

UNIT I

Introduction to java

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 describes 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 kinds 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:

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 the 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 the 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 let's 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, the 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 instance 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
```

Control Statements

There may be a situation 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 the 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);  
    }  
}
```

```
    }while( x < 20 );  
  }  
}
```

This would produce the 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 the 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 evaluates 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 the 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 the 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 the following result:

```
10
20
40
50
```

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, the following are true:

- Animal is the superclass of Mammal class.
- Animal is the superclass of Reptile class.
- Mammal and Reptile are subclasses 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 the 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 the 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 Van 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.

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 programmer 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 classes.

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\computers`.

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
-

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 subclass. 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 the 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);
        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 the 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 method consists 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
}
```

Interface

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.

A class 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 the 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 interfaces.
- 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.

Exception

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:

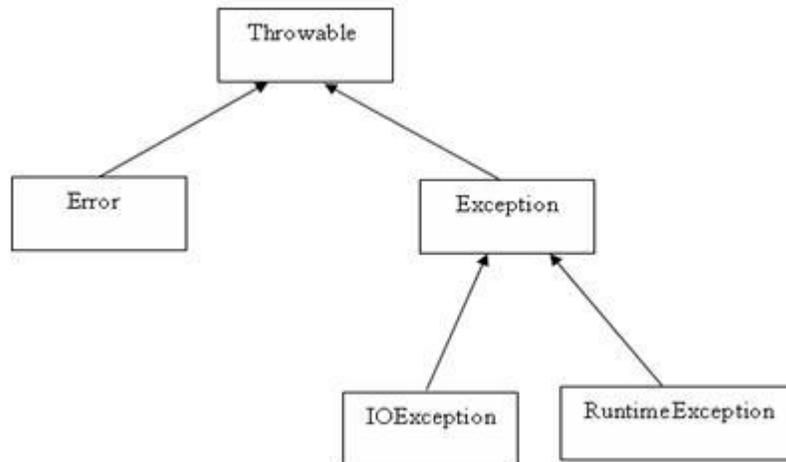
- **Checked exceptions:** A checked exception is an exception that is typically a user error or a problem that cannot be foreseen by the programmer. For example, if a file is to be opened, but the file cannot be found, an exception occurs. These exceptions cannot simply be ignored at the time of compilation.
- **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 compilation.
- **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 from 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 methods available in the Throwable class.

SN

Methods with Description

public String getMessage()

1 Returns a detailed message about the exception that has occurred. This message is initialized in the Throwable constructor.

public Throwable getCause()

2 Returns the cause of the exception as represented by a Throwable object.

public String toString()

3 Returns the name of the class concatenated with the result of getMessage()

public void printStackTrace()

4 Prints the result of toString() along with the stack trace to System.err, the error output stream.

public StackTraceElement [] getStackTrace()

5 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.

public Throwable fillInStackTrace()

6 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 the 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 ExceptTest{

    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 the 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 the 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 categories 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.

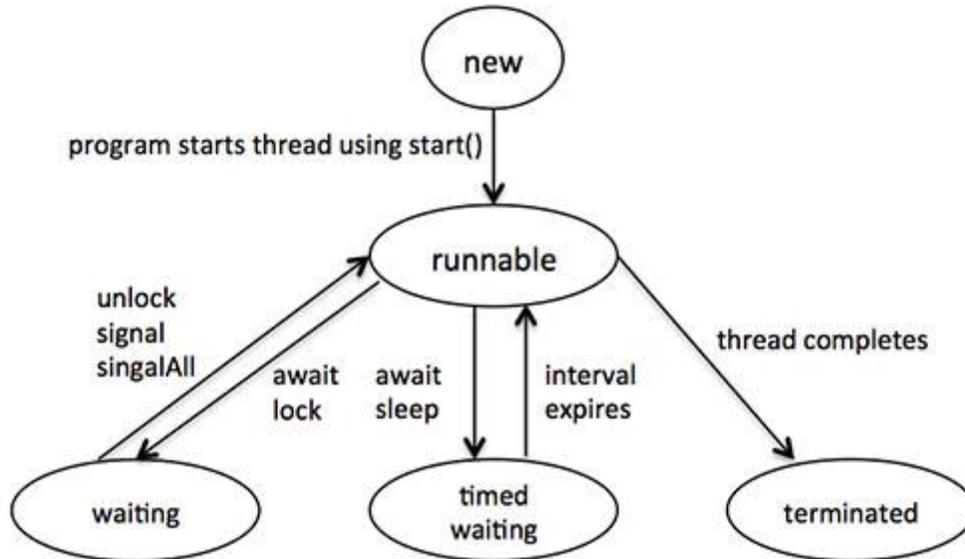
Java is *multithreaded programming language* which means we can develop multithreaded program using Java. A multithreaded program contains two or more parts that can run concurrently and each part can handle different task at the same time making optimal use of the available resources specially when your computer has multiple CPUs.

By definition multitasking is when multiple processes share common processing resources such as a CPU. Multithreading extends the idea of multitasking into applications where you can subdivide specific operations within a single application into individual threads. Each of the threads can run in parallel. The OS divides processing time not only among different applications, but also among each thread within an application.

Multithreading enables you to write in a way where multiple activities can proceed concurrently in the same program.

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 thread 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 dependent.

Create Thread by Implementing Runnable Interface:

If your class is intended to be executed as a thread then you can achieve this by implementing **Runnable** interface. You will need to follow three basic steps:

Step 1:

As a first step you need to implement a `run()` method provided by **Runnable** interface. This method provides entry point for the thread and you will put your complete business logic inside this method. Following is simple syntax of `run()` method:

```
public void run( )
```

Step 2:

At second step you will instantiate a **Thread** object using the following constructor:

```
Thread(Runnable threadObj, String threadName);
```

Where, *threadObj* is an instance of a class that implements the **Runnable** interface and **threadName** is the name given to the new thread.

Step 3

Once Thread object is created, you can start it by calling **start()** method, which executes a call to `run()` method. Following is simple syntax of `start()` method:

```
void start();
```

Example:

Here is an example that creates a new thread and starts it running:

```
class RunnableDemo implements Runnable {
    private Thread t;
    private String threadName;

    RunnableDemo( String name){
        threadName = name;
        System.out.println("Creating " + threadName );
    }
    public void run() {
```

```
System.out.println("Running " + threadName );
try {
    for(int i = 4; i > 0; i--) {
        System.out.println("Thread: " + threadName + ", " + i);
        // Let the thread sleep for a while.
        Thread.sleep(50);
    }
} catch (InterruptedException e) {
    System.out.println("Thread " + threadName + " interrupted.");
}
System.out.println("Thread " + threadName + " exiting.");
}

public void start ()
{
    System.out.println("Starting " + threadName );
    if (t == null)
    {
        t = new Thread (this, threadName);
        t.start ();
    }
}

}

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

        RunnableDemo R1 = new RunnableDemo( "Thread-1");
        R1.start();

        RunnableDemo R2 = new RunnableDemo( "Thread-2");
        R2.start();
    }
}
```

This would produce the following result:

```
Creating Thread-1
Starting Thread-1
Creating Thread-2
Starting Thread-2
Running Thread-1
Thread: Thread-1, 4
Running Thread-2
Thread: Thread-2, 4
```

Thread: Thread-1, 3
Thread: Thread-2, 3
Thread: Thread-1, 2
Thread: Thread-2, 2
Thread: Thread-1, 1
Thread: Thread-2, 1
Thread Thread-1 exiting.
Thread Thread-2 exiting.

Create Thread by Extending Thread Class:

The second way to create a thread is to create a new class that extends **Thread** class using the following two simple steps. This approach provides more flexibility in handling multiple threads created using available methods in Thread class.

Step 1

You will need to override **run()** method available in Thread class. This method provides entry point for the thread and you will put your complete business logic inside this method. Following is simple syntax of run() method:

```
public void run( )
```

Step 2

Once Thread object is created, you can start it by calling **start()** method, which executes a call to run() method. Following is simple syntax of start() method:

```
void start( );
```

Example:

Here is the preceding program rewritten to extend Thread:

```
class ThreadDemo extends Thread {  
    private Thread t;  
    private String threadName;  
  
    ThreadDemo( String name){  
        threadName = name;  
        System.out.println("Creating " + threadName );  
    }  
    public void run() {  
        System.out.println("Running " + threadName );  
        try {  
            for(int i = 4; i > 0; i--) {
```

```
        System.out.println("Thread: " + threadName + ", " + i);
        // Let the thread sleep for a while.
        Thread.sleep(50);
    }
} catch (InterruptedException e) {
    System.out.println("Thread " + threadName + " interrupted.");
}
System.out.println("Thread " + threadName + " exiting.");
}

public void start ()
{
    System.out.println("Starting " + threadName );
    if (t == null)
    {
        t = new Thread (this, threadName);
        t.start ();
    }
}

}

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

        ThreadDemo T1 = new ThreadDemo( "Thread-1");
        T1.start();

        ThreadDemo T2 = new ThreadDemo( "Thread-2");
        T2.start();
    }
}
```

This would produce the following result:

```
Creating Thread-1
Starting Thread-1
Creating Thread-2
Starting Thread-2
Running Thread-1
Thread: Thread-1, 4
Running Thread-2
Thread: Thread-2, 4
Thread: Thread-1, 3
Thread: Thread-2, 3
Thread: Thread-1, 2
```

Thread: Thread-2, 2
Thread: Thread-1, 1
Thread: Thread-2, 1
Thread Thread-1 exiting.
Thread Thread-2 exiting.

Thread Methods:

Following is the list of important methods 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


```
    while(true)
    {
        System.out.println(message);
    }
}
}
```

Following is another class which extends Thread class:

```
// File Name : GuessANumber.java
// Create a thread to extend 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.**");
    }
}
```

Following is the main program which makes use of above defined classes:

```
// 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(bye);
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 the 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
Goodbye
Goodbye
Goodbye
Goodbye
Goodbye
.....
```

IO Stream

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 provides strong but flexible support for I/O related to Files and networks but this tutorial covers very basic functionality related to streams and I/O. We would see most commonly used example one by one:

Byte Streams

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

```
import java.io.*;

public class CopyFile {
    public static void main(String args[]) throws IOException
    {
        FileInputStream in = null;
        FileOutputStream out = null;

        try {
            in = new FileInputStream("input.txt");
            out = new FileOutputStream("output.txt");

            int c;
            while ((c = in.read()) != -1) {
                out.write(c);
            }
        } finally {
            if (in != null) {
                in.close();
            }
            if (out != null) {
                out.close();
            }
        }
    }
}
```

Now let's have a file **input.txt** with the following content:

This is test for copy file.

As a next step, compile above program and execute it, which will result in creating output.txt file with the same content as we have in input.txt. So let's put above code in CopyFile.java file and do the following:

```
$javac CopyFile.java
$java CopyFile
```

Character Streams

Java **Byte** streams are used to perform input and output of 8-bit bytes, where as Java **Character** streams are used to perform input and output for 16-bit unicode. Though there are many classes related to character streams but the most frequently used classes are , **FileReader** and **FileWriter**.. Though internally FileReader uses FileInputStream and FileWriter uses FileOutputStream but here major difference is that FileReader reads two bytes at a time and FileWriter writes two bytes at a time.

We can re-write above example which makes use of these two classes to copy an input file (having unicode characters) into an output file:

```
import java.io.*;

public class CopyFile {
    public static void main(String args[]) throws IOException
    {
        FileReader in = null;
        FileWriter out = null;

        try {
            in = new FileReader("input.txt");
            out = new FileWriter("output.txt");

            int c;
            while ((c = in.read()) != -1) {
                out.write(c);
            }
        }finally {
            if (in != null) {
                in.close();
            }
            if (out != null) {
                out.close();
            }
        }
    }
}
```

```
    }  
  }  
}
```

Now let's have a file **input.txt** with the following content:

This is test for copy file.

As a next step, compile above program and execute it, which will result in creating output.txt file with the same content as we have in input.txt. So let's put above code in CopyFile.java file and do the following:

```
$javac CopyFile.java  
$java CopyFile
```

Standard Streams

All the programming languages provide support for standard I/O where user's program can take input from a keyboard and then produce output on the computer screen. If you are aware of C or C++ programming languages, then you must be aware of three standard devices STDIN, STDOUT and STDERR. Similar way Java provides following three standard streams

- **Standard Input:** This is used to feed the data to user's program and usually a keyboard is used as standard input stream and represented as **System.in**.
- **Standard Output:** This is used to output the data produced by the user's program and usually a computer screen is used to standard output stream and represented as **System.out**.
- **Standard Error:** This is used to output the error data produced by the user's program and usually a computer screen is used to standard error stream and represented as **System.err**.

Following is a simple program which creates **InputStreamReader** to read standard input stream until the user types a "q":

```
import java.io.*;  
  
public class ReadConsole {  
    public static void main(String args[]) throws IOException  
    {  
        InputStreamReader cin = null;  
  
        try {  
            cin = new InputStreamReader(System.in);  
            System.out.println("Enter characters, 'q' to quit.");  
            char c;  
            do {
```

```

        c = (char) cin.read();
        System.out.print(c);
    } while(c != 'q');
}finally {
    if (cin != null) {
        cin.close();
    }
}
}
}
}

```

Let's keep above code in ReadConsole.java file and try to compile and execute it as below. This program continues reading and outputting same character until we press 'q':

```

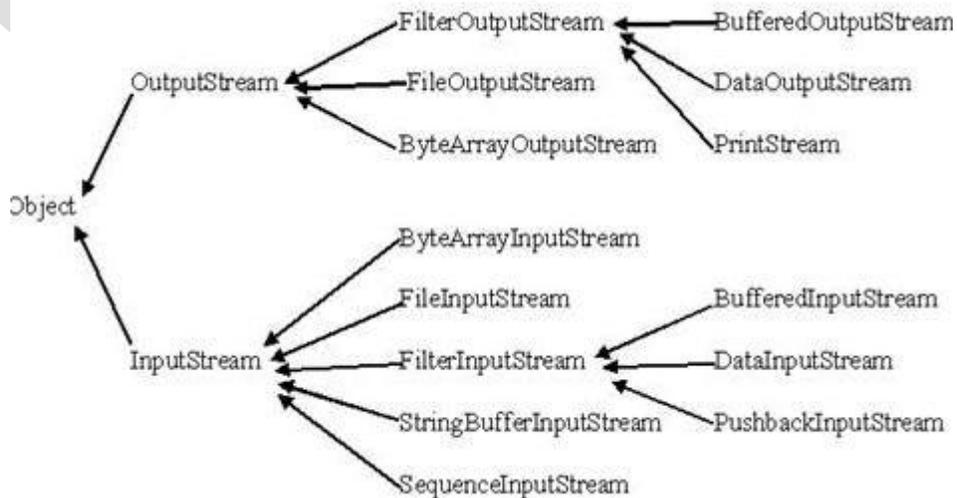
$javac ReadConsole.java
$java ReadConsole
Enter characters, 'q' to quit.
l
l
e
e
q
q

```

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

```
public void close() throws IOException{}
```

1 This method closes the file output stream. Releases any system resources associated with the file. Throws an `IOException`

```
protected void finalize()throws IOException {}
```

2 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`.

```
public int read(int r)throws IOException{}
```

3 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.

```
public int read(byte[] r) throws IOException{}
```

4 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.

```
public int available() throws IOException{}
```

5

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
----	--------------------------

	public void close() throws IOException{}
--	---

- | | |
|---|--|
| 1 | This method closes the file output stream. Releases any system resources associated with the file. Throws an IOException |
|---|--|

	protected void finalize()throws IOException {}
--	---

- | | |
|---|---|
| 2 | 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. |
|---|---|

	public void write(int w)throws IOException{}
--	---

- | | |
|---|--|
| 3 | This methods writes the specified byte to the output stream. |
|---|--|

	public void write(byte[] w)
--	------------------------------------

- | | |
|---|--|
| 4 | 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("test.txt");
            for(int x=0; x < bWrite.length ; x++){
                os.write( bWrite[x] ); // writes the bytes
            }
            os.close();

            InputStream is = new FileInputStream("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:

A directory is a File which can contains a list of other files and directories. You use **File** object to create directories, to list down files available in a directory. For complete detail check a list of all the methods which you can call on File object and what are related to directories.

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.

Listing Directories:

You can use **list()** method provided by **File** object to list down all the files and directories available in a directory as follows:

```
import java.io.File;

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

```
File file = null;
String[] paths;

try{
    // create new file object
    file = new File("/tmp");

    // array of files and directory
    paths = file.list();

    // for each name in the path array
    for(String path:paths)
    {
        // prints filename and directory name
        System.out.println(path);
    }
} catch(Exception e){
    // if any error occurs
    e.printStackTrace();
}
}
```

This would produce following result based on the directories and files available in your **/tmp** directory:

```
test1.txt
test2.txt
ReadDir.java
ReadDir.class
```

Applet

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`.

In order 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.javalicenses.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>
```