JAVA

Java Basics **Java - Home** **Java - Overview**

Java - Environment Setup
Java - Basic Syntax
Java - Object & Classes
Java - Basic Datatypes
Java - Variable Types
Java - Modifier Types
Java - Basic Operators
Java - Loop Control
Java - Decision Making
Java - Numbers
Java - Characters
Java - Strings
Java - Arrays
Java - Date & Time
Java - Regular Expressions
Java - Methods
Java - Files and I/O
Java - Exceptions

Java Object Oriented

Java - Inheritance
Java - Overriding
Java - Polymorphism
Java - Abstraction
Java - Encapsulation
Java - Interfaces
Java - Packages

Java Advanced

Java - Data Structures
Java - Collections
Java - Generics
Java - Serialization
Java - Networking
Java - Sending Email
Java - Multithreading
Java - Applet Basics
Java – Documentation

Java Useful Resources Java - Quick Guide

JAVA

Java – Overview

Java programming language was originally developed by Sun Microsystems, which was initiated by James Gosling and released in 1995 as core component of Sun Microsystems.s Java platform (Java 1.0 [J2SE]).

As of December 08 the latest release of the Java Standard Edition is 6 (J2SE). With the advancement of Java and its wide spread popularity, multiple configurations were built to suite various types of platforms. Ex: J2EE for Enterprise Applications, J2ME for Mobile Applications.

Sun Microsystems has renamed the new J2 versions as Java SE, Java EE and Java ME respectively. Java is guaranteed to be **Write Once, Run Anywhere**

Java is:

- **Object Oriented :** In java everything is an Object. Java can be easily extended since it is based on the Object model.
- **Platform independent:** Unlike many other programming languages including C and C++ when Java is compiled, it is not compiled into platform specific machine, rather into platform independent byte code. This byte code is distributed over the web and interpreted by virtual Machine (JVM) on whichever platform it is being run.
- **Simple :**Java is designed to be easy to learn. If you understand the basic concept of OOP java would be easy to master.
- **Secure :** With Java's secure feature it enables to develop virus-free, tamper-free systems. Authentication techniques are based on public-key encryption.
- **Architectural- neutral :**Java compiler generates an architecture-neutral object file format which makes the compiled code to be executable on many processors, with the presence Java runtime system.
- **Portable :**being architectural neutral and having no implementation dependent aspects of the specification makes Java portable. Compiler and Java is written in ANSI C with a clean portability boundary which is a POSIX subset.
- **Robust :**Java makes an effort to eliminate error prone situations by emphasizing mainly on compile time error checking and runtime checking.
- **Multi-threaded :** With Java's multi-threaded feature it is possible to write programs that can do many tasks simultaneously. This design feature allows developers to construct smoothly running interactive applications.
- **Interpreted :**Java byte code is translated on the fly to native machine instructions and is not stored anywhere. The development process is more rapid and analytical since the linking is an incremental and light weight process.
- **High Performance:** With the use of Just-In-Time compilers Java enables high performance.
- **Distributed :**Java is designed for the distributed environment of the internet.
- **Dynamic :** Java is considered to be more dynamic than C or C++ since it is designed to adapt to an evolving environment. Java programs can carry extensive amount of run-time information that can be used to verify and resolve accesses to objects on run-time.

JAVA

History of Java:

James Gosling initiated the Java language project in June 1991 for use in one of his many set-top box projects. The language, initially called Oak after an oak tree that stood outside Gosling's office, also went by the name Green and ended up later renamed as Java, from a list of random words.

Sun released the first public implementation as Java 1.0 in 1995. It promised **Write Once, Run Anywhere** (WORA), providing no-cost run-times on popular platforms.

On 13 November 2006, Sun released much of Java as free and open source software under the terms of the GNU General Public License (GPL).

On 8 May 2007 Sun finished the process, making all of Java's core code free and open-source, aside from a small portion of code to which Sun did not hold the copyright.

Tools you will need:

For performing the examples discussed in this tutorial, you will need a Pentium 200-MHz computer with a minimum of 64 MB of RAM (128 MB of RAM recommended).

You also will need the following softwares:

- Linux 7.1 or Windows 95/98/2000/XP operating system.
- Java JDK 5
- Microsoft Notepad or any other text editor

This tutorial will provide the necessary skills to create GUI, networking, and Web applications using Java

Java Basic Syntax

When we consider a Java program it can be defined as a collection of objects that communicate via invoking each others methods. Let us now briefly look into what do class, object, methods and instance variables mean.

- **Object -** Objects have states and behaviors. Example: A dog has states-color, name, breed as well as behaviors -wagging, barking, eating. An object is an instance of a class.
- **Class -** A class can be defined as a template/ blue print that describe the behaviors/states that object of its type support.
- **Methods -** A method is basically a behavior. A class can contain many methods. It is in methods where the logics are written, data is manipulated and all the actions are executed.
- **Instance Variables -** Each object has its unique set of instance variables. An object's state is created by the values assigned to these instance variables.

First Java Program:

Let us look at a simple code that would print the words *Hello World*.

```
public class MyFirstJavaProgram {
```

```
   /* This is my first java program.
    * This will print 'Hello World' as the output
    */

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

Lets look at how to save the file, compile and run the program. Please follow the steps given below:

- Open notepad and add the code as above.
- Save the file as : MyFirstJavaProgram.java.
- Open a command prompt window and go o the directory where you saved the class. Assume its C:\.
- Type ' javac MyFirstJavaProgram.java ' and press enter to compile your code. If there are no errors in your code the command prompt will take you to the next line.(Assumption : The path variable is set).
- Now type ' java MyFirstJavaProgram ' to run your program.
- You will be able to see ' Hello World ' printed on the window.

```
C : > javac MyFirstJavaProgram.java
C : > java MyFirstJavaProgram
Hello World
```

Basic Syntax:

About Java programs, it is very important to keep in mind the following points.

- **Case Sensitivity -** Java is case sensitive which means identifier **Hello** and **hello** would have different meaning in Java.
- **Class Names -** For all class names the first letter should be in Upper Case.

  If several words are used to form a name of the class each inner words first letter should be in Upper Case.

  Example *class MyFirstJavaClass*
- **Method Names -** All method names should start with a Lower Case letter.

  If several words are used to form the name of the method, then each inner word's first letter should be in Upper Case.

  Example *public void myMethodName()*
- **Program File Name -** Name of the program file should exactly match the class name.

  When saving the file you should save it using the class name (Remember java is case sensitive) and append '.java' to the end of the name. (if the file name and the class name do not match your program will not compile).

Example : Assume 'MyFirstJavaProgram' is the class name. Then the file should be saved as *'MyFirstJavaProgram.java'*

- **public static void main(String args[]) -** java program processing starts from the main() method which is a mandatory part of every java program..

Java Identifiers:

All java components require names. Names used for classes, variables and methods are called identifiers.

In java there are several points to remember about identifiers. They are as follows:

- All identifiers should begin with a letter (A to Z or a to z), currency character ($) or an underscore (_).
- After the first character identifiers can have any combination of characters.
- A key word cannot be used as an identifier.
- Most importantly identifiers are case sensitive.
- Examples of legal identifiers:age, $salary, _value, __1_value
- Examples of illegal identifiers : 123abc, -salary

Java Modifiers:

Like other languages it is possible to modify classes, methods etc by using modifiers. There are two categories of modifiers.

- **Access Modifiers :** default, public , protected, private
- **Non-access Modifiers :** final, abstract, strictfp

We will be looking into more details about modifiers in the next section.

Java Variables:

We would see following type of variables in Java:

- Local Variables
- Class Variables (Static Variables)
- Instance Variables (Non static variables)

Java Arrays:

Arrays are objects that store multiple variables of the same type. However an Array itself is an object on the heap. We will look into how to declare, construct and initialize in the upcoming chapters.

Java Enums:

Enums were introduced in java 5.0. Enums restrict a variable to have one of only a few predefined values. The values in this enumerated list are called enums.

With the use of enums it is possible to reduce the number of bugs in your code.

For example if we consider an application for a fresh juice shop it would be possible to restrict the glass size to small, medium and Large. This would make sure that it would not allow anyone to order any size other than the small, medium or large.

Example:

```
class FreshJuice {

   enum FreshJuiceSize{ SMALL, MEDUIM, LARGE }
   FreshJuiceSize size;
}

public class FreshJuiceTest {

   public static void main(String args[]){
      FreshJuice juice = new FreshJuice();
      juice.size = FreshJuice. FreshJuiceSize.MEDUIM ;
   }
}
```

Note: enums can be declared as their own or inside a class. Methods, variables, constructors can be defined inside enums as well.

Java Keywords:

The following list shows the reserved words in Java. These reserved words may not be used as constant or variable or any other identifier names.

Abstract	assert	boolean	break
Byte	case	catch	char
Class	const	continue	default
Do	double	else	enum
Extends	final	finally	float
For	goto	if	implements
Import	instanceof	int	interface
Long	native	new	package

Private	protected	public	return
Short	static	strictfp	super
Switch	synchronized	this	throw
Throws	transient	try	void
Volatile	while		

Comments in Java

Java supports single line and multi-line comments very similar to c and c++. All characters available inside any comment are ignored by Java compiler.

```
public class MyFirstJavaProgram{

   /* This is my first java program.
    * This will print 'Hello World' as the output
    * This is an example of multi-line comments.
    */

    public static void main(String []args){
       // This is an example of single line comment
       /* This is also an example of single line comment. */
       System.out.println("Hello World");
    }
}
```

Using Blank Lines:

A line containing only whitespace, possibly with a comment, is known as a blank line, and Java totally ignores it.

Inheritance:

In java classes can be derived from classes. Basically if you need to create a new class and here is already a class that has some of the code you require, then it is possible to derive your new class from the already existing code.

This concept allows you to reuse the fields and methods of the existing class with out having to rewrite the code in a new class. In this scenario the existing class is called the super class and the derived class is called the subclass.

Interfaces:

In Java language an interface can be defined as a contract between objects on how to communicate with each other. Interfaces play a vital role when it comes to the concept of inheritance.

An interface defines the methods, a deriving class(subclass) should use. But the implementation of the methods is totally up to the subclass.

JAVA

Java - Objects and Classes

Java is an Object Oriented Language. As a language that has the Object Oriented feature Java supports the following fundamental concepts:

- Polymorphism
- Inheritance
- Encapsulation
- Abstraction
- Classes
- Objects
- Instance
- Method
- Message Parsing

In this chapter we will look into the concepts Classes and Objects.

- **Object -** Objects have states and behaviors. Example: A dog has states-color, name, breed as well as behaviors -wagging, barking, eating. An object is an instance of a class.

- **Class -** A class can be defined as a template/ blue print that describe the behaviors/states that object of its type support.

Objects in Java:

Let us now look deep into what are objects. If we consider the real-world we can find many objects around us, Cars, Dogs, Humans etc. All these objects have a state and behavior.

If we consider a dog then its state is - name, breed, color, and the behavior is - barking, wagging, running

If you compare the software object with a real world object, they have very similar characteristics.

Software objects also have a state and behavior. A software object's state is stored in fields and behavior is shown via methods.

So in software development methods operate on the internal state of an object and the object-to-object communication is done via methods.

Classes in Java:

A class is a blue print from which individual objects are created.

A sample of a class is given below:

```
public class Dog{
   String breed;
   int age;
   String color;


   void barking(){
   }
```

```
   void hungry(){
   }

   void sleeping(){
   }
}
```

A class can contain any of the following variable types.

- **Local variables .** variables defined inside methods, constructors or blocks are called local variables. The variable will be declared and initialized within the method and the variable will be destroyed when the method has completed.

- **Instance variables .** Instance variables are variables within a class but outside any method. These variables are instantiated when the class is loaded. Instance variables can be accessed from inside any method, constructor or blocks of that particular class.

- **Class variables .** Class variables are variables declared with in a class, outside any method, with the static keyword.

A class can have any number of methods to access the value of various kind of methods. In the above example, barking(), hungry() and sleeping() are methods.

Below mentioned are some of the important topics that need to be discussed when looking into classes of the Java Language.

Constructors:

When discussing about classes one of the most important sub topic would be constructors. Every class has a constructor. If we do not explicitly write a constructor for a class the java compiler builds a default constructor for that class.

Each time a new object is created at least one constructor will be invoked. The main rule of constructors is that they should have the same name as the class. A class can have more than one constructor.

Example of a constructor is given below:

```
public class Puppy{
   public puppy(){
   }

   public puppy(String name){
      // This constructor has one parameter, name.
   }
}
```

Java also supports Singleton Classes where you would be able to create only one instance of a class.

Creating an Object:

JAVA

As mentioned previously a class provides the blueprints for objects. So basically an object is created from a class. In java the new key word is used to create new objects.

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.

Example of creating an object is given below:

```
public class Puppy{

   public Puppy(String name){
      // This constructor has one parameter, name.
      System.out.println("Passed Name is :" + name );
   }
   public static void main(String []args){
      // Following statement would create an object myPuppy
      Puppy myPuppy = new Puppy( "tommy" );
   }
}
```

If we compile and run the above program then it would produce following result:

```
Passed Name is :tommy
```

Accessing Instance Variables and Methods:

Instance variables and methods are accessed via created objects. To access an instance variable the fully qualified path should be as follows:

```
/* First create an object */
ObjectReference = new Constructor();

/* Now call a variable as follows */
ObjectReference.variableName;

/* Now you can call a class method as follows */
ObjectReference.MethodName();
```

Example:

This example explains how to access instance variables and methods of a class:

```
public class Puppy{
```

```
   int puppyAge;

   public Puppy(String name){
      // This constructor has one parameter, name.
      System.out.println("Passed Name is :" + name );
   }
   public void setAge( int age ){
       puppyAge = age;
   }

   public int getAge( ){
       System.out.println("Puppy's age is :" + puppyAge );
       return puppyAge;
   }
   public static void main(String []args){
      /* Object creation */
      Puppy myPuppy = new Puppy( "tommy" );

      /* Call class method to set puppy's age */
      myPuppy.setAge( 2 );

      /* Call another class method to get puppy's age */
      myPuppy.getAge( );

      /* You can access instance variable as follows as well */
      System.out.println("Variable Value :" + myPuppy.puppyAge );
   }
}
```

If we compile and run the above program then it would produce following result:

```
Passed Name is :tommy
Puppy's age is :2
Variable Value :2
```

Source file declaration rules:

As the last part of this section lets us now look into the source file declaration rules. These rules are essential when declaring classes, *import* statements and *package* statements in a source file.

- There can be only one public class per source file.
- A source file can have multiple non public classes.

- The public class name should be the name of the source file as well which should be appended by **.java** at the end. For example : The class name is . *public class Employee{}* Then the source file should be as Employee.java.

- If the class is defined inside a package, then the package statement should be the first statement in the source file.

- If import statements are present then they must be written between the package statement and the class declaration. If there are no package statements then the import statement should be the first line in the source file.

- Import and package statements will imply to all the classes present in the source file. It is not possible to declare different import and/or package statements to different classes in the source file.

Classes have several access levels and there are different types of classes; abstract classes, final classes etc. I will be explaining about all these in the access modifiers chapter.

Apart from the above mentioned types of classes, Java also has some special classes called Inner classes and Anonymous classes.

Java Package:

In simple it is a way of categorizing the classes and interfaces. When developing applications in Java, hundreds of classes and interfaces will be written, therefore categorizing these classes is a must as well as makes life much easier.

Import statements:

In java if a fully qualified name, which includes the package and the class name, is given then the compiler can easily locate the source code or classes. Import statement is a way of giving the proper location for the compiler to find that particular class.

For example following line would ask compiler to load all the classes available in directory java_installation/java/io :

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

A Simple Case Study:

For our case study we will be creating two classes. They are Employee and EmployeeTest.

First open notepad and add the following code. Remember this is the Employee class and the class is a public class. Now save this source file with the name Employee.java.

The Employee class has four class variables name, age, designation and salary. The class has one explicitly defined constructor which takes a parameter.

```
import java.io.*;

public class Employee{

   String name;

   int age;

   String designation;

   double salary;
```

```
   // This is the constructor of the class Employee
   public Employee(String name){
      this.name = name;
   }
   // Assign the age of the Employee  to the variable age.
   public void empAge(int empAge){
      age =  empAge;
   }
   /* Assign the designation to the variable designation.*/
   public void empDesignation(String empDesig){
      designation = empDesig;
   }
   /* Assign the salary to the variable         salary.*/
   public void empSalary(double empSalary){
      salary = empSalary;
   }
   /* Print the Employee details */
   public void printEmployee(){
      System.out.println("Name:"+ name );
      System.out.println("Age:" + age );
      System.out.println("Designation:" + designation );
      System.out.println("Salary:" + salary);
   }
}
```

As mentioned previously in this tutorial processing starts from the main method. Therefore in-order for us to run this Employee class there should be main method and objects should be created. We will be creating a separate class for these tasks.

Given below is the *EmployeeTest* class which creates two instances of the class Employee and invokes the methods for each object to assign values for each variable.

Save the following code in EmployeeTest.java file

```
import java.io.*;
public class EmployeeTest{


   public static void main(String args[]){
      /* Create two objects using constructor */
      Employee empOne = new Employee("James Smith");
      Employee empTwo = new Employee("Mary Anne");


      // Invoking methods for each object created
      empOne.empAge(26);
```

```
      empOne.empDesignation("Senior Software Engineer");
      empOne.empSalary(1000);
      empOne.printEmployee();

      empTwo.empAge(21);
      empTwo.empDesignation("Software Engineer");
      empTwo.empSalary(500);
      empTwo.printEmployee();
   }
}
```

Now compile both the classes and then run *EmployeeTest* to see the result as follows:

```
C :> javac Employee.java
C :> vi EmployeeTest.java
C :> javac  EmployeeTest.java
C :> java EmployeeTest
Name:James Smith
Age:26
Designation:Senior Software Engineer
Salary:1000.0
Name:Mary Anne
Age:21
Designation:Software Engineer
Salary:500.0
```

Java Basic Data Types

Variables are nothing but reserved memory locations to store values. This means that when you create a variable you reserve some space in memory.

Based on the data type of a variable, the operating system allocates memory and decides what can be stored in the reserved memory. Therefore, by assigning different data types to variables, you can store integers, decimals, or characters in these variables.

There are two data types available in Java:

- Primitive Data Types
- Reference/Object Data Types

Primitive Data Types:

There are eight primitive data types supported by Java. Primitive data types are predefined by the language and named by a key word. Let us now look into detail about the eight primitive data types.

byte:

- Byte data type is a 8-bit signed two's complement integer.
- Minimum value is -128 (-2^7)
- Maximum value is 127 (inclusive)(2^7 -1)
- Default value is 0
- Byte data type is used to save space in large arrays, mainly in place of integers, since a byte is four times smaller than an int.
- Example : byte a = 100 , byte b = -50

short:

- Short data type is a 16-bit signed two's complement integer.
- Minimum value is -32,768 (-2^15)
- Maximum value is 32,767(inclusive) (2^15 -1)
- Short data type can also be used to save memory as byte data type. A short is 2 times smaller than an int
- Default value is 0.
- Example : short s= 10000 , short r = -20000

int:

- Int data type is a 32-bit signed two's complement integer.
- Minimum value is - 2,147,483,648.(-2^31)
- Maximum value is 2,147,483,647(inclusive).(2^31 -1)
- Int is generally used as the default data type for integral values unless there is a concern about memory.
- The default value is 0.
- Example : int a = 100000, int b = -200000

long:

- Long data type is a 64-bit signed two's complement integer.
- Minimum value is -9,223,372,036,854,775,808.(-2^63)
- Maximum value is 9,223,372,036,854,775,807 (inclusive). (2^63 -1)
- This type is used when a wider range than int is needed.
- Default value is 0L.

- Example : long a = 100000L, int b = -200000L

float:

- Float data type is a single-precision 32-bit IEEE 754 floating point.
- Float is mainly used to save memory in large arrays of floating point numbers.
- Default value is 0.0f.
- Float data type is never used for precise values such as currency.
- Example : float f1 = 234.5f

double:

- double data type is a double-precision 64-bit IEEE 754 floating point.
- This data type is generally used as the default data type for decimal values. generally the default choice.
- Double data type should never be used for precise values such as currency.
- Default value is 0.0d.
- Example : double d1 = 123.4

boolean:

- boolean data type represents one bit of information.
- There are only two possible values : true and false.
- This data type is used for simple flags that track true/false conditions.
- Default value is false.
- Example : boolean one = true

char:

- char data type is a single 16-bit Unicode character.
- Minimum value is '\u0000' (or 0).
- Maximum value is '\uffff' (or 65,535 inclusive).
- Char data type is used to store any character.
- Example . char letterA ='A'

Reference Data Types:

- Reference variables are created using defined constructors of the classes. They are used to access objects. These variables are declared to be of a specific type that cannot be changed. For example, Employee, Puppy etc.
- Class objects, and various type of array variables come under reference data type.
- Default value of any reference variable is null.
- A reference variable can be used to refer to any object of the declared type or any compatible type.
- Example : Animal animal = new Animal("giraffe");

Java Literals:

A literal is a source code representation of a fixed value. They are represented directly in the code without any computation.

Literals can be assigned to any primitive type variable. For example:

```
byte a = 68;
char a = 'A'
```

byte, int, long, and short can be expressed in decimal(base 10),hexadecimal(base 16) or octal(base 8) number systems as well.

Prefix 0 is used to indicates octal and prefix 0x indicates hexadecimal when using these number systems for literals. For example:

```
int decimal = 100;
int octal = 0144;
int hexa =  0x64;
```

String literals in Java are specified like they are in most other languages by enclosing a sequence of characters between a pair of double quotes. Examples of string literals are:

```
"Hello World"
"two\nlines"
"\"This is in quotes\""
```

String and char types of literals can contain any Unicode characters. For example:

```
char a = '\u0001';
String a = "\u0001";
```

Java language supports few special escape sequences for String and char literals as well. They are:

Notation	Character represented
\n	Newline (0x0a)

\r	Carriage return (0x0d)
\f	Formfeed (0x0c)
\b	Backspace (0x08)
\s	Space (0x20)
\t	tab
\"	Double quote
\'	Single quote
\\	backslash
\ddd	Octal character (ddd)
\uxxxx	Hexadecimal UNICODE character (xxxx)

Variable Types

In Java, all variables must be declared before they can be used. The basic form of a variable declaration is shown here:

```
type identifier [ = value][, identifier [= value] ...] ;
```

The *type* is one of Java's datatypes. The *identifier* is the name of the variable. To declare more than one variable of the specified type, use a comma-separated list.

Here are several examples of variable declarations of various types. Note that some include an initialization.

```
int a, b, c;         // declares three ints, a, b, and c.
int d = 3, e, f = 5; // declares three more ints, initializing
                     // d and f.
byte z = 22;         // initializes z.
double pi = 3.14159; // declares an approximation of pi.
char x = 'x';        // the variable x has the value 'x'.
```

This chapter will explain various variable types available in Java Language. There are three kinds of variables in Java:

- Local variables
- Instance variables

- Class/static variables

Local variables :

- Local variables are declared in methods, constructors, or blocks.
- Local variables are created when the method, constructor or block is entered and the variable will be destroyed once it exits the method, constructor or block.
- Access modifiers cannot be used for local variables.
- Local variables are visible only within the declared method, constructor or block.
- Local variables are implemented at stack level internally.
- There is no default value for local variables so local variables should be declared and an initial value should be assigned before the first use.

Example:

Here *age* is a local variable. This is defined inside *pupAge()* method and its scope is limited to this method only.

```
public class Test{
   public void pupAge(){
      int age = 0;
      age = age + 7;
      System.out.println("Puppy age is : " + age);
   }

   public static void main(String args[]){
      Test test = new Test();
      test.pupAge();
   }
}
```

This would produce following result:

```
Puppy age is: 7
```

Example:

Following example uses *age* without initializing it, so it would give an error at the time of compilation.

```
public class Test{
   public void pupAge(){
      int age;
      age = age + 7;
      System.out.println("Puppy age is : " + age);
```

```
   }

   public static void main(String args[]){
      Test test = new Test();
      test.pupAge();
   }
}
```

This would produce following error while compiling it:

```
Test.java:4:variable number might not have been initialized
age = age + 7;
         ^
1 error
```

Instance variables :

- Instance variables are declared in a class, but outside a method, constructor or any block.
- When a space is allocated for an object in the heap a slot for each instance variable value is created.
- Instance variables are created when an object is created with the use of the key word 'new' and destroyed when the object is destroyed.
- Instance variables hold values that must be referenced by more than one method, constructor or block, or essential parts of an object.s state that must be present through out the class.
- Instance variables can be declared in class level before or after use.
- Access modifiers can be given for instance variables.
- The instance variables are visible for all methods, constructors and block in the class. Normally it is recommended to make these variables private (access level).However visibility for subclasses can be given for these variables with the use of access modifiers.
- Instance variables have default values. For numbers the default value is 0, for Booleans it is false and for object references it is null. Values can be assigned during the declaration or within the constructor.
- Instance variables can be accessed directly by calling the variable name inside the class. However within static methods and different class (when instance variables are given accessibility) the should be called using the fully qualified name . *ObjectReference.VariableName*.

Example:

```
import java.io.*;

public class Employee{
   // this instance variable is visible for any child class.
   public String name;
```

```
   // salary  variable is visible in Employee class only.
   private double salary;

   // The name variable is assigned in the constructor.
   public Employee (String empName){
      name = empName;
   }

   // The salary variable is assigned a value.
   public void setSalary(double empSal){
      salary = empSal;
   }

   // This method prints the employee details.
   public void printEmp(){
      System.out.println("name  : " + name );
      System.out.println("salary :" + salary);
   }

   public static void main(String args[]){
      Employee empOne = new Employee("Ransika");
      empOne.setSalary(1000);
      empOne.printEmp();
   }
}
```

This would produce following result:

```
name  : Ransika
salary :1000.0
```

Class/static variables :

- Class variables also known as static variables are declared with the *static* keyword in a class, but outside a method, constructor or a block.
- There would only be one copy of each class variable per class, regardless of how many objects are created from it.
- Static variables are rarely used other than being declared as constants. Constants are variables that are declared as public/private, final and static. Constant variables never change from their initial value.
- Static variables are stored in static memory. It is rare to use static variables other than declared final and used as either public or private constants.
- Static variables are created when the program starts and destroyed when the program stops.

- Visibility is similar to instance variables. However, most static variables are declared public since they must be available for users of the class.
- Default values are same as instance variables. For numbers the default value is 0, for Booleans it is false and for object references it is null. Values can be assigned during the declaration or within the constructor. Additionally values can be assigned in special static initializer blocks.
- Static variables can be accessed by calling with the class name . *ClassName.VariableName*.
- When declaring class variables as public static final, then variables names (constants) are all in upper case. If the static variables are not public and final the naming syntax is the same as instance and local variables.

Example:

```
import java.io.*;


public class Employee{
   // salary  variable is a private static variable
   private static double salary;


   // DEPARTMENT is a constant
   public static final String DEPARTMENT = "Development ";


   public static void main(String args[]){
      salary = 1000;
      System.out.println(DEPARTMENT+"average salary:"+salary);
   }
}
```

This would produce following result:

```
Development average salary:1000
```

Note: If the variables are access from an outside class the constant should be accessed as Employee.DEPARTMENT

Java Modifier Types

Modifiers are keywords that you add to those definitions to change their meanings. The Java language has a wide variety of modifiers, including the following:

- Java Access Modifiers
- Non Access Modifiers

To use a modifier, you include its keyword in the definition of a class, method, or variable. The modifier precedes the rest of the statement, as in the following examples (Italic ones):

```
public class className {
   // ...
```

```
}
private boolean myFlag;
static final double weeks = 9.5;
protected static final int BOXWIDTH = 42;
public static void main(String[] arguments) {
   // body of method
}
```

Access Control Modifiers:

Java provides a number of access modifiers to set access levels for classes, variables, methods and constructors. The four access levels are:

- Visible to the package. the default. No modifiers are needed.
- Visible to the class only (private).
- Visible to the world (public).
- Visible to the package and all subclasses (protected).

Non Access Modifiers:

Java provides a number of non-access modifiers to achieve many other functionality.

- The *static* modifier for creating class methods and variables
- The *final* modifier for finalizing the implementations of classes, methods, and variables.
- The *abstract* modifier for creating abstract classes and methods.
- The *synchronized* and *volatile* modifiers, which are used for threads.

Java Basic Operators

Java provides a rich set of operators to manipulate variables. We can divide all the Java operators into the following groups:

- Arithmetic Operators
- Relational Operators
- Bitwise Operators
- Logical Operators
- Assignment Operators
- Misc Operators

The Arithmetic Operators:

JAVA

Arithmetic operators are used in mathematical expressions in the same way that they are used in algebra. The following table lists the arithmetic operators:

Assume integer variable A holds 10 and variable B holds 20 then:

Show Examples

Operator	Description	Example
+	Addition - Adds values on either side of the operator	A + B will give 30
-	Subtraction - Subtracts right hand operand from left hand operand	A - B will give -10
*	Multiplication - Multiplies values on either side of the operator	A * B will give 200
/	Division - Divides left hand operand by right hand operand	B / A will give 2
%	Modulus - Divides left hand operand by right hand operand and returns remainder	B % A will give 0
++	Increment - Increase the value of operand by 1	B++ gives 21
--	Decrement - Decrease the value of operand by 1	B-- gives 19

The Relational Operators:

There are following relational operators supported by Java language

Assume variable A holds 10 and variable B holds 20 then:

Show Examples

Operator	Description	Example
==	Checks if the value of two operands are equal or not, if yes then condition becomes true.	(A == B) is not true.
!=	Checks if the value of two operands are equal or not, if values are not equal then condition becomes true.	(A != B) is true.
>	Checks if the value of left operand is greater than the value of right operand, if yes then condition becomes true.	(A > B) is not true.
<	Checks if the value of left operand is less than the value of right operand, if yes then condition becomes true.	(A < B) is true.
>=	Checks if the value of left operand is greater than or equal to the value of right operand, if yes then condition becomes true.	(A >= B) is not true.

<=	Checks if the value of left operand is less than or equal to the value of right operand, if yes then condition becomes true.	(A <= B) is true.

The Bitwise Operators:

Java defines several bitwise operators which can be applied to the integer types, long, int, short, char, and byte.

Bitwise operator works on bits and perform bit by bit operation. Assume if a = 60; and b = 13; Now in binary format they will be as follows:

a = 0011 1100

b = 0000 1101

---

a&b = 0000 1100

a|b = 0011 1101

a^b = 0011 0001

~a = 1100 0011

The following table lists the bitwise operators:

Assume integer variable A holds 60 and variable B holds 13 then:

Show Examples

Operator	Description	Example
&	Binary AND Operator copies a bit to the result if it exists in both operands.	(A & B) will give 12 which is 0000 1100
|	Binary OR Operator copies a bit if it exists in eather operand.	(A | B) will give 61 which is 0011 1101
^	Binary XOR Operator copies the bit if it is set in one operand but not both.	(A ^ B) will give 49 which is 0011 0001
~	Binary Ones Complement Operator is unary and has the efect of 'flipping' bits.	(~A) will give -60 which is 1100 0011
<<	Binary Left Shift Operator. The left operands value is moved left by the number of bits specified by the right operand.	A << 2 will give 240 which is 1111 0000
>>	Binary Right Shift Operator. The left operands value is moved right by the number of bits specified by the right operand.	A >> 2 will give 15 which is 1111
>>>	Shift right zero fill operator. The left operands value is moved right by the number	A >>>2 will give 15

	of bits specified by the right operand and shifted values are filled up with zeros.	which is 0000 1111

The Logical Operators:

The following table lists the logical operators:

Assume boolean variables A holds true and variable B holds false then:

Show Examples

Operator	Description	Example
&&	Called Logical AND operator. If both the operands are non zero then then condition becomes true.	(A && B) is false.
||	Called Logical OR Operator. If any of the two operands are non zero then then condition becomes true.	(A || B) is true.
!	Called Logical NOT Operator. Use to reverses the logical state of its operand. If a condition is true then Logical NOT operator will make false.	!(A && B) is true.

The Assignment Operators:

There are following assignment operators supported by Java language:

Show Examples

Operator	Description	Example
=	Simple assignment operator, Assigns values from right side operands to left side operand	C = A + B will assigne value of A + B into C
+=	Add AND assignment operator, It adds right operand to the left operand and assign the result to left operand	C += A is equivalent to C = C + A
-=	Subtract AND assignment operator, It subtracts right operand from the left operand and assign the result to left operand	C -= A is equivalent to C = C - A
*=	Multiply AND assignment operator, It multiplies right operand with the left operand and assign the result to left operand	C *= A is equivalent to C = C * A
/=	Divide AND assignment operator, It divides left operand with the right operand and assign the result to left operand	C /= A is equivalent to C = C / A
%=	Modulus AND assignment operator, It takes modulus using two operands and assign the result to left operand	C %= A is equivalent to C = C % A
<<=	Left shift AND assignment operator	C <<= 2 is same as C = C << 2

>>=	Right shift AND assignment operator	C >>= 2 is same as C = C >> 2
&=	Bitwise AND assignment operator	C &= 2 is same as C = C & 2
^=	bitwise exclusive OR and assignment operator	C ^= 2 is same as C = C ^ 2
|=	bitwise inclusive OR and assignment operator	C |= 2 is same as C = C | 2

Misc Operators

There are few other operators supported by Java Language.

Conditional Operator (? :):

Conditional operator is also known as the ternary operator. This operator consists of three operands and is used to evaluate boolean expressions. The goal of the operator is to decide which value should be assigned to the variable. The operator is written as :

```
variable x = (expression) ? value if true : value if false
```

Following is the example:

```
public class Test {

   public static void main(String args[]){
      int a , b;
      a = 10;
      b = (a == 1) ? 20: 30;
      System.out.println( "Value of b is : " +  b );

      b = (a == 10) ? 20: 30;
      System.out.println( "Value of b is : " + b );
   }
}
```

This would produce following result:

```
Value of b is : 30
Value of b is : 20
```

instanceOf Operator:

This operator is used only for object reference variables. The operator checks whether the object is of a particular type(class type or interface type). instanceOf operator is wriiten as:

```
( Object reference variable ) instanceOf  (class/interface type)
```

If the object referred by the variable on the left side of the operator passes the IS-A check for the class/interface type on the right side then the result will be true. Following is the example:

```
String name = = 'James';

boolean result = name instanceOf String;

// This will return true since name is type of String
```

This operator will still return true if the object being compared is the assignment compatible with the type on the right. Following is one more example:

```
class Vehicle {}


public class Car extends Vehicle {

   public static void main(String args[]){

      Vehicle a = new Car();

      boolean result =  a instanceof Car;

      System.out.println( result);

   }

}
```

This would produce following result:

```
true
```

Precedence of Java Operators:

Operator precedence determines the grouping of terms in an expression. This affects how an expression is evaluated. Certain operators have higher precedence than others; for example, the multiplication operator has higher precedence than the addition operator:

For example x = 7 + 3 * 2; Here x is assigned 13, not 20 because operator * has higher precedenace than + so it first get multiplied with 3*2 and then adds into 7.

Here operators with the highest precedence appear at the top of the table, those with the lowest appear at the bottom. Within an expression, higher precedenace operators will be evaluated first.

Category	**Operator**	**Associativity**
Postfix	() [] . (dot operator)	Left to right
Unary	++ - - ! ~	Right to left
Multiplicative	* / %	Left to right
Additive	+ -	Left to right

Shift	>> >>> <<	Left to right
Relational	> >= < <=	Left to right
Equality	== !=	Left to right
Bitwise AND	&	Left to right
Bitwise XOR	^	Left to right
Bitwise OR	|	Left to right
Logical AND	&&	Left to right
Logical OR	||	Left to right
Conditional	?:	Right to left
Assignment	= += -= *= /= %=>> = <<= &= ^= |=	Right to left
Comma	,	Left to right

Loops - for, while and do...while

There may be a sitution when we need to execute a block of code several number of times, and is often referred to as a loop.

Java has very flexible three looping mechanisms. You can use one of the following three loops:

- while Loop
- do...while Loop
- for Loop

As of java 5 the *enhanced for loop* was introduced. This is mainly used for Arrays.

The while Loop:

A while loop is a control structure that allows you to repeat a task a certain number of times.

Syntax:

The syntax of a while loop is:

```
while(Boolean_expression)
{
```

```
   //Statements
}
```

When executing, if the *boolean_expression* result is true then the actions inside the loop will be executed. This will continue as long as the expression result is true.

Here key point of the *while* loop is that the loop might not ever run. When the expression is tested and the result is false, the loop body will be skipped and the first statement after the while loop will be executed.

Example:

```
public class Test {


   public static void main(String args[]) {
      int x = 10;


      while( x < 20 ) {
         System.out.print("value of x : " + x );
         x++;
         System.out.print("\n");
      }
   }
}
```

This would produce following result:

```
value of x : 10
value of x : 11
value of x : 12
value of x : 13
value of x : 14
value of x : 15
value of x : 16
value of x : 17
value of x : 18
value of x : 19
```

The do...while Loop:

A do...while loop is similar to a while loop, except that a do...while loop is guaranteed to execute at least one time.

Syntax:

The syntax of a do...while loop is:

```
do
{
```

```
   //Statements
}while(Boolean_expression);
```

Notice that the Boolean expression appears at the end of the loop, so the statements in the loop execute once before the Boolean is tested.

If the Boolean expression is true, the flow of control jumps back up to do, and the statements in the loop execute again. This process repeats until the Boolean expression is false.

Example:

```
public class Test {


   public static void main(String args[]){
      int x = 10;


      do{
         System.out.print("value of x : " + x );
         x++;
         System.out.print("\n");
      }while( x < 20 );
   }
}
```

This would produce following result:

```
value of x : 10
value of x : 11
value of x : 12
value of x : 13
value of x : 14
value of x : 15
value of x : 16
value of x : 17
value of x : 18
value of x : 19
```

The for Loop:

A for loop is a repetition control structure that allows you to efficiently write a loop that needs to execute a specific number of times.

A for loop is useful when you know how many times a task is to be repeated.

Syntax:

The syntax of a for loop is:

```
for(initialization; Boolean_expression; update)
{
   //Statements
}
```

Here is the flow of control in a for loop:

- The initialization step is executed first, and only once. This step allows you to declare and initialize any loop control variables. You are not required to put a statement here, as long as a semicolon appears.
- Next, the Boolean expression is evaluated. If it is true, the body of the loop is executed. If it is false, the body of the loop does not execute and flow of control jumps to the next statement past the for loop.
- After the body of the for loop executes, the flow of control jumps back up to the update statement. This statement allows you to update any loop control variables. This statement can be left blank, as long as a semicolon appears after the Boolean expression.
- The Boolean expression is now evaluated again. If it is true, the loop executes and the process repeats itself (body of loop, then update step,then Boolean expression). After the Boolean expression is false, the for loop terminates.

Example:

```
public class Test {


   public static void main(String args[]) {


      for(int x = 10; x < 20; x = x+1) {
         System.out.print("value of x : " + x );
         System.out.print("\n");
      }
   }
}
```

This would produce following result:

```
value of x : 10
value of x : 11
value of x : 12
value of x : 13
value of x : 14
value of x : 15
value of x : 16
value of x : 17
value of x : 18
value of x : 19
```

Enhanced for loop in Java:

As of java 5 the enhanced for loop was introduced. This is mainly used for Arrays.

Syntax:

The syntax of enhanced for loop is:

```
for(declaration : expression)
{
   //Statements
}
```

- **Declaration** . The newly declared block variable, which is of a type compatible with the elements of the array you are accessing. The variable will be available within the for block and its value would be the same as the current array element.

- **Expression** . This evaluate to the array you need to loop through. The expression can be an array variable or method call that returns an array.

Example:

```
public class Test {


   public static void main(String args[]){
      int [] numbers = {10, 20, 30, 40, 50};


      for(int x : numbers ){
         System.out.print( x );
         System.out.print(",");
      }
      System.out.print("\n");
      String [] names ={"James", "Larry", "Tom", "Lacy"};
      for( String name : names ) {
         System.out.print( name );
         System.out.print(",");
      }
   }
}
```

This would produce following result:

```
10,20,30,40,50,
James,Larry,Tom,Lacy,
```

The break Keyword:

The *break* keyword is used to stop the entire loop. The break keyword must be used inside any loop or a switch statement.

The break keyword will stop the execution of the innermost loop and start executing the next line of code after the block.

Syntax:

The syntax of a break is a single statement inside any loop:

```
break;
```

Example:

```
public class Test {


   public static void main(String args[]) {
      int [] numbers = {10, 20, 30, 40, 50};


      for(int x : numbers ) {
         if( x == 30 ) {
                break;
         }
         System.out.print( x );
         System.out.print("\n");
      }
   }
}
```

This would produce following result:

```
10
20
```

The continue Keyword:

The *continue* keyword can be used in any of the loop control structures. It causes the loop to immediately jump to the next iteration of the loop.

- In a for loop, the continue keyword causes flow of control to immediately jump to the update statement.
- In a while loop or do/while loop, flow of control immediately jumps to the Boolean expression.

Syntax:

The syntax of a continue is a single statement inside any loop:

```
continue;
```

Example:

```
public class Test {


   public static void main(String args[]) {
      int [] numbers = {10, 20, 30, 40, 50};


      for(int x : numbers ) {
         if( x == 30 ) {
                continue;
         }
         System.out.print( x );
         System.out.print("\n");
      }
   }
}
```

This would produce following result:

```
10
20
40
50
```

Java Decision Making

There are two types of decision making statements in Java. They are:

- if statements
- switch statements

The if Statement:

An if statement consists of a Boolean expression followed by one or more statements.

Syntax:

The syntax of an if statement is:

```
if(Boolean_expression)
{
   //Statements will execute if the Boolean expression is true
}
```

If the boolean expression evaluates to true then the block of code inside the if statement will be executed. If not the first set of code after the end of the if statement(after the closing curly brace) will be executed.

Example:

```
public class Test {

   public static void main(String args[]){
      int x = 10;

      if( x < 20 ){
         System.out.print("This is if statement");
      }
   }
}
```

This would produce following result:

```
This is if statement
```

The if...else Statement:

An if statement can be followed by an optional *else* statement, which executes when the Boolean expression is false.

Syntax:

The syntax of a if...else is:

```
if(Boolean_expression){
   //Executes when the Boolean expression is true
}else{
   //Executes when the Boolean expression is false
}
```

Example:

```
public class Test {

   public static void main(String args[]){
      int x = 30;

      if( x < 20 ){
         System.out.print("This is if statement");
      }else{
         System.out.print("This is else statement");
      }
   }
}
```

This would produce following result:

```
This is else statement
```

The if...else if...else Statement:

An if statement can be followed by an optional *else if...else* statement, which is very usefull to test various conditions using single if...else if statement.

When using if , else if , else statements there are few points to keep in mind.

- An if can have zero or one else's and it must come after any else if's.
- An if can have zero to many else if's and they must come before the else.
- Once an else if succeeds, none of he remaining else if's or else's will be tested.

Syntax:

The syntax of a if...else is:

```
if(Boolean_expression 1){
   //Executes when the Boolean expression 1 is true
}else if(Boolean_expression 2){
   //Executes when the Boolean expression 2 is true
}else if(Boolean_expression 3){
   //Executes when the Boolean expression 3 is true
}else {
   //Executes when the none of the above condition is true.
}
```

Example:

```
public class Test {


   public static void main(String args[]){
      int x = 30;


      if( x == 10 ){
         System.out.print("Value of X is 10");
      }else if( x == 20 ){
         System.out.print("Value of X is 20");
      }else if( x == 30 ){
         System.out.print("Value of X is 30");
      }else{
         System.out.print("This is else statement");
      }
```

```
   }
}
```

This would produce following result:

```
Value of X is 30
```

Nested if...else Statement:

It is always legal to nest if-else statements, which means you can use one if or else if statement inside another if or else if statement.

Syntax:

The syntax for a nested if...else is as follows:

```
if(Boolean_expression 1){
   //Executes when the Boolean expression 1 is true
   if(Boolean_expression 2){
      //Executes when the Boolean expression 2 is true
   }
}
```

You can nest *else if...else* in the similar way as we have nested *if* statement.

Example:

```
public class Test {


   public static void main(String args[]){
      int x = 30;
      int y = 10;


      if( x == 30 ){
         if( y == 10 ){
         System.out.print("X = 30 and Y = 10");
      }
   }
}
```

This would produce following result:

```
X = 30 and Y = 10
```

The switch Statement:

A *switch* statement allows a variable to be tested for equality against a list of values. Each value is called a case, and the variable being switched on is checked for each case.

Syntax:

The syntax of enhanced for loop is:

```
switch(expression){
    case value :
       //Statements
       break; //optional
    case value :
       //Statements
       break; //optional
    //You can have any number of case statements.
    default : //Optional
       //Statements
}
```

The following rules apply to a switch statement:

- The variable used in a switch statement can only be a byte, short, int, or char.
- You can have any number of case statements within a switch. Each case is followed by the value to be compared to and a colon.
- The value for a case must be the same data type as the variable in the switch, and it must be a constant or a literal.
- When the variable being switched on is equal to a case, the statements following that case will execute until a *break* statement is reached.
- When a *break* statement is reached, the switch terminates, and the flow of control jumps to the next line following the switch statement.
- Not every case needs to contain a break. If no break appears, the flow of control will *fall through* to subsequent cases until a break is reached.
- A *switch* statement can have an optional default case, which must appear at the end of the switch. The default case can be used for performing a task when none of the cases is true. No break is needed in the default case.

Example:

```
public class Test {


   public static void main(String args[]){
      char grade = args[0].charAt(0);


      switch(grade)
      {
         case 'A' :
```

```
            System.out.println("Excellent!");
            break;
         case 'B' :
         case 'C' :
            System.out.println("Well done");
            break;
         case 'D' :
            System.out.println("You passed");
         case 'F' :
            System.out.println("Better try again");
            break;
         default :
            System.out.println("Invalid grade");
      }
      System.out.println("Your grade is " + grade);
   }
}
```

Compile and run above program using various command line arguments. This would produce following result:

```
$ java Test a
Invalid grade
Your grade is a a
$ java Test A
Excellent!
Your grade is a A
$ java Test C
Well done
Your grade is a C
$
```

Java - Numbers Class

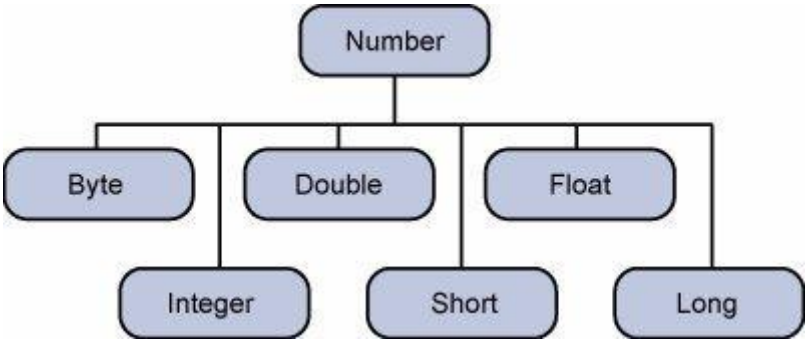
Normally, when we work with Numbers, we use primitive data types such as byte, int, long, double etc.

Example:

```
int i = 5000;
float gpa = 13.65;
byte mask = 0xaf;
```

However in development we come across situations were we need to use objects instead of primitive data types. In-order to achieve this Java provides wrapper classes for each primitive data type.

All the wrapper classes (Integer, Long, Byte, Double, Float, Short) are subclasses of the abstract class Number.



This wrapping is taken care of by the compiler The process is called boxing. So when a primitive is used when an object is required the compiler boxes the primitive type in its wrapper class. Similarly the compiler unboxes the object to a primitive as well. The **Number** is part of the java.lang package.

Here is an example of boxing and unboxing:

```
public class Test{


   public static void main(String args[]){
      Integer x = 5; // boxes int to an Integer object
      x =  x + 10;   // unboxes the Integer to a int
      System.out.println(x);
   }
}
```

This would produce following result:

```
15
```

When x is assigned integer values, the compiler boxes the integer because x is integer objects. Later, x is unboxed so that they can be added as integers.

Number Methods:

Here is the list of the instance methods that all the subclasses of the Number class implement:

SN	Methods with Description
1	xxxValue() Converts the value of *this* Number object to the xxx data type and returned it.
2	compareTo() Compares *this* Number object to the argument.
3	equals() Determines whether *this* number object is equal to the argument.
4	valueOf()

	Returns an Integer object holding the value of the specified primitive.
5	toString() Returns a String object representing the value of specified int or Integer.
6	parseInt() This method is used to get the primitive data type of a certain String.
7	abs() Returns the absolute value of the argument.
8	ceil() Returns the smallest integer that is greater than or equal to the argument. Returned as a double.
9	floor() Returns the largest integer that is less than or equal to the argument. Returned as a double.
10	rint() Returns the integer that is closest in value to the argument. Returned as a double.
11	round() Returns the closest long or int, as indicated by the method's return type, to the argument.
12	min() Returns the smaller of the two arguments.
13	max() Returns the larger of the two arguments.
14	exp() Returns the base of the natural logarithms, e, to the power of the argument.
15	log() Returns the natural logarithm of the argument.
16	pow() Returns the value of the first argument raised to the power of the second argument.
17	sqrt() Returns the square root of the argument.
18	sin() Returns the sine of the specified double value.
19	cos() Returns the cosine of the specified double value.
20	tan()

	Returns the tangent of the specified double value.
21	asin() Returns the arcsine of the specified double value.
22	acos() Returns the arccosine of the specified double value.
23	atan() Returns the arctangent of the specified double value.
24	atan2() Converts rectangular coordinates (x, y) to polar coordinate (r, theta) and returns theta.
25	toDegrees() Converts the argument to degrees
26	toRadians() Converts the argument to radians.
27	random() Returns a random number.

Java - Character Class

Normally, when we work with characters, we use primitive data types char.

Example:

```
char ch = 'a';


// Unicode for uppercase Greek omega character
char uniChar = '\u039A';


// an array of chars
char[] charArray ={ 'a', 'b', 'c', 'd', 'e' };
```

However in development we come across situations were we need to use objects instead of primitive data types. In-order to achieve this Java provides wrapper classe **Character** for primitive data type char.

The Character class offers a number of useful class (i.e., static) methods for manipulating characters. You can create a Character object with the Character constructor:

```
Character ch = new Character('a');
```

The Java compiler will also create a Character object for you under some circumstances. For example, if you pass a primitive char into a method that expects an object, the compiler automatically converts the char to a Character for you. This feature is called autoboxing or unboxing, if the conversion goes the other way.

Example:

```
// Here following primitive char 'a'
// is boxed into the Character object ch
Character ch = 'a';

// Here primitive 'x' is boxed for method test,
// return is unboxed to char 'c'
char c = test('x');
```

Escape Sequences:

A character preceded by a backslash (\) is an escape sequence and has special meaning to the compiler.

The newline character (\n) has been used frequently in this tutorial in System.out.println() statements to advance to the next line after the string is printed.

Following table shows the Java escape sequences:

Escape Sequence	Description
\t	Insert a tab in the text at this point.
\b	Insert a backspace in the text at this point.
\n	Insert a newline in the text at this point.
\r	Insert a carriage return in the text at this point.
\f	Insert a form feed in the text at this point.
\'	Insert a single quote character in the text at this point.
\"	Insert a double quote character in the text at this point.
\\	Insert a backslash character in the text at this point.

When an escape sequence is encountered in a print statement, the compiler interprets it accordingly.

Example:

If you want to put quotes within quotes you must use the escape sequence, \", on the interior quotes:

```
public class Test {

   public static void main(String args[]) {
      System.out.println("She said \"Hello!\" to me.");
   }
```

```
}
```

This would produce following result:

```
She said "Hello!" to me.
```

Character Methods:

Here is the list of the important instance methods that all the subclasses of the Character class implement:

SN	Methods with Description
1	isLetter() Determines whether the specified char value is a letter.
2	isDigit() Determines whether the specified char value is a digit.
3	isWhitespace() Determines whether the specified char value is white space.
4	isUpperCase() Determines whether the specified char value is uppercase.
5	isLowerCase() Determines whether the specified char value is lowercase.
6	toUpperCase() Returns the uppercase form of the specified char value.
7	toLowerCase() Returns the lowercase form of the specified char value.
8	toString() Returns a String object representing the specified character valuethat is, a one-character string.

For a complete list of methods, please refer to the java.lang.Character API specification.

Java - String Class

Strings, which are widely used in Java programming, are a sequence of characters. In the Java programming language, strings are objects.

The Java platform provides the String class to create and manipulate strings.

Creating Strings:

The most direct way to create a string is to write:

```
String greeting = "Hello world!";
```

Whenever it encounters a string literal in your code, the compiler creates a String object with its valuein this case, "Hello world!'.

As with any other object, you can create String objects by using the new keyword and a constructor. The String class has eleven constructors that allow you to provide the initial value of the string using different sources, such as an array of characters:

```
public class StringDemo{


   public static void main(String args[]){
      char[] helloArray = { 'h', 'e', 'l', 'l', 'o', '.'};
      String helloString = new String(helloArray);
      System.out.println( helloString );
   }
}
```

This would produce following result:

```
hello
```

Note: The String class is immutable, so that once it is created a String object cannot be changed. If there is a necessity to make alot of modifications to Strings of characters then you should use String Buffer & String Builder Classes.

String Length:

Methods used to obtain information about an object are known as accessor methods. One accessor method that you can use with strings is the length() method, which returns the number of characters contained in the string object.

After the following two lines of code have been executed, len equals 17:

```
public class StringDemo {


   public static void main(String args[]) {
      String palindrome = "Dot saw I was Tod";
      int len = palindrome.length();
      System.out.println( "String Length is : " + len );
   }
}
```

This would produce following result:

```
String Length is : 17
```

Concatenating Strings:

The String class includes a method for concatenating two strings:

```
string1.concat(string2);
```

This returns a new string that is string1 with string2 added to it at the end. You can also use the concat() method with string literals, as in:

```
"My name is ".concat("Zara");
```

Strings are more commonly concatenated with the + operator, as in:

```
"Hello," + " world" + "!"
```

which results in:

```
"Hello, world!"
```

Let us look at the followinge example:

```
public class StringDemo {


   public static void main(String args[]) {
      String string1 = "saw I was ";
      System.out.println("Dot " + string1 + "Tod");
   }
}
```

This would produce following result:

```
Dot saw I was Tod
```

Creating Format Strings:

You have printf() and format() methods to print output with formatted numbers. The String class has an equivalent class method, format(), that returns a String object rather than a PrintStream object.

Using String's static format() method allows you to create a formatted string that you can reuse, as opposed to a one-time print statement. For example, instead of:

```
System.out.printf("The value of the float variable is " +
                  "%f, while the value of the integer " +
                  "variable is %d, and the string " +
                  "is %s", floatVar, intVar, stringVar);
```

you can write:

```
String fs;
fs = String.format("The value of the float variable is " +
                   "%f, while the value of the integer " +
                   "variable is %d, and the string " +
                   "is %s", floatVar, intVar, stringVar);
System.out.println(fs);
```

JAVA

String Methods:

Here is the list methods supported by String class:

SN	Methods with Description
1	char charAt(int index) Returns the character at the specified index.
2	int compareTo(Object o) Compares this String to another Object.
3	int compareTo(String anotherString) Compares two strings lexicographically.
4	int compareToIgnoreCase(String str) Compares two strings lexicographically, ignoring case differences.
5	String concat(String str) Concatenates the specified string to the end of this string.
6	boolean contentEquals(StringBuffer sb) Returns true if and only if this String represents the same sequence of characters as the specified StringBuffer.
7	static String copyValueOf(char[] data) Returns a String that represents the character sequence in the array specified.
8	static String copyValueOf(char[] data, int offset, int count) Returns a String that represents the character sequence in the array specified.
9	boolean endsWith(String suffix) Tests if this string ends with the specified suffix.
10	boolean equals(Object anObject) Compares this string to the specified object.
11	boolean equalsIgnoreCase(String anotherString) Compares this String to another String, ignoring case considerations.
12	byte getBytes() Encodes this String into a sequence of bytes using the platform's default charset, storing the result into a new byte array.
13	byte[] getBytes(String charsetName Encodes this String into a sequence of bytes using the named charset, storing the result into a new byte array.
14	void getChars(int srcBegin, int srcEnd, char[] dst, int dstBegin) Copies characters from this string into the destination character array.

15	int hashCode() Returns a hash code for this string.
16	int indexOf(int ch) Returns the index within this string of the first occurrence of the specified character.
17	int indexOf(int ch, int fromIndex) Returns the index within this string of the first occurrence of the specified character, starting the search at the specified index.
18	int indexOf(String str) Returns the index within this string of the first occurrence of the specified substring.
19	int indexOf(String str, int fromIndex) Returns the index within this string of the first occurrence of the specified substring, starting at the specified index.
20	String intern() Returns a canonical representation for the string object.
21	int lastIndexOf(int ch) Returns the index within this string of the last occurrence of the specified character.
22	int lastIndexOf(int ch, int fromIndex) Returns the index within this string of the last occurrence of the specified character, searching backward starting at the specified index.
23	int lastIndexOf(String str) Returns the index within this string of the rightmost occurrence of the specified substring.
24	int lastIndexOf(String str, int fromIndex) Returns the index within this string of the last occurrence of the specified substring, searching backward starting at the specified index.
25	int length() Returns the length of this string.
26	boolean matches(String regex) Tells whether or not this string matches the given regular expression.
27	boolean regionMatches(boolean ignoreCase, int toffset, String other, int ooffset, int len) Tests if two string regions are equal.
28	boolean regionMatches(int toffset, String other, int ooffset, int len) Tests if two string regions are equal.
29	String replace(char oldChar, char newChar) Returns a new string resulting from replacing all occurrences of oldChar in this string with newChar.

30	String replaceAll(String regex, String replacement Replaces each substring of this string that matches the given regular expression with the given replacement.
31	String replaceFirst(String regex, String replacement) Replaces the first substring of this string that matches the given regular expression with the given replacement.
32	String[] split(String regex) Splits this string around matches of the given regular expression.
33	String[] split(String regex, int limit) Splits this string around matches of the given regular expression.
34	boolean startsWith(String prefix) Tests if this string starts with the specified prefix.
35	boolean startsWith(String prefix, int toffset) Tests if this string starts with the specified prefix beginning a specified index.
36	CharSequence subSequence(int beginIndex, int endIndex) Returns a new character sequence that is a subsequence of this sequence.
37	String substring(int beginIndex) Returns a new string that is a substring of this string.
38	String substring(int beginIndex, int endIndex) Returns a new string that is a substring of this string.
39	char[] toCharArray() Converts this string to a new character array.
40	String toLowerCase() Converts all of the characters in this String to lower case using the rules of the default locale.
41	String toLowerCase(Locale locale) Converts all of the characters in this String to lower case using the rules of the given Locale.
42	String toString() This object (which is already a string!) is itself returned.
43	String toUpperCase() Converts all of the characters in this String to upper case using the rules of the default locale.
44	String toUpperCase(Locale locale) Converts all of the characters in this String to upper case using the rules of the given Locale.
45	String trim() Returns a copy of the string, with leading and trailing whitespace omitted.

46	static String valueOf(primitive data type x) Returns the string representation of the passed data type argument.

Java – Arrays

Java provides a data structure, the **array**, which stores a fixed-size sequential collection of elements of the same type. An array is used to store a collection of data, but it is often more useful to think of an array as a collection of variables of the same type.

Instead of declaring individual variables, such as number0, number1, ..., and number99, you declare one array variable such as numbers and use numbers[0], numbers[1], and ..., numbers[99] to represent individual variables.

This tutorial introduces how to declare array variables, create arrays, and process arrays using indexed variables.

Declaring Array Variables:

To use an array in a program, you must declare a variable to reference the array, and you must specify the type of array the variable can reference. Here is the syntax for declaring an array variable:

```
dataType[] arrayRefVar;   // preferred way.


or


dataType arrayRefVar[];  //  works but not preferred way.
```

Note: The style **dataType[] arrayRefVar** is preferred. The style **dataType arrayRefVar[]** comes from the C/C++ language and was adopted in Java to accommodate C/C++ programmers.

Example:

The following code snippets are examples of this syntax:

```
double[] myList;         // preferred way.


or


double myList[];         //  works but not preferred way.
```

Creating Arrays:

You can create an array by using the new operator with the following syntax:

```
arrayRefVar = new dataType[arraySize];
```

The above statement does two things:

- It creates an array using new dataType[arraySize];
- It assigns the reference of the newly created array to the variable arrayRefVar.

Declaring an array variable, creating an array, and assigning the reference of the array to the variable can be combined in one statement, as shown below:

```
dataType[] arrayRefVar = new dataType[arraySize];
```

Alternatively you can create arrays as follows:

```
dataType[] arrayRefVar = {value0, value1, ..., valuek};
```

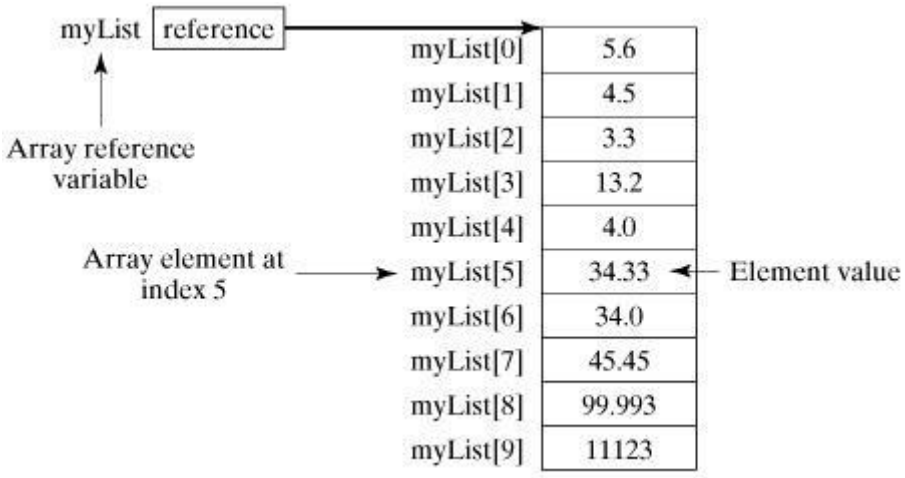
The array elements are accessed through the **index**. Array indices are 0-based; that is, they start from 0 to **arrayRefVar.length-1**.

Example:

Following statement declares an array variable, myList, creates an array of 10 elements of double type, and assigns its reference to myList.:

```
double[] myList = new double[10];
```

Following picture represents array myList. Here myList holds ten double values and the indices are from 0 to 9.



Processing Arrays:

When processing array elements, we often use either for loop or foreach loop because all of the elements in an array are of the same type and the size of the array is known.

Example:

Here is a complete example of showing how to create, initialize and process arrays:

```
public class TestArray {

   public static void main(String[] args) {
      double[] myList = {1.9, 2.9, 3.4, 3.5};

      // Print all the array elements
```

```
      for (int i = 0; i < myList.length; i++) {
         System.out.println(myList[i] + " ");
      }
      // Summing all elements
      double total = 0;
      for (int i = 0; i < myList.length; i++) {
         total += myList[i];
      }
      System.out.println("Total is " + total);
      // Finding the largest element
      double max = myList[0];
      for (int i = 1; i < myList.length; i++) {
         if (myList[i] > max) max = myList[i];
      }
      System.out.println("Max is " + max);
   }
}
```

This would produce following result:

```
1.9
2.9
3.4
3.5
Total is 11.7
Max is 3.5
```

The foreach Loops:

JDK 1.5 introduced a new for loop, known as foreach loop or enhanced for loop, which enables you to traverse the complete array sequentially without using an index variable.

Example:

The following code displays all the elements in the array myList:

```
public class TestArray {


   public static void main(String[] args) {
      double[] myList = {1.9, 2.9, 3.4, 3.5};


      // Print all the array elements
      for (double element: myList) {
         System.out.println(element);
```

```
      }
   }
}
```

This would produce following result:

```
1.9
2.9
3.4
3.5
```

Passing Arrays to Methods:

Just as you can pass primitive type values to methods, you can also pass arrays to methods. For example, the following method displays the elements in an int array:

```
public static void printArray(int[] array) {
  for (int i = 0; i < array.length; i++) {
    System.out.print(array[i] + " ");
  }
}
```

You can invoke it by passing an array. For example, the following statement invokes the printArray method to display 3, 1, 2, 6, 4, and 2:

```
printArray(new int[]{3, 1, 2, 6, 4, 2});
```

Returning an Array from a Method:

A method may also return an array. For example, the method shown below returns an array that is the reversal of another array:

```
public static int[] reverse(int[] list) {
  int[] result = new int[list.length];

  for (int i = 0, j = result.length - 1; i < list.length; i++, j--) {
    result[j] = list[i];
  }
  return result;
}
```

The Arrays Class:

The java.util.Arrays class contains various static methods for sorting and searching arrays, comparing arrays, and filling array elements. These methods are overloaded for all primitive types.

SN	Methods with Description

1	**public static int binarySearch(Object[] a, Object key)** Searches the specified array of Object (Byte, Int , double etc) for the specified value using the binary search algorithm. The array must be sorted prior to making this call. This returns index of the search key, if it is contained in the list; otherwise, (-(insertion point + 1).
2	**public static boolean equals(long[] a, long[] a2)** Returns true if the two specified arrays of longs are equal to one another. Two arrays are considered equal if both arrays contain the same number of elements, and all corresponding pairs of elements in the two arrays are equal. This returns true if the two arrays are equal. Same method could be used by all other premitive data types (Byte, short, Int etc.)
3	**public static void fill(int[] a, int val)** Assigns the specified int value to each element of the specified array of ints. Same method could be used by all other premitive data types (Byte, short, Int etc.)
4	**public static void sort(Object[] a)** Sorts the specified array of objects into ascending order, according to the natural ordering of its elements. Same method could be used by all other premitive data types (Byte, short, Int etc.)

Java - Date & Time

Java provides the **Date** class available in **java.util** package, this class encapsulates the current date and time.

The Date class supports two constructors. The first constructor initializes the object with the current date and time.

```
Date( )
```

The following constructor accepts one argument that equals the number of milliseconds that have elapsed since midnight, January 1, 1970

```
Date(long millisec)
```

Once you have a Date object available, you can call any of the following support methods to play with dates:

SN	Methods with Description
1	**boolean after(Date date)** Returns true if the invoking Date object contains a date that is later than the one specified by date, otherwise, it returns false.
2	**boolean before(Date date)** Returns true if the invoking Date object contains a date that is earlier than the one specified by date, otherwise, it returns false.
3	**Object clone( )** Duplicates the invoking Date object.
4	**int compareTo(Date date)** Compares the value of the invoking object with that of date. Returns 0 if the values are equal. Returns a negative value if the invoking object is earlier than date. Returns a positive value if the invoking object is later

	than date.
5	**int compareTo(Object obj)** Operates identically to compareTo(Date) if obj is of class Date. Otherwise, it throws a ClassCastException.
6	**boolean equals(Object date)** Returns true if the invoking Date object contains the same time and date as the one specified by date, otherwise, it returns false.
7	**long getTime( )** Returns the number of milliseconds that have elapsed since January 1, 1970.
8	**int hashCode( )** Returns a hash code for the invoking object.
9	**void setTime(long time)** Sets the time and date as specified by time, which represents an elapsed time in milliseconds from midnight, January 1, 1970
10	**String toString( )** Converts the invoking Date object into a string and returns the result.

Getting Current Date & Time

This is very easy to get current date and time in Java. You can use a simple Date object with *toString()* method to print current date and time as follows:

```
import java.util.Date;


public class DateDemo {

   public static void main(String args[]) {

       // Instantiate a Date object

       Date date = new Date();


       // display time and date using toString()

       System.out.println(date.toString());

   }

}
```

This would produce following result:

```
Mon May 04 09:51:52 CDT 2009
```

Date Comparison:

There are following three ways to compare two dates:

- You can use getTime( ) to obtain the number of milliseconds that have elapsed since midnight, January 1, 1970, for both objects and then compare these two values.

- You can use the methods before( ), after( ), and equals( ). Because the 12th of the month comes before the 18th, for example, new Date(99, 2, 12).before(new Date (99, 2, 18)) returns true.

- You can use the compareTo( ) method, which is defined by the Comparable interface and implemented by Date.

Date Formatting using SimpleDateFormat:

SimpleDateFormat is a concrete class for formatting and parsing dates in a locale-sensitive manner. SimpleDateFormat allows you to start by choosing any user-defined patterns for date-time formatting. For example:

```
import java.util.*;
import java.text.*;


public class DateDemo {
   public static void main(String args[]) {


      Date dNow = new Date( );
      SimpleDateFormat ft =
      new SimpleDateFormat ("E yyyy.MM.dd 'at' hh:mm:ss a zzz");


      System.out.println("Current Date: " + ft.format(dNow));
   }
}
```

This would produce following result:

```
Current Date: Sun 2004.07.18 at 04:14:09 PM PDT
```

Simple DateFormat format codes:

To specify the time format use a time pattern string. In this pattern, all ASCII letters are reserved as pattern letters, which are defined as the following:

Character	Description	Example
G	Era designator	AD
Y	Year in four digits	2001
M	Month in year	July or 07
D	Day in month	10
H	Hour in A.M./P.M. (1~12)	12

H	Hour in day (0~23)	22
M	Minute in hour	30
S	Second in minute	55
S	Millisecond	234
E	Day in week	Tuesday
D	Day in year	360
F	Day of week in month	2 (second Wed. in July)
W	Week in year	40
W	Week in month	1
A	A.M./P.M. marker	PM
K	Hour in day (1~24)	24
K	Hour in A.M./P.M. (0~11)	10
Z	Time zone	Eastern Standard Time
'	Escape for text	Delimiter
"	Single quote	`

Date Formatting using printf:

Date and time formatting can be done very easily using **printf** method. You use a two-letter format, starting with **t** and ending in one of the letters of the table given below. For example:

```
import java.util.Date;


public class DateDemo {


  public static void main(String args[]) {
     // Instantiate a Date object
     Date date = new Date();


     // display time and date using toString()
```

```
     String str = String.format("Current Date/Time : %tc", date );


     System.out.printf(str);
  }
}
```

This would produce following result:

```
Current Date/Time : Sat Dec 15 16:37:57 MST 2012
```

It would be a bit silly if you had to supply the date multiple times to format each part. For that reason, a format string can indicate the index of the argument to be formatted.

The index must immediately follow the %, and it must be terminated by a $. For example:

```
import java.util.Date;


public class DateDemo {


   public static void main(String args[]) {
       // Instantiate a Date object
       Date date = new Date();


       // display time and date using toString()
       System.out.printf("%1$s %2$tB %2$td, %2$tY",
                         "Due date:", date);
   }
}
```

This would produce following result:

```
Due date: February 09, 2004
```

Alternatively, you can use the < flag. It indicates that the same argument as in the preceding format specification should be used again. For example:

```
import java.util.Date;


public class DateDemo {


   public static void main(String args[]) {
       // Instantiate a Date object
       Date date = new Date();


       // display formatted date
       System.out.printf("%s %tB %<te, %<tY",
```

```
                        "Due date:", date);
   }
}
```

This would produce following result:

```
Due date: February 09, 2004
```

Date and Time Conversion Characters:

Character	Description	Example
C	Complete date and time	Mon May 04 09:51:52 CDT 2009
F	ISO 8601 date	2004-02-09
D	U.S. formatted date (month/day/year)	02/09/2004
T	24-hour time	18:05:19
R	12-hour time	06:05:19 pm
R	24-hour time, no seconds	18:05
Y	Four-digit year (with leading zeroes)	2004
Y	Last two digits of the year (with leading zeroes)	04
C	First two digits of the year (with leading zeroes)	20
B	Full month name	February
B	Abbreviated month name	Feb
N	Two-digit month (with leading zeroes)	02
D	Two-digit day (with leading zeroes)	03
E	Two-digit day (without leading zeroes)	9
A	Full weekday name	Monday
A	Abbreviated weekday name	Mon
J	Three-digit day of year (with leading zeroes)	069

H	Two-digit hour (with leading zeroes), between 00 and 23	18
K	Two-digit hour (without leading zeroes), between 0 and 23	18
I	Two-digit hour (with leading zeroes), between 01 and 12	06
L	Two-digit hour (without leading zeroes), between 1 and 12	6
M	Two-digit minutes (with leading zeroes)	05
S	Two-digit seconds (with leading zeroes)	19
L	Three-digit milliseconds (with leading zeroes)	047
N	Nine-digit nanoseconds (with leading zeroes)	047000000
P	Uppercase morning or afternoon marker	PM
P	Lowercase morning or afternoon marker	Pm
Z	RFC 822 numeric offset from GMT	-0800
Z	Time zone	PST
S	Seconds since 1970-01-01 00:00:00 GMT	1078884319
Q	Milliseconds since 1970-01-01 00:00:00 GMT	1078884319047

There are other useful classes related to Date and time. For more detail you can refer to Java Standard documentation.

Parsing Strings into Dates:

The SimpleDateFormat class has some additional methods, notably parse( ) , which tries to parse a string according to the format stored in the given SimpleDateFormat object. For example:

```
import java.util.*;
import java.text.*;

public class DateDemo {

   public static void main(String args[]) {
      SimpleDateFormat ft = new SimpleDateFormat ("yyyy-MM-dd");

      String input = args.length == 0 ? "1818-11-11" : args[0];
```

```
      System.out.print(input + " Parses as ");

      Date t;

      try {
          t = ft.parse(input);
          System.out.println(t);
      } catch (ParseException e) {
          System.out.println("Unparseable using " + ft);
      }
   }
}
```

A sample run of the above program would produce following result:

```
$ java DateDemo
1818-11-11 Parses as Wed Nov 11 00:00:00 GMT 1818
$ java DateDemo 2007-12-01
2007-12-01 Parses as Sat Dec 01 00:00:00 GMT 2007
```

Sleeping for a While:

You can sleep for any period of time from one millisecond up to the lifetime of your computer. For example, following program would sleep for 10 seconds:

```
import java.util.*;

public class SleepDemo {
   public static void main(String args[]) {
      try {
         System.out.println(new Date( ) + "\n");
         Thread.sleep(5*60*10);
         System.out.println(new Date( ) + "\n");
      } catch (Exception e) {
          System.out.println("Got an exception!");
      }
   }
}
```

This would produce following result:

```
Sun May 03 18:04:41 GMT 2009
```

```
Sun May 03 18:04:51 GMT 2009
```

Measuring Elapsed Time:

Sometime you may need to measure point in time in milliseconds. So let's re-write above example once again:

```
import java.util.*;


public class DiffDemo {


   public static void main(String args[]) {
      try {
         long start = System.currentTimeMillis( );
         System.out.println(new Date( ) + "\n");
         Thread.sleep(5*60*10);
         System.out.println(new Date( ) + "\n");
         long end = System.currentTimeMillis( );
         long diff = end - start;
         System.out.println("Difference is : " + diff);
      } catch (Exception e) {
         System.out.println("Got an exception!");
      }
   }
}
```

This would produce following result:

```
Sun May 03 18:16:51 GMT 2009


Sun May 03 18:16:57 GMT 2009


Difference is : 5993
```

GregorianCalendar Class:

GregorianCalendar is a concrete implementation of a Calendar class that implements the normal Gregorian calendar with which you are familiar. I did not discuss Calender class in this tutorial, you can look standard Java documentation for this.

The **getInstance( )** method of Calendar returns a GregorianCalendar initialized with the current date and time in the default locale and time zone. GregorianCalendar defines two fields: AD and BC. These represent the two eras defined by the Gregorian calendar.

There are also several constructors for GregorianCalendar objects:

SN	Constructor with Description

1	**GregorianCalendar()** Constructs a default GregorianCalendar using the current time in the default time zone with the default locale.
2	**GregorianCalendar(int year, int month, int date)** Constructs a GregorianCalendar with the given date set in the default time zone with the default locale.
3	**GregorianCalendar(int year, int month, int date, int hour, int minute)** Constructs a GregorianCalendar with the given date and time set for the default time zone with the default locale.
4	**GregorianCalendar(int year, int month, int date, int hour, int minute, int second)** Constructs a GregorianCalendar with the given date and time set for the default time zone with the default locale.
5	**GregorianCalendar(Locale aLocale)** Constructs a GregorianCalendar based on the current time in the default time zone with the given locale.
6	**GregorianCalendar(TimeZone zone)** Constructs a GregorianCalendar based on the current time in the given time zone with the default locale.
7	**GregorianCalendar(TimeZone zone, Locale aLocale)** Constructs a GregorianCalendar based on the current time in the given time zone with the given locale.

Here is the list of few useful support methods provided by GregorianCalendar class:

SN	Medthos with Description
1	**void add(int field, int amount)** Adds the specified (signed) amount of time to the given time field, based on the calendar's rules.
2	**protected void computeFields()** Converts UTC as milliseconds to time field values.
3	**protected void computeTime()** Overrides Calendar Converts time field values to UTC as milliseconds.
4	**boolean equals(Object obj)** Compares this GregorianCalendar to an object reference.
5	**int get(int field)** Gets the value for a given time field.
6	**int getActualMaximum(int field)** Return the maximum value that this field could have, given the current date.
7	**int getActualMinimum(int field)** Return the minimum value that this field could have, given the current date.

8	**int getGreatestMinimum(int field)** Returns highest minimum value for the given field if varies.
9	**Date getGregorianChange()** Gets the Gregorian Calendar change date.
10	**int getLeastMaximum(int field)** Returns lowest maximum value for the given field if varies.
11	**int getMaximum(int field)** Returns maximum value for the given field.
12	**Date getTime()** Gets this Calendar's current time.
13	**long getTimeInMillis()** Gets this Calendar's current time as a long.
14	**TimeZone getTimeZone()** Gets the time zone.
15	**int getMinimum(int field)** Returns minimum value for the given field.
16	**int hashCode()** Override hashCode.
17	**boolean isLeapYear(int year)** Determines if the given year is a leap year.
18	**void roll(int field, boolean up)** Adds or subtracts (up/down) a single unit of time on the given time field without changing larger fields.
19	**void set(int field, int value)** Sets the time field with the given value.
20	**void set(int year, int month, int date)** Sets the values for the fields year, month, and date.
21	**void set(int year, int month, int date, int hour, int minute)** Sets the values for the fields year, month, date, hour, and minute.
22	**void set(int year, int month, int date, int hour, int minute, int second)** Sets the values for the fields year, month, date, hour, minute, and second.
23	**void setGregorianChange(Date date)** Sets the GregorianCalendar change date.

24	**void setTime(Date date)** Sets this Calendar's current time with the given Date.
25	**void setTimeInMillis(long millis)** Sets this Calendar's current time from the given long value.
26	**void setTimeZone(TimeZone value)** Sets the time zone with the given time zone value.
27	**String toString()** Return a string representation of this calendar.

Example:

```
import java.util.*;


public class GregorianCalendarDemo {


   public static void main(String args[]) {
      String months[] = {
      "Jan", "Feb", "Mar", "Apr",
      "May", "Jun", "Jul", "Aug",
      "Sep", "Oct", "Nov", "Dec"};


      int year;
      // Create a Gregorian calendar initialized
      // with the current date and time in the
      // default locale and timezone.
      GregorianCalendar gcalendar = new GregorianCalendar();
      // Display current time and date information.
      System.out.print("Date: ");
      System.out.print(months[gcalendar.get(Calendar.MONTH)]);
      System.out.print(" " + gcalendar.get(Calendar.DATE) + " ");
      System.out.println(year = gcalendar.get(Calendar.YEAR));
      System.out.print("Time: ");
      System.out.print(gcalendar.get(Calendar.HOUR) + ":");
      System.out.print(gcalendar.get(Calendar.MINUTE) + ":");
      System.out.println(gcalendar.get(Calendar.SECOND));


      // Test if the current year is a leap year
      if(gcalendar.isLeapYear(year)) {
         System.out.println("The current year is a leap year");
      }
```

```
      else {
         System.out.println("The current year is not a leap year");
      }
   }
}
```

This would produce following result:

```
Date: Apr 22 2009
Time: 11:25:27
The current year is not a leap year
```

For a complete list of constant available in Calender class, you can refer to standard Java documentation.

JAVA REGULAR EXPRESSIONS

Java provides the java.util.regex package for pattern matching with regular expressions. Java regular expressions are very similar to the Perl programming language and very easy to learn.

A regular expression is a special sequence of characters that helps you match or find other strings or sets of strings, using a specialized syntax held in a pattern. They can be used to search, edit, or manipulate text and data.

The java.util.regex package primarily consists of the following three classes:

- **Pattern Class:** A Pattern object is a compiled representation of a regular expression. The Pattern class provides no public constructors. To create a pattern, you must first invoke one of its public static compile methods, which will then return a Pattern object. These methods accept a regular expression as the first argument.

- **Matcher Class:** A Matcher object is the engine that interprets the pattern and performs match operations against an input string. Like the Pattern class, Matcher defines no public constructors. You obtain a Matcher object by invoking the matcher method on a Pattern object.

- **PatternSyntaxException:** A PatternSyntaxException object is an unchecked exception that indicates a syntax error in a regular expression pattern.

Capturing Groups:

Capturing groups are a way to treat multiple characters as a single unit. They are created by placing the characters to be grouped inside a set of parentheses. For example, the regular expression (dog) creates a single group containing the letters "d", "o", and "g".

Capturing groups are numbered by counting their opening parentheses from left to right. In the expression ((A)(B(C))), for example, there are four such groups:

- ((A)(B(C)))

- (A)

- (B(C))

- (C)

To find out how many groups are present in the expression, call the groupCount method on a matcher object. The groupCount method returns an int showing the number of capturing groups present in the matcher's pattern.

There is also a special group, group 0, which always represents the entire expression. This group is not included in the total reported by groupCount.

Example:

Following example illustrate how to find a digit string from the given alphanumeric string:

```
import java.util.regex.Matcher;
import java.util.regex.Pattern;

public class RegexMatches
{
    public static void main( String args[] ){

      // String to be scanned to find the pattern.
      String line = "This order was places for QT3000! OK?";
      String pattern = "(.*)(\\d+)(.*)";

      // Create a Pattern object
      Pattern r = Pattern.compile(pattern);

      // Now create matcher object.
      Matcher m = r.matcher(line);
      if (m.find( )) {
         System.out.println("Found value: " + m.group(0) );
         System.out.println("Found value: " + m.group(1) );
         System.out.println("Found value: " + m.group(2) );
      } else {
         System.out.println("NO MATCH");
      }
   }
}
```

This would produce following result:

```
Found value: This order was places for QT3000! OK?
Found value: This order was places for QT300
Found value: 0
```

Regular Expression Syntax:

Here is the table listing down all the regular expression metacharacter syntax available in Java:

Subexpression	Matches
^	Matches beginning of line.
$	Matches end of line.
.	Matches any single character except newline. Using m option allows it to match newline as well.
[...]	Matches any single character in brackets.
[^...]	Matches any single character not in brackets
\A	Beginning of entire string
\z	End of entire string
\Z	End of entire string except allowable final line terminator.
re*	Matches 0 or more occurrences of preceding expression.
re+	Matches 1 or more of the previous thing
re?	Matches 0 or 1 occurrence of preceding expression.
re{ n}	Matches exactly n number of occurrences of preceding expression.
re{ n,}	Matches n or more occurrences of preceding expression.
re{ n, m}	Matches at least n and at most m occurrences of preceding expression.
a| b	Matches either a or b.
(re)	Groups regular expressions and remembers matched text.
(?: re)	Groups regular expressions without remembering matched text.
(?> re)	Matches independent pattern without backtracking.
\w	Matches word characters.
\W	Matches nonword characters.
\s	Matches whitespace. Equivalent to [\t\n\r\f].

\S	Matches nonwhitespace.
\d	Matches digits. Equivalent to [0-9].
\D	Matches nondigits.
\A	Matches beginning of string.
\Z	Matches end of string. If a newline exists, it matches just before newline.
\z	Matches end of string.
\G	Matches point where last match finished.
\n	Back-reference to capture group number "n"
\b	Matches word boundaries when outside brackets. Matches backspace (0x08) when inside brackets.
\B	Matches nonword boundaries.
\n, \t, etc.	Matches newlines, carriage returns, tabs, etc.
\Q	Escape (quote) all characters up to \E
\E	Ends quoting begun with \Q

Methods of the Matcher Class:

Here is the lists of useful instance methods:

Index Methods:

Index methods provide useful index values that show precisely where the match was found in the input string:

SN	Methods with Description
1	**public int start()** Returns the start index of the previous match.
2	**public int start(int group)** Returns the start index of the subsequence captured by the given group during the previous match operation.
3	**public int end()** Returns the offset after the last character matched.

4	**public int end(int group)** Returns the offset after the last character of the subsequence captured by the given group during the previous match operation.

Study Methods:

Study methods review the input string and return a boolean indicating whether or not the pattern is found:

SN	Methods with Description
1	**public boolean lookingAt()** Attempts to match the input sequence, starting at the beginning of the region, against the pattern.
2	**public boolean find()** Attempts to find the next subsequence of the input sequence that matches the pattern.
3	**public boolean find(int start** Resets this matcher and then attempts to find the next subsequence of the input sequence that matches the pattern, starting at the specified index.
4	**public boolean matches()** Attempts to match the entire region against the pattern.

Replacement Methods:

Replacement methods are useful methods for replacing text in an input string:

SN	Methods with Description
1	**public Matcher appendReplacement(StringBuffer sb, String replacement)** Implements a non-terminal append-and-replace step.
2	**public StringBuffer appendTail(StringBuffer sb)** Implements a terminal append-and-replace step.
3	**public String replaceAll(String replacement)** Replaces every subsequence of the input sequence that matches the pattern with the given replacement string.
4	**public String replaceFirst(String replacement)** Replaces the first subsequence of the input sequence that matches the pattern with the given replacement string.
5	**public static String quoteReplacement(String s)** Returns a literal replacement String for the specified String. This method produces a String that will work as a literal replacement s in the appendReplacement method of the Matcher class.

The *start* and *end* Methods:

Following is the example that counts the number of times the word "cats" appears in the input string:

```
import java.util.regex.Matcher;
import java.util.regex.Pattern;

public class RegexMatches
{
    private static final String REGEX = "\\bcat\\b";
    private static final String INPUT =
                                    "cat cat cat cattie cat";

    public static void main( String args[] ){
       Pattern p = Pattern.compile(REGEX);
       Matcher m = p.matcher(INPUT); // get a matcher object
       int count = 0;

       while(m.find()) {
         count++;
         System.out.println("Match number "+count);
         System.out.println("start(): "+m.start());
         System.out.println("end(): "+m.end());
      }
   }
}
```

This would produce following result:

```
Match number 1
start(): 0
end(): 3
Match number 2
start(): 4
end(): 7
Match number 3
start(): 8
end(): 11
Match number 4
start(): 19
end(): 22
```

You can see that this example uses word boundaries to ensure that the letters "c" "a" "t" are not merely a substring in a longer word. It also gives some useful information about where in the input string the match has occurred.

The start method returns the start index of the subsequence captured by the given group during the previous match operation, and end returns the index of the last character matched, plus one.

The *matches* and *lookingAt* Methods:

The matches and lookingAt methods both attempt to match an input sequence against a pattern. The difference, however, is that matches requires the entire input sequence to be matched, while lookingAt does not.

Both methods always start at the beginning of the input string. Here is the example explaining the functionality:

```
import java.util.regex.Matcher;
import java.util.regex.Pattern;

public class RegexMatches
{
    private static final String REGEX = "foo";
    private static final String INPUT = "fooooooooooooooooo";
    private static Pattern pattern;
    private static Matcher matcher;

    public static void main( String args[] ){
       pattern = Pattern.compile(REGEX);
       matcher = pattern.matcher(INPUT);

       System.out.println("Current REGEX is: "+REGEX);
       System.out.println("Current INPUT is: "+INPUT);

       System.out.println("lookingAt(): "+matcher.lookingAt());
       System.out.println("matches(): "+matcher.matches());
   }
}
```

This would produce following result:

```
Current REGEX is: foo
Current INPUT is: foooooooooooooooo
lookingAt(): true
matches(): false
```

The *replaceFirst* and *replaceAll* Methods:

The replaceFirst and replaceAll methods replace text that matches a given regular expression. As their names indicate, replaceFirst replaces the first occurrence, and replaceAll replaces all occurences.

Here is the example explaining the functionality:

```
import java.util.regex.Matcher;
import java.util.regex.Pattern;
```

```
public class RegexMatches
{
    private static String REGEX = "dog";
    private static String INPUT = "The dog says meow. " +
                                  "All dogs say meow.";
    private static String REPLACE = "cat";

    public static void main(String[] args) {
       Pattern p = Pattern.compile(REGEX);
       // get a matcher object
       Matcher m = p.matcher(INPUT);
       INPUT = m.replaceAll(REPLACE);
       System.out.println(INPUT);
   }
}
```

This would produce following result:

```
The cat says meow. All cats say meow.
```

The *appendReplacement* and *appendTail* Methods:

The Matcher class also provides appendReplacement and appendTail methods for text replacement.

Here is the example explaining the functionality:

```
import java.util.regex.Matcher;
import java.util.regex.Pattern;

public class RegexMatches
{
   private static String REGEX = "a*b";
   private static String INPUT = "aabfooaabfooabfoob";
   private static String REPLACE = "-";
   public static void main(String[] args) {
      Pattern p = Pattern.compile(REGEX);
      // get a matcher object
      Matcher m = p.matcher(INPUT);
      StringBuffer sb = new StringBuffer();
      while(m.find()){
         m.appendReplacement(sb,REPLACE);
      }
      m.appendTail(sb);
```

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

This would produce following result:

```
-foo-foo-foo-
```

PatternSyntaxException Class Methods:

A PatternSyntaxException is an unchecked exception that indicates a syntax error in a regular expression pattern. The PatternSyntaxException class provides the following methods to help you determine what went wrong:

SN	Methods with Description
1	**public String getDescription()** Retrieves the description of the error.
2	**public int getIndex()** Retrieves the error index.
3	**public String getPattern()** Retrieves the erroneous regular expression pattern.
4	**public String getMessage()** Returns a multi-line string containing the description of the syntax error and its index, the erroneous regular expression pattern, and a visual indication of the error index within the pattern.

Java – Methods

A Java method is a collection of statements that are grouped together to perform an operation. When you call the System.out.println method, for example, the system actually executes several statements in order to display a message on the console.

Now you will learn how to create your own methods with or without return values, invoke a method with or without parameters, overload methods using the same names, and apply method abstraction in the program design.

Creating a Method:

In general, a method has the following syntax:

```
modifier returnValueType methodName(list of parameters) {
  // Method body;
}
```

A method definition consists of a method header and a method body. Here are all the parts of a method:

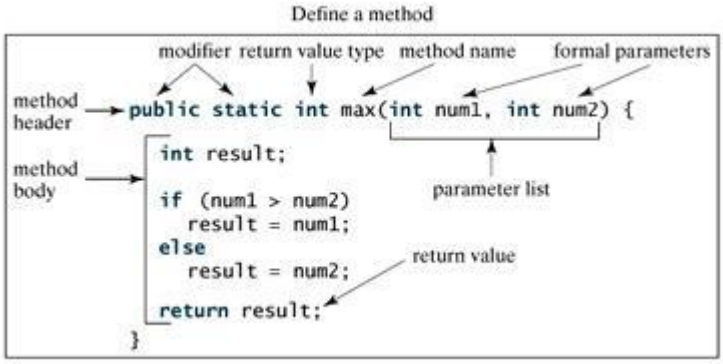
- **Modifiers:** The modifier, which is optional, tells the compiler how to call the method. This defines the access type of the method.

- **Return Type:** A method may return a value. The returnValueType is the data type of the value the method returns. Some methods perform the desired operations without returning a value. In this case, the returnValueType is the keyword **void**.

- **Method Name:** This is the actual name of the method. The method name and the parameter list together constitute the method signature.

- **Parameters:** A parameter is like a placeholder. When a method is invoked, you pass a value to the parameter. This value is referred to as actual parameter or argument. The parameter list refers to the type, order, and number of the parameters of a method. Parameters are optional; that is, a method may contain no parameters.

- **Method Body:** The method body contains a collection of statements that define what the method does.



Note: In certain other languages, methods are referred to as procedures and functions. A method with a nonvoid return value type is called a function; a method with a void return value type is called a procedure.

Example:

Here is the source code of the above defined method called max(). This method takes two parameters num1 and num2 and returns the maximum between the two:

```
/** Return the max between two numbers */
public static int max(int num1, int num2) {
   int result;
   if (num1 > num2)
      result = num1;
   else
      result = num2;

   return result;
}
```

Calling a Method:

In creating a method, you give a definition of what the method is to do. To use a method, you have to call or invoke it. There are two ways to call a method; the choice is based on whether the method returns a value or not.

When a program calls a method, program control is transferred to the called method. A called method returns control to the caller when its return statement is executed or when its method-ending closing brace is reached.

If the method returns a value, a call to the method is usually treated as a value. For example:

```
int larger = max(30, 40);
```

If the method returns void, a call to the method must be a statement. For example, the method println returns void. The following call is a statement:

```
System.out.println("Welcome to Java!");
```

Example:

Following is the example to demonstrate how to define a method and how to call it:

```
public class TestMax {
   /** Main method */
   public static void main(String[] args) {
      int i = 5;
      int j = 2;
      int k = max(i, j);
      System.out.println("The maximum between " + i +
                  " and " + j + " is " + k);
   }

   /** Return the max between two numbers */
   public static int max(int num1, int num2) {
      int result;
      if (num1 > num2)
         result = num1;
      else
         result = num2;

      return result;
   }
}
```

This would produce following result:

```
The maximum between 5 and 2 is 5
```

This program contains the main method and the max method. The main method is just like any other method except that it is invoked by the JVM.

The main method's header is always the same, like the one in this example, with the modifiers public and static, return value type void, method name main, and a parameter of the String[] type. String[] indicates that the parameter is an array of String.

The void Keyword:

This section shows how to declare and invoke a void method. Following example gives a program that declares a method named printGrade and invokes it to print the grade for a given score.

Example:

```
public class TestVoidMethod {


   public static void main(String[] args) {
      printGrade(78.5);
   }


   public static void printGrade(double score) {
      if (score >= 90.0) {
         System.out.println('A');
      }
      else if (score >= 80.0) {
         System.out.println('B');
      }
      else if (score >= 70.0) {
         System.out.println('C');
      }
      else if (score >= 60.0) {
         System.out.println('D');
      }
      else {
         System.out.println('F');
      }
   }
}
```

This would produce following result:

```
C
```

Here the printGrade method is a void method. It does not return any value. A call to a void method must be a statement. So, it is invoked as a statement in line 3 in the main method. This statement is like any Java statement terminated with a semicolon.

Passing Parameters by Values:

When calling a method, you need to provide arguments, which must be given in the same order as their respective parameters in the method specification. This is known as parameter order association.

For example, the following method prints a message n times:

```
public static void nPrintln(String message, int n) {
  for (int i = 0; i < n; i++)
    System.out.println(message);
}
```

Here, you can use nPrintln("Hello", 3) to print "Hello" three times. The nPrintln("Hello", 3) statement passes the actual string parameter, "Hello", to the parameter, message; passes 3 to n; and prints "Hello" three times. However, the statement nPrintln(3, "Hello") would be wrong.

When you invoke a method with a parameter, the value of the argument is passed to the parameter. This is referred to as pass-by-value. If the argument is a variable rather than a literal value, the value of the variable is passed to the parameter. The variable is not affected, regardless of the changes made to the parameter inside the method.

For simplicity, Java programmers often say passing an argument x to a parameter y, which actually means passing the value of x to y.

Example:

Following is a program that demonstrates the effect of passing by value. The program creates a method for swapping two variables. The swap method is invoked by passing two arguments. Interestingly, the values of the arguments are not changed after the method is invoked.

```
public class TestPassByValue {


   public static void main(String[] args) {
      int num1 = 1;
      int num2 = 2;


      System.out.println("Before swap method, num1 is " +
                          num1 + " and num2 is " + num2);


      // Invoke the swap method
      swap(num1, num2);
      System.out.println("After swap method, num1 is " +
                         num1 + " and num2 is " + num2);
   }
   /** Method to swap two variables */
   public static void swap(int n1, int n2) {
      System.out.println("\tInside the swap method");
      System.out.println("\t\tBefore swapping n1 is " + n1
                           + " n2 is " + n2);
      // Swap n1 with n2
      int temp = n1;
      n1 = n2;
      n2 = temp;
```

```
      System.out.println("\t\tAfter swapping n1 is " + n1
                         + " n2 is " + n2);
   }
}
```

This would produce following result:

```
Before swap method, num1 is 1 and num2 is 2
        Inside the swap method
                Before swapping n1 is 1 n2 is 2
                After swapping n1 is 2 n2 is 1
After swap method, num1 is 1 and num2 is 2
```

Overloading Methods:

The max method that was used earlier works only with the int data type. But what if you need to find which of two floating-point numbers has the maximum value? The solution is to create another method with the same name but different parameters, as shown in the following code:

```
public static double max(double num1, double num2) {
  if (num1 > num2)
    return num1;
  else
    return num2;
}
```

If you call max with int parameters, the max method that expects int parameters will be invoked; if you call max with double parameters, the max method that expects double parameters will be invoked. This is referred to as **method overloading**; that is, two methods have the same name but different parameter lists within one class.

The Java compiler determines which method is used based on the method signature. Overloading methods can make programs clearer and more readable. Methods that perform closely related tasks should be given the same name.

Overloaded methods must have different parameter lists. You cannot overload methods based on different modifiers or return types. Sometimes there are two or more possible matches for an invocation of a method due to similar method signature, so the compiler cannot determine the most specific match. This is referred to as ambiguous invocation.

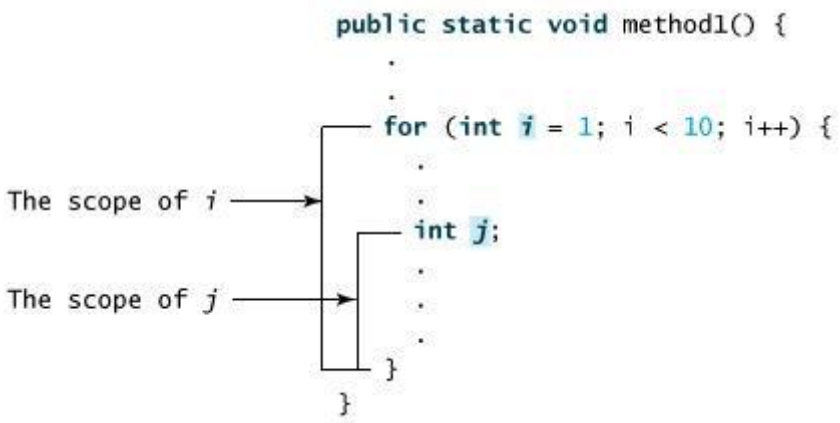
The Scope of Variables:

The scope of a variable is the part of the program where the variable can be referenced. A variable defined inside a method is referred to as a local variable.

The scope of a local variable starts from its declaration and continues to the end of the block that contains the variable. A local variable must be declared before it can be used.

A parameter is actually a local variable. The scope of a method parameter covers the entire method.

A variable declared in the initial action part of a for loop header has its scope in the entire loop. But a variable declared inside a for loop body has its scope limited in the loop body from its declaration to the end of the block that contains the variable as shown below:



You can declare a local variable with the same name multiple times in different non-nesting blocks in a method, but you cannot declare a local variable twice in nested blocks.

Using Command-Line Arguments:

Sometimes you will want to pass information into a program when you run it. This is accomplished by passing command-line arguments to main( ).

A command-line argument is the information that directly follows the program's name on the command line when it is executed. To access the command-line arguments inside a Java program is quite easy.they are stored as strings in the String array passed to main( ).

Example:

The following program displays all of the command-line arguments that it is called with:

```
public class CommandLine {


   public static void main(String args[]){
      for(int i=0; i<args.length; i++){
         System.out.println("args[" + i + "]: " +
                                            args[i]);
      }
   }
}
```

Try executing this program, as shown here:

```
java CommandLine this is a command line 200 -100
```

This would produce following result:

```
args[0]: this
args[1]: is
args[2]: a
args[3]: command
```

```
args[4]: line
args[5]: 200
args[6]: -100
```

The Constructors:

A constructor initializes an object when it is created. It has the same name as its class and is syntactically similar to a method. However, constructors have no explicit return type.

Typically, you will use a constructor to give initial values to the instance variables defined by the class, or to perform any other startup procedures required to create a fully formed object.

All classes have constructors, whether you define one or not, because Java automatically provides a default constructor that initializes all member variables to zero. However, once you define your own constructor, the default constructor is no longer used.

Example:

Here is a simple example that uses a constructor:

```
// A simple constructor.
class MyClass {
   int x;

   // Following is the constructor
   MyClass() {
      x = 10;
   }
}
```

You would call constructor to initialize objects as follows:

```
public class ConsDemo {

   public static void main(String args[]) {
      MyClass t1 = new MyClass();
      MyClass t2 = new MyClass();
      System.out.println(t1.x + " " + t2.x);
   }
}
```

Most often you will need a constructor that accepts one or more parameters. Parameters are added to a constructor in the same way that they are added to a method:just declare them inside the parentheses after the constructor's name.

Example:

Here is a simple example that uses a constructor:

```
// A simple constructor.
class MyClass {
   int x;

   // Following is the constructor
   MyClass(int i ) {
      x = i;
   }
}
```

You would call constructor to initialize objects as follows:

```
public class ConsDemo {

   public static void main(String args[]) {
      MyClass t1 = new MyClass( 10 );
      MyClass t2 = new MyClass( 20 );
      System.out.println(t1.x + " " + t2.x);
   }
}
```

This would produce following result:

```
10 20
```

Variable Arguments(var-args):

JDK 1.5 enables you to pass a variable number of arguments of the same type to a method. The parameter in the method is declared as follows:

```
typeName... parameterName
```

In the method declaration, you specify the type followed by an ellipsis (...) Only one variable-length parameter may be specified in a method, and this parameter must be the last parameter. Any regular parameters must precede it.

Example:

```
public class VarargsDemo {

   public static void main(String args[]) {
      // Call method with variable args
            printMax(34, 3, 3, 2, 56.5);
      printMax(new double[]{1, 2, 3});
   }

   public static void printMax( double... numbers) {
```

```
   if (numbers.length == 0) {
      System.out.println("No argument passed");
      return;
   }

   double result = numbers[0];

   for (int i = 1; i <  numbers.length; i++)
      if (numbers[i] >  result)
      result = numbers[i];
      System.out.println("The max value is " + result);
   }
}
```

This would produce following result:

```
The max value is 56.5
The max value is 3.0
```

The finalize( ) Method:

It is possible to define a method that will be called just before an object's final destruction by the garbage collector. This method is called **finalize( )**, and it can be used to ensure that an object terminates cleanly.

For example, you might use finalize( ) to make sure that an open file owned by that object is closed.

To add a finalizer to a class, you simply define the finalize( ) method. The Java runtime calls that method whenever it is about to recycle an object of that class.

Inside the finalize( ) method you will specify those actions that must be performed before an object is destroyed.

The 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.

This means that you cannot know when.or even if.finalize( ) will be executed. For example, if your program ends before garbage collection occurs, finalize( ) will not execute.

Java - Streams, Files and I/O

The java.io package contains nearly every class you might ever need to perform input and output (I/O) in Java. All these streams represent an input source and an output destination. The stream in the java.io package supports many data such as primitives, Object, localized characters etc.

A stream can be defined as a sequence of data. The InputStream is used to read data from a source and the OutputStream is used for writing data to a destination.

Java does provide strong, flexible support for I/O as it relates to files and networks but this tutorial covers very basic functionlity related to streams and I/O. We would see most commonly used example one by one:

Reading Console Input:

Java input console is accomplished by reading from **System.in**. To obtain a character-based stream that is attached to the console, you wrap **System.in** in a**BufferedReader** object, to create a character stream. Here is most common syntax to obtain BufferedReader:

```
BufferedReader br = new BufferedReader(new
                      InputStreamReader(System.in));
```

Once BufferedReader is obtained, we can use read( ) method to reach a character or readLine( ) method to read a string from the console.

Reading Characters from Console:

To read a character from a BufferedReader, we would read( ) method whose sytax is as follows:

```
int read( ) throws IOException
```

Each time that read( ) is called, it reads a character from the input stream and returns it as an integer value. It returns .1 when the end of the stream is encountered. As you can see, it can throw an IOException.

The following program demonstrates read( ) by reading characters from the console until the user types a "q":

```
// Use a BufferedReader to read characters from the console.


import java.io.*;


public class BRRead {
   public static void main(String args[]) throws IOException
   {
      char c;
      // Create a BufferedReader using System.in
      BufferedReader br = new BufferedReader(new
                          InputStreamReader(System.in));
      System.out.println("Enter characters, 'q' to quit.");
      // read characters
      do {
         c = (char) br.read();
         System.out.println(c);
      } while(c != 'q');
   }
}
```

Here is a sample run:

```
Enter characters, 'q' to quit.
123abcq
1
2
3
a
b
c
q
```

Reading Strings from Console:

To read a string from the keyboard, use the version of readLine( ) that is a member of the BufferedReader class. Its general form is shown here:

```
String readLine( ) throws IOException
```

The following program demonstrates BufferedReader and the readLine( ) method. The program reads and displays lines of text until you enter the word "end":

```
// Read a string from console using a BufferedReader.
import java.io.*;
public class BRReadLines {
   public static void main(String args[]) throws IOException
   {
      // Create a BufferedReader using System.in
      BufferedReader br = new BufferedReader(new
                              InputStreamReader(System.in));
      String str;
      System.out.println("Enter lines of text.");
      System.out.println("Enter 'end' to quit.");
      do {
         str = br.readLine();
         System.out.println(str);
      } while(!str.equals("end"));
   }
}
```

Here is a sample run:

```
Enter lines of text.
Enter 'end' to quit.
This is line one
```

```
This is line one

This is line two

This is line two

end

end
```

Writing Console Output:

Console output is most easily accomplished with **print( )** and **println( )**, described earlier. These methods are defined by the class **PrintStream** which is the type of the object referenced by **System.out**. Even though System.out is a byte stream, using it for simple program output is still acceptable.

Because PrintStream is an output stream derived from OutputStream, it also implements the low-level method write( ). Thus, write( ) can be used to write to the console. The simplest form of write( ) defined by PrintStream is shown here:

```
void write(int byteval)
```

This method writes to the stream the byte specified by byteval. Although byteval is declared as an integer, only the low-order eight bits are written.

Example:

Here is a short example that uses write( ) to output the character "A" followed by a newline to the screen:

```
import java.io.*;


// Demonstrate System.out.write().
public class WriteDemo {
   public static void main(String args[]) {
      int b;
      b = 'A';
      System.out.write(b);
      System.out.write('\n');
   }
}
```

This would produce simply 'A' character on the output screen.

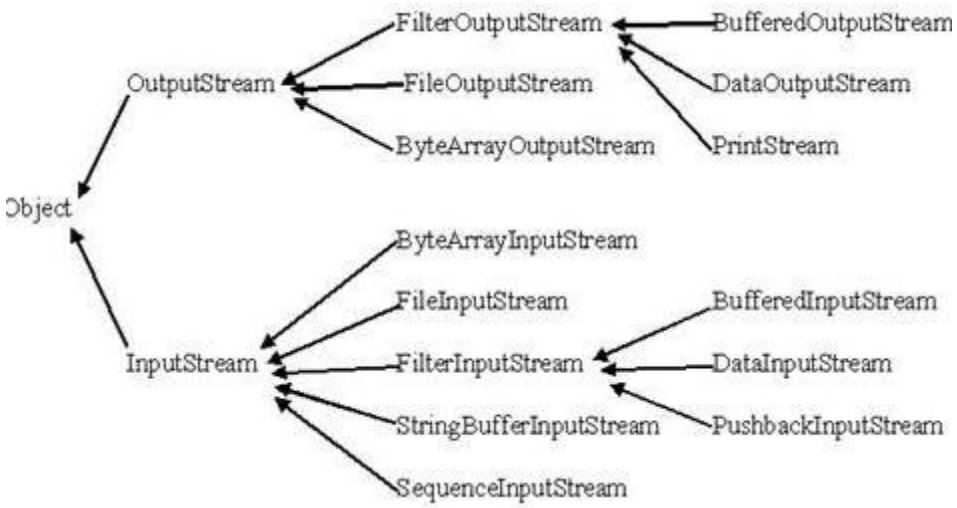
```
A
```

Note: You will not often use write( ) to perform console output because print( ) and println( ) are substantially easier to use.

Reading and Writing Files:

As described earlier, A stream can be defined as a sequence of data. The InputStream is used to read data from a source and the OutputStream is used for writing data to a destination.

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



The two important streams are FileInputStream and FileOutputStream which would be discussed in this tutorial:

FileInputStream:

This stream is used for reading data from the files. Objects can be created using the keyword new and there are several types of constructors available.

Following constructor takes a file name as a string to create an input stream object to read the file.:

```
InputStream f = new FileInputStream("C:/java/hello");
```

Following constructor takes a file object to create an input stream object to read the file. First we create a file object using File() method as follows:

```
File f = new File("C:/java/hello");
InputStream f = new FileInputStream(f);
```

Once you have *InputStream* object in hand then there is a list of helper methods which can be used to read to stream or to do other operations on the stream.

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.

There are other important input streams available, for more detail you can refer to the following links:

- ByteArrayInputStream
- DataInputStream

FileOutputStream:

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.

Here are two constructors which can be used to create a FileOutputStream object.

Following constructor takes a file name as a string to create an input stream object to write the file.:

```
OutputStream f = new FileOutputStream("C:/java/hello")
```

Following constructor takes a file object to create an output stream object to write the file. First we create a file object using File() method as follows:

```
File f = new File("C:/java/hello");
OutputStream f = new FileOutputStream(f);
```

Once you have *OutputStream* object in hand then there is a list of helper methods which can be used to write to stream or to do other operations on the stream.

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.

There are other important output streams available, for more detail you can refer to the following links:

- ByteArrayOutputStream
- DataOutputStream

Example:

Following is the example to demonstrate InputStream and OutputStream:

```
import java.io.*;

public class fileStreamTest{

   public static void main(String args[]){

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

      InputStream is = new FileInputStream("C:/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");
   }
   }
}
```

The above code would create file test.txt and would write given numbers in binary format. Same would be output on the stdout screen.

File Navigation and I/O:

There are several other classes that we would be going through to get to know the basics of File Navigation and I/O.

- File Class
- FileReader Class

- FileWriter Class

Directories in Java:

Creating Directories:

There are two useful **File** utility methods which can be used to create directories:

- The **mkdir( )** method creates a directory, returning true on success and false on failure. Failure indicates that the path specified in the File object already exists, or that the directory cannot be created because the entire path does not exist yet.

- The **mkdirs()** method creates both a directory and all the parents of the directory.

Following example creates "/tmp/user/java/bin" directory:

```
import java.io.File;


public class CreateDir {
   public static void main(String args[]) {
      String dirname = "/tmp/user/java/bin";
      File d = new File(dirname);
      // Create directory now.
      d.mkdirs();
  }
}
```

Compile and execute above code to create "/tmp/user/java/bin".

Note: Java automatically takes care of path separators on UNIX and Windows as per conventions. If you use a forward slash (/) on a Windows version of Java, the path will still resolve correctly.

Reading Directories:

A directory is a File that contains a list of other files and directories. When you create a File object and it is a directory, the isDirectory( ) method will return true.

You can call list( ) on that object to extract the list of other files and directories inside. The program shown here illustrates how to use list( ) to examine the contents of a directory:

```
import java.io.File;


public class DirList {
   public static void main(String args[]) {
      String dirname = "/java";
      File f1 = new File(dirname);
      if (f1.isDirectory()) {
         System.out.println( "Directory of " + dirname);
         String s[] = f1.list();
```

```
         for (int i=0; i < s.length; i++) {
            File f = new File(dirname + "/" + s[i]);
            if (f.isDirectory()) {
               System.out.println(s[i] + " is a directory");
            } else {
               System.out.println(s[i] + " is a file");
            }
         }
      } else {
         System.out.println(dirname + " is not a directory");
    }
  }
}
```

This would produce following result:

```
Directory of /mysql
bin is a directory
lib is a directory
demo is a directory
test.txt is a file
README is a file
index.html is a file
include is a directory
```

Java - Exceptions Handling

An exception is a problem that arises during the execution of a program. An exception can occur for many different reasons, including the following:

- A user has entered invalid data.
- A file that needs to be opened cannot be found.
- A network connection has been lost in the middle of communications, or the JVM has run out of memory.

Some of these exceptions are caused by user error, others by programmer error, and others by physical resources that have failed in some manner.

To understand how exception handling works in Java, you need to understand the three categories of exceptions:

- **Checked exceptions:** A checked exception is an exception that is typically a user error or a problem that cannot be foreseen by the programmer. For example, if a file is to be opened, but the file cannot be found, an exception occurs. These exceptions cannot simply be ignored at the time of compilation.
- **Runtime exceptions:** A runtime exception is an exception that occurs that probably could have been avoided by the programmer. As opposed to checked exceptions, runtime exceptions are ignored at the time of compliation.

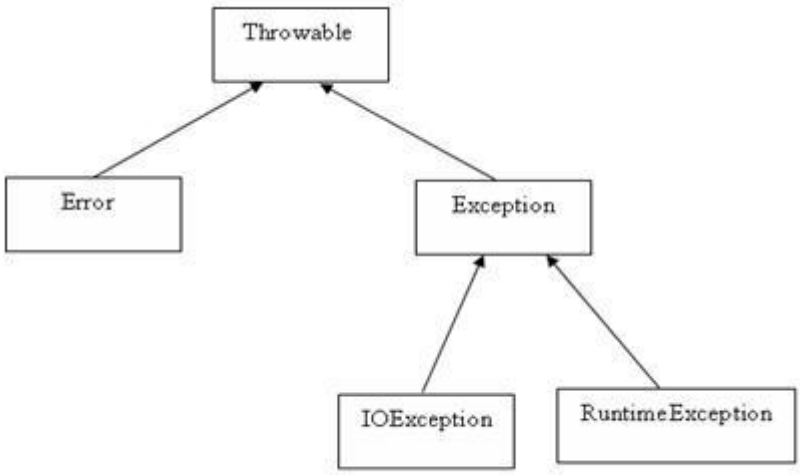
- **Errors:** These are not exceptions at all, but problems that arise beyond the control of the user or the programmer. Errors are typically ignored in your code because you can rarely do anything about an error. For example, if a stack overflow occurs, an error will arise. They are also ignored at the time of compilation.

Exception Hierarchy:

All exception classes are subtypes of the java.lang.Exception class. The exception class is a subclass of the Throwable class. Other than the exception class there is another subclass called Error which is derived from the Throwable class.

Errors are not normally trapped form the Java programs. These conditions normally happen in case of severe failures, which are not handled by the java programs. Errors are generated to indicate errors generated by the runtime environment. Example : JVM is out of Memory. Normally programs cannot recover from errors.

The Exception class has two main subclasses : IOException class and RuntimeException Class.



Here is a list of most common checked and unchecked Java's Built-in Exceptions.

Exceptions Methods:

Following is the list of important medthods available in the Throwable class.

SN	Methods with Description
1	**public String getMessage()** Returns a detailed message about the exception that has occurred. This message is initialized in the Throwable constructor.
2	**public Throwable getCause()** Returns the cause of the exception as represented by a Throwable object.
3	**public String toString()** Returns the name of the class concatenated with the result of getMessage()

4	**public void printStackTrace()** Prints the result of toString() along with the stack trace to System.err, the error output stream.
5	**public StackTraceElement [] getStackTrace()** Returns an array containing each element on the stack trace. The element at index 0 represents the top of the call stack, and the last element in the array represents the method at the bottom of the call stack.
6	**public Throwable fillInStackTrace()** Fills the stack trace of this Throwable object with the current stack trace, adding to any previous information in the stack trace.

Catching Exceptions:

A method catches an exception using a combination of the **try** and **catch** keywords. A try/catch block is placed around the code that might generate an exception. Code within a try/catch block is referred to as protected code, and the syntax for using try/catch looks like the following:

```
try
{
   //Protected code
}catch(ExceptionName e1)
{
   //Catch block
}
```

A catch statement involves declaring the type of exception you are trying to catch. If an exception occurs in protected code, the catch block (or blocks) that follows the try is checked. If the type of exception that occurred is listed in a catch block, the exception is passed to the catch block much as an argument is passed into a method parameter.

Example:

The following is an array is declared with 2 elements. Then the code tries to access the 3rd element of the array which throws an exception.

```
// File Name : ExcepTest.java
import java.io.*;
public class ExcepTest{


   public static void main(String args[]){
      try{
         int a[] = new int[2];
         System.out.println("Access element three :" + a[3]);
      }catch(ArrayIndexOutOfBoundsException e){
         System.out.println("Exception thrown  :" + e);
      }
      System.out.println("Out of the block");
   }
```

```
}
```

This would produce following result:

```
Exception thrown  :java.lang.ArrayIndexOutOfBoundsException: 3
Out of the block
```

Multiple catch Blocks:

A try block can be followed by multiple catch blocks. The syntax for multiple catch blocks looks like the following:

```
try
{
   //Protected code
}catch(ExceptionType1 e1)
{
   //Catch block
}catch(ExceptionType2 e2)
{
   //Catch block
}catch(ExceptionType3 e3)
{
   //Catch block
}
```

The previous statements demonstrate three catch blocks, but you can have any number of them after a single try. If an exception occurs in the protected code, the exception is thrown to the first catch block in the list. If the data type of the exception thrown matches ExceptionType1, it gets caught there. If not, the exception passes down to the second catch statement. This continues until the exception either is caught or falls through all catches, in which case the current method stops execution and the exception is thrown down to the previous method on the call stack.

Example:

Here is code segment showing how to use multiple try/catch statements.

```
try
{
   file = new FileInputStream(fileName);
   x = (byte) file.read();
}catch(IOException i)
{
   i.printStackTrace();
   return -1;
}catch(FileNotFoundException f) //Not valid!
{
   f.printStackTrace();
```

```
   return -1;
}
```

The throws/throw Keywords:

If a method does not handle a checked exception, the method must declare it using the **throws** keyword. The throws keyword appears at the end of a method's signature.

You can throw an exception, either a newly instantiated one or an exception that you just caught, by using the **throw** keyword. Try to understand the different in throws and throw keywords.

The following method declares that it throws a RemoteException:

```
import java.io.*;
public class className
{
   public void deposit(double amount) throws RemoteException
   {
      // Method implementation
      throw new RemoteException();
   }
   //Remainder of class definition
}
```

A method can declare that it throws more than one exception, in which case the exceptions are declared in a list separated by commas. For example, the following method declares that it throws a RemoteException and an InsufficientFundsException:

```
import java.io.*;
public class className
{
   public void withdraw(double amount) throws RemoteException,
                              InsufficientFundsException
   {
       // Method implementation
   }
   //Remainder of class definition
}
```

The finally Keyword

The finally keyword is used to create a block of code that follows a try block. A finally block of code always executes, whether or not an exception has occurred.

Using a finally block allows you to run any cleanup-type statements that you want to execute, no matter what happens in the protected code.

A finally block appears at the end of the catch blocks and has the following syntax:

```
try
{
   //Protected code
}catch(ExceptionType1 e1)
{
   //Catch block
}catch(ExceptionType2 e2)
{
   //Catch block
}catch(ExceptionType3 e3)
{
   //Catch block
}finally
{
   //The finally block always executes.
}
```

Example:

```
public class ExcepTest{


   public static void main(String args[]){
      int a[] = new int[2];
      try{
         System.out.println("Access element three :" + a[3]);
      }catch(ArrayIndexOutOfBoundsException e){
         System.out.println("Exception thrown  :" + e);
      }
      finally{
         a[0] = 6;
         System.out.println("First element value: " +a[0]);
         System.out.println("The finally statement is executed");
      }
   }
}
```

This would produce following result:

```
Exception thrown  :java.lang.ArrayIndexOutOfBoundsException: 3
First element value: 6
The finally statement is executed
```

Note the following:

- A catch clause cannot exist without a try statement.
- It is not compulsory to have finally clauses when ever a try/catch block is present.
- The try block cannot be present without either catch clause or finally clause.
- Any code cannot be present in between the try, catch, finally blocks.

Declaring you own Exception:

You can create your own exceptions in Java. Keep the following points in mind when writing your own exception classes:

- All exceptions must be a child of Throwable.
- If you want to write a checked exception that is automatically enforced by the Handle or Declare Rule, you need to extend the Exception class.
- If you want to write a runtime exception, you need to extend the RuntimeException class.

We can define our own Exception class as below:

```
class MyException extends Exception{
}
```

You just need to extend the Exception class to create your own Exception class. These are considered to be checked exceptions. The following InsufficientFundsException class is a user-defined exception that extends the Exception class, making it a checked exception. An exception class is like any other class, containing useful fields and methods.

Example:

```
// File Name InsufficientFundsException.java
import java.io.*;


public class InsufficientFundsException extends Exception
{
   private double amount;
   public InsufficientFundsException(double amount)
   {
      this.amount = amount;
   }
   public double getAmount()
   {
      return amount;
   }
}
```

To demonstrate using our user-defined exception, the following CheckingAccount class contains a withdraw() method that throws an InsufficientFundsException.

```
// File Name CheckingAccount.java
import java.io.*;

public class CheckingAccount
{
   private double balance;
   private int number;
   public CheckingAccount(int number)
   {
      this.number = number;
   }
   public void deposit(double amount)
   {
      balance += amount;
   }
   public void withdraw(double amount) throws
                              InsufficientFundsException
   {
      if(amount <= balance)
      {
         balance -= amount;
      }
      else
      {
         double needs = amount - balance;
         throw new InsufficientFundsException(needs);
      }
   }
   public double getBalance()
   {
      return balance;
   }
   public int getNumber()
   {
      return number;
   }
}
```

The following BankDemo program demonstrates invoking the deposit() and withdraw() methods of CheckingAccount.

```
// File Name BankDemo.java
```

```
public class BankDemo
{
   public static void main(String [] args)
   {
      CheckingAccount c = new CheckingAccount(101);
      System.out.println("Depositing $500...");
      c.deposit(500.00);
      try
      {
         System.out.println("\nWithdrawing $100...");
         c.withdraw(100.00);
         System.out.println("\nWithdrawing $600...");
         c.withdraw(600.00);
      }catch(InsufficientFundsException e)
      {
         System.out.println("Sorry, but you are short $"
                                 + e.getAmount());
         e.printStackTrace();
      }
    }
}
```

Compile all the above three files and run BankDemo, this would produce following result:

```
Depositing $500...


Withdrawing $100...


Withdrawing $600...
Sorry, but you are short $200.0
InsufficientFundsException
        at CheckingAccount.withdraw(CheckingAccount.java:25)
        at BankDemo.main(BankDemo.java:13)
```

Common Exceptions:

In java it is possible to define two catergories of Exceptions and Errors.

- **JVM Exceptions:** - These are exceptions/errors that are exclusively or logically thrown by the JVM. Examples : NullPointerException, ArrayIndexOutOfBoundsException, ClassCastException,

- **Programmatic exceptions** . These exceptions are thrown explicitly by the application or the API programmers Examples: IllegalArgumentException, IllegalStateException.

JAVA

Java Object Oriented

Java – Inheritance

Inheritance can be defined as the process where one object acquires the properties of another. With the use of inheritance the information is made manageable in a hierarchical order.

When we talk about inheritance the most commonly used keyword would be **extends** and **implements**. These words would determine whether one object IS-A type of another. By using these keywords we can make one object acquire the properties of another object.

IS-A Relationship:

IS-A is a way of saying : This object is a type of that object. Let us see how the **extends** keyword is used to achieve inheritance.

```
public class Animal{
}

public class Mammal extends Animal{
}

public class Reptile extends Animal{
}

public class Dog extends Mammal{
}
```

Now based on the above example, In Object Oriented terms following are true:

- Animal is the superclass of Mammal class.
- Animal is the superclass of Reptile class.
- Mammal and Reptile are sub classes of Animal class.
- Dog is the subclass of both Mammal and Animal classes.

Now if we consider the IS-A relationship we can say:

- Mammal IS-A Animal
- Reptile IS-A Animal
- Dog IS-A Mammal
- Hence : Dog IS-A Animal as well

With use of the extends keyword the subclasses will be able to inherit all the properties of the superclass except for the private properties of the superclass.

We can assure that Mammal is actually an Animal with the use of the instance operator.

Example:

```
public class Dog extends Mammal{


   public static void main(String args[]){


      Animal a = new Animal();
      Mammal m = new Mammal();
      Dog d = new Dog();


      System.out.println(m instanceof Animal);
      System.out.println(d instanceof Mammal);
      System.out.println(d instanceof Animal);
   }
}
```

This would produce following result:

```
true
true
true
```

Since we have a good understanding of the **extends** keyword let us look into how the **implements** keyword is used to get the IS-A relationship.

The **implements** keyword is used by classes by inherit from interfaces. Interfaces can never be extended by the classes.

Example:

```
public interface Animal {}


public class Mammal implements Animal{
}


public class Dog extends Mammal{
}
```

The instanceof Keyword:

Let us use the **instanceof** operator to check determine whether Mammal is actually an Animal, and dog is actually an Animal

```
interface Animal{}
```

```
class Mammal implements Animal{}


public class Dog extends Mammal{
   public static void main(String args[]){


      Mammal m = new Mammal();
      Dog d = new Dog();


      System.out.println(m instanceof Animal);
      System.out.println(d instanceof Mammal);
      System.out.println(d instanceof Animal);
   }
}
```

This would produce following result:

```
true
true
true
```

HAS-A relationship:

These relationships are mainly based on the usage. This determines whether a certain class **HAS-A** certain thing. This relationship helps to reduce duplication of code as well as bugs.

Lets us look into an example:

```
public class Vehicle{}
public class Speed{}
public class Van extends Vehicle{
         private Speed sp;
}
```

This shows that class Van HAS-A Speed. By having a separate class for Speed we do not have to put the entire code that belongs to speed inside the Van class., which makes it possible to reuse the Speed class in multiple applications.

In Object Oriented feature the users do not need to bother about which object is doing the real work. To achieve this, the Van class hides the implementation details from the users of the Van class. SO basically what happens is the users would ask the Van class to do a certain action and the Vann class will either do the work by itself or ask another class to perform the action.

A very important fact to remember is that Java only supports only single inheritance. This means that a class cannot extend more than one class. Therefore following is illegal:

```
public class extends Animal, Mammal{}
```

However a class can implement one or more interfaces. This has made Java get rid of the impossibility of multiple inheritance

Java – Overriding

In the previous chapter we talked about super classes and sub classes. If a class inherits a method from its super class, then there is a chance to override the method provided that it is not marked final.

The benefit of overriding is: ability to define a behavior that's specific to the sub class type. Which means a subclass can implement a parent class method based on its requirement.

In object oriented terms, overriding means to override the functionality of any existing method.

Example:

Let us look at an example.

```
class Animal{

   public void move(){
      System.out.println("Animals can move");
   }
}

class Dog extends Animal{

   public void move(){
      System.out.println("Dogs can walk and run");
   }
}

public class TestDog{

   public static void main(String args[]){
      Animal a = new Animal(); // Animal reference and object
      Animal b = new Dog(); // Animal reference but Dog object

      a.move();// runs the method in Animal class

      b.move();//Runs the method in Dog class
   }
}
```

This would produce following result:

```
Animals can move
Dogs can walk and run
```

In the above example you can see that the even though **b** is a type of Animal it runs the move method in the Dog class. The reason for this is : In compile time the check is made on the reference type. However in the runtime JVM figures out the object type and would run the method that belongs to that particular object.

Therefore in the above example, the program will compile properly since Animal class has the method move. Then at the runtime it runs the method specific for that object.

Consider the following example :

```
class Animal{


   public void move(){
      System.out.println("Animals can move");
   }
}


class Dog extends Animal{


   public void move(){
      System.out.println("Dogs can walk and run");
   }
   public void bark(){
      System.out.println("Dogs can bark");
   }
}


public class TestDog{


   public static void main(String args[]){
      Animal a = new Animal(); // Animal reference and object
      Animal b = new Dog(); // Animal reference but Dog object


      a.move();// runs the method in Animal class
      b.move();//Runs the method in Dog class
      b.bark();
   }
}
```

This would produce following result:

```
TestDog.java:30: cannot find symbol
symbol  : method bark()
location: class Animal
                b.bark();
```

```
                    ^
```

This program will throw a compile time error since b's reference type Animal doesn't have a method by the name of bark.

Rules for method overriding:

- The argument list should be exactly the same as that of the overridden method.
- The return type should be the same or a subtype of the return type declared in the original overridden method in the super class.
- The access level cannot be more restrictive than the overridden method's access level. For example: if the super class method is declared public then the overridding method in the sub class cannot be either private or protected.
- Instance methods can be overridden only if they are inherited by the subclass.
- A method declared final cannot be overridden.
- A method declared static cannot be overridden but can be re-declared.
- If a method cannot be inherited then it cannot be overridden.
- A subclass within the same package as the instance's superclass can override any superclass method that is not declared private or final.
- A subclass in a different package can only override the non-final methods declared public or protected.
- An overriding method can throw any uncheck exceptions, regardless of whether the overridden method throws exceptions or not. However the overriding method should not throw checked exceptions that are new or broader than the ones declared by the overridden method. The overriding method can throw narrower or fewer exceptions than the overridden method.
- Constructors cannot be overridden.

Using the super keyword:

When invoking a superclass version of an overridden method the **super** keyword is used.

```
class Animal{


   public void move(){
      System.out.println("Animals can move");
   }
}


class Dog extends Animal{


   public void move(){
      super.move(); // invokes the super class method
```

```
      System.out.println("Dogs can walk and run");
   }
}


public class TestDog{

   public static void main(String args[]){

      Animal b = new Dog(); // Animal reference but Dog object
      b.move(); //Runs the method in Dog class

   }
}
```

This would produce following result:

```
Animals can move
Dogs can walk and run
```

Java – Polymorphism

Polymorphism is the ability of an object to take on many forms. The most common use of polymorphism in OOP occurs when a parent class reference is used to refer to a child class object.

Any java object that can pass more than one IS-A test is considered to be polymorphic. In Java, all java objects are polymorphic since any object will pass the IS-A test for their own type and for the class Object.

It is important to know that the only possible way to access an object is through a reference variable. A reference variable can be of only one type. Once declared the type of a reference variable cannot be changed.

The reference variable can be reassigned to other objects provided that it is not declared final. The type of the reference variable would determine the methods that it can invoke on the object.

A reference variable can refer to any object of its declared type or any subtype of its declared type. A reference variable can be declared as a class or interface type.

Example:

Let us look at an example.

```
public interface Vegetarian{}
public class Animal{}
public class Deer extends Animal implements Vegetarian{}
```

Now the Deer class is considered to be polymorphic since this has multiple inheritance. Following are true for the above example:

- A Deer IS-A Animal
- A Deer IS-A Vegetarian

- A Deer IS-A Deer
- A Deer IS-A Object

When we apply the reference variable facts to a Deer object reference, the following declarations are legal:

```
Deer d = new Deer();
Animal a = d;
Vegetarian v = d;
Object o = d;
```

All the reference variables d,a,v,o refer to the same Deer object in the heap.

Virtual Methods:

In this section, I will show you how the behavior of overridden methods in Java allows you to take advantage of polymorphism when designing your classes.

We already have discussed method overriding, where a child class can override a method in its parent. An overridden method is essentially hidden in the parent class, and is not invoked unless the child class uses the super keyword within the overriding method.

```
/* File name : Employee.java */
public class Employee
{
   private String name;
   private String address;
   private int number;
   public Employee(String name, String address, int number)
   {
      System.out.println("Constructing an Employee");
      this.name = name;
      this.address = address;
      this.number = number;
   }
   public void mailCheck()
   {
      System.out.println("Mailing a check to " + this.name
       + " " + this.address);
   }
   public String toString()
   {
      return name + " " + address + " " + number;
   }
   public String getName()
   {
```

```
      return name;
   }
   public String getAddress()
   {
      return address;
   }
   public void setAddress(String newAddress)
   {
      address = newAddress;
   }
   public int getNumber()
   {
     return number;
   }
}
```

Now suppose we extend Employee class as follows:

```
/* File name : Salary.java */
public class Salary extends Employee
{
   private double salary; //Annual salary
   public Salary(String name, String address, int number, double
      salary)
   {
       super(name, address, number);
       setSalary(salary);
   }
   public void mailCheck()
   {
       System.out.println("Within mailCheck of Salary class ");
       System.out.println("Mailing check to " + getName()
       + " with salary " + salary);
   }
   public double getSalary()
   {
       return salary;
   }
   public void setSalary(double newSalary)
   {
       if(newSalary >= 0.0)
```

```
      {
         salary = newSalary;
      }
   }
   public double computePay()
   {
      System.out.println("Computing salary pay for " + getName());
      return salary/52;
   }
}
```

Now you study the following program carefully and try to determine its output:

```
/* File name : VirtualDemo.java */
public class VirtualDemo
{
   public static void main(String [] args)
   {
      Salary s = new Salary("Mohd Mohtashim", "Ambehta,  UP",
                                3, 3600.00);
      Employee e = new Salary("John Adams", "Boston, MA",
                                2, 2400.00);
      System.out.println("Call mailCheck using
                                  Salary reference --");
      s.mailCheck();
      System.out.println("\n Call mailCheck using
                                  Employee reference--");
      e.mailCheck();
    }
}
```

This would produce following result:

```
Constructing an Employee
Constructing an Employee
Call mailCheck using Salary reference --
Within mailCheck of Salary class
Mailing check to Mohd Mohtashim with salary 3600.0

Call mailCheck using Employee reference--
Within mailCheck of Salary class
Mailing check to John Adams with salary 2400.0
```

Here we instantiate two Salary objects . one using a Salary reference s, and the other using an Employee reference e.

While invoking *s.mailCheck()* the compiler sees mailCheck() in the Salary class at compile time, and the JVM invokes mailCheck() in the Salary class at run time.

Invoking mailCheck() on e is quite different because e is an Employee reference. When the compiler sees*e.mailCheck()*, the compiler sees the mailCheck() method in the Employee class.

Here, at compile time, the compiler used mailCheck() in Employee to validate this statement. At run time, however, the JVM invokes mailCheck() in the Salary class.

This behavior is referred to as virtual method invocation, and the methods are referred to as virtual methods. All methods in Java behave in this manner, whereby an overridden method is invoked at run time, no matter what data type the reference is that was used in the source code at compile time.

Java – Abstraction

Abstraction refers to the ability to make a class abstract in OOP. An abstract class is one that cannot be instantiated. All other functionality of the class still exists, and its fields, methods, and constructors are all accessed in the same manner. You just cannot create an instance of the abstract class.

If a class is abstract and cannot be instantiated, the class does not have much use unless it is subclassed. This is typically how abstract classes come about during the design phase. A parent class contains the common functionality of a collection of child classes, but the parent class itself is too abstract to be used on its own.

Abstract Class:

Use the **abstract** keyword to declare a class abstract. The keyword appears in the class declaration somewhere before the class keyword.

```
/* File name : Employee.java */
public abstract class Employee
{
   private String name;
   private String address;
   private int number;
   public Employee(String name, String address, int number)
   {
      System.out.println("Constructing an Employee");
      this.name = name;
      this.address = address;
      this.number = number;
   }
   public double computePay()
   {
     System.out.println("Inside Employee computePay");
     return 0.0;
   }
   public void mailCheck()
```

```
   {
      System.out.println("Mailing a check to " + this.name
       + " " + this.address);
   }
   public String toString()
   {
      return name + " " + address + " " + number;
   }
   public String getName()
   {
      return name;
   }
   public String getAddress()
   {
      return address;
   }
   public void setAddress(String newAddress)
   {
      address = newAddress;
   }
   public int getNumber()
   {
     return number;
   }
}
```

Notice that nothing is different in this Employee class. The class is now abstract, but it still has three fields, seven methods, and one constructor.

Now if you would try as follows:

```
/* File name : AbstractDemo.java */
public class AbstractDemo
{
   public static void main(String [] args)
   {
      /* Following is not allowed and would raise error */
      Employee e = new Employee("George W.", "Houston, TX", 43);

      System.out.println("\n Call mailCheck using
                                 Employee reference--");
      e.mailCheck();
```

```
    }
}
```

When you would compile above class then you would get following error:

```
Employee.java:46: Employee is abstract; cannot be instantiated
      Employee e = new Employee("George W.", "Houston, TX", 43);
                   ^
1 error
```

Extending Abstract Class:

We can extend Employee class in normal way as follows:

```
/* File name : Salary.java */
public class Salary extends Employee
{
   private double salary; //Annual salary
   public Salary(String name, String address, int number, double
      salary)
   {
       super(name, address, number);
       setSalary(salary);
   }
   public void mailCheck()
   {
       System.out.println("Within mailCheck of Salary class ");
       System.out.println("Mailing check to " + getName()
       + " with salary " + salary);
   }
   public double getSalary()
   {
       return salary;
   }
   public void setSalary(double newSalary)
   {
       if(newSalary >= 0.0)
       {
          salary = newSalary;
       }
   }
   public double computePay()
   {
```

```
      System.out.println("Computing salary pay for " + getName());
      return salary/52;
   }
}
```

Here we cannot instantiate a new Employee, but if we instantiate a new Salary object, the Salary object will inherit the three fields and seven methods from Employee.

```
/* File name : AbstractDemo.java */
public class AbstractDemo
{
   public static void main(String [] args)
   {
      Salary s = new Salary("Mohd Mohtashim", "Ambehta,  UP",
                                3, 3600.00);
      Salary e = new Salary("John Adams", "Boston, MA",
                                2, 2400.00);

      System.out.println("Call mailCheck using
                                  Salary reference --");
      s.mailCheck();
      System.out.println("\n Call mailCheck using
                                  Employee reference--");
      e.mailCheck();
    }
}
```

This would produce following result:

```
Constructing an Employee
Constructing an Employee
Call mailCheck using  Salary reference --
Within mailCheck of Salary class
Mailing check to Mohd Mohtashim with salary 3600.0

Call mailCheck using Employee reference--
Within mailCheck of Salary class
Mailing check to John Adams with salary 2400.
```

Abstract Methods:

If you want a class to contain a particular method but you want the actual implementation of that method to be determined by child classes, you can declare the method in the parent class as abstract.

The abstract keyword is also used to declare a method as abstract.An abstract methods consist of a method signature, but no method body.

Abstract method would have no definition, and its signature is followed by a semicolon, not curly braces as follows:

```
public abstract class Employee
{
   private String name;
   private String address;
   private int number;

   public abstract double computePay();

   //Remainder of class definition
}
```

Declaring a method as abstract has two results:

- The class must also be declared abstract. If a class contains an abstract method, the class must be abstract as well.
- Any child class must either override the abstract method or declare itself abstract.

A child class that inherits an abstract method must override it. If they do not, they must be abstract,and any of their children must override it.

Eventually, a descendant class has to implement the abstract method; otherwise, you would have a hierarchy of abstract classes that cannot be instantiated.

If Salary is extending Employee class then it is required to implement computePay() method as follows:

```
/* File name : Salary.java */
public class Salary extends Employee
{
   private double salary; // Annual salary

   public double computePay()
   {
      System.out.println("Computing salary pay for " + getName());
      return salary/52;
   }

   //Remainder of class definition
}
```

JAVA

Java – Encapsulation

Encapsulation is one of the four fundamental OOP concepts. The other three are inheritance, polymorphism, and abstraction.

Encapsulation is the technique of making the fields in a class private and providing access to the fields via public methods. If a field is declared private, it cannot be accessed by anyone outside the class, thereby hiding the fields within the class. For this reason, encapsulation is also referred to as data hiding.

Encapsulation can be described as a protective barrier that prevents the code and data being randomly accessed by other code defined outside the class. Access to the data and code is tightly controlled by an interface.

The main benefit of encapsulation is the ability to modify our implemented code without breaking the code of others who use our code. With this feature Encapsulation gives maintainability, flexibility and extensibility to our code.

Example:

Let us look at an example that depicts encapsulation:

```
/* File name : EncapTest.java */
public class EncapTest{

   private String name;
   private String idNum;
   private int age;

   public int getAge(){
      return age;
   }

   public String getName(){
      return name;
   }

   public String getIdNum(){
      return idNum;
   }

   public void setAge( int newAge){
      age = newAge;
   }

   public void setName(String newName){
      name = newName;
   }
```

```
   public void setIdNum( String newId){
      idNum = newId;
   }
}
```

The public methods are the access points to this class's fields from the outside java world. Normally these methods are referred as getters and setters. Therefore any class that wants to access the variables should access them through these getters and setters.

The variables of the EncapTest class can be access as below::

```
/* File name : RunEncap.java */
public class RunEncap{

   public static void main(String args[]){
      EncapTest encap = new EncapTest();
      encap.setName("James");
      encap.setAge(20);
      encap.setIdNum("12343ms");

      System.out.print("Name : " + encap.getName()+
                              " Age : "+ encap.getAge());
    }
}
```

This would produce following result:

```
Name : James Age : 20
```

Benefits of Encapsulation:

- The fields of a class can be made read-only or write-only.
- A class can have total control over what is stored in its fields.
- The users of a class do not know how the class stores its data. A class can change the data type of a field, and users of the class do not need to change any of their code.

Java – Interfaces

An interface is a collection of abstract methods. A class implements an interface, thereby inheriting the abstract methods of the interface.

An interface is not a class. Writing an interface is similar to writing a class, but they are two different concepts. A class describes the attributes and behaviors of an object. An interface contains behaviors that a class implements.

Unless the class that implements the interface is abstract, all the methods of the interface need to be defined in the class.

An interface is similar to a class in the following ways:

- An interface can contain any number of methods.
- An interface is written in a file with a **.java** extension, with the name of the interface matching the name of the file.
- The bytecode of an interface appears in a **.class** file.
- Interfaces appear in packages, and their corresponding bytecode file must be in a directory structure that matches the package name.

However, an interface is different from a class in several ways, including:

- You cannot instantiate an interface.
- An interface does not contain any constructors.
- All of the methods in an interface are abstract.
- An interface cannot contain instance fields. The only fields that can appear in an interface must be declared both static and final.
- An interface is not extended by a class; it is implemented by a class.
- An interface can extend multiple interfaces.

Declaring Interfaces:

The **interface** keyword is used to declare an interface. Here is a simple example to declare an interface:

Example:

Let us look at an example that depicts encapsulation:

```
/* File name : NameOfInterface.java */
import java.lang.*;
//Any number of import statements

public interface NameOfInterface
{
   //Any number of final, static fields
   //Any number of abstract method declarations\
}
```

Interfaces have the following properties:

- An interface is implicitly abstract. You do not need to use the **abstract** keyword when declaring an interface.
- Each method in an interface is also implicitly abstract, so the abstract keyword is not needed.
- Methods in an interface are implicitly public.

Example:

```
/* File name : Animal.java */
interface Animal {

   public void eat();
   public void travel();
}
```

Implementing Interfaces:

When a class implements an interface, you can think of the class as signing a contract, agreeing to perform the specific behaviors of the interface. If a class does not perform all the behaviors of the interface, the class must declare itself as abstract.

Aclass uses the **implements** keyword to implement an interface. The implements keyword appears in the class declaration following the extends portion of the declaration.

```
/* File name : MammalInt.java */
public class MammalInt implements Animal{

   public void eat(){
      System.out.println("Mammal eats");
   }

   public void travel(){
      System.out.println("Mammal travels");
   }

   public int noOfLegs(){
      return 0;
   }

   public static void main(String args[]){
      MammalInt m = new MammalInt();
      m.eat();
      m.travel();
   }
}
```

This would produce following result:

```
Mammal eats
Mammal travels
```

When overriding methods defined in interfaces there are several rules to be followed:

- Checked exceptions should not be declared on implementation methods other than the ones declared by the interface method or subclasses of those declared by the interface method.
- The signature of the interface method and the same return type or subtype should be maintained when overriding the methods.
- An implementation class itself can be abstract and if so interface methods need not be implemented.

When implementation interfaces there are several rules:

- A class can implement more than one interface at a time.
- A class can extend only one class, but implement many interface.
- An interface can extend another interface, similarly to the way that a class can extend another class.

Extending Interfaces:

An interface can extend another interface, similarly to the way that a class can extend another class. The **extends** keyword is used to extend an interface, and the child interface inherits the methods of the parent interface.

The following Sports interface is extended by Hockey and Football interfaces.

```
//Filename: Sports.java
public interface Sports
{
   public void setHomeTeam(String name);
   public void setVisitingTeam(String name);
}

//Filename: Football.java
public interface Football extends Sports
{
   public void homeTeamScored(int points);
   public void visitingTeamScored(int points);
   public void endOfQuarter(int quarter);
}

//Filename: Hockey.java
public interface Hockey extends Sports
{
   public void homeGoalScored();
   public void visitingGoalScored();
   public void endOfPeriod(int period);
   public void overtimePeriod(int ot);
}
```

The Hockey interface has four methods, but it inherits two from Sports; thus, a class that implements Hockey needs to implement all six methods. Similarly, a class that implements Football needs to define the three methods from Football and the two methods from Sports.

Extending Multiple Interfaces:

A Java class can only extend one parent class. Multiple inheritance is not allowed. Interfaces are not classes, however, and an interface can extend more than one parent interface.

The extends keyword is used once, and the parent interfaces are declared in a comma-separated list.

For example, if the Hockey interface extended both Sports and Event, it would be declared as:

```
public interface Hockey extends Sports, Event
```

Tagging Interfaces:

The most common use of extending interfaces occurs when the parent interface does not contain any methods. For example, the MouseListener interface in the java.awt.event package extended java.util.EventListener, which is defined as:

```
package java.util;

public interface EventListener

{}
```

An interface with no methods in it is referred to as a **tagging** interface. There are two basic design purposes of tagging interfaces:

Creates a common parent: As with the EventListener interface, which is extended by dozens of other interfaces in the Java API, you can use a tagging interface to create a common parent among a group of interfaces. For example, when an interface extends EventListener, the JVM knows that this particular interface is going to be used in an event delegation scenario.

Adds a data type to a class: This situation is where the term tagging comes from. A class that implements a tagging interface does not need to define any methods (since the interface does not have any), but the class becomes an interface type through polymorphism.

Java – Packages

Packages are used in Java in-order to prevent naming conflicts, to control access, to make searching/locating and usage of classes, interfaces, enumerations and annotations easier etc.

A Package can be defined as a grouping of related types(classes, interfaces, enumerations and annotations) providing access protection and name space management.

Some of the existing packages in Java are::

- **java.lang** - bundles the fundamental classes
- **java.io** - classes for input , output functions are bundled in this package

Programmers can define their own packages to bundle group of classes/interfaces etc. It is a good practice to group related classes implemented by you so that a programmers can easily determine that the classes, interfaces, enumerations, annotations are related.

Since the package creates a new namespace there won't be any name conflicts with names in other packages. Using packages, it is easier to provide access control and it is also easier to locate the related classed.

Creating a package:

When creating a package, you should choose a name for the package and put a **package** statement with that name at the top of every source file that contains the classes, interfaces, enumerations, and annotation types that you want to include in the package.

The **package** statement should be the first line in the source file. There can be only one package statement in each source file, and it applies to all types in the file.

If a package statement is not used then the class, interfaces, enumerations, and annotation types will be put into an unnamed package.

Example:

Let us look at an example that creates a package called **animals**. It is common practice to use lowercased names of packages to avoid any conflicts with the names of classes, interfaces.

Put an interface in the package *animals*:

```
/* File name : Animal.java */
package animals;


interface Animal {
   public void eat();
   public void travel();
}
```

Now put an implementation in the same package *animals*:

```
package animals;


/* File name : MammalInt.java */
public class MammalInt implements Animal{


   public void eat(){
      System.out.println("Mammal eats");
   }


   public void travel(){
      System.out.println("Mammal travels");
   }


   public int noOfLegs(){
      return 0;
   }
```

```
   public static void main(String args[]){
      MammalInt m = new MammalInt();
      m.eat();
      m.travel();
   }
}
```

Now you compile these two files and put them in a sub-directory called **animals** and try to run as follows:

```
$ mkdir animals
$ cp Animal.class  MammalInt.class animals
$ java animals/MammalInt
Mammal eats
Mammal travels
```

The import Keyword:

If a class wants to use another class in the same package, the package name does not need to be used. Classes in the same package find each other without any special syntax.

Example:

Here a class named Boss is added to the payroll package that already contains Employee. The Boss can then refer to the Employee class without using the payroll prefix, as demonstrated by the following Boss class.

```
package payroll;


public class Boss
{
   public void payEmployee(Employee e)
   {
      e.mailCheck();
   }
}
```

What happens if Boss is not in the payroll package? The Boss class must then use one of the following techniques for referring to a class in a different package.

- The fully qualified name of the class can be used. For example:

```
payroll.Employee
```

- The package can be imported using the import keyword and the wild card (*). For example:

```
import payroll.*;
```

- The class itself can be imported using the import keyword. For example:

```
import payroll.Employee;
```

Note: A class file can contain any number of import statements. The import statements must appear after the package statement and before the class declaration.

The Directory Structure of Packages:

Two major results occur when a class is placed in a package:

- The name of the package becomes a part of the name of the class, as we just discussed in the previous section.

- The name of the package must match the directory structure where the corresponding bytecode resides.

Here is simple way of managing your files in java:

Put the source code for a class, interface, enumeration, or annotation type in a text file whose name is the simple name of the type and whose extension is **.java**. For example:

```
// File Name :  Car.java


package vehicle;


public class Car {
   // Class implementation.
}
```

Now put the source file in a directory whose name reflects the name of the package to which the class belongs:

```
....\vehicle\Car.java
```

Now the qualified class name and pathname would be as below:

- Class name -> vehicle.Car

- Path name -> vehicle\Car.java (in windows)

In general a company uses its reversed Internet domain name for its package names. Example: A company's Internet domain name is apple.com, then all its package names would start with com.apple. Each component of the package name corresponds to a subdirectory.

Example: The company had a com.apple.computers package that contained a Dell.java source file, it would be contained in a series of subdirectories like this:

```
....\com\apple\computers\Dell.java
```

At the time of compilation, the compiler creates a different output file for each class, interface and enumeration defined in it. The base name of the output file is the name of the type, and its extension is **.class**

For example:

```
// File Name: Dell.java


package com.apple.computers;

public class Dell{


}

class Ups{


}
```

Now compile this file as follows using -d option:

```
$javac -d . Dell.java
```

This would put compiled files as follows:

```
.\com\apple\computers\Dell.class

.\com\apple\computers\Ups.class
```

You can import all the classes or interfaces defined in *\com\apple\computers* as follows:

```
import com.apple.computers.*;
```

Like the .java source files, the compiled .class files should be in a series of directories that reflect the package name. However, the path to the .class files does not have to be the same as the path to the .java source files. You can arrange your source and class directories separately, as:

```
<path-one>\sources\com\apple\computers\Dell.java


<path-two>\classes\com\apple\computers\Dell.class
```

By doing this, it is possible to give the classes directory to other programmers without revealing your sources. You also need to manage source and class files in this manner so that the compiler and the Java Virtual Machine (JVM) can find all the types your program uses.

The full path to the classes directory, <path-two>\classes, is called the class path, and is set with the CLASSPATH system variable. Both the compiler and the JVM construct the path to your .class files by adding the package name to the class path.

Say <path-two>\classes is the class path, and the package name is com.apple.computers, then the compiler and JVM will look for .class files in <path-two>\classes\com\apple\comptuers.

A class path may include several paths. Multiple paths should be separated by a semicolon (Windows) or colon (Unix). By default, the compiler and the JVM search the current directory and the JAR file containing the Java platform classes so that these directories are automatically in the class path.

Set CLASSPATH System Variable:

To display the current CLASSPATH variable, use the following commands in Windows and Unix (Bourne shell):

- In Windows -> C:\> set CLASSPATH

- In Unix -> % echo $CLASSPATH

To delete the current contents of the CLASSPATH variable, use :

- In Windows -> C:\> set CLASSPATH=
- In Unix -> % unset CLASSPATH; export CLASSPATH

To set the CLASSPATH variable:

- In Windows -> set CLASSPATH=C:\users\jack\java\classes
- In Unix -> % CLASSPATH=/home/jack/java/classes; export CLASSPATH

JAVA ADVANCED.

Java - Data Structures

The data structures provided by the Java utility package are very powerful and perform a wide range of functions. These data structures consist of the following interface and classes:

- Enumeration
- BitSet
- Vector
- Stack
- Dictionary
- Hashtable
- Properties

All these classes are now legacy and Java-2 has introduced a new framework called Collections Framework which is discussed in next tutorial:

The Enumeration:

The Enumeration interface isn't itself a data structure, but it is very important within the context of other data structures. The Enumeration interface defines a means to retrieve successive elements from a data structure.

For example, Enumeration defines a method called nextElement that is used to get the next element in a data structure that contains multiple elements.

To have more detail about this interface, check down The Enumeration.

The Enumeration interface defines the methods by which you can enumerate (obtain one at a time) the elements in a collection of objects.

This legacy interface has been superceded by Iterator. Although not deprecated, Enumeration is considered obsolete for new code. However, it is used by several methods defined by the legacy classes such as Vector and Properties, is used by several other API classes, and is currently in widespread use in application code.

The methods declared by Enumeration are summarized in the following table:

SN	Methods with Description
1	**boolean hasMoreElements( )** When implemented, it must return true while there are still more elements to extract, and false when all the elements have been enumerated.
2	**Object nextElement( )** This returns the next object in the enumeration as a generic Object reference.

Example:

Following is the example showing usage of Enumeration.

```
import java.util.Vector;
import java.util.Enumeration;


public class EnumerationTester {


   public static void main(String args[]) {
      Enumeration days;
      Vector dayNames = new Vector();
      dayNames.add("Sunday");
      dayNames.add("Monday");
      dayNames.add("Tuesday");
      dayNames.add("Wednesday");
      dayNames.add("Thursday");
      dayNames.add("Friday");
      dayNames.add("Saturday");
      days = dayNames.elements();
      while (days.hasMoreElements()){
         System.out.println(days.nextElement());
      }
   }
}
```

This would produce following result:

```
Sunday
Monday
Tuesday
Wednesday
Thursday
```

```
Friday

Saturday
```

The BitSet

The BitSet class implements a group of bits, or flags, that can be set and cleared individually.

This class is very useful in cases where you need to keep up with a set of boolean values; you just assign a bit to each value and set or clear it as appropriate.

To have more detail about this class, check The BitSet.

A BitSet class creates a special type of array that holds bit values. The BitSet array can increase in size as needed. This makes it similar to a vector of bits.

This is a legacy class but it has been completely re-engineered in Java 2, version 1.4

The BitSet defines two constructors. The first version creates a default object:

```
BitSet( )
```

The second version allows you to specify its initial size i.e. the number of bits that it can hold. All bits are initialized to zero.

```
BitSet(int size)
```

BitSet implements the Cloneable interface and defines the methods listed in table below:

SN	Methods with Description
1	**void and(BitSet bitSet)** ANDs the contents of the invoking BitSet object with those specified by bitSet. The result is placed into the invoking object.
2	**void andNot(BitSet bitSet)** For each 1 bit in bitSet, the corresponding bit in the invoking BitSet is cleared.
3	**int cardinality( )** Returns the number of set bits in the invoking object.
4	**void clear( )** Zeros all bits.
5	**void clear(int index)** Zeros the bit specified by index.
6	**void clear(int startIndex, int endIndex)** Zeros the bits from startIndex to endIndex.1.

7	**Object clone( )** Duplicates the invoking BitSet object.
8	**boolean equals(Object bitSet)** Returns true if the invoking bit set is equivalent to the one passed in bitSet. Otherwise, the method returns false.
9	**void flip(int index)** Reverses the bit specified by index. (
10	**void flip(int startIndex, int endIndex)** Reverses the bits from startIndex to endIndex.1.
11	**boolean get(int index)** Returns the current state of the bit at the specified index.
12	**BitSet get(int startIndex, int endIndex)** Returns a BitSet that consists of the bits from startIndex to endIndex.1. The invoking object is not changed.
13	**int hashCode( )** Returns the hash code for the invoking object.
14	**boolean intersects(BitSet bitSet)** Returns true if at least one pair of corresponding bits within the invoking object and bitSet are 1.
15	**boolean isEmpty( )** Returns true if all bits in the invoking object are zero.
16	**int length( )** Returns the number of bits required to hold the contents of the invoking BitSet. This value is determined by the location of the last 1 bit.
17	**int nextClearBit(int startIndex)** Returns the index of the next cleared bit, (that is, the next zero bit), starting from the index specified by startIndex
18	**int nextSetBit(int startIndex)** Returns the index of the next set bit (that is, the next 1 bit), starting from the index specified by startIndex. If no bit is set, .1 is returned.
19	**void or(BitSet bitSet)** ORs the contents of the invoking BitSet object with that specified by bitSet. The result is placed into the invoking object.
20	**void set(int index)** Sets the bit specified by index.
21	**void set(int index, boolean v)** Sets the bit specified by index to the value passed in v. true sets the bit, false clears the bit.

22	**void set(int startIndex, int endIndex)** Sets the bits from startIndex to endIndex.1.
23	**void set(int startIndex, int endIndex, boolean v)** Sets the bits from startIndex to endIndex.1, to the value passed in v. true sets the bits, false clears the bits.
24	**int size( )** Returns the number of bits in the invoking BitSet object.
25	**String toString( )** Returns the string equivalent of the invoking BitSet object.
26	**void xor(BitSet bitSet)** XORs the contents of the invoking BitSet object with that specified by bitSet. The result is placed into the invoking object

Example:

The following program illustrates several of the methods supported by this data structure:

```
import java.util.BitSet;


public class BitSetDemo {


  public static void main(String args[]) {
     BitSet bits1 = new BitSet(16);
     BitSet bits2 = new BitSet(16);


     // set some bits
     for(int i=0; i<16; i++) {
        if((i%2) == 0) bits1.set(i);
        if((i%5) != 0) bits2.set(i);
     }
     System.out.println("Initial pattern in bits1: ");
     System.out.println(bits1);
     System.out.println("\nInitial pattern in bits2: ");
     System.out.println(bits2);


     // AND bits
     bits2.and(bits1);
     System.out.println("\nbits2 AND bits1: ");
     System.out.println(bits2);


     // OR bits
```

```
     bits2.or(bits1);
     System.out.println("\nbits2 OR bits1: ");
     System.out.println(bits2);

     // XOR bits
     bits2.xor(bits1);
     System.out.println("\nbits2 XOR bits1: ");
     System.out.println(bits2);
  }
}
```

This would produce following result:

```
Initial pattern in bits1:
{0, 2, 4, 6, 8, 10, 12, 14}

Initial pattern in bits2:
{1, 2, 3, 4, 6, 7, 8, 9, 11, 12, 13, 14}

bits2 AND bits1:
{2, 4, 6, 8, 12, 14}

bits2 OR bits1:
{0, 2, 4, 6, 8, 10, 12, 14}

bits2 XOR bits1:
{}
```

The Vector

The Vector class is similar to a traditional Java array, except that it can grow as necessary to accommodate new elements.

Like an array, elements of a Vector object can be accessed via an index into the vector.

The nice thing about using the Vector class is that you don't have to worry about setting it to a specific size upon creation; it shrinks and grows automatically when necessary.

To have more detail about this class, check The Vector.

Vector implements a dynamic array. It is similar to ArrayList, but with two differences:

- Vector is synchronized.
- Vector contains many legacy methods that are not part of the collections framework.

Vector proves to be very useful if you don't know the size of the array in advance, or you just need one that can change sizes over the lifetime of a program.

The Vector class supports four constructors. The first form creates a default vector, which has an initial size of 10:

```
Vector( )
```

The second form creates a vector whose initial capacity is specified by size:

```
Vector(int size)
```

The third form creates a vector whose initial capacity is specified by size and whose increment is specified by incr. The increment specifies the number of elements to allocate each time that a vector is resized upward:

```
Vector(int size, int incr)
```

The fourth form creates a vector that contains the elements of collection c:

```
Vector(Collection c)
```

Apart from the methods inherited from its parent classes, Vector defines following methods:

SN	Methods with Description
1	**void add(int index, Object element)** Inserts the specified element at the specified position in this Vector.
2	**boolean add(Object o)** Appends the specified element to the end of this Vector.
3	**boolean addAll(Collection c)** Appends all of the elements in the specified Collection to the end of this Vector, in the order that they are returned by the specified Collection's Iterator.
4	**boolean addAll(int index, Collection c)** Inserts all of the elements in in the specified Collection into this Vector at the specified position.
5	**void addElement(Object obj)** Adds the specified component to the end of this vector, increasing its size by one.
6	**int capacity()** Returns the current capacity of this vector.
7	**void clear()** Removes all of the elements from this Vector.
8	**Object clone()** Returns a clone of this vector.
9	**boolean contains(Object elem)**

	Tests if the specified object is a component in this vector.
10	**boolean containsAll(Collection c)** Returns true if this Vector contains all of the elements in the specified Collection.
11	**void copyInto(Object[] anArray)** Copies the components of this vector into the specified array.
12	**Object elementAt(int index)** Returns the component at the specified index.
13	**Enumeration elements()** Returns an enumeration of the components of this vector.
14	**void ensureCapacity(int minCapacity)** Increases the capacity of this vector, if necessary, to ensure that it can hold at least the number of components specified by the minimum capacity argument.
15	**boolean equals(Object o)** Compares the specified Object with this Vector for equality.
16	**Object firstElement()** Returns the first component (the item at index 0) of this vector.
17	**Object get(int index)** Returns the element at the specified position in this Vector.
18	**int hashCode()** Returns the hash code value for this Vector.
19	**int indexOf(Object elem)** Searches for the first occurence of the given argument, testing for equality using the equals method.
20	**int indexOf(Object elem, int index)** Searches for the first occurence of the given argument, beginning the search at index, and testing for equality using the equals method.
21	**void insertElementAt(Object obj, int index)** Inserts the specified object as a component in this vector at the specified index.
22	**boolean isEmpty()** Tests if this vector has no components.
23	**Object lastElement()** Returns the last component of the vector.
24	**int lastIndexOf(Object elem)** Returns the index of the last occurrence of the specified object in this vector.

25	**int lastIndexOf(Object elem, int index)** Searches backwards for the specified object, starting from the specified index, and returns an index to it.
26	**Object remove(int index)** Removes the element at the specified position in this Vector.
27	**boolean remove(Object o)** Removes the first occurrence of the specified element in this Vector If the Vector does not contain the element, it is unchanged.
28	**boolean removeAll(Collection c)** Removes from this Vector all of its elements that are contained in the specified Collection.
29	**void removeAllElements()** Removes all components from this vector and sets its size to zero.
30	**boolean removeElement(Object obj)** Removes the first (lowest-indexed) occurrence of the argument from this vector.
31	**void removeElementAt(int index)** removeElementAt(int index)
32	**protected void removeRange(int fromIndex, int toIndex)** Removes from this List all of the elements whose index is between fromIndex, inclusive and toIndex, exclusive.
33	**boolean retainAll(Collection c)** Retains only the elements in this Vector that are contained in the specified Collection.
34	**Object set(int index, Object element)** Replaces the element at the specified position in this Vector with the specified element.
35	**void setElementAt(Object obj, int index)** Sets the component at the specified index of this vector to be the specified object.
36	**void setSize(int newSize)** Sets the size of this vector.
37	**int size()** Returns the number of components in this vector.
38	**List subList(int fromIndex, int toIndex)** Returns a view of the portion of this List between fromIndex, inclusive, and toIndex, exclusive.
39	**Object[] toArray()** Returns an array containing all of the elements in this Vector in the correct order.
40	**Object[] toArray(Object[] a)** Returns an array containing all of the elements in this Vector in the correct order; the runtime type of the returned array is that of the specified array.

41	**String toString()** Returns a string representation of this Vector, containing the String representation of each element.
42	**void trimToSize()** Trims the capacity of this vector to be the vector's current size.

Example:

The following program illustrates several of the methods supported by this collection:

```
import java.util.*;


public class VectorDemo {


   public static void main(String args[]) {
      // initial size is 3, increment is 2
      Vector v = new Vector(3, 2);
      System.out.println("Initial size: " + v.size());
      System.out.println("Initial capacity: " +
      v.capacity());
      v.addElement(new Integer(1));
      v.addElement(new Integer(2));
      v.addElement(new Integer(3));
      v.addElement(new Integer(4));
      System.out.println("Capacity after four additions: " +
          v.capacity());


      v.addElement(new Double(5.45));
      System.out.println("Current capacity: " +
      v.capacity());
      v.addElement(new Double(6.08));
      v.addElement(new Integer(7));
      System.out.println("Current capacity: " +
      v.capacity());
      v.addElement(new Float(9.4));
      v.addElement(new Integer(10));
      System.out.println("Current capacity: " +
      v.capacity());
      v.addElement(new Integer(11));
      v.addElement(new Integer(12));
      System.out.println("First element: " +
         (Integer)v.firstElement());
```

```
      System.out.println("Last element: " +
         (Integer)v.lastElement());
      if(v.contains(new Integer(3)))
         System.out.println("Vector contains 3.");
      // enumerate the elements in the vector.
      Enumeration vEnum = v.elements();
      System.out.println("\nElements in vector:");
      while(vEnum.hasMoreElements())
         System.out.print(vEnum.nextElement() + " ");
      System.out.println();
   }
}
```

This would produce following result:

```
Initial size: 0
Initial capacity: 3
Capacity after four additions: 5
Current capacity: 5
Current capacity: 7
Current capacity: 9
First element: 1
Last element: 12
Vector contains 3.

Elements in vector:
1 2 3 4 5.45 6.08 7 9.4 10 11 12
```

The Stack

The Stack class implements a last-in-first-out (LIFO) stack of elements.

You can think of a stack literally as a vertical stack of objects; when you add a new element, it gets stacked on top of the others.

When you pull an element off the stack, it comes off the top. In other words, the last element you added to the stack is the first one to come back off.

To have more detail about this class, check The Stack.

Stack is a subclass of Vector that implements a standard last-in, first-out stack.

Stack only defines the default constructor, which creates an empty stack. Stack includes all the methods defined by Vector, and adds several of its own.

```
Stack( )
```

Apart from the methods inherited from its parent class Vector, Stack defines following methods:

SN	Methods with Description
1	**boolean empty()** Tests if this stack is empty. Returns true if the stack is empty, and returns false if the stack contains elements.
2	**Object peek( )** Returns the element on the top of the stack, but does not remove it.
3	**Object pop( )** Returns the element on the top of the stack, removing it in the process.
4	**Object push(Object element)** Pushes element onto the stack. element is also returned.
5	**int search(Object element)** Searches for element in the stack. If found, its offset from the top of the stack is returned. Otherwise, .1 is returned.

Example:

The following program illustrates several of the methods supported by this collection:

```
import java.util.*;

public class StackDemo {

   static void showpush(Stack st, int a) {
      st.push(new Integer(a));
      System.out.println("push(" + a + ")");
      System.out.println("stack: " + st);
   }

   static void showpop(Stack st) {
      System.out.print("pop -> ");
      Integer a = (Integer) st.pop();
      System.out.println(a);
      System.out.println("stack: " + st);
   }

   public static void main(String args[]) {
      Stack st = new Stack();
```

```
      System.out.println("stack: " + st);
      showpush(st, 42);
      showpush(st, 66);
      showpush(st, 99);
      showpop(st);
      showpop(st);
      showpop(st);
      try {
         showpop(st);
      } catch (EmptyStackException e) {
         System.out.println("empty stack");
      }
   }
}
```

This would produce following result:

```
stack: [ ]
push(42)
stack: [42]
push(66)
stack: [42, 66]
push(99)
stack: [42, 66, 99]
pop -> 99
stack: [42, 66]
pop -> 66
stack: [42]
pop -> 42
stack: [ ]
pop -> empty stack
```

The Dictionary

The Dictionary class is an abstract class that defines a data structure for mapping keys to values.

This is useful in cases where you want to be able to access data via a particular key rather than an integer index.

Since the Dictionary class is abstract, it provides only the framework for a key-mapped data structure rather than a specific implementation.

To have more detail about this class, check The Dictionary.

Dictionary is an abstract class that represents a key/value storage repository and operates much like Map.

Given a key and value, you can store the value in a Dictionary object. Once the value is stored, you can retrieve it by using its key. Thus, like a map, a dictionary can be thought of as a list of key/value pairs.

The abstract methods defined by Dictionary are listed below:

SN	Methods with Description
1	**Enumeration elements( )** Returns an enumeration of the values contained in the dictionary.
2	**Object get(Object key)** Returns the object that contains the value associated with key. If key is not in the dictionary, a null object is returned.
3	**boolean isEmpty( )** Returns true if the dictionary is empty, and returns false if it contains at least one key.
4	**Enumeration keys( )** Returns an enumeration of the keys contained in the dictionary.
5	**Object put(Object key, Object value)** Inserts a key and its value into the dictionary. Returns null if key is not already in the dictionary; returns the previous value associated with key if key is already in the dictionary.
6	**Object remove(Object key)** Removes key and its value. Returns the value associated with key. If key is not in the dictionary, a null is returned.
7	**int size( )** Returns the number of entries in the dictionary.

The Dictionary class is obsolete. You should implement the Map interface to obtain key/value storage functionality.

The Hashtable

The Hashtable class provides a means of organizing data based on some user-defined key structure.

For example, in an address list hash table you could store and sort data based on a key such as ZIP code rather than on a person's name.

The specific meaning of keys in regard to hash tables is totally dependent on the usage of the hash table and the data it contains.

To have more detail about this class, check The Hashtable.

Hashtable was part of the original java.util and is a concrete implementation of a Dictionary.

However, Java 2 reengineered Hashtable so that it also implements the Map interface. Thus, Hashtable is now integrated into the collections framework. It is similar to HashMap, but is synchronized.

Like HashMap, Hashtable stores key/value pairs in a hash table. When using a Hashtable, you specify an object that is used as a key, and the value that you want linked to that key. The key is then hashed, and the resulting hash code is used as the index at which the value is stored within the table.

The Hashtable defines four constructors. The first version is the default constructor:

```
Hashtable( )
```

The second version creates a hash table that has an initial size specified by size:

```
Hashtable(int size)
```

The third version creates a hash table that has an initial size specified by size and a fill ratio specified by fillRatio.

This ratio must be between 0.0 and 1.0, and it determines how full the hash table can be before it is resized upward.

```
Hashtable(int size, float fillRatio)
```

The fourth version creates a hash table that is initialized with the elements in m.

The capacity of the hash table is set to twice the number of elements in m. The default load factor of 0.75 is used.

```
Hashtable(Map m)
```

Apart from the methods defined by Map interface, Hashtable defines following methods:

SN	Methods with Description
1	**void clear( )** Resets and empties the hash table.
2	**Object clone( )** Returns a duplicate of the invoking object.
3	**boolean contains(Object value)** Returns true if some value equal to value exists within the hash table. Returns false if the value isn't found.
4	**boolean containsKey(Object key)** Returns true if some key equal to key exists within the hash table. Returns false if the key isn't found.
5	**boolean containsValue(Object value)** Returns true if some value equal to value exists within the hash table. Returns false if the value isn't found.
6	**Enumeration elements( )** Returns an enumeration of the values contained in the hash table.
7	**Object get(Object key)** Returns the object that contains the value associated with key. If key is not in the hash table, a null object is returned.
8	**boolean isEmpty( )**

	Returns true if the hash table is empty; returns false if it contains at least one key.
9	**Enumeration keys( )** Returns an enumeration of the keys contained in the hash table.
10	**Object put(Object key, Object value)** Inserts a key and a value into the hash table. Returns null if key isn't already in the hash table; returns the previous value associated with key if key is already in the hash table.
11	**void rehash( )** Increases the size of the hash table and rehashes all of its keys.
12	**Object remove(Object key)** Removes key and its value. Returns the value associated with key. If key is not in the hash table, a null object is returned.
13	**int size( )** Returns the number of entries in the hash table.
14	**String toString( )** Returns the string equivalent of a hash table.

Example:

The following program illustrates several of the methods supported by this data structure:

```
import java.util.*;


public class HashTableDemo {


   public static void main(String args[]) {
      // Create a hash map
      Hashtable balance = new Hashtable();
      Enumeration names;
      String str;
      double bal;


      balance.put("Zara", new Double(3434.34));
      balance.put("Mahnaz", new Double(123.22));
      balance.put("Ayan", new Double(1378.00));
      balance.put("Daisy", new Double(99.22));
      balance.put("Qadir", new Double(-19.08));


      // Show all balances in hash table.
      names = balance.keys();
      while(names.hasMoreElements()) {
```

```
         str = (String) names.nextElement();
         System.out.println(str + ": " +
         balance.get(str));
      }
      System.out.println();
      // Deposit 1,000 into Zara's account
      bal = ((Double)balance.get("Zara")).doubleValue();
      balance.put("Zara", new Double(bal+1000));
      System.out.println("Zara's new balance: " +
      balance.get("Zara"));
   }
}
```

This would produce following result:

```
Qadir: -19.08
Zara: 3434.34
Mahnaz: 123.22
Daisy: 99.22
Ayan: 1378.0

Zara's new balance: 4434.34
```

The Properties

Properties is a subclass of Hashtable. It is used to maintain lists of values in which the key is a String and the value is also a String.

The Properties class is used by many other Java classes. For example, it is the type of object returned by System.getProperties( ) when obtaining environmental values.

To have more detail about this class, check The Properties.

Properties is a subclass of Hashtable. It is used to maintain lists of values in which the key is a String and the value is also a String.

The Properties class is used by many other Java classes. For example, it is the type of object returned by System.getProperties( ) when obtaining environmental values.

Properties defines the following instance variable. This variable holds a default property list associated with a Properties object.

```
Properties defaults;
```

The Properties defines two constructors. The first version creates a Properties object that has no default values:

```
Properties( )
```

The second creates an object that uses propDefault for its default values. In both cases, the property list is empty:

```
Properties(Properties propDefault)
```

Apart from the methods defined by Hashtable, Properties defines following methods:

SN	Methods with Description
1	**String getProperty(String key)** Returns the value associated with key. A null object is returned if key is neither in the list nor in the default property list.
2	**String getProperty(String key, String defaultProperty)** Returns the value associated with key. defaultProperty is returned if key is neither in the list nor in the default property list.
3	**void list(PrintStream streamOut)** Sends the property list to the output stream linked to streamOut.
4	**void list(PrintWriter streamOut)** Sends the property list to the output stream linked to streamOut.
5	**void load(InputStream streamIn) throws IOException** Inputs a property list from the input stream linked to streamIn.
6	**Enumeration propertyNames( )** Returns an enumeration of the keys. This includes those keys found in the default property list, too.
7	**Object setProperty(String key, String value)** Associates value with key. Returns the previous value associated with key, or returns null if no such association exists.
8	**void store(OutputStream streamOut, String description)** After writing the string specified by description, the property list is written to the output stream linked to streamOut.

Example:

The following program illustrates several of the methods supported by this data structure:

```
import java.util.*;


public class PropDemo {


   public static void main(String args[]) {
      Properties capitals = new Properties();
      Set states;
```

```
      String str;

      capitals.put("Illinois", "Springfield");
      capitals.put("Missouri", "Jefferson City");
      capitals.put("Washington", "Olympia");
      capitals.put("California", "Sacramento");
      capitals.put("Indiana", "Indianapolis");

      // Show all states and capitals in hashtable.
      states = capitals.keySet(); // get set-view of keys
      Iterator itr = states.iterator();
      while(itr.hasNext()) {
         str = (String) itr.next();
         System.out.println("The capital of " +
            str + " is " + capitals.getProperty(str) + ".");
      }
      System.out.println();

      // look for state not in list -- specify default
      str = capitals.getProperty("Florida", "Not Found");
      System.out.println("The capital of Florida is "
          + str + ".");
   }
}
```

This would produce following result:

```
The capital of Missouri is Jefferson City.
The capital of Illinois is Springfield.
The capital of Indiana is Indianapolis.
The capital of California is Sacramento.
The capital of Washington is Olympia.

The capital of Florida is Not Found.
```

JAVA

Prior to Java 2, Java provided ad hoc classes such as **Dictionary, Vector, Stack**, and **Properties** to store and manipulate groups of objects. Although these classes were quite useful, they lacked a central, unifying theme. Thus, the way that you used Vector was different from the way that you used Properties.

The collections framework was designed to meet several goals.

- The framework had to be high-performance. The implementations for the fundamental collections (dynamic arrays, linked lists, trees, and hash tables) are highly efficient.
- The framework had to allow different types of collections to work in a similar manner and with a high degree of interoperability.
- Extending and/or adapting a collection had to be easy.

Toward this end, the entire collections framework is designed around a set of standard interfaces. Several standard implementations such as **LinkedList, HashSet,** and **TreeSet**, of these interfaces are provided that you may use as-is and you may also implement your own collection, if you choose.

A collections framework is a unified architecture for representing and manipulating collections. All collections frameworks contain the following:

- **Interfaces:** These are abstract data types that represent collections. Interfaces allow collections to be manipulated independently of the details of their representation. In object-oriented languages, interfaces generally form a hierarchy.
- **Implementations i.e. Classes:** These are the concrete implementations of the collection interfaces. In essence, they are reusable data structures.
- **Algorithms:** These are the methods that perform useful computations, such as searching and sorting, on objects that implement collection interfaces. The algorithms are said to be polymorphic: that is, the same method can be used on many different implementations of the appropriate collection interface.

In addition to collections, the framework defines several map interfaces and classes. Maps store key/value pairs. Although maps are not *collections* in the proper use of the term, but they are fully integrated with collections.

The Collection Interfaces:

The collections framework defines several interfaces. This section provides an overview of each interface:

SN	Interfaces with Description
1	The Collection Interface This enables you to work with groups of objects; it is at the top of the collections hierarchy.
2	The List Interface This extends **Collection** and an instance of List stores an ordered collection of elements.
3	The Set This extends Collection to handle sets, which must contain unique elements
4	The SortedSet This extends Set to handle sorted sets

5	The Map This maps unique keys to values.
6	The Map.Entry This describes an element (a key/value pair) in a map. This is an inner class of Map.
7	The SortedMap This extends Map so that the keys are maintained in ascending order.
8	The Enumeration This is legacy interface and defines the methods by which you can enumerate (obtain one at a time) the elements in a collection of objects. This legacy interface has been superceded by Iterator.

The Collection Classes:

Java provides a set of standard collection classes that implement Collection interfaces. Some of the classes provide full implementations that can be used as-is and others are abstract class, providing skeletal implementations that are used as starting points for creating concrete collections.

The standard collection classes are summarized in the following table:

SN	Classes with Description
1	**AbstractCollection** Implements most of the Collection interface.
2	**AbstractList** Extends AbstractCollection and implements most of the List interface.
3	**AbstractSequentialList** Extends AbstractList for use by a collection that uses sequential rather than random access of its elements.
4	LinkedList Implements a linked list by extending AbstractSequentialList.
5	ArrayList Implements a dynamic array by extending AbstractList.
6	**AbstractSet** Extends AbstractCollection and implements most of the Set interface.
7	HashSet Extends AbstractSet for use with a hash table.
8	LinkedHashSet Extends HashSet to allow insertion-order iterations.
9	TreeSet Implements a set stored in a tree. Extends AbstractSet.

10	**AbstractMap** Implements most of the Map interface.
11	HashMap Extends AbstractMap to use a hash table.
12	TreeMap Extends AbstractMap to use a tree.
13	WeakHashMap Extends AbstractMap to use a hash table with weak keys.
14	LinkedHashMap Extends HashMap to allow insertion-order iterations.
15	IdentityHashMap Extends AbstractMap and uses reference equality when comparing documents.

The *AbstractCollection, AbstractSet, AbstractList, AbstractSequentialList* and *AbstractMap* classes provide skeletal implementations of the core collection interfaces, to minimize the effort required to implement them.

The following legacy classes defined by java.util has been discussed in previous tutorial:

SN	Classes with Description
1	Vector This implements a dynamic array. It is similar to ArrayList, but with some differences.
2	Stack Stack is a subclass of Vector that implements a standard last-in, first-out stack.
3	Dictionary Dictionary is an abstract class that represents a key/value storage repository and operates much like Map.
4	Hashtable Hashtable was part of the original java.util and is a concrete implementation of a Dictionary.
5	Properties Properties is a subclass of Hashtable. It is used to maintain lists of values in which the key is a String and the value is also a String.
6	BitSet A BitSet class creates a special type of array that holds bit values. This array can increase in size as needed.

The Collection Algorithms:

The collections framework defines several algorithms that can be applied to collections and maps. These algorithms are defined as static methods within the Collections class.

Several of the methods can throw a **ClassCastException**, which occurs when an attempt is made to compare incompatible types, or an **UnsupportedOperationException**, which occurs when an attempt is made to modify an unmodifiable collection.

Collections defines three static variables: EMPTY_SET, EMPTY_LIST, and EMPTY_MAP. All are immutable.

SN	Algorithms with Description
1	The Collection Algorithms Here is a list of all the algorithm implementation.

How to use an Iterator ?

Often, you will want to cycle through the elements in a collection. For example, you might want to display each element.

The easiest way to do this is to employ an iterator, which is an object that implements either the Iterator or the ListIterator interface.

Iterator enables you to cycle through a collection, obtaining or removing elements. ListIterator extends Iterator to allow bidirectional traversal of a list, and the modification of elements.

SN	Iterator Methods with Description
1	Using Java Iterator Here is a list of all the methods with examples provided by Iterator and ListIterator interfaces.

How to use an Comparator ?

Both TreeSet and TreeMap store elements in sorted order. However, it is the comparator that defines precisely what *sorted order* means.

This interface lets us sort a given collection any number of different ways. Also this interface can be used to sort any instances of any class.(even classes we cannot modify).

SN	Iterator Methods with Description
1	Using Java Comparator Here is a list of all the methods with examples provided by Comparator Interface.

Summary:

The Java collections framework gives the programmer access to prepackaged data structures as well as to algorithms for manipulating them.

A collection is an object that can hold references to other objects. The collection interfaces declare the operations that can be performed on each type of collection.

The classes and interfaces of the collections framework are in package java.util,

It would be nice if we could write a single sort method that could sort the elements in an Integer array, a String array or an array of any type that supports ordering.

Java **Generic** methods and generic classes enable programmers to specify, with a single method declaration, a set of related methods or, with a single class declaration, a set of related types, respectively.

Generics also provide compile-time type safety that allows programmers to catch invalid types at compile time.

Using Java Generic concept we might write a generic method for sorting an array of objects, then invoke the generic method with Integer arrays, Double arrays, String arrays and so on, to sort the array elements.

Generic Methods:

You can write a single generic method declaration that can be called with arguments of different types. Based on the types of the arguments passed to the generic method, the compiler handles each method call appropriately. Following are the rules to define Generic Methods:

- All generic method declarations have a type parameter section delimited by angle brackets (< and >) that precedes the method's return type (< E > in the next example).
- Each type parameter section contains one or more type parameters separated by commas. A type parameter, also known as a type variable, is an identifier that specifies a generic type name.
- The type parameters can be used to declare the return type and act as placeholders for the types of the arguments passed to the generic method, which are known as actual type arguments.
- A generic method's body is declared like that of any other method. Note that type parameters can represent only reference types not primitive types (like int, double and char).

Example:

Following example illustrate how we can print array of different type using a single Generic method:

```
public class GenericMethodTest
{
   // generic method printArray
   public static < E > void printArray( E[] inputArray )
   {
      // Display array elements
         for ( E element : inputArray ){
            System.out.printf( "%s ", element );
         }
         System.out.println();
    }

    public static void main( String args[] )
    {
        // Create arrays of Integer, Double and Character
        Integer[] intArray = { 1, 2, 3, 4, 5 };
        Double[] doubleArray = { 1.1, 2.2, 3.3, 4.4 };
```

```
        Character[] charArray = { 'H', 'E', 'L', 'L', 'O' };

        System.out.println( "Array integerArray contains:" );
        printArray( intArray  ); // pass an Integer array

        System.out.println( "\nArray doubleArray contains:" );
        printArray( doubleArray ); // pass a Double array

        System.out.println( "\nArray characterArray contains:" );
        printArray( charArray ); // pass a Character array
    }
}
```

This would produce following result:

```
Array integerArray contains:
1 2 3 4 5 6

Array doubleArray contains:
1.1 2.2 3.3 4.4

Array characterArray contains:
H E L L O
```

Bounded Type Parameters:

There may be times when you'll want to restrict the kinds of types that are allowed to be passed to a type parameter. For example, a method that operates on numbers might only want to accept instances of Number or its subclasses. This is what bounded type parameters are for.

To declare a bounded type parameter, list the type parameter's name, followed by the extends keyword, followed by its upper bound.

Example:

Following example illustrate how extends is used in a general sense to mean either "extends" (as in classes) or "implements" (as in interfaces). This example is Generic method to return the largest of three Comparable objects:

```
public class MaximumTest
{
   // determines the largest of three Comparable objects
   public static <T extends Comparable<T>> T maximum(T x, T y, T z)
   {
      T max = x; // assume x is initially the largest
      if ( y.compareTo( max ) > 0 ){
         max = y; // y is the largest so far
```

```
      }
      if ( z.compareTo( max ) > 0 ){
         max = z; // z is the largest now
      }
      return max; // returns the largest object
   }
   public static void main( String args[] )
   {
      System.out.printf( "Max of %d, %d and %d is %d\n\n",
                  3, 4, 5, maximum( 3, 4, 5 ) );

      System.out.printf( "Maxm of %.1f,%.1f and %.1f is %.1f\n\n",
                  6.6, 8.8, 7.7, maximum( 6.6, 8.8, 7.7 ) );

      System.out.printf( "Max of %s, %s and %s is %s\n","pear",
         "apple", "orange", maximum( "pear", "apple", "orange" ) );
   }
}
```

This would produce following result:

```
Maximum of 3, 4 and 5 is 5

Maximum of 6.6, 8.8 and 7.7 is 8.8

Maximum of pear, apple and orange is pear
```

Generic Classes:

A generic class declaration looks like a non-generic class declaration, except that the class name is followed by a type parameter section.

As with generic methods, the type parameter section of a generic class can have one or more type parameters separated by commas. These classes are known as parameterized classes or parameterized types because they accept one or more parameters.

Example:

Following example illustrate how we can define a generic class:

```
public class Box<T> {

  private T t;

  public void add(T t) {
```

```
    this.t = t;
  }

  public T get() {
    return t;
  }

  public static void main(String[] args) {
     Box<Integer> integerBox = new Box<Integer>();
     Box<String> stringBox = new Box<String>();

     integerBox.add(new Integer(10));
     stringBox.add(new String("Hello World"));

     System.out.printf("Integer Value :%d\n\n", integerBox.get());
     System.out.printf("String Value :%s\n", stringBox.get());
  }
}
```

This would produce following result:

```
Integer Value :10


String Value :Hello World
```

Java-Serialization.

Java provides a mechanism, called object serialization where an object can be represented as a sequence of bytes that includes the object's data as well as information about the object's type and the types of data stored in the object.

After a serialized object has been written into a file, it can be read from the file and deserialized that is, the type information and bytes that represent the object and its data can be used to recreate the object in memory.

Most impressive is that the entire process is JVM independent, meaning an object can be serialized on one platform and deserialized on an entirely different platform.

Classes **ObjectInputStream** and **ObjectOutputStream** are high-level streams that contain the methods for serializing and deserializing an object.

The ObjectOutputStream class contains many write methods for writing various data types, but one method in particular stands out:

```
public final void writeObject(Object x) throws IOException
```

The above method serializes an Object and sends it to the output stream. Similarly, the ObjectInputStream class contains the following method for deserializing an object:

```
public final Object readObject() throws IOException,
                                 ClassNotFoundException
```

This method retrieves the next Object out of the stream and deserializes it. The return value is Object, so you will need to cast it to its appropriate data type.

To demonstrate how serialization works in Java, I am going to use the Employee class that we discussed early on in the book. Suppose that we have the following Employee class, which implements the Serializable interface:

```
public class Employee implements java.io.Serializable
{
   public String name;
   public String address;
   public transient int SSN;
   public int number;
   public void mailCheck()
   {
      System.out.println("Mailing a check to " + name
                         + " " + address);
   }
}
```

Notice that for a class to be serialized successfully, two conditions must be met:

- The class must implement the java.io.Serializable interface.
- All of the fields in the class must be serializable. If a field is not serializable, it must be marked transient.

If you are curious to know if a Java Satandard Class is serializable or not, check the documentation for the class. The test is simple: If the class implements java.io.Serializable, then it is serializable; otherwise, it's not.

Serializing an Object:

The ObjectOutputStream class is used to serialize an Object. The following SerializeDemo program instantiates an Employee object and serializes it to a file.

When the program is done executing, a file named employee.ser is created. The program does not generate any output, but study the code and try to determine what the program is doing.

Note: When serializing an object to a file, the standard convention in Java is to give the file a **.ser** extension.

```
import java.io.*;


public class SerializeDemo
{
   public static void main(String [] args)
   {
      Employee e = new Employee();
```

```
      e.name = "Reyan Ali";
      e.address = "Phokka Kuan, Ambehta Peer";
      e.SSN = 11122333;
      e.number = 101;
      try
      {
         FileOutputStream fileOut =
         new FileOutputStream("employee.ser");
         ObjectOutputStream out =
                           new ObjectOutputStream(fileOut);
         out.writeObject(e);
         out.close();
          fileOut.close();
      }catch(IOException i)
      {
          i.printStackTrace();
      }
   }
}
```

Deserializing an Object:

The following DeserializeDemo program deserializes the Employee object created in the SerializeDemo program. Study the program and try to determine its output:

```
import java.io.*;
public class DeserializeDemo
{
   public static void main(String [] args)
   {
      Employee e = null;
      try
      {
         FileInputStream fileIn =
                         new FileInputStream("employee.ser");
         ObjectInputStream in = new ObjectInputStream(fileIn);
         e = (Employee) in.readObject();
         in.close();
         fileIn.close();
      }catch(IOException i)
      {
         i.printStackTrace();
```

```
         return;
      }catch(ClassNotFoundException c)
      {
         System.out.println("Employee class not found");
         c.printStackTrace();
         return;
      }
      System.out.println("Deserialized Employee...");
      System.out.println("Name: " + e.name);
      System.out.println("Address: " + e.address);
      System.out.println("SSN: " + e.SSN);
      System.out.println("Number: " + e.number);
    }
}
```

This would produce following result:

```
Deserialized Employee...
Name: Reyan Ali
Address:Phokka Kuan, Ambehta Peer
SSN: 0
Number:101
```

Here are following important points to be noted:

- The try/catch block tries to catch a ClassNotFoundException, which is declared by the readObject() method. For a JVM to be able to deserialize an object, it must be able to find the bytecode for the class. If the JVM can't find a class during the deserialization of an object, it throws a ClassNotFoundException.
- Notice that the return value of readObject() is cast to an Employee reference.
- The value of the SSN field was 11122333 when the object was serialized, but because the field is transient, this value was not sent to the output stream. The SSN field of the deserialized Employee object is 0.

Java - Networking (Socket Programming)

The term *network programming* refers to writing programs that execute across multiple devices (computers), in which the devices are all connected to each other using a network.

The java.net package of the J2SE APIs contains a collection of classes and interfaces that provide the low-level communication details, allowing you to write programs that focus on solving the problem at hand.

The java.net package provides support for the two common network protocols:

- **TCP:** TCP stands for Transmission Control Protocol, which allows for reliable communication between two applications. TCP is typically used over the Internet Protocol, which is referred to as TCP/IP.

- **UDP:** UDP stands for User Datagram Protocol, a connection-less protocol that allows for packets of data to be transmitted between applications.

This tutorial gives good understanding on the following two subjects:

- **Socket Programming**: This is most widely used concept in Networking and it has been explained in very detail.

- **URL Processing**: This would be covered separately. Click here to learn about URL Processing in Java language.

Socket Programming:

Sockets provide the communication mechanism between two computers using TCP. A client program creates a socket on its end of the communication and attempts to connect that socket to a server.

When the connection is made, the server creates a socket object on its end of the communication. The client and server can now communicate by writing to and reading from the socket.

The java.net.Socket class represents a socket, and the java.net.ServerSocket class provides a mechanism for the server program to listen for clients and establish connections with them.

The following steps occur when establishing a TCP connection between two computers using sockets:

- The server instantiates a ServerSocket object, denoting which port number communication is to occur on.

- The server invokes the accept() method of the ServerSocket class. This method waits until a client connects to the server on the given port.

- After the server is waiting, a client instantiates a Socket object, specifying the server name and port number to connect to.

- The constructor of the Socket class attempts to connect the client to the specified server and port number. If communication is established, the client now has a Socket object capable of communicating with the server.

- On the server side, the accept() method returns a reference to a new socket on the server that is connected to the client's socket.

After the connections are established, communication can occur using I/O streams. Each socket has both an OutputStream and an InputStream. The client's OutputStream is connected to the server's InputStream, and the client's InputStream is connected to the server's OutputStream.

TCP is a twoway communication protocol, so data can be sent across both streams at the same time. There are following usefull classes providing complete set of methods to implement sockets.

ServerSocket Class Methods:

The **java.net.ServerSocket** class is used by server applications to obtain a port and listen for client requests

The ServerSocket class has four constructors:

SN	Methods with Description
1	**public ServerSocket(int port) throws IOException** Attempts to create a server socket bound to the specified port. An exception occurs if the port is already bound by another application.
2	**public ServerSocket(int port, int backlog) throws IOException** Similar to the previous constructor, the backlog parameter specifies how many incoming clients to store in a wait queue.
3	**public ServerSocket(int port, int backlog, InetAddress address) throws IOException** Similar to the previous constructor, the InetAddress parameter specifies the local IP address to bind to. The InetAddress is used for servers that may have multiple IP addresses, allowing the server to specify which of its IP addresses to accept client requests on
4	**public ServerSocket() throws IOException** Creates an unbound server socket. When using this constructor, use the bind() method when you are ready to bind the server socket

If the ServerSocket constructor does not throw an exception, it means that your application has successfully bound to the specified port and is ready for client requests.

Here are some of the common methods of the ServerSocket class:

SN	Methods with Description
1	**public int getLocalPort()** Returns the port that the server socket is listening on. This method is useful if you passed in 0 as the port number in a constructor and let the server find a port for you.
2	**public Socket accept() throws IOException** Waits for an incoming client. This method blocks until either a client connects to the server on the specified port or the socket times out, assuming that the time-out value has been set using the setSoTimeout() method. Otherwise, this method blocks indefinitely
3	**public void setSoTimeout(int timeout)** Sets the time-out value for how long the server socket waits for a client during the accept().
4	**public void bind(SocketAddress host, int backlog)** Binds the socket to the specified server and port in the SocketAddress object. Use this method if you instantiated the ServerSocket using the no-argument constructor.

When the ServerSocket invokes accept(), the method does not return until a client connects. After a client does connect, the ServerSocket creates a new Socket on an unspecified port and returns a reference to this new Socket. A TCP connection now exists between the client and server, and communication can begin.

Socket Class Methods:

The **java.net.Socket** class represents the socket that both the client and server use to communicate with each other. The client obtains a Socket object by instantiating one, whereas the server obtains a Socket object from the return value of the accept() method.

The Socket class has five constructors that a client uses to connect to a server:

SN	Methods with Description
1	**public Socket(String host, int port) throws UnknownHostException, IOException.** This method attempts to connect to the specified server at the specified port. If this constructor does not throw an exception, the connection is successful and the client is connected to the server.
2	**public Socket(InetAddress host, int port) throws IOException** This method is identical to the previous constructor, except that the host is denoted by an InetAddress object.
3	**public Socket(String host, int port, InetAddress localAddress, int localPort) throws IOException.** Connects to the specified host and port, creating a socket on the local host at the specified address and port.
4	**public Socket(InetAddress host, int port, InetAddress localAddress, int localPort) throws IOException.** This method is identical to the previous constructor, except that the host is denoted by an InetAddress object instead of a String
5	**public Socket()** Creates an unconnected socket. Use the connect() method to connect this socket to a server.

When the Socket constructor returns, it does not simply instantiate a Socket object but it actually attempts to connect to the specified server and port.

Some methods of interest in the Socket class are listed here. Notice that both the client and server have a Socket object, so these methods can be invoked by both the client and server.

SN	Methods with Description
1	**public void connect(SocketAddress host, int timeout) throws IOException** This method connects the socket to the specified host. This method is needed only when you instantiated the Socket using the no-argument constructor.
2	**public InetAddress getInetAddress()** This method returns the address of the other computer that this socket is connected to.
3	**public int getPort()** Returns the port the socket is bound to on the remote machine.
4	**public int getLocalPort()** Returns the port the socket is bound to on the local machine.
5	**public SocketAddress getRemoteSocketAddress()** Returns the address of the remote socket.

6	**public InputStream getInputStream() throws IOException** Returns the input stream of the socket. The input stream is connected to the output stream of the remote socket.
7	**public OutputStream getOutputStream() throws IOException** Returns the output stream of the socket. The output stream is connected to the input stream of the remote socket
8	**public void close() throws IOException** Closes the socket, which makes this Socket object no longer capable of connecting again to any server

InetAddress Class Methods:

This class represents an Internet Protocol (IP) address. Here are following usefull methods which you would need while doing socket programming:

SN	Methods with Description
1	**static InetAddress getByAddress(byte[] addr)** Returns an InetAddress object given the raw IP address .
2	**static InetAddress getByAddress(String host, byte[] addr)** Create an InetAddress based on the provided host name and IP address.
3	**static InetAddress getByName(String host)** Determines the IP address of a host, given the host's name.
4	**String getHostAddress()** Returns the IP address string in textual presentation.
5	**String getHostName()** Gets the host name for this IP address.
6	**static InetAddress InetAddress getLocalHost()** Returns the local host.
7	**String toString()** Converts this IP address to a String.

Socket Client Example:

The following GreetingClient is a client program that connects to a server by using a socket and sends a greeting, and then waits for a response.

```
// File Name GreetingClient.java


import java.net.*;

import java.io.*;
```

```
public class GreetingClient
{
   public static void main(String [] args)
   {
      String serverName = args[0];
      int port = Integer.parseInt(args[1]);
      try
      {
         System.out.println("Connecting to " + serverName
                            + " on port " + port);
         Socket client = new Socket(serverName, port);
         System.out.println("Just connected to "
                     + client.getRemoteSocketAddress());
         OutputStream outToServer = client.getOutputStream();
         DataOutputStream out =
                      new DataOutputStream(outToServer);

         out.writeUTF("Hello from "
                     + client.getLocalSocketAddress());
         InputStream inFromServer = client.getInputStream();
         DataInputStream in =
                       new DataInputStream(inFromServer);
         System.out.println("Server says " + in.readUTF());
         client.close();
      }catch(IOException e)
      {
         e.printStackTrace();
      }
   }
}
```

Socket Server Example:

The following GreetingServer program is an example of a server application that uses the Socket class to listen for clients on a port number specified by a command-line argument:

```
// File Name GreetingServer.java


import java.net.*;
import java.io.*;


public class GreetingServer extends Thread
```

```
{
   private ServerSocket serverSocket;

   public GreetingServer(int port) throws IOException
   {
      serverSocket = new ServerSocket(port);
      serverSocket.setSoTimeout(10000);
   }

   public void run()
   {
      while(true)
      {
         try
         {
            System.out.println("Waiting for client on port " +
            serverSocket.getLocalPort() + "...");
            Socket server = serverSocket.accept();
            System.out.println("Just connected to "
                  + server.getRemoteSocketAddress());
            DataInputStream in =
                  new DataInputStream(server.getInputStream());
            System.out.println(in.readUTF());
            DataOutputStream out =
                 new DataOutputStream(server.getOutputStream());
            out.writeUTF("Thank you for connecting to "
              + server.getLocalSocketAddress() + "\nGoodbye!");
            server.close();
         }catch(SocketTimeoutException s)
         {
            System.out.println("Socket timed out!");
            break;
         }catch(IOException e)
         {
            e.printStackTrace();
            break;
         }
      }
   }
   public static void main(String [] args)
   {
```

```
      int port = Integer.parseInt(args[0]);
      try
      {
         Thread t = new GreetingServer(port);
         t.start();
      }catch(IOException e)
      {
         e.printStackTrace();
      }
   }
}
```

Compile client and server and then start server as follows:

```
$ java GreetingServer 6066
Waiting for client on port 6066...
```

Check client program as follows:

```
$ java GreetingClient localhost 6066
Connecting to localhost on port 6066
Just connected to localhost/127.0.0.1:6066
Server says Thank you for connecting to /127.0.0.1:6066
Goodbye!
```

Java - Sending Email

To send an email using your Java Application is simple enough but to start with you should have **JavaMail API** and **Java Activation Framework (JAF)** installed on your machine.

- You can download latest version of JavaMail (Version 1.2) from Java's standard website.
- You can download latest version of JAF (Version 1.1.1) from Java's standard website.

Download and unzip these files, in the newly created top level directories you will find a number of jar files for both the applications. You need to add **mail.jar** and **activation.jar** files in your CLASSPATH.

Send a Simple Email:

Here is an example to send a simple email from your machine. Here it is assumed that your **localhost** is connected to the internet and capable enough to send an email.

```
// File Name SendEmail.java


import java.util.*;
import javax.mail.*;
```

```
import javax.mail.internet.*;
import javax.activation.*;

public class SendEmail
{
   public static void main(String [] args)
   {
      // Recipient's email ID needs to be mentioned.
      String to = "abcd@gmail.com";

      // Sender's email ID needs to be mentioned
      String from = "web@gmail.com";

      // Assuming you are sending email from localhost
      String host = "localhost";

      // Get system properties
      Properties properties = System.getProperties();

      // Setup mail server
      properties.setProperty("mail.smtp.host", host);

      // Get the default Session object.
      Session session = Session.getDefaultInstance(properties);

      try{
         // Create a default MimeMessage object.
         MimeMessage message = new MimeMessage(session);

         // Set From: header field of the header.
         message.setFrom(new InternetAddress(from));

         // Set To: header field of the header.
         message.addRecipient(Message.RecipientType.TO,
                                 new InternetAddress(to));

         // Set Subject: header field
         message.setSubject("This is the Subject Line!");

         // Now set the actual message
         message.setText("This is actual message");
```

```
         // Send message
         Transport.send(message);
         System.out.println("Sent message successfully....");
      }catch (MessagingException mex) {
         mex.printStackTrace();
      }
   }
}
```

Compile and run this program to send a simple email:

```
$ java SendEmail
Sent message successfully....
```

If you want to send an email to multiple recipients then following methods would be used to specify multiple email IDs:

```
void addRecipients(Message.RecipientType type,
                   Address[] addresses)
throws MessagingException
```

Here is the description of the parameters:

- **type:** This would be set to TO, CC or BCC. Here CC represents Carbon Copy and BCC represents Black Carbon Copy. Example *Message.RecipientType.TO*
- **addresses:** This is the array of email ID. You would need to use InternetAddress() method while specifying email IDs

Send an HTML Email:

Here is an example to send an HTML email from your machine. Here it is assumed that your **localhost** is connected to the internet and capable enough to send an email.

This example is very similar to previous one, except here we are using setContent() method to set content whose second argument is "text/html" to specify that the HTML content is included in the message.

Using this example, you can send as big as HTML content you like.

```
// File Name SendHTMLEmail.java


import java.util.*;
import javax.mail.*;
import javax.mail.internet.*;
import javax.activation.*;


public class SendHTMLEmail
```

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

      // Recipient's email ID needs to be mentioned.
      String to = "abcd@gmail.com";

      // Sender's email ID needs to be mentioned
      String from = "web@gmail.com";

      // Assuming you are sending email from localhost
      String host = "localhost";

      // Get system properties
      Properties properties = System.getProperties();

      // Setup mail server
      properties.setProperty("mail.smtp.host", host);

      // Get the default Session object.
      Session session = Session.getDefaultInstance(properties);

      try{
         // Create a default MimeMessage object.
         MimeMessage message = new MimeMessage(session);

         // Set From: header field of the header.
         message.setFrom(new InternetAddress(from));

         // Set To: header field of the header.
         message.addRecipient(Message.RecipientType.TO,
                                  new InternetAddress(to));

         // Set Subject: header field
         message.setSubject("This is the Subject Line!");

         // Send the actual HTML message, as big as you like
         message.setContent("<h1>This is actual message</h1>",
                            "text/html" );

         // Send message
```

```
         Transport.send(message);
         System.out.println("Sent message successfully....");
      }catch (MessagingException mex) {
         mex.printStackTrace();
      }
   }
}
```

Compile and run this program to send an HTML email:

```
$ java SendHTMLEmail
Sent message successfully....
```

Send Attachment in Email:

Here is an example to send an email with attachment from your machine. Here it is assumed that your **localhost** is connected to the internet and capable enough to send an email.

```
// File Name SendFileEmail.java

import java.util.*;
import javax.mail.*;
import javax.mail.internet.*;
import javax.activation.*;

public class SendFileEmail
{
   public static void main(String [] args)
   {

      // Recipient's email ID needs to be mentioned.
      String to = "abcd@gmail.com";

      // Sender's email ID needs to be mentioned
      String from = "web@gmail.com";

      // Assuming you are sending email from localhost
      String host = "localhost";

      // Get system properties
      Properties properties = System.getProperties();

      // Setup mail server
```

```
      properties.setProperty("mail.smtp.host", host);

      // Get the default Session object.
      Session session = Session.getDefaultInstance(properties);

      try{
         // Create a default MimeMessage object.
         MimeMessage message = new MimeMessage(session);

         // Set From: header field of the header.
         message.setFrom(new InternetAddress(from));

         // Set To: header field of the header.
         message.addRecipient(Message.RecipientType.TO,
                                  new InternetAddress(to));

         // Set Subject: header field
         message.setSubject("This is the Subject Line!");

         // Create the message part
         BodyPart messageBodyPart = new MimeBodyPart();

         // Fill the message
         messageBodyPart.setText("This is message body");

         // Create a multipar message
         Multipart multipart = new MimeMultipart();

         // Set text message part
         multipart.addBodyPart(messageBodyPart);

         // Part two is attachment
         messageBodyPart = new MimeBodyPart();
         String filename = "file.txt";
         DataSource source = new FileDataSource(filename);
         messageBodyPart.setDataHandler(new DataHandler(source));
         messageBodyPart.setFileName(filename);
         multipart.addBodyPart(messageBodyPart);

         // Send the complete message parts
         message.setContent(multipart );
```

```
         // Send message
         Transport.send(message);
         System.out.println("Sent message successfully....");
      }catch (MessagingException mex) {
         mex.printStackTrace();
      }
   }
}
```

Compile and run this program to send an HTML email:

```
$ java SendFileEmail
Sent message successfully....
```

User Authentication Part:

If it is required to provide user ID and Password to the email server for authentication purpose then you can set these properties as follows:

```
 props.setProperty("mail.user", "myuser");
 props.setProperty("mail.password", "mypwd");
```

Rest of the email sending mechanism would remain as explained above.

Java – Multithreading

Java provides built-in support for *multithreaded programming*. A multithreaded program contains two or more parts that can run concurrently. Each part of such a program is called a thread, and each thread defines a separate path of execution.

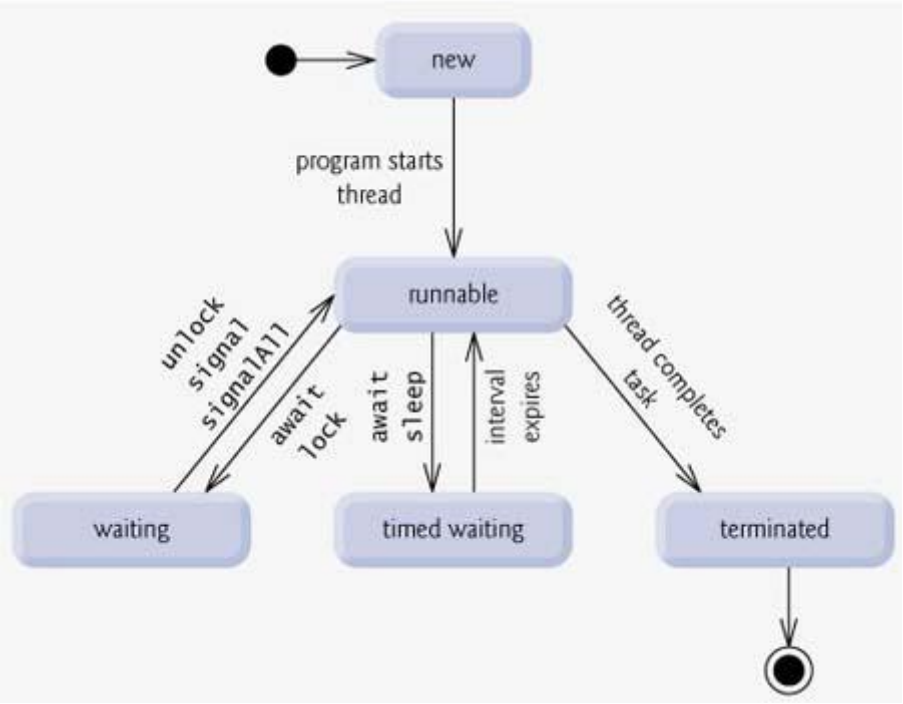
A multithreading is a specialized form of multitasking. Multithreading requires less overhead than multitasking processing.

I need to define another term related to threads: **process:** A process consists of the memory space allocated by the operating system that can contain one or more threads. A thread cannot exist on its own; it must be a part of a process. A process remains running until all of the non-daemon threads are done executing.

Multithreading enables you to write very efficient programs that make maximum use of the CPU, because idle time can be kept to a minimum.

Life Cycle of a Thread:

A thread goes through various stages in its life cycle. For example, a thread is born, started, runs, and then dies. Following diagram shows complete life cycle of a thread.



Above mentioned stages are explained here:

- **New:** A new thread begins its life cycle in the new state. It remains in this state until the program starts the thread. It is also referred to as a born thread.
- **Runnable:** After a newly born thread is started, the thread becomes runnable. A thread in this state is considered to be executing its task.
- **Waiting:** Sometimes a thread transitions to the waiting state while the thread waits for another thread to perform a task.A thread transitions back to the runnable state only when another thread signals the waiting thread to continue executing.
- **Timed waiting:** A runnable thread can enter the timed waiting state for a specified interval of time. A thread in this state transitions back to the runnable state when that time interval expires or when the event it is waiting for occurs.
- **Terminated:** A runnable thread enters the terminated state when it completes its task or otherwise terminates.

Thread Priorities:

Every Java thread has a priority that helps the operating system determine the order in which threads are scheduled.

Java priorities are in the range between MIN_PRIORITY (a constant of 1) and MAX_PRIORITY (a constant of 10). By default, every thread is given priority NORM_PRIORITY (a constant of 5).

Threads with higher priority are more important to a program and should be allocated processor time before lower-priority threads. However, thread priorities cannot guarantee the order in which threads execute and very much platform dependentant.

Creating a Thread:

Java defines two ways in which this can be accomplished:

- You can implement the Runnable interface.
- You can extend the Thread class, itself.

Create Thread by Implementing Runnable:

The easiest way to create a thread is to create a class that implements the **Runnable** interface.

To implement Runnable, a class need only implement a single method called **run( )**, which is declared like this:

```
public void run( )
```

You will define the code that constitutes the new thread inside run() method. It is important to understand that run() can call other methods, use other classes, and declare variables, just like the main thread can.

After you create a class that implements Runnable, you will instantiate an object of type Thread from within that class. Thread defines several constructors. The one that we will use is shown here:

```
Thread(Runnable threadOb, String threadName);
```

Here *threadOb* is an instance of a class that implements the Runnable interface and the name of the new thread is specified by *threadName*.

After the new thread is created, it will not start running until you call its **start( )** method, which is declared within Thread. The start( ) method is shown here:

```
void start( );
```

Example:

Here is an example that creates a new thread and starts it running:

```
// Create a new thread.
class NewThread implements Runnable {
   Thread t;
   NewThread() {
      // Create a new, second thread
      t = new Thread(this, "Demo Thread");
      System.out.println("Child thread: " + t);
      t.start(); // Start the thread
   }

   // This is the entry point for the second thread.
   public void run() {
      try {
         for(int i = 5; i > 0; i--) {
            System.out.println("Child Thread: " + i);
```

```
            // Let the thread sleep for a while.
            Thread.sleep(500);
         }
     } catch (InterruptedException e) {
         System.out.println("Child interrupted.");
     }
     System.out.println("Exiting child thread.");
   }
}

public class ThreadDemo {
   public static void main(String args[]) {
      new NewThread(); // create a new thread
      try {
         for(int i = 5; i > 0; i--) {
           System.out.println("Main Thread: " + i);
           Thread.sleep(1000);
         }
      } catch (InterruptedException e) {
         System.out.println("Main thread interrupted.");
      }
      System.out.println("Main thread exiting.");
   }
}
```

This would produce following result:

```
Child thread: Thread[Demo Thread,5,main]
Main Thread: 5
Child Thread: 5
Child Thread: 4
Main Thread: 4
Child Thread: 3
Child Thread: 2
Main Thread: 3
Child Thread: 1
Exiting child thread.
Main Thread: 2
Main Thread: 1
Main thread exiting.
```

Create Thread by Extending Thread:

The second way to create a thread is to create a new class that extends **Thread**, and then to create an instance of that class.

The extending class must override the **run( )** method, which is the entry point for the new thread. It must also call **start( )** to begin execution of the new thread.

Example:

Here is the preceding program rewritten to extend Thread:

```
// Create a second thread by extending Thread
class NewThread extends Thread {
   NewThread() {
      // Create a new, second thread
      super("Demo Thread");
      System.out.println("Child thread: " + this);
      start(); // Start the thread
   }

   // This is the entry point for the second thread.
   public void run() {
      try {
         for(int i = 5; i > 0; i--) {
            System.out.println("Child Thread: " + i);
                           // Let the thread sleep for a while.
            Thread.sleep(500);
         }
      } catch (InterruptedException e) {
         System.out.println("Child interrupted.");
      }
      System.out.println("Exiting child thread.");
   }
}

public class ExtendThread {
   public static void main(String args[]) {
      new NewThread(); // create a new thread
      try {
         for(int i = 5; i > 0; i--) {
            System.out.println("Main Thread: " + i);
            Thread.sleep(1000);
         }
      } catch (InterruptedException e) {
```

```
         System.out.println("Main thread interrupted.");
      }
      System.out.println("Main thread exiting.");
   }
}
```

This would produce following result:

```
Child thread: Thread[Demo Thread,5,main]
Main Thread: 5
Child Thread: 5
Child Thread: 4
Main Thread: 4
Child Thread: 3
Child Thread: 2
Main Thread: 3
Child Thread: 1
Exiting child thread.
Main Thread: 2
Main Thread: 1
Main thread exiting.
```

Thread Methods:

Following is the list of important medthods available in the Thread class.

SN	Methods with Description
1	**public void start()** Starts the thread in a separate path of execution, then invokes the run() method on this Thread object.
2	**public void run()** If this Thread object was instantiated using a separate Runnable target, the run() method is invoked on that Runnable object.
3	**public final void setName(String name)** Changes the name of the Thread object. There is also a getName() method for retrieving the name.
4	**public final void setPriority(int priority)** Sets the priority of this Thread object. The possible values are between 1 and 10.
5	**public final void setDaemon(boolean on)** A parameter of true denotes this Thread as a daemon thread.
6	**public final void join(long millisec)** The current thread invokes this method on a second thread, causing the current thread to block until the second

	thread terminates or the specified number of milliseconds passes.
7	**public void interrupt()** Interrupts this thread, causing it to continue execution if it was blocked for any reason.
8	**public final boolean isAlive()** Returns true if the thread is alive, which is any time after the thread has been started but before it runs to completion.

The previous methods are invoked on a particular Thread object. The following methods in the Thread class are static. Invoking one of the static methods performs the operation on the currently running thread

SN	Methods with Description
1	**public static void yield()** Causes the currently running thread to yield to any other threads of the same priority that are waiting to be scheduled
2	**public static void sleep(long millisec)** Causes the currently running thread to block for at least the specified number of milliseconds
3	**public static boolean holdsLock(Object x)** Returns true if the current thread holds the lock on the given Object.
4	**public static Thread currentThread()** Returns a reference to the currently running thread, which is the thread that invokes this method.
5	**public static void dumpStack()** Prints the stack trace for the currently running thread, which is useful when debugging a multithreaded application.

Example:

The following ThreadClassDemo program demonstrates some of these methods of the Thread class:

```
// File Name : DisplayMessage.java
// Create a thread to implement Runnable
public class DisplayMessage implements Runnable
{
   private String message;
   public DisplayMessage(String message)
   {
      this.message = message;
   }
   public void run()
   {
      while(true)
```

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

// File Name : GuessANumber.java
// Create a thread to extentd Thread
public class GuessANumber extends Thread
{
   private int number;
   public GuessANumber(int number)
   {
      this.number = number;
   }
   public void run()
   {
      int counter = 0;
      int guess = 0;
      do
      {
          guess = (int) (Math.random() * 100 + 1);
          System.out.println(this.getName()
                      + " guesses " + guess);
          counter++;
      }while(guess != number);
      System.out.println("** Correct! " + this.getName()
                      + " in " + counter + " guesses.**");
   }
}

// File Name : ThreadClassDemo.java
public class ThreadClassDemo
{
   public static void main(String [] args)
   {
      Runnable hello = new DisplayMessage("Hello");
      Thread thread1 = new Thread(hello);
      thread1.setDaemon(true);
      thread1.setName("hello");
      System.out.println("Starting hello thread...");
```

```
      thread1.start();

      Runnable bye = new DisplayMessage("Goodbye");
      Thread thread2 = new Thread(hello);
      thread2.setPriority(Thread.MIN_PRIORITY);
      thread2.setDaemon(true);
      System.out.println("Starting goodbye thread...");
      thread2.start();

      System.out.println("Starting thread3...");
      Thread thread3 = new GuessANumber(27);
      thread3.start();
      try
      {
         thread3.join();
      }catch(InterruptedException e)
      {
         System.out.println("Thread interrupted.");
      }
      System.out.println("Starting thread4...");
      Thread thread4 = new GuessANumber(75);

           thread4.start();
      System.out.println("main() is ending...");
   }
}
```

This would produce following result. You can try this example again and again and you would get different result every time.

```
Starting hello thread...
Starting goodbye thread...
Hello
Hello
Hello
Hello
Hello
Hello
Hello
Hello
Hello
Thread-2 guesses 27
```

```
Hello

** Correct! Thread-2 in 102 guesses.**

Hello

Starting thread4...

Hello

Hello

.........remaining result produced.
```

Major Thread Concepts:

While doing Multithreading programming, you would need to have following concepts very handy:

- Thread Synchronization
- Interthread Communication
- Thread Deadlock
- Thread Control: Suspend, Stop and Resume

Using Multithreading:

The key to utilizing multithreading support effectively is to think concurrently rather than serially. For example, when you have two subsystems within a program that can execute concurrently, make them individual threads.

With the careful use of multithreading, you can create very efficient programs. A word of caution is in order, however: If you create too many threads, you can actually degrade the performance of your program rather than enhance it.

Remember, some overhead is associated with context switching. If you create too many threads, more CPU time will be spent changing contexts than executing your program!

Java - Applet Basics

An applet is a Java program that runs in a Web browser. An applet can be a fully functional Java application because it has the entire Java API at its disposal.

There are some important differences between an applet and a standalone Java application, including the following:

- An applet is a Java class that extends the java.applet.Applet class.
- A main() method is not invoked on an applet, and an applet class will not define main().
- Applets are designed to be embedded within an HTML page.
- When a user views an HTML page that contains an applet, the code for the applet is downloaded to the user's machine.
- A JVM is required to view an applet. The JVM can be either a plug-in of the Web browser or a separate runtime environment.
- The JVM on the user's machine creates an instance of the applet class and invokes various methods during the applet's lifetime.

- Applets have strict security rules that are enforced by the Web browser. The security of an applet is often referred to as sandbox security, comparing the applet to a child playing in a sandbox with various rules that must be followed.

- Other classes that the applet needs can be downloaded in a single Java Archive (JAR) file.

Life Cycle of an Applet:

Four methods in the Applet class give you the framework on which you build any serious applet:

- **init:** This method is intended for whatever initialization is needed for your applet. It is called after the param tags inside the applet tag have been processed.

- **start:** This method is automatically called after the browser calls the init method. It is also called whenever the user returns to the page containing the applet after having gone off to other pages.

- **stop:** This method is automatically called when the user moves off the page on which the applet sits. It can, therefore, be called repeatedly in the same applet.

- **destroy:** This method is only called when the browser shuts down normally. Because applets are meant to live on an HTML page, you should not normally leave resources behind after a user leaves the page that contains the applet.

- **paint:** Invoked immediately after the start() method, and also any time the applet needs to repaint itself in the browser. The paint() method is actually inherited from the java.awt.

A "Hello, World" Applet:

The following is a simple applet named HelloWorldApplet.java:

```
import java.applet.*;
import java.awt.*;


public class HelloWorldApplet extends Applet
{
   public void paint (Graphics g)
   {
      g.drawString ("Hello World", 25, 50);
   }
}
```

These import statements bring the classes into the scope of our applet class:

- java.applet.Applet.

- java.awt.Graphics.

Without those import statements, the Java compiler would not recognize the classes Applet and Graphics, which the applet class refers to.

The Applet CLASS:

Every applet is an extension of the *java.applet.Applet class*. The base Applet class provides methods that a derived Applet class may call to obtain information and services from the browser context.

These include methods that do the following:

- Get applet parameters
- Get the network location of the HTML file that contains the applet
- Get the network location of the applet class directory
- Print a status message in the browser
- Fetch an image
- Fetch an audio clip
- Play an audio clip
- Resize the applet

Additionally, the Applet class provides an interface by which the viewer or browser obtains information about the applet and controls the applet's execution. The viewer may:

- request information about the author, version and copyright of the applet
- request a description of the parameters the applet recognizes
- initialize the applet
- destroy the applet
- start the applet's execution
- stop the applet's execution

The Applet class provides default implementations of each of these methods. Those implementations may be overridden as necessary.

The "Hello, World" applet is complete as it stands. The only method overridden is the paint method.

Invoking an Applet:

An applet may be invoked by embedding directives in an HTML file and viewing the file through an applet viewer or Java-enabled browser.

The <applet> tag is the basis for embedding an applet in an HTML file. Below is an example that invokes the "Hello, World" applet:

```
<html>
<title>The Hello, World Applet</title>
<hr>
<applet code="HelloWorldApplet.class" width="320" height="120">
If your browser was Java-enabled, a "Hello, World"
```

```
message would appear here.
</applet>
<hr>
</html>
```

Based on the above examples, here is the live applet example: Applet Example.

Note: You can refer to HTML Applet Tag to understand more about calling applet from HTML.

The code attribute of the <applet> tag is required. It specifies the Applet class to run. Width and height are also required to specify the initial size of the panel in which an applet runs. The applet directive must be closed with a </applet> tag.

If an applet takes parameters, values may be passed for the parameters by adding <param> tags between <applet> and </applet>. The browser ignores text and other tags between the applet tags.

Non-Java-enabled browsers do not process <applet> and </applet>. Therefore, anything that appears between the tags, not related to the applet, is visible in non-Java-enabled browsers.

The viewer or browser looks for the compiled Java code at the location of the document. To specify otherwise, use the codebase attribute of the <applet> tag as shown:

```
<applet codebase="http://amrood.com/applets"
code="HelloWorldApplet.class" width="320" height="120">
```

If an applet resides in a package other than the default, the holding package must be specified in the code attribute using the period character (.) to separate package/class components. For example:

```
<applet code="mypackage.subpackage.TestApplet.class"
           width="320" height="120">
```

Getting Applet Parameters:

The following example demonstrates how to make an applet respond to setup parameters specified in the document. This applet displays a checkerboard pattern of black and a second color.

The second color and the size of each square may be specified as parameters to the applet within the document.

CheckerApplet gets its parameters in the init() method. It may also get its parameters in the paint() method. However, getting the values and saving the settings once at the start of the applet, instead of at every refresh, is convenient and efficient.

The applet viewer or browser calls the init() method of each applet it runs. The viewer calls init() once, immediately after loading the applet. (Applet.init() is implemented to do nothing.) Override the default implementation to insert custom initialization code.

The Applet.getParameter() method fetches a parameter given the parameter's name (the value of a parameter is always a string). If the value is numeric or other non-character data, the string must be parsed.

The following is a skeleton of CheckerApplet.java:

```
import java.applet.*;
import java.awt.*;
public class CheckerApplet extends Applet
```

```
{
   int squareSize = 50;// initialized to default size
   public void init () {}
   private void parseSquareSize (String param) {}
   private Color parseColor (String param) {}
   public void paint (Graphics g) {}
}
```

Here are CheckerApplet's init() and private parseSquareSize() methods:

```
public void init ()
{
   String squareSizeParam = getParameter ("squareSize");
   parseSquareSize (squareSizeParam);
   String colorParam = getParameter ("color");
   Color fg = parseColor (colorParam);
   setBackground (Color.black);
   setForeground (fg);
}
private void parseSquareSize (String param)
{
   if (param == null) return;
   try {
      squareSize = Integer.parseInt (param);
   }
   catch (Exception e) {
     // Let default value remain
   }
}
```

The applet calls parseSquareSize() to parse the squareSize parameter. parseSquareSize() calls the library method Integer.parseInt(), which parses a string and returns an integer. Integer.parseInt() throws an exception whenever its argument is invalid.

Therefore, parseSquareSize() catches exceptions, rather than allowing the applet to fail on bad input.

The applet calls parseColor() to parse the color parameter into a Color value. parseColor() does a series of string comparisons to match the parameter value to the name of a predefined color. You need to implement these methods to make this applet works.

Specifying Applet Parameters:

The following is an example of an HTML file with a CheckerApplet embedded in it. The HTML file specifies both parameters to the applet by means of the <param> tag.

```
<html>
```

```
<title>Checkerboard Applet</title>
<hr>
<applet code="CheckerApplet.class" width="480" height="320">
<param name="color" value="blue">
<param name="squaresize" value="30">
</applet>
<hr>
</html>
```

Note: Parameter names are not case sensitive.

Application Conversion to Applets:

It is easy to convert a graphical Java application (that is, an application that uses the AWT and that you can start with the java program launcher) into an applet that you can embed in a web page.

Here are the specific steps for converting an application to an applet.

- Make an HTML page with the appropriate tag to load the applet code.
- Supply a subclass of the JApplet class. Make this class public. Otherwise, the applet cannot be loaded.
- Eliminate the main method in the application. Do not construct a frame window for the application. Your application will be displayed inside the browser.
- Move any initialization code from the frame window constructor to the init method of the applet. You don't need to explicitly construct the applet object.the browser instantiates it for you and calls the init method.
- Remove the call to setSize; for applets, sizing is done with the width and height parameters in the HTML file.
- Remove the call to setDefaultCloseOperation. An applet cannot be closed; it terminates when the browser exits.
- If the application calls setTitle, eliminate the call to the method. Applets cannot have title bars. (You can, of course, title the web page itself, using the HTML title tag.)
- Don't call setVisible(true). The applet is displayed automatically.

Event Handling:

Applets inherit a group of event-handling methods from the Container class. The Container class defines several methods, such as processKeyEvent and processMouseEvent, for handling particular types of events, and then one catch-all method called processEvent.

Inorder to react an event, an applet must override the appropriate event-specific method.

```
import java.awt.event.MouseListener;
import java.awt.event.MouseEvent;
import java.applet.Applet;
import java.awt.Graphics;
```

```
public class ExampleEventHandling extends Applet
                              implements MouseListener {

    StringBuffer strBuffer;

    public void init() {
          addMouseListener(this);
          strBuffer = new StringBuffer();
        addItem("initializing the apple ");
    }

    public void start() {
        addItem("starting the applet ");
    }

    public void stop() {
        addItem("stopping the applet ");
    }

    public void destroy() {
        addItem("unloading the applet");
    }

    void addItem(String word) {
        System.out.println(word);
        strBuffer.append(word);
        repaint();
    }

    public void paint(Graphics g) {
          //Draw a Rectangle around the applet's display area.
        g.drawRect(0, 0,
                   getWidth() - 1,
                   getHeight() - 1);

          //display the string inside the rectangle.
        g.drawString(strBuffer.toString(), 10, 20);
    }



    public void mouseEntered(MouseEvent event) {
```

```
    }
    public void mouseExited(MouseEvent event) {
    }
    public void mousePressed(MouseEvent event) {
    }
    public void mouseReleased(MouseEvent event) {
    }

    public void mouseClicked(MouseEvent event) {
          addItem("mouse clicked! ");
    }
}
```

Now let us call this applet as follows:

```
<html>
<title>Event Handling</title>
<hr>
<applet code="ExampleEventHandling.class"
width="300" height="300">
</applet>
<hr>
</html>
```

Initially the applet will display "initializing the applet. Starting the applet." Then once you click inside the rectangle "mouse clicked" will be displayed as well.

Based on the above examples, here is the live applet example: Applet Example.

Displaying Images:

An applet can display images of the format GIF, JPEG, BMP, and others. To display an image within the applet, you use the drawImage() method found in the java.awt.Graphics class.

Following is the example showing all the steps to show images:

```
import java.applet.*;
import java.awt.*;
import java.net.*;
public class ImageDemo extends Applet
{
  private Image image;
  private AppletContext context;
  public void init()
  {
```

```
      context = this.getAppletContext();
      String imageURL = this.getParameter("image");
      if(imageURL == null)
      {
         imageURL = "java.jpg";
      }
      try
      {
         URL url = new URL(this.getDocumentBase(), imageURL);
         image = context.getImage(url);
      }catch(MalformedURLException e)
      {
         e.printStackTrace();
         // Display in browser status bar
         context.showStatus("Could not load image!");
      }
   }
   public void paint(Graphics g)
   {
      context.showStatus("Displaying image");
      g.drawImage(image, 0, 0, 200, 84, null);
      g.drawString("www.javalicense.com", 35, 100);
   }
}
```

Now let us call this applet as follows:

```
<html>
<title>The ImageDemo applet</title>
<hr>
<applet code="ImageDemo.class" width="300" height="200">
<param name="image" value="java.jpg">
</applet>
<hr>
</html>
```

Based on the above examples, here is the live applet example: Applet Example.

Playing Audio:

An applet can play an audio file represented by the AudioClip interface in the java.applet package. The AudioClip interface has three methods, including:

- **public void play():** Plays the audio clip one time, from the beginning.

- **public void loop():** Causes the audio clip to replay continually.
- **public void stop():** Stops playing the audio clip.

To obtain an AudioClip object, you must invoke the getAudioClip() method of the Applet class. The getAudioClip() method returns immediately, whether or not the URL resolves to an actual audio file. The audio file is not downloaded until an attempt is made to play the audio clip.

Following is the example showing all the steps to play an audio:

```
import java.applet.*;
import java.awt.*;
import java.net.*;
public class AudioDemo extends Applet
{
   private AudioClip clip;
   private AppletContext context;
   public void init()
   {
      context = this.getAppletContext();
      String audioURL = this.getParameter("audio");
      if(audioURL == null)
      {
         audioURL = "default.au";
      }
      try
      {
         URL url = new URL(this.getDocumentBase(), audioURL);
         clip = context.getAudioClip(url);
      }catch(MalformedURLException e)
      {
         e.printStackTrace();
         context.showStatus("Could not load audio file!");
      }
   }
   public void start()
   {
      if(clip != null)
      {
         clip.loop();
      }
   }
   public void stop()
```

```
   {
      if(clip != null)
      {
         clip.stop();
      }
   }
}
```

Now let us call this applet as follows:

```
<html>
<title>The ImageDemo applet</title>
<hr>
<applet code="ImageDemo.class" width="0" height="0">
<param name="audio" value="test.wav">
</applet>
<hr>
</html>
```

You can use your test.wav at your PC to test the above example.

Java Documentation Comments

Java supports three types of comments. The first two are the // and the /* */. The third type is called a documentation comment. It begins with the character sequence /** and it ends with */.

Documentation comments allow you to embed information about your program into the program itself. You can then use the javadoc utility program to extract the information and put it into an HTML file.

Documentation comments make it convenient to document your programs.

The javadoc Tags:

The javadoc utility recognizes the following tags:

Tag	Description	Example
@author	Identifies the author of a class.	@author description
@deprecated	Specifies that a class or member is deprecated.	@deprecated description
{@docRoot}	Specifies the path to the root directory of the current documentation	Directory Path
@exception	Identifies an exception thrown by a method.	@exception exception-name explanation

{@inheritDoc}	Inherits a comment from the immediate superclass.	Inherits a comment from the immediate surperclass.
{@link}	Inserts an in-line link to another topic.	{@link name text}
{@linkplain}	Inserts an in-line link to another topic, but the link is displayed in a plain-text font.	Inserts an in-line link to another topic.
@param	Documents a method's parameter.	@param parameter-name explanation
@return	Documents a method's return value.	@return explanation
@see	Specifies a link to another topic.	@see anchor
@serial	Documents a default serializable field.	@serial description
@serialData	Documents the data written by the writeObject( ) or writeExternal( ) methods	@serialData description
@serialField	Documents an ObjectStreamField component.	@serialField name type description
@since	States the release when a specific change was introduced.	@since release
@throws	Same as @exception.	The @throws tag has the same meaning as the @exception tag.
{@value}	Displays the value of a constant, which must be a static field.	Displays the value of a constant, which must be a static field.
@version	Specifies the version of a class.	@version info

Documentation Comment:

After the beginning /**, the first line or lines become the main description of your class, variable, or method.

After that, you can include one or more of the various @ tags. Each @ tag must start at the beginning of a new line or follow an asterisk (*) that is at the start of a line.

Multiple tags of the same type should be grouped together. For example, if you have three @see tags, put them one after the other.

Here is an example of a documentation comment for a class:

```
/**
* This class draws a bar chart.
* @author Zara Ali
* @version 1.2
```

```
*/
```

What javadoc Outputs?

The javadoc program takes as input your Java program's source file and outputs several HTML files that contain the program's documentation.

Information about each class will be in its own HTML file. Java utility **javadoc** will also output an index and a hierarchy tree. Other HTML files can be generated.

Since different implementations of javadoc may work differently, you will need to check the instructions that accompany your Java development system for details specific to your version.

Example:

Following is a sample program that uses documentation comments. Notice the way each comment immediately precedes the item that it describes.

After being processed by javadoc, the documentation about the SquareNum class will be found in SquareNum.html.

```
import java.io.*;


/**
* This class demonstrates documentation comments.
* @author Ayan Amhed
* @version 1.2
*/
public class SquareNum {
   /**
   * This method returns the square of num.
   * This is a multiline description. You can use
   * as many lines as you like.
   * @param num The value to be squared.
   * @return num squared.
   */
   public double square(double num) {
      return num * num;
   }
   /**
   * This method inputs a number from the user.
   * @return The value input as a double.
   * @exception IOException On input error.
   * @see IOException
   */
   public double getNumber() throws IOException {
      InputStreamReader isr = new InputStreamReader(System.in);
```

```
      BufferedReader inData = new BufferedReader(isr);
      String str;
      str = inData.readLine();
      return (new Double(str)).doubleValue();
   }
   /**
   * This method demonstrates square().
   * @param args Unused.
   * @return Nothing.
   * @exception IOException On input error.
   * @see IOException
   */
   public static void main(String args[]) throws IOException
   {
      SquareNum ob = new SquareNum();
      double val;
      System.out.println("Enter value to be squared: ");
      val = ob.getNumber();
      val = ob.square(val);
      System.out.println("Squared value is " + val);
   }
}
```

Now process above SquareNum.java file using javadoc utility as follows:

```
$ javadoc SquareNum.java
Loading source file SquareNum.java...
Constructing Javadoc information...
Standard Doclet version 1.5.0_13
Building tree for all the packages and classes...
Generating SquareNum.html...
SquareNum.java:39: warning - @return tag cannot be used\
                      in method with void return type.
Generating package-frame.html...
Generating package-summary.html...
Generating package-tree.html...
Generating constant-values.html...
Building index for all the packages and classes...
Generating overview-tree.html...
Generating index-all.html...
Generating deprecated-list.html...
```

```
Building index for all classes...

Generating allclasses-frame.html...

Generating allclasses-noframe.html...

Generating index.html...

Generating help-doc.html...

Generating stylesheet.css...

1 warning

$
```

You can check all the generated documentation here: SquareNum

Java - Library Classes

This tutorial would cover package **java.lang** which provides classes that are fundamental to the design of the Java programming language. The most important classes are Object, which is the root of the class hierarchy, and Class, instances of which represent classes at run time.

Here is the list of classes of ackage **java.lang**. These classes are very important to know for a Java programmer. Click a class link to know more detail about that class. For a further drill, you can refer standard Java documentation.

SN	Methods with Description
1	Boolean Boolean
2	Byte The Byte class wraps a value of primitive type byte in an object.
3	Character The Character class wraps a value of the primitive type char in an object.
4	Class Instances of the class Class represent classes and interfaces in a running Java application.
5	ClassLoader A class loader is an object that is responsible for loading classes.
6	Compiler The Compiler class is provided to support Java-to-native-code compilers and related services.
7	Double The Double class wraps a value of the primitive type double in an object.
8	Float The Float class wraps a value of primitive type float in an object.
9	Integer The Integer class wraps a value of the primitive type int in an object.

10	Long The Long class wraps a value of the primitive type long in an object.
11	Math The class Math contains methods for performing basic numeric operations such as the elementary exponential, logarithm, square root, and trigonometric functions.
12	Number The abstract class Number is the superclass of classes BigDecimal, BigInteger, Byte, Double, Float, Integer, Long, and Short.
13	Object Class Object is the root of the class hierarchy.
14	Package Package objects contain version information about the implementation and specification of a Java package.
15	Process The Runtime.exec methods create a native process and return an instance of a subclass of Process that can be used to control the process and obtain information about it.
16	Runtime Every Java application has a single instance of class Runtime that allows the application to interface with the environment in which the application is running.
17	RuntimePermission This class is for runtime permissions.
18	SecurityManager The security manager is a class that allows applications to implement a security policy.
19	Short The Short class wraps a value of primitive type short in an object.
20	StackTraceElement An element in a stack trace, as returned by Throwable.getStackTrace().
21	StrictMath The class StrictMath contains methods for performing basic numeric operations such as the elementary exponential, logarithm, square root, and trigonometric functions.
22	String The String class represents character strings.
23	StringBuffer A string buffer implements a mutable sequence of characters.
24	System The System class contains several useful class fields and methods.

25	Thread A thread is a thread of execution in a program.
26	ThreadGroup A thread group represents a set of threads.
27	ThreadLocal This class provides thread-local variables.
28	Throwable The Throwable class is the superclass of all errors and exceptions in the Java language.
29	Void The Void class is an uninstantiable placeholder class to hold a reference to the Class object representing the Java keyword void.