" as input to the "**appletviewer**" to see the output or open the file using the applet enabled web broser.

Output:



Getting the Input from the User

Applets work in the graphical environment. Therefore, applets treat inputs as text strings. we must first create an area of the screen in which user can type and edit input items. we can do this by using the **TextField** class of the applet package. Once text fields are created, user can enter and edit the content.

Next step is to retrieve the contents from the text fields for display of calculations, if any. The text fields contain the item in the form of String. They need to be converted to the right form, before they are used in any computations.

Example Program:

```
import java.applet.*;
import java.awt.*;
public class InputApplet extends Applet
{
	TextField text1,text2;
	Label l1,l2;
	public void init()
	{
		text1=new TextField(8);
		l1=new Label("Enter First No");
		text2=new TextField(8);
		l2=new Label("Enter second No");
		add(l1);
		add(text1);
		add(l2);
		add(text2);
		text1.setText("0");
		text2.setText("0");
	}	public void paint(Graphics g)
	{
		int x=0,y=0,z=0;
		String s1,s2,res;
		g.drawString("Enter a number in each text box",50,100);
		try
		{
			s1=text1.getText();
			x=Integer.parseInt(s1);
			s2=text2.getText();
			y=Integer.parseInt(s2);
		}
		catch(Exception e)
		{

		}
```

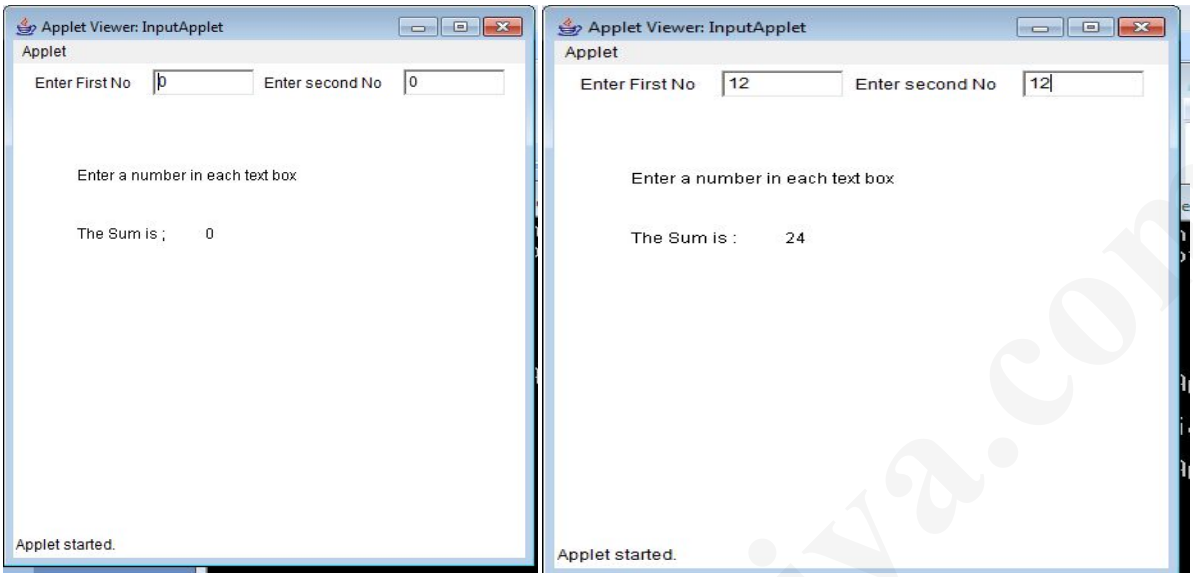
```
                    z=x+y;
                    res=String.valueOf(z);
                g.drawString("The Sum is :",50,150);
                g.drawString(res,150,150);
        }
        public boolean action(Event e,Object obj)
        {
                repaint();
                return true;
        }
}
```

sum.html

```
<html>

<head><title>Geting input from the User </title>

</head>

<body>

        <applet code="InputApplet.class" width=400 height=400 ></applet>

</body>

</html>
```

Output:

Before Entering the Input **After Entering the input**



Part –II: Event Handling

Types of Event Handling Mechanisms

There are two types of Event handling mechanisms supported by Java. First, the approach supported by Java 1.0, where the generated event is given hierarchically to the objects until it was handled. This can be also called as ***Hierarchical Event Handling Model***.

Second, the approach supported by Java 1.1, which registers the listeners to the source of the event and the registered listener processes the event and returns response to the source. This is called "***Event Delegation Model***"

The Event Delegation Model

The modern approach to handling events is based on the ***delegation event model***, which definesstandard and consistent mechanisms to generate and process ***events (1)***. Its concept is quite simple:a ***source (2)***generates an event and sends it to one or more ***listeners (3)***. In this scheme, the listenersimply waits until it receives an event. Once an event is received, the listener processes the eventand then returns response.

The ***advantage*** of this design is that the ***application logic*** that processes eventsis cleanly separated from the user ***interface logic*** that generates those events. Auser interfaceelement is able to "delegate" the processing of an event to a separate piece of code.

In the delegation event model, listeners must ***register*** with a source in order to receive anevent notification. This provides an important benefit: notifications are sent only to listenersthat want to receive them. This is a more efficient way to handle events than the design usedby the old Java 1.0 approach. Previously, an event was propagated up the containment hierarchyuntil it was handled by a component. This required components to receive events that they didnot process, and it wasted valuable time. The delegation event model eliminates this overhead.

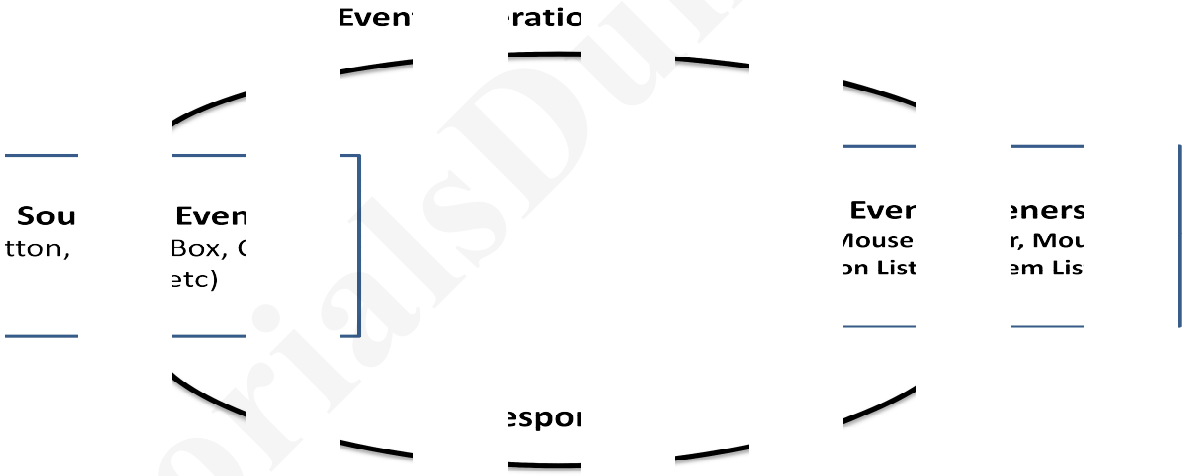


Fig 3: The Event Delegation Model

1. What is an Event?

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.

Examples: Pressing a button, entering a character via the keyboard, selecting an item in a list, and clicking the mouse etc..

2. 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 add*Type*Listener(*Type*Listener *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 listener 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 that register to receive them.
Some sources may allow only one listener to register. The general form of such a methodis this:

public void add*Type*Listener(*Type*Listener *el*) throws java.util.TooManyListenersException

Here, ***Type***is the name of the event, and ***el***is a reference to the event listener. When suchan 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 **unregister** an interestin a specific type of event. The general form of such a method is this:

public void remove*Type*Listener(*Type*Listener *el*)

Here, ***Type***is the name of the event, and *el* is a reference to the event listener. For example,to remove a keyboard listener, you would call **removeKeyListener( )**.The methods that add or remove listeners are provided by the source that generatesevents. For example, the **Component** class provides methods to add and remove keyboardand mouse event listeners.

3. Event Listeners

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 toreceive notifications when the mouse is dragged or moved.

The Event classes

The Event classes represent the event. Java provides us various Event classes but we will discuss those which are more frequently used.

EventObject class

It is the root class from which all event state objects shall be derived. All Events are constructed with a reference to the object, the **source**, that is logically deemed to be the object upon which the Event in question initially occurred upon. This class is defined in java.util package.

The summary

- **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 many types of events that are generated by varioususer interface elements. Table 1 shows several commonly used event classes and providesa brief description of when they are generated. Commonly used constructors and methods ineach class are described in the following sections

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 superclass for all component input event classes.
ItemEvent	Generated when a check box 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.
MouseWheelEvent	Generated when the mouse wheel is moved.
TextEvent	Generated when the value of a text area or text field is changed.
WindowEvent	Generated when a window is activated, closed, deactivated, deiconified, iconified, opened, or quit.

Table 1:List of Event Classes

1. **The ActionEvent class**

The ActionEvent class defines 5integer constant and also defines 3 methods.

Integer constants:(instance variable)

ALT_MASK, **CTRL_MASK**,**META_MASK**, and **SHIFT_MASK** ,these are used to identify any modifier associated with an action event.

Methods:

i. ***String getActionCommand()*** - this is used to obtain the command name. For example when a button is pressed, the button contains the label, this method is used to obtain that.
ii. ***int getModifiers()*** - returns a value that indicates which modifier keys (ALT, CTRL,META, and/or SHIFT) were pressed when the event was generated.
iii. ***long getWhen()*** -used to get the time when the event took place.

2. The AdjustmentEvent Class

Integer Constants:

An **AdjustmentEvent** is generated by a scroll bar. There are five types of adjustment events. The **AdjustmentEvent** class defines integer constants that can be used to identify them. The constants and their meanings are shown here:

BLOCK_DECREMENT	The user clicked inside the scroll bar to decrease its value.
BLOCK_INCREMENT	The user clicked inside the scroll bar to increase its value.
TRACK	The slider was dragged.
UNIT_DECREMENT	The button at the end of the scroll bar was clicked to decrease its value.
UNIT_INCREMENT	The button at the end of the scroll bar was clicked to increase its value.

Methods:

i. ***Adjustable getAdjustable()*** - returns the object that generated the event.
ii. ***int getAdjustmentType()*** - the type of adjustment event can be obtained by this method.
iii. ***int getValue()*** -the amount of adjustment can be obtained by this method

3. The ComponentEvent Class

A**ComponentEvent** 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 their meanings are shown here:

COMPONENT_HIDDEN	The component was hidden.
COMPONENT_MOVED	The component was moved.
COMPONENT_RESIZED	The component was resized.
COMPONENT_SHOWN	The component became visible.

Method:

Component getComponent( ) -this is used to return the component that has generated the event.

4. 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.

Integer Constants:

COMPONENT_ADDED and **COMPONENT_REMOVED**.

The **ContainerEvent** class defines **int** constants that can be used to identify them.

Methods:

i. ***container getContainer()*** - used to get thereference of the container that has generated this event

ii. ***Component getChild()*** - used to get the component that has been added or removed.

5. The FocusEvent Class

A **FocusEvent** is generated when a component gains or loses input focus.

Integer Constants:

FOCUS_GAINED and **FOCUS_LOST**.

Methods:

For example, assume that the focus is in a text field. If the user moves the mouse to adjust a scroll bar, the focus is temporarily lost.) The other component involved in the focus change, called the ***opposite component***, is passed in *other*.

i. ***Component getOppositeComponent( )***- used to determine the other opposite component.

ii. ***boolean isTemporary( )***- used to determine focus is changed temporarily

6. The InputEvent Class

The abstract class **InputEvent** is a subclass of **ComponentEvent** and is the superclass for component input events. Its subclasses are **KeyEvent** and **MouseEvent**.

Integer Constants:

ALT_DOWN_MASK	BUTTON2_DOWN_MASK	META_DOWN_MASK
ALT_GRAPH_DOWN_MASK	BUTTON3_DOWN_MASK	SHIFT_DOWN_MASK
BUTTON1_DOWN_MASK	CTRL_DOWN_MASK	

Methods:these are used determine which key is pressed

i. boolean isAltDown( ) -

ii. boolean isAltGraphDown( )

iii. boolean isControlDown( )

iv. boolean isMetaDown( )

v. boolean isShiftDown( )

7. The ItemEvent Class

An **ItemEvent** is generated when a check box or a list item is clicked or when a checkable menu item is selected or deselected.

Integer Constants:

DESELECTED	The user deselected an item.
SELECTED	The user selected an item.

Methods:

i. ***Object getItem( )*** –used to get the reference of the item that has generated the event
ii. ***ItemSelectable getItemSelectable( )***- used to get reference to the selectable items.
iii. ***int getStateChange( )***- return the state change for the event.

8. 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_RELEASED**, 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. Remember, not all keypresses result in characters. For example, pressing SHIFT does not generate a character.

Integer Constants:

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 equivalents of the numbers andletters. Here are some others:

VK_ALT	VK_DOWN	VK_LEFT	VK_RIGHT
VK_CANCEL	VK_ENTER	VK_PAGE_DOWN	VK_SHIFT
VK_CONTROL	VK_ESCAPE	VK_PAGE_UP	VK_UP

Methods:

The **KeyEvent** class defines several methods, but the most commonly used ones are **getKeyChar( )**, which returns the character that was entered, and **getKeyCode( )**, which returns the key code. Their general forms are shown here:

char getKeyChar( )
int getKeyCode( )

9. The MouseEvent Class

There are eight types of mouse events. The **MouseEvent** class defines the following integerconstants that can be used to identify them:

Integer Constants:

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_WHEEL	The mouse wheel was moved.

Methods:

Two commonly used methods in this class are **getX( )** and **getY( )**. These return the X andY coordinates of the mouse within the component when the event occurred. Their forms are shown here:

i. **int getX( ) –** used to get the X coordinate
ii. **int getY( ) –**used to get the Y coordinate
iii. Point getPoint( ) –used to obtain the coordinates of the mouse
iv. **getClickCount( ) -** method obtains the number of mouse clicks for this event. Its signature is shown here:
v. **isPopupTrigger() -** method tests if this event causes a pop-up menu to appear on thisplatform

Java SE 6 added three methods to **MouseEvent** that obtain the coordinates of the mouse relative to the screen rather than the component. They are shown here:

Point getLocationOnScreen( )
int getXOnScreen( )
int getYOnScreen( )

10. The MouseWheelEvent Class

The **MouseWheelEvent** class encapsulates a mouse wheel event. It is a subclass of **MouseEvent**.Not all mice have wheels. If a mouse has a wheel, it is located between the left and rightbuttons. Mouse wheels are used for scrolling. **MouseWheelEvent** defines these two integerconstants:

WHEEL_BLOCK_SCROLL	A page-up or page-down scroll event occurred.
WHEEL_UNIT_SCROLL	A line-up or line-down scroll event occurred.

Methods:

MouseWheelEvent defines methods that give you access to the wheel event. To obtain the number of rotational units, call **getWheelRotation( )**, shown here:

int getWheelRotation( )

It returns the number of rotational units. If the value is positive, the wheel moved counterclockwise. If the value is negative, the wheel moved clockwise. To obtain the type of scroll, call **getScrollType( )**, shown next:

int getScrollType( )

It returns either **WHEEL_UNIT_SCROLL** or **WHEEL_BLOCK_SCROLL**. If the scroll type is **WHEEL_UNIT_SCROLL**, you can obtain the number of units to scroll by calling **getScrollAmount( )**. It is shown here:

int getScrollAmount( )

11. The TextEvent Class

Instances of this class describe text events. These are generated by text fields and text areaswhen characters are entered by a user or program.
TextEvent defines the integer constant:

TEXT_VALUE_CHANGED.
The one constructor for this class is shown here:

TextEvent(Object *src*, int *type*)

Here, *src* is a reference to the object that generated this event. The type of the event isspecified by *type*.

12. The WindowEvent Class

There are ten types of window events. The**WindowEvent** class defines integer constants that can be used to identify them. The constants and their meanings are shown here:

Integer Constants:

WINDOW_ACTIVATED	The window was activated.
WINDOW_CLOSED	The window has been closed.
WINDOW_CLOSING	The user requested that the window be closed.
WINDOW_DEACTIVATED	The window was deactivated.
WINDOW_DEICONIFIED	The window was deiconified.
WINDOW_GAINED_FOCUS	The window gained input focus.
WINDOW_ICONIFIED	The window was iconified.
WINDOW_LOST_FOCUS	The window lost input focus.
WINDOW_OPENED	The window was opened.
WINDOW_STATE_CHANGED	The state of the window changed.

Methods:
Acommonly used method in this class is **getWindow( )**. It returns the **Window** object that generated the event. Its general form is shown here:

Window getWindow( )

WindowEvent also defines methods that return the opposite window (when a focus or activation event has occurred), the previous window state, and the current window state. These methods are shown here:

Window getOppositeWindow( )
int getOldState( )
int getNewState( )

Event Sources

The following are the event source classes, that actually generate the event.

Event Source	Description
Button	Generates action events when the button is pressed.
Check box	Generates item events when the check box is selected or deselected.
Choice	Generates item events when the choice is changed.
List	Generates action events when an item is double-clicked; generates item events when an item is selected or deselected.
Menu Item	Generates action events when a menu item is selected; generates item events when a checkable menu item is selected or deselected.
Scroll bar	Generates adjustment events when the scroll bar is manipulated.
Text components	Generates text events when the user enters a character.
Window	Generates window events when a window is activated, closed, deactivated, deiconified, iconified, opened, or quit.

Event Listener Interfaces

As explained, the delegation event model has two parts: sources and listeners. Listeners arecreated by implementing one or more of the interfaces defined by the **java.awt.event** package.When an event occurs, the event source invokes the appropriate method defined by thelistener and provides an event object as its argument.

Interface	Description
ActionListener	Defines one method to receive action events.
AdjustmentListener	Defines one method to receive adjustment events.
ComponentListener	Defines four methods to recognize when a component is hidden, moved, resized, or shown.
ContainerListener	Defines two methods to recognize when a component is added to or removed from a container.
FocusListener	Defines two methods to recognize when a component gains or loses keyboard focus.
ItemListener	Defines one method to recognize when the state of an item changes.
KeyListener	Defines three methods to recognize when a key is pressed, released, or typed.
MouseListener	Defines five methods to recognize when the mouse is clicked, enters a component, exits a component, is pressed, or is released.
MouseMotionListener	Defines two methods to recognize when the mouse is dragged or moved.
MouseWheelListener	Defines one method to recognize when the mouse wheel is moved.
TextListener	Defines one method to recognize when a text value changes.
WindowFocusListener	Defines two methods to recognize when a window gains or loses input focus.
WindowListener	Defines seven methods to recognize when a window is activated, closed, deactivated, deiconified, iconified, opened, or quit.

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*)

The AdjustmentListener Interface

This interface defines the **adjustmentValueChanged( )** method that is invoked when anadjustment event occurs. Its general form is shown here:

void adjustmentValueChanged(AdjustmentEvent *ae*)

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*)

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)

The 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(FocusEvent fe)
void focusLost(FocusEvent fe)

The ItemListener Interface

This interface defines the **itemStateChanged( )** method that is invoked when the state of anitem changes. Its general form is shown here:

void itemStateChanged(ItemEvent ie)

The KeyListener Interface

This interface defines three methods. The **keyPressed( )** and **keyReleased( )** methods areinvoked when a key is pressed and released, respectively. The **keyTyped( )** method is invokedwhen 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 eventsare 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*)

The MouseListener Interface

This interface defines five methods. If the mouse is pressed and released at the same point,**mouseClicked( )** is invoked. When the mouse enters a component, the **mouseEntered( )**method is called. When it leaves, **mouseExited( )** is called. The **mousePressed( )** and**mouseReleased( )** 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*)

The MouseMotionListener Interface
This interface defines two methods. The **mouseDragged( )** method is called multiple timesas the mouse is dragged. The **mouseMoved( )** method is called multiple times as the mouseis moved. Their general forms are shown here:

void mouseDragged(MouseEvent *me*)
void mouseMoved(MouseEvent *me*)

The MouseWheelListener Interface
This interface defines the **mouseWheelMoved( )** method that is invoked when the mousewheel is moved. Its general form is shown here:

void mouseWheelMoved(MouseWheelEvent *mwe*)

The TextListener Interface

This interface defines the **textChanged( )** method that is invoked when a change occurs in a text area or text field. Its general form is shown here:

void textChanged(TextEvent *te*)

The WindowFocusListener Interface

This interface defines two methods: **windowGainedFocus( )** and **windowLostFocus( )**. These are called when a window gains or loses input focus. Their general forms are shown here:

void windowGainedFocus(WindowEvent *we*)
void windowLostFocus(WindowEvent *we*)

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 windowis 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 windowDeactivated(WindowEvent *we*)
void windowDeiconified(WindowEvent *we*)
void windowIconified(WindowEvent *we*)
void windowOpened(WindowEvent *we*)

Using the Delegation Model for Handling the Mouse Events

To handle mouse events, you must implement the **MouseListener** and the **MouseMotionListener**interfaces. It displays the current coordinatesof the mouse in the applet's status window. Each time a button is pressed, the word "Down"is displayed at the location of the mouse pointer. Each time the button is released, the word"Up" is shown. If a button is clicked, the message "Mouse clicked" is displayed in the upperleftcorner of the applet display area

As the mouse enters or exits the applet window, a message is displayed in the upper-leftcorner of the applet display area. When dragging the mouse, a * is shown, which tracks withthe mouse pointer as it is dragged. Notice that the two variables, **mouseX** and **mouseY**, storethe location of the mouse when a mouse pressed, released, or dragged event occurs. Thesecoordinates are then used by **paint( )** to display output at the point of these occurrences.

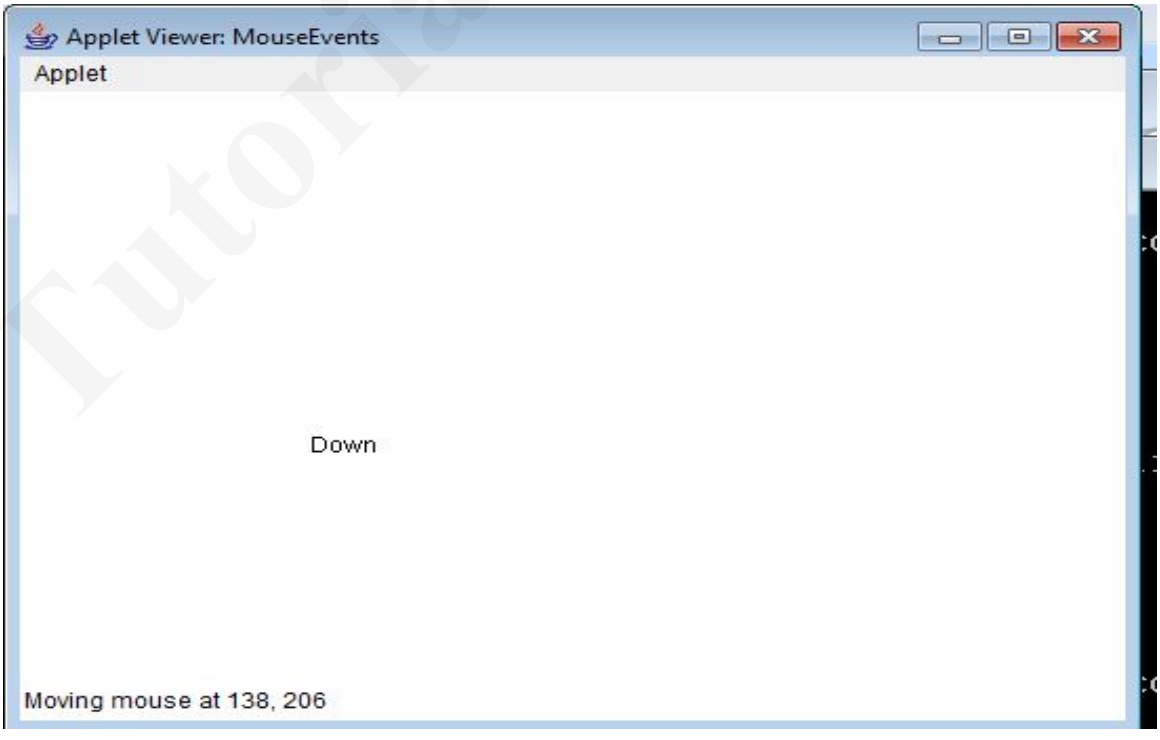
Example program for Handling Mouse Events

```
// Demonstrate the mouse event handlers.
import java.awt.*;
import java.awt.event.*;
import java.applet.*;
/*
<applet code="MouseEvents" width=300 height=100>
</applet>
*/
public class MouseEvents extends Applet
implements MouseListener, MouseMotionListener
{
String msg = "";
int mouseX = 0, mouseY = 0; // coordinates of mouse
public void init()
{
     addMouseListener(this);
```

```
        addMouseMotionListener(this);
    }
    // Handle mouse clicked.
    public void mouseClicked(MouseEvent me)
    {
    // save coordinates
        mouseX = 0;
        mouseY = 10;
        msg = "Mouse clicked.";
        repaint();
    }
    // Handle mouse entered.
    public void mouseEntered(MouseEvent me)
    {
    // save coordinates
        mouseX = 0;
        mouseY = 10;
        msg = "Mouse entered.";
        repaint();
    }
    // Handle mouse exited.
    public void mouseExited(MouseEvent me)
    {
    // save coordinates
        mouseX = 0;
        mouseY = 10;
        msg = "Mouse exited.";
        repaint();
    }
    // Handle button pressed.
    public void mousePressed(MouseEvent me)
    {
    // save coordinates
        mouseX = me.getX();
        mouseY = me.getY();
        msg = "Down";
        repaint();
    }
    // Handle button released.
    public void mouseReleased(MouseEvent me)
    {
    // save coordinates
        mouseX = me.getX();
        mouseY = me.getY();
        msg = "Up";
        repaint();
    }
```

```
// Handle mouse dragged.
public void mouseDragged(MouseEvent me)
{
// save coordinates
     mouseX = me.getX();
     mouseY = me.getY();
     msg = "*";
     showStatus("Dragging mouse at " + mouseX + ", " + mouseY);
     repaint();
}
// Handle mouse moved.
public void mouseMoved(MouseEvent me)
{
// show status
     showStatus("Moving mouse at " + me.getX() + ", " +
     me.getY());
}
// Display msg in applet window at current X,Y location.
public void paint(Graphics g)
{
     g.drawString(msg, mouseX, mouseY);
}
}
```

OutPut:



Adapter Classes

Java provides a special feature, called an ***adapter class***, that can simplify the creation of eventhandlers in certain situations. An adapter class provides an empty implementation of allmethods in an event listener interface. Adapter classes are **useful** when you want to receiveand process only ***some of the events*** that are handled by a particular event listener interface.You can define a new class to act as an event listener by extending one of the adapter classesand implementing only those events in which you are interested.

For example, the **MouseMotionAdapter** class has two methods, **mouseDragged( )**and **mouseMoved( )**, which are the methods defined by the **MouseMotionListener**interface. If you were interested in only mouse drag events, then you could simply extend**MouseMotionAdapter** and override **mouseDragged( )**. The empty implementation of**mouseMoved( )** would handle the mouse motion events for you.

The Following table provide the commonly used adapter classes:

Adapter Class	Listener Interface
ComponentAdapter	ComponentListener
ContainerAdapter	ContainerListener
FocusAdapter	FocusListener
KeyAdapter	KeyListener
MouseAdapter	MouseListener
MouseMotionAdapter	MouseMotionListener
WindowAdapter	WindowListener

Example program using the Adapter Class

```
// Demonstrate an adapter.
import java.awt.*;
import java.awt.event.*;
import java.applet.*;
/*
<applet code="AdapterDemo" width=300 height=100>
</applet>
*/
public class AdapterDemo extends Applet
{
public void init()
{
```

```
        addMouseListener(new MyMouseAdapter(this));
        addMouseMotionListener(new MyMouseMotionAdapter(this));
}
}
class MyMouseAdapter extends MouseAdapter
{
AdapterDemo adapterDemo;
public MyMouseAdapter(AdapterDemo adapterDemo)
{
        this.adapterDemo = adapterDemo;
}
// Handle mouse clicked.
public void mouseClicked(MouseEvent me)
{
        adapterDemo.showStatus("Mouse clicked");
}
}
class MyMouseMotionAdapter extends MouseMotionAdapter
{
        AdapterDemo adapterDemo;
        public MyMouseMotionAdapter(AdapterDemo adapterDemo)
        {
                this.adapterDemo = adapterDemo;
        }
// Handle mouse dragged.
public void mouseDragged(MouseEvent me)
{
        adapterDemo.showStatus("Mouse dragged");
}
}
```

Inner Classes

An *inner class* is a class defined within another class, or even within an expression.

Example program:

```
// Inner class demo.
import java.applet.*;
import java.awt.event.*;
/*
<applet code="InnerClassDemo" width=200 height=100>
</applet>
*/
public class InnerClassDemo extends Applet

{
```

```
public void init()
      {
            addMouseListener(new MyMouseAdapter());
      }

//inner class
class MyMouseAdapter extends MouseAdapter
{
      //overriding the mousePressed() method
      public void mousePressed(MouseEvent me)
      {
            showStatus("Mouse Pressed");
      }
}
}
```

OutPut:

********** End of 5th Unit **********

TutorialsDuniya.com

Download FREE Computer Science Notes, Programs, Projects, Books PDF for any university student of BCA, MCA, B.Sc, B.Tech CSE, M.Sc, M.Tech at https://www.tutorialsduniya.com

- Algorithms Notes
- Artificial Intelligence
- Android Programming
- C & C++ Programming
- Combinatorial Optimization
- Computer Graphics
- Computer Networks
- Computer System Architecture
- DBMS & SQL Notes
- Data Analysis & Visualization
- Data Mining
- Data Science
- Data Structures
- Deep Learning
- Digital Image Processing
- Discrete Mathematics
- Information Security
- Internet Technologies
- Java Programming
- JavaScript & jQuery
- Machine Learning
- Microprocessor
- Operating System
- Operational Research
- PHP Notes
- Python Programming
- R Programming
- Software Engineering
- System Programming
- Theory of Computation
- Unix Network Programming
- Web Design & Development

Please Share these Notes with your Friends as well