

## Introduction to Java

### History:

#### **Since 1995, Java has changed our world . . . and our expectations..**

Today, with technology such a part of our daily lives, we take it for granted that we can be connected and access applications and content anywhere, anytime. Because of Java, we expect digital devices to be smarter, more functional, and way more entertaining.

In the early 90s, extending the power of network computing to the activities of everyday life was a radical vision. In 1991, a small group of Sun engineers called the "Green Team" believed that the next wave in computing was the union of digital consumer devices and computers. Led by James Gosling, the team worked around the clock and created the programming language that would revolutionize our world Java.

The Green Team demonstrated their new language with an interactive, handheld home-entertainment controller that was originally targeted at the digital cable television industry. Unfortunately, the concept was much too advanced for the them at the time. But it was just right for the Internet, which was just starting to take off. In 1995, the team announced that the Netscape Navigator Internet browser would incorporate Java technology.

Today, Java not only permeates the Internet, but also is the invisible force behind many of the applications and devices that power our day-to-day lives. From mobile phones to handheld devices, games and navigation systems to e-business solutions, Java is everywhere!

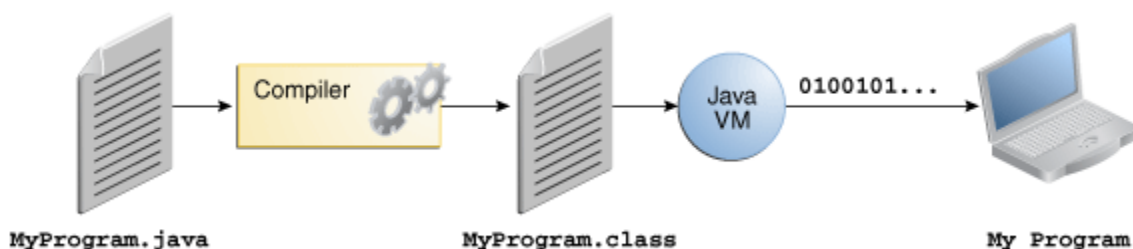
## INTRODUCTION TO JAVA APPLICATION

Java technology is both a programming language and a platform.

### The Java Programming Language

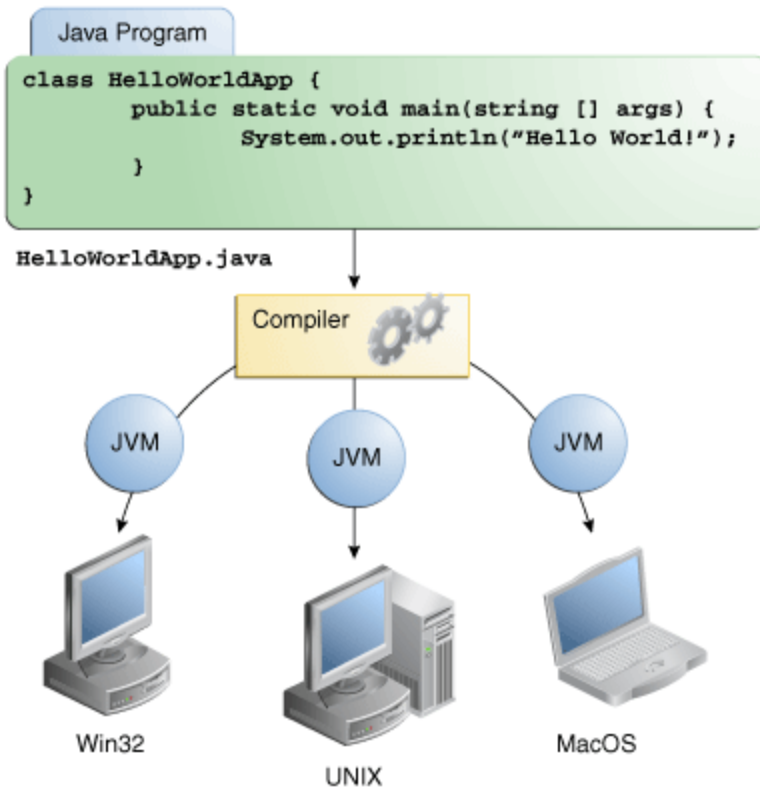
The Java programming language is a high-level language that can be characterized by all of the following buzzwords:

<ul style="list-style-type: none"><li>• Simple</li><li>• Object oriented</li><li>• Distributed</li><li>• Multithreaded</li><li>• Dynamic</li></ul>	<ul style="list-style-type: none"><li>• Architecture neutral</li><li>• Portable</li><li>• High performance</li><li>• Robust</li><li>• Secure</li></ul>
--	--



In the Java programming language, all source code is first written in plain text files ending with the `.java` extension. Those source files are then compiled into `.class` files by the `javac` compiler. A `.class` file does not contain code that is native to your processor; it instead contains bytecodes the machine language of the Java Virtual Machine (Java VM). The `java` launcher tool then runs your application with an instance of the Java Virtual Machine.

Because the Java VM is available on many different operating systems, the same `.class` files are capable of running on Microsoft Windows, the Solaris™ Operating System (Solaris OS), Linux, or Mac OS. Some virtual machines, such as the Java SE HotSpot at a Glance, perform additional steps at runtime to give your application a performance boost. This includes various tasks such as finding performance bottlenecks and recompiling (to native code) frequently used sections of code.



Through the Java VM, the same application is capable of running on multiple platforms.

### The Java Platform

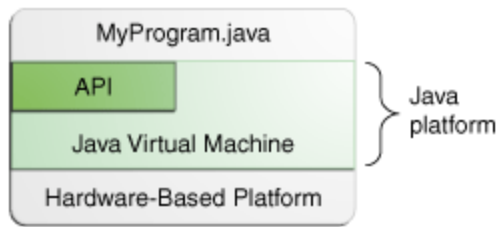
A platform is the hardware or software environment in which a program runs. We've already mentioned some of the most popular platforms like Microsoft Windows, Linux, Solaris OS, and Mac OS. Most platforms can be described as a combination of the operating system and underlying hardware. The Java platform differs from most other platforms in that it's a software-only platform that runs on top of other hardware-based platforms.

#### The Java platform has two components:

- The Java Virtual Machine
- The Java Application Programming Interface (API)

You've already been introduced to the Java Virtual Machine; it's the base for the Java platform and is ported onto various hardware-based platforms.

The API is a large collection of ready-made software components that provide many useful capabilities. It is grouped into libraries of related classes and interfaces; these libraries are known as packages. The next section, What Can Java Technology Do? highlights some of the functionality provided by the API.



The API and Java Virtual Machine insulate the program from the underlying hardware.

As a platform-independent environment, the Java platform can be a bit slower than native code. However, advances in compiler and virtual machine technologies are bringing performance close to that of native code without threatening portability.

The terms "Java Virtual Machine" and "JVM" mean a Virtual Machine for the Java platform.

## INTRODUCTION TO CLASSES, OBJECTS AND PACKAGES

With the knowledge you now have of the basics of the Java programming language, you can learn to write your own classes. In this lesson, you will find information about defining your own classes, including declaring member variables, methods, and constructors.

You will learn to use your classes to create objects, and how to use the objects you create.

Classes:

### What is Class :

A class is nothing but a blueprint or a template for creating different objects which defines its properties and behaviors. Java class objects exhibit the properties and behaviors defined by its class. A class can contain fields and methods to describe the behavior of an **object**.

### **Declaring Classes :**

You've seen classes defined in the following way:

```
class < class Name > {  
  
    // field, constructor, and  
  
    // method declarations  
  
}
```

This is a *class declaration*. The class body (the area between the braces) contains all the code that provides for the life cycle of the objects created from the class: constructors for initializing new objects, declarations for the fields that provide the state of the class and its objects, and methods to implement the behavior of the class and its objects.

The preceding class declaration is a minimal one. It contains only those components of a class declaration that are required. You can provide more information about the class, such as the name of its superclass, whether it implements any interfaces, and so on, at the start of the class declaration. For example,

```
class MyClass extends MySuperClass implements YourInterface {  
  
    // field, constructor, and
```

```
// method declarations
```

```
}
```

means that `MyClass` is a subclass of `MySuperClass` and that it implements the `YourInterface` interface.

You can also add modifiers like *public* or *private* at the very beginning—so you can see that the opening line of a class declaration can become quite complicated. The modifiers *public* and *private*, which determine what other classes can access `MyClass`, are discussed later in this lesson. The lesson on interfaces and inheritance will explain how and why you would use the *extends* and *implements* keywords in a class declaration. For the moment you do not need to worry about these extra complications.

In general, class declarations can include these components, in order:

1. Modifiers such as *public*, *private*, and a number of others that you will encounter later.
2. The class name, with the initial letter capitalized by convention.
3. The name of the class's parent (superclass), if any, preceded by the keyword *extends*. A class can only *extend* (subclass) one parent.
4. A comma-separated list of interfaces implemented by the class, if any, preceded by the keyword *implements*. A class can *implement* more than one interface.
5. The class body, surrounded by braces, `{}`.

This section shows you the anatomy of a class, and how to declare fields, methods, and constructors.

The introduction to object-oriented concepts in the lesson titled `Object-oriented Programming Concepts` used a bicycle class as an example, with racing bikes, mountain bikes, and tandem bikes as subclasses. Here is sample code for a possible implementation of a `Bicycle` class, to give you an overview of a class declaration. Subsequent sections of this lesson will back up and explain class declarations step by step. For the moment, don't concern yourself with the details.

```
public class Bicycle {
```

```
    // the Bicycle class has
```

```
    // three fields
```

```
    public int cadence;
```

```
    public int gear;
```

```
    public int speed;
```

**// the Bicycle class has**

**// one *constructor***

```
public Bicycle(int startCadence, int startSpeed, int startGear) {  
    gear = startGear;  
    cadence = startCadence;  
    speed = startSpeed;  
}
```

**// the Bicycle class has**

**// four *methods***

```
public void setCadence(int newValue) {  
    cadence = newValue;  
}
```

```
public void setGear(int newValue) {  
    gear = newValue;  
}
```

```
public void applyBrake(int decrement) {  
    speed -= decrement;  
}
```

```
public void speedUp(int increment) {  
    speed += increment;  
}
```

```
}
```

A class declaration for a MountainBike class that is a subclass of Bicycle might look like this:

```
public class MountainBike extends Bicycle {
```

```
// the MountainBike subclass has
```

```
// one field
```

```
public int seatHeight;
```

```
// the MountainBike subclass has
```

```
// one constructor
```

```
public MountainBike(int startHeight, int startCadence,  
                    int startSpeed, int startGear) {  
    super(startCadence, startSpeed, startGear);  
    seatHeight = startHeight;  
}
```

```
// the MountainBike subclass has
```

```
// one method
```

```
public void setHeight(int newValue) {  
    seatHeight = newValue;  
}
```

```
}
```

MountainBike inherits all the fields and methods of Bicycle and adds the field `seatHeight` and a method to set it (mountain bikes have seats that can be moved up and down as the terrain demands).

## Declaring Member Variables :

There are several kinds of variables:

- Member variables in a class—these are called *fields*.
- Variables in a method or block of code—these are called *local variables*.
- Variables in method declarations—these are called *parameters*.

The Bicycle class uses the following lines of code to define its fields:



```
public int cadence;  
  
public int gear;  
  
public int speed;
```

Field declarations are composed of three components, in order:

1. Zero or more modifiers, such as public or private.
2. The field's type.
3. The field's name.

The fields of Bicycle are named cadence, gear, and speed and are all of data type integer (int). The public keyword identifies these fields as public members, accessible by any object that can access the class.

### Access Modifiers

The first (left-most) modifier used lets you control what other classes have access to a member field. For the moment, consider only public and private. Other access modifiers will be discussed later.

- public modifier—the field is accessible from all classes.
- private modifier—the field is accessible only within its own class.

In the spirit of encapsulation, it is common to make fields private. This means that they can only be *directly* accessed from the Bicycle class. We still need access to these values, however. This can be done *indirectly* by adding public methods that obtain the field values for us:

```
public class Bicycle {  
  
    private int cadence;  
  
    private int gear;  
  
    private int speed;  
  
    public Bicycle(int startCadence, int startSpeed, int startGear) {  
        gear = startGear;  
        cadence = startCadence;  
        speed = startSpeed;  
    }  
}
```

```
public int getCadence() {  
    return cadence;  
}
```

```
public void setCadence(int newValue) {  
    cadence = newValue;  
}
```

```
public int getGear() {  
    return gear;  
}
```

```
public void setGear(int newValue) {  
    gear = newValue;  
}
```

```
public int getSpeed() {  
    return speed;  
}
```

```
public void applyBrake(int decrement) {  
    speed -= decrement;  
}
```

```
public void speedUp(int increment) {  
    speed += increment;  
}  
}
```

## Defining Methods:

In java, a method is like function i.e. used to expose behaviour of an object.

### Here is an example of a typical method declaration:

```
public double calculateAnswer(double wingSpan, int numberOfEngines,  
                               double length, double grossTons) {  
    //do the calculation here  
}
```

The only required elements of a method declaration are the method's return type, name, a pair of parentheses, (), and a body between braces, {}.

More generally, method declarations have six components, in order:

1. Modifiers—such as public, private, and others you will learn about later.
2. The return type—the data type of the value returned by the method, or void if the method does not return a value.
3. The method name—the rules for field names apply to method names as well, but the convention is a little different.
4. The parameter list in parenthesis—a comma-delimited list of input parameters, preceded by their data types, enclosed by parentheses, (). If there are no parameters, you must use empty parentheses.
5. An exception list—to be discussed later.
6. The method body, enclosed between braces—the method's code, including the declaration of local variables, goes here.

## Naming a Method

Although a method name can be any legal identifier, code conventions restrict method names. By convention, method names should be a verb in lowercase or a multi-word name that begins with a verb in lowercase, followed by adjectives, nouns, etc. In multi-word names, the first letter of each of the second and following words should be capitalized. Here are some examples:

run

runFast

getBackground

getFinalData

compareTo

setX

isEmpty

Typically, a method has a unique name within its class. However, a method might have the same name as other methods due to *method overloading*.

## Overloading Methods

The Java programming language supports *overloading* methods, and Java can distinguish between methods with different *method signatures*. This means that methods within a class can have the same name if they have different parameter lists (there are some qualifications to this that will be discussed in the lesson titled "Interfaces and Inheritance").

Suppose that you have a class that can use calligraphy to draw various types of data (strings, integers, and so on) and that contains a method for drawing each data type. It is cumbersome to use a new name for each method—for example, drawString, drawInteger, drawFloat, and so on. In the Java programming language, you can use the same name for all the drawing methods but pass a different argument list to each method. Thus, the data drawing class might declare four methods named draw, each of which has a different parameter list.

```
public class DataArtist {  
    ...  
    public void draw(String s) {  
        ...  
    }  
    public void draw(int i) {  
        ...  
    }  
    public void draw(double f) {  
        ...  
    }  
    public void draw(int i, double f) {  
        ...  
    }  
}
```

Overloaded methods are differentiated by the number and the type of the arguments passed into the method. In the code sample, `draw(String s)` and `draw(int i)` are distinct and unique methods because they require different argument types.

You cannot declare more than one method with the same name and the same number and type of arguments, because the compiler cannot tell them apart.

The compiler does not consider return type when differentiating methods, so you cannot declare two methods with the same signature even if they have a different return type.

**Note:** Overloaded methods should be used sparingly, as they can make code much less readable.

### **Advantage of Method**

- Code Reusability
- Code Optimization

### **Objects:**

**Objects are key to understanding *object-oriented* technology. Look around right now and you'll find many examples of real-world objects: your dog, your desk, your television set, your bicycle.**

Real-world objects share two characteristics: They all have *state* and *behavior*. Dogs have state (name, color, breed, hungry) and behavior (barking, fetching, wagging tail). Bicycles also have state (current gear, current pedal cadence, current speed) and behavior (changing gear, changing pedal cadence, applying brakes). Identifying the state and behavior for real-world objects is a great way to begin thinking in terms of object-oriented programming.

A typical Java program creates many objects, which as you know, interact by invoking methods. Through these object interactions, a program can carry out various tasks, such as implementing a GUI, running an animation, or sending and receiving information over a network. Once an object has completed the work for which it was created, its resources are recycled for use by other objects.

Here's a small program, called `CreateObjectDemo`, that creates three objects: one `Point` object and two `Rectangle` objects. You will need all three source files to compile this program.

```
public class CreateObjectDemo {
```

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

```
        // Declare and create a point object and two rectangle objects.
```

```

Point originOne = new Point(23, 94);
Rectangle rectOne = new Rectangle(originOne, 100, 200);
Rectangle rectTwo = new Rectangle(50, 100);

// display rectOne's width, height, and area
System.out.println("Width of rectOne: " + rectOne.width);
System.out.println("Height of rectOne: " + rectOne.height);
System.out.println("Area of rectOne: " + rectOne.getArea());

// set rectTwo's position
rectTwo.origin = originOne;

// display rectTwo's position
System.out.println("X Position of rectTwo: " + rectTwo.origin.x);
System.out.println("Y Position of rectTwo: " + rectTwo.origin.y);

// move rectTwo and display its new position
rectTwo.move(40, 72);
System.out.println("X Position of rectTwo: " + rectTwo.origin.x);
System.out.println("Y Position of rectTwo: " + rectTwo.origin.y);
}
}

```

This program creates, manipulates, and displays information about various objects. Here's the output:

Width of rectOne: 100

Height of rectOne: 200

Area of rectOne: 20000

X Position of rectTwo: 23

Y Position of rectTwo: 94

X Position of rectTwo: 40

Y Position of rectTwo: 72

### **Packages In Java :**

A package is a namespace that organizes a set of related classes and interfaces. Conceptually you can think of packages as being similar to different folders on your computer. You might keep HTML pages in one folder, images in another, and scripts or applications in yet another. Because software written in the Java programming language can be composed of hundreds *orthousands* of individual classes, it makes sense to keep things organized by placing related classes and interfaces into packages.

Package in java can be categorized in two form, built-in package and user-defined package.

There are many built-in packages such as java, lang, awt, javax, swing, net, io, util, sql etc.

#### **Ex:**

```
package com.ics.demo;  
public class SampleDemo {  
    // .....  
    // some code  
    //.....  
  
}
```

### **How to compile a Java File :**

#### **Syntax:**

**javac -d directory javafileName**

#### **Ex:**

**javac -d . SampleDemo.java**

The -d switch specifies the destination where to put the generated class file. You can use any directory name like /home (in case of Linux), d:/abc (in case of windows) etc. If you want to keep the package within the same directory, you can use . (dot).

### **How to Run Java Package Program :**

#### **Syntax:**

**java** <fully qualified Name>

#### **Ex:**

**To Run: java com.ics.demo.SampleDemo**



## CONTROL STATEMENTS

The statements inside your source files are generally executed from top to bottom, in the order that they appear. *Control flow statements*, however, break up the flow of execution by employing decision making, looping, and branching, enabling your program to *conditionally* execute particular blocks of code. This section describes the decision-making statements (if-then, if-then-else, switch), the looping statements (for, while, do-while), and the branching statements (break, continue, return) supported by the Java programming language.

The if-then statement is the most basic of all the control flow statements. It tells your program to execute a certain section of code *only if* a particular test evaluates to true. For example, the Bicycle class could allow the brakes to decrease the bicycle's speed *only if* the bicycle is already in motion. One possible implementation of the `applyBrakes` method could be as follows:

```
void applyBrakes() {  
    // the "if" clause: bicycle must be moving  
    if (isMoving){  
        // the "then" clause: decrease current speed  
        currentSpeed--;  
    }  
}
```

If this test evaluates to false (meaning that the bicycle is not in motion), control jumps to the end of the if-then statement.

In addition, the opening and closing braces are optional, provided that the "then" clause contains only one statement:

```
void applyBrakes() {  
    // same as above, but without braces  
    if (isMoving)  
        currentSpeed--;  
}
```

Deciding when to omit the braces is a matter of personal taste. Omitting them can make the code more brittle. If a second statement is later added to the "then" clause, a common mistake

would be forgetting to add the newly required braces. The compiler cannot catch this sort of error; you'll just get the wrong results.

### The if-then-else Statement

The if-then-else statement provides a secondary path of execution when an "if" clause evaluates to false. You could use an if-then-else statement in the `applyBrakes` method to take some action if the brakes are applied when the bicycle is not in motion. In this case, the action is to simply print an error message stating that the bicycle has already stopped.

```
void applyBrakes() {  
    if (isMoving) {  
        currentSpeed--;  
    } else {  
        System.err.println("The bicycle has already stopped!");  
    }  
}
```

The following program, `IfElseDemo`, assigns a grade based on the value of a test score: an A for a score of 90% or above, a B for a score of 80% or above, and so on.

```
class IfElseDemo {  
    public static void main(String[] args) {  
  
        int testscore = 76;  
        char grade;  
  
        if (testscore >= 90) {  
            grade = 'A';  
        } else if (testscore >= 80) {  
            grade = 'B';  
        } else if (testscore >= 70) {  
            grade = 'C';  
        }  
    }  
}
```

```

    } else if (testscore >= 60) {
        grade = 'D';
    } else {
        grade = 'F';
    }
    System.out.println("Grade = " + grade);
}
}

```

The output from the program is:

```
Grade = C
```

You may have noticed that the value of testscore can satisfy more than one expression in the compound statement: `76 >= 70` and `76 >= 60`. However, once a condition is satisfied, the appropriate statements are executed (`grade = 'C';`) and the remaining conditions are not evaluated.

### The switch Statement :

Unlike if-then and if-then-else statements, the switch statement can have a number of possible execution paths. A switch works with the byte, short, char, and int primitive data types. It also works with *enumerated types* (discussed in Enum Types), the String class, and a few special classes that wrap certain primitive types: Character, Byte, Short, and Integer (discussed in Numbers and Strings).

The following code example, SwitchDemo, declares an int named month whose value represents a month. The code displays the name of the month, based on the value of month, using the switch statement.

```

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

```

```
int month = 8;
String monthString;
switch (month) {
    case 1: monthString = "January";
        break;
    case 2: monthString = "February";
        break;
    case 3: monthString = "March";
        break;
    case 4: monthString = "April";
        break;
    case 5: monthString = "May";
        break;
    case 6: monthString = "June";
        break;
    case 7: monthString = "July";
        break;
    case 8: monthString = "August";
        break;
    case 9: monthString = "September";
        break;
    case 10: monthString = "October";
        break;
    case 11: monthString = "November";
        break;
    case 12: monthString = "December";
        break;
    default: monthString = "Invalid month";
```

```

        break;
    }
    System.out.println(monthString);
}
}

```

In this case, August is printed to standard output.

The body of a switch statement is known as a *switch block*. A statement in the switch block can be labeled with one or more case or default labels. The switch statement evaluates its expression, then executes all statements that follow the matching case label.

You could also display the name of the month with if-then-else statements:

```

int month = 8;
if (month == 1) {
    System.out.println("January");
} else if (month == 2) {
    System.out.println("February");
}
... // and so on

```

Deciding whether to use if-then-else statements or a switch statement is based on readability and the expression that the statement is testing. An if-then-else statement can test expressions based on ranges of values or conditions, whereas a switch statement tests expressions based only on a single integer, enumerated value, or String object.

### **The while and do-while Statements**

The while statement continually executes a block of statements while a particular condition is true. Its syntax can be expressed as:

```

while (expression) {
    statement(s)
}

```

The while statement evaluates *expression*, which must return a boolean value. If the expression evaluates to true, the while statement executes the statement(s) in the while block. The while statement continues testing the expression and executing its block until the expression evaluates to false. Using the while statement to print the values from 1 through 10 can be accomplished as in the following WhileDemo program:

```
class WhileDemo {  
    public static void main(String[] args){  
        int count = 1;  
        while (count < 11) {  
            System.out.println("Count is: " + count);  
            count++;  
        }  
    }  
}
```

You can implement an infinite loop using the while statement as follows:

```
while (true){  
    // your code goes here  
}
```

The Java programming language also provides a do-while statement, which can be expressed as follows:

```
do {  
    statement(s)  
} while (expression);
```

The difference between do-while and while is that do-while evaluates its expression at the bottom of the loop instead of the top. Therefore, the statements within the do block are always executed at least once, as shown in the following DoWhileDemo program:

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

```

int count = 1;

do {
    System.out.println("Count is: " + count);
    count++;
} while (count < 11);
}
}

```

## The for Statement

The for statement provides a compact way to iterate over a range of values. Programmers often refer to it as the "for loop" because of the way in which it repeatedly loops until a particular condition is satisfied. The general form of the for statement can be expressed as follows:

```

for (initialization; termination;
    increment) {
    statement(s)
}

```

When using this version of the for statement, keep in mind that:

- The initialization expression initializes the loop; it's executed once, as the loop begins.
- When the termination expression evaluates to false, the loop terminates.
- The increment expression is invoked after each iteration through the loop; it is perfectly acceptable for this expression to increment *or* decrement a value.

The following program, ForDemo, uses the general form of the for statement to print the numbers 1 through 10 to standard output:

```

class ForDemo {
    public static void main(String[] args){
        for(int i=1; i<11; i++){
            System.out.println("Count is: " + i);
        }
    }
}

```

```
}  
}
```

The output of this program is:

```
Count is: 1  
Count is: 2  
Count is: 3  
Count is: 4  
Count is: 5  
Count is: 6  
Count is: 7  
Count is: 8  
Count is: 9  
Count is: 10
```

Notice how the code declares a variable within the initialization expression. The scope of this variable extends from its declaration to the end of the block governed by the for statement, so it can be used in the termination and increment expressions as well. If the variable that controls a for statement is not needed outside of the loop, it's best to declare the variable in the initialization expression. The names *i*, *j*, and *k* are often used to control for loops; declaring them within the initialization expression limits their life span and reduces errors.

The three expressions of the for loop are optional; an infinite loop can be created as follows:

```
// infinite loop  
for ( ; ; ) {  
    // your code goes here  
}
```



The for statement also has another form designed for iteration through Collections and arrays. This form is sometimes referred to as the *enhanced for* statement, and can be used to make your loops more compact and easy to read. To demonstrate, consider the following array, which holds the numbers 1 through 10:

```
int[] numbers = {1,2,3,4,5,6,7,8,9,10};
```

The following program, EnhancedForDemo, uses the enhanced for to loop through the array:

```
class EnhancedForDemo {  
    public static void main(String[] args){  
        int[] numbers =  
            {1,2,3,4,5,6,7,8,9,10};  
        for (int item : numbers) {  
            System.out.println("Count is: " + item);  
        }  
    }  
}
```

In this example, the variable `item` holds the current value from the `numbers` array. The output from this program is the same as before:

Count is: 1

Count is: 2

Count is: 3

Count is: 4

Count is: 5

Count is: 6

Count is: 7

Count is: 8

Count is: 9

Count is: 10

We recommend using this form of the for statement instead of the general form whenever possible.

## Branching Statements :

### The break Statement

The break statement has two forms: labeled and unlabeled. You saw the unlabeled form in the previous discussion of the switch statement. You can also use an unlabeled break to terminate a for, while, or do-while loop, as shown in the following [BreakDemo](#) program:

```
class BreakDemo {  
    public static void main(String[] args) {  
  
        int[] arrayOfInts =  
            { 32, 87, 3, 589,  
              12, 1076, 2000,  
              8, 622, 127 };  
        int searchfor = 12;  
  
        int i;  
        boolean foundIt = false;  
  
        for (i = 0; i < arrayOfInts.length; i++) {  
            if (arrayOfInts[i] == searchfor) {  
                foundIt = true;  
                break;  
            }  
        }  
    }  
}
```

```

        if (foundIt) {
            System.out.println("Found " + searchfor + " at index " + i);
        } else {
            System.out.println(searchfor + " not in the array");
        }
    }
}

```

This program searches for the number 12 in an array. The **break** statement, shown in boldface, terminates the for loop when that value is found. Control flow then transfers to the statement after the for loop. This program's output is:

```
Found 12 at index 4
```

An unlabeled **break** statement terminates the innermost switch, for, while, or do-while statement, but a labeled **break** terminates an outer statement. The following program, [BreakWithLabelDemo](#), is similar to the previous program, but uses nested for loops to search for a value in a two-dimensional array. When the value is found, a labeled **break** terminates the outer for loop (labeled "search"):

```

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

        int[][] arrayOfInts = {
            { 32, 87, 3, 589 },
            { 12, 1076, 2000, 8 },
            { 622, 127, 77, 955 }
        };

        int searchfor = 12;

        int i;
        int j = 0;
        boolean foundIt = false;

        search:

```

```

for (i = 0; i < arrayOfInts.length; i++) {
    for (j = 0; j < arrayOfInts[i].length;
        j++) {
        if (arrayOfInts[i][j] == searchfor) {
            foundIt = true;
            break search;
        }
    }
}

if (foundIt) {
    System.out.println("Found " + searchfor + " at " + i + ", " + j);
} else {
    System.out.println(searchfor + " not in the array");
}
}
}

```

This is the output of the program.

Found 12 at 1, 0

The break statement terminates the labeled statement; it does not transfer the flow of control to the label. Control flow is transferred to the statement immediately following the labeled (terminated) statement.

### The continue Statement

The continue statement skips the current iteration of a for, while, or do-while loop. The unlabeled form skips to the end of the innermost loop's body and evaluates the boolean expression that controls the loop. The following program, [ContinueDemo](#), steps through a String, counting the occurrences of the letter "p". If the current character is not a p, the continue statement skips the rest of the loop and proceeds to the next character. If it is a "p", the program increments the letter count.

```

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

```

```

String searchMe = "peter piper picked a " + "peck of pickled peppers";
int max = searchMe.length();
int numPs = 0;

for (int i = 0; i < max; i++) {
    // interested only in p's
    if (searchMe.charAt(i) != 'p')
        continue;

    // process p's
    numPs++;
}

System.out.println("Found " + numPs + " p's in the string.");
}
}

```

Here is the output of this program:

Found 9 p's in the string.

To see this effect more clearly, try removing the continue statement and recompiling. When you run the program again, the count will be wrong, saying that it found 35 p's instead of 9.

continue statement skips the current iteration of an outer loop marked with the given label. The following example program, [ContinueWithLabelDemo](#), uses nested loops to search for a substring within another string. Two nested loops are required: one to iterate over the substring and one to iterate over the string being searched. The following program, [ContinueWithLabelDemo](#), uses the labeled form of continue to skip an iteration in the outer loop.

```

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

        String searchMe = "Look for a substring in me";
        String substring = "sub";
    }
}

```

```
boolean foundIt = false;
```

```
int max = searchMe.length() -  
    substring.length();
```

test:

```
for (int i = 0; i <= max; i++) {  
    int n = substring.length();  
    int j = i;  
    int k = 0;  
    while (n-- != 0) {  
        if (searchMe.charAt(j++) != substring.charAt(k++)) {  
            continue test;  
        }  
    }  
    foundIt = true;  
    break test;  
}  
System.out.println(foundIt ? "Found it" : "Didn't find it");  
}  
}
```

Here is the output from this program.

it

## The return Statem

### ent

The last of the branching statements is the return statement. The return statement exits from the current method, and control flow returns to where the method was invoked. The returnstatement has two forms: one that returns a value, and one that doesn't. To return a value, simply put the value (or an expression that calculates the value) after the return keyword.

```
return ++count;
```

The data type of the returned value must match the type of the method's declared return value. When a method is declared void, use the form of return that doesn't return a value.

## **Summary of Control Flow Statements**

The if-then statement is the most basic of all the control flow statements. It tells your program to execute a certain section of code *only if* a particular test evaluates to true. The if-then-else statement provides a secondary path of execution when an "if" clause evaluates to false. Unlike if-then and if-then-else, the switch statement allows for any number of possible execution paths. The while and do-while statements continually execute a block of statements while a particular condition is true. The difference between do-while and while is that do-while evaluates its expression at the bottom of the loop instead of the top. Therefore, the statements within the do block are always executed at least once. The for statement provides a compact way to iterate over a range of values. It has two forms, one of which was designed for looping through collections and arrays.

## ARRAYS

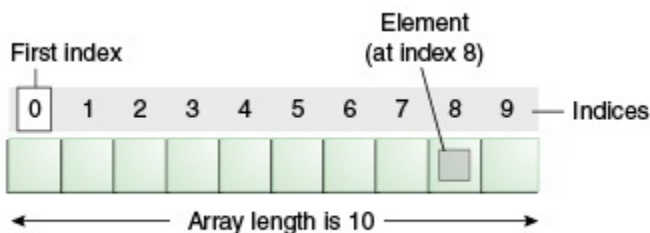
An *array* is a container object that holds a fixed number of values of a single type. The length of an array is established when the array is created. After creation, its length is fixed.

Array in java is index based, first element of the array is stored at 0 index .

### Advantage of Java Array:

**Code Optimization:** It makes the code optimized, we can retrieve or sort the data easily.

**Random access:** We can get any data located at any index position.



An array of 10 elements.

Each item in an array is called an *element*, and each element is accessed by its numerical *index*. As shown in the preceding illustration, numbering begins with 0. The 9th element, for example, would therefore be accessed at index 8.

The following program, [ArrayDemo](#), creates an array of integers, puts some values in the array, and prints each value to standard output.

```
class ArrayDemo {  
    public static void main(String[] args) {  
        // declares an array of integers  
        int[] anArray;  
  
        // allocates memory for 10 integers  
        anArray = new int[10];  
    }  
}
```



```
// initialize first element
anArray[0] = 100;
// initialize second element
anArray[1] = 200;
// and so forth
anArray[2] = 300;
anArray[3] = 400;
anArray[4] = 500;
anArray[5] = 600;
anArray[6] = 700;
anArray[7] = 800;
anArray[8] = 900;
anArray[9] = 1000;
```

```
System.out.println("Element at index 0: "
    + anArray[0]);
System.out.println("Element at index 1: "
    + anArray[1]);
System.out.println("Element at index 2: "
    + anArray[2]);
System.out.println("Element at index 3: "
    + anArray[3]);
System.out.println("Element at index 4: "
    + anArray[4]);
System.out.println("Element at index 5: "
    + anArray[5]);
System.out.println("Element at index 6: "
    + anArray[6]);
```

```
        System.out.println("Element at index 7: "
            + anArray[7]);
        System.out.println("Element at index 8: "
            + anArray[8]);
        System.out.println("Element at index 9: "
            + anArray[9]);
    }
}
```

The output from this program is:

```
Element at index 0: 100
Element at index 1: 200
Element at index 2: 300
Element at index 3: 400
Element at index 4: 500
Element at index 5: 600
Element at index 6: 700
Element at index 7: 800
Element at index 8: 900
Element at index 9: 1000
```

In a real-world programming situation, you would probably use one of the supported *looping constructs* to iterate through each element of the array, rather than write each line individually as in the preceding example. However, the example clearly illustrates the array syntax.

### **Declaring a Variable to Refer to an Array :**

The preceding program declares an array (named anArray) with the following line of code:

```
// declares an array of integers
int[] anArray;
```

Like declarations for variables of other types, an array declaration has two components: the array's type and the array's name. An array's type is written as `type[]`, where `type` is the data type of the contained elements; the brackets are special symbols indicating that this variable holds an array. The size of the array is not part of its type (which is why the brackets are empty). An array's name can be anything you want, provided that it follows the rules and conventions as previously discussed in the [naming](#) section. As with variables of other types, the declaration does not actually create an array; it simply tells the compiler that this variable will hold an array of the specified type.

Similarly, you can declare arrays of other types:

```
byte[] anArrayOfBytes;  
short[] anArrayOfShorts;  
long[] anArrayOfLongs;  
float[] anArrayOfFloats;  
double[] anArrayOfDoubles;  
boolean[] anArrayOfBooleans;  
char[] anArrayOfChars;  
String[] anArrayOfStrings;
```

You can also place the brackets after the array's name:

```
// this form is discouraged  
float anArrayOfFloats[];
```

However, convention discourages this form; the brackets identify the array type and should appear with the type designation.

### **Creating, Initializing, and Accessing an Array :**

One way to create an array is with the `new` operator. The next statement in the `ArrayDemo` program allocates an array with enough memory for 10 integer elements and assigns the array to the `anArray` variable.

```
// create an array of integers  
anArray = new int[10];
```

If this statement is missing, then the compiler prints an error like the following, and compilation fails:

ArrayDemo.java:4: Variable anArray may not have been initialized.

The next few lines assign values to each element of the array:

```
anArray[0] = 100; // initialize first element
```

```
anArray[1] = 200; // initialize second element
```

```
anArray[2] = 300; // and so forth
```

Each array element is accessed by its numerical index:

```
System.out.println("Element 1 at index 0: " + anArray[0]);
```

```
System.out.println("Element 2 at index 1: " + anArray[1]);
```

```
System.out.println("Element 3 at index 2: " + anArray[2]);
```

Alternatively, you can use the shortcut syntax to create and initialize an array:

```
int[] anArray = {  
    100, 200, 300,  
    400, 500, 600,  
    700, 800, 900, 1000  
};
```

Here the length of the array is determined by the number of values provided between braces and separated by commas.

You can also declare an array of arrays (also known as a multidimensional array) by using two or more sets of brackets, such as `String[][] names`. Each element, therefore, must be accessed by a corresponding number of index values.

In the Java programming language, a multidimensional array is an array whose components are themselves arrays. This is unlike arrays in C or Fortran. A consequence of this is that the rows are allowed to vary in length, as shown in the following [MultiDimArrayDemo](#) program:

```
class MultiDimArrayDemo {  
    public static void main(String[] args) {  
        String[][] names = {  
            {"Mr. ", "Mrs. ", "Ms. "},  
            {"Smith", "Jones"}  
        };  
        // Mr. Smith
```

```

        System.out.println(names[0][0] + names[1][0]);

        // Ms. Jones

        System.out.println(names[0][2] + names[1][1]);
    }
}

```

The output from this program is:

Mr. Smith

Ms. Jones

Finally, you can use the built-in length property to determine the size of any array. The following code prints the array's size to standard output:

```
System.out.println(anArray.length);
```

### Copying Arrays :

The System class has an arraycopy method that you can use to efficiently copy data from one array into another:

```
public static void arraycopy(Object src, int srcPos,
                             Object dest, int destPos, int length)
```

The two Object arguments specify the array to copy *from* and the array to copy *to*. The three int arguments specify the starting position in the source array, the starting position in the destination array, and the number of array elements to copy.

The following program, [ArrayCopyDemo](#), declares an array of char elements, spelling the word "decaffeinated." It uses the System.arraycopy method to copy a subsequence of array components into a second array:

```
class ArrayCopyDemo {
    public static void main(String[] args) {
        char[] copyFrom = { 'd', 'e', 'c', 'a', 'f', 'f', 'e',
                             'i', 'n', 'a', 't', 'e', 'd' };
        char[] copyTo = new char[7];
    }
}

```

```

        System.arraycopy(copyFrom, 2, copyTo, 0, 7);
        System.out.println(new String(copyTo));
    }
}

```

The output from this program is:

```
cafein
```

### Array Manipulations :

Arrays are a powerful and useful concept used in programming. Java SE provides methods to perform some of the most common manipulations related to arrays. For instance, the [ArrayCopyDemo](#) example uses the `arraycopy` method of the `System` class instead of manually iterating through the elements of the source array and placing each one into the destination array. This is performed behind the scenes, enabling the developer to use just one line of code to call the method.

For your convenience, Java SE provides several methods for performing array manipulations (common tasks, such as copying, sorting and searching arrays) in the [java.util.Arrays](#) class. For instance, the previous example can be modified to use the `copyOfRange` method of the `java.util.Arrays` class, as you can see in the [ArrayCopyOfDemo](#) example. The difference is that using the `copyOfRange` method does not require you to create the destination array before calling the method, because the destination array is returned by the method:

```

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

        char[] copyFrom = {'d', 'e', 'c', 'a', 'f', 'f', 'e',
                           'i', 'n', 'a', 't', 'e', 'd'};

        char[] copyTo = java.util.Arrays.copyOfRange(copyFrom, 2, 9);

        System.out.println(new String(copyTo));
    }
}

```

As you can see, the output from this program is the same (caffeine), although it requires fewer lines of code. Note that the second parameter of the `copyOfRange` method is the initial index of the range to be copied, inclusively, while the third parameter is the final index of the range to be copied, exclusively. In this example, the range to be copied does not include the array element at index 9 (which contains the character a).

Some other useful operations provided by methods in the `java.util.Arrays` class, are:

- Searching an array for a specific value to get the index at which it is placed (the `binarySearch` method).
- Comparing two arrays to determine if they are equal or not (the `equals` method).
- Filling an array to place a specific value at each index (the `fill` method).
- Sorting an array into ascending order. This can be done either sequentially, using the `sort` method, or concurrently, using the `parallelSort` method introduced in Java SE 8. Parallel sorting of large arrays on multiprocessor systems is faster than sequential array sorting.

## STRING PROCESSING

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

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

### Creating Strings :

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

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

In this case, "Hello world!" is a *string literal*—a series of characters in your code that is enclosed in double quotes. Whenever it encounters a string literal in your code, the compiler creates a `String` object with its value—in this case, Hello world!.

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

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

The last line of this code snippet displays **hello**.

**Note:** `String` objects are stored in a special memory area known as string constant pool.

**Note:** The `String` class is immutable, so that once it is created a `String` object cannot be changed. The `String` class has a number of methods, some of which will be discussed below, that appear to modify strings. Since strings are immutable, what these methods really do is create and return a new string that contains the result of the operation.

The Java language provides special support for the string concatenation operator ( `+` ), and for conversion of other objects to strings. String concatenation is implemented through the `StringBuilder`(or `StringBuffer`) class and its `append` method. String conversions are implemented through the method `toString`, defined by `Object` and inherited by all classes in Java. For additional information on string concatenation and conversion, see Gosling, Joy, and Steele, *The Java Language Specification*.



Unless otherwise noted, passing a null argument to a constructor or method in this class will cause a `NullPointerException` to be thrown.

### **StringBuffer :**

A thread-safe, mutable sequence of characters. A `StringBuffer` is like a `String`, but can be modified. At any point in time it contains some particular sequence of characters, but the length and content of the sequence can be changed through certain method calls.

`String` buffers are safe for use by multiple threads. The methods are synchronized where necessary so that all the operations on any particular instance behave as if they occur in some serial order that is consistent with the order of the method calls made by each of the individual threads involved.

The principal operations on a `StringBuffer` are the `append` and `insert` methods, which are overloaded so as to accept data of any type. Each effectively converts a given datum to a string and then appends or inserts the characters of that string to the string buffer. The `append` method always adds these characters at the end of the buffer; the `insert` method adds the characters at a specified point.

For example :

if `z` refers to a string buffer object whose current contents are "start", then the method call `z.append("le")` would cause the string buffer to contain "startle", whereas `z.insert(4, "le")` would alter the string buffer to contain "starlet".

In general, if `sb` refers to an instance of a `StringBuffer`, then `sb.append(x)` has the same effect as `sb.insert(sb.length(), x)`.

Whenever an operation occurs involving a source sequence (such as appending or inserting from a source sequence) this class synchronizes only on the string buffer performing the operation, not on the source.

Every string buffer has a capacity. As long as the length of the character sequence contained in the string buffer does not exceed the capacity, it is not necessary to allocate a new internal buffer array. If the internal buffer overflows, it is automatically made larger. As of release JDK 5,

this class has been supplemented with an equivalent class designed for use by a single thread, `StringBuilder`. The `StringBuilder` class should generally be used in preference to this one, as it supports all of the same operations but it is faster, as it performs no synchronization.

## Java String Compare :

We can compare string in java on the basis of content and reference.

It is used in **authentication** (by `equals()` method), **sorting** (by `compareTo()` method), **reference matching** (by `==` operator) etc.

There are three ways to compare string in java:

1. By `equals()` method
2. By `==` operator
3. By `compareTo()` method

### 1) String compare by `equals()` method

The `String equals()` method compares the original content of the string. It compares values of string for equality. `String` class provides two methods:

**`public boolean equals(Object another)`** : compares this string to the specified object.

**`public boolean equalsIgnoreCase(String another)`** : compares this `String` to another string, ignoring case.

**Ex :**

```
package com.ics.demo;

public class StringDemo {
    public static void main(String[] abc) {
        String s1="ICS";
        String s2="ICS";
        String s3=new String("ICS");
        String s4="Infics";
        System.out.println(s1.equals(s2)); //true
        System.out.println(s1.equals(s3)); //true
        System.out.println(s1.equals(s4)); //false
    }
}
```

```
}  
}
```

## 2) String compare by == operator :

The == operator compares references not values.

**Ex:**

```
String s1="ICS";  
String s2="ICS";  
String s3=new String("ICS");  
System.out.println(s1==s2);//true (because both refer to same instance)  
System.out.println(s1==s3);//false(because s3 refers to instance created in nonpool)
```

## 3) String compare by compareTo() method :

The String compareTo() method compares values lexicographically and returns an integer value that describes if first string is less than, equal to or greater than second string.

Suppose s1 and s2 are two string variables. If:

- **s1 == s2** : 0
- **s1 > s2** :positive value
- **s1 < s2** :negative value

**Ex:**

```
String s1="ICS";  
String s2="ICS";  
String s3=new String("Infics");  
System.out.println(s1.compareTo(s2));//0  
System.out.println(s1.compareTo(s3));// -43 (because s1<s3)  
System.out.println(s3.compareTo(s1));// 43(because s3 > s1 )
```

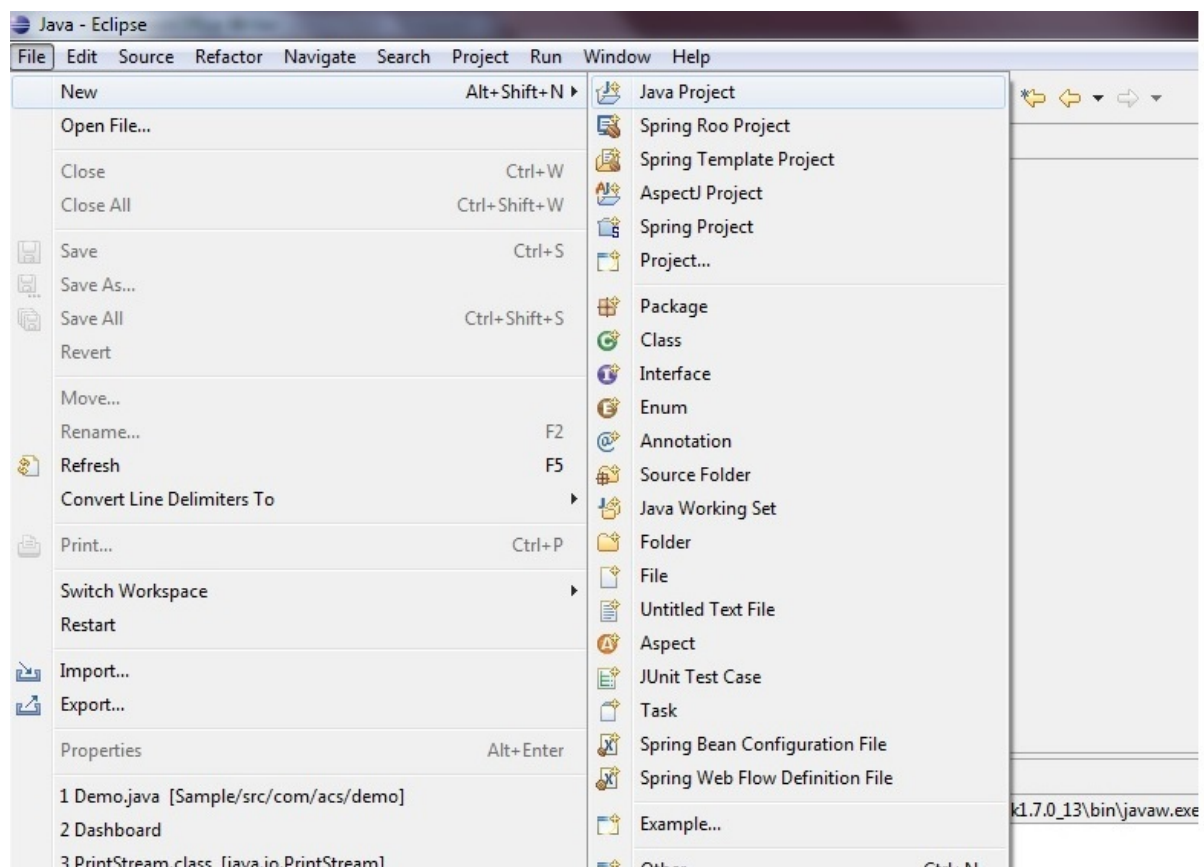
## INTRODUCTION TO TOOL FOR DEVELOPMENT E.G. ECLIPSE

An integrated development environment (**IDE**) is a software application that provides comprehensive facilities to computer programmers for software development. An **IDE** normally consists of a source code editor, build automation tools and a debugger.

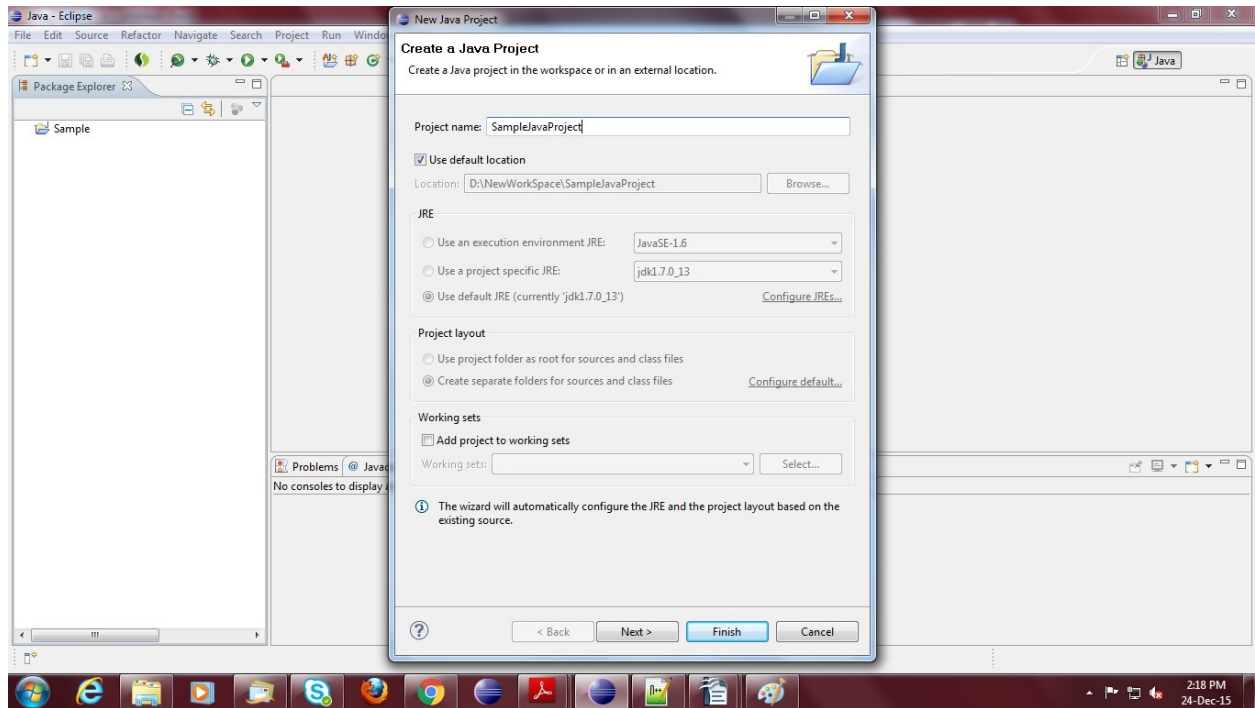
There are So many IDE's are available in market some of them are Eclipse, NetBeans, MyEclipse, RAD (IBM)...etc

### How to Create a JAVA Project in IDE:

Step 1: File > New > Java Project



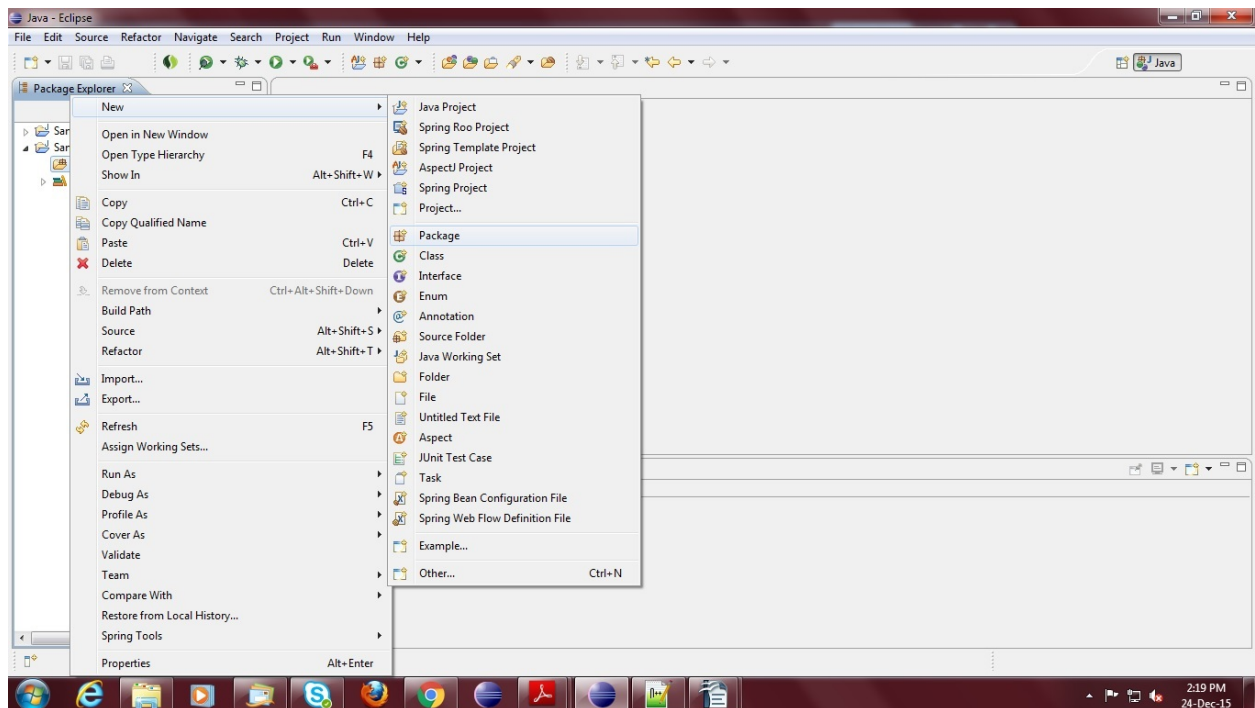
Select 'JavaProject' after that a new Perspective will open and enter Project Name > next > Finish



> create a Package :

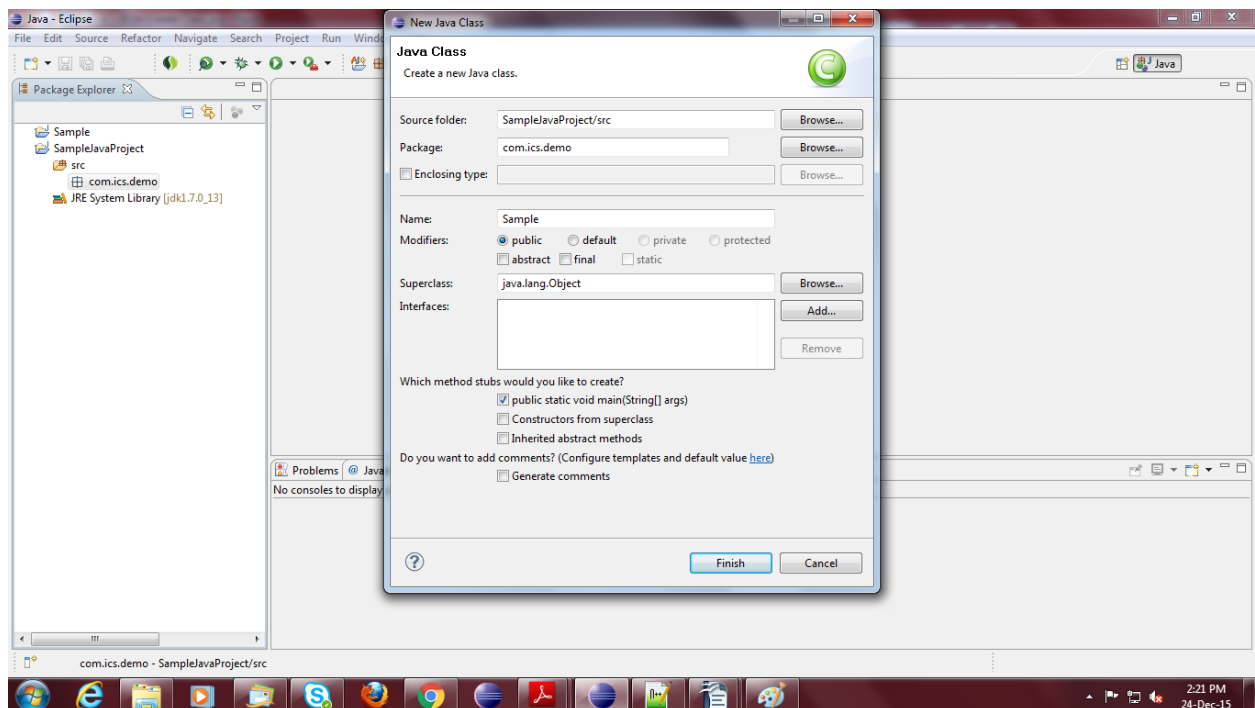
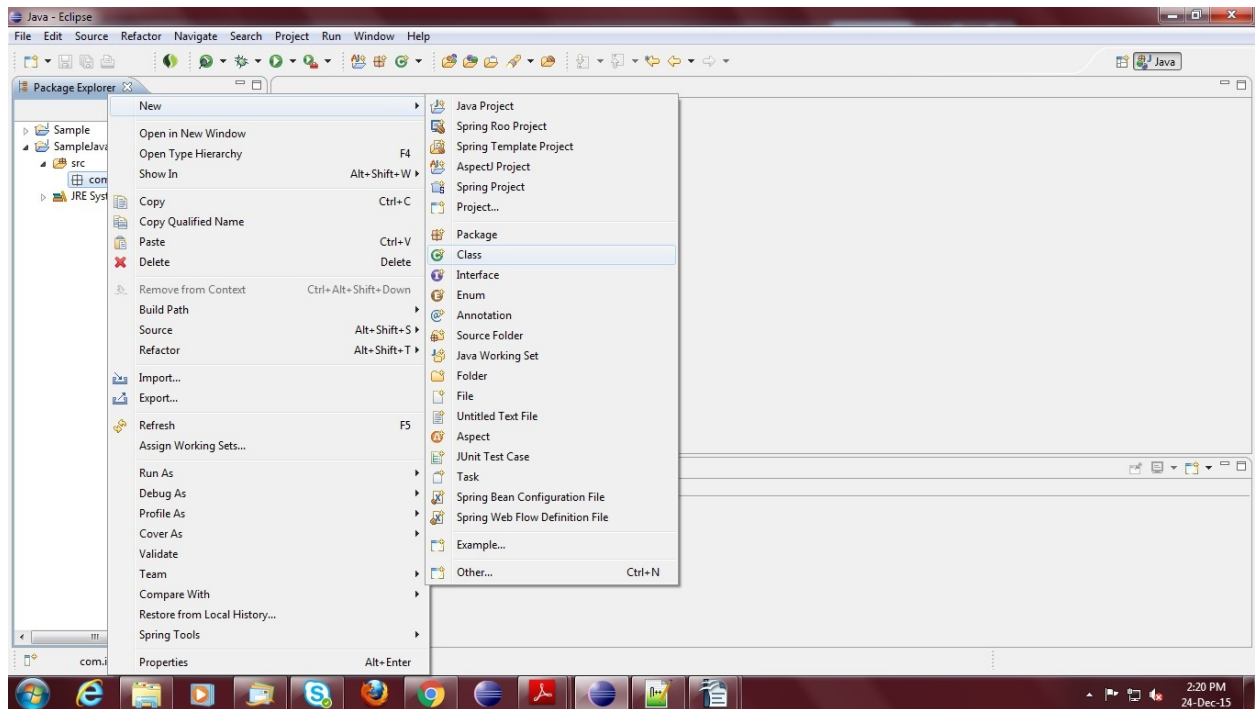
go to Project folder explore it .

src > New > package > give a package name.



Create a Class :

Go to Package right click > new > Class



Class name: Give a Class Name and Finish.

Run a Class : Go to Class File Right Click > Run As > Java Application

## OBJECT-ORIENTED PROGRAMMING: INHERITANCE

**Definitions:** A class that is derived from another class is called a *subclass* (also a *derived class*, *extended class*, or *child class*). The class from which the subclass is derived is called a *superclass* (also a *base class* or a *parent class*)

Excepting *Object*, which has no superclass, every class has one and only one direct superclass (single inheritance). In the absence of any other explicit superclass, every class is implicitly a subclass of *Object*.

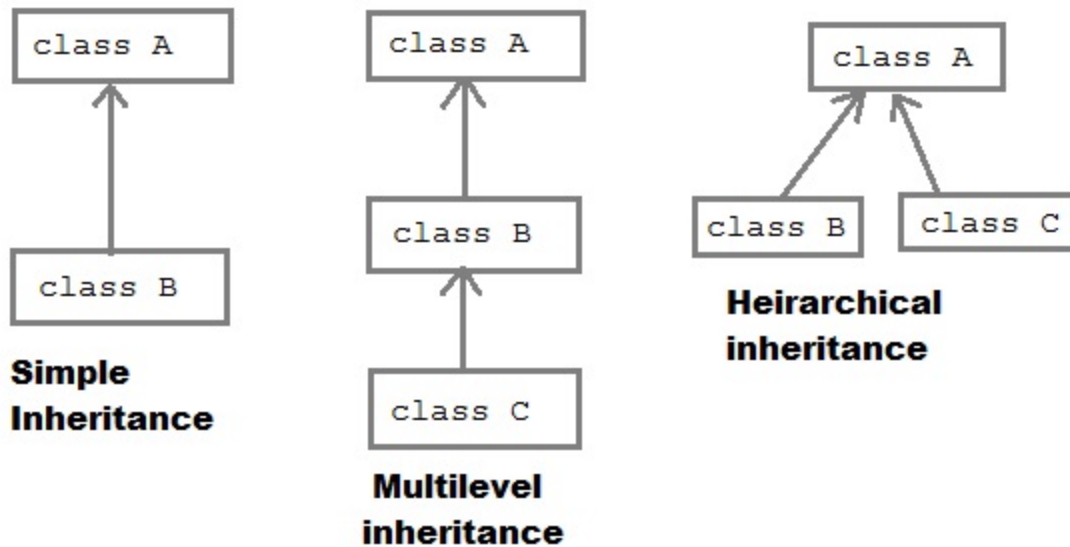
Classes can be derived from classes that are derived from classes that are derived from classes, and so on, and ultimately derived from the topmost class, *Object*. Such a class is said to be *descended* from all the classes in the inheritance chain stretching back to *Object*.

The idea of inheritance is simple but powerful: When you want to create a new class and there is already a class that includes some of the code that you want, you can derive your new class from the existing class. In doing this, you can reuse the fields and methods of the existing class without having to write (and debug!) them yourself.

A subclass inherits all the *members* (fields, methods, and nested classes) from its superclass. Constructors are not members, so they are not inherited by subclasses, but the constructor of the superclass can be invoked from the subclass.

### The Java Platform Class Hierarchy

The *Object* class, defined in the `java.lang` package, defines and implements behavior common to all classes—including the ones that you write. In the Java platform, many classes derive directly from *Object*, other classes derive from some of those classes, and so on, forming a hierarchy of classes.



### An Example of Inheritance

Here is the sample code for a possible implementation of a Bicycle class that was presented in the Classes and Objects lesson:

```
public class Bicycle {  
  
    // the Bicycle class has three fields  
    public int cadence;  
    public int gear;  
    public int speed;  
  
    // the Bicycle class has one constructor  
    public Bicycle(int startCadence, int startSpeed, int startGear) {  
        gear = startGear;  
        cadence = startCadence;  
        speed = startSpeed;  
    }  
  
    // the Bicycle class has four methods
```



```

public void setCadence(int newValue) {
    cadence = newValue;
}

public void setGear(int newValue) {
    gear = newValue;
}

public void applyBrake(int decrement) {
    speed -= decrement;
}

public void speedUp(int increment) {
    speed += increment;
}
}

```

A class declaration for a MountainBike class that is a subclass of Bicycle might look like this:

```

public class MountainBike extends Bicycle {

    // the MountainBike subclass adds one field
    public int seatHeight;

    // the MountainBike subclass has one constructor
    public MountainBike(int startHeight,
        int startCadence,
        int startSpeed,
        int startGear) {
        super(startCadence, startSpeed, startGear);
        seatHeight = startHeight;
    }

    // the MountainBike subclass adds one method

```

```

    public void setHeight(int newValue) {
        seatHeight = newValue;
    }
}

```

MountainBike inherits all the fields and methods of Bicycle and adds the field `seatHeight` and a method to set it. Except for the constructor, it is as if you had written a new `MountainBike` class entirely from scratch, with four fields and five methods. However, you didn't have to do all the work. This would be especially valuable if the methods in the `Bicycle` class were complex and had taken substantial time to debug.

**Note :** Inheritance represents the **IS-A relationship**, also known as *parent-child* relationship.

### What You Can Do in a Subclass

A subclass inherits all of the *public* and *protected* members of its parent, no matter what package the subclass is in. If the subclass is in the same package as its parent, it also inherits the *package-private* members of the parent. You can use the inherited members as is, replace them, hide them, or supplement them with new members:

- The inherited fields can be used directly, just like any other fields.
- You can declare a field in the subclass with the same name as the one in the superclass, thus *hiding* it (not recommended).
- You can declare new fields in the subclass that are not in the superclass.
- The inherited methods can be used directly as they are.
- You can write a new *instance* method in the subclass that has the same signature as the one in the superclass, thus *overriding* it.
- You can write a new *static* method in the subclass that has the same signature as the one in the superclass, thus *hiding* it.
- You can declare new methods in the subclass that are not in the superclass.
- You can write a subclass constructor that invokes the constructor of the superclass, either implicitly or by using the keyword `super`.

The following sections in this lesson will expand on these topics.

## Private Members in a Superclass

A subclass does not inherit the private members of its parent class. However, if the superclass has public or protected methods for accessing its private fields, these can also be used by the subclass.

A nested class has access to all the private members of its enclosing class—both fields and methods. Therefore, a public or protected nested class inherited by a subclass has indirect access to all of the private members of the superclass.

## Casting Objects

We have seen that an object is of the data type of the class from which it was instantiated. For example, if we write

```
public MountainBike myBike = new MountainBike();
```

then `myBike` is of type `MountainBike`.

`MountainBike` is descended from `Bicycle` and `Object`. Therefore, a `MountainBike` is a `Bicycle` and is also an `Object`, and it can be used wherever `Bicycle` or `Object` objects are called for.

The reverse is not necessarily true: a `Bicycle` *may be* a `MountainBike`, but it isn't necessarily. Similarly, an `Object` *may be* a `Bicycle` or a `MountainBike`, but it isn't necessarily.

*Casting* shows the use of an object of one type in place of another type, among the objects permitted by inheritance and implementations. For example, if we write

```
Object obj = new MountainBike();
```

then `obj` is both an `Object` and a `MountainBike` (until such time as `obj` is assigned another object that is *not* a `MountainBike`). This is called *implicit casting*.

If, on the other hand, we write

```
MountainBike myBike = obj;
```

we would get a compile-time error because `obj` is not known to the compiler to be a `MountainBike`. However, we can *tell* the compiler that we promise to assign a `MountainBike` to `obj` by *explicit casting*:

```
MountainBike myBike = (MountainBike)obj;
```

This cast inserts a runtime check that obj is assigned a MountainBike so that the compiler can safely assume that obj is a MountainBike. If obj is not a MountainBike at runtime, an exception will be thrown.

**Note:** You can make a logical test as to the type of a particular object using the instanceof operator. This can save you from a runtime error owing to an improper cast. For example:

```
if (obj instanceof MountainBike) {  
    MountainBike myBike = (MountainBike)obj;  
}
```

Here the instanceof operator verifies that obj refers to a MountainBike so that we can make the cast with knowledge that there will be no runtime exception thrown.

**Note :** If a class have an entity reference, it is known as Aggregation.

Aggregation represents **HAS-A relationship**.

## Overriding and Hiding Methods :

### Instance Methods

An instance method in a subclass with the same signature (name, plus the number and the type of its parameters) and return type as an instance method in the superclass *overrides* the superclass's method.

The ability of a subclass to override a method allows a class to inherit from a superclass whose behavior is "close enough" and then to modify behavior as needed. The overriding method has the same name, number and type of parameters, and return type as the method that it overrides. An overriding method can also return a subtype of the type returned by the overridden method. This subtype is called a *covariant return type*.

When overriding a method, you might want to use the `@Override` annotation that instructs the compiler that you intend to override a method in the superclass. If, for some reason, the compiler detects that the method does not exist in one of the superclasses, then it will generate an error. For more information on `@Override`, see [Annotations](#).

## Static Methods

If a subclass defines a static method with the same signature as a static method in the superclass, then the method in the subclass *hides* the one in the superclass.

The distinction between hiding a static method and overriding an instance method has important implications:

- The version of the overridden instance method that gets invoked is the one in the subclass.
- The version of the hidden static method that gets invoked depends on whether it is invoked from the superclass or the subclass.

Consider an example that contains two classes. The first is `Animal`, which contains one instance method and one static method:

```
public class Animal {  
    public static void testClassMethod() {  
        System.out.println("The static method in Animal");  
    }  
    public void testInstanceMethod() {  
        System.out.println("The instance method in Animal");  
    }  
}
```

The second class, a subclass of `Animal`, is called `Cat`:

```
public class Cat extends Animal {  
    public static void testClassMethod() {  
        System.out.println("The static method in Cat");  
    }  
    public void testInstanceMethod() {  
        System.out.println("The instance method in Cat");  
    }  
  
    public static void main(String[] args) {
```

```

    Cat myCat = new Cat();
    Animal myAnimal = myCat;
    Animal.testClassMethod();
    myAnimal.testInstanceMethod();
}
}

```

The Cat class overrides the instance method in Animal and hides the static method in Animal. The main method in this class creates an instance of Cat and invokes testClassMethod() on the class and testInstanceMethod() on the instance.

The output from this program is as follows:

The static method in Animal

The instance method in Cat

As promised, the version of the hidden static method that gets invoked is the one in the superclass, and the version of the overridden instance method that gets invoked is the one in the subclass.

## Interface Methods

[Default methods](#) and [abstract methods](#) in interfaces are inherited like instance methods. However, when the supertypes of a class or interface provide multiple default methods with the same signature, the Java compiler follows inheritance rules to resolve the name conflict. These rules are driven by the following two principles:

Instance methods are preferred over interface default methods.

Consider the following classes and interfaces:

```

public class Horse {
    public String identifyMyself() {
        return "I am a horse.";
    }
}

```

```

public interface Flyer {
    default public String identifyMyself() {
        return "I am able to fly.";
    }
}

public interface Mythical {
    default public String identifyMyself() {
        return "I am a mythical creature.";
    }
}

public class Pegasus extends Horse implements Flyer, Mythical {
    public static void main(String... args) {
        Pegasus myApp = new Pegasus();
        System.out.println(myApp.identifyMyself());
    }
}

```

- }
- The method Pegasus.identifyMyself returns the string I am a horse.

Methods that are already overridden by other candidates are ignored. This circumstance can arise when supertypes share a common ancestor.

Consider the following interfaces and classes:

```

public interface Animal {
    default public String identifyMyself() {
        return "I am an animal.";
    }
}

public interface EggLayer extends Animal {
    default public String identifyMyself() {
        return "I am able to lay eggs.";
    }
}

public interface FireBreather extends Animal {}

```

```

public class Dragon implements EggLayer, FireBreather {
    public static void main (String... args) {
        Dragon myApp = new Dragon();
        System.out.println(myApp.identifyMyself());
    }
}

```

- 

The method `Dragon.identifyMyself` returns the string I am able to lay eggs.

If two or more independently defined default methods conflict, or a default method conflicts with an abstract method, then the Java compiler produces a compiler error. You must explicitly override the supertype methods.

Consider the example about computer-controlled cars that can now fly. You have two interfaces (`OperateCar` and `FlyCar`) that provide default implementations for the same method, (`startEngine`):

```

public interface OperateCar {
    // ...
    default public int startEngine(EncryptedKey key) {
        // Implementation
    }
}

public interface FlyCar {
    // ...
    default public int startEngine(EncryptedKey key) {
        // Implementation
    }
}

```

A class that implements both `OperateCar` and `FlyCar` must override the method `startEngine`. You could invoke any of the of the default implementations with the `super` keyword.

```

public class FlyingCar implements OperateCar, FlyCar {

```



```
// ...
public int startEngine(EncryptedKey key) {
    FlyCar.super.startEngine(key);
    OperateCar.super.startEngine(key);
}
}
```

The name preceding `super` (in this example, `FlyCar` or `OperateCar`) must refer to a direct superinterface that defines or inherits a default for the invoked method. This form of method invocation is not restricted to differentiating between multiple implemented interfaces that contain default methods with the same signature. You can use the `super` keyword to invoke a default method in both classes and interfaces.

Inherited instance methods from classes can override abstract interface methods. Consider the following interfaces and classes:

```
public interface Mammal {
    String identifyMyself();
}

public class Horse {
    public String identifyMyself() {
        return "I am a horse.";
    }
}

public class Mustang extends Horse implements Mammal {
    public static void main(String... args) {
        Mustang myApp = new Mustang();
        System.out.println(myApp.identifyMyself());
    }
}
```

The method `Mustang.identifyMyself` returns the string `I am a horse`. The class `Mustang` inherits the method `identifyMyself` from the class `Horse`, which overrides the abstract method of the same name in the interface `Mammal`.

Note: Static methods in interfaces are never inherited.

## Modifiers

The access specifier for an overriding method can allow more, but not less, access than the overridden method. For example, a protected instance method in the superclass can be made public, but not private, in the subclass.

You will get a compile-time error if you attempt to change an instance method in the superclass to a static method in the subclass, and vice versa.

## Summary

The following table summarizes what happens when you define a method with the same signature as a method in a superclass.

Defining a Method with the Same Signature as a Superclass's Method

		Superclass Instance Method	Superclass Static Method
Subclass Instance Method	Overrides		Generates a compile-time error
Subclass Static Method	Generates a compile-time error		Hides

**Note:** In a subclass, you can overload the methods inherited from the superclass. Such overloaded methods neither hide nor override the superclass instance methods—they are new methods, unique to the subclass.

## OBJECT-ORIENTED PROGRAMMING: POLYMORPHISM

The dictionary definition of polymorphism refers to a principle in biology in which an organism or species can have many different forms or stages. This principle can also be applied to object-oriented programming and languages like the Java language. Subclasses of a class can define their own unique behaviors and yet share some of the same functionality of the parent class.

Polymorphism is the capability of a method to do different things based on the object that it is acting upon. In other words, polymorphism allows you define one interface and have multiple implementations.

- It is a feature that allows one interface to be used for a general class of actions.
- An operation may exhibit different behavior in different instances.
- The behavior depends on the types of data used in the operation.
- It plays an important role in allowing objects having different internal structures to share the same external interface.
- Polymorphism is extensively used in implementing inheritance.

There are two types of polymorphism in java- **Runtime polymorphism( Dynamic polymorphism)** and **Compile time polymorphism (static polymorphism)**.

### **Runtime Polymorphism( or Dynamic polymorphism)**

Method Overriding is a perfect example of runtime polymorphism. In this kind of polymorphism, reference of `class X` can hold object of `class X` or an object of any sub classes of `class X`. For e.g. if `class Y` extends `class X` then both of the following statements are valid:

```
Y obj = new Y();
```

```
//Parent class reference can be assigned to child object
```

```
X obj = new Y();
```

Since in method overriding both the classes(base class and child class) have same method, compile doesn't figure out which method to call at compile-time. In this case JVM(java virtual

machine) decides which method to call at runtime that's why it is known as runtime or dynamic polymorphism.

Lets see an example to understand it better.

```
public class X
{
    public void methodA() //Base class method
    {
        System.out.println ("hello, I'm methodA of class X");
    }
}
```

```
public class Y extends X
{
    public void methodA() //Derived Class method
    {
        System.out.println ("hello, I'm methodA of class Y");
    }
}
```

```
public class Z
{
    public static void main (String args []) {
        X obj1 = new X(); // Reference and object X
        X obj2 = new Y(); // X reference but Y object
        obj1.methodA();
        obj2.methodA();
    }
}
```

Output:

hello, I'm methodA of class X

hello, I'm methodA of class Y

As you can see the `methodA` has different-2 forms in child and parent class thus we can say `methodA` here is polymorphic.

### **Rules for Method Overriding:**

1. applies only to inherited methods
2. object type (NOT reference variable type) determines which overridden method will be used at runtime
3. Overriding method can have different return type.
4. Overriding method must not have more restrictive access modifier
5. Abstract methods must be overridden
6. Static and final methods cannot be overridden
7. Constructors cannot be overridden
8. It is also known as Runtime polymorphism.

### **super keyword in Overriding:**

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

Example:

```
class Vehicle {  
  
    public void move () {  
        System.out.println ("Vehicles are used for moving from one place to another ");  
    }  
}  
  
class Car extends Vehicle {  
    public void move () {  
        super. move (); // invokes the super class method  
        System.out.println ("Car is a good medium of transport ");  
    }  
}
```

```

public class TestCar {
    public static void main (String args []){
        Vehicle b = new Car (); // Vehicle reference but Car object
        b.move (); //Calls the method in Car class
    }
}

```

Output:

Vehicles are used for moving from one place to another

Car is a good medium of transport

### **Compile time Polymorphism( or Static polymorphism)**

Compile time polymorphism is nothing but the method overloading in java. In simple terms we can say that a class can have more than one methods with same name but with different number of arguments or different types of arguments or both.

Lets see the below example to understand it better-

```

class X
{
    void methodA(int num)
    {
        System.out.println ("methodA:" + num);
    }
    void methodA(int num1, int num2)
    {
        System.out.println ("methodA:" + num1 + "," + num2);
    }
    double methodA(double num) {
        System.out.println("methodA:" + num);
        return num;
    }
}

```

```

class Y
{
    public static void main (String args [])
    {
        X Obj = new X();
        double result;
        Obj.methodA(20);
        Obj.methodA(20, 30);
        result = Obj.methodA(5.5);
        System.out.println("Answer is:" + result);
    }
}

```

Output:

methodA:20

methodA:20,30

methodA:5.5

Answer is:5.5

As you can see in the above example that the class has three variance of `methodA` or we can say `methodA` is polymorphic in nature since it is having three different forms. In such scenario, compiler is able to figure out the method call at compile-time that's the reason it is known as compile time polymorphism.

### Rules for Method Overloading

1. Overloading can take place in the same class or in its sub-class.
2. Constructor in Java can be overloaded
3. Overloaded methods must have a different argument list.
4. Overloaded method should always be the part of the same class (can also take place in sub class), with same name but different parameters.
5. The parameters may differ in their type or number, or in both.
6. They may have the same or different return types.
7. It is also known as compile time polymorphism.

## INTRODUCTION TO GRAPHICAL USER INTERFACES (GUIS):

JFC is short for Java Foundation Classes, which encompass a group of features for building graphical user interfaces (GUIs) and adding rich graphics functionality and interactivity to Java applications. It is defined as containing the features shown in the table below.

Feature	Description
Swing Components	GUI Includes everything from buttons to split panes to tables. Many components are capable of sorting, printing, and drag and drop, to name a few of the supported features.
Pluggable Look-and-Feel Support	The look and feel of Swing applications is pluggable, allowing a choice of look and feel. For example, the same program can use either the Java or the Windows look and feel. Additionally, the Java platform supports the GTK+ look and feel, which makes hundreds of existing look and feels available to Swing programs. Many more look-and-feel packages are available from various sources.
Accessibility API	Enables assistive technologies, such as screen readers and Braille displays, to get information from the user interface.
Java 2D API	Enables developers to easily incorporate high-quality 2D graphics, text, and images in applications and applets. Java 2D includes extensive APIs for generating and sending high-quality output to printing devices.



Internationalization Allows developers to build applications that can interact with users worldwide in their own languages and cultural conventions. With the input method framework developers can build applications that accept text in languages that use thousands of different characters, such as Japanese, Chinese, or Korean.

This trail concentrates on the Swing components. We help you choose the appropriate components for your GUI, tell you how to use them, and give you the background information you need to use them effectively. We also discuss other features as they apply to Swing components.

### Which Swing Packages Should I Use?

The Swing API is powerful, flexible — and immense. The Swing API has 18 public packages:

<code>javax.accessibility</code>	<code>javax.swing.plaf</code>	<code>javax.swing.text</code>
<code>javax.swing</code>	<code>javax.swing.plaf.basic</code>	<code>javax.swing.text.html</code>
<code>javax.swing.border</code>	<code>javax.swing.plaf.metal</code>	<code>javax.swing.text.html.parser</code>
<code>javax.swing.colorchooser</code>	<code>javax.swing.plaf.multi</code>	<code>javax.swing.text.rtf</code>
<code>javax.swing.event</code>	<code>javax.swing.plaf.synth</code>	<code>javax.swing.tree</code>
<code>javax.swing.filechooser</code>	<code>javax.swing.table</code>	<code>javax.swing.undo</code>

Fortunately, most programs use only a small subset of the API. This trail sorts out the API for you, giving you examples of common code and pointing you to methods and classes you're likely to need. Most of the code in this trail uses only one or two Swing packages:

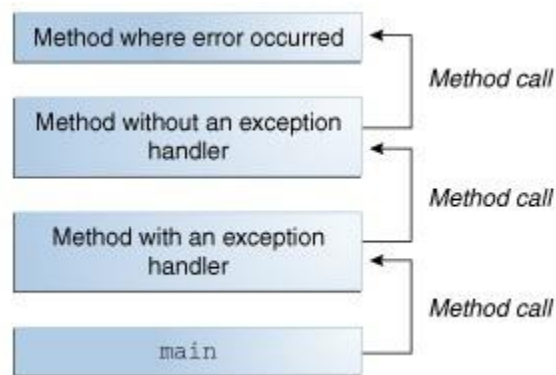
- `javax.swing`
- `javax.swing.event` (not always required)

## EXCEPTION HANDLING

**Definition:** An *exception* is an event, which occurs during the execution of a program, that disrupts the normal flow of the program's instructions.

When an error occurs within a method, the method creates an object and hands it off to the runtime system. The object, called an exception object, contains information about the error, including its type and the state of the program when the error occurred. Creating an exception object and handing it to the runtime system is called throwing an exception.

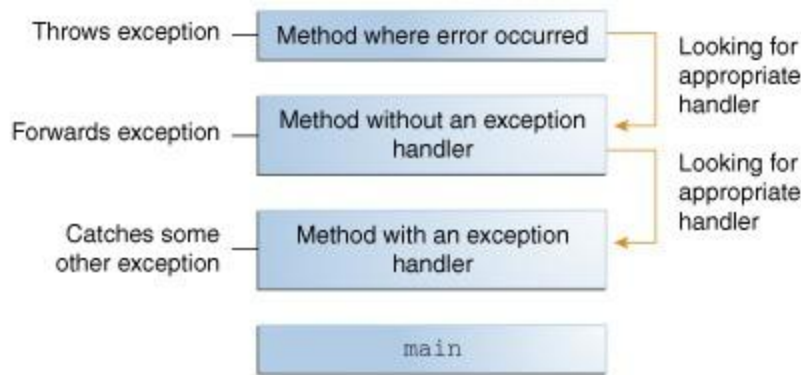
After a method throws an exception, the runtime system attempts to find something to handle it. The set of possible "somethings" to handle the exception is the ordered list of methods that had been called to get to the method where the error occurred. The list of methods is known as the call stack (see the next figure).



**The call stack**

The runtime system searches the call stack for a method that contains a block of code that can handle the exception. This block of code is called an exception handler. The search begins with the method in which the error occurred and proceeds through the call stack in the reverse order in which the methods were called. When an appropriate handler is found, the runtime system passes the exception to the handler. An exception handler is considered appropriate if the type of the exception object thrown matches the type that can be handled by the handler.

The exception handler chosen is said to catch the exception. If the runtime system exhaustively searches all the methods on the call stack without finding an appropriate exception handler, as shown in the next figure, the runtime system (and, consequently, the program) terminates.



### Searching the call stack for the exception handler

Using exceptions to manage errors has some advantages over traditional error-management techniques .

### The Catch or Specify Requirement

Valid Java programming language code must honor the *Catch or Specify Requirement*. This means that code that might throw certain exceptions must be enclosed by either of the following:

- A try statement that catches the exception. The try must provide a handler for the exception.
- A method that specifies that it can throw the exception. The method must provide a throws clause that lists the exception.

Code that fails to honor the Catch or Specify Requirement will not compile.

Not all exceptions are subject to the Catch or Specify Requirement. To understand why, we need to look at the three basic categories of exceptions, only one of which is subject to the Requirement.

### The Three Kinds of Exceptions

The first kind of exception is the *checked exception*. These are exceptional conditions that a well-written application should anticipate and recover from. For example, suppose an application prompts a user for an input file name, then opens the file by passing the name to the

constructor for `java.io.FileReader`. Normally, the user provides the name of an existing, readable file, so the construction of the `FileReader` object succeeds, and the execution of the application proceeds normally. But sometimes the user supplies the name of a nonexistent file, and the constructor throws `java.io.FileNotFoundException`. A well-written program will catch this exception and notify the user of the mistake, possibly prompting for a corrected file name.

Checked exceptions *are subject* to the Catch or Specify Requirement. All exceptions are checked exceptions, except for those indicated by `Error`, `RuntimeException`, and their subclasses.

The second kind of exception is the *error*. These are exceptional conditions that are external to the application, and that the application usually cannot anticipate or recover from. For example, suppose that an application successfully opens a file for input, but is unable to read the file because of a hardware or system malfunction. The unsuccessful read will throw `java.io.IOException`. An application might choose to catch this exception, in order to notify the user of the problem — but it also might make sense for the program to print a stack trace and exit.

Errors *are not subject* to the Catch or Specify Requirement. Errors are those exceptions indicated by `Error` and its subclasses.

The third kind of exception is the *runtime exception*. These are exceptional conditions that are internal to the application, and that the application usually cannot anticipate or recover from. These usually indicate programming bugs, such as logic errors or improper use of an API. For example, consider the application described previously that passes a file name to the constructor for `FileReader`. If a logic error causes a null to be passed to the constructor, the constructor will throw `NullPointerException`. The application can catch this exception, but it probably makes more sense to eliminate the bug that caused the exception to occur.

Runtime exceptions *are not subject* to the Catch or Specify Requirement. Runtime exceptions are those indicated by `RuntimeException` and its subclasses.

Errors and runtime exceptions are collectively known as *unchecked exceptions*.

## **Catching and Handling Exceptions**

This section describes how to use the three exception handler components — the `try`, `catch`, and `finally` blocks — to write an exception handler. Then, the `try-with-resources` statement,

introduced in Java SE 7, is explained. The try-with-resources statement is particularly suited to situations that use Closeable resources, such as streams.

The last part of this section walks through an example and analyzes what occurs during various scenarios.

The following example defines and implements a class named ListOfNumbers. When constructed, ListOfNumbers creates an ArrayList that contains 10 Integer elements with sequential values 0 through 9. The ListOfNumbers class also defines a method named writeList, which writes the list of numbers into a text file called OutFile.txt. This example uses output classes defined in java.io.

// Note: This class will not compile yet.

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

```
import java.util.List;
```

```
import java.util.ArrayList;
```

```
public class ListOfNumbers {
```

```
    private List<Integer> list;
```

```
    private static final int SIZE = 10;
```

```
    public ListOfNumbers () {
```

```
        list = new ArrayList<Integer>(SIZE);
```

```
        for (int i = 0; i < SIZE; i++) {
```

```
            list.add(new Integer(i));
```

```
        }
```

```
    }
```

```

public void writeList() {

    // The FileWriter constructor throws IOException, which must be caught.

    PrintWriter out = new PrintWriter(new FileWriter("OutFile.txt"));

    for (int i = 0; i < SIZE; i++) {

        // The get(int) method throws IndexOutOfBoundsException, which must be caught.

        out.println("Value at: " + i + " = " + list.get(i));

    }

    out.close();

}
}

```

The first line in boldface is a call to a constructor. The constructor initializes an output stream on a file. If the file cannot be opened, the constructor throws an `IOException`. The second boldface line is a call to the `ArrayList` class's `get` method, which throws an `IndexOutOfBoundsException` if the value of its argument is too small (less than 0) or too large (more than the number of elements currently contained by the `ArrayList`).

If you try to compile the [ListOfNumbers](#) class, the compiler prints an error message about the exception thrown by the `FileWriter` constructor. However, it does not display an error message about the exception thrown by `get`. The reason is that the exception thrown by the constructor, `IOException`, is a checked exception, and the one thrown by the `get` method, `IndexOutOfBoundsException`, is an unchecked exception.

Now that you're familiar with the `ListOfNumbers` class and where the exceptions can be thrown within it, you're ready to write exception handlers to catch and handle those exceptions.

## Java Exception Handling Keywords:

There are 5 keywords used in java exception handling.

1. Try
2. catch
3. finally
4. throw
5. throws

### 1)The try Block:

The first step in constructing an exception handler is to enclose the code that might throw an exception within a try block. In general, a try block looks like the following:

```
try {
```

```
    // code
```

```
}
```

catch and finally blocks . . .

The segment in the example labeled code contains one or more legal lines of code that could throw an exception. (The catch and finally blocks are explained in the next two subsections.)

To construct an exception handler for the writeList method from the ListOfNumbers class, enclose the exception-throwing statements of the writeList method within a try block. There is more than one way to do this. You can put each line of code that might throw an exception within its own try block and provide separate exception handlers for each. Or, you can put all the writeList code within a single try block and associate multiple handlers with it. The following listing uses one try block for the entire method because the code in question is very short.

```
private List<Integer> list;
```



```

private static final int SIZE = 10;

public void writeList() {

    PrintWriter out = null;

    try {

        System.out.println("Entered try statement");

        out = new PrintWriter(new FileWriter("OutFile.txt"));

        for (int i = 0; i < SIZE; i++) {

            out.println("Value at: " + i + " = " + list.get(i));

        }

    }

    catch and finally blocks . . .

}

```

If an exception occurs within the try block, that exception is handled by an exception handler associated with it. To associate an exception handler with a try block, you must put a catchblock after it; the next section, [The catch Blocks](#), shows you how.

## **2)The catch Blocks :**

You associate exception handlers with a try block by providing one or more catch blocks directly after the try block. No code can be between the end of the try block and the beginning of the first catch block.

```

try {

    . . .

} catch (ExceptionType name) {

```

```
} catch (ExceptionType name) {  
  
}
```

Each catch block is an exception handler that handles the type of exception indicated by its argument. The argument type, `ExceptionType`, declares the type of exception that the handler can handle and must be the name of a class that inherits from the `Throwable` class. The handler can refer to the exception with `name`.

The catch block contains code that is executed if and when the exception handler is invoked. The runtime system invokes the exception handler when the handler is the first one in the call stack whose `ExceptionType` matches the type of the exception thrown. The system considers it a match if the thrown object can legally be assigned to the exception handler's argument.

The following are two exception handlers for the `writeList` method:

```
try {  
  
  
} catch (IndexOutOfBoundsException e) {  
    System.err.println("IndexOutOfBoundsException: " + e.getMessage());  
} catch (IOException e) {  
    System.err.println("Caught IOException: " + e.getMessage());  
}
```

Exception handlers can do more than just print error messages or halt the program. They can do error recovery, prompt the user to make a decision, or propagate the error up to a higher-level handler using chained exceptions, as described in the [Chained Exceptions](#) section.

### **Catching More Than One Type of Exception with One Exception Handler**

In Java SE 7 and later, a single catch block can handle more than one type of exception. This feature can reduce code duplication and lessen the temptation to catch an overly broad exception.

In the catch clause, specify the types of exceptions that block can handle, and separate each exception type with a vertical bar (|):

```
catch (IOException|SQLException ex) {  
  
    logger.log(ex);  
  
    throw ex;  
  
}
```

**Note:** If a catch block handles more than one exception type, then the catch parameter is implicitly final. In this example, the catch parameter `ex` is final and therefore you cannot assign any values to it within the catch block.

### **3)The finally Block:**

The finally block *always* executes when the try block exits. This ensures that the finally block is executed even if an unexpected exception occurs. But finally is useful for more than just exception handling — it allows the programmer to avoid having cleanup code accidentally bypassed by a return, continue, or break. Putting cleanup code in a finally block is always a good practice, even when no exceptions are anticipated.

**Note:** If the JVM exits while the try or catch code is being executed, then the finally block may not execute. Likewise, if the thread executing the try or catch code is interrupted or killed, the finally block may not execute even though the application as a whole continues.

The try block of the writeList method that you've been working with here opens a PrintWriter. The program should close that stream before exiting the writeList method. This poses a somewhat complicated problem because writeList's try block can exit in one of three ways.

1. The new FileWriter statement fails and throws an IOException.
2. The list.get(i) statement fails and throws an IndexOutOfBoundsException.
3. Everything succeeds and the try block exits normally.

The runtime system always executes the statements within the finally block regardless of what happens within the try block. So it's the perfect place to perform cleanup.

The following finally block for the writeList method cleans up and then closes the PrintWriter.

```
finally {  
  
    if (out != null) {  
  
        System.out.println("Closing PrintWriter");  
  
        out.close();  
  
    } else {  
  
        System.out.println("PrintWriter not open");  
  
    }  
  
}
```

**Important:** The finally block is a key tool for preventing resource leaks. When closing a file or otherwise recovering resources, place the code in a finally block to ensure that resource is *always* recovered.

#### 4) Java throw Keyword :

The Java throw keyword is used to explicitly throw an exception.

We can throw either checked or unchecked exception in java by throw keyword. The throw keyword is mainly used to throw custom exception. We will see custom exceptions later.

The syntax of java throw keyword is given below.

```
throw exception;
```

### **Java throw Keyword Example :**

In this example, we have created the validate method that takes integer value as a parameter. If the age is less than 18, we are throwing the ArithmeticException otherwise print a message welcome to vote.

```
public class DemoOfThrow{  
  
    static void validate(int age){  
        if(age<18)  
            throw new ArithmeticException("not valid");  
        else  
            System.out.println("welcome to vote");  
        }  
        public static void main(String args[]){  
            validate(13);  
            System.out.println("rest of the code...");  
        }  
    }  
}
```

### **5) Java throws Keyword :**

The **Java throws keyword** is used to declare an exception. It gives an information to the programmer that there may occur an exception so it is better for the programmer to provide the exception handling code so that normal flow can be maintained.

Exception Handling is mainly used to handle the checked exceptions. If there occurs any unchecked exception such as `NullPointerException`, it is programmers fault that he is not performing check up before the code being used.

### **Syntax:**

```
return_type method_name() throws exception class_name {  
  
    // Method Body  
  
}
```

### **Specifying the Exceptions Thrown by a Method :**

The previous section showed how to write an exception handler for the `writeList` method in the `ListofNumbers` class. Sometimes, it's appropriate for code to catch exceptions that can occur within it. In other cases, however, it's better to let a method further up the call stack handle the exception. For example, if you were providing the `ListofNumbers` class as part of a package of classes, you probably couldn't anticipate the needs of all the users of your package. In this case, it's better to not catch the exception and to allow a method further up the call stack to handle it.

If the `writeList` method doesn't catch the checked exceptions that can occur within it, the `writeList` method must specify that it can throw these exceptions. Let's modify the

original writeList method to specify the exceptions it can throw instead of catching them. To remind you, here's the original version of the writeList method that won't compile.

```
public void writeList() {  
  
    PrintWriter out = new PrintWriter(new FileWriter("OutFile.txt"));  
  
    for (int i = 0; i < SIZE; i++) {  
  
        out.println("Value at: " + i + " = " + list.get(i));  
  
    }  
  
    out.close();  
  
}
```

To specify that writeList can throw two exceptions, add a throws clause to the method declaration for the writeList method. The throws clause comprises the throws keyword followed by a comma-separated list of all the exceptions thrown by that method. The clause goes after the method name and argument list and before the brace that defines the scope of the method; here's an example.

```
public void writeList() throws IOException, IndexOutOfBoundsException {
```

Remember that IndexOutOfBoundsException is an unchecked exception; including it in the throws clause is not mandatory. You could just write the following.

```
public void writeList() throws IOException {
```

### How to Throw Exceptions:

Before you can catch an exception, some code somewhere must throw one. Any code can throw an exception: your code, code from a package written by someone else such as the

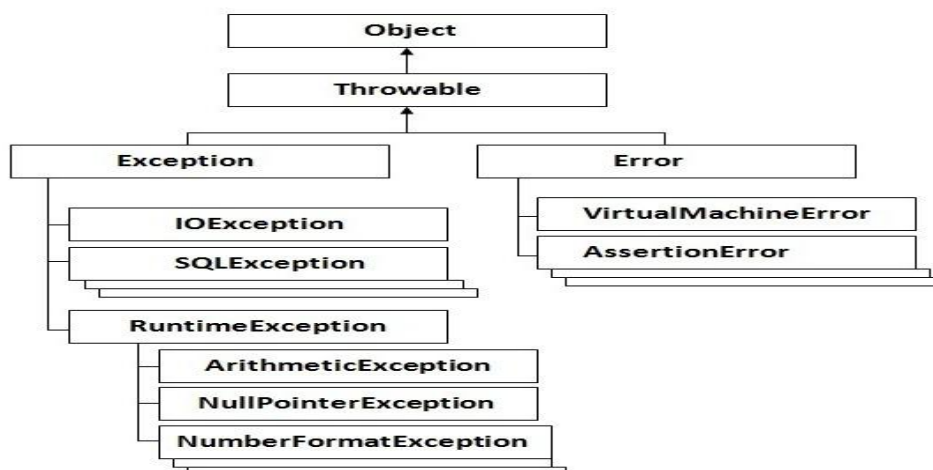
packages that come with the Java platform, or the Java runtime environment. Regardless of what throws the exception, it's always thrown with the throw statement.

As you have probably noticed, the Java platform provides numerous exception classes. All the classes are descendants of the [Throwable](#) class, and all allow programs to differentiate among the various types of exceptions that can occur during the execution of a program.

You can also create your own exception classes to represent problems that can occur within the classes you write. In fact, if you are a package developer, you might have to create your own set of exception classes to allow users to differentiate an error that can occur in your package from errors that occur in the Java platform or other packages.

### Throwable Class and Its Subclasses :

The objects that inherit from the Throwable class include direct descendants (objects that inherit directly from the Throwable class) and indirect descendants (objects that inherit from children or grandchildren of the Throwable class). The figure below illustrates the class hierarchy of the Throwable class and its most significant subclasses. As you can see, Throwable has two direct descendants: Exception and Error.





## Error Class :

When a dynamic linking failure or other hard failure in the Java virtual machine occurs, the virtual machine throws an Error. Simple programs typically do not catch or throw Errors.

## Exception Class :

Most programs throw and catch objects that derive from the Exception class. An Exception indicates that a problem occurred, but it is not a serious system problem. Most programs you write will throw and catch Exceptions as opposed to Errors.

The Java platform defines the many descendants of the Exception class. These descendants indicate various types of exceptions that can occur. For example, `IllegalAccessExceptio` signals that a particular method could not be found, and `NegativeArraySizeException` indicates that a program attempted to create an array with a negative size.

One Exception subclass, `RuntimeException`, is reserved for exceptions that indicate incorrect use of an API. An example of a runtime exception is `NullPointerException`, which occurs when a method tries to access a member of an object through a null reference.

## Custom Exceptions:

If we are creating our own Exception that is known as custom exception or user-defined exception. Java custom exceptions are used to customize the exception according to user need.

By the help of custom exception, you can have your own exception and message.

Let's see a simple example of java custom exception.

```
public class TestCustomException {  
  
    static void validate(int age)throws InvalidAgeException{  
        if(age<18)  
            throw new InvalidAgeException("You are not eligible for Voting ");  
        else  
            System.out.println("Congrats ! You are Eligible For Voting...");  
    }  
}
```

```
}
```

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

```
    try{
```

```
        validate(13);
```

```
    }catch(Exception m){System.out.println("Exception occurred: "+m);}
    System.out.println("Remaing Code....");
```

```
}
```

```
}
```

## CollectionFramework

**Definition:** Collection is a java object that is used to group homogeneous and heterogeneous object's without size limitation for carrying multiple objects at a time from one application to another application among multiple layers of MVC architecture as method arguments and return type.

- java.util package contains several classes to group/collect homogeneous and heterogeneous objects without size limitation. These classes are usually called collection framework classes.

### Limitation of Object type Arrays:

#### Arrays are fixed in size:

- That is once we created an array there is no chance of increasing or decreasing the size based on our requirement.
- Hence to use arrays concept compulsory we should know the size in advance, which mayn't possible always

#### Arrays can hold only homogeneous data type elements.

```
String[] s=new String[10];
```

```
s[0]=new String();
```

```
s[1]=new String();
```

```
s[2]="ICS";
```

we can overcome this limitation by using object type Arrays.

```
Object o[] = new Object[10];
```

```
o[0]=new String();
```

```
o[1]=new String();
```

```
o[2]="ICS";
```

Arrays concept isn't implemented based on some data structure hence ready made method support isn't available for arrays so for every requirement programmer is responsible to write the code.

- To overcome the above limitations should go for collections.
- Collections are growable in nature that is based on our requirement we can increase or decrease the size
- Collections can hold both homogeneous and heterogeneous elements.

- For every collection underlying data structure is available hence readymade method support is available for the every requirement.

**Collection framework:** It defines several classes and interface which can be used to represent group of objects as single unit. All the collection classes and interfaces are present in java.util package.

### What Is a Collections Framework?

A collections framework is a unified architecture for representing and manipulating collections.

All collections frameworks contain the following:

- **Interfaces:** These are abstract data types that represent collections. Interfaces allow collections to be manipulated independently of the details of their representation. In object-oriented languages, interfaces generally form a hierarchy.
- **Implementations:** These are the concrete implementations of the collection interfaces. In essence, they are reusable data structures.
- **Algorithms:** These are the methods that perform useful computations, such as searching and sorting, on objects that implement collection interfaces. The algorithms are said to be polymorphic: that is, the same method can be used on many different implementations of the appropriate collection interface. In essence, algorithms are reusable functionality.

Apart from the Java Collections Framework, the best-known examples of collections frameworks are the C++ Standard Template Library (STL) and Smalltalk's collection hierarchy. Historically, collections frameworks have been quite complex, which gave them a reputation for having a steep learning curve.

### Benefits of the Java Collections Framework

The Java Collections Framework provides the following benefits:

- **Reduces programming effort:** By providing useful data structures and algorithms, the Collections Framework frees you to concentrate on the important parts of your program rather than on the low-level "plumbing" required to make it work. By facilitating interoperability among unrelated APIs, the Java Collections Framework frees you from writing adapter objects or conversion code to connect APIs.
- **Increases program speed and quality:** This Collections Framework provides high-performance, high-quality implementations of useful data structures and algorithms. The various implementations of each interface are interchangeable, so programs can be easily tuned by switching collection implementations. Because you're freed from the

drudgery of writing your own data structures, you'll have more time to devote to improving programs' quality and performance.

- Allows interoperability among unrelated APIs: The collection interfaces are the vernacular by which APIs pass collections back and forth. If my network administration API furnishes a collection of node names and if your GUI toolkit expects a collection of column headings, our APIs will interoperate seamlessly, even though they were written independently.
- Reduces effort to learn and to use new APIs: Many APIs naturally take collections on input and furnish them as output. In the past, each such API had a small sub-API devoted to manipulating its collections. There was little consistency among these ad hoc collections sub-APIs, so you had to learn each one from scratch, and it was easy to make mistakes when using them. With the advent of standard collection interfaces, the problem went away.
- Reduces effort to design new APIs: This is the flip side of the previous advantage. Designers and implementers don't have to reinvent the wheel each time they create an API that relies on collections; instead, they can use standard collection interfaces.
- Fosters software reuse: New data structures that conform to the standard collection interfaces are by nature reusable. The same goes for new algorithms that operate on objects that implement these interfaces.

### **Why the name collection framework?**

**Framework** :- Framework is a semi finished reusable application which provides Some common low level services and that can be customized according to our Requirements.

**Java.util package** classes are provides some low level services with well defined Data structures to solve collection heterogeneous dynamic number of object's As a single unit. Due to this reason java.util package classes are called collection f/w.

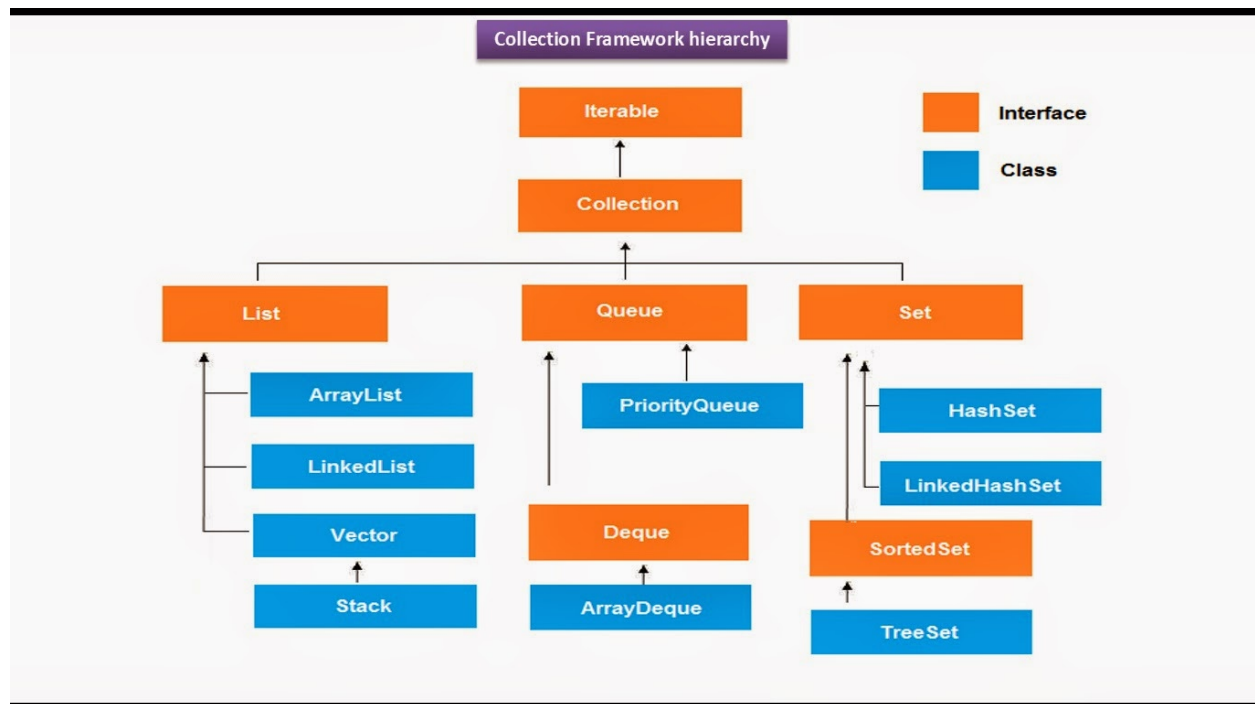
All the collection classes are implementing from java.io.Serializable so the collection Object along with all it's internal object's can be stored in a local file system(OR) can be sent across n/w to remote computer. Here Rule is Object's stored in collection are Also should be serializable types to store collection in file.

### **In How many formats can we collect Object's?**

We can collect Objects in 2 ways

1. In Array Format. In this format object does not identity
2. In(key,value) pair format. In this format object has identity

Collection is an interface which can be used to represent a group of individual objects as a single entity, whereas collections are a utility class to define several utility methods for collection object.



In the above hierarchy, vector and stack classes are available since java 1.0, LinkedHashSet class is available since java 1.4 queue is available since java 5, and NavigableSet is available since java 6, and all other remaining classes and interfaces are available since java 1.2 version.

### List interface:

- It is the child interface of collection.
- If we want to represent a group of individual objects where insertion order is preserved and duplicate objects are allowed, then we should go for list.
- We can differentiate duplicate objects and we can maintain insertion order by using index, hence index played very important role in list.

### ArrayList class:

- ArrayList class extends AbstractList class and implements List interface.
- ArrayList Object allowed duplicate elements.
- ArrayList Object maintains insertion order.
- ArrayList Object is not synchronized.

- ArrayList Object accepts null value.

#### **Note :-**

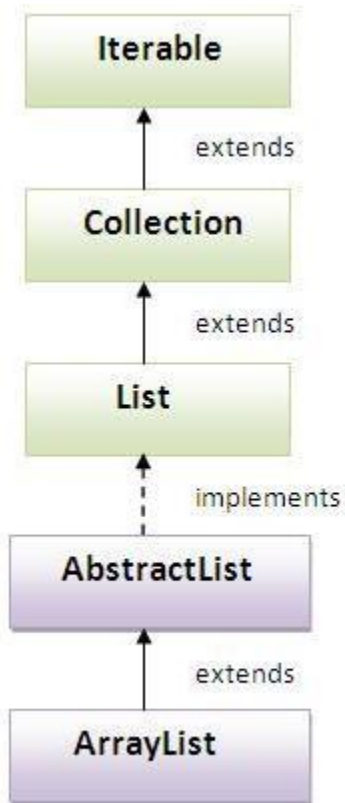
Usually we can use collections to hold and transfer the data across network. To support this requirement every collection class already implements serializable and clonable interfaces.

Array list classes and vector classes implements RandomAccess interface so that We can access any Random element with the same speed.

Hence if our frequent operation is retrieval operation ArrayList is the best choice.

ArrayList is not a good choice if you want to perform insertion (or) deletion in the middle . Because internally several shift operations are required.

#### **Hierarchy of ArrayList class:**



#### **ArrayList class Constructors :-**

##### **1) ArrayList():**

Creates an empty arraylist obj with an initial capacity of 10(ten).

If the ArrayList object reaches its max capacity a new ArrayList object will be created with

**NewCapacity = currentCapacity\*3/2+1**

## 2) ArrayList(int initialCapacity)

Creates an empty ArrayList object with specified initial capacity.

## 3) ArrayList(Collection c)

Creates an equivalent ArrayList object for the given collection

**Ex:**

```
import java.util.*;

class ArrayListDemo{

    public static void main(String args[]){

        ArrayList al=new ArrayList();

        al.add("Ravi");

        al.add("Vijay");

        al.add("Ravi");//duplicated String object

        al.add("Ajay");

        al.add(null);

        System.out.println(al);

    }

}
```

## Vector class:

- The underlying data structure is resizable array or growable array.
- Insertion order is preserved
- Duplication objects are allowed.
- Null insertion possible
- Implements Serializable, Cloneable and RandomAccess interfaces.
- Every method present in Vector is synchronized and hence Vector object is Thread Safe
- Best suitable if our frequent operation is retrieval
- Worst choice if our frequent operations are insertion or deletion in the middle.

## ArrayList , Vector:

- We should choose these two classes to store elements in indexed order and to retrieve them randomly, it means retrieving the element directly without retrieving (n-1) elements, because these two classes are subclasses of RandomAccess interface.



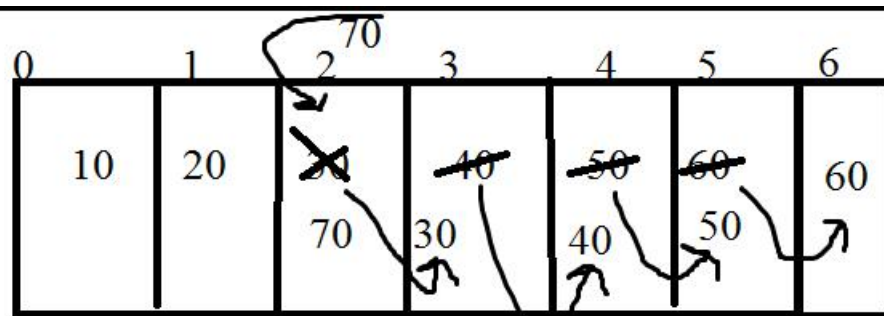
### Functionalities of these two classes:

1. Duplicate objects are allowed in indexed order
2. Heterogeneous objects are allowed
3. Insertion order is preserved
4. Implemented datastructure is growable array.
5. Null insertion is possible, more than one null is allowed.
6. Initial capacity is 10, incremental capacity is double for Vector and ArrayList used below formula( $\text{currentCapacity} * 3/2 + 1$ ). whenever the collection object size reaches its max capacity, that class internal API creates new collection object based on its incremental capacity value. Then in new collection object old collection object elements are copied.

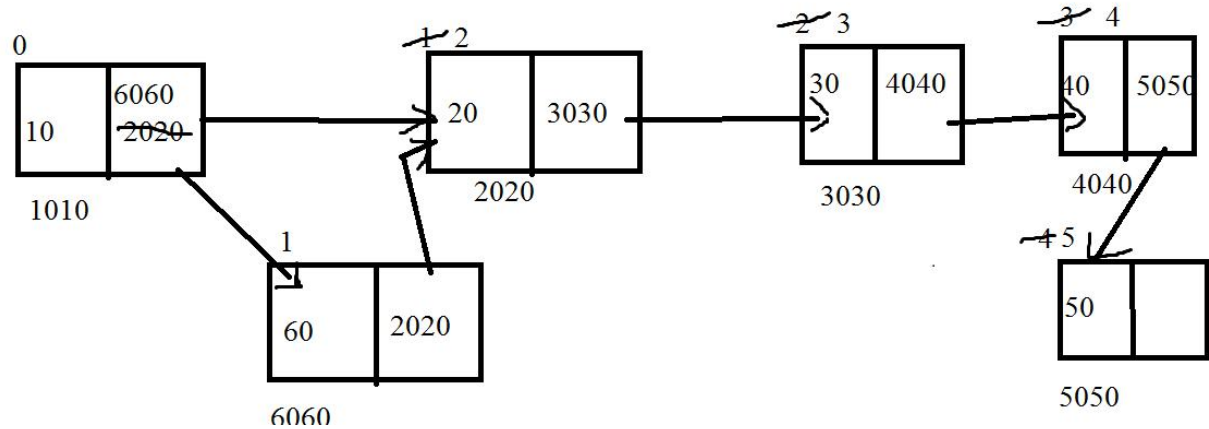
**Vector object is thread-safe:** It means synchronized object so multiple threads **cannot** modify this object concurrently. It is best suitable in multithreaded model application to get desired result. But in single thread model application it gives poor performance, activity in single thread model application. To solve this performance issue in collection framework ArrayList class is given.

- ArrayList object is not thread-safe it means it is not synchronized object. It is best suitable in multithreaded model application, also in multithreaded model application it you ensure there is not data corruption.

**Limitation:** Inserting and removing elements in middle is costlier operation. Because for every insert operation elements must move to right and for every delete operation elements must move to left from that location.



To solve this issue we must choose LinkedList, It gives high Performance In inserting (OR) deleting elements in Middle. It Internally uses Linked List Data structure, so there will not be any data movements, instead we will have Only changing links ,and indexes



### LinkedList class:

- LinkedList uses doubly linked list to store the elements. It extends the AbstractList class and implements List and Deque interfaces.
- LinkedList object allows duplicate elements.
- LinkedList object maintains insertion order.
- LinkedList Object is not synchronized.
- LinkedList implements Serializable , clonable interfaces but not RandomAccess.
- LinkedList best suitable our frequent operation is Insertion and deletion in the middle. Linked List is Worst choice if our frequent operation is retrievable operation.

```
import java.util.*;

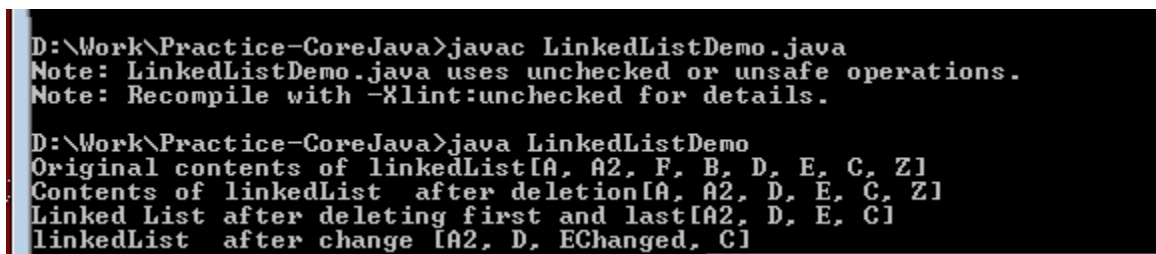
class LinkedListDemo {
    public static void main(String args[]) {
        // create a linked list
        LinkedList linkedList = new LinkedList();
        // add elements to the linked list
        linkedList .add("F");
        linkedList .add("B");
        linkedList .add("D");
        linkedList .add("E");
        linkedList .add("C");
    }
}
```

```

linkedList .addLast("Z");
linkedList .addFirst("A");
linkedList .add(1, "A2");
System.out.println("Original contents of linkedList : " + linkedList );
// remove elements from the linked list
linkedList .remove("F");
linkedList .remove(2);
System.out.println("Contents of linkedList  after deletion: "+ linkedList );
// remove first and last elements
linkedList .removeFirst();
linkedList .removeLast();
System.out.println("ll after deleting first and last: "+ linkedList );
// get and set a value
Object val = linkedList .get(2);
linkedList .set(2, (String) val + " Changed");
System.out.println("linkedList  after change: " + linkedList );
}
}

```

Output :-



```

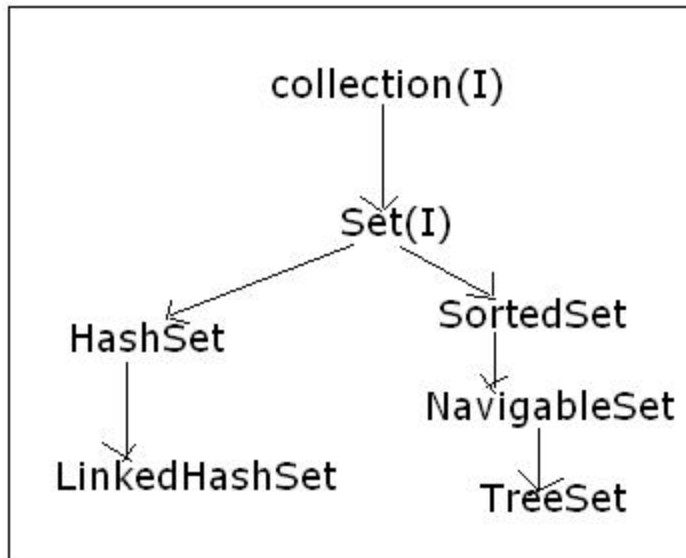
D:\Work\Practice-CoreJava>javac LinkedListDemo.java
Note: LinkedListDemo.java uses unchecked or unsafe operations.
Note: Recompile with -Xlint:unchecked for details.

D:\Work\Practice-CoreJava>java LinkedListDemo
Original contents of linkedList[A, A2, F, B, D, E, C, Z]
Contents of linkedList  after deletion[A, A2, D, E, C, Z]
Linked List after deleting first and last[A2, D, E, C]
linkedList  after change [A2, D, EChanged, C]

```

## Set

- It is the child interface of collection.
- If we want to represent a group of individual objects where duplicates aren't allowed and insertion order isn't preserved then we should go for Set.



- Set Interface doesn't contain any new methods we have to use only collection interface methods

#### HashSet:

- The underlying data structure is "Hashtable".
- Insertion order is n't follows.
- Duplicate objects are n't allowed and if we are trying to insert duplicate object we don't get any compile time or runtime exceptions, simple add() method returns false.
- Heterogeneous objects are allowed.
- null insertion is possible
- HashSet implements serializable and cloneable interface.
- If our frequent operation is SearchOperation then HashSet is best choice.

#### Constructor:

1. **HashSet h=new HashSet();**

Creates an empty HashSet object with the default initial capacity is 16 and default fillratio is ".75"

2. **HashSet h=new HashSet(int initialcapacity);**

Creates an empty HashSet object with the specified initial capacity and default fillratio 0.75%.

3. **HashSet h=new HashSet(int initialcapacity,float fillratio);**
4. **HashSet h=new HashSet(Collection c);**

**Ex:**

```
import java.util.HashSet;

public class HashSetDemo {

    public static void main(String[] args) {
        HashSet hs=new HashSet();
        hs.add("a");
        hs.add("b");
        hs.add("c");
        hs.add("d");
        hs.add(null);
        hs.add(10);
        hs.add(null);//false
        System.out.println(hs);
    }
}
```

**Output:**

[null, d, b, c, a, 10]

**LinkedHashSet:**

- It is the child class of HashSet
- It is exactly same as HashSet(including constructor and method) except the following differences.

### HashSet

The underlying data structure is Hashtable

Insertion order isn't preserved

### LinkedHashSet

The underlying data structure is the combination of the Hashtable and LinkedList.

Insertion order is preserved.

Introduced in 1.2 version

Introduced in 1.4 version.

**Ex:write a sample program using LinkedHashSet**

```
import java.util.HashSet;
import java.util.LinkedHashSet;
public class LinkedHashSetDemo {
    public static void main(String[] args) {
        LinkedHashSet hs=new LinkedHashSet();
        hs.add("a");
        hs.add("b");
        hs.add("c");
        hs.add("d");
        hs.add(null);
        hs.add(10);
        hs.add(null);//false
        System.out.println(hs);
    }
}
```

**Output:**

[a, b, c, d, null, 10]

**sortedSet:**

- It is the child interface of Set
- If we want to represent a group of individual objects according to some sorting order without duplicates, then we should go for SortedSet.
- SortedSet interface defines the following methods.

**TreeSet:**

- Underlying data structure is BalancedTree.
- Insertion order isn't preserved and it is based on the some SortingOrder.
- Null insertion is possible (only once) when it is the first element.

- Heterogeneous objects aren't allowed and if we are trying to insert the heterogeneous objects we will get the RuntimeException saying ClassCastException.

### **Constructor:**

1. **TreeSet t=new TreeSet();**

If creates an empty TreeSet object where all objects will be inserted according to Natural sorting order.

2. **TreeSet t=new TreeSet(Comparator c);**

Creates an empty TreeSet object where all objects will be inserted according to customized Sorting order, which is the described by comparator object.

3. **TreeSet t=new TreeSet(SortedSet c);**

4. **TreeSet t=new TreeSet(Collection c);**

### **Null acceptance:**

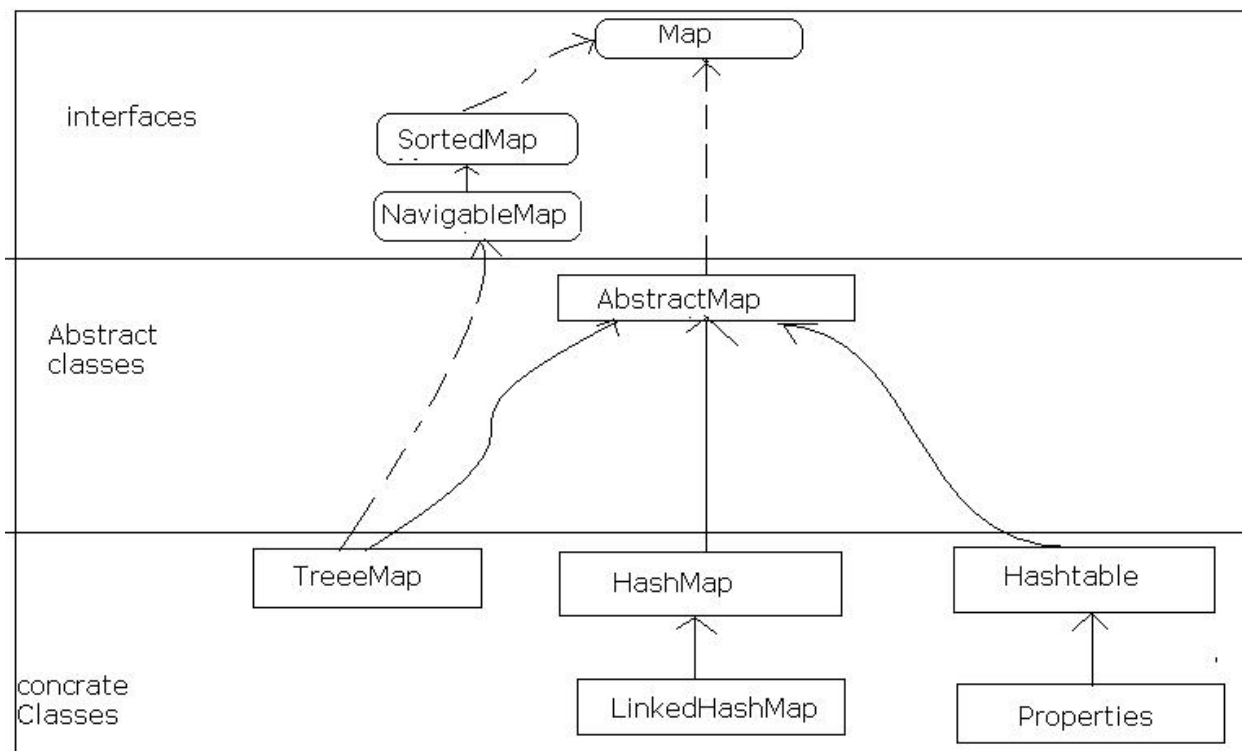
For the nonempty TreeSet, if we are trying to inserting the null and we will get NullPointerException

- For the empty TreeSet, as the first element null insertion is possible. But after inserting that null if we are trying to insert any other element then we will get NullPointerException.

**From java 7 empty tree set also not allows the null value.**

### **Map hierarchy:**

Map hierarchy classes are used to collect elements in (key,value)pair format. In a map, keys should be unique and values can be duplicated. Each (key,value) is called an entry . In map by default entries are not sorted.



### Map interface:

- Map is n't child interface of Collection.
- If we want to represent a group of objects as key,value pairs then we should go for Map.
- Both key and value are objects only.
- Duplicate keys aren't allowed but the values can be duplicated.
- In map Each key,value pair is called as one entry.

### Entry interface:

#### Entry is an Inner interface of Map

Each key,value pair is called as one entry .Hence map is considered as a group of Entries.

Without existing map object there is no chance of existing entry object.

Methods of Entry interface:-

**public Object getKey()**

**public Object getValue()**



### HashMap:

- The underlying datastructure is Hashtable.
- Insertion order is n't follows.
- Duplicate key's aren't allowed but the values can be duplicated.
- Heterogeneous objects are allowed for both keys and values
- Null keys are allowed for key (only once)and for the values (any no of times)

### HashMap

### Hashtable

No method is synchronized

Every method is synchronized

HashMap object can be accessed by multiple threads and hence it isn't Thread safe.

Hashtable object can be accessed by only one thread at a time and hence it is thread safe.

Relatively performance is high.

Relatively performance is low.

We can insert null for both key and values

Null isn't allowed for both key and values otherwise we will get NullPointerException

Introduced in 1.2 version  
It is non legacy.

Introduced in 1.0 version and it is legacy.

### Constructor:

1. **HashMap m=new HashMap();**

Creates an empty hashmap() object with default initial capacity is 16 and default fillratio is .75

2. **HashMap m=new HashMap(int initialcapacity);**
3. **HashMap m=new HashMap(int initialcapacity,float fillratio);**

#### 4. **HashMap m=new HashMap(Map m);**

**Refer the examples in the classroomExampleProgramms folder**

#### **LinkedHashMap:**

- It is exactly same as HashMap except the following differences.
- The underlying datastructure is Hashtable,the underlying datastructure is a combination of Hashtable +LinkedList.
- In LinkedHashMap Insertion is preserved ,but in the HashMap Insertion Order is not preserved

#### **IdentityHashMap :**

Identity HashMap is the Implemented class of Map interface.Identity HashMap is same as HashMap Except the following differences. For identifying duplicate keys In the case of HashMap JVM Uses equals() method, Where as In the case of Identity HashMap JVM Uses the “==” (double equal) operator.

#### **WeakHashMap:-**

WeakHashMap is the Implemented class of Map interface.WeakHashMap is same as HashMap Except the following differences.

- In case of HashMap an Object is not eligible for garbage collection if it is associated with HashMap . Even though it doesnt have any external references. i.e, HashMap dominates garbage collector.
- But in case of WeakHashMap ,if an Object is not having any external references then it is always eligible for garabage collector even though it is associated with WeakHashMap . i.e, garbage collector dominates WeakHashMap

#### **Sorted Map interface :**

- The Collection Framework provides a special Map interface for maintaining elements in a sorted order called **SortedMap** . SortedMap is the sub interface of Map.
- If we want to represent a group of key value pairs according to some sorting order of keys then we should go for Sorted Map.
- In Sorted Map we can perform sorting only based on the keys but not values.

#### **SortedMap interface Specifc Methods**

Object firstKey()

object lastKey()

SortedMap headMap(Object key)

SortedMap tailMap(Object key)

Sotredmap subMap(Object key1,Object key2)

Comparator comparator()

**Note :** Working with a **SortedMap** is just similar to a **SortedSet** except, the sort is done on the map keys.

### **NavigableMap**

The java.util.NavigableMap(java 1.6) interface is a subinterface of the **java.util.SortedMap** interface. It has a few extensions to the SortedSet which makes it possible to navigate the map. I will take a closer look at these navigation methods in this text.

The java.util package only has one implementation of the NavigableMap interface: java.util.TreeMap.

**Here is a list of the topics covered in this text:**

1. descendingKeySet() and descendingMap()
2. headMap(), tailMap() and subMap()
3. ceilingKey(), floorKey(), higherKey() and lowerKey()
4. ceilingEntry(), floorEntry(), higherEntry() and lowerEntry()
5. pollFirstEntry() and pollLastEntry()

### **descendingKeySet() and descendingMap() :**

The first interesting navigation methods are the descendingKeySet() and descendingMap() methods. The descendingKeySet() method returns a NavigableSet in which the order of the elements is reversed compared to the original key set. The returned "view" is backed by the original NavigableSet key set, so changes to the descending set are also reflected in the original set. However, you should not remove elements directly from the key set. Use the Map.remove() method instead.

**Here is a simple example:**

```
NavigableSet reverse = map.descendingKeySet();
```

The `descendingMap()` method returns a `NavigableMap` which is a view of the original `Map`. The order of the elements in this view map is reverse of the order of the original map. Being a view of the original map, any changes to this view is also reflected in the original map.

**Here is a simple example:**

```
NavigableMap descending = map.descendingMap();
```

### **headMap(), tailMap() and subMap() :**

The `headMap()` method returns a view of the original `NavigableMap` which only contains elements that are "less than" the given element. Here is an example:

```
NavigableMap original = new TreeMap();
original.put("1", "1");
original.put("2", "2");
original.put("3", "3");
//this headmap1 will contain "1" and "2"
SortedMap headmap1 = original.headMap("3");
//this headmap2 will contain "1", "2", and "3" because "inclusive"=true
NavigableMap headmap2 = original.headMap("3", true);
```

The `tailMap()` method works the same way, except it returns all elements that are **higher** than the given parameter element.

The `subMap()` allows you to pass two parameters demarcating the boundaries of the view map to return. Here is an example:

```
NavigableMap original = new TreeMap();
original.put("1", "1");
original.add("2", "2");
original.add("3", "3");
original.add("4", "4");
original.add("5", "5");
//this submap1 will contain "3", "3"
SortedMap submap1 = original.subMap("2", "4");
//this submap2 will contain ("2", "2") ("3", "3") and ("4", "4") because
```

```
// fromInclusive=true, and toInclusive=true
NavigableMap submap2 = original.subMap("2", true, "4", true);
ceilingKey(), floorKey(), higherKey() and lowerKey() :
The ceilingKey() method returns the least (smallest) key in this map that is greater than or equal
to the element passed as parameter to the ceilingKey() method. Here is an example:
NavigableMap original = new TreeMap();
original.put("1", "1");
original.put("2", "2");
original.put("3", "3");

//ceilingKey will be "2".
Object ceilingKey = original.ceilingKey("2");
//floorKey will be "2".
Object floorKey = original.floorKey("2");
```

The floorKey() method does the opposite of ceilingKey()

The higherKey() method returns the least (smallest) element in this map that is greater than (not equal too) the element passed as parameter to the higherKey() method. **Here is an example:**

```
NavigableMap original = new TreeMap();
original.put("1", "1");
original.put("2", "2");
original.put("3", "3");
//higherKey will be "3".
Object higherKey = original.higherKey("2");
//lowerKey will be "1"
Object lowerKey = original.lowerKey("2");
```

The lowerKey() method does the opposite of the higherKey() method.

### **ceilingEntry(), floorEntry(), higherEntry(), lowerEntry() :**

The NavigableMap also has methods to get the entry for a given key, rather than the key itself. These methods behave like the ceilingKey() etc. methods, except they return an Map.Entry instead of the key object itself.

A Map.Entry maps a single key to a single value.

Here is a simple example. For more details, check out the JavaDoc.

```
NavigableMap original = new TreeMap();
original.put("1", "1");
original.put("2", "2");
original.put("3", "3");
//higherEntry will be ("3", "3").
Map.Entry higherEntry = original.higherEntry("2");
//lowerEntry will be ("1", "1")
Map.Entry lowerEntry = original.lowerEntry("2");
```

### **pollFirstEntry() and pollLastEntry() :**

The pollFirstEntry() method returns and removes the "first" entry (key + value) in the NavigableMap or null if the map is empty. The pollLastEntry() returns and removes the "last" element in the map or null if the map is empty. "First" means smallest element according to the sort order of the keys. "Last" means largest key according to the element sorting order of the map.

### **Here are two examples:**

```
NavigableMap original = new TreeMap();
original.put("1", "1");
original.put("2", "2");
original.put("3", "3");
//first is ("1", "1")
Map.Entry first = original.pollFirstEntry();
//last is ("3", "3")
Map.Entry last = original.pollLastEntry();
```

### **TreeMap:**

- The underlying datastructure is RED-BLACK Tree
- Duplicate keys aren't allowed but the values can be duplicated.
- Insertion order isn't preserved and all entries will be inserted according to some sorting order of keys
- If we are depending on default natural sorting order then compulsory keys should be homogeneous and comparable otherwise we will get ClassCastException.

But if we are defining our own sorting by comparator then the keys need n't be comparable and homogeneous.

- There are no restriction for values, they can be heterogeneous and non comparable.
- Upto java 6 version For empty TreeMap as first entry with null key is allowed but after inserting the entry .if we are trying to insert any other entry we will get "NullPointerException".
- Upto java 6 version For non empty TreeMap if we are trying to insert an entry with null key then we will get NPE.
- But From java 7 version null key's are not allowed .
- There are no restrictions for null values.

### **Constructors:**

1. **TreeMap tm=new TreeMap()**

For default natural sorting order.

2. **TreeMap tm=new TreeMap(Comparator c)**

For customized sorting order.

3. **TreeMap tm=new TreeMap(SortedMap c)**

Interconversion of sortedMap

4. **TreeMap tm=new TreeMap(Map c)**

Interconversion between Map.

### **Hashtable:**

- Underlying datastructure for Hashtable is Hashtable
- Duplicate keys aren't allowed but values can be duplicated
- Insertion order isn't preserved and it is based on Hashcode of the keys.
- Heterogeneous objects are allowed for both keys and values.
- Null insertion isn't possible fore both key and value otherwise we will get NPE.
- Every method present in Hashtable is synchronized and hence Hashtable object is Threadsafe.

### **Constructor:**

- 1) Hashtable h =new Hashtable();

- Creates an empty Hashtable object with default initial capacity "11" and default fillration .75.

1. Hashtable h =new Hashtable(int initialcapacity);
2. Hashtable h =new Hashtable(int initialcapacity,float fillratio);
3. Hashtable h =new Hashtable(Map);

### **HashtableDemo.java**

```
import java.util.Enumeration;
import java.util.Hashtable;
import java.util.Set;

public class HashtableDemo {
    public static void main(String args[]) {
        // Creating Hashtable for example
        Hashtable companies = new Hashtable();
        // to put object into Hashtable
        companies.put("Google", "United States");
        companies.put("Nokia", "Finland");
        companies.put("Sony", "Japan");

        // to get Object from Hashtable
        companies.get("Google");
        // Use containsKey(Object) method to check if an Object exists as key in
        // hashtable
        System.out.println("Does hashtable contains Google as key: "
            + companies.containsKey("Google"));
        /* just like containsKey(), containsValue returns true if hashtable
        contains specified object as value*/
        System.out.println("Does hashtable contains Japan as value: "
            + companies.containsValue("Japan"));
        // Hashtable elements() return enumeration of all hashtable values
        Enumeration enumeration = companies.elements();
        while (enumeration.hasMoreElements()) {
            System.out.println("hashtable values: " + enumeration.nextElement());
        }
        // use size() method to find size of hashtable in Java
```



```

System.out.println("Size of hashtable in Java: " + companies.size());
// use keySet() method to get a Set of all the keys of hashtable
Set hashtableKeys = companies.keySet();
// get enumeration of all keys by using method keys()
Enumeration hashtableKeysEnum = companies.keys();
}
}

```

Output :

```

F:\corejava>javac HashtableDemo.java
Note: HashtableDemo.java uses unchecked or unsafe operations.
Note: Recompile with -Xlint:unchecked for details.

F:\corejava>java HashtableDemo
Does hashtable contains Google as key: true
Does hashtable contains Japan as value: true
hashtable values: Finland
hashtable values: United States
hashtable values: Japan
Size of hashtable in Java: 3

F:\corejava>

```

### Properties:

- In our program if any thing which changes frequently( like username, password etc) are never recommended to hardcode those values in our java program because for every change we have to recompile ,rebuilt, redeploy application and sometimes we have to restart server is also required which creates a big business impact to the client.
- Such type of values we have to place inside the properties file and we have to read those values from the properties file into the java application.
- The main advantage of this application is if any change in properties file to reflect those changes just redeployment is enough which won't create any business object to hold properties from the properties file.
- Properties class is the child class of the Hashtable and both key and values should be String type.

### **Constructor:**

1. **Properties p =new Properties();**

### **Methods:**

1. String getProperty(String pname);

Return the value associated with specified property.

2. String setProperty(string pname,string pvalue)

To add a new property

3. Enumeration propertyNames();
4. Void load (InputStream is)

To load properties from properties file into java properties object.

5. void store(OutputStream os,String comment) to store the properties from java application into Properties file.

**Ex:**

```
import java.io.FileInputStream;
import java.io.FileNotFoundException;
import java.io.FileOutputStream;
import java.io.IOException;
import java.util.Properties;
class propertiesDemo
{
    public static void main(String[] args) throws IOException {
        Properties p=new Properties();
        FileInputStream fis=new FileInputStream("src/ics.properties");
        p.load(fis);
        System.out.println(p);
        String s=p.getProperty("ics");
        System.out.println(s);
        p.setProperty("ics", "9090");
        FileOutputStream fos=new FileOutputStream("abc.properties");
        p.store(fos, "update is successfully");
    }
}
```

**Output**

abc.properties

#update is successfully

#Sat Aug 24 21:00:20 GMT+05:30 2013

laxman=core java

ics=9090

## Generics

### **Introduction**

#### **case(i):**

once we created an array for a type we can add only that particular Type of objects. By mistake if we are trying to add any other type we will get Compile Time Error.

Hence we can give gurantee for the type of elements in array and Hence arrays are Type -safe.

EX:-

```
String s[]=new String[100];
s[0]="AAA";
s[1]="BBB";
s[2]=10;//Compile Time Error
```

Hence we can give the gurantee that String array can contain only String Type of Objects.

But Collections are not Type Safe,i.e we can't give the gurantee for the type of elements in collection.

For Example if our programming requirement is to hold String objects and we can choose ArrayList,by mistake if you are trying to add any other type we won't get any compiletime Error. but there may be a chance of failing the app at runtime.

**EX:-**

```
ArrayList l=new ArrayList();
l.add("ICS");
l.add("ICS");
l.add((100));
String s1=(String)l.set(0,"aaa");
String s2=(String)l.set(1,"sss");
String s3=(String)l.set(2,"ddd"); //R:Ex:--ClassCastException
```

There is no gurantee the collection can hold Type of objects.

Hence with respect to type, Arrays are Type safe to use.

Where as Collection are not safe to use.

#### **case(ii):**

while reteriving Array elements it is not required to perform TypeCasting . we can assign Array Elements directly to the Corresponding Type variables.

```
String[] s=new String[10];
s[0]="ICS";
s[1]="ICS";
String s=s[0];//ICS
type casting is not required
```

But while retrieving elements from collection compulsory. we should perform Type casting. otherwise we will get CompileTimeError.

1)Hence TypeCasting is the biggest headck in collections. To overcome the above problems of collections (Typesafety,Type casting) sun micro systems has introduced Generics concept in java 5 version.

To Hold only String Type of Objects a Generic version of ArrayList can declare as follows  
ArrayList<String> l=new ArrayList<String>();  
basetype parametertype for this arraylist we can hold only String type of objects, other wise we will get CE:

2)Hence generics can provide TypeSafty to collections .

At time of reterival it is not required to perform TypeCasting.  
Hence main objective of generics are

- 1) To provide TypeSafty for colletions
- 2) To reslove TypeCasting problem

### **Conclusion--I:**

**Ex:**

- 1) AL<String> l=new AL<String>();//valid
- 2) Al<Object> l=new AL<String>();//Compile Time Error

### **conclusion-II:**

At the type parameter we can use any class (or) Interface name and we can't provide primitive Type.

- 1)AL<Integer> l=new AL<Integer>();
- 2) AL<int> l=new AL<int> ();//Compile Time Error

### **Generic class:**

non generic version of ArrayList class is declared as follows  
class ArrayList

```
{
add(Object o)
Object get(int index)
}
```

the add() method can take object as argument .

Hence we can add any type of object to ArrayList . Due to this we are not getting typesafety. The return type of get(-)method is object hence at the time of retrieval compulsory we should perform type casting.

But from 1.5 version a generic version of ArrayList class is define as follows.

```
class ArrayList<T>{
add(T t)
T get(int index)
}
```

Based on our requirement "T" will be replaced with corresponding Type.

-->Through Generics we can associate type parameter for the class such type of parametrized classes are called Generic classes.

--> we can create our own Generic classes also

**Ex:-**

```
class Bank<T>
{
}
Bank<Icici> b=new Bank<Icici>();
```

**Bounded Types:**

We can bound the type parameters for a particular rangesuch types are called bounded types.

**Ex:-**

```
1)
class Test<T>
{ //UnBoundedType
}
```

as the type parameter we can pass any type

There are no restrictions.

Hence it is unbound type.

**Ex:**

```
class Test<T extends X> //x is class/interface
```

```
{  
}
```

If X is a class then as the type parameter we can pass either X type (or) it's sub classes.

If X is an interface then as the type parameter we can pass either Xtype (or) it's implementations classes.

**Ex:-**

```
class Test<T extends Number>{  
}  
Test<Integer> t=new Test<Integer>();  
Test<String> t=new Test<String>();//Compile Time Error
```

we can bound the type parameters only by using extends keyword. and we cannot bound by using implements and super keywords.

**Creation of our own generics:**

```
class Test<T>{  
    T obj;  
    Test(T obj) {  
        this.obj=obj;  
    }  
    public void show(){  
        System.out.println(" The Type of Obj is :"+obj.getClass().getName());  
    }  
    public T getObj(){  
        return obj;  
    }  
}
```

```
class GenericsDemo{  
    public static void main(String[] args) {  
        Test<String> g1=new Test<String>("ICS");  
        g1.show();//the type obj is java.lang.String  
        System.out.println(g1.getObj());//ICS  
        Test<Integer> g2=new Test<Integer>(10);  
        g2.show();//java.lang.Integer  
        System.out.println(g2.getObj());//10  
        Test<Double> g3=new Test<Double>(10.55);  
        g3.show();//java.lang.Double  
        System.out.println(g3.getObj());//10.55  
    }  
}
```

Type Inference for Generic instance creation (or) Diamond Syntax:-

You can replace the type arguments required to invoke the constructor of a generic class with an empty set of type parameters (< >).

This pair of angle brackets is called the diamond.

For example, consider the following variable declarations:

```
ArrayList<String> al=new ArrayList<>();// valid from java 7
```

```
ArrayList<Integer> al=new ArrayList<>();// valid from java 7
```

## INTRODUCTION TO THREAD AND THE NEED

### Threads:

Threads are sometimes called *lightweight processes*. Both processes and threads provide an execution environment, but creating a new thread requires fewer resources than creating a new process.

Threads exist within a process every process has at least one. Threads share the process's resources, including memory and open files. This makes for efficient, but potentially problematic, communication.

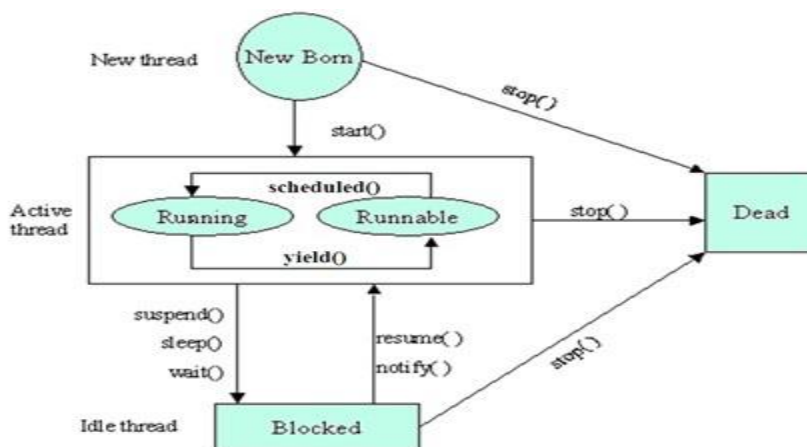
Multithreaded execution is an essential feature of the Java platform. Every application has at least one thread or several, if you count "system" threads that do things like memory management and signal handling. But from the application programmer's point of view, you start with just one thread, called the *main thread*. This thread has the ability to create additional threads.

**Note: At a time one thread is executed only.**

### Life cycle of a Thread (Thread States) :

A thread can be in one of the five states. According to sun, there is only 4 states in **thread life cycle in java** new, runnable, non-runnable and terminated. There is no running state.

But for better understanding the threads, here explaining it in the 5 states.



The life cycle of the thread in java is controlled by JVM. The java thread states are as follows:



1. New
2. Runnable
3. Running
4. Non-Runnable (Blocked)
5. Terminated

### **1) New :**

The thread is in new state if you create an instance of Thread class but before the invocation of start() method.

### **2) Runnable**

The thread is in runnable state after invocation of start() method, but the thread scheduler has not selected it to be the running thread.

### **3) Running**

The thread is in running state if the thread scheduler has selected it.

### **4) Non-Runnable (Blocked)**

This is the state when the thread is still alive, but is currently not eligible to run.

### **5) Terminated**

A thread is in terminated or dead state when its run() method exits.

### **How to create thread :**

There are two ways to create a thread:

1. By extending Thread class
2. By implementing Runnable interface.

#### **1) By Extending Thread Class:**

Thread class provide constructors and methods to create and perform operations on a thread.

Thread class extends Object class and implements Runnable interface.

Commonly used Constructors of Thread Class as Follows ,

- Thread()

- Thread(String name)
- Thread(Runnable r)
- Thread(Runnable r,String name)

**Ex:**

```
public class ThreadDemo extends Thread{
```

```
    public void run() {
```

```
        System.out.println("thread is running...");
```

```
    }
```

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

```
        ThreadDemo thread = new ThreadDemo();
```

```
        thread.start();
```

```
    }
```

```
}
```

**Output :** thread is running...

**Note : Thread class constructor** allocates a new thread object. When we are creating object of ThreadDemo class, our class constructor is invoked (provided by Compiler) from where Thread class constructor is invoked (by super() as first statement). So our ThreadDemo class object is ' thread object ' now.

2) By Implementing Runnable Interface :

The Runnable interface should be implemented by any class whose instances are intended to be executed by a thread. Runnable interface has only one method named run().

**public void run():** is used to perform action for a thread.

**Ex :**

```
public class ThreadDemoOfRunnable implements Runnable {
```

```
    public void run() {
```

```

System.out.println(" Thread is running...");
}
public static void main(String args[]) {
ThreadDemoOfRunnable demoObj = new ThreadDemoOfRunnable();
Thread thread = new Thread(demoObj);
thread.start();
}
}

```

**Output :** Thread is running...

**Note :** If we are not extending the Thread class, our class object would not be treated as a thread object. So we need to explicitly create Thread class object and passing the object of our class that implements Runnable so that our class run() method may execute.

### **Priority of a Thread (Thread Priority):**

Each thread have a priority. Priorities are represented by a number between 1 and 10. In most cases, thread scheduler schedules the threads according to their priority (known as preemptive scheduling). But it is not guaranteed because it depends on JVM specification that which scheduling it chooses.

**There are three Constants Defined in Thread Class , they are**

**public static int MIN\_PRIORITY;**

**public static int NORM\_PRIORITY;**

**public static int MAX\_PRIORITY;**

Default priority of a thread is **5 (NORM\_PRIORITY)**. The value of **MIN\_PRIORITY is 1** and the value of **MAX\_PRIORITY is 10**.

### **Need of Thread**

- Multithreading is a technique that allows a program or a process to execute many tasks concurrently (at the same time and parallel). It allows a process to run its tasks in parallel mode on a single processor system.
- In the multithreading concept, several multiple lightweight processes are run in a single process/task or program by a single processor. For Example, When you use a word

processor you perform a many different tasks such as printing, spell checking and so on. Multithreaded software treats each process as a separate program.

- In Java, the Java Virtual Machine (JVM) allows an application to have multiple threads of execution running concurrently. It allows a program to be more responsive to the user. When a program contains multiple threads then the CPU can switch between the two threads to execute them at the same time.