Java Tutorial

Preface

This is the first article of w3resource's Java programming tutorial. The aim of this tutorial is to make beginners conversant with Java programming language.

Introduction to Java programming language

Today Java programming language is one of the most popular programming languages which is used in critical applications like stock market trading system on BSE, banking systems or android mobile application.

Java was developed by James Gosling from Sun Microsystems in 1995 as an object-oriented language for general-purpose business applications and for interactive, Web-based Internet applications. The goal was to provide a platform-independent alternative to C++. In other terms, it is architecturally neutral, which means that you can use Java to write a program that will run on any platform or device (operating system). Java program can run on a wide variety of computers because it does not execute instructions on a computer directly. Instead, Java runs on a Java Virtual Machine (JVM).

Java is a general-purpose programming language that's used in all industries for almost any type of application. If you master it, your chances of getting employed as a software developer will be higher than if you specialize in some domain-specific programming languages. The Java language is object-oriented (OO), which allows you to easily relate program constructs to objects from the real world.

History of Java Releases

This program runs fine under GNU Gcc compiler. We have tested this on a Ubuntu Linux system. But if you are using any other compiler like Turbo C++, the program needs to be modified to be executed successfully. For the sake of simplicity, we have not included that additional stuff here.

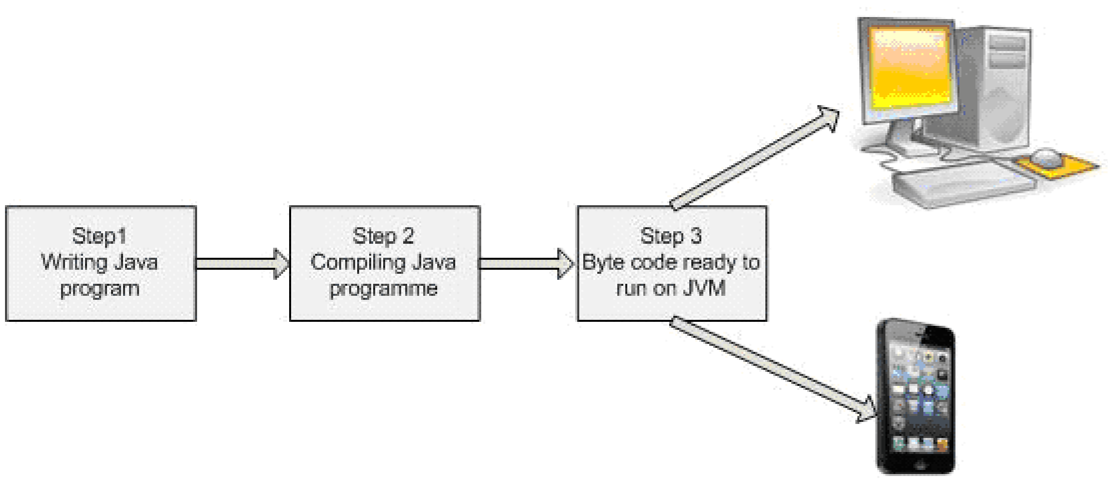
Java Version/CodeName	Release Date	Important Features/Code Name
JDK 1.0 (Oak)	23rd Jan 1996	Initial release

JDK 1.1	19th Feb 1997	Reflection, JDBC, Inner classes, RMI
J2SE 1.2 (Playground)	8th Dec 1998	Collection, JIT, String memory map
J2SE 1.3 (Kestrel)	8th May 2000	Java Sound, Java Indexing, JNDI
J2SE 1.4 (merlin)	6th Feb 2002	Assert, regex, exception chaining,
J2SE 5.0 (Tiger)	30th Sept 2004	Generics, autoboxing, enums
Java SE 6.0 (Mustang)	11th Dec 2006	JDBC 4.0, java compiler API, Annotations
Java SE 7.0 (Dolphin)	28th July 2011	String in switch-case, Java nio, exception handling new way

The Java Program Life Cycle

Java requires the source code of your program to be compiled first. It gets converted to either machine-specific code or a byte code that is understood by some run-time engine or a java virtual machine.

Not only will the program be checked for syntax errors by a Java compiler, but some other libraries of Java code can be added (linked) to your program after the compilation is complete (deployment stage).



Step1 : Create a source document using any editor and save file as .java (e.g. abc.java)

Step2 : Compile the abc.java file using “javac” command or eclipse will compile it automatically.

Step3 : Byte Code (abc.class) will be generated on disk.

Step4 : This byte code can run on any platform or device having JVM (java.exe convert byte code in machine language)

Let’s get familiar with various terminologies used by java programmers.

JDK (Java Development Kit): JDK contains JRE along with various development tools like Java libraries, Java source compilers, Java debuggers, bundling and deployment tools

JRE (Java Runtime Environment): It is part of JDK but can be used independently to run any byte code (compiled java program). It can be called as JVM implementation.

JVM (Java Virtual Machine): ‘JVM’ is software that can be ported to various hardware platforms. JVM will become an instance of JRE at runtime of java program. Byte codes are the machine language for the JVM. Like a real computing machine, JVM has an instruction set which manipulates various memory areas at run time. Thus for different hardware platforms, one has corresponding the implementation of JVM available as vendor supplied JREs.

Java API (Application Programming Interface) : Set of classes’ written using Java programming language which runs on JVM. These classes will help programmers by providing standard methods like reading from the console, writing to the console, saving objects in data structure etc.

Advantages of Java programming language

- Built-in support for multi-threading, socket communication, and memory management (automatic garbage collection).
- Object Oriented (OO).
- Better portability than other languages across operating systems.
- Supports Web-based applications (Applet, Servlet, and JSP), distributed applications (sockets, RMI, EJB etc.) and network protocols (HTTP, JRMP etc.) with the help of extensive standardized APIs (Application Programming Interfaces).

Summary

- Java is platform independent programming language which means compile once and run anywhere.
- Java provides built-in functionality for Thread, collection, File IO etc.

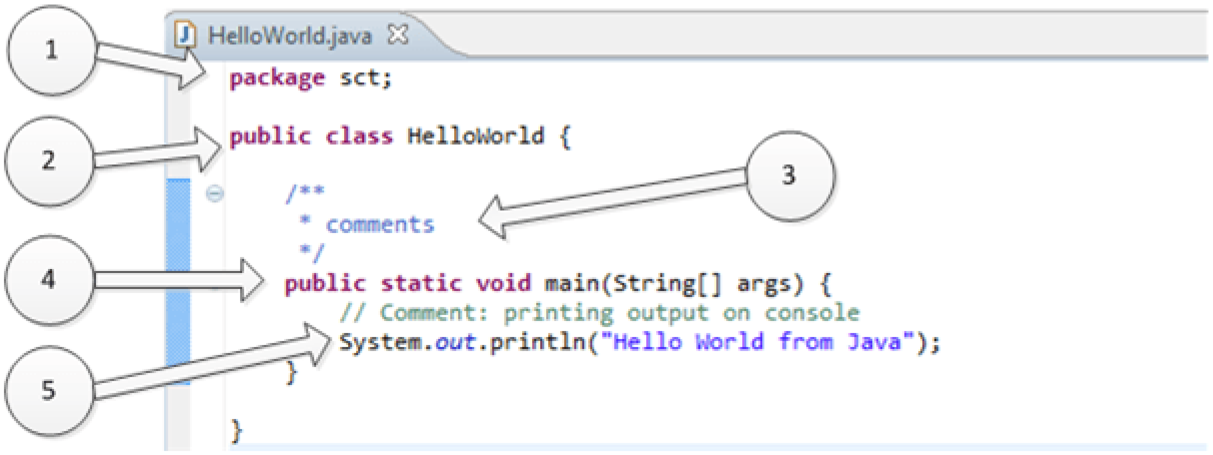
- The Java language is object-oriented (OO) programming language which will allow the programmer to relate java domain objects with real life objects.

In next session, we will discuss how to install JDK, Eclipse (IDE) and the basic structure of Java program. Compiling, running and debugging java program.

Java Program Structure

Description

Let's use the example of HelloWorld Java program to understand structure and features of the class. This program is written on few lines, and its only task is to print "Hello World from Java" on the screen. Refer the following picture.



```java
package sct;

public class HelloWorld {

    /**
     * comments
     */
    public static void main(String[] args) {
        // Comment: printing output on console
        System.out.println("Hello World from Java");
    }

}
```

1."package sct":

It is package declaration statement. The package statement defines a namespace in which classes are stored. The package is used to organize the classes based on functionality. If you omit the package statement, the class names are put into the default package, which has no name. Package statement cannot appear anywhere in the program. It must be the first line of your program or you can omit it.

2."public class HelloWorld":

This line has various aspects of java programming.

a. public: This is access modifier keyword which tells compiler access to class. Various values of access modifiers can be public, protected,private or default (no value).

b. class: This keyword used to declare a class. Name of class (HelloWorld) followed by this keyword.

3. Comments section:

We can write comments in java in two ways.

a. Line comments: It starts with two forward slashes (//) and continues to the end of the current line. Line comments do not require an ending symbol.

b. Block comments start with a forward slash and an asterisk (/*) and end with an asterisk and a forward slash (*/).Block comments can also extend across as many lines as needed.

4. “public static void main (String [ ] args)”:

Its method (Function) named main with string array as an argument.

a. public: Access Modifier

b. static: static is a reserved keyword which means that a method is accessible and usable even though no objects of the class exist.

c. void: This keyword declares nothing would be returned from the method. The method can return any primitive or object.

d. Method content inside curly braces. { } asdfla;sd

5. System.out.println("Hello World from Java") :

a. System: It is the name of Java utility class.

b. out:It is an object which belongs to System class.

c. println: It is utility method name which is used to send any String to the console.

d. “Hello World from Java”: It is String literal set as argument to println method.

More Information regarding Java Class:

- Java is an object-oriented language, which means that it has constructs to represent objects from the real world. Each Java program has at least one class that knows how to do certain things or how to represent some type of object. For example, the simplest class, HelloWorld,knows how to greet the world.

- Classes in Java may have methods (or functions) and fields (or attributes or properties).
- Let’s take example of Car object which has various properties like color, max speed etc. along with it has functions like run and stop. In Java world we will represent it like below:

```
package sct;

public class Car {

private String color;

private int maxSpeed;

public String carInfo(){

return color + " Max Speed:-" + maxSpeed;

}

//This is constructor of Car Class

Car(String carColor, int speedLimit){

this.color = carColor;

this.maxSpeed =speedLimit;

}

}
```

- Lets make a class named CarTestwhich will instantiate the car class object and call carInfo method of it and see output.

```
package sct;

//This is car test class to instantiate and call Car objects.

public class CarTest {

public static void main(String[] args) {

Car maruti = new Car("Red", 160);

Car ferrari = new Car ("Yellow", 400);

System.out.println(maruti.carInfo());

System.out.println(ferrari.carInfo());

}

}
```

Output of above CarTest java class is as below. We can run CarTest java program because it has main method. Main method is starting point for any java program execution. Running a program means telling the Java VIrtual Machine (JVM) to "Load theclass, then start executing its main () method. Keep running 'til all thecode in main is finished."

```
Red-Make Year:1999 Max Speed:-160
Yellow-Make Year:2012 Max Speed:-400
```

Common Programming guidelines:

- Java identifiers must start with a letter, a currency character ($), or a connecting character such as the underscore (_). Identifiers cannot start with a number. After first character identifiers can contain any combination of letters,currency characters, connecting characters, or numbers. For example,
  - int variable1 = 10 ; //This is valid
  - int 4var =10 ; // this is invalid, identifier can't start with digit.
- Identifiers, method names, class names are case-sensitive; var and Var are two different identifiers.
- You can't use Java keywords as identifiers, Below table shows a list of Java keywords.

abstract	Boolean	break	byte	case	catch
char	Class	const	continue	default	do
double	Clse	extends	final	finally	float
for	Goto	if	implements	import	instanceof
int	Interface	long	native	new	package
private	Protected	public	return	short	static
strictfp	Super	switch	synchronized	this	throw
throws	Transient	try	void	volatile	while

assert	Enum				

- Classes and interfaces: The first letter should be capitalized, and if several words are linked together to form the name, the first letter of the inner words should be uppercase (a format that's sometimes called "camelCase").
- Methods: The first letter should be lowercase, and then normal camelCaserules should be used.For example:
    - getBalance
    - doCalculation
    - setCustomerName
- Variables: Like methods, the camelCase format should be used, starting with a lowercase letter. Sun recommends short, meaningful names, which sounds good to us. Some examples:
    - buttonWidth
    - accountBalance
    - empName
- Constants: Java constants are created by marking variables static and final. They should be named using uppercase letters with underscore characters as separators:
    - MIN_HEIGHT
- There can be only one public class per source code file.
- Comments can appear at the beginning or end of any line in the source code file; they are independent of any of the positioning rules discussed here.
- If there is a public class in a file, the name of the file must match the name of the public class. For example, a class declared as “public class Dog { }” must be in a source code file named Dog.java.
- To understand more about access modifiers applicable to classes, methods, variables in access modifier tutorial.

We will understand more about constructors, access modifiers in coming tutorials. The source code of sample discussed attached here to run directly on your system.

Summary

- Java program has a first statement as package statement (if package declared).

- One Java file can have multiple classes declared in it but only one public class which should be same as file name.
- Java class can have instance variable such as methods, instance variable.

In next session we will discuss Setup Classpath/ Environment variable and Compiling, running and debugging Java programs.

Java Primitive data type

Description

Not everything in Java is an object. There is a special group of data types (also known as primitive types) that will be used quite often in programming. For performance reasons, the designers of the Java language decided to include these primitive types. Java determines the size of each primitive type. These sizes do not change from one operating system to another. This is one of the key features of the language that makes Java so portable. Java defines eight primitive types of data: byte, short, int, long, char, float, double, and boolean. The primitive types are also commonly referred to as simple types which can be put in four groups

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

Let’s discuss each in details:

byte

The smallest integer type is byte. It has a minimum value of -128 and a maximum value of 127 (inclusive). The byte data type can be useful for saving memory in large arrays, where the memory savings actually matters. Byte variables are declared by use of the byte keyword. For example, the following declares and initialize byte variables called b:

```
byte  b =100;
```

short:

The short data type is a 16-bit signed two's complement integer. It has a minimum value of -32,768 and a maximum value of 32,767 (inclusive). As with byte, the same guidelines apply: you can use a short to save memory in large arrays, in situations where the memory savings actually matters. Following example declares and initialize short variable called s:

```
short  s =123;
```

int:

The most commonly used integer type is int. It is a signed 32-bit type that has a range from –2,147,483,648 to 2,147,483,647. In addition to other uses, variables of type int are commonly employed to control loops and to index arrays. This data type will most likely be large enough for the numbers your program will use, but if you need a wider range of values, use long instead.

```
int  v = 123543;
int  calc = -9876345;
```

long:

long is a signed 64-bit type and is useful for those occasions where an int type is not large enough to hold the desired value. It has a minimum value of -9,223,372,036,854,775,808 and a maximum value of 9,223,372,036,854,775,807 (inclusive). Use of this data type might be in banking application when large amount is to be calculated and stored.

```
long  amountVal = 1234567891;
```

float:

Floating-point numbers, also known as real numbers, are used when evaluating expressions that require fractional precision. For example interest rate calculation or calculating square root. The float data type is a single-precision 32-bit IEEE 754 floating point. As with the recommendations for byte and short, use a float (instead of double) if you need to save memory in large arrays of floating point numbers. The type float specifies a single-precision value that uses 32 bits of storage. Single precision is faster on some processors and takes half as much space as double precision. The declaration and initialization syntax for float variables given below, please note “f” after value initialization.

```
float  intrestRate = 12.25f;
```

double:

Double precision, as denoted by the double keyword, uses 64 bits to store a value. Double precision is actually faster than single precision on some modern processors that have been optimized for high-speed mathematical calculations. All transcendental math functions, such as sin( ), cos( ), and sqrt( ), return double values. The declaration and initialization syntax for double variables given below, please note “d” after value initialization.

```
double  sineVal = 12345.234d;
```

boolean:

The boolean data type has only two possible values: true and false. Use this data type for simple flags that track true/false conditions. This is the type returned by all relational operators, as in the case of a < b. boolean is also the type required by the conditional expressions that govern the control statements such as if or while.

```
boolean  flag = true;
booleanval  = false;
```

char:

In Java, the data type used to store characters is char. The char data type is a single 16-bit Unicode character. It has a minimum value of '\u0000' (or 0) and a maximum value of '\uffff' (or 65,535 inclusive). There are no negative chars.

```
char ch1 = 88; // code for X
char  ch2 = 'Y';
```

Primitive Variables can be of two types

(1) Class level (instance) variable:

It's not mandatory to initialize Class level (instance) variable. If we do not initialize instance variable compiler will assign default value to it. Generally speaking, this default will be zero or null, depending on the data type. Relying on such default values, however, is generally considered bad coding practice.The following chart summarizes the default values for the above data types.

Primitive Data Type	**Default Value 3.6**
byte	0
short	0

int	0
long	0L
float	0.0f
double	0.0d
char	'\u0000'
boolean	false

(2) Method local variable:

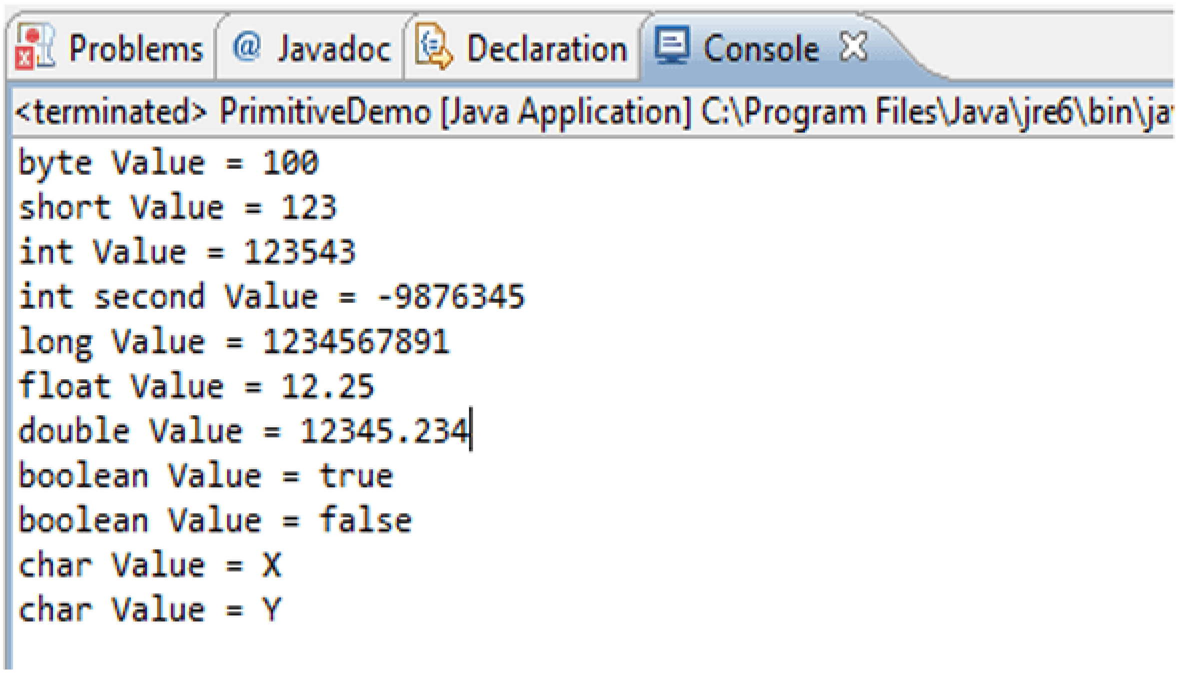
Method local variables have to be initialized before using it. The compiler never assigns a default value to an un-initialized local variable. If you cannot initialize your local variable where it is declared, make sure to assign it a value before you attempt to use it. Accessing an un-initialized local variable will result in a compile-time error.

Let’s See simple java program which declares, initialize and print all of primitive types.

Java class PrimitiveDemo as below:

```
package primitive;

public class PrimitiveDemo {

        public static void main(String[] args) {

                byte b =100;

                short s =123;

                int v = 123543;

                int calc = -9876345;

                long amountVal = 1234567891;

                float intrestRate = 12.25f;

                double sineVal = 12345.234d;

                boolean flag = true;

                boolean val = false;
                char ch1 = 88; // code for X

                char ch2 = 'Y';

                System.out.println("byte Value = "+ b);

                System.out.println("short Value = "+ s);

                System.out.println("int Value = "+ v);

                System.out.println("int second Value = "+ calc);

                System.out.println("long Value = "+ amountVal);

                System.out.println("float Value = "+ intrestRate);

                System.out.println("double Value = "+ sineVal);

                System.out.println("boolean Value = "+ flag);

                System.out.println("boolean Value = "+ val);

                System.out.println("char Value = "+ ch1);

                System.out.println("char Value = "+ ch2);

        }

}
```

Output of above PrimitiveDemo class as below:



```
<terminated> PrimitiveDemo [Java Application] C:\Program Files\Java\jre6\bin\ja
byte Value = 100
short Value = 123
int Value = 123543
int second Value = -9876345
long Value = 1234567891
float Value = 12.25
double Value = 12345.234
boolean Value = true
boolean Value = false
char Value = X
char Value = Y
```

Summary of Data Types

Primitive Type	**Size**	**Minimum Value**	**Maximum Value**	**Wrapper Type**
char	16-bit	Unicode 0	Unicode 216-1	Character
byte	8-bit	-128	+127	Byte
short	16-bit	-215 (-32,768)	+215-1 (32,767)	Short
int	32-bit	-231 (-2,147,483,648)	+231-1 (2,147,483,647)	Integer
long	64-bit	-263 (-9,223,372,036,854,775,808)	+263-1 (9,223,372,036,854,775,807)	Long
float	32-bit	Approx range 1.4e-045 to 3.4e+038		Float
double	64-bit	Approx range 4.9e-324 to 1.8e+308		Double
boolean	1-bit	true or false		Boolean

Summary

- Java has group of variable types called primitive data type which are not object.
- Primitive types are categorized as Integer, Floating point, characters and boolean.
- Primitive types help for better performance of the application.

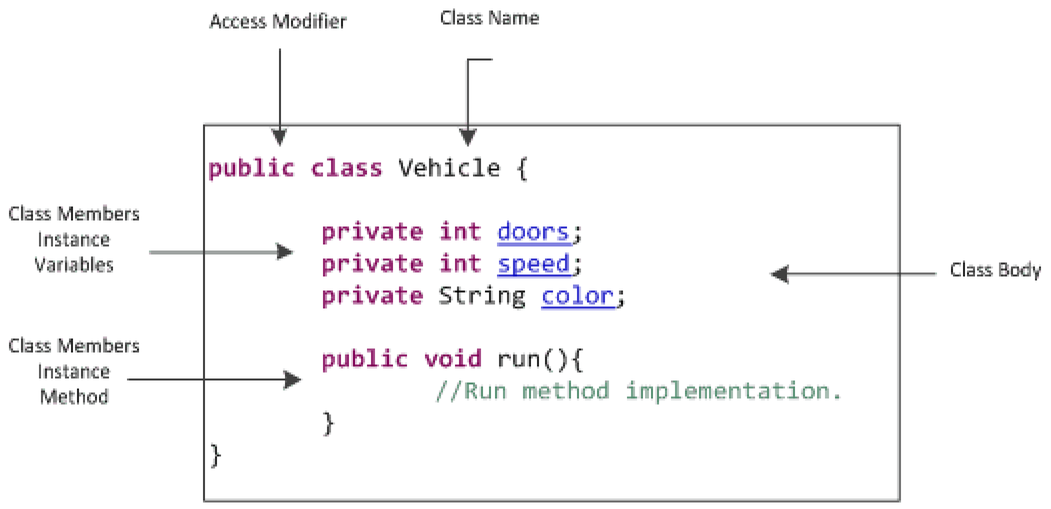
Java Class, methods, instance variables

Java Declaration and Access Modifiers

All computer programs consist of two elements: code and data. Furthermore, a program can be conceptually organized around its code or around its data. The first way is called process-oriented model. Procedural languages such as C employ this model to considerable success. To manage increasing complexity the second approach called object-oriented programming was conceived. An object-oriented program can be characterized as data controlling access to the code. Java is object-oriented programming language. Java classes consist of variables and methods (also known as instance members). Java variables are two types either primitive types or reference types. First, let us discuss how to declare a class, variables and methods then we will discuss access modifiers.

Declaration of Class:

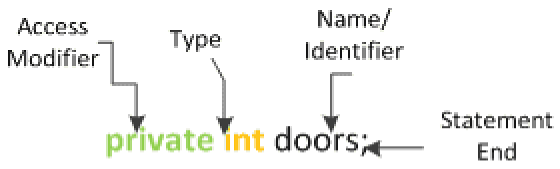
A class is declared by use of the class keyword. The class body is enclosed between curly braces { and }. The data or variables, defined within a class are called instance variables. The code is contained within methods. Collectively, the methods and variables defined within a class are called members of the class.


Access Modifier
Class Name
Class Members Instance Variables
Class Members Instance Method
Class Body

```
public class Vehicle {

        private int doors;
        private int speed;
        private String color;

        public void run(){
                //Run method implementation.
        }
}
```

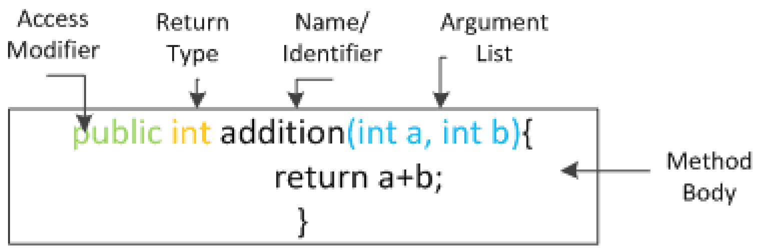
Declaration of Instance Variables :

Variables defined within a class are called instance variables because each instance of the class (that is, each object of the class) contains its own copy of these variables. Thus, the data for one object is separate and unique from the data for another. An instance variable can be declared public or private or default (no modifier). When we do not want our variable's value to be changed out-side our class we should declare them private. public variables can be accessed and changed from outside of the class. We will have more information in OOP concept tutorial. The syntax is shown below.



Declaration of Methods :

A method is a program module that contains a series of statements that carry out a task. To execute a method, you invoke or call it from another method; the calling method makes a method call, which invokes the called method. Any class can contain an unlimited number of methods, and each method can be called an unlimited number of times. The syntax to declare method is given below.

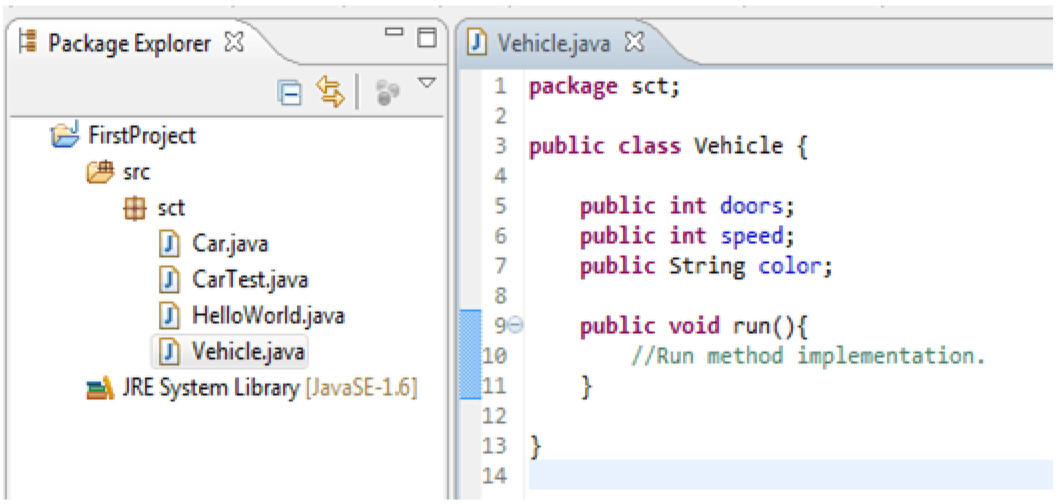


```
public int addition(int a, int b){
        return a+b;
}
```

Access modifiers:

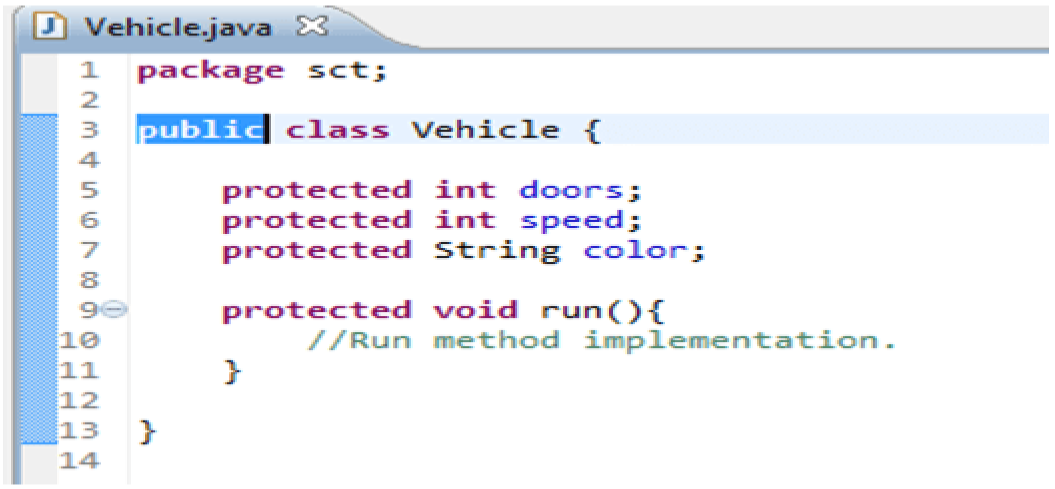
Each object has members (members can be variable and methods) which can be declared to have specific access. Java has 4 access level and 3 access modifiers. Access levels are listed below in the least to most restrictive order.

public: Members (variables, methods, and constructors) declared public (least restrictive) within a public class are visible to any class in the Java program, whether these classes are in the same package or in another package. Below screen shot shows eclipse view of public class with public members.



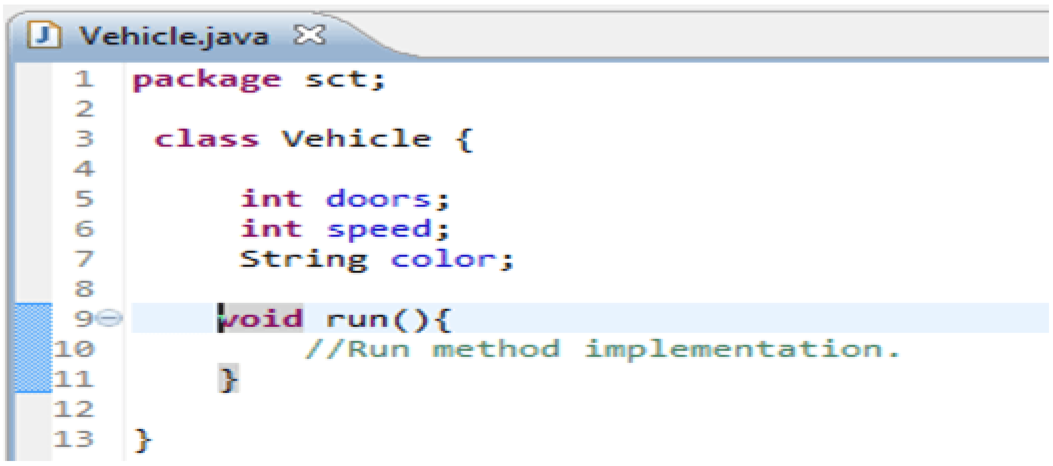
```
package sct;

public class Vehicle {

    public int doors;
    public int speed;
    public String color;

    public void run(){
        //Run method implementation.
    }

}
```

protected: The protected fields or methods, cannot be used for classes and Interfaces. Fields, methods and constructors declared protected in a super-class can be accessed only by subclasses in other packages. Classes in the same package can also access protected fields, methods and constructors as well, even if they are not a subclass of the protected member's class.



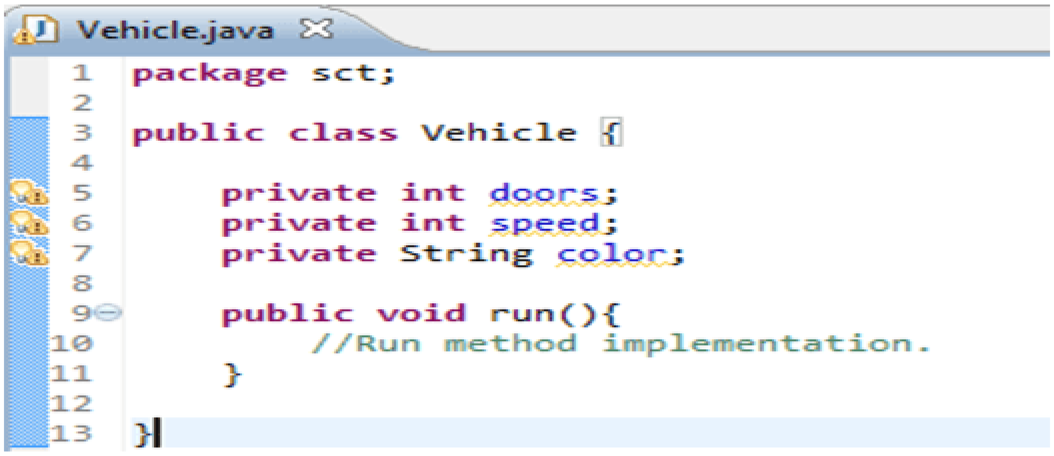
```
package sct;

public class Vehicle {

    protected int doors;
    protected int speed;
    protected String color;

    protected void run(){
        //Run method implementation.
    }

}
```

Default (no value):The default access level is declared by not writing any access modifier at all. Any class, field, method or constructor that has no declared access modifier is accessible only by classes in the same package.



```
package sct;

 class Vehicle {

     int doors;
     int speed;
     String color;

     void run(){
         //Run method implementation.
     }

}
```

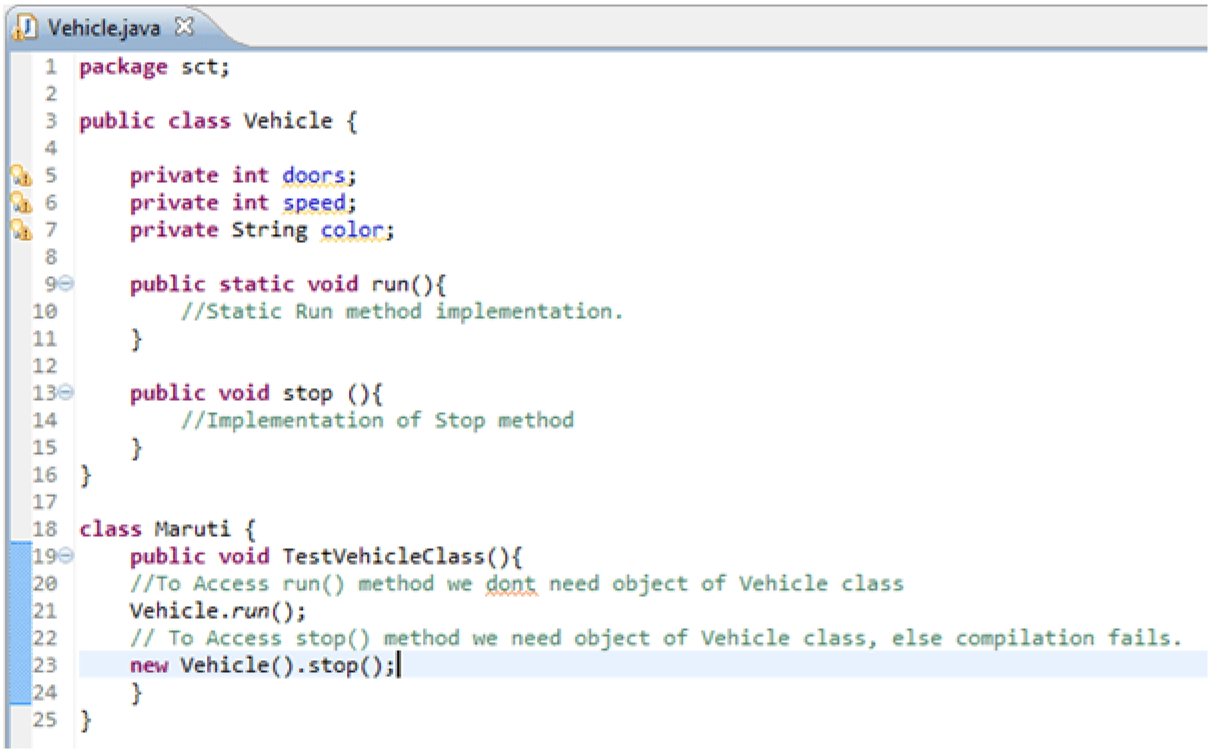
private: The private (most restrictive) modifiers can be used for members but cannot be used for classes and Interfaces. Fields, methods or constructors declared private are strictly controlled, which means they cannot be accessed by anywhere outside the enclosing class.



```
package sct;

public class Vehicle {

    private int doors;
    private int speed;
    private String color;

    public void run(){
        //Run method implementation.
    }

}
```

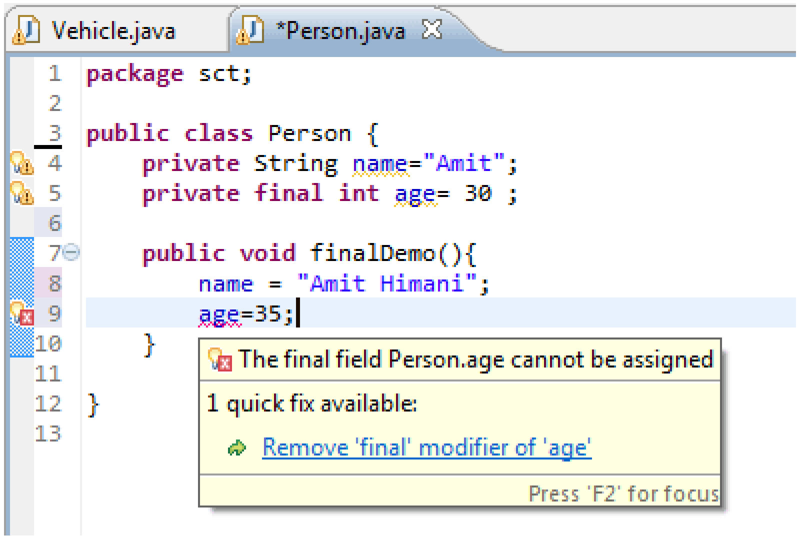
Java has modifiers other than access modifiers listed below:

static: static can be used for members of a class. The static members of the class can be accessed without creating an object of a class. Let's take an example of Vehicle class which has run () as a static method and stop () as a non-static method. In Maruti class we can see how to access static method run () and non-static method stop ().



```
package sct;

public class Vehicle {

    private int doors;
    private int speed;
    private String color;

    public static void run(){
        //Static Run method implementation.
    }

    public void stop (){
        //Implementation of Stop method
    }
}

class Maruti {
    public void TestVehicleClass(){
    //To Access run() method we dont need object of Vehicle class
    Vehicle.run();
    // To Access stop() method we need object of Vehicle class, else compilation fails.
    new Vehicle().stop();
    }
}
```

final: This modifier applicable to class, method, and variables. This modifier tells the compiler not to change the value of a variable once assigned. If applied to class, it cannot be sub-classed. If applied to a method, the method cannot be overridden in sub-class. In below sample, we can see compiler errors while trying to change the value of filed age because it is defined as final while we can change the value of name field.



```
package sct;

public class Person {
    private String name="Amit";
    private final int age= 30 ;

    public void finalDemo(){
        name = "Amit Himani";
        age=35;
    }

}
```

The final field Person.age cannot be assigned

1 quick fix available:

Remove 'final' modifier of 'age'

abstract: There are situations in which you will want to define a superclass that declares the structure of a given abstraction without providing a complete implementation of every method. This modifier is applicable to class and methods only. We will discuss abstract class in detail in separate Tutorial.

Below Table summarizes the access modifiers

Modifier	**class**	**constructor**	**method**	**Data/variables**
Public	Yes	Yes	Yes	Yes
Protected		Yes	Yes	Yes
default	Yes	Yes	Yes	Yes
private		Yes	Yes	Yes
static			Yes	
final	Yes		Yes	

Let’s take first column example to interpret. A “class” can have public, default, final and abstract access modifiers.

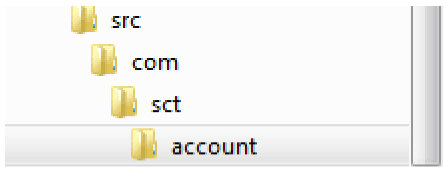
Summary

- Access modifiers help to implement encapsulation principle of object orientation programming.
- Java has 4 access modifiers public, protected, default, private.
- Java has other modifiers like static, final and abstract.

Java Packages

Introduction

A decent sized project can have hundreds of Java classes and you need to organize them in packages (think file directories). This will allow you to categorize files, control access to your classes, and avoid potential naming conflicts. Sometimes you’ll be using third-party libraries of Java classes written in a different department or even outside your firm. To minimize the chances that package names will be the same, it’s common to use so-called reverse domain name conventions. For example, if you work for a company called SCT, which has the website sct.com, you can prefix all package names with com.sct. To place a Java class in a certain package, add the package statement at the beginning of the class (it must be the first non-comment statement in the class).For example, if the class SalesTax has been developed for the Accounting department, you can declare it as follows, which will result in creation of directory structure as below:



```
packagecom.sct.account;

classSalesTax {

// the class body goes here

}
```

Java classes are also organized into packages and the fully qualified name of a class consists of the package name followed by the class name. For example, the full name of the Java class Double isjava.lang.Double, where java.lang is the package name.

To use a public package member from outside its package, you must do one of the following:

- Refer to the member by its fully qualified name: Here no need of import statement but the code will become non-readable due to the longer statement.

```
newpackageName.className().methodName( argument1, argument2);
```

- Import the package member: This way we can import one class from one package. Import statement should be put after package declaration statement.

```
importpackageName.className; newclassName().methodName(argument 1, argument2);
```

- Import the member's entire package: This will help to import all the classes of a particular package.

```
importpackageName.*;
newclassName().methodName(argument1, argument2);
```

We can put our class file on “classpath” to have global access without the need of import but it is not advisable programming way. We have earlier discussed how to create/update “classpath”. We can add our java compiled directory to classpath like below,

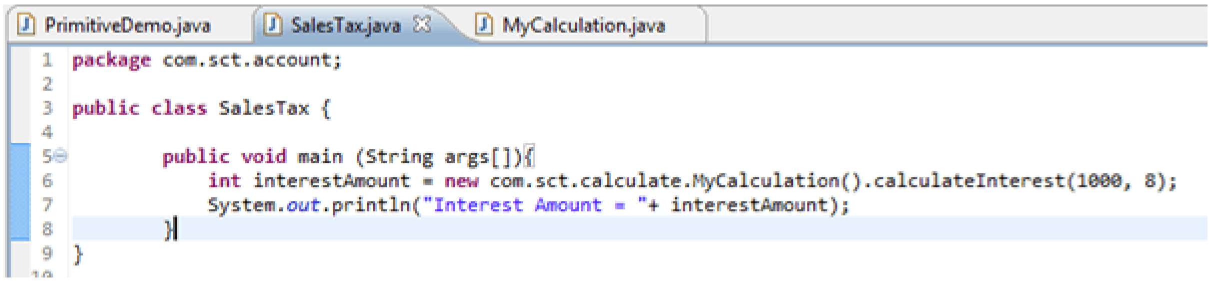
```
 [packageDir]\packageName\ClassName.java
```

Lets assume our SalesTax class needs to use calculateInterest() method of class “MyCalculation” declared in package called “com.sct.calculate”.

```
package com.sct.calculate;

public class MyCalculation

 {

        //This method calculate interst

        public int calculateInterest(int amount, int rate)

         {

          int intrerstAmount = amount * rate/100;

      return intrerstAmount;

     }

 }
```

We can call calculateInterest() method in two way.

1. Calling with the fully qualified name: As seen below we are calling the method with full path.



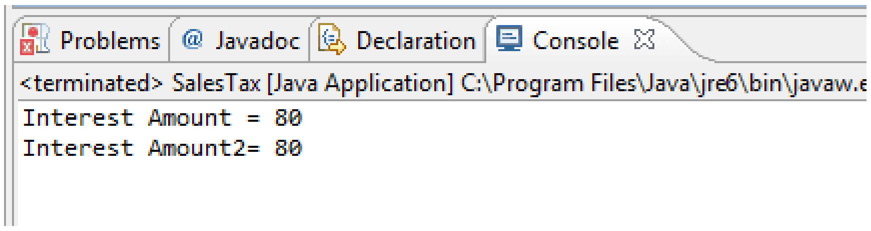
```
 1 package com.sct.account;
 2
 3 public class SalesTax {
 4
 5         public void main (String args[]){
 6             int interestAmount = new com.sct.calculate.MyCalculation().calculateInterest(1000, 8);
 7             System.out.println("Interest Amount = "+ interestAmount);
 8         }
 9 }
```

2. package import and then directly calling the method: Here we have added line number 3 which is import statement and line number 11 will be doing exactly same as line number 8.

```
package com.sct.account;

import com.sct.calculate.*;

public class SalesTax {

                public static void main (String args[]){

                                int interestAmount = new
com.sct.calculate.MyCalculation().calculateInterest(1000, 8);

                                System.out.println("Interest Amount = "+ interestAmount);

                                int intAmount2 = new MyCalculation().calculateInterest(1000,8);

                                System.out.println("Interest Amount2= "+ intAmount2);

                }

}
```

Output of above SalesTax class will be as below

Sample Output:



```
<terminated> SalesTax [Java Application] C:\Program Files\Java\jre6\bin\javaw.e
Interest Amount = 80
Interest Amount2= 80
```

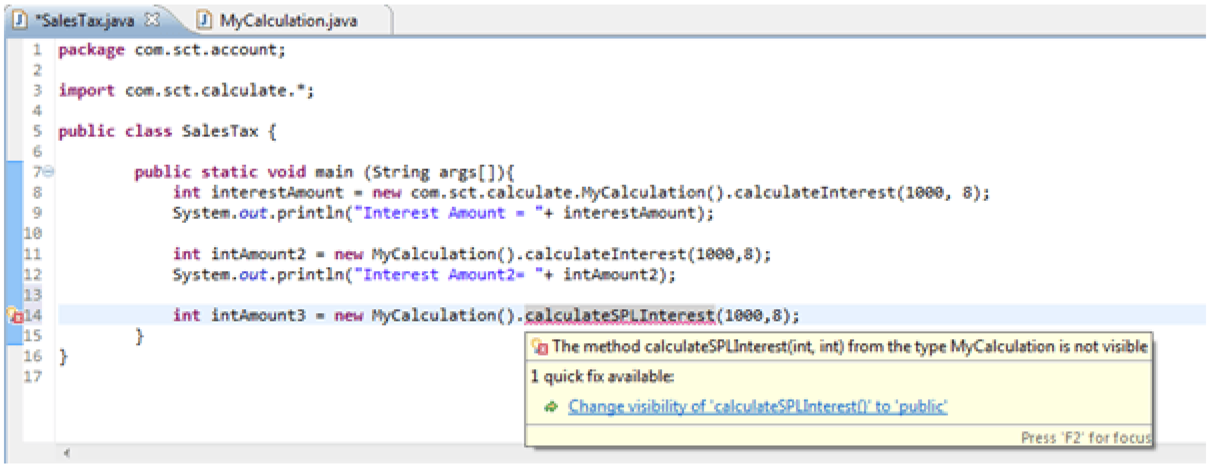
If you need to import several classes from the same package, use the wildcard in the import statement rather than listing each of the classes on a separate line as shown below,

```
package com.sct.account;

import com.sct.calculate.*;

public class SalesTax

{

                public static void main (String args[])

          {

                        int interestAmount = new
com.sct.calculate.MyCalculation().calculateInterest(1000, 8);

                        System.out.println("Interest Amount = "+ interestAmount);

                        int intAmount2 = new MyCalculation().calculateInterest(1000,8);

                        System.out.println("Interest Amount2= "+ intAmount2);

          }

}
```

Package is mainly used to implement Encapsulation principle of Java OOP. Lets take an example. We have requirement to have one method calculateSPLInterest() accessible only in same package ie. “com.sct.calculate”. This might be for bank’s own employee interest calculation. So our “MyCalculation” class will look like below

```
package com.sct.calculate;

public class MyCalculation {

        //This method calculate interst
        public int calculateInterest(int amount, int rate){

        int intrerstAmount = amount * rate/100;

                return intrerstAmount;

        }

        //This is Special Interst Calculation

        protected int calculateSPLInterest (int amount, int rate){

                int intrerstAmount = (amount * rate/100) + 100;

                return intrerstAmount;

        }

}
```

While trying to access this method from outside package, compiler will give error as shown in below screen. This way we can hide particular class member implementation outside package.


*SalesTax.java   MyCalculation.java

```
package com.sct.account;

import com.sct.calculate.*;

public class SalesTax {

        public static void main (String args[]){
            int interestAmount = new com.sct.calculate.MyCalculation().calculateInterest(1000, 8);
            System.out.println("Interest Amount = "+ interestAmount);

            int intAmount2 = new MyCalculation().calculateInterest(1000,8);
            System.out.println("Interest Amount2= "+ intAmount2);

            int intAmount3 = new MyCalculation().calculateSPLInterest(1000,8);
        }
}
```

```
The method calculateSPLInterest(int, int) from the type MyCalculation is not visible
1 quick fix available:
Change visibility of 'calculateSPLInterest()' to 'public'
Press 'F2' for focus
```

Here the programs are small and use a limited number of classes, you might find that you don't need to explore packages at all. But as you begin creating more sophisticated projects with many classes related to each other by inheritance, you might discover the benefit of organizing them into packages.

Packages are useful for several broad reasons:

- They enable you to organize your classes into units. Just as you have folders or directories on your hard disk to organize your files and applications, packages enable you to organize your classes into groups so that you use only what you need for each program.
- They reduce problems with conflicts about names. As the number of Java classes grows, so does the likelihood that you'll use the same class name as another developer, opening up the possibility of naming clashes and error messages if you try to integrate groups of classes into a single program. Packages provide a way to refer specifically to the desired class, even if it shares a name with a class in another package.

- They enable you to protect classes, variables, and methods in larger ways than on a class-by-class basis, as you learned today. You learn more about protections with packages later.
- Packages can be used to uniquely identify your work.

Summary

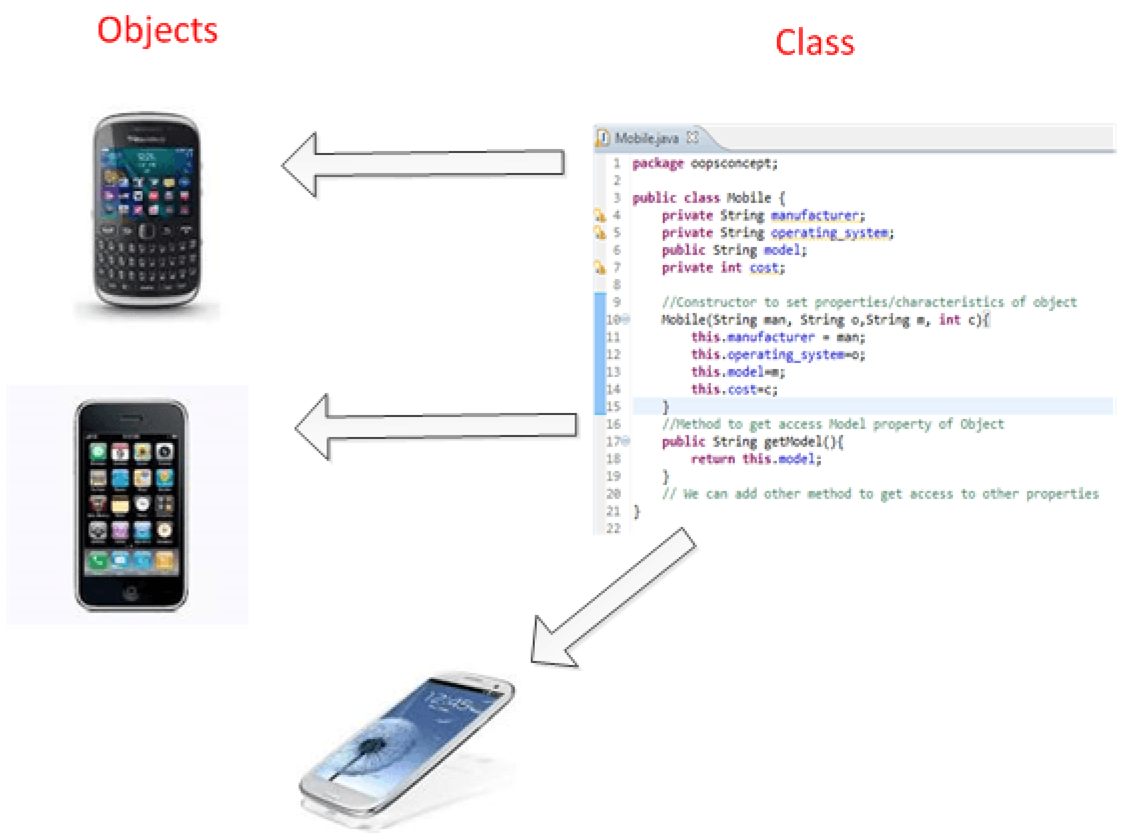
- Java package is a way to organize the classes in a large project and it also helps to in encapsulation implementation.
- To access code residing outside the current package, either import the class or use the fully qualified class name.

Java Object Oriented Programming concepts

Introduction

This tutorial will help you to understand about Java OOP'S concepts with examples. Let's discuss what are the features of Object Oriented Programming. Writing object-oriented programs involves creating classes, creating objects from those classes, and creating applications, which are stand-alone executable programs that use those objects.

A class is a template, blueprint,or contract that defines what an object's data fields and methods will be. An object is an instance of a class. You can create many instances of a class. A Java class uses variables to define data fields and methods to define actions. Additionally,a class provides methods of a special type, known as constructors, which are invoked to create a new object. A constructor can perform any action, but constructors are designed to perform initializing actions, such as initializing the data fields of objects.



```java
package oopsconcept;

public class Mobile {
    private String manufacturer;
    private String operating_system;
    public String model;
    private int cost;

    //Constructor to set properties/characteristics of object
    Mobile(String man, String o,String m, int c){
        this.manufacturer = man;
        this.operating_system=o;
        this.model=m;
        this.cost=c;
    }
    //Method to get access Model property of Object
    public String getModel(){
        return this.model;
    }
    // We can add other method to get access to other properties
}
```

Objects are made up of attributes and methods. Attributes are the characteristics that define an object; the values contained in attributes differentiate objects of the same class from one another. To understand this better let's take the example of Mobile as an object. Mobile has characteristics like a model, manufacturer, cost, operating system etc. So if we create "Samsung" mobile object and "IPhone" mobile object we can distinguish them from characteristics. The values of the attributes of an object are also referred to as the object's state.

There are three main features of OOPS.

1) Encapsulation

2) Inheritance

3) Polymorphism

Let's we discuss the features in details.

Encapsulation

Encapsulation means putting together all the variables (instance variables) and the methods into a single unit called Class. It also means hiding data and methods within an Object. Encapsulation provides the security that keeps data and methods safe from inadvertent changes. Programmers sometimes refer to encapsulation as using a “black box,” or a device that you can use without regard to the internal mechanisms. A programmer can access and use the methods and data contained in the black box but cannot change them. Below example shows Mobile class with properties, which can be set once while creating object using constructor arguments. Properties can be accessed using getXXX() methods which are having public access modifiers.

```
package oopsconcept;

public class Mobile {

        private String manufacturer;

        private String operating_system;

        public String model;

        private int cost;

        //Constructor to set properties/characteristics of object

        Mobile(String man, String o,String m, int c){

                this.manufacturer = man;

                this.operating_system=o;

                this.model=m;

                this.cost=c;

        }

        //Method to get access Model property of Object

        public String getModel(){

                return this.model;

        }

        // We can add other method to get access to other properties

}
```

Inheritance

An important feature of object-oriented programs is inheritance—the ability to create classes that share the attributes and methods of existing classes, but with more specific features. Inheritance is mainly used for code reusability. So you are making use of already written the classes and further extending on that. That why we discussed the code reusability the concept. In general one line definition, we can tell that deriving a new class from existing class, it’s called as Inheritance. You can look into the following example for inheritance concept. Here we have Mobile class extended by other specific class like Android and Blackberry.

```
package oopsconcept;

public class Android extends Mobile{

                //Constructor to set properties/characteristics of object

                Android(String man, String o,String m, int c){

                                super(man, o, m, c);

                        }

                //Method to get access Model property of Object

                public String getModel(){

                        return "This is Android Mobile- " + model;

                }

}


package oopsconcept;

public class Blackberry extends Mobile{

        //Constructor to set properties/characteristics of object

        Blackberry(String man, String o,String m, int c){

                                        super(man, o, m, c);

                                }

        public String getModel(){

                return "This is Blackberry-"+ model;

        }

}
```

Polymorphism

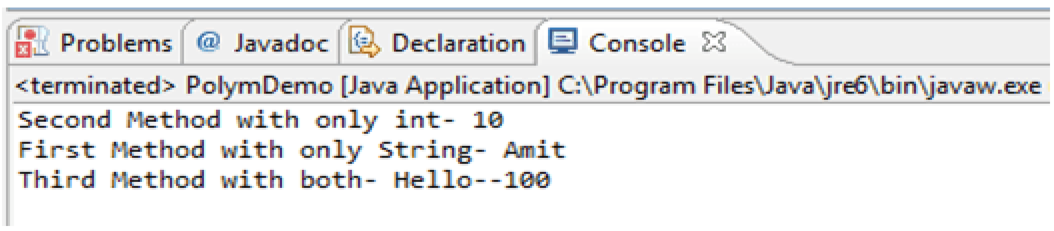
In Core, Java Polymorphism is one of easy concept to understand. Polymorphism definition is that Poly means many and morphos means forms. It describes the feature of languages that allows the same word or symbol to be interpreted correctly in different situations based on the context. There are two types of Polymorphism available in Java. For example, in English, the verb “run” means different things if you use it with “a footrace,” a “business,” or “a computer.” You understand the meaning of “run” based on the other words used with it. Object-oriented programs are written so that the methods having the same name works differently in different context. Java provides two ways to implement polymorphism.

Static Polymorphism (compile time polymorphism/ Method overloading):

The ability to execute different method implementations by altering the argument used with the method name is known as method overloading. In below program, we have three print methods each with different arguments. When you properly overload a method, you can call it providing different argument lists, and the appropriate version of the method executes.

```
package oopsconcept;

class Overloadsample {

        public void print(String s){

                System.out.println("First Method with only String- "+ s);

        }

        public void print (int i){

                System.out.println("Second Method with only int- "+ i);

        }

        public void print (String s, int i){

                System.out.println("Third Method with both- "+ s + "--" + i);

        }

}

public class PolymDemo {

        public static void main(String[] args) {

                Overloadsample obj = new Overloadsample();

                obj.print(10);

                obj.print("Amit");

                obj.print("Hello", 100);

        }

}
```

Output:



```
<terminated> PolymDemo [Java Application] C:\Program Files\Java\jre6\bin\javaw.exe
Second Method with only int- 10
First Method with only String- Amit
Third Method with both- Hello--100
```

Dynamic Polymorphism (run time polymorphism/ Method Overriding)

When you create a subclass by extending an existing class, the new subclass contains data and methods that were defined in the original superclass. In other words, any child class object has all the attributes of its parent. Sometimes, however, the superclass data fields and methods are not entirely appropriate for the subclass objects; in these cases, you want to override the parent class members. Let's take the example used in inheritance explanation.

```java
package oopsconcept;

public class OverridingDemo {

        public static void main(String[] args) {

                //Creating Object of SuperClass and calling getModel Method

                Mobile m = new Mobile("Nokia", "Win8", "Lumia",10000);

                System.out.println(m.getModel());

                //Creating Object of Sublcass and calling getModel Method

                Android a = new Android("Samsung", "Android", "Grand",30000);

                System.out.println(a.getModel());

                //Creating Object of Sublcass and calling getModel Method

                Blackberry b = new Blackberry("BlackB", "RIM", "Curve",20000);

                System.out.println(b.getModel());

        }

}
```

Abstraction

All programming languages provide abstractions. It can be argued that the complexity of the problems you're able to solve is directly related to the kind and quality of abstraction. An essential element of object-oriented programming is an abstraction. Humans manage complexity through abstraction. When you drive your car you do not have to be concerned with the exact internal working of your car(unless you are a mechanic). What you are concerned with is interacting with your car via its interfaces like steering wheel, brake pedal, accelerator pedal etc. Various manufacturers of car have different implementation of the car working but its basic interface has not changed (i.e. you still use the steering wheel, brake pedal, accelerator pedal etc to interact with your car). Hence the knowledge you have of your car is abstract.

A powerful way to manage abstraction is through the use of hierarchical classifications. This allows you to layer the semantics of complex systems, breaking them into more manageable pieces. From the outside, a car is a single object. Once inside, you see that the car consists of several subsystems: steering, brakes, sound system, seat belts, heating, cellular phone, and so on. In turn, each of these subsystems is made up of more specialized units. For instance, the sound system consists of a radio, a CD player, and/or a tape player. The point is that you manage the complexity of the car (or any other complex system)through the use of hierarchical abstractions.

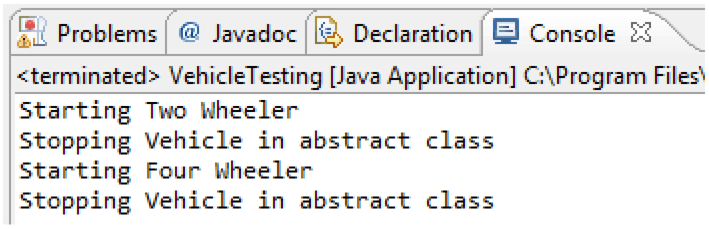
An abstract class is something which is incomplete and you can not create an instance of the abstract class. If you want to use it you need to make it complete or concrete by extending it. A class is called concrete if it does not contain any abstract method and implements all abstract method inherited from abstract class or interface it has implemented or extended. By the way, Java has a concept of abstract classes, abstract method but a variable can not be abstract in Java.

Let's take an example of Java Abstract Class called Vehicle. When I am creating a class called Vehicle, I know there should be methods like start() and Stop() but don't know start and stop mechanism of every vehicle since they could have different start and stop mechanism e.g some can be started by a kick or some can be by pressing buttons.

The advantage of Abstraction is if there is a new type of vehicle introduced we might just need to add one class which extends Vehicle Abstract class and implement specific methods. The interface of start and stop method would be same.

```
package oopsconcept;

public abstract class VehicleAbstract {

        public abstract void start();

        public void stop(){

                System.out.println("Stopping Vehicle in abstract class");

        }

}

class TwoWheeler extends VehicleAbstract{
        @Override
        public void start() {
                System.out.println("Starting Two Wheeler");
w       }
}
class FourWheeler extends VehicleAbstract{
        @Override
        public void start() {
                System.out.println("Starting Four Wheeler");
        }
}


package oopsconcept;
public class VehicleTesting {
        public static void main(String[] args) {
                VehicleAbstract my2Wheeler = new TwoWheeler();
                VehicleAbstract my4Wheeler = new FourWheeler();
                my2Wheeler.start();
                my2Wheeler.stop();
                my4Wheeler.start();
                my4Wheeler.stop();
        }
}
```

Output :



```
<terminated> VehicleTesting [Java Application] C:\Program Files\
Starting Two Wheeler
Stopping Vehicle in abstract class
Starting Four Wheeler
Stopping Vehicle in abstract class
```

Summary

- An object is an instance of a class.
- Encapsulation provides the security that keeps data and methods safe from inadvertent changes.
- Inheritance is a parent-child relationship of a class which is mainly used for code reusability.
- Polymorphism definition is that Poly means many and morphos means forms.

  li>Using abstraction one can simulate real world objects.
- Abstraction provides advantage of code reuse
- Abstraction enables program open for extension

Inheritance (IS-A) vs. Composition (HAS-A) Relationship

Description

One of the advantages of an Object-Oriented programming language is code reuse. There are two ways we can do code reuse either by the vimplementation of inheritance (IS-A relationship), or object composition (HAS-A relationship). Although the compiler and Java virtual machine (JVM) will do a lot of work for you when you use inheritance, you can also get at the functionality of inheritance when you use composition.

IS-A Relationship:

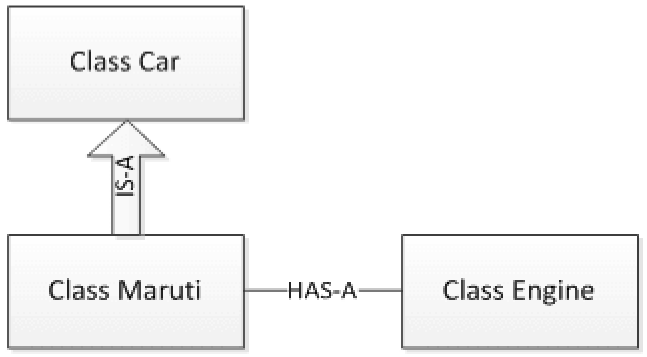
In object-oriented programming, the concept of IS-A is a totally based on Inheritance, which can be of two types Class Inheritance or Interface Inheritance. It is just like saying "A is a B type of thing". For example, Apple is a Fruit, Car is a Vehicle etc. Inheritance is uni-directional. For example, House is a Building. But Building is not a House.

It is a key point to note that you can easily identify the IS-A relationship. Wherever you see an extends keyword or implements keyword in a class declaration, then this class is said to have IS-A relationship.

HAS-A Relationship:

Composition(HAS-A) simply mean the use of instance variables that are references to other objects. For example Maruti has Engine, or House has Bathroom.

Let’s understand these concepts with an example of Car class.



```
package relationships;

class Car {

        // Methods implementation and class/Instance members

        private String color;

        private int maxSpeed;

        public void carInfo(){

                System.out.println("Car Color= "+color + " Max Speed= " + maxSpeed);

        }

        public void setColor(String color) {

                this.color = color;

        }

        public void setMaxSpeed(int maxSpeed) {

                this.maxSpeed = maxSpeed;

        }

}
```

As shown above, Car class has a couple of instance variable and few methods. Maruti is a specific type of Car which extends Car class means Maruti IS-A Car.

```
class Maruti extends Car{
        //Maruti extends Car and thus inherits all methods from Car (except final and static)

        //Maruti can also define all its specific functionality

        public void MarutiStartDemo(){

                Engine MarutiEngine = new Engine();

                MarutiEngine.start();

                }

        }
```

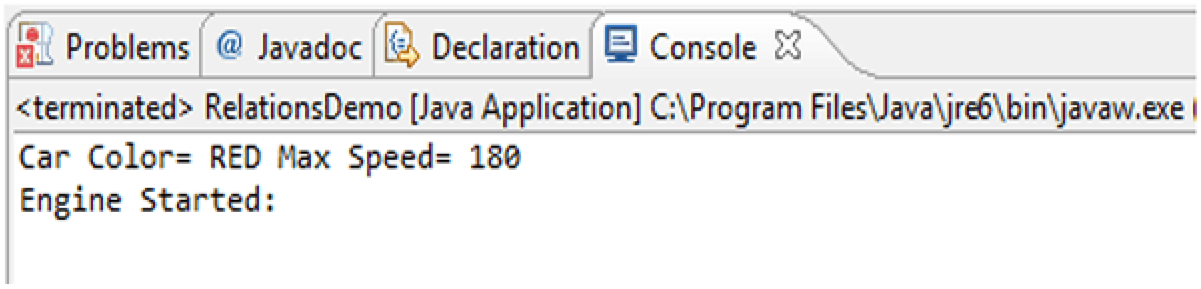
Maruti class uses Engine object’s start() method via composition. We can say that Maruti class HAS-A Engine.

```
package relationships;

public class Engine {

        public void start(){

                System.out.println("Engine Started:");

        }

        public void stop(){

                System.out.println("Engine Stopped:");

        }

}
```

RelationsDemo class is making object of Maruti class and initialized it. Though Maruti class does not have setColor(), setMaxSpeed() and carInfo() methods still we can use it due to IS-A relationship of Maruti class with Car class.

```
package relationships;

public class RelationsDemo {

        public static void main(String[] args) {

                Maruti myMaruti = new Maruti();

                myMaruti.setColor("RED");
                myMaruti.setMaxSpeed(180);

                myMaruti.carInfo();

                myMaruti.MarutiStartDemo();

        }

}
```

If we run RelationsDemo class we can see output like below.



```
<terminated> RelationsDemo [Java Application] C:\Program Files\Java\jre6\bin\javaw.exe
Car Color= RED Max Speed= 180
Engine Started:
```

Comparing Composition and Inheritance

- It is easier to change the class implementing composition than inheritance. The change of a superclass impacts the inheritance hierarchy to subclasses.
- You can't add to a subclass a method with the same signature but a different return type as a method inherited from a superclass. Composition, on the other hand, allows you to change the interface of a front-end class without affecting back-end classes.
- Composition is dynamic binding (run-time binding) while Inheritance is static binding (compile time binding)
- It is easier to add new subclasses (inheritance) than it is to add new front-end classes (composition) because inheritance comes with polymorphism. If you have a bit of code that relies only on a superclass interface, that code can work with a new subclass without change. This is not true of composition unless you use composition with interfaces. Used together, composition and interfaces make a very powerful design tool.
- With both composition and inheritance, changing the implementation (not the interface) of any class is easy. The ripple effect of implementation changes remains inside the same class.
    - Don't use inheritance just to get code reuse If all you really want is to reuse code and there is no is-a relationship in sight, use composition.
    - Don't use inheritance just to get at polymorphism If all you really want is a polymorphism, but there is no natural is-a relationship, use composition with interfaces.

Summary

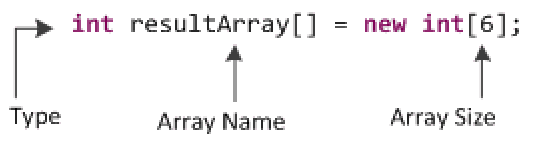
- IS-A relationship based on Inheritance, which can be of two types Class Inheritance or Interface Inheritance.
- Has-a relationship is composition relationship which is a productive way of code reuse.

Java Arrays

Description

An array is a group of similar typed variables that are referred to by a common name. Arrays of any type can be created and may have one or more dimensions. A specific element in an array is accessed by its index. The array is a simple type of data structure which can store primitive variable or objects. For example, imagine if you had to store the result of six subjects we can do it using an array. To create an array value in Java, you use the new keyword, just as you do to create an object.

Defining and constructing one dimensional array

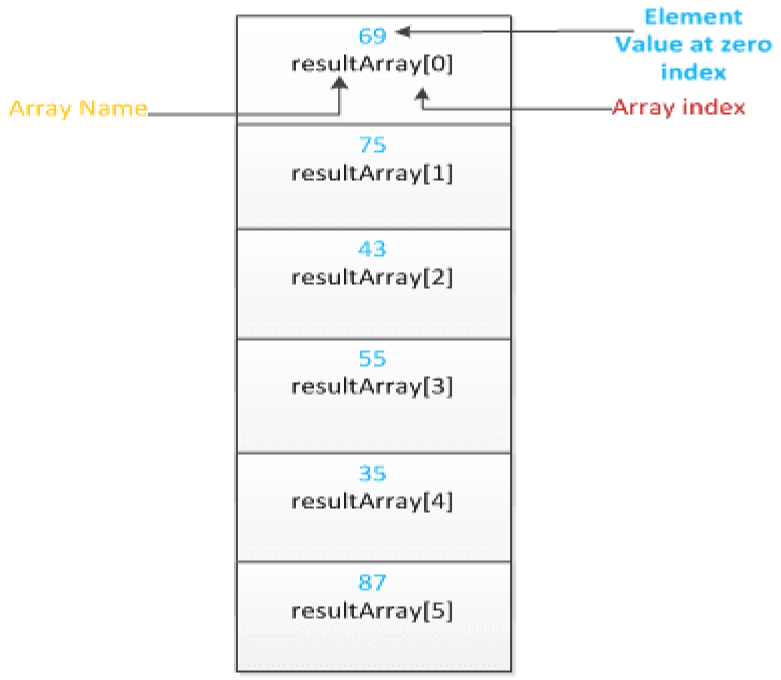

```
int resultArray[] = new int[6];
```

Here, type specifies the type of variables (int, boolean, char, float etc) being stored, size specifies the number of elements in the array, and arrayname is the variable name that is the reference to the array. Array size must be specified while creating an array. If you are creating a int[], for example, you must specify how many int values you want it to hold (in above statement resultArray[] is having size 6 int values). Once an array is created, it can never grow or shrink.

Initializing array: You can initialize specific element in the array by specifying its index within square brackets. All array indexes start at zero.

```
resultArray[0]=69;
```

This will initialize first element (index zero) of resultArray[] with integer value 69. Array elements can be initialized/accessed in any order. In memory, it will create a structure similar to below figure.



Array Literals

The null literal used to represent the absence of an object can also be used to represent the absence of an array. For example:

```
String [] name = null;
```

In addition to the null literal, Java also defines a special syntax that allows you to specify array values literally in your programs. This syntax can be used only when declaring a variable of array type. It combines the creation of the array object with the initialization of the array elements:

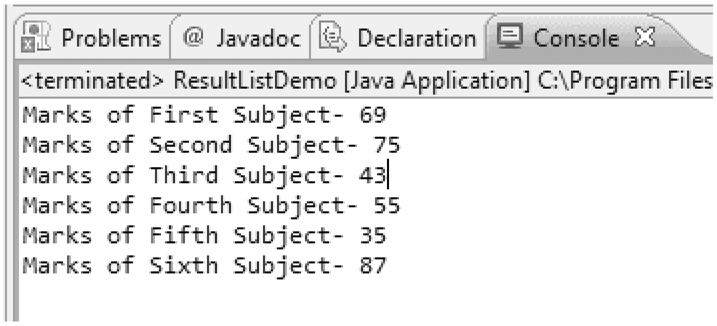
```
String[] daysOfWeek = {"Sunday", "Monday", "Tuesday",
"Wednesday", "Thursday", "Friday", "Saturday"};
```

This creates an array that contains the seven string element representing days of the week within the curly braces. Note that we don't use the new keyword or specify the type of the array in this array literal syntax. The type is implicit in the variable declaration of which the initializer is a part. Also, the array length is not specified explicitly with this syntax; it is determined implicitly by counting the number of elements listed between the curly braces.

Let's see sample java program to understand this concept better. This program will help to understand initializing and accessing specific array elements.

```
package arrayDemo;

import java.util.Arrays;

public class ResultListDemo {

        public static void main(String[] args) {
                //Array Declaration

                int resultArray[] = new int[6];

                //Array Initialization

                                resultArray[0]=69;

                                resultArray[1]=75;

                                resultArray[2]=43;

                                resultArray[3]=55;

                                resultArray[4]=35;

                                resultArray[5]=87;

                //Array elements access

                System.out.println("Marks of First Subject- "+ resultArray[0]);

                System.out.println("Marks of Second Subject- "+ resultArray[1]);

                System.out.println("Marks of Third Subject- "+ resultArray[2]);

                System.out.println("Marks of Fourth Subject- "+ resultArray[3]);

                System.out.println("Marks of Fifth Subject- "+ resultArray[4]);

                System.out.println("Marks of Sixth Subject- "+ resultArray[5]);

        }

}
```

Output:



```
<terminated> ResultListDemo [Java Application] C:\Program Files
Marks of First Subject- 69
Marks of Second Subject- 75
Marks of Third Subject- 43
Marks of Fourth Subject- 55
Marks of Fifth Subject- 35
Marks of Sixth Subject- 87
```

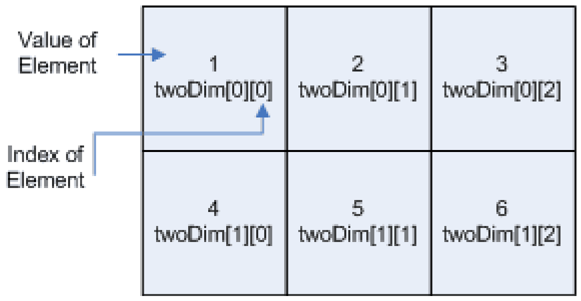
Alternative syntax for declaring, initializing of array in the same statement

```
int [] resultArray = {69,75,43,55,35,87};
```

Multidimensional Arrays

In Java, multidimensional arrays are actually arrays of arrays. These, as you might expect, look and act like regular multidimensional arrays. However, as you will see, there are a couple of subtle differences. To declare a multidimensional array variable, specify each additional index using another set of square brackets. For example, the following declares a two-dimensional array variable called twoDim. This will create a matrix of the size 2x3 in memory.

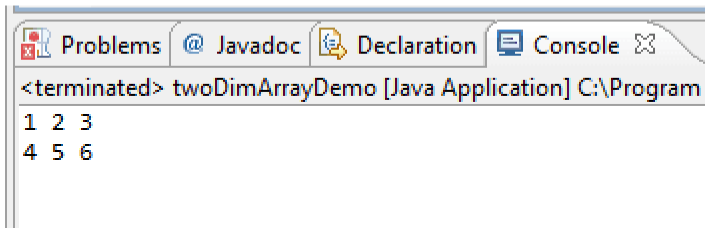
```
int twoDim[][] = new int[2][3];
```



Let’s have look at below program to understand 2-dimentional array

```
package arrayDemo;

public class twoDimArrayDemo {

        public static void main (String []args){

                int twoDim [][] = new int [2][3];

                twoDim[0][0]=1;

                twoDim[0][1]=2;

                twoDim[0][2]=3;

                twoDim[1][0]=4;

                twoDim[1][1]=5;

                twoDim[1][2]=6;

                System.out.println(twoDim[0][0] + " " + twoDim[0][1] + " " + twoDim[0][2]);

                System.out.println(twoDim[1][0] + " " + twoDim[1][1] + " " + twoDim[1][2]);

        }

}
```

Output:



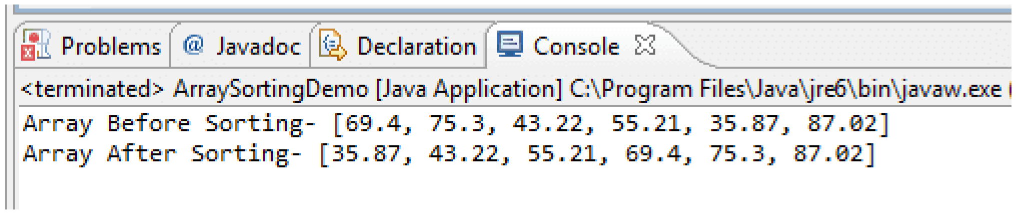
```
<terminated> twoDimArrayDemo [Java Application] C:\Program
1 2 3
4 5 6
```

Inbuilt Helper Class (java.util.Arrays) for Arrays Manipulation:

Java provides very important helper class (java.util.Arrays) for array manipulation. This class has many utility methods like array sorting, printing values of all array elements, searching of an element, copy one array into another array etc. Let’s see sample program to understand this class for better programming. In below program float array has been declared. We are printing the array elements before sorting and after sorting.

```
package arrayDemo;

import java.util.Arrays;

public class ArraySortingDemo {

        public static void main(String[] args) {

                //Declaring array of float elements

                float [] resultArray = {69.4f,75.3f,43.22f,55.21f,35.87f,87.02f};

                System.out.println("Array Before Sorting- " + Arrays.toString(resultArray));

                //below line will sort the array in ascending order

                Arrays.sort(resultArray);

                System.out.println("Array After Sorting- " + Arrays.toString(resultArray));

        }

}
```

Output:



```
<terminated> ArraySortingDemo [Java Application] C:\Program Files\Java\jre6\bin\javaw.exe
Array Before Sorting- [69.4, 75.3, 43.22, 55.21, 35.87, 87.02]
Array After Sorting- [35.87, 43.22, 55.21, 69.4, 75.3, 87.02]
```

Similar to “java.util.Arrays” System class also has a functionality of efficiently copying data from one array to another. Syntax as below,

```
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.

Important points:

- You will get “ArrayIndexOutOfBoundsException” if you try to access an array with an illegal index, that is with a negative number or with a number greater than or equal to its size.
- Arrays are widely used with loops (for loops, while loops). There will be more sample program of arrays with loops tutorial.

Summary

- An array is a group of similar typed variables that are referred to by a common name.
- The array can be declared using indexes or literals.
- The single dimensional array is widely used but we can have multi-dimensional.

Java Wrapper Classes

Description

Each of Java's eight primitive data types has a class dedicated to it. These are known as wrapper classes because they "wrap" the primitive data type into an object of that class. The wrapper classes are part of the java.lang package, which is imported by default into all Java programs.

The wrapper classes in java servers two primary purposes.

- To provide a mechanism to ‘wrap’ primitive values in an object so that primitives can do activities reserved for the objects like being added to ArrayList, Hashset, HashMap etc. collection.
- To provide an assortment of utility functions for primitives like converting primitive types to and from string objects, converting to various bases like binary, octal or hexadecimal, or comparing various objects.

The following two statements illustrate the difference between a primitive data type and an object of a wrapper class:

```
int   x = 25;
Integer   y = new Integer(33);
```

The first statement declares an int variable named x and initializes it with the value 25. The second statement instantiates an Integer object. The object is initialized with the value 33 and a reference to the object is assigned to the object variable y.

Below table lists wrapper classes in Java API with constructor details.

Primitive	**Wrapper Class**	**Constructor Argument**
boolean	Boolean	boolean or String
byte	Byte	byte or String
char	Character	char
int	Integer	int or String
float	Float	float, double or String
double	Double	double or String
long	Long	long or String
short	Short	short or String

Below is wrapper class hierarchy as per Java API

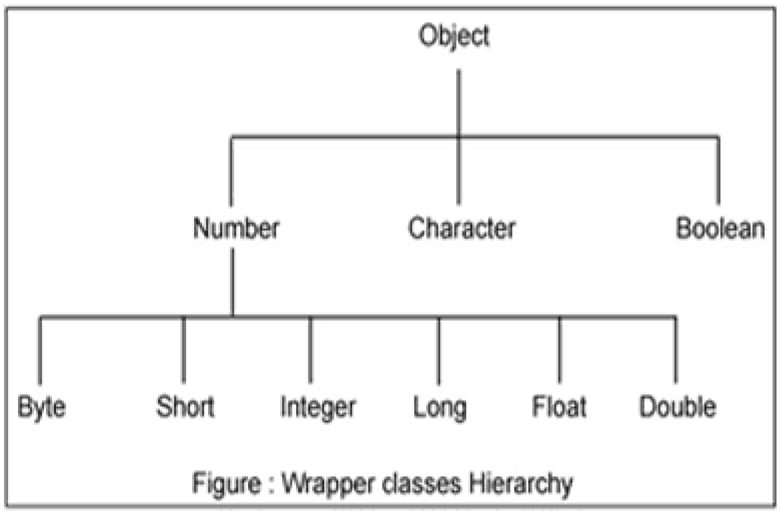


Figure : Wrapper classes Hierarchy

As explain in above table all wrapper classes (except Character) take String as argument constructor. Please note we might get NumberFormatException if we try to assign invalid argument in the constructor. For example to create Integer object we can have the following syntax.

```
Integer intObj = new Integer (25);
Integer intObj2 = new Integer ("25");
```

Here in we can provide any number as string argument but not the words etc. Below statement will throw run time exception (NumberFormatException)

```
Integer intObj3 = new Integer ("Two");
```

The following discussion focuses on the Integer wrapperclass, but applies in a general sense to all eight wrapper classes.

The most common methods of the Integer wrapper class are summarized in below table. Similar methods for the other wrapper classes are found in the Java API documentation.

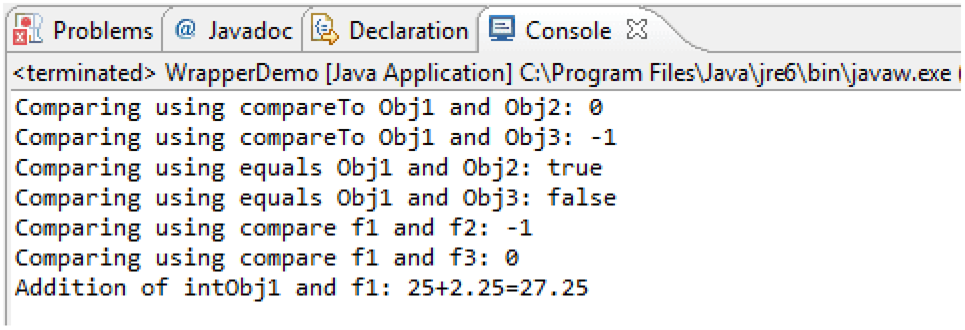
Method	**Purpose**
parseInt(s)	returns a signed decimal integer value equivalent to string s
toString(i)	returns a new String object representing the integer i
byteValue()	returns the value of this Integer as a byte
doubleValue()	returns the value of this Integer as a double
floatValue()	returns the value of this Integer as a float
intValue()	returns the value of this Integer as an int

shortValue()	returns the value of this Integer as a short
longValue()	returns the value of this Integer as a long
int compareTo(int i)	Compares the numerical value of the invoking object with that of i. Returns 0 if the values are equal. Returns a negative value if the invoking object has a lower value. Returns a positive value if the invoking object has a greater value.
static int compare(int num1, int num2)	Compares the values of num1 and num2. Returns 0 if the values are equal. Returns a negative value if num1 is less than num2. Returns a positive value if num1 is greater than num2.
boolean equals(Object intObj)	Returns true if the invoking Integer object is equivalent to intObj. Otherwise, it returns false.

Let's see java program which explains few wrapper classes methods.

```
package WrapperIntro;

public class WrapperDemo {

        public static void main (String args[]){

                Integer intObj1 = new Integer (25);

                Integer intObj2 = new Integer ("25");

                Integer intObj3= new Integer (35);

                //compareTo demo

                System.out.println("Comparing using compareTo Obj1 and Obj2: " +
intObj1.compareTo(intObj2));

                System.out.println("Comparing using compareTo Obj1 and Obj3: " +
intObj1.compareTo(intObj3));

                //Equals demo

                System.out.println("Comparing using equals Obj1 and Obj2: " +
intObj1.equals(intObj2));

                System.out.println("Comparing using equals Obj1 and Obj3: " +
intObj1.equals(intObj3));

                Float f1 = new Float("2.25f");

                Float f2 = new Float("20.43f");

                Float f3 = new Float(2.25f);

                System.out.println("Comparing using compare f1 and f2: " +Float.compare(f1,f2));

                System.out.println("Comparing using compare f1 and f3: " +Float.compare(f1,f3));

                //Addition of Integer with Float
                Float f = intObj1.floatValue() + f1;

                System.out.println("Addition of intObj1 and f1: "+ intObj1 +"+" +f1+"=" +f );

        }


}
```

Output:



Problems Javadoc Declaration Console

```
<terminated> WrapperDemo [Java Application] C:\Program Files\Java\jre6\bin\javaw.exe
Comparing using compareTo Obj1 and Obj2: 0
Comparing using compareTo Obj1 and Obj3: -1
Comparing using equals Obj1 and Obj2: true
Comparing using equals Obj1 and Obj3: false
Comparing using compare f1 and f2: -1
Comparing using compare f1 and f3: 0
Addition of intObj1 and f1: 25+2.25=27.25
```

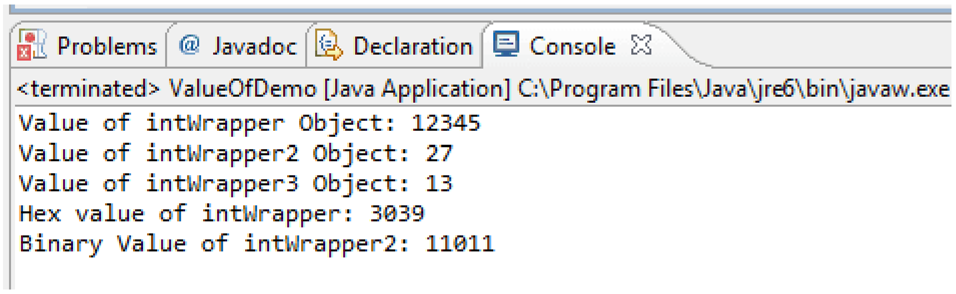
valueOf (), toHexString(), toOctalString() and toBinaryString() Methods:

This is another approach to creating wrapper objects. We can convert from binary or octal or hexadecimal before assigning a value to wrapper object using two argument constructor. Below program explains the method in details.

```
package WrapperIntro;

public class ValueOfDemo {

        public static void main(String[] args) {

                Integer intWrapper = Integer.valueOf("12345");

                //Converting from binary to decimal

                Integer intWrapper2 = Integer.valueOf("11011", 2);

                //Converting from hexadecimal to decimal

                Integer intWrapper3 = Integer.valueOf("D", 16);

                System.out.println("Value of intWrapper Object: "+ intWrapper);

                System.out.println("Value of intWrapper2 Object: "+ intWrapper2);

                System.out.println("Value of intWrapper3 Object: "+ intWrapper3);

                System.out.println("Hex value of intWrapper: " +
Integer.toHexString(intWrapper));
```

```
                System.out.println("Binary Value of intWrapper2: "+
Integer.toBinaryString(intWrapper2));

        }

}
```

Output:



```
<terminated> ValueOfDemo [Java Application] C:\Program Files\Java\jre6\bin\javaw.exe
Value of intWrapper Object: 12345
Value of intWrapper2 Object: 27
Value of intWrapper3 Object: 13
Hex value of intWrapper: 3039
Binary Value of intWrapper2: 11011
```

Summary

- Each of primitive data types has dedicated class in java library.
- Wrapper class provides many methods while using collections like sorting, searching etc.

Java Wrapper Classes

Description

Each of Java's eight primitive data types has a class dedicated to it. These are known as wrapper classes because they "wrap" the primitive data type into an object of that class. The wrapper classes are part of the java.lang package, which is imported by default into all Java programs.

The wrapper classes in java servers two primary purposes.

- To provide a mechanism to ‘wrap’ primitive values in an object so that primitives can do activities reserved for the objects like being added to ArrayList, Hashset, HashMap etc. collection.
- To provide an assortment of utility functions for primitives like converting primitive types to and from string objects, converting to various bases like binary, octal or hexadecimal, or comparing various objects.

The following two statements illustrate the difference between a primitive data type and an object of a wrapper class:

```
int   x = 25;
Integer   y = new Integer(33);
```

The first statement declares an int variable named x and initializes it with the value 25. The second statement instantiates an Integer object. The object is initialized with the value 33 and a reference to the object is assigned to the object variable y.

Below table lists wrapper classes in Java API with constructor details.

Primitive	**Wrapper Class**	**Constructor Argument**
boolean	Boolean	boolean or String
byte	Byte	byte or String
char	Character	char
int	Integer	int or String
float	Float	float, double or String
double	Double	double or String
long	Long	long or String
short	Short	short or String

Below is wrapper class hierarchy as per Java API

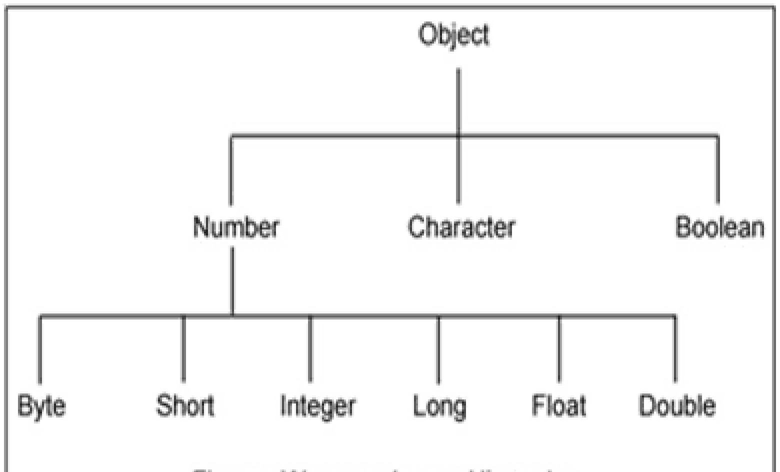


Figure : Wrapper classes Hierarchy

As explain in above table all wrapper classes (except Character) take String as argument constructor. Please note we might get NumberFormatException if we try to assign invalid argument in the constructor. For example to create Integer object we can have the following syntax.

```
Integer intObj = new Integer (25);
Integer intObj2 = new Integer ("25");
```

Here in we can provide any number as string argument but not the words etc. Below statement will throw run time exception (NumberFormatException)

```
Integer intObj3 = new Integer ("Two");
```

The following discussion focuses on the Integer wrapperclass, but applies in a general sense to all eight wrapper classes.

The most common methods of the Integer wrapper class are summarized in below table. Similar methods for the other wrapper classes are found in the Java API documentation.

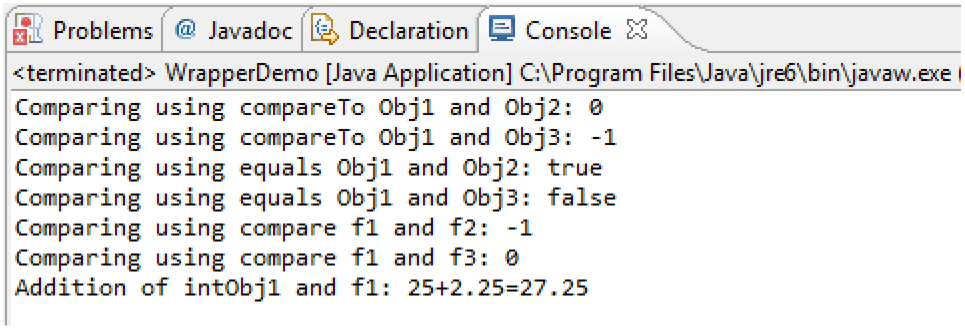
Method	**Purpose**
parseInt(s)	returns a signed decimal integer value equivalent to string s
toString(i)	returns a new String object representing the integer i
byteValue()	returns the value of this Integer as a byte
doubleValue()	returns the value of this Integer as a double
floatValue()	returns the value of this Integer as a float
intValue()	returns the value of this Integer as an int

shortValue()	returns the value of this Integer as a short
longValue()	returns the value of this Integer as a long
int compareTo(int i)	Compares the numerical value of the invoking object with that of i. Returns 0 if the values are equal. Returns a negative value if the invoking object has a lower value. Returns a positive value if the invoking object has a greater value.
static int compare(int num1, int num2)	Compares the values of num1 and num2. Returns 0 if the values are equal. Returns a negative value if num1 is less than num2. Returns a positive value if num1 is greater than num2.
boolean equals(Object intObj)	Returns true if the invoking Integer object is equivalent to intObj. Otherwise, it returns false.

Let’s see java program which explains few wrapper classes methods.

```
package WrapperIntro;

public class WrapperDemo {

        public static void main (String args[]){

                Integer intObj1 = new Integer (25);

                Integer intObj2 = new Integer ("25");

                Integer intObj3= new Integer (35);

                //compareTo demo

                System.out.println("Comparing using compareTo Obj1 and Obj2: " +
intObj1.compareTo(intObj2));

                System.out.println("Comparing using compareTo Obj1 and Obj3: " +
intObj1.compareTo(intObj3));

                //Equals demo

                System.out.println("Comparing using equals Obj1 and Obj2: " +
intObj1.equals(intObj2));

                System.out.println("Comparing using equals Obj1 and Obj3: " +
intObj1.equals(intObj3));

                Float f1 = new Float("2.25f");

                Float f2 = new Float("20.43f");

                Float f3 = new Float(2.25f);

                System.out.println("Comparing using compare f1 and f2: " +Float.compare(f1,f2));

                System.out.println("Comparing using compare f1 and f3: " +Float.compare(f1,f3));

                //Addition of Integer with Float
                Float f = intObj1.floatValue() + f1;

                System.out.println("Addition of intObj1 and f1: "+ intObj1 +"+" +f1+"=" +f );

        }


}
```

Output:



Problems | Javadoc | Declaration | Console

```
<terminated> WrapperDemo [Java Application] C:\Program Files\Java\jre6\bin\javaw.exe (
Comparing using compareTo Obj1 and Obj2: 0
Comparing using compareTo Obj1 and Obj3: -1
Comparing using equals Obj1 and Obj2: true
Comparing using equals Obj1 and Obj3: false
Comparing using compare f1 and f2: -1
Comparing using compare f1 and f3: 0
Addition of intObj1 and f1: 25+2.25=27.25
```

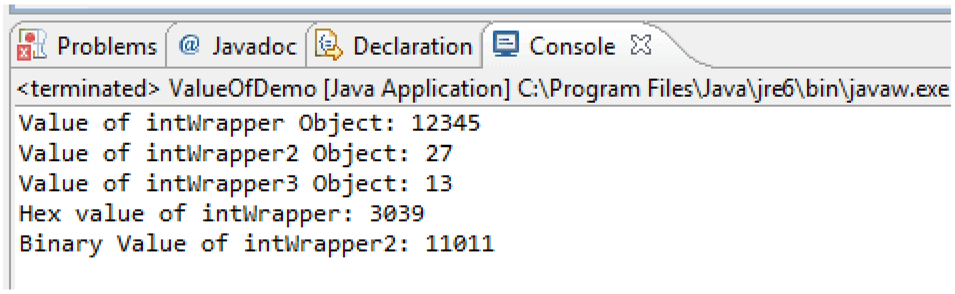
valueOf (), toHexString(), toOctalString() and toBinaryString() Methods:

This is another approach to creating wrapper objects. We can convert from binary or octal or hexadecimal before assigning a value to wrapper object using two argument constructor. Below program explains the method in details.

```
package WrapperIntro;

public class ValueOfDemo {

        public static void main(String[] args) {

                Integer intWrapper = Integer.valueOf("12345");

                //Converting from binary to decimal

                Integer intWrapper2 = Integer.valueOf("11011", 2);

                //Converting from hexadecimal to decimal

                Integer intWrapper3 = Integer.valueOf("D", 16);

                System.out.println("Value of intWrapper Object: "+ intWrapper);

                System.out.println("Value of intWrapper2 Object: "+ intWrapper2);

                System.out.println("Value of intWrapper3 Object: "+ intWrapper3);

                System.out.println("Hex value of intWrapper: " +
Integer.toHexString(intWrapper));
```

```
                System.out.println("Binary Value of intWrapper2: "+
Integer.toBinaryString(intWrapper2));

        }

}
```

Output:



```
<terminated> ValueOfDemo [Java Application] C:\Program Files\Java\jre6\bin\javaw.exe
Value of intWrapper Object: 12345
Value of intWrapper2 Object: 27
Value of intWrapper3 Object: 13
Hex value of intWrapper: 3039
Binary Value of intWrapper2: 11011
```

Summary

- Each of primitive data types has dedicated class in java library.
- Wrapper class provides many methods while using collections like sorting, searching etc.

Java Arithmetic Operators

Description

We can use arithmetic operators to perform calculations with values in programs. Arithmetic operators are used in mathematical expressions in the same way that they are used in algebra. A value used on either side of an operator is called an operand. For example, in below statement the expression 47 + 3, the numbers 47 and 3 are operands. The arithmetic operators are examples of binary operators because they require two operands. The operands of the arithmetic operators must be of a numeric type. You cannot use them on boolean types, but you can use them on char types, since the char type in Java is, essentially, a subset of int.

```
int a = 47+3;
```

Operator	Use	Description	Example
+	x + y	Adds x and y	float num = 23.4 + 1.6; // num=25
-	x - y	Subtracts y from x	long n = 12.456 – 2.456; //n=10
	-x	Arithmetically negates x	int i = 10; -i; // i = -10
*	x * y	Multiplies x by y	int m = 10*2; // m=20
/	x / y	Divides x by y	float div = 20/100 ; // div = 0.2
%	x % y	Computes the remainder of dividing x by y	int rm = 20/3; // rm = 2

In Java, you need to be aware of the type of the result of a binary (two-argument) arithmetic operator.

- If either operand is of type double, the other is converted to double.
- Otherwise, if either operand is of type float, the other is converted to float.
- Otherwise, if either operand is of type long, the other is converted to long.
- Otherwise, both operands are converted to type int.

For unary (single-argument) arithmetic operators:

- If the operand is of type byte, short, or char then the result is a value of type int.
- Otherwise, a unary numeric operand remains as is and is not converted.

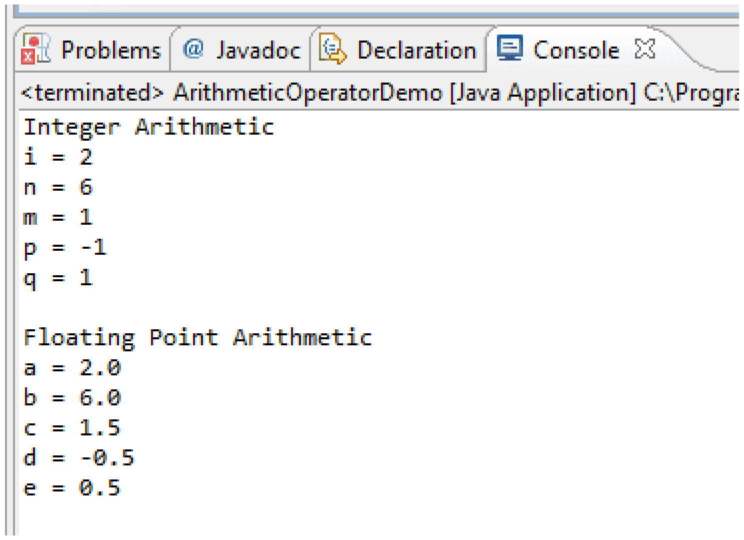
The basic arithmetic operations—addition, subtraction, multiplication, and division— all behave as you would expect for all numeric types. The minus operator also has a unary form that negates its single operand. Remember that when the division operator is applied to an integer type, there will be no fractional component attached to the result.

The following simple program demonstrates the arithmetic operators. It also illustrates the difference between floating-point division and integer division.

```
public class ArithmeticOperatorDemo {
        // Demonstrate the basic arithmetic operators.
```

```
               public static void main(String args[]) {
               // arithmetic using integers
               System.out.println("Integer Arithmetic");
               int i = 1 + 1;
               int n = i * 3;
               int m = n / 4;
               int p = m - i;
               int q = -p;
               System.out.println("i = " + i);
               System.out.println("n = " + n);
               System.out.println("m = " + m);
               System.out.println("p = " + p);
               System.out.println("q = " + q);
               // arithmetic using doubles
               System.out.println("\nFloating Point Arithmetic");
               double a = 1 + 1;
               double b = a * 3;
               double c = b / 4;
               double d = c - a;
               double e = -d;
               System.out.println("a = " + a);
               System.out.println("b = " + b);
               System.out.println("c = " + c);
               System.out.println("d = " + d);
               System.out.println("e = " + e);
       }
}
```

Output:



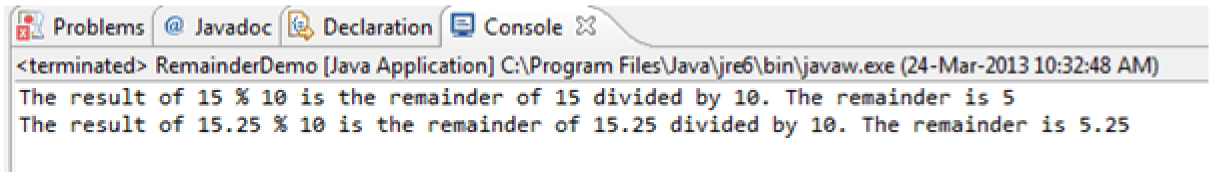
```
<terminated> ArithmeticOperatorDemo [Java Application] C:\Progra
Integer Arithmetic
i = 2
n = 6
m = 1
p = -1
q = 1

Floating Point Arithmetic
a = 2.0
b = 6.0
c = 1.5
d = -0.5
e = 0.5
```

The Modulus Operator

The modulus operator, %, returns the remainder of a division operation. It can be applied to floating-point types as well as integer types. The following example program demonstrates the %:

```
public class RemainderDemo {
               public static void main (String [] args) {
               int x = 15;
               int int_remainder = x % 10;
               System.out.println("The result of 15 % 10 is the "
                              + "remainder of 15 divided by 10. The remainder is " +
int_remainder);
               double d = 15.25;
               double double_remainder= d % 10;
               System.out.println("The result of 15.25 % 10 is the "
                              + "remainder of 15.25 divided by 10. The remainder is " +
double_remainder);
               }
}
```

Output:



```
<terminated> RemainderDemo [Java Application] C:\Program Files\Java\jre6\bin\javaw.exe (24-Mar-2013 10:32:48 AM)
The result of 15 % 10 is the remainder of 15 divided by 10. The remainder is 5
The result of 15.25 % 10 is the remainder of 15.25 divided by 10. The remainder is 5.25
```

Also, there are a couple of quirks to keep in mind regarding division by 0:

- A non-zero floating-point value divided by 0 results in a signed Infinity. 0.0/0.0 results isNaN.
- A non-zero integer value divided by integer 0 will result in ArithmeticException at runtime

Shortcut Arithmetic Operators (Increment and decrement operator)

The increment operator increases its operand by one. The decrement operator decreases its operand by one. For example, this statement:

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

Same way decrement operator

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

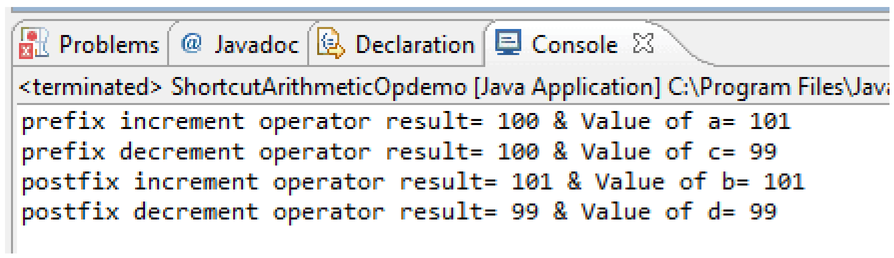
is equivalent to

```
x--;
```

These operators are unique in that they can appear both in postfix form, where they follow the operand as just shown, and prefix form, where they precede the operand. In the foregoing examples, there is no difference between the prefix and postfix forms. However, when the increment and/or decrement operators are part of a larger expression, then a subtle, yet powerful, difference between these two forms appears. In the prefix form, the operand is incremented or decremented before the value is obtained for use in the expression. In postfix form, the previous value is obtained for use in the expression, and then the operand is modified.Let’s understand this concept with help of example below,

```
public class ShortcutArithmeticOpdemo {
      public static void main(String[] args) {
            int a,b,c,d;
            a=b=c=d=100;
            int p = a++;
            int r = c--;
            int q = ++b;
            int s = --d;
            System.out.println("prefix increment operator result= "+ p + "
& Value of a= "+ a);
            System.out.println("prefix decrement operator result= "+ r + "
& Value of c= "+c);
            System.out.println("postfix increment operator result= "+ q +
" & Value of b= "+ b);
            System.out.println("postfix decrement operator result= "+ s +
" & Value of d= "+d);
      }
}
```

Output:



```
<terminated> ShortcutArithmeticOpdemo [Java Application] C:\Program Files\Jav
prefix increment operator result= 100 & Value of a= 101
prefix decrement operator result= 100 & Value of c= 99
postfix increment operator result= 101 & Value of b= 101
postfix decrement operator result= 99 & Value of d= 99
```

Summary

- Arithmetic operators are used in mathematical expressions.
- Arithmetic operators are +(addition) , -(subtraction), * (multiplication), / (division) and % (reminder).
- Java provides built-in short-circuit addition and subtraction operators.

Java Conditional or Relational Operators

Description

If you need to change the execution of the program based on a certain condition you can use "if" statements. The relational operators determine the relationship that one operand has to the other. Specifically, they determine equality condition. Java provides six relational operators, which are listed in below table.

Operator	Description	Example (a=10, b=15)	Result

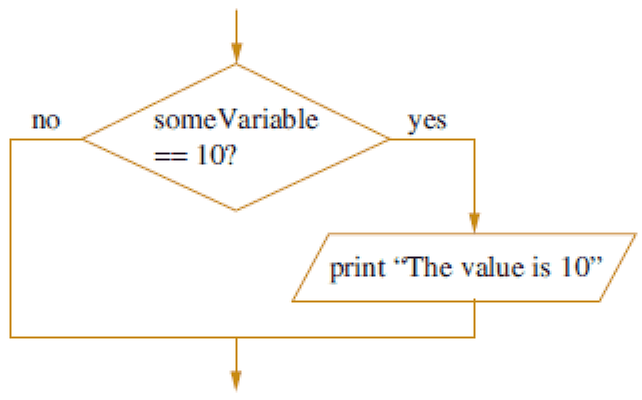
==	Equality operator	a==b	false
!=	Not Equal to operator	a!=b	true
>	Greater than	a>b	false
<	Less than	a<b	true
>=	Greater than or equal to	a>=b	false
<=	Less than or equal to	a<=b	true

The outcome of these operations is a boolean value. The relational operators are most frequently used in the expressions that control the if statement and the various loop statements. Any type in Java, including integers, floating-point numbers, characters, and booleans can be compared using the equality test, ==, and the inequality test, !=. Notice that in Java equality is denoted with two equal signs (“==”), not one (“=”).

In Java, the simplest statement you can use to make a decision is the if statement. Its simplest form is shown here:

```
if(condition) statement;
   or
if (condition) statement1;
else statement2;
```

Here, the condition is a Boolean expression. If the condition is true, then the statement is executed. If the condition is false, then the statement is bypassed. For example, suppose you have declared an integer variable named someVariable, and you want to print a message when the value of someVariable is 10. Flow chart and java code of the operation looks like below,



```
if (someVariable ==10){

System.out.println("The Value is 10");

}
```

We can have different flavors of if statements.

Nested if blocks:

A nested if is an if statement that is the target of another if or else. In other terms, we can consider one or multiple if statement within one if block to check various condition. For example, we have two variables and want to check particular condition for both we can use nested if blocks.

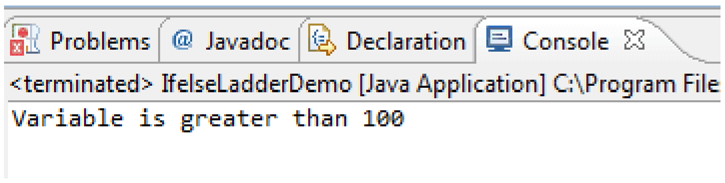
```
public class NestedIfDemo {

        public static void main(String[] args) {

                int a =10;

                int b =5;

                if(a==10){

                        if(b==5){

                                System.out.println("Inside Nested Loop");

                        }

                }

        }

}
```

if –else if ladder:

We might get a situation where we need to check value multiple times to find exact matching condition. Below program explains the same thing. Let's see we have a requirement to check if the variable value is less than 100, equal to 100 or more than 100. Below code explains the same logic using if-else-if ladder.

```
public class IfelseLadderDemo {

        public static void main(String[] args) {

                int a =120;

                if(a< 100){

                        System.out.println("Variable is less than 100");

                }

                else if(a==100)

                {

                        System.out.println("Variable is equal to 100");

                }

                else if (a>100)

                {

                        System.out.println("Variable is greater than 100");

                }

        }

}
```

Output:

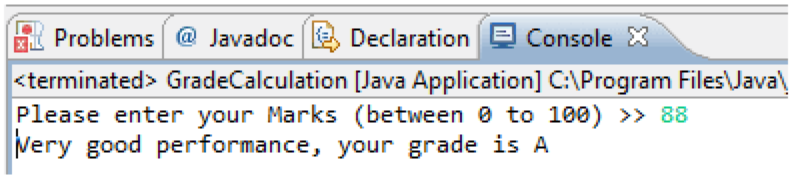


```
<terminated> IfelseLadderDemo [Java Application] C:\Program File
Variable is greater than 100
```

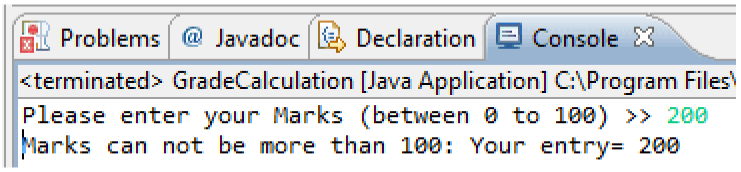
Let's take an example to understand above logical operators and conditional statements. We have a requirement to print result in a grade based on marks entered by the user. We are taking input from an user at runtime and evaluate grades. This program also validates input and prints appropriate message if the input is a negative or alphabetic entry.

```
import java.util.Scanner;

public class GradeCalculation {

        public static void main(String[] args) {

                int marks=0      ;

                try{

                        //Scanner class is wrapper class of System.in object

                Scanner inputDevice = new Scanner(System.in);

                System.out.print("Please enter your Marks (between 0 to 100) >> ");

                marks = inputDevice.nextInt();

                //Checking input validity and grade based on input value

                if(marks< 0){

                        System.out.println("Marks can not be negative: Your entry= "+ marks );

                }else if(marks==0){

                        System.out.println("You got Zero Marks: Go to ZOO");

                }else if (marks>100){

                        System.out.println("Marks can not be more than 100: Your entry= "+ marks
);

                }else if (marks>0 & marks< 35){

                        System.out.println("You need to work hard: You failed this time with
marks ="+ marks);

                }else if (marks>=35 & marks < 50){

                        System.out.println("Your score is not bad, but you need improvement, you
got C Grade");

                }else if (marks>=50 & marks < 60){

                        System.out.println("You can improve performance, you got C+ grade");

                }else if (marks>=60 & marks < 70){

                        System.out.println("Good performance with B grade");

                }else if (marks>=70 & marks < 80){

                        System.out.println("You can get better grade with little more efforts,
your grade is B+");

                }else if (marks>=80 & marks < 90){

                        System.out.println("Very good performance, your grade is A ");
                }else if (marks>=90){
                        System.out.println("One of the best performance, Your grade is A+");
                }
                }catch (Exception e){
                        //This is to handle non-numeric values
                        System.out.println("Invalid entry for marks:" );
                }
        }
}
```

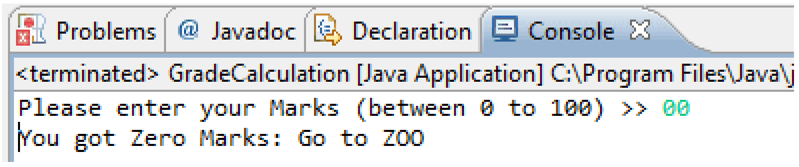
Outputs based on users input :



```
<terminated> GradeCalculation [Java Application] C:\Program Files\Java\
Please enter your Marks (between 0 to 100) >> 88
Very good performance, your grade is A
```



```
<terminated> GradeCalculation [Java Application] C:\Program Files\
Please enter your Marks (between 0 to 100) >> 200
Marks can not be more than 100: Your entry= 200
```



```
<terminated> GradeCalculation [Java Application] C:\Program Files\Java\j
Please enter your Marks (between 0 to 100) >> 00
You got Zero Marks: Go to ZOO
```

Short-Circuit Logical Operators

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

- && short-circuit AND
- || short-circuit OR

They are used to link little boolean expressions together to form bigger boolean expressions. The && and || operators evaluate only boolean values. For an AND (&&) expression to be true, both operands must be true. For example, The below statement evaluates to true because of both operand one (2 < 3) and operand two (3 < 4) evaluate to true.

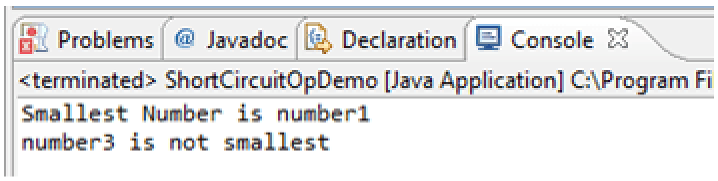
```
if ((2 < 3) && (3 < 4)) { }
```

The short-circuit feature of the && operator is so named because it doesn't waste its time on pointless evaluations. A short-circuit && evaluates the left side of the operation first (operand one), and if it resolves to false, the && operator doesn't bother looking at the right side of the expression (operand two) since the && operator already knows that the complete expression can't possibly be true.

The || operator is similar to the && operator, except that it evaluates to true if EITHER of the operands is true. If the first operand in an OR operation is true, the result will be true, so the short-circuit || doesn't waste time looking at the right side of the equation. If the first operand is false, however, the short-circuit || has to evaluate the second operand to see if the result of the OR operation will be true or false.

```
public class ShortCircuitOpDemo {

        public static void main(String[] args) {

                float number1 = 120.345f;

                float number2 = 345.21f;

                float number3 = 234.21f;

                if(number1 < number2 && number1< number3){

                        System.out.println("Smallest Number is number1");

                }

                if(number3 >number2 || number3 > number1){

                        System.out.println("number3 is not smallest");

                }

        }

}
```

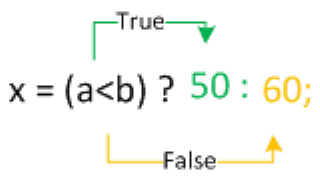
Output:



```
<terminated> ShortCircuitOpDemo [Java Application] C:\Program Fi
Smallest Number is number1
number3 is not smallest
```

Ternary Operator (or ? Operator or Conditional Operator)

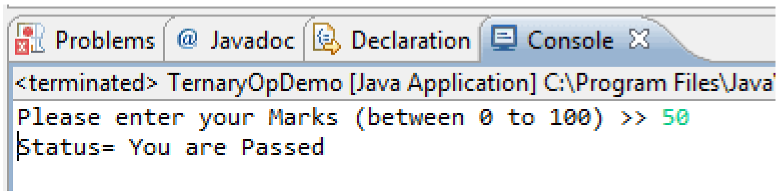
The conditional operator is a ternary operator (it has three operands) and is used to evaluate boolean expressions, much like an if statement except instead of executing a block of code if the test is true, a conditional operator will assign a value to a variable. A conditional operator starts with a boolean operation, followed by two possible values for the variable to the left of the assignment (=) operator. The first value (the one to the left of the colon) is assigned if the conditional (boolean) test is true, and the second value is assigned if the conditional test is false. In below example, if variable a is less than b then tje variable x value would be 50 else x =60.



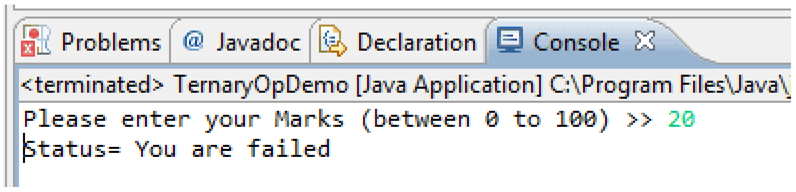
below example, we are deciding the status based on user input if pass or failed.

```
import java.util.Scanner;

public class TernaryOpDemo {

        public static void main(String[] args) {

                //Checking if marks greater than 35 or not

                String status;

                int marks;

                Scanner inputDevice = new Scanner(System.in);

                System.out.print("Please enter your Marks (between 0 to 100) >> ");

                marks = inputDevice.nextInt();

                status= marks>=35 ?"You are Passed":"You are failed";

                System.out.println("Status= " + status);

        }

}
```

Outputs based on users input :



```
<terminated> TernaryOpDemo [Java Application] C:\Program Files\Java
Please enter your Marks (between 0 to 100) >> 50
Status= You are Passed
```



```
<terminated> TernaryOpDemo [Java Application] C:\Program Files\Java\
Please enter your Marks (between 0 to 100) >> 20
Status= You are failed
```

Summary

- Java provides six conditional operators == (equality), > (greater than), < (less than), >=(greater or equal), <= (less or equal), != (not equal)
- The relational operators are most frequently used to control the flow of program.
- Short-Circuit logical operators are && and ||
- The ternary operator is one which is similar to if else block but which is used to assign value based on condition.

Java Switch Statement

Description

A programming language uses control statements to cause the flow of execution to advance and branch based on changes to the state of a program. Java supports two flow control statements: if and switch. These statements allow you to control the flow of your program’s execution based upon conditions known only during run time. We have discussed if statement in logical operator tutorial.

Switch Statement

The switch statement is Java’s multiway branch statement. It provides an easy way to dispatch execution to different parts of your code based on the value of an expression. Here is the general form of a switch statement:

```
switch (expression) {
```

```
case value1:
// statement sequence
break;
case value2:
// statement sequence
break;
….
case valueN:
// statement sequence
break;
default:
// default statement sequence
}
```

The expression must be of type byte, short, int, or char; each of the values specified in the case statements must be of a type compatible with the expression. An enumeration value can also be used to control a switch statement. From Java 7 onward String is also allowed as case expression. Each case value must be a unique literal (that is, it must be a constant, not a variable). Duplicate case values are not allowed.

How switch statement works

The value of the expression is compared with each of the literal values in the case statements. If a match is found, the code sequence following that case statement is executed. If none of the constants matches the value of the expression, then the default statement is executed. However, the default statement is optional. If no case matches and no default is present, then no further action is taken.

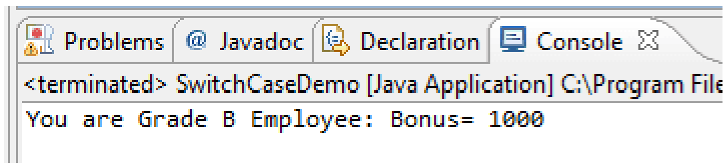
The break statement is used inside the switch to terminate a statement sequence. When a break statement is encountered, execution branches to the first line of code that follows the entire switch statement.

Let's understand the concept with the program. Below program print bonus amount based on the grade of employee. An employee can be of grade A, B, C or default (anything other than A, B & C).

```
public class SwitchCaseDemo {

        public static void main(String[] args) {

                char Grade ='B';

                switch (Grade){

                case 'A':

                        System.out.println("You are Grade A Employee: Bonus= "+ 2000);

                        break;
                case 'B':

                        System.out.println("You are Grade B Employee: Bonus= "+ 1000);

                        break;

                case 'C':

                        System.out.println("You are Grade C Employee: Bonus= "+ 500);

                        break;

                default:

                        System.out.println("You are Default Employee: Bonus= "+ 100);

                        break;

                }

        }

}
```

Copy

Output:



```
<terminated> SwitchCaseDemo [Java Application] C:\Program File
You are Grade B Employee: Bonus= 1000
```

Nested switch Statements

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

Important Points related to Switch-Case statements:

- The switch can only check for equality. This means that the other relational operators such as greater than are rendered unusable in a case.
- Case constants are evaluated from the top down, and the first case constant that matches the switch's expression is the execution entry point. If no break statement used, all the case after entry point will be executed.
- No two case constants in the same switch can have identical values. Of course, a switch statement and an enclosing outer switch can have case constants in common.

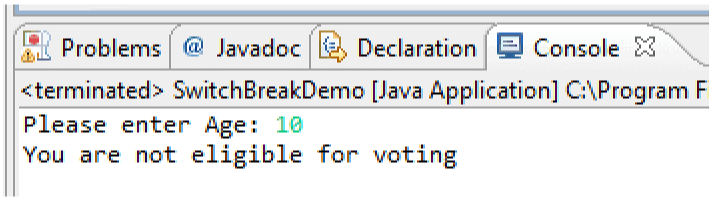
- The default case can be located at the end, middle, or top. Generally, default appears at end of all cases.

Break Statements

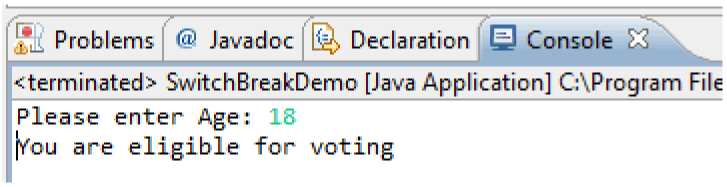
The break statement included with each case section determines when to stop executing statements in response to a matching case. Without a break statement in a case section, after a match is made, the statements for that match and all the statements further down the switch are executed until a break or the end of the switch is found. In some situations, this might be exactly what you want to do. Otherwise, you should include break statements to ensure that only the right code is executed. It is sometimes desirable to have multiple cases without break statements between them. For example here program prints same value until some condition reaches.

```java
import java.util.Scanner;

public class SwitchBreakDemo {

        public static void main(String[] args) {

                int age;

                Scanner inputDevice = new Scanner(System.in);

                System.out.print("Please enter Age: ");

                age = inputDevice.nextInt();

                switch (age){

                case 10:

                case 15:

                case 17:

                        System.out.println("You are not eligible for voting");

                        break;

                case 18:

                        System.out.println("You are eligible for voting");

                }

        }

}
```

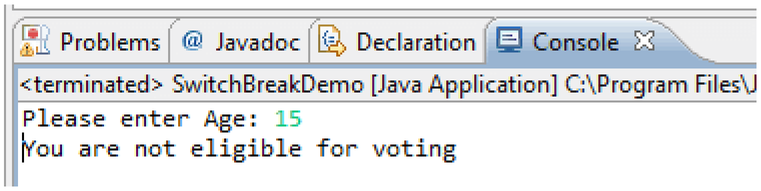
Output:



```
<terminated> SwitchBreakDemo [Java Application] C:\Program F
Please enter Age: 10
You are not eligible for voting
```



```
<terminated> SwitchBreakDemo [Java Application] C:\Program File
Please enter Age: 18
You are eligible for voting
```



```
<terminated> SwitchBreakDemo [Java Application] C:\Program Files\J
Please enter Age: 15
You are not eligible for voting
```

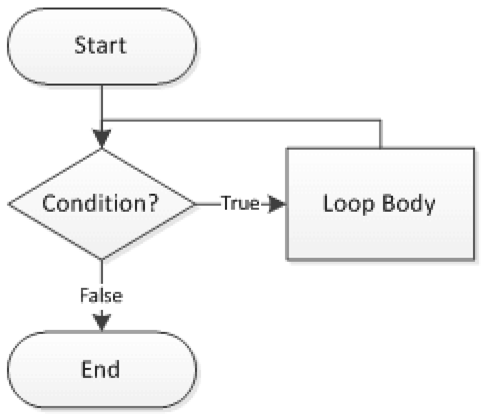
Summary

- The switch statement is Java’s multi-way branch statement.
- The switch can only check for equality. This means that the other relational operators such as greater than are rendered unusable in a case.
- The break statement is used to stop current iteration of loop or end Switch-case block.

While Loops in Java

Description

In computer programming, a loop is a sequence of instruction s that is continually repeated until a certain condition is reached. Imagine you have to write a program which performs a repetitive task such as printing 1 to 100. Writing 100 print statements would not be wise thing to do. Loops are specifically designed to perform repetitive tasks with one set of code. Loops save a lot of time. A loop is a structure that allows repeated execution of a block of statements. Within a looping structure, a Boolean expression is evaluated. If it is true, a block of statements called the loop body executes and the Boolean expression is evaluated again. As long as the expression is true, the statements in the loop body continue to execute. When the Boolean evaluation is false, the loop ends.



In Java there are three types of loops:

- A while loop, in which the loop-controlling Boolean expression is the first statement in the loop
- A for loop, which is usually used as a concise format in which to execute loops
- A do...while loop, in which the loop-controlling Boolean expression is the last statement in the loop

While Loops

The while loop is good for scenarios where you don't know how many times a block or statement should repeat, but you want to continue looping as long as some condition is true. A while statement looks like below. In Java, a while loop consists of the keyword while followed by a Boolean expression within parentheses, followed by the body of the loop, which can be a single statement or a block of statements surrounded by curly braces.

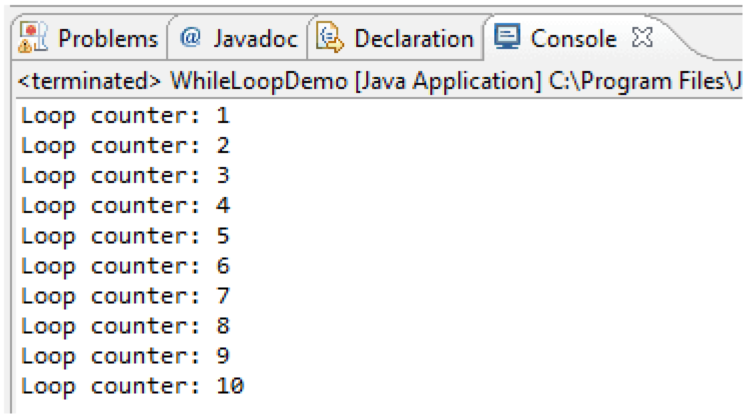
```
while (expression) {// do stuff}
```

You can use a while loop when you need to perform a task a predetermined number of times. A loop that executes a specific number of times is a definite loop or a counted loop. To write a definite loop, you initialize a loop control variable, a variable whose value determines whether loop execution continues. While the Boolean value that results from comparing the loop control variable and another value is true, the body of the while loop continues to execute. In the body of the loop, you must include a statement that alters the loop control variable.

```
public class WhileLoopDemo {

        public static void main(String[] args) {

                int var = 1;

                int limit=11;

                while (var < limit)

                {

                System.out.println("Loop counter: " + var);
```

```
                                        var++;

                        }

                }

}
```

Output:



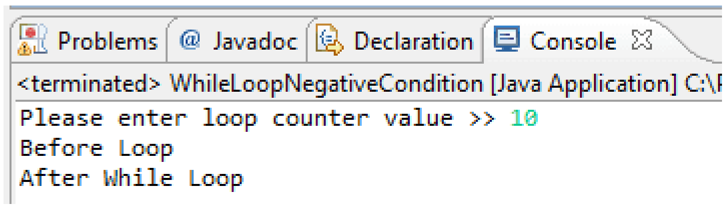
```
<terminated> WhileLoopDemo [Java Application] C:\Program Files\J
Loop counter: 1
Loop counter: 2
Loop counter: 3
Loop counter: 4
Loop counter: 5
Loop counter: 6
Loop counter: 7
Loop counter: 8
Loop counter: 9
Loop counter: 10
```

The key point to remember about a while loop is that it might not ever run. If the test expression is false the first time the while expression is checked, the loop body will be skipped and the program will begin executing at the first statement after the while loop.Let's take an example to understand this. In below program user enters the value of loop counter variable, If counter variable is less than 5, then the loop will execute and increment the counter variable by one till counter value is equal to 5. If counter variable is more than or equal to 5, the loop will not execute.

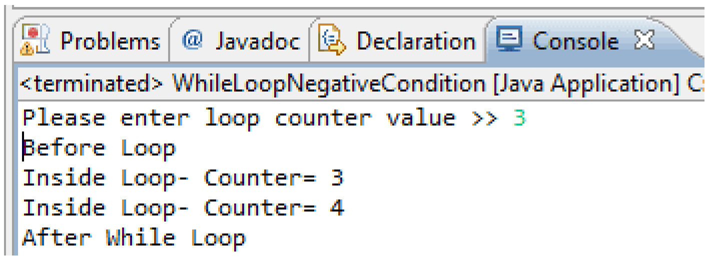
```
import java.util.Scanner;

public class WhileLoopNegativeCondition {

        public static void main(String[] args)

        {

        int counter;

        Scanner inputDevice = new Scanner(System.in);

        System.out.print("Please enter loop counter value >> ");

        counter = inputDevice.nextInt();

        System.out.println("Before Loop");

        while (counter < 5)

        {
```

```
        System.out.println("Inside Loop- Counter= "+ counter);

        counter++;

        }

        System.out.println("After While Loop");

    }

}
```

Output:



```
<terminated> WhileLoopNegativeCondition [Java Application] C:\
Please enter loop counter value >> 10
Before Loop
After While Loop
```



```
<terminated> WhileLoopNegativeCondition [Java Application] C
Please enter loop counter value >> 3
Before Loop
Inside Loop- Counter= 3
Inside Loop- Counter= 4
After While Loop
```

Important Points while using loops

- Loop condition/expression can be true always, which makes our loop infinite. This is bad programming practice as it might result in memory exception. Below statement is valid but not good to have in our program.

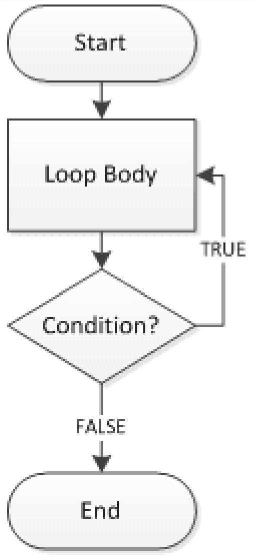
```
while (true) {// Some stuff
while (2<4)// Some stuff
```

- It is very common to alter the value of a loop control variable by adding 1 to it or incrementing the variable. However, not all loops are controlled by adding 1 sometimes we can control by subtracting 1 from a loop control variable or decrementing it.

do … while Loops

The do loop is similar to the while loop, except that the expression is not evaluated until after the do loop's code is executed. Therefore the code in a do loop is guaranteed to execute at least once. The following shows a do loop syntax:

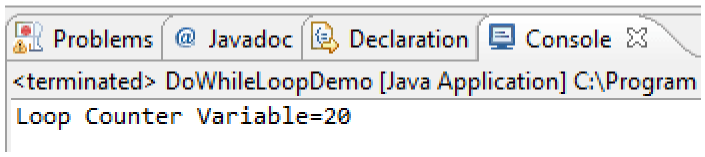
```
do {//Loop Body} while(Condition);
```



If so, you want to write a loop that checks at the “bottom” of the loop after the first iteration. The do...while loop checks the value of the loop control variable at the bottom of the loop after one repetition has occurred. Below sample code explain do… while loop.

```
public class DoWhileLoopDemo {

public static void main(String[] args) {

        int i =10;

        do{

        i=i+10;

        System.out.println("Loop Counter Variable=" +i);

        }

        while (i< 10);

        }

}
```

Output:



```
<terminated> DoWhileLoopDemo [Java Application] C:\Program
Loop Counter Variable=20
```

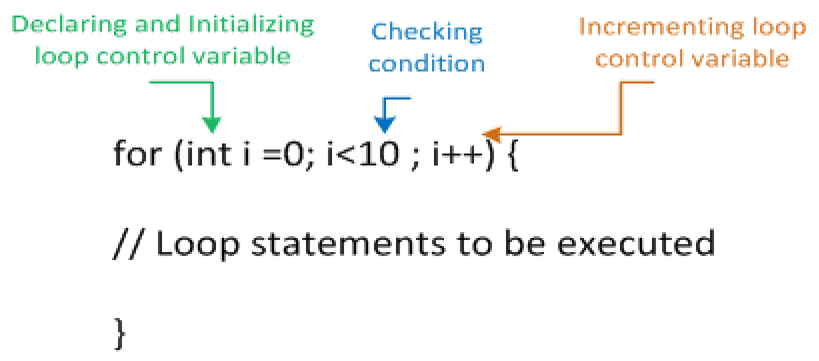
Summary

- The while loop is used for scenarios where you don't know how many times a block of code should be executed.
- The do loop is similar to the while loop, except that the expression is not evaluated until after the do loop's code is executed.

Java For Loop

Description

A for loop is a special loop that is used when a definite number of loop iterations is required. Although a while loop can also be used to meet this requirement, the for loop provides you with a shorthand notation for this type of loop. When you use a for loop, you can indicate the starting value for the loop control variable, the test condition that controls loop entry, and the expression that alters the loop control variable—all in one convenient place. Below is the syntax of a conventional for loop.


Declaring and Initializing loop control variable
Checking condition
Incrementing loop control variable

```
for (int i =0; i<10 ; i++) {

// Loop statements to be executed

}
```

For loop will begin with the keyword for followed by a set of parentheses. Within the parentheses are three sections separated by exactly two semicolons. The three sections are usually used for the following:

- Initializing the loop control variable

- Testing the loop control variable
- Updating the loop control variable

Within the parentheses of the for statement shown in below program, the first section prior to the first semicolon declares a variable named count and initializes it to 1. The program executes this statement once, no matter how many times the body of the for loop executes. After initialization, program control passes to the middle, or test section, of the for statement. If the Boolean expression found there evaluates to true, the body of the for loop is entered. In the program, the counter is set to 1, so when counter < 11 is tested, it evaluates to true. The loop body prints the counter value. If you want multiple statements to execute within the loop, they have to be blocked within a pair of curly braces. After the loop body executes, the final one-third of the for loop executes, and the counter is increased to 2. Following the third section in the for statement, program control returns to the second section, where theh counter is compared to 11 a second time. Because the counter is still less than 11, the body executes: counter (now 2) prints, and then the third altering portion of the for loop executes again. The variable counter increases to 3 and the for loop continues.

Eventually, when the counter is not less than 11 (after 1 through 10 have printed), the for loop ends and the program continues with any statements that follow the for loop.

```
for (int counter =1; counter<11 ; counter++)
{System.out.println(counter);}
```

For loop is very useful in java programming and widely used in java programs. Let's see few more samples for loop declaration.

- Initialization of more than one variable by placing commas between the separate statements, as in the following:

```
for(g = 0, h = 1; g < 6; ++g)
```

- Checking of more than one condition using AND or OR operators, as in the following:

```
  for(g = 0; g < 3 && h > 1; ++g, h--)
```

- Decrementing loop control variable and checking of some other condition, as in the following:

```
 for(g = 5; g >= 1; --g)
```

- Altering more than one value, as in the following:

```
 for(g = 0; g < 10; ++g, ++h, sum += g)
```

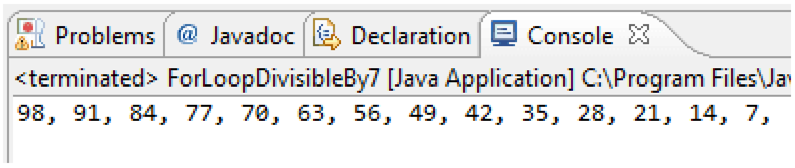
- You can leave one or more portions of a for loop empty, although the two semicolons are still required as placeholders. For example, if x has been initialized in a previous program statement, you might write the following:

```
for(; x < 10; ++x)
```

Let’s see below example which prints all the values divisible by 7 in the range of 1 to 100 in reverse order.

```
public class ForLoopDivisibleBy7 {

         public static void main(String[] args) {

                 for(int i=100; i>0; --i)

                 {

         if(i%7 == 0)

         {

     System.out.print(i);

     System.out.print(", ");

    }

   }

  }

 }
```

Output:



```
<terminated> ForLoopDivisibleBy7 [Java Application] C:\Program Files\Ja
98, 91, 84, 77, 70, 63, 56, 49, 42, 35, 28, 21, 14, 7,
```

Enhanced for loop

The enhanced for loop allows you to cycle through an array/ Collection without specifying the starting and ending points for the loop control variable. The general form of the enhanced for loop version is shown here:

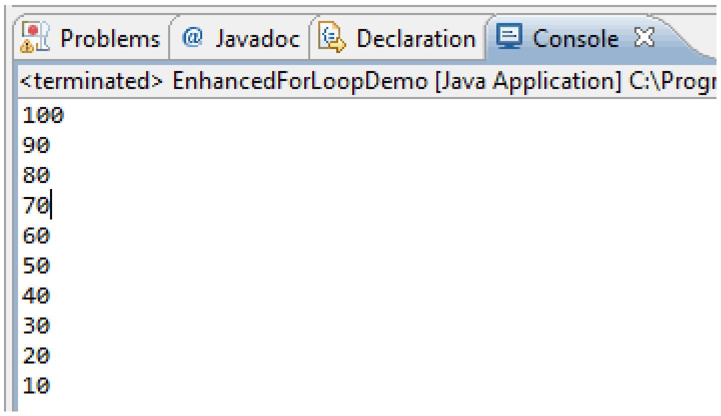
```
for(type itr-var : collection) statement-block
```

Here, type specifies the type and itr-var specifies the name of an iteration variable that will receive the elements from a collection, one at a time, from beginning to end. Because the iteration variable receives values from the collection, the type must be the same as (or compatible with) the elements stored in the collection. Thus, when iterating over arrays, the type must be compatible with the base type of the array.

Below example shows the use of enhanced for loop.

```
public class EnhancedForLoopDemo {

public static void main(String[] args) {

                int [] myArray = new int[10];

                int i=0;

                //Traditional for loop to populate

                for(int k=100; k>0; k=k-10, i++)

                {

                myArray[i]=k;

                }

                //Enhanced for loop to print elements of array

                for(int loopVal: myArray)

                {

                System.out.println(loopVal);

                }

        }

}
```

Output:



```
<terminated> EnhancedForLoopDemo [Java Application] C:\Progr
100
90
80
70
60
50
40
30
20
10
```

As this output shows, the enhanced for loop style automatically cycles through an array in sequence from the lowest index to the highest.

Summary

- A for loop is a special loop that is used when a definite number of loop iterations is required.

- For loop have 3 sections, loop variable initialization, testing loop control variable, updating loop control variable.
- Enhanced for loop can be used to iterate through Array or collections.

Java Branching Statements

Description

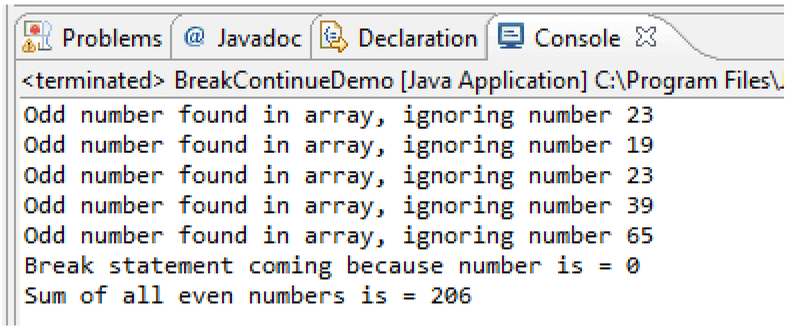
Java provides three branching statements break, continue and return. The break and continue in Java are two essential keyword beginners needs to familiar while using loops (for loop, while loop and do while loop). break statement in java is used to break the loop and transfers control to the line immediate outside of loop while continue is used to escape current execution (iteration) and transfers control back to the start of the loop. Both break and continue allow the programmer to create sophisticated algorithm and looping constructs.

In this java tutorial, we will see the example of break and continue statement in Java and some important points related to breaking the loop using label and break statement. break keyword can also be used inside switch statement to break current choice and if not used it can cause fall-through on switch. Both the break statement and the continue statement can be unlabeled or labeled. Although it's far more common to use a break and continue unlabeled.

Let’s understand unlabeled continue and break statement using java program. Below program will do the addition of all even numbers of array till it encounters 0 or negative number from an array.

```
public class BreakContinueDemo {

        public static void main(String[] args) {

                int [] numbers = {10,23,19,34,54,23,76,39,65,24,8,0,12,55};

                int sum =0;

                for(int i=0; i< numbers.length; i++){

                        if(numbers[i]<=0){

                                System.out.println("Break statement coming because number is =
"+ numbers[i]);

                                break;
                        }else if (numbers[i]%2 != 0){

                                System.out.println("Odd number found in array, ignoring number "
+ numbers[i]);

                                continue;

                        }else

                        sum = sum + numbers[i];

                }

                System.out.println("Sum of all even numbers is = " + sum);

        }

}
```

Output:



```
<terminated> BreakContinueDemo [Java Application] C:\Program Files\
Odd number found in array, ignoring number 23
Odd number found in array, ignoring number 19
Odd number found in array, ignoring number 23
Odd number found in array, ignoring number 39
Odd number found in array, ignoring number 65
Break statement coming because number is = 0
Sum of all even numbers is = 206
```

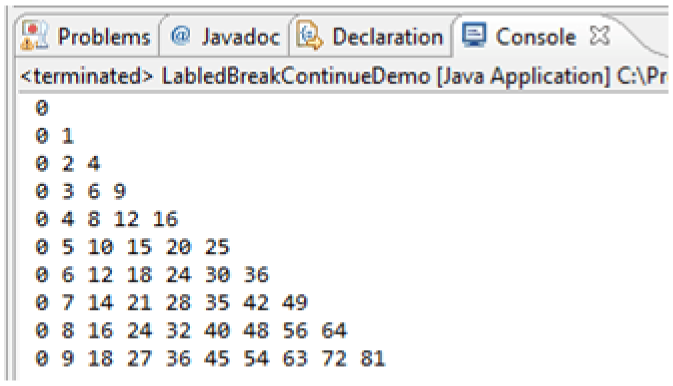
Labeled Statements

Although many statements in a Java program can be labeled, it's most common to use labels with loop statements like for or while, in conjunction with break and continue statements. A label statement must be placed just before the statement being labeled, and it consists of a valid identifier that ends with a colon (:).

You need to understand the difference between labeled and unlabeled break and continue. The labeled varieties are needed only in situations where you have nested loop and need to indicate which of the nested loops you want to break from,or from which of the nested loops you want to continue with the next iteration. A break statement will exit out of the labeled loop, as opposed to the innermost loop,if the break keyword is combined with a label.Here is an example program that uses continue to print a triangular multiplication table for 0 through 9.

```
public class LabledBreakContinueDemo {

        public static void main(String[] args) {

                int breaklimit = 9;
                outer: for (int i = 0; ; i++) {

                        for (int j = 0; j < 10; j++) {

                                if (j > i) {

                                        System.out.println();

                                        continue outer;

                                }

                                System.out.print(" " + (i * j));

                        }

                        if(i==breaklimit){

                                break outer;

                        }

                }

                System.out.println();

        }

}
```

Output:



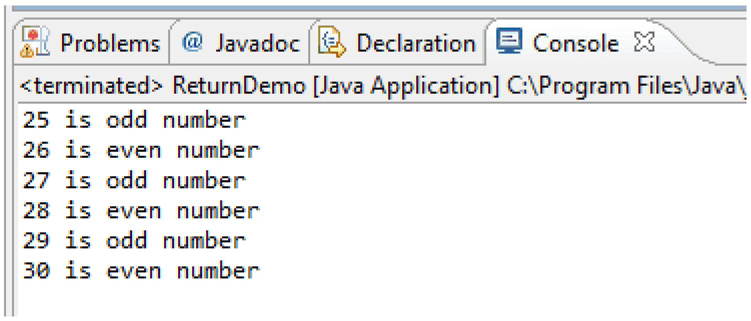
```
<terminated> LabledBreakContinueDemo [Java Application] C:\Pr
 0
 0 1
 0 2 4
 0 3 6 9
 0 4 8 12 16
 0 5 10 15 20 25
 0 6 12 18 24 30 36
 0 7 14 21 28 35 42 49
 0 8 16 24 32 40 48 56 64
 0 9 18 27 36 45 54 63 72 81
```

The return statement

The last control statement is return. The return statement is used to explicitly return from a method. That is, it causes program control to transfer back to the caller of the method. As such, it is categorized as a jump statement. At any time in a method, the return statement can be used to cause execution to branch back to the caller of the method. Thus, the return statement immediately terminates the method in which it is executed. The following example illustrates this point. In below program main () is calling method and checkEven() is called the method. Execution of checkEven() method ends when return statement encountered.

```
public class ReturnDemo {

        public static void main(String[] args) {

                for (int k =25; k< 31; k++){

                        new ReturnDemo().checkEven(k);

                }

        }

        public boolean checkEven(int a){

                if (a%2 == 0){

                        System.out.println(a + " is even number");

                        return true;

                }

                System.out.println(a + " is odd number");

                return false;

        }

}
```

Output:



```
<terminated> ReturnDemo [Java Application] C:\Program Files\Java\
25 is odd number
26 is even number
27 is odd number
28 is even number
29 is odd number
30 is even number
```

Summary

- Java provides 3 branching statement named break, continue and return.
- Branching statements are used to change the normal flow of execution based on some condition.
- The return statement is used to explicitly return from a method.

Exceptions in Java programming language

Introduction

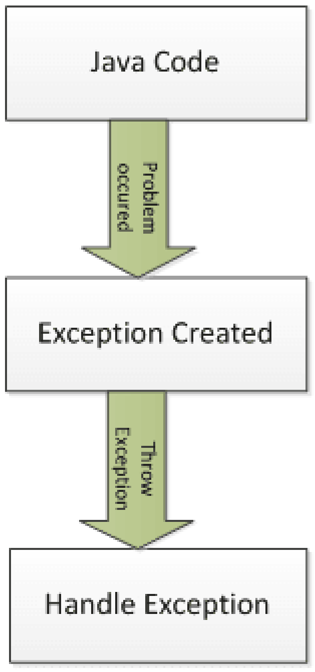
A Java exception is an object that describes an exceptional (that is, error) condition that has occurred in a piece of code. When an exceptional condition arises, an object representing that exception is created and thrown in the method that caused the error. That method may choose to handle the exception itself or pass it on. Either way, at some point, the exception is caught and processed.

The programs you write can generate many types of potential exceptions, such as when you do the following:

In Java there are three types of loops:

- You issue a command to read a file from a disk, but the file does not exist there.
- You attempt to write data to a disk, but the disk is full or unformatted.
- Your program asks for user input, but the user enters invalid data.
- The program attempts to divide a value by 0, access an array with a subscript that is too large or calculate a value that is too large for the answer's variable type.

These errors are called exceptions because, presumably, they are not usual occurrences; they are "exceptional." The object-oriented techniques to manage such errors comprise the group of methods known as exception handling.

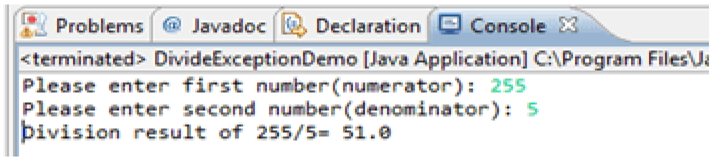


Exception handling works by transferring the execution of a program to an appropriate exception handler when an exception occurs. Let’s take an example program which will do take two numbers from user and print division result on screen. This might lead to exception condition if the denominator is zero.

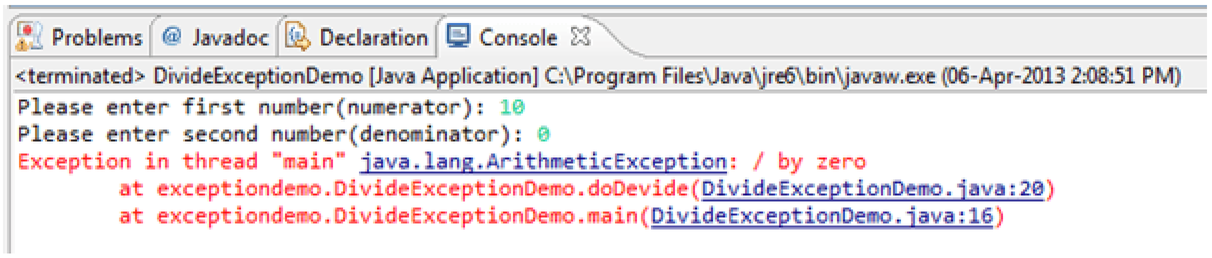
```
import java.util.Scanner;

public class DivideExceptionDemo {

        public static void main(String[] args) {

                //Scanner class is wrapper class of System.in object

                Scanner inputDevice = new Scanner(System.in);

                System.out.print("Please enter first number(numerator): ");

                int numerator = inputDevice.nextInt();

                System.out.print("Please enter second number(denominator): ");

                int denominator = inputDevice.nextInt();

                new DivideExceptionDemo().doDivide(numerator, denominator);

        }

        public void doDivide(int a, int b){

                float result = a/b;

                System.out.println("Division result of "+ a +"/"+b +"= " +result);

        }

}
```

Output:

Outputs based on user input combinations:



```
Problems  @ Javadoc  Declaration  Console
<terminated> DivideExceptionDemo [Java Application] C:\Program Files\Ja
Please enter first number(numerator): 255
Please enter second number(denominator): 5
Division result of 255/5= 51.0
```



```
Problems  @ Javadoc  Declaration  Console
<terminated> DivideExceptionDemo [Java Application] C:\Program Files\Java\jre6\bin\javaw.exe (06-Apr-2013 2:08:51 PM)
Please enter first number(numerator): 10
Please enter second number(denominator): 0
Exception in thread "main" java.lang.ArithmeticException: / by zero
	at exceptiondemo.DivideExceptionDemo.doDevide(DivideExceptionDemo.java:20)
	at exceptiondemo.DivideExceptionDemo.main(DivideExceptionDemo.java:16)
```

Handling Exceptions:

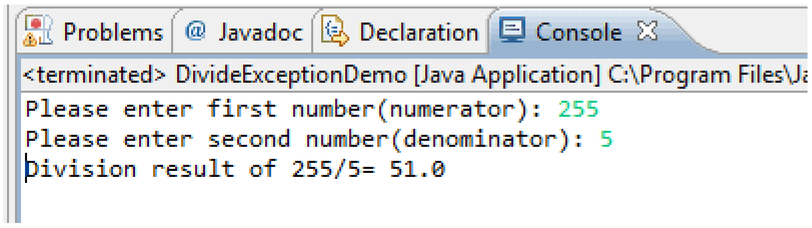
There are two ways of handling the exception, first catch the exception and take corrective action or throws the exception to the calling method which will force the calling method to handle it.

1. In above program the execution is unexpected and ended in an error condition in case of the denominator is zero. We can avoid this by handling exception using a try-catch block. Let's update program for exception handling. Here we will write exception prone code inside try block (guarded block) and catch block will follow the try block.

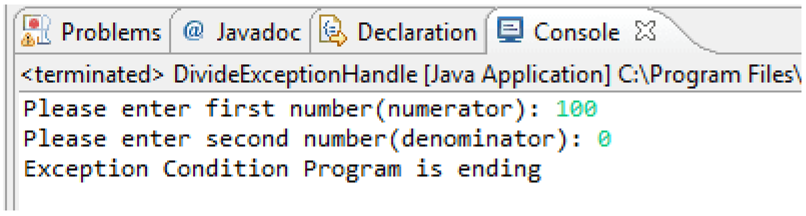
```java
import java.util.Scanner;

public class DivideExceptionHandle {

        public static void main(String[] args) {

                Scanner inputDevice = new Scanner(System.in);

                System.out.print("Please enter first number(numerator): ");

                int numerator = inputDevice.nextInt();

                System.out.print("Please enter second number(denominator): ");

                int denominator = inputDevice.nextInt();

                new DivideExceptionHandle().doDivide(numerator, denominator);

        }

        public void doDivide(int a, int b){
                try{

                        float result = a/b;

                        System.out.println("Division result of "+ a +"/"+b +"= " +result);

                }catch(ArithmeticException e){

                        System.out.println("Exception Condition Program is ending ");

                }

        }

}
```

Output:

Outputs based on user input combination:



```
<terminated> DivideExceptionDemo [Java Application] C:\Program Files\J
Please enter first number(numerator): 255
Please enter second number(denominator): 5
Division result of 255/5= 51.0
```



```
<terminated> DivideExceptionHandle [Java Application] C:\Program Files\
Please enter first number(numerator): 100
Please enter second number(denominator): 0
Exception Condition Program is ending
```

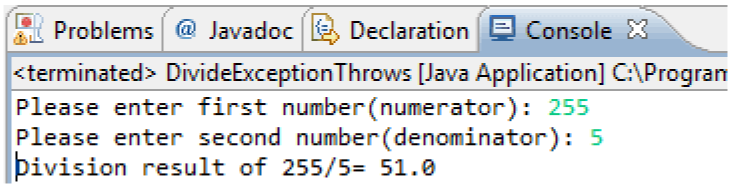
2. When a Java method is going to throw an exception, to indicate that as part of the method signature 'throws' keyword should be used followed by the exception. It means that the caller of this method should handle the exception given in the throws clause. There can be multiple exceptions declared to be thrown by a method. If the caller of that method does not handle the exception, then it propagates to one level higher in the method call stack to the previous caller and similarly till it reaches base of the the method call stack which will be the java's runtime system. For this approach, we use throws keyword in method declaration which will instruct the compiler to handle exception using try-catch block. When we add throws keyword in divide method declaration compile time error will be seen as below,

```
import java.util.Scanner;

public class DivideExceptionThrows {

        public static void main(String[] args){
                Scanner inputDevice = new Scanner(System.in);

                System.out.print("Please enter first number(numerator): ");

                int numerator = inputDevice.nextInt();

                System.out.print("Please enter second number(denominator): ");

                int denominator = inputDevice.nextInt();

                try {

                        new DivideExceptionThrows().doDivide(numerator, denominator);

                } catch (Exception e) {

                        System.out.println("Exception Condition Program is ending ");

                }

        }

        public void doDivide(int a, int b) throws Exception{

                        float result = a/b;

                        System.out.println("Division result of "+ a +"/"+b +"= " +result);

        }

}
```

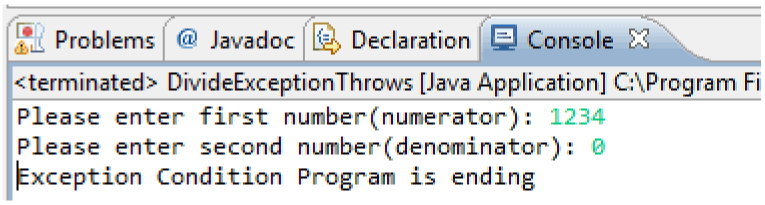
As you can see either we can surround the code with try-catch block or we can re-throw it to be handled by calling the method. In this case, we are calling a method from the main() method so if we re-throw the exception it would be handled by JVM. Let us update the code and see output based on input combination

Output:

Outputs based on user input combination:



```
<terminated> DivideExceptionThrows [Java Application] C:\Progran
Please enter first number(numerator): 255
Please enter second number(denominator): 5
Division result of 255/5= 51.0
```



```
<terminated> DivideExceptionThrows [Java Application] C:\Program Fi
Please enter first number(numerator): 1234
Please enter second number(denominator): 0
Exception Condition Program is ending
```

Nested Try-catch block:

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

```
public class NestedTryblockDemo {

        public static void main(String[] args) {

                try{

                        //some code      which can throw Exception

                        try {

                                //Some code which can throw Arithmatic exception

                                try {

                                        //Some code which can throw number format exception

                                }catch(NumberFormatException n){

                                        //Number format Exception handling

                                }

                        }catch(ArithmeticException a){

                                //ArithmeticException Handling

                        }

                }catch(Exception e ){

                        //General Exception(SuperClass of all Exception) Handling
                }

        }

}
```

Use of finally block:

When you have actions you must perform at the end of a try...catch sequence, you can use a finally block. The code within a finally block executes regardless of whether the preceding try block identifies an Exception. Usually, you use a finally block to perform cleanup tasks that must happen whether or not any Exceptions occurred, and whether or not any Exceptions that occurred were caught. In an application where database connection, files are being operated, we need to take of closing those resources in exceptional condition as well as normal condition.

```
public class TryCatchFinally {

        public void Demo() {

                try {

                        // statements to try

                } catch (Exception e) {

                        // actions that occur if exception was thrown

                } finally {

                        // actions that occur whether catch block executed or not

                }

        }

}
```

Summary:

We have learned about how exceptions are generated and various ways of handling exceptions. Catching exception or propagating exceptions. We have learned keywords like try, catch, finally, throws and programmatic use of these keywords.

Exceptions in Java programming language

Introduction

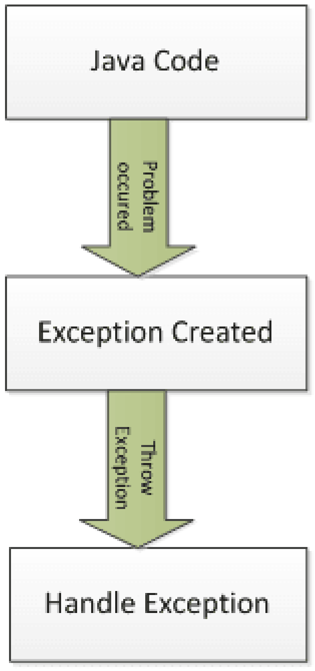
A Java exception is an object that describes an exceptional (that is, error) condition that has occurred in a piece of code. When an exceptional condition arises, an object representing that exception is created and thrown in the method that caused the error. That method may choose to handle the exception itself or pass it on. Either way, at some point, the exception is caught and processed.

The programs you write can generate many types of potential exceptions, such as when you do the following:

In Java there are three types of loops:

- You issue a command to read a file from a disk, but the file does not exist there.
- You attempt to write data to a disk, but the disk is full or unformatted.
- Your program asks for user input, but the user enters invalid data.
- The program attempts to divide a value by 0, access an array with a subscript that is too large or calculate a value that is too large for the answer's variable type.

These errors are called exceptions because, presumably, they are not usual occurrences; they are "exceptional." The object-oriented techniques to manage such errors comprise the group of methods known as exception handling.

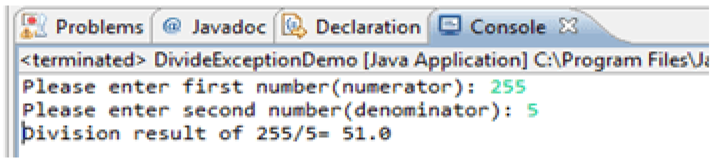


Exception handling works by transferring the execution of a program to an appropriate exception handler when an exception occurs. Let’s take an example program which will do take two numbers from user and print division result on screen. This might lead to exception condition if the denominator is zero.

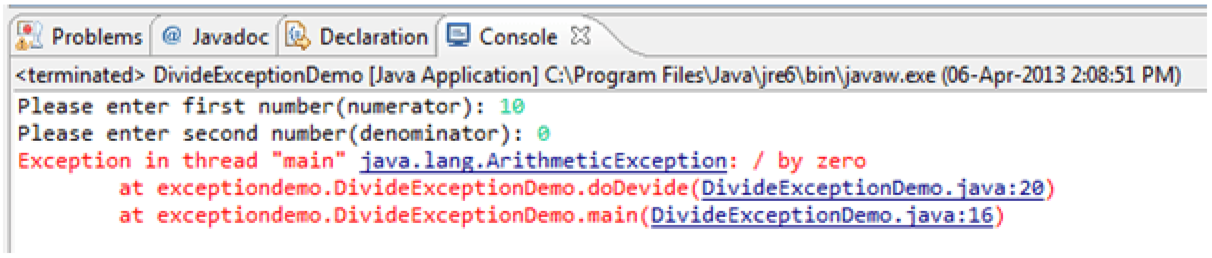
```
import java.util.Scanner;

public class DivideExceptionDemo {

        public static void main(String[] args) {

                //Scanner class is wrapper class of System.in object

                Scanner inputDevice = new Scanner(System.in);

                System.out.print("Please enter first number(numerator): ");

                int numerator = inputDevice.nextInt();

                System.out.print("Please enter second number(denominator): ");

                int denominator = inputDevice.nextInt();

                new DivideExceptionDemo().doDivide(numerator, denominator);

        }

        public void doDivide(int a, int b){

                float result = a/b;

                System.out.println("Division result of "+ a +"/"+b +"= " +result);

        }

}
```

Output:

Outputs based on user input combinations:



```
<terminated> DivideExceptionDemo [Java Application] C:\Program Files\J
Please enter first number(numerator): 255
Please enter second number(denominator): 5
Division result of 255/5= 51.0
```



```
<terminated> DivideExceptionDemo [Java Application] C:\Program Files\Java\jre6\bin\javaw.exe (06-Apr-2013 2:08:51 PM)
Please enter first number(numerator): 10
Please enter second number(denominator): 0
Exception in thread "main" java.lang.ArithmeticException: / by zero
	at exceptiondemo.DivideExceptionDemo.doDevide(DivideExceptionDemo.java:20)
	at exceptiondemo.DivideExceptionDemo.main(DivideExceptionDemo.java:16)
```

Handling Exceptions:

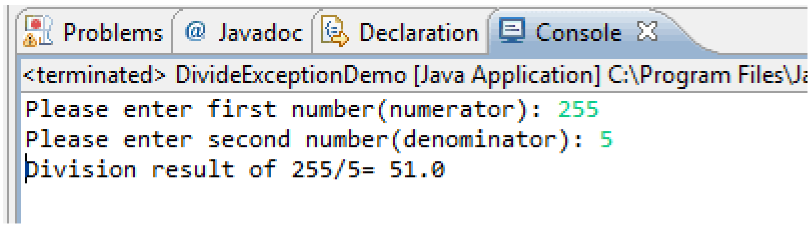
There are two ways of handling the exception, first catch the exception and take corrective action or throws the exception to the calling method which will force the calling method to handle it.

1. In above program the execution is unexpected and ended in an error condition in case of the denominator is zero. We can avoid this by handling exception using a try-catch block. Let’s update program for exception handling. Here we will write exception prone code inside try block (guarded block) and catch block will follow the try block.

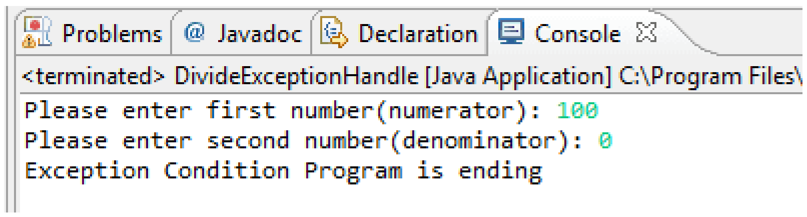
```
import java.util.Scanner;

public class DivideExceptionHandle {

        public static void main(String[] args) {

                Scanner inputDevice = new Scanner(System.in);

                System.out.print("Please enter first number(numerator): ");

                int numerator = inputDevice.nextInt();

                System.out.print("Please enter second number(denominator): ");

                int denominator = inputDevice.nextInt();

                new DivideExceptionHandle().doDivide(numerator, denominator);

        }

        public void doDivide(int a, int b){

                try{

                        float result = a/b;

                        System.out.println("Division result of "+ a +"/"+b +"= " +result);

                }catch(ArithmeticException e){
                        System.out.println("Exception Condition Program is ending ");
                }
        }
}
```

Output:

Outputs based on user input combination:



```
<terminated> DivideExceptionDemo [Java Application] C:\Program Files\Ja
Please enter first number(numerator): 255
Please enter second number(denominator): 5
Division result of 255/5= 51.0
```



```
<terminated> DivideExceptionHandle [Java Application] C:\Program Files\
Please enter first number(numerator): 100
Please enter second number(denominator): 0
Exception Condition Program is ending
```

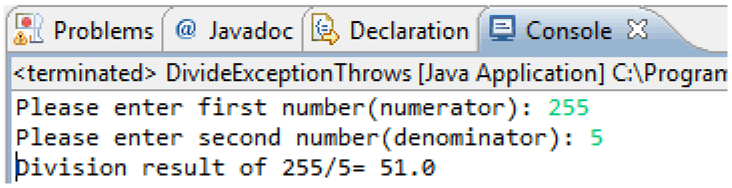
2. When a Java method is going to throw an exception, to indicate that as part of the method signature ‘throws‘ keyword should be used followed by the exception. It means that the caller of this method should handle the exception given in the throws clause. There can be multiple exceptions declared to be thrown by a method. If the caller of that method does not handle the exception, then it propagates to one level higher in the method call stack to the previous caller and similarly till it reaches base of the the method call stack which will be the java’s runtime system. For this approach, we use throws keyword in method declaration which will instruct the compiler to handle exception using try-catch block. When we add throws keyword in divide method declaration compile time error will be seen as below,

```
import java.util.Scanner;
public class DivideExceptionThrows {
        public static void main(String[] args){
                Scanner inputDevice = new Scanner(System.in);
                System.out.print("Please enter first number(numerator): ");
                int numerator = inputDevice.nextInt();
                System.out.print("Please enter second number(denominator): ");
                int denominator = inputDevice.nextInt();

                try {

                        new DivideExceptionThrows().doDivide(numerator, denominator);

                } catch (Exception e) {

                        System.out.println("Exception Condition Program is ending ");

                }

        }

        public void doDivide(int a, int b) throws Exception{

                        float result = a/b;

                        System.out.println("Division result of "+ a +"/"+b +"= " +result);

        }

}
```

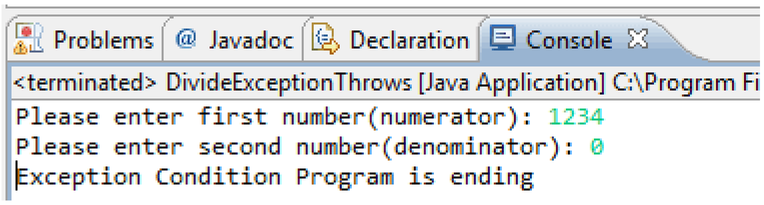
As you can see either we can surround the code with try-catch block or we can re-throw it to be handled by calling the method. In this case, we are calling a method from the main() method so if we re-throw the exception it would be handled by JVM. Let us update the code and see output based on input combination

Output:

Outputs based on user input combination:



```
<terminated> DivideExceptionThrows [Java Application] C:\Progran
Please enter first number(numerator): 255
Please enter second number(denominator): 5
Division result of 255/5= 51.0
```



```
<terminated> DivideExceptionThrows [Java Application] C:\Program Fi
Please enter first number(numerator): 1234
Please enter second number(denominator): 0
Exception Condition Program is ending
```

Nested Try-catch block:

The try statement can be nested. That is, a try statement can be inside the block of another try.Each time a try statement is entered, the context of that exception is pushed on the stack.

If an inner try statement does not have a catch handler for a particular exception, the stack is unwound and the next try statement's catch handlers are inspected for a match. This continues until one of the catch statements succeeds, or until all of the nested try statements are exhausted.If no catch statement matches, then the Java runtime system will handle the exception. Below is the syntax of nested try-catch block.

```
public class NestedTryblockDemo {

        public static void main(String[] args) {

                try{

                        //some code      which can throw Exception

                        try {

                                //Some code which can throw Arithmatic exception

                                try {

                                        //Some code which can throw number format exception

                                }catch(NumberFormatException n){

                                        //Number format Exception handling

                                }

                        }catch(ArithmeticException a){

                                //ArithmeticException Handling

                        }

                }catch(Exception e ){

                        //General Exception(SuperClass of all Exception) Handling

                }

        }

}
```

Use of finally block:

When you have actions you must perform at the end of a try...catch sequence, you can use a finally block. The code within a finally block executes regardless of whether the preceding try block identifies an Exception. Usually, you use a finally block to perform cleanup tasks that must happen whether or not any Exceptions occurred, and whether or not any Exceptions that occurred were caught. In an application where database connection, files are being operated, we need to take of closing those resources in exceptional condition as well as normal condition.

```
public class TryCatchFinally {

        public void Demo() {

                try {

                        // statements to try

                } catch (Exception e) {

                        // actions that occur if exception was thrown

                } finally {

                        // actions that occur whether catch block executed or not

                }

        }

}
```

Summary:

We have learned about how exceptions are generated and various ways of handling exceptions. Catching exception or propagating exceptions. We have learned keywords like try, catch, finally, throws and programmatic use of these keywords.

Custom Exceptions

Introduction

We have talked about how to handle exceptions when they're produced by calling methods in Java APIs. Java also lets you create and use custom exceptions—classes of your own exception as per application need which will be used to represent errors. Normally, you create a custom exception to represent some type of error within your application—to give a new, distinct meaning to one or more problems that can occur within your code. You may do this to show similarities between errors that exist in several places throughout your code, to differentiate one or more errors from similar problems that could occur as your code runs, or to give special meaning to a group of errors within your application.

It’s fairly easy to create and use a custom exception. There are three basic steps that you need to follow. We will explain this by an example of bank account balance. Here we want to have flexi-deposit functionality i.e. when account balance goes beyond 20k, a new deposit will be created.

Define the exception class

You typically represent a custom exception by defining a new class. In many cases, all you need to do is to create a subclass of an existing exception class:

Java Code:

```
public class AccountBalanceException extends Exception {

        private float accountBalance ;

        public AccountBalanceException(float f){

                super();

                this.accountBalance =f;

        }

        public AccountBalanceException(String message){

                super(message);

         }

        public float getAccountBalance(){

                return accountBalance;

        }

}
```

At a minimum, you need to subclass Throwable or one of its subclasses. Often,you’ll also define one or more constructors to store information like an error message in the object, as shown in lines 6-12. Our exception class AccountBalanceException has two constructors. One with String argument and the second constructor is having float argument. When you subclass any exception, you automatically inherit some standard features from the Throwable class, such as:

- Error message

- Stack trace
- Exception wrapping

Declare that your exception-producing method throws your custom exception

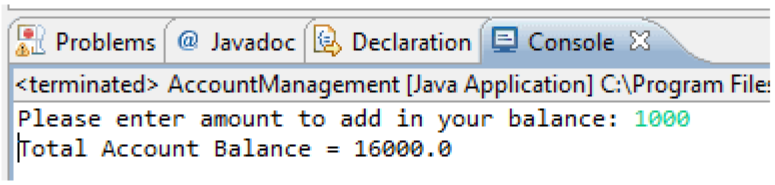
Using throws keyword we can declare the method which might be exception producing. In order to use a custom exception, you must show classes that call your code that they need to plan for this new type of exception. You do this by declaring that one or more of your methods throws the exception. Below is code for account balance management. AccountManagement class has main() method and two utility methods addAmount() and createFixDeposit(). Here we have assumecurrentBalance as Rs. 15000. If account balance goes beyond Rs. 20000, the amount above 20,000 will be passed to make fix-deposit.

```
import java.util.Scanner;

public class AccountManagement {

        private float currentBalance =15000f;

        public static void main(String[] args) {

                Scanner inputDevice = new Scanner(System.in);

                System.out.print("Please enter amount to add in your balance: ");

                float newAmount = inputDevice.nextFloat();

                try{

                        float totalAmount = new AccountManagement().AddAmount(newAmount);

                        System.out.println("Total Account Balance = "+ totalAmount);

                }catch (AccountBalanceException a){

                        float fdAmount = a.getAccountBalance() - 20000 ;

                        System.out.println("Your account balance is more than 20K now, So
creating FD of Amount: "+ fdAmount);

                        new AccountManagement().createFixDeposit(fdAmount);

                        System.out.println("Account Balance = "+20000);

                }

        }

        public float AddAmount(float amount) throws AccountBalanceException{

                float total = currentBalance+amount;

                if (total>20000){

                        throw new AccountBalanceException(total);
                }

                return total;

        }

        public void createFixDeposit(float fxAmount){

                //Implimentation of FD creation

        }

}


class AccountBalanceException extends Exception {

        private float accountBalance ;

        public AccountBalanceException(float f){

                super();

                this.accountBalance =f;

        }

        public AccountBalanceException(String message){

                super(message);

         }

        public float getAccountBalance(){

                return accountBalance;

        }

}
```

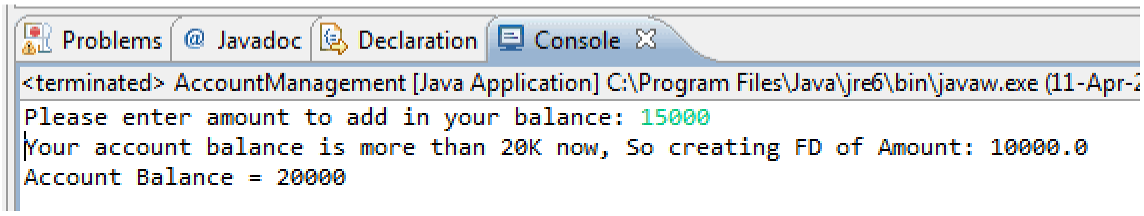
Find the point(s) of failure in your error-producing method, create the exception and **dispatch it using the keyword "throw"**

The third and final step is to actually create the object and propagate it through the system. To do this, you need to know where your code needs special processing in the method. Here, Throwable Instance must be an object of type Throwable or a subclass of Throwable. There are two ways you can obtain a Throwable object: using a parameter in a catch clause or creating one with the new operator. The flow of execution stops immediately after the throw statement; any subsequent statements are not executed. The nearest enclosing try block is inspected to see if it has a catch statement that matches the type of exception. If it does find a match, control is transferred to that statement. If not, then the next enclosing try statement is inspected, and so on.

Output of above program based on various input parameters as below,



```
<terminated> AccountManagement [Java Application] C:\Program File
Please enter amount to add in your balance: 1000
Total Account Balance = 16000.0
```



```
<terminated> AccountManagement [Java Application] C:\Program Files\Java\jre6\bin\javaw.exe (11-Apr-
Please enter amount to add in your balance: 15000
Your account balance is more than 20K now, So creating FD of Amount: 10000.0
Account Balance = 20000
```

Summary:

- Java provides exception classes for most of the common type of errors as well as provides a mechanism to define custom exception classes.
- Three steps to create and use custom exception class (1) Define exception class (2) Declare exception prone method with throws keyword (3) Check condition to throw new exception object to be handled by calling the method.

The String Class

Introduction

In Java String is an object represented as a sequence of characters. Handling "strings" of characters is a fundamental aspect of most programming languages. Implementing strings as built-in objects allows Java to provide a full complement of features that make string handling convenient. For example, Java has methods to compare two strings, search for a substring, concatenate two strings, and change the case of letters within a string. String class has 11 constructors and over 40 utility methods. String objects are immutable means once a String object has been created, you cannot change the characters that comprise that string. This is not a limitation of String class but it will be very helpful in a multithreaded application. You can perform all the operation on your String object only things changed is new String object will be created each time. This approach is used because fixed, immutable strings can be implemented more efficiently than changeable ones.

Creating String Object

In Java, strings are objects. Just like other objects, you can create an instance of a String with the new keyword, as follows:

Step 1

```
String name = new String(“Amit”);
String name = "Amit";
```

This line of code creates a new object of class String and assigns it to the reference variable name. There are some vital differences between these options that we'll discuss later, but what they have in common is that they both create a new String object, with a value of "Amit", and assign it to a reference variable name. Now let's say that you want a second reference to the String object referred to by s, and we are calling concat method on name object, this will create new String object with value “Amit Himani”, while reference s2 still points to object “Amit”.

Step 2

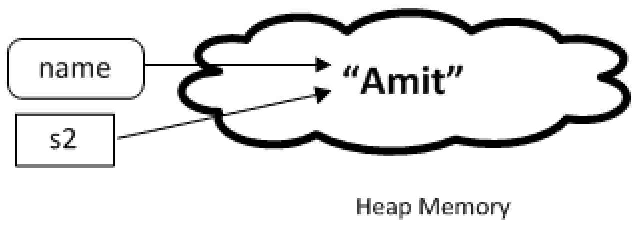
```
String s2 = name; // refer s2 to the same String as name
```

Step 3

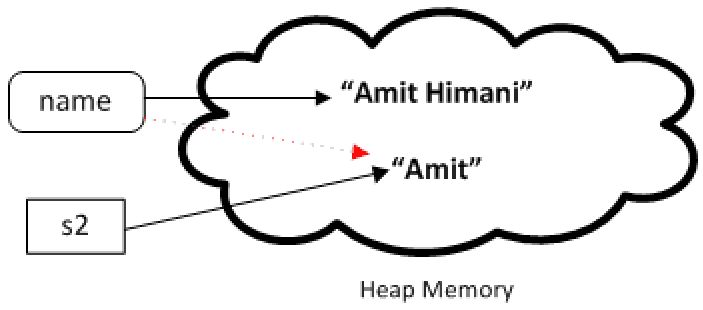
```
name = name.concat(" Himani"); // the concat() method 'appends' a literal
to the end
```

Let’s understand this concept graphically,

Step 1 and Step 2



Step 3



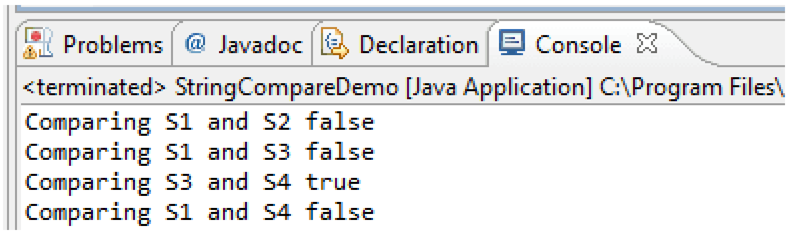
As we know that the strings within objects of type String are unchangeable means that the contents of the String instance cannot be changed after it has been created. However, a variable declared as a String reference can be changed to point at some other String object at any time.

Comparing String Objects Reference

As we have seen earlier String object can be created using constructor using a new keyword as well as we can use String literals. There is a fundamental difference between both ways of String creation. Let's understand this with help of Java program. We are creating two String object using literals and two objects by calling String class constructor. We are comparing String objects using == operator.

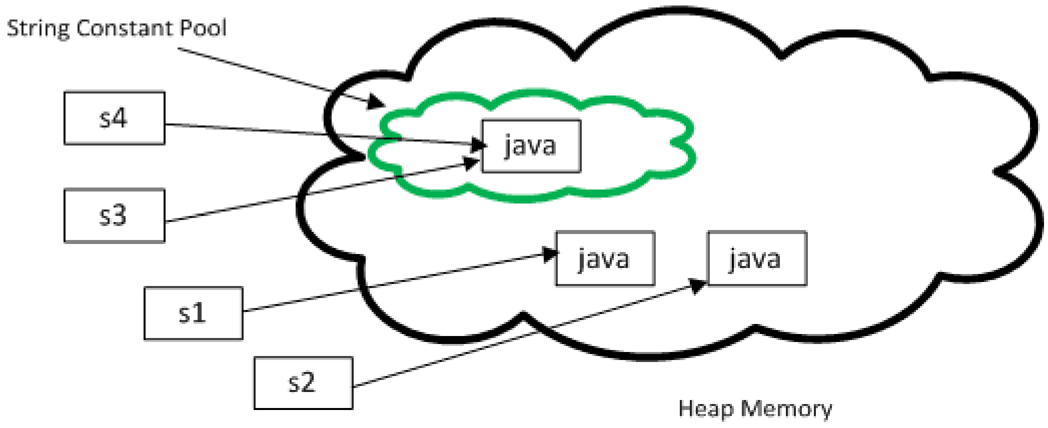
```java
public class StringDeepCompareDemo {

        public static void main(String[] args) {


                String s1 = new String("java");

                String s2 = new String("java");

                String s3 = "java";

                String s4 = "java";


                System.out.print("Comparing S1 and S2 ");

                System.out.println(s1==s2);

                System.out.print("Comparing S1 and S3 ");

                System.out.println(s1==s3);

                System.out.print("Comparing S3 and S4 ");

                System.out.println(s3==s4);

                System.out.print("Comparing S1 and S4 ");

                System.out.println(s1==s4);
        }


}
```

Output:



```
<terminated> StringCompareDemo [Java Application] C:\Program Files\
Comparing S1 and S2 false
Comparing S1 and S3 false
Comparing S3 and S4 true
Comparing S1 and S4 false
```

As shown above, String objects created using literals are passing equality test rest all are failing. For the “literal” String assignment if the assignment value is identical to another String assignment value created then a new String object is not created. A reference to the existing String object is returned. Below picture explains this concept.



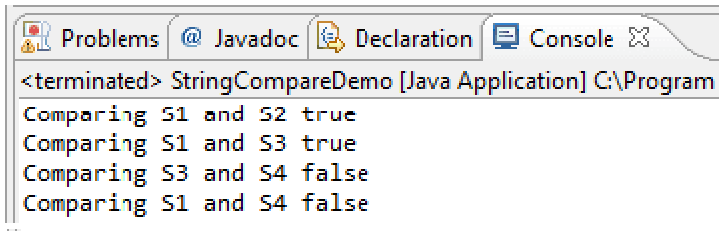
One of the key goals of any good programming language is to make efficient use of memory. As applications grow, it's very common for String literals to occupy large amounts of a program's memory, and there is often a lot of redundancy within the universe of String literals for a program. To make Java more memory efficient, the JVM sets aside a special area of memory called the "String constant pool." When the compiler encounters a String literal, it checks the pool to see if an identical String already exists. If a match is found, the reference to the new literal is directed to the existing String, and no new String literal object is created.

Comparing String Object Values

String class provides many utility methods one of them is equals method. The equals method is used to compare object’s value. There is slight difference between “==” operator and equals() method. Comparison using “==” is called shallow comparison because of == returns true, if the variable reference points to the same object in memory. Comparison using equals() method is called deep comparison because it will compare attribute values. Let’s understand this concept by java program.

```
public class StringCompareDemo {

public static void main(String[] args)

 {

 String s1 = new String("Hello");

 String s2 = new String("Hello");

 String s3 = "Hello"; String s4 = "Java";

 System.out.print("Comparing S1 and S2 ");

 System.out.println(s1.equals(s2));

 System.out.print("Comparing S1 and S3 ");

 System.out.println(s1.equals(s3));

 System.out.print("Comparing S3 and S4 ");

 System.out.println(s3.equals(s4));

 System.out.print("Comparing S1 and S4 ");

 System.out.println(s1.equals(s4));

 }

}
```

Output:



```
<terminated> StringCompareDemo [Java Application] C:\Program
Comparing S1 and S2 true
Comparing S1 and S3 true
Comparing S3 and S4 false
Comparing S1 and S4 false
```

Summary:

- In java String is an object represented by sequence of characters
- A sequence of characters enclosed in double quotation marks is a literal string. You can create a String object by using the keyword new and the String constructor.

- String object comparison can be done using == operator which is called shallow comparison. String object comparison using equals() method is called deep comparison.

Exploring Methods of String Class

Introduction

String manipulation is arguably one of the most common activities in computer programming. String class has a variety of methods for string manipulation. We will discuss basic methods with examples.

public char charAt(int index)

This method requires an integer argument that indicates the position of the character that the method returns.This method returns the character located at the String's specified index. Remember, String indexes are zero-based—for example,

```
String x = "airplane";
System.out.println( x.charAt(2) ); // output is 'r'
```

public String concat(String s)

This method returns a String with the value of the String passed in to the method appended to the end of the String used to invoke the method—for example,

```
String x = "book";
System.out.println( x.concat(" author") ); // output is "book author"
```

The overloaded + and += operators perform functions similar to the concat()method—for example,

```
String x = "library";
System.out.println( x + " card"); // output is "library card"
String x = "United";
x += " States"
System.out.println( x ); // output is "United States"
```

public boolean equalsIgnoreCase(String s)

This method returns a boolean value (true or false) depending on whether the value of the String in the argument is the same as the value of the String used to invoke the method. This method will return true even when characters in the String objects being compared have differing cases—for example,

```
String x = "Exit";
System.out.println( x.equalsIgnoreCase("EXIT")); // is "true"
System.out.println( x.equalsIgnoreCase("tixe")); // is "false"
```

public int length()

This method returns the length of the String used to invoke the method—for example,

```
String x = "01234567";
System.out.println( x.length() ); // returns "8"
```

public String replace(char old, char new)

This method returns a String whose value is that of the String used to invoke the method, updated so that any occurrence of the char in the first argument is replaced by the char in the second argument—for example,

```
String x = "oxoxoxox";
System.out.println( x.replace('x', 'X') ); // output is  "oXoXoXoX"
```

public String substring(int begin)/ public String substring(int begin, int end)

The substring() method is used to return a part (or substring) of the String used to invoke the method. The first argument represents the starting location (zero-based) of the substring. If the call has only one argument, the substring returned will include the characters to the end of the original String. If the call has two arguments, the substring returned will end with the character located in the nth position of the original String where n is the second argument. Unfortunately, the ending argument is not zero-based, so if the second argument is 7, the last character in the returned String will be in the original String's 7 position, which is index 6. Let's look at some examples:

```
String x = "0123456789"; // the value of each char is the same as its
index!
System.out.println( x.substring(5) ); // output is "56789"
System.out.println( x.substring(5, 8)); // output is "567"
```

public String toLowerCase()

This method returns a String whose value is the String used to invoke the method, but with any uppercase characters converted to lowercase—for example,

```
String x = "A New Java Book";
System.out.println( x.toLowerCase() ); // output is "a new java book"
```

public String toUpperCase()

This method returns a String whose value is the String used to invoke the method, but with any lowercase characters converted touppercase—for example,

```
String x = "A New Java Book";
System.out.println( x.toUpperCase() ); // output is"A NEW JAVA BOOK"
```

public String trim()

This method returns a String whose value is the String used to invoke the method, but with any leading or trailing blank spaces removed—for example,

```
String x = " hi ";
System.out.println( x + "x" ); // result is" hi x"
System.out.println(x.trim() + "x"); // result is "hix"
```

public char[ ] toCharArray( )

This method will produce an array of characters from characters of String object. For example

```
String s = “Java”;
Char [] arrayChar = s.toCharArray();  //this will produce array of size 4
```

public boolean contains(“searchString”)

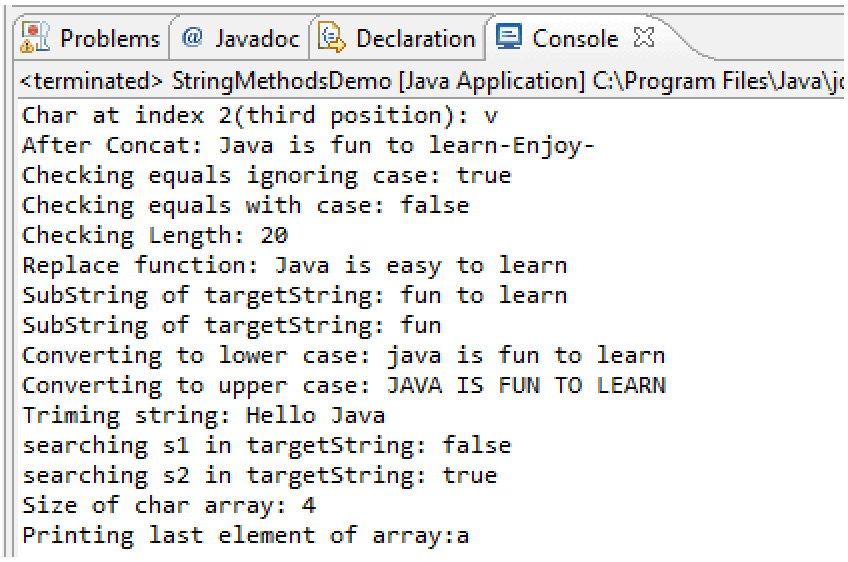
This method returns true of target String is containing search String provided in the argument. For example-

```
String x = “Java is programming language”;
System.out.println(x.contains(“Amit”)); // This will print false
System.out.println(x.contains(“Java”)); // This will print true
```

Below program demonstrate all above methods.

```
public class StringMethodsDemo {

        public static void main(String[] args) {

                String targetString = "Java is fun to learn";

                String s1= "JAVA";

                String s2= "Java";

                String s3 = "  Hello Java  ";


                System.out.println("Char at index 2(third position): " + targetString.charAt(2));

                System.out.println("After Concat: "+ targetString.concat("-Enjoy-"));

                System.out.println("Checking equals ignoring case: " +s2.equalsIgnoreCase(s1));

                System.out.println("Checking equals with case: " +s2.equals(s1));
                System.out.println("Checking Length: "+ targetString.length());

                System.out.println("Replace function: "+ targetString.replace("fun", "easy"));

                System.out.println("SubString of targetString: "+ targetString.substring(8));

                System.out.println("SubString of targetString: "+ targetString.substring(8, 12));

                System.out.println("Converting to lower case: "+ targetString.toLowerCase());

                System.out.println("Converting to upper case: "+ targetString.toUpperCase());

                System.out.println("Triming string: " + s3.trim());

                System.out.println("searching s1 in targetString: " + targetString.contains(s1));

                System.out.println("searching s2 in targetString: " + targetString.contains(s2));


                char [] charArray = s2.toCharArray();

                System.out.println("Size of char array: " + charArray.length);

                System.out.println("Printing last element of array: " + charArray[3]);


        }


}
```

Output:



```
<terminated> StringMethodsDemo [Java Application] C:\Program Files\Java\j
Char at index 2(third position): v
After Concat: Java is fun to learn-Enjoy-
Checking equals ignoring case: true
Checking equals with case: false
Checking Length: 20
Replace function: Java is easy to learn
SubString of targetString: fun to learn
SubString of targetString: fun
Converting to lower case: java is fun to learn
Converting to upper case: JAVA IS FUN TO LEARN
Triming string: Hello Java
searching s1 in targetString: false
searching s2 in targetString: true
Size of char array: 4
Printing last element of array:a
```

Summary

- String manipulation is one of the most widely performed activities in java programming
- Java library is having various built-in methods like substring, concat, replace, converting to uppercase or lowercase etc

StringBuffer / StringBuilder Class

Introduction

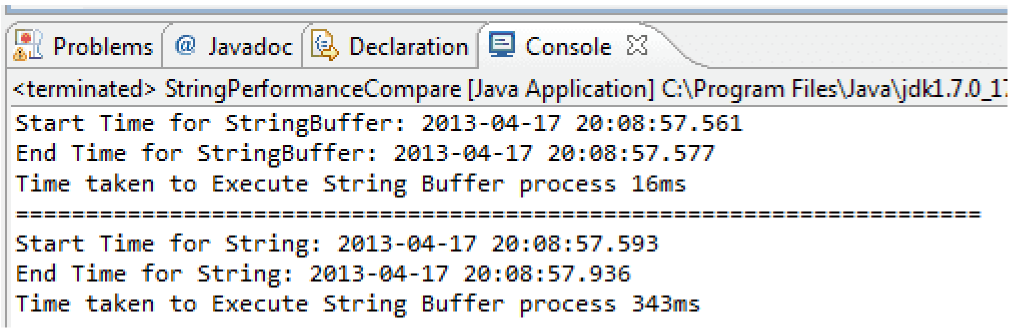
In Java String is immutable objects means once created it cannot be changed, reference will point to a new object. If our application has string manipulation activity, then there will be many discarded string object in heap memory which might result in performance impact. To circumvent these limitations, you can use either the StringBuilder or StringBuffer class. You use one of these classes, which are alternatives to the String class, when you know a String will be modified; usually, you can use a StringBuilder or StringBuffer object anywhere you would use a String. StringBuffer may have characters and substrings inserted in the middle or appended to the end. StringBuffer will automatically grow to make room for such additions and often has more characters pre-allocated than are actually needed, to allow room for growth. Similar to String class, these two classes are part of the java.lang package and are automatically imported into every program. The classes are identical except for the following:

- StringBuilder is more efficient.
- StringBuffer is thread safe because of all synchronized methods.

Let us see below program which is comparing the performance of String and String buffer objects. In this program, we are creating new string object interactively inside for loop. In next program, we are replacing String object with StringBuffer. Application time is calculated and displayed in a millisecond.

```
import java.util.Date;

import java.sql.Timestamp;

public class StringPerformanceCompare {


        public static void main(String[] args) {

                Date sDate = new Date();

                long sTime = sDate.getTime();

                System.out.println("Start Time for StringBuffer: " + new Timestamp(sTime));

                StringBuffer s = new StringBuffer("AA");

                for (int i=0; i< 10000; i++){

                                s.append(i);

                }

                Date eDate = new Date();

                long eTime = eDate.getTime();

                System.out.println("End Time for StringBuffer: " + new Timestamp(eTime));

                System.out.println("Time taken to Execute StringBuffer process " + (eTime-sTime)
+ "ms");

        System.out.println("=================================================================
");

                Date strDate = new Date();

                long strTime = strDate.getTime();

                System.out.println("Start Time for String: " + new Timestamp(strTime));

                String str = new String("AA");

                for (int i=0; i< 10000; i++){

                                str+=i;

                }

                Date eStrDate = new Date();

                long eStrTime = eStrDate.getTime();

                System.out.println("End Time for String: " + new Timestamp(eStrTime));
                System.out.println("Time taken to Execute String process " + (eStrTime-strTime) +
"ms");

        }

}
```

Output:



```
<terminated> StringPerformanceCompare [Java Application] C:\Program Files\Java\jdk1.7.0_1
Start Time for StringBuffer: 2013-04-17 20:08:57.561
End Time for StringBuffer: 2013-04-17 20:08:57.577
Time taken to Execute String Buffer process 16ms
=====================================================================
Start Time for String: 2013-04-17 20:08:57.593
End Time for String: 2013-04-17 20:08:57.936
Time taken to Execute String Buffer process 343ms
```

As seen in output screen, String objects are slightly slower in performance due to new object creation.

Important Methods of StringBuffer & StringBuilder classes

StringBuffer and StringBuilder class has one difference that StringBuffer is having all the methods synchronized while StringBuilder is not thread safe so better performance. Here we will discuss methods of StringBuilder which is applicable to StringBuffer as well.

StringBuilder delete (int startIndex, int endIndex)

This method deletes a portion of StringBuilder's character sequence. We need to provide two int argument startIndex and endIndex. For example

```
StringBuilder sb = new StringBuilder("JavaWorld");
sb.delete(4, 8);//sb value would be Javad
```

StringBuilder insert(int offset, String s)

This method used when we want to insert particular char sequence, string or any primitive types at particular index inside StringBuilder character sequence. For example

```
StringBuilder sb = new StringBuilder("ABC");
sb.insert(1, "xyz"); // Here sb value is ABxyzC
```

StringBuilder replace (int start, int end, String s)

This method will replace particular portion of character sequence with third argument (String) mentioned in method call. Syntax example:

```
StringBuilder sb = new StringBuilder("ABCDEF");
```

```
sb.replace(1, 3, "XYZ");
System.out.println(sb); //here sb value is AXYZDEF
```

StringBuilder reverse()

As the name suggest, this method will reverse the sequence of characters in target StringBuilder object. Syntax example

```
StringBuilder sb = new StringBuilder("ABCDEF");
sb.reverse();// here sb value is FEDCBA
```

void setCharAt(int index, char ch)

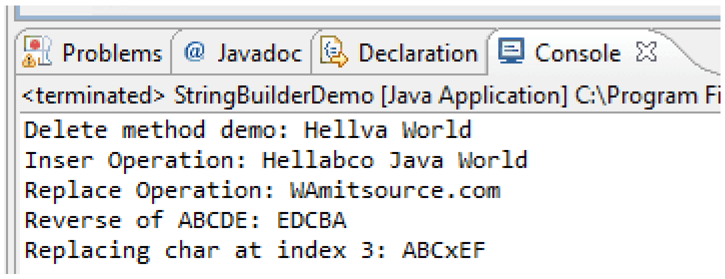
This method should be used when we want to replace one character at particular index with the second argument of the method. Syntax example,

```
StringBuilder sb = new StringBuilder("ABCDEF");
sb.setCharAt(3, 'x'); //This point sb value is ABCxEF
```

Below Java program shows how to use these important methods in java program along with output.

```
public class StringBuilderDemo {

        public static void main(String[] args) {

                StringBuilder sb1 = new StringBuilder("Hello Java World");

                sb1.delete(4, 8);

                System.out.println("Delete method demo: " + sb1);

                StringBuilder sb2 = new StringBuilder("Hello Java World");

                sb2.insert(4, "abc");

                System.out.println("Inser Operation: "+sb2);

                StringBuilder sb3 = new StringBuilder("W3resource.com");

                sb3.replace(1, 4, "Amit");

                System.out.println("Replace Operation: "+sb3);

                StringBuilder sb4 = new StringBuilder("ABCDE");

                System.out.println("Reverse of ABCDE: "+ sb4.reverse());

                StringBuilder sb5 = new StringBuilder("ABCDEF");

                sb5.setCharAt(3, 'x');

                System.out.println("Replacing char at index 3: "+ sb5);


        }
}
```

Output:



```
<terminated> StringBuilderDemo [Java Application] C:\Program Fi
Delete method demo: Hellva World
Inser Operation: Hellabco Java World
Replace Operation: WAmitsource.com
Reverse of ABCDE: EDCBA
Replacing char at index 3: ABCxEF
```

Summary

- String objects are immutable which makes performance of application bit slower
- Java provides StringBuffer and StringBuilder classes for improved string manipulation operation.
- StringBuffer and StringBuilder are similar classes except StringBuffer has all methods synchronized while StringBuilder has non-synchronized methods.

Java.util.ArrayList Class

Introduction

public class ArrayList<E>
extends AbstractList<E>
implements List<E>, RandomAccess, Cloneable, Serializable

ArrayList Class represents a dynamically sized, index-based collection of objects. Implements all optional list operations, and permits all elements, including null. In addition to implementing the List interface, this class provides methods to manipulate the size of the array that is used internally to store the list.

Constructor Summary:

Name	Description
ArrayList()	Constructs an empty list with an initial capacity of ten.
ArrayList(Collection<? extends E> c)	Constructs a list containing the elements of the specified collection, in the order, they are returned by the collection's iterator.
ArrayList(int initialCapacity)</a>	Constructs an empty list with the specified initial capacity.

Method Summary:

Name	Type	Description
trimToSize()	void	Trims the capacity of this ArrayList instance to be the list's current size.
ensureCapacity(int minCapacity)	void	Increases the capacity of this ArrayList instance, if necessary, to ensure that it can hold at least the number of elements specified by the minimum capacity argument.
size()	int	Provides the number of elements in an ArrayList object.
isEmpty()	boolean	Returns true if this list contains no elements.
contains(Object o)	boolean	Determines whether an element exists in an ArrayList object.
indexOf(Object o)	int	Returns the index of the first occurrence of the specified element in this list, or -1 if this list does not contain the element.

lastIndexOf(Object o)	int	Returns the index of the last occurrence of the specified element in this list, or -1 if this list does not contain the element.
clone()	Object	Creates a new instance of an ArrayList object that is a shallow copy of an existing ArrayList object.
toArray()	Object[]	Returns an array containing all of the elements in this list in proper sequence (from first to the last element).
toArray(T[] a)	<T> T[]	Returns an array containing all of the elements in this list in proper sequence (from first to the last element); the runtime type of the returned array is that of the specified array.
get(int index)	E	Returns the element at the specified position in this list.
set(int index, E element)	E	Replaces the element at the specified position in this list with the specified element.
add(E e)	boolean	Adds an element to the end of an ArrayList.
add(int index,E element)	void	Inserts the specified element at the specified position in this list.
remove(int index)	E	Removes the element at the specified position in this list.
remove(Object o)	boolean	Removes the first occurrence of the specified element from this list if it is present.
clear()	void	Removes all items from an ArrayList instance.
addAll(Collection<? extends E> c)	boolean	Appends all the items in an existing collection into an ArrayList object.

addAll(int index, Collection<? extends E> c)	boolean	Inserts all of the elements in the specified collection into this list, starting at the specified position.
removeRange(int fromIndex, int toIndex)	protected void	Removes from this list all of the elements whose index is between fromIndex, inclusive, and toIndex, exclusive.
removeAll(Collection<?> c)	boolean	Removes from this list all of its elements that are contained in the specified collection.
retainAll(Collection<?> c)	boolean	Retains only the elements in this list that are contained in the specified collection.
listIterator(int index)	ListIterator<E>	Returns a list iterator over the elements in this list (in proper sequence), starting at the specified position in the list.
listIterator()	ListIterator<E>	Returns a list iterator over the elements in this list (in proper sequence).
iterator()	Iterator<E>	Returns an iterator over the elements in this list in proper sequence.
subList(int fromIndex, int toIndex)	List<E>	Returns a view of the portion of this list between the specified fromIndex, inclusive, and toIndex, exclusive.
forEach(Consumer<? super E> action)	void	Performs the given action for each element of the Iterable until all elements have been processed or the action throws an exception.
spliterator()	Spliterator<E>	Creates a late-binding and fail-fast Spliterator over the elements in this list.
removeIf(Predicate<? super E> filter)	boolean	Removes all of the elements of this collection that satisfy the given predicate.

replaceAll(UnaryOperator<E> operator)	void	Replaces each element of this list with the result of applying the operator to that element.
sort(Comparator<? super E> c)	void	Sorts this list according to the order induced by the specified Comparator.

Java Collection Framework

Introduction

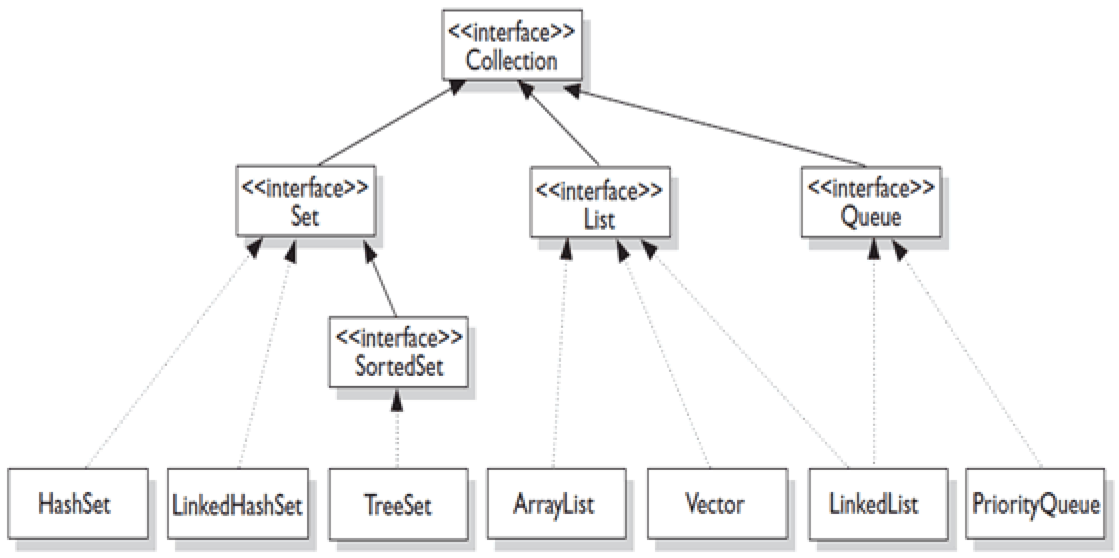
A data structure is a collection of data organized in some fashion. The structure not only stores data but also supports operations for accessing and manipulating the data. The java.util package contains one of Java's most powerful subsystems: The Collections Framework. The Collections Framework is a sophisticated hierarchy of interfaces and classes that provide state-of-the-art technology for managing groups of objects.

You can perform following activity using Java collection framework,

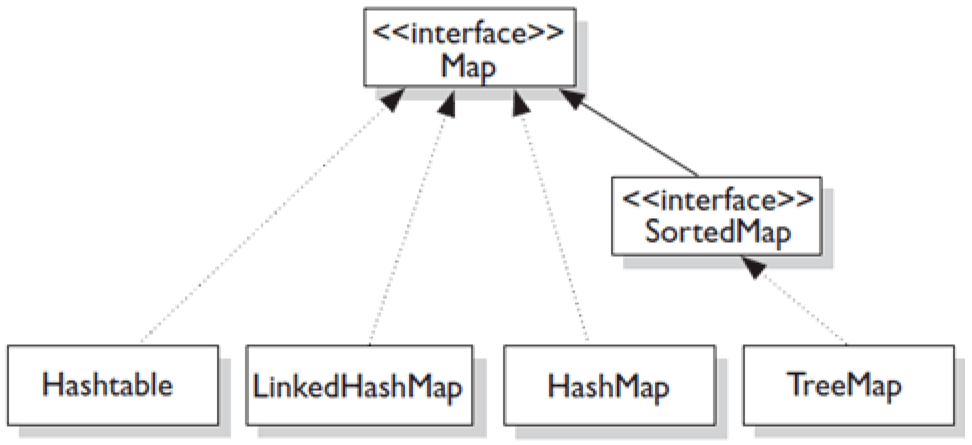
- Add objects to collection
- Remove objects from collection
- Search for an object in collection
- Retrieve/get object from collection
- Iterate through the collection for business specific functionality.

Key Interfaces and classes of collection framework

- **collection** (lowercase c): It represents any of the data structures in which objects are stored and iterated over.
- **Collection** (capital C): It is actually the java.util.Collection interface from which Set, List, and Queue extend.
- **Collections** (capital C and ends with s): It is the java.util.Collections class that holds a pile of static utility methods for use with collections.



There are some other classes in collection framework which do not extend Collection Interface they implement Map interface.



We can say collection has in 4 basic flavors as below,

- **Lists:**
  The List interface extends Collection to define an ordered collection with duplicates allowed. The List interface adds position-oriented operations, as well as a new list iterator that enables the user to traverse the list bi-directionally. ArrayList, LinkedList and vector are classes implementing List interface.

- **Sets:**
  The Set interface extends the Collection interface. It will make sure that an instance of Set contains no duplicate elements. The concrete class implements hashcode and equals methods to make sure uniqueness of objects. Three concrete classes of Set are HashSet, LinkedHashSet and TreeSet.

- **Maps:**
  A map is a container that stores the elements along with the keys. The keys are like indexes. In List, the indexes are integers. In Map, the keys can be any objects. A map cannot contain duplicate keys. Each key maps to one value. A key and its corresponding value from an entry, which is actually stored in a map. HashMap, HashTable,TreeMap and LinkedHashMap are classes implementing Map interface.

- **Queues:**
  A queue is a first-in, first-out data structure. Elements are appended to the end of the queue and are removed from the beginning of the queue. In a priority queue, elements are assigned priorities. When accessing elements, the element with the highest priority is removed first.

We can have sub-flavors of collection classes like sorted, unsorted, ordered and unordered.

Ordered: When a collection is ordered, it means you can iterate through the collection in a specific (not random) order

Sorted: A sorted collection means that the order of objects in the collection is determined according to some rule or rules, known as the sorting order. A sort order has nothing to do with when an object was added to the collection, or when was the last time it was accessed, or what "position" it was added at.

Iterator Interface

Iterator enables you to cycle through a collection, obtaining or removing elements. ListIterator extends Iterator to allow bidirectional traversal of a list, and the modification of elements.

An Iterator is an object that's associated with a specific collection. It let's you loop through the collection step by step. There are two important Iterator methods.

- **boolean hasNext():** Returns true if there is at least one more element in the collection being traversed. Invoking has Next() does NOT move you to the next element of the collection.
- **object next() :** This method returns the next object in the collection, AND moves you forward to the element after the element just returned.

We will see an example of the use of Iterator Interface in LinkedHashSet and HashMap Tutorial.

Comparator Interface

The Comparator interface gives you the capability to sort a given collection any number of different ways. The another handy thing about the Comparator interface is that you can use it to sort instances of any class—even classes you can't modify—unlike the Comparable interface, which forces you to change the class whose instances you want to sort.

The Comparator interface is also very easy to implement, having only one method, compare().The Comparator.compare() method returns an int.

Method Signature

```
int compare(objOne, objTwo)
```

Compare() method returns

- negative if objOne < objTwo
- zero if objOne == objTwo
- positive if objOne > objTwo

We will see Java program using this method in TreeMap/TreeSet Tutorial.

Summary:

Below table summarizes the discussion on collection framework.

	Collection Class name	Ordered	Sorted
Map	Hashtable	No	No
	HashMap	No	No
	TreeMap	Sorted	By natural order or custom order
	LinkedHashMap	By insertion order or last access order	No
Set	HashSet	No	No
	TreeSet	Sorted	By natural order or custom order
	LinkedHashSet	By insertion order	No
List	ArrayList	Indexed	No
	Vector	Indexed	No
	LinkedList	Indexed	No
	Priority queue	Sorted	By to-do order

Java ArrayList and Vector

Introduction

In addition to the Arrays class, Java provides an ArrayList class which can be used to create containers that store lists of objects. ArrayList can be considered as a growable array. It gives you fast iteration and fast random access. ArrayList implements the new RandomAccess interface—a marker interface (meaning it has no methods) that says, "This list supports fast (generally constant time) random access." Choose this over a LinkedList when you need fast iteration but aren't as likely to be doing a lot of insertion and deletion.

Earlier versions of Java have one legacy collection class called Vector which is very much similar to ArrayList. Vector implements a dynamic array. A Vector is basically the same as an ArrayList, but Vector methods are synchronized for thread safety. You'll normally want to use ArrayList instead of Vector because the synchronized methods add a performance hit you might not need. In this tutorial, we will discuss ArrayList only considering all is applicable to Vector as well.

The java.util.ArrayList class is one of the most commonly used of all the classes in the Collections Framework.An ArrayList is dynamically resizable, meaning that its size can change during program execution. This means that:

- You can add an item at any point in an ArrayList container and the array size expands automatically to accommodate the new item.
- You can remove an item at any point in an ArrayList container and the array size contracts automatically.

To state the obvious: Arraylist is an ordered collection (by index), but not sorted.

To use the ArrayList class, you must use the following import statement:

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

Then, to declare an ArrayList, you can use the default constructor, as in the following example:

```
ArrayList names = new ArrayList();
```

The default constructor creates an ArrayList with a capacity of 10 items. The capacity of an ArrayList is the number of items it can hold without having to increase its size. Other constructors of ArrayList as follows,

```
ArrayList names = new ArrayList(int size);
ArrayList names = new ArrayList(Collection c);
```

You can also specify a capacity/size of initial ArrayList as well as create ArrayList from other collection types.

Some of the advantages ArrayList has over arrays are

- It can grow dynamically.
- It provides more powerful insertion and search mechanisms than arrays.

ArrayList methods

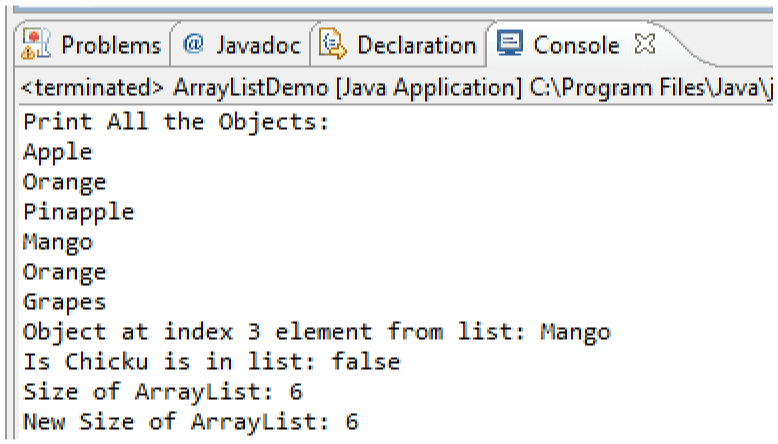
Method	**Purpose**
public void add(Object) public void add(int, Object)	Adds an item to an ArrayList. The default version adds an item at the next available location; an overloaded version allows you to specify a position at which to add the item
public void remove(int)	Removes an item from an ArrayList at a specified location
public void set(int, Object)	Alters an item at a specified ArrayList location
Object get(int)	Retrieves an item from a specified location in an ArrayList
public int size()	Returns the current ArrayList size

Java Program to demonstrate the use of all above methods described above. Here we are creating ArrayList named myList and adding objects using add() method as well as using index based add method, then printing all the objects using for loop. Then there we demonstrate use of get(), contains(), and size() methods. the output of program is shown below the java code.

```
import java.util.ArrayList;

public class ArrayListDemo {

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

```
                //declaring Arraylist of String objects

                ArrayList<String> myList = new ArrayList<String>();

                //Adding objects to Array List at default index

                myList.add("Apple");

                myList.add("Mango");

                myList.add("Orange");

                myList.add("Grapes");

                //Adding object at specific index

                myList.add(1, "Orange");

                myList.add(2,"Pinapple");

                System.out.println("Print All the Objects:");

                for(String s:myList){

                System.out.println(s);

                }

                System.out.println("Object at index 3 element from list: "+ myList.get(3));

                System.out.println("Is Chicku is in list: " + myList.contains("Chicku"));

                System.out.println("Size of ArrayList: " + myList.size());

                myList.remove("Papaya");

                System.out.println("New Size of ArrayList: "+ myList.size());

        }

}
```

Output:



```
<terminated> ArrayListDemo [Java Application] C:\Program Files\Java\j
Print All the Objects:
Apple
Orange
Pinapple
Mango
Orange
Grapes
Object at index 3 element from list: Mango
Is Chicku is in list: false
Size of ArrayList: 6
New Size of ArrayList: 6
```

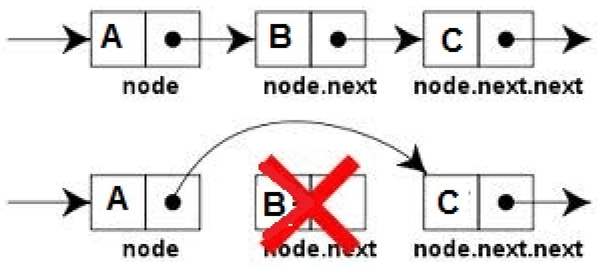
Summary:

- ArrayList and Vector are similar classes only difference is Vector has all method synchronized. Both class simple terms can be considered as a growable array.
- ArrayList should be used in an application when we need to search objects from the list based on the index.
- ArrayList performance degrades when there is lots of insert and update operation in the middle of the list.

Java LinkedList Class

Introduction

A LinkedList is ordered by index position, like ArrayList, except that the elements are doubly-linked to one another. This linkage gives you new methods (beyond what you get from the List interface) for adding and removing from the beginning or end, which makes it an easy choice for implementing a stack or queue. Linked list has a concept of nodes and data. Here Node is storing values of next node while data stores the value it is holding. Below diagram shows how LinkedList storing values. There are three elements in LinkedList A, B and C. We are removing element B from the middle of the LinkedList which will just change node value of element A's node to point to node C.



Keep in mind that a LinkedList may iterate more slowly than an ArrayList, but it's a good choice when you need fast insertion and deletion.

LinkedList has the two constructors shown here:

```
iLinkedList( )
LinkedList(Collection<? extends E> c)
```

The first constructor builds an empty linked list. The second constructor builds a linked list that is initialized with the elements of the collection c.

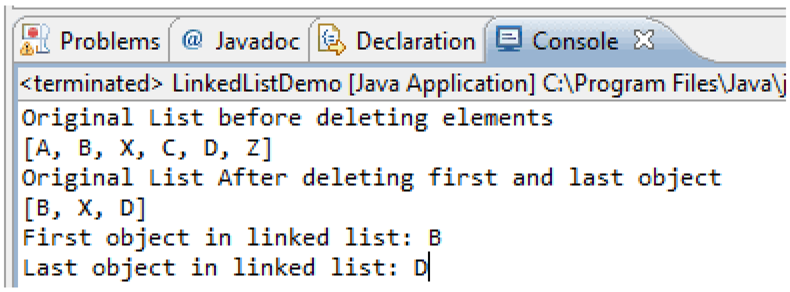
Important methods of LinkedList class:

Method	**Description**
addFirst() or offerFirst( )	To add elements to the start of a list
addLast( ) or offerLast( ) or add()	To add elements to the end of the list
getFirst( ) or peekFirst( )	To obtain the first element of the list
getLast( ) or peekLast( )	To obtain the last element of the list
removeFirst( ) or pollFirst( ) or remove()	To remove the first element of the list
removeLast( ) or pollLast( )	To remove the last element of the list

Java Program to demonstrate use of all above methods described above. Here we are creating LinkedList named myLinkedList and adding objects using add(), addFirst() and addLast() methods as well as using index based add() method, then printing all the objects. Then modifying the list using remove(), removeLast() and remove(Object o) methods. Then we demonstrate use of getFirst() and getLast() methods. Output of program is shown below the java code.

```
import java.util.LinkedList;

public class LinkedListDemo {

        public static void main(String[] args) {

                LinkedList<String> myLinkedList = new LinkedList<String>();

                myLinkedList.addFirst("A");

                myLinkedList.add("B");

                myLinkedList.add("C");

                myLinkedList.add("D");

                myLinkedList.add(2, "X");//This will add C at index 2

                myLinkedList.addLast("Z");

                System.out.println("Original List before deleting elements");

                System.out.println(myLinkedList);

                myLinkedList.remove();

                myLinkedList.removeLast();

                myLinkedList.remove("C");

                System.out.println("Original List After deleting first and last object");

                System.out.println(myLinkedList);

                System.out.println("First object in linked list: "+ myLinkedList.getFirst());

                System.out.println("Last object in linked list: "+ myLinkedList.peekLast());

        }

}
```

Output:



```
<terminated> LinkedListDemo [Java Application] C:\Program Files\Java\j
Original List before deleting elements
[A, B, X, C, D, Z]
Original List After deleting first and last object
[B, X, D]
First object in linked list: B
Last object in linked list: D
```

Summary:

- LinkedList is good for adding elements to the ends, i.e., stacks and queues
- LinkedList performs best while removing objects from middle of the list
- LinkedList implements List, Deque, and Queue interfaces so LinkedList can be used to implement queues and stacks.

Java HashSet

Introduction

A Set is a Collection that cannot contain duplicate elements. It models the mathematical set abstraction. HashSet extends AbstractSet and implements the Set interface. It creates a collection that uses a hash table for storage. A hash table stores information by using a mechanism called hashing. In hashing, the informational content of a key is used to determine a unique value, called its hash code. The hash code is then used as the index at which the data associated with the key is stored. The transformation of the key into its hash code is performed automatically.

When you put an object into a Hashset it uses the object's hashcode value to determine where to put the object in the Set. But it also compares the object's hashcode to the hashcode of all the other objects in the Hash Set, and if there's no matching hashcode,the HashSet assumes that this new object is not a duplicate. HashSet finds a matching hashcode for two objects. one you're inserting and one already in the set-the HashSet will then call one of the object's equals() methods to see if these hashcode-matched objects really are equal. And if they're equal, the HashSet knows that the object you're attempting to add is a duplicate of something in the Set, so the add doesn't happen.

A HashSet is an unsorted, unordered Set. It uses the hashcode of the object being inserted, so the more efficient your hashCode() implementation the better access performance you'll get. Use this class when you want a collection with no duplicates and you don't care about the order, when you iterate through it.

```
HashSet() // Default constructor
HashSet(Collection c) // It creates HashSet from collection c
HashSet( int capacity) // it creates HashSet with initial capacity
mentioned
HashSet(int capacity,  float loadFactor) // This creates HashSet with
capacity and load factor
```

Choosing an initial capacity that's too high can waste both space and time. On the other hand, choosing an initial capacity that's too low wastes time by copying the data structure each time it's forced to increase its capacity. If you don't specify an initial capacity, the default is 16.

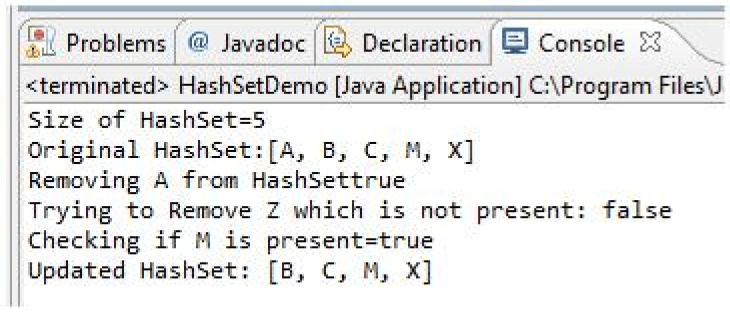
The HashSet class has one other tuning parameter called the load factor. HashSet size grows as per load factor defined or default double size.

HashSet Methods

Method	**Purpose**
public boolean add(Object o)	Adds an object to a HashSet if already not present in HashSet.
public boolean remove(Object o)	Removes an object from a HashSet if found in HashSet.
public boolean contains(Object o)	Returns true if object found else return false
public boolean isEmpty()	Returns true if HashSet is empty else return false
public int size()	Returns number of elements in the HashSet

```
import java.util.HashSet;

public class HashSetDemo {

public static void main(String[] args) {

        HashSet<String> hs = new HashSet<String>();

        // Adding element to HashSet

        hs.add("M");

                        hs.add("B");
                        hs.add("C");

                        hs.add("A");

                        hs.add("M");

                        hs.add("X");

                        System.out.println("Size of HashSet=" + hs.size());

                        System.out.println("Original HashSet:" + hs);

                        System.out.println("Removing A from HashSet: " + hs.remove("A"));

                        System.out.println("Trying to Remove Z which is not present: "

                                        + hs.remove("Z"));

                        System.out.println("Checking if M is present=" + hs.contains("M"));

                        System.out.println("Updated HashSet: " + hs);

        }

}
```

Output:



```
<terminated> HashSetDemo [Java Application] C:\Program Files\J
Size of HashSet=5
Original HashSet:[A, B, C, M, X]
Removing A from HashSettrue
Trying to Remove Z which is not present: false
Checking if M is present=true
Updated HashSet: [B, C, M, X]
```

Summary:

- HashSet provides collection of unique objects
- HashSet is unsorted, unordered and non-indexed based collection class
- HashSet can have only one Null element

Java TreeSet

Introduction

The TreeSet is one of two sorted collections (the other being TreeMap).TreeSet extends AbstractSet and implements the NavigableSet interface. It creates a collection that uses a tree for storage. Objects are stored in sorted, ascending order according to the natural order. Optionally, you can construct a TreeSet with a constructor that lets you give the collection your own rules for what the order should be (rather than relying on the ordering defined by the elements' class) by using a Comparable or Comparator.

Access and retrieval times are quite fast, which makes TreeSet an excellent choice when storing large amounts of sorted information that must be found quickly. TreeSet might not be used when our application has requirement of modification of set in terms of frequent addition of elements.

Constructors

```
TreeSet( ); // Default Constructor
TreeSet(Collection<? extends E> c); //TreeSet from Collection C
TreeSet(Comparator<? super E> comp); // TreeSet with custom ordering as
per Comparator
TreeSet(SortedSet<E>ss); //TreeSet that contains the elements of ss.
```

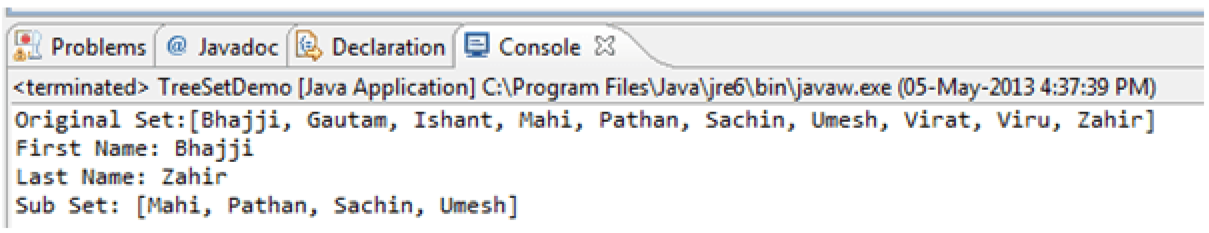
Important Methods of TreeSet Class

Method	**Description**
void add(Object o)	Adds the specified element to this set if it is not already present.
void clear()	Removes all of the elements from this set.
Object first()	Returns the first (lowest) element currently in this sorted set.
Object last()	Returns the last (highest) element currently in this sorted set.
boolean isEmpty()	Returns true if this set contains no elements.
boolean remove(Object o)	Removes the specified element from this set if it is present.

SortedSetsubSet(Object fromElement, Object toElement)	Returns a view of the portion of this set whose elements range from fromElement, inclusive, to toElement, exclusive.

```
import java.util.TreeSet;

public class TreeSetDemo {

        public static void main(String[] args) {

        TreeSet<String> playerSet = new TreeSet<String>();

        playerSet.add("Sachin");

        playerSet.add("Zahir");

        playerSet.add("Mahi");

        playerSet.add("Bhajji");

        playerSet.add("Viru");

        playerSet.add("Gautam");

        playerSet.add("Ishant");

        playerSet.add("Umesh");

        playerSet.add("Pathan");

        playerSet.add("Virat");

        playerSet.add("Sachin"); // This is duplicate element so will not be added again

        //below will print list in alphabetic order

        System.out.println("Original Set:" + playerSet);

        System.out.println("First Name: "+ playerSet.first());

        System.out.println("Last Name: "+ playerSet.last());

        TreeSet<String> newPlySet = (TreeSet<String>) playerSet.subSet("Mahi", "Virat");

        System.out.println("Sub Set: "+ newPlySet);

        }

}
```

Output:



```
<terminated> TreeSetDemo [Java Application] C:\Program Files\Java\jre6\bin\javaw.exe (05-May-2013 4:37:39 PM)
Original Set:[Bhajji, Gautam, Ishant, Mahi, Pathan, Sachin, Umesh, Virat, Viru, Zahir]
First Name: Bhajji
Last Name: Zahir
Sub Set: [Mahi, Pathan, Sachin, Umesh]
```

In above example, we are creating TreeSet of String object. String class is having comparable interface implemented by Java library. Let's think of a case when we need to have our own objects to be stored in Set and ordering of objects as per our rule. Below example shows Cricketers as an object having two properties name and battingPosition. We want to store all Cricketer object as per batting position defined means when we iterate through collection we get names as per batting positions.

Java Code (Cricketer.java)

```
package treeset;

public class Cricketer {

        private String name;

        private int battingPosition;

        Cricketer(String cricketerName, int cBattingPosition){

                this.name = cricketerName;

                this.battingPosition = cBattingPosition;

        }

        public String getName() {

                return name;

        }

        public int getBattingPosition() {

                return battingPosition;

        }

}
```

CompareCricketer.java: Here we are defining rules how to organize Cricketer objects in TreeMap.

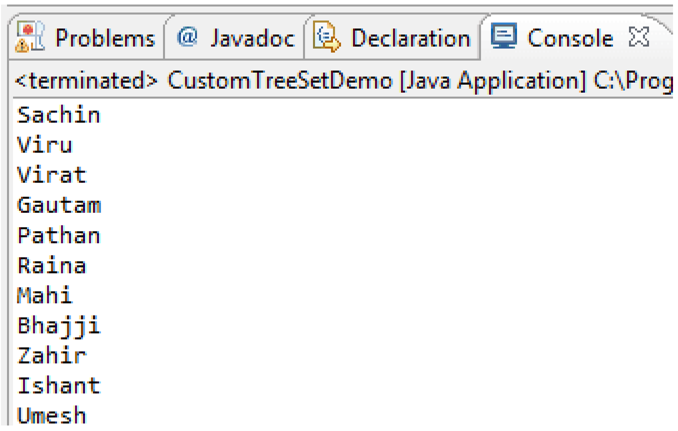
Java Code (CompareCricketer.java)

```java
package treeset;

import java.util.Comparator;

public class CompareCricketer implements Comparator <Cricketer> {

        @Override

        public int compare(Cricketer arg0, Cricketer arg1) {

                if(arg0.getBattingPosition() > arg1.getBattingPosition())

                        return 1;

                else if (arg0.getBattingPosition() < arg1.getBattingPosition())

                        return -1;

                else return 0;

        }

}
```

Java Code:

```java
package treeset;

import java.util.Iterator;

import java.util.TreeSet;

public class CustomTreeSetDemo {

 public static void main(String[] args) {

        TreeSet<Cricketer> playerSet = new TreeSet<Cricketer>(

        new CompareCricketer());

        playerSet.add(new Cricketer("Sachin", 1));

        playerSet.add(new Cricketer("Zahir", 9));

        playerSet.add(new Cricketer("Mahi", 7));

        playerSet.add(new Cricketer("Bhajji", 8));

        playerSet.add(new Cricketer("Viru", 2));

        playerSet.add(new Cricketer("Gautam", 4));

        playerSet.add(new Cricketer("Ishant", 10));

        playerSet.add(new Cricketer("Umesh", 11));

        playerSet.add(new Cricketer("Pathan", 5));
        playerSet.add(new Cricketer("Virat", 3));

        playerSet.add(new Cricketer("Raina", 6));

        Iterator<Cricketer> it = playerSet.iterator();

        while (it.hasNext()) {

        System.out.println(it.next().getName());

    }

  }

}
```

Output:



```
<terminated> CustomTreeSetDemo [Java Application] C:\Prog
Sachin
Viru
Virat
Gautam
Pathan
Raina
Mahi
Bhajji
Zahir
Ishant
Umesh
```

Summary:

- TreeSet is sorted collection class
- Sorting of objects in TreeSet in natural order by default or we can provide comparator class reference for customized sorting.
- While working with large amount of data access and retrieval is faster with TreeSet

Java LinkedHashSet

Introduction

A LinkedHashSet is an ordered version of HashSet that maintains a doubly-linked List across all elements. Use this class instead of HashSet when you care about the iteration order. When you iterate through a HashSet the order is unpredictable, while a LinkedHashSet lets you iterate through the elements in the order in which they were inserted.when cycling through LinkedHashSet using an iterator, the elements will be returned in the order in which they were inserted.

LinkedHashSet constructors

```
LinkedHashSet() // Default constructor
LinkedHashSet(Collection c) // It creates LinkedHashSet from collection c
LinkedHashSet( int capacity) // it creates LinkedHashSet with initial
capacity mentioned
LinkedHashSet(int capacity,  float loadFactor) // This creates
LinkedHashSet with capacity and load factor
```

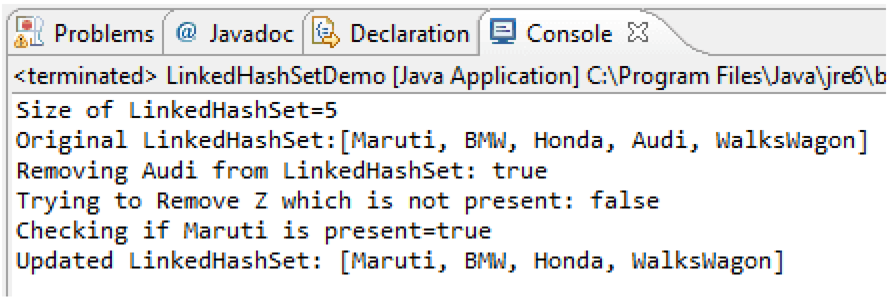
LinkedHashSet Methods

Method	**Purpose**
public boolean add(Object o)	Adds an object to a LinkedHashSet if already not present in HashSet.
public boolean remove(Object o)	Removes an object from LinkedHashSet if found in HashSet.
public boolean contains(Object o)	Returns true if object found else return false
public boolean isEmpty()	Returns true if LinkedHashSet is empty else return false
public int size()	Returns number of elements in the LinkedHashSet

```
import java.util.LinkedHashSet;

public class LinkedHashSetDemo {

        public static void main(String[] args) {

                LinkedHashSet<String> linkedset = new LinkedHashSet<String>();

                // Adding element to LinkedHashSet

                linkedset.add("Maruti");

                linkedset.add("BMW");
```

```
                 linkedset.add("Honda");

                 linkedset.add("Audi");

                 linkedset.add("Maruti"); //This will not add new element as Maruti already exists

                 linkedset.add("WalksWagon");

                 System.out.println("Size of LinkedHashSet=" + linkedset.size());

                 System.out.println("Original LinkedHashSet:" + linkedset);

                 System.out.println("Removing Audi from LinkedHashSet: " +
linkedset.remove("Audi"));

                 System.out.println("Trying to Remove Z which is not present: "

                                  + linkedset.remove("Z"));

                 System.out.println("Checking if Maruti is present=" +
linkedset.contains("Maruti"));

                 System.out.println("Updated LinkedHashSet: " + linkedset);

         }

}
```

Output:



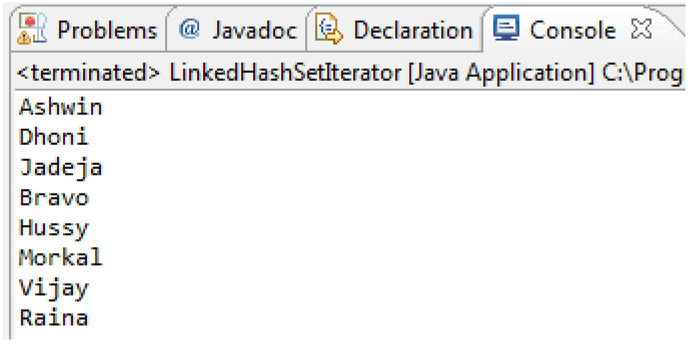
```
<terminated> LinkedHashSetDemo [Java Application] C:\Program Files\Java\jre6\b
Size of LinkedHashSet=5
Original LinkedHashSet:[Maruti, BMW, Honda, Audi, WalksWagon]
Removing Audi from LinkedHashSet: true
Trying to Remove Z which is not present: false
Checking if Maruti is present=true
Updated LinkedHashSet: [Maruti, BMW, Honda, WalksWagon]
```

We can user Iterator object to iterate through our collection. While iterating we can add or remove objects from the collection. Below program demonstrates the use of iterator in LinkedHashSet collection.

Java Code:

```
package linkedHashSet;

import java.util.Iterator;

import java.util.LinkedHashSet;

import java.util.Set;

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

                Set<String> myCricketerSet = new LinkedHashSet<String>();

                myCricketerSet.add("Ashwin");

                myCricketerSet.add("Dhoni");

                myCricketerSet.add("Jadeja");

                myCricketerSet.add("Bravo");

                myCricketerSet.add("Hussy");

                myCricketerSet.add("Morkal");

                myCricketerSet.add("Vijay");

                myCricketerSet.add("Raina");

                Iterator<String> setIterator = myCricketerSet.iterator();

                while(setIterator.hasNext()){

                System.out.println(setIterator.next());

        }

}
```

Output:



```
<terminated> LinkedHashSetIterator [Java Application] C:\Prog
Ashwin
Dhoni
Jadeja
Bravo
Hussy
Morkal
Vijay
Raina
```

Summary:

- LinkedHashSet provides collection of unique objects
- LinkedHashSet is unsorted and non-indexed based collection class
- Iteration in LinkedHashSet is as per insertion order

Java HashMap/Hashtable, LinkedHashMap and TreeMap

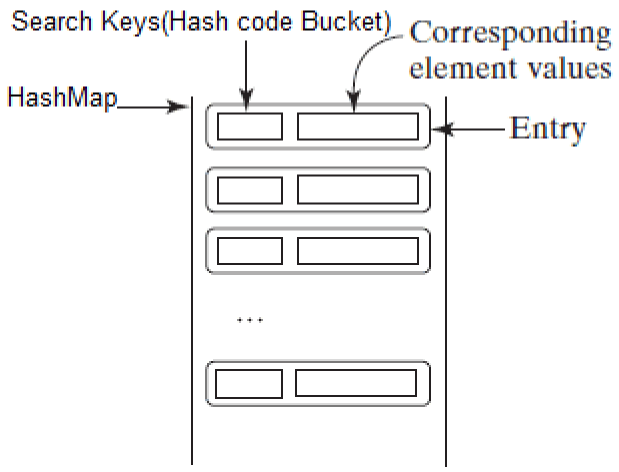
Introduction

The basic idea of a map is that it maintains key-value associations (pairs) so you can look up a value using a key. In this tutorial, we will discuss Java HashMap/Hashtable, LinkedHashMap, and TreeMap.

HashMap/Hashtable

HashMap has implementation based on a hash table. (Use this class instead of Hashtable which is legacy class) .The HashMap gives you an unsorted, unordered Map. When you need a Map and you don't care about the order (when you iterate through it), then HashMap is the right choice. Keys of HashMap is like Set means no duplicates allowed and unordered while values can be any object even null or duplicate is also allowed. HashMap is very much similar to Hashtable only difference is Hashtable has all method synchronized for thread safety while HashMap has non-synchronized methods for better performance.

We can visualize HashMap as below diagram where we have keys as per hash-code and corresponding values.



HashMap provides constant-time performance for inserting and locating pairs. Performance can be adjusted via constructors that allow you to set the capacity and load factor of the hash table.

HashMap Constructors

```
HashMap( )
```

Default HashMap Constructor (with default capacity of 16 and load factor 0.75)

```
HashMap(Map<? extends KeyObject, ? extends ValueObject> m)
```
This is used to create HashMap based on existing map implementation m.

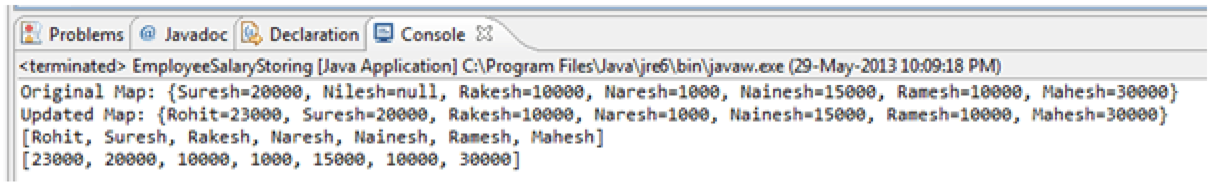
```
HashMap(int capacity)
```
This is used to initialize HashMap with capacity and default load factor.

```
HashMap(int capacity, float loadFactor)
```
This is used to initialize HashMap with capacity and custom load factor.

The basic operations of HashMap (put, get, containsKey, containsValue, size, and is Empty) behave exactly like their counterparts in Hashtable. HashMap has `toString( )` method overridden to print the key-value pairs easily. The following program illustrates HashMap. It maps names to salary. Notice how a set-view is obtained and used.

```
import java.util.*;

public class EmployeeSalaryStoring {

        public static void main(String[] args) {

                //Below Line will create HashMap with initial size 10 and 0.5 load factor

                Map<String, Integer>empSal = new HashMap<String, Integer>(10, 0.5f);

                //Adding employee name and salary to map

                empSal.put("Ramesh", 10000);

                empSal.put("Suresh", 20000);

                empSal.put("Mahesh", 30000);

                empSal.put("Naresh", 1000);

                empSal.put("Nainesh", 15000);

                empSal.put("Rakesh", 10000); // Duplicate Value also allowed but Keys should not
be duplicate

                empSal.put("Nilesh", null); //Value can be null as well

                System.out.println("Original Map: "+ empSal);// Printing full Map

                //Adding new employee the Map to see ordering of object changes

                empSal.put("Rohit", 23000);

                //Removing one key-value pair

                empSal.remove("Nilesh");

                System.out.println("Updated Map: "+empSal);// Printing full Map

                //Printing all Keys

                System.out.println(empSal.keySet());
                //Printing all Values

                System.out.println(empSal.values());

        }

}
```

Output:



Problems | Javadoc | Declaration | Console

```
<terminated> EmployeeSalaryStoring [Java Application] C:\Program Files\Java\jre6\bin\javaw.exe (29-May-2013 10:09:18 PM)
Original Map: {Suresh=20000, Nilesh=null, Rakesh=10000, Naresh=1000, Nainesh=15000, Ramesh=10000, Mahesh=30000}
Updated Map: {Rohit=23000, Suresh=20000, Rakesh=10000, Naresh=1000, Nainesh=15000, Ramesh=10000, Mahesh=30000}
[Rohit, Suresh, Rakesh, Naresh, Nainesh, Ramesh, Mahesh]
[23000, 20000, 10000, 1000, 15000, 10000, 30000]
```

Java LinkedHashMap

LinkedHashMap extends HashMap. It maintains a linked list of the entries in the map, in the order in which they were inserted. This allows insertion-order iteration over the map. That is,when iterating through a collection-view of a LinkedHashMap, the elements will be returned in the order in which they were inserted. Also if one inserts the key again into the LinkedHashMap the original orders are retained. This allows insertion-order iteration over the map. That is, when iterating a LinkedHashMap, the elements will be returned in the order in which they were inserted. You can also create a LinkedHashMap that returns its elements in the order in which they were last accessed.

Constructors

`LinkedHashMap( )`

This constructor constructs an empty insertion-ordered LinkedHashMap instance with the default initial capacity (16) and load factor (0.75).

`LinkedHashMap(int capacity)`

This constructor constructs an empty LinkedHashMap with the specified initial capacity.

`LinkedHashMap(int capacity, float fillRatio)`

This constructor constructs an empty LinkedHashMapwith the specified initial capacity and load factor.

`LinkedHashMap(Map m)`

This constructor constructs an insertion-ordered Linked HashMap with the same mappings as the specified Map.

`LinkedHashMap(int capacity, float fillRatio, boolean Order)`

This constructor constructs an empty LinkedHashMap instance with the specified initial capacity, load factor and ordering mode.

Important methods supported by LinkedHashMap

```
Class clear( )
```
Removes all mappings from the map.

```
containsValue(object value )>
```
Returns true if this map maps one or more keys to the specified value.

```
get(Object key)
```
Returns the value to which the specified key is mapped, or null if this map contains no mapping for the key.

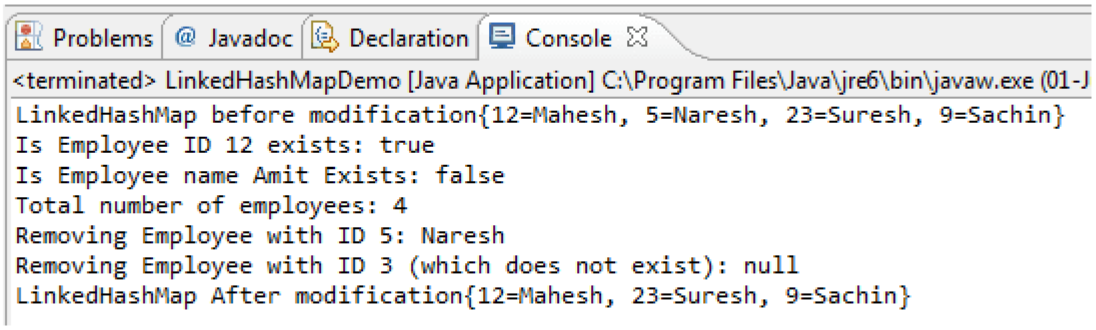
```
removeEldestEntry(Map.Entry eldest)
```
Returns true if this map should remove its eldest entry.

Java Program demonstrate use of LinkedHashMap:

Java Code:

```java
package linkedhashmap;

import java.util.LinkedHashMap;

import java.util.Map;

public class LinkedHashMapDemo {

        public static void main (String args[]){

                //Here Insertion order maintains

                Map<Integer, String>lmap = new LinkedHashMap<Integer, String>();

                lmap.put(12, "Mahesh");

                lmap.put(5, "Naresh");

                lmap.put(23, "Suresh");

                lmap.put(9, "Sachin");

                System.out.println("LinkedHashMap before modification" + lmap);

                System.out.println("Is Employee ID 12 exists: " +lmap.containsKey(12));

                System.out.println("Is Employee name Amit Exists: "+lmap.containsValue("Amit"));

                System.out.println("Total number of employees: "+ lmap.size());

                System.out.println("Removing Employee with ID 5: " + lmap.remove(5));
                System.out.println("Removing Employee with ID 3 (which does not exist): " +
lmap.remove(3));

                System.out.println("LinkedHashMap After modification" + lmap);

        }

}
```

Output:



```
<terminated> LinkedHashMapDemo [Java Application] C:\Program Files\Java\jre6\bin\javaw.exe (01-J
LinkedHashMap before modification{12=Mahesh, 5=Naresh, 23=Suresh, 9=Sachin}
Is Employee ID 12 exists: true
Is Employee name Amit Exists: false
Total number of employees: 4
Removing Employee with ID 5: Naresh
Removing Employee with ID 3 (which does not exist): null
LinkedHashMap After modification{12=Mahesh, 23=Suresh, 9=Sachin}
```

Java TreeMap

A TreeMap is a Map that maintains its entries in ascending order, sorted according to the keys' natural ordering, or according to a Comparator provided at the time of the TreeMap constructor argument.The TreeMap class is efficient for traversing the keys in a sorted order. The keys can be sorted using the Comparable interface or the Comparator interface. SortedMap is a subinterface of Map, which guarantees that the entries in the map are sorted. Additionally, it provides the methods firstKey() and lastKey() for returning the first and last keys in the map, and headMap(toKey) and tailMap(fromKey) for returning a portion of the map whose keys are less than toKey and greater than or equal to fromKey.

TreeMap Constructors

`TreeMap( )`

Default TreeMap Constructor

`TreeMap(Map m)`

This is used to create TreeMap based on existing map implementation m.

`TreeMap(SortedMap m)`

This is used to create TreeMap based on existing map implementation m.

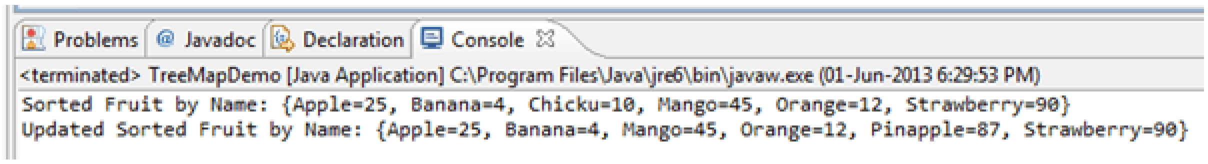
`TreeMap(Comparator ())`

This is used to create TreeMap with ordering based on comparator output.

Java Program which explains some important methods of the tree map.

```
import java.util.Map;

import java.util.TreeMap;

public class TreeMapDemo {

        public static void main(String[] args) {

                //Creating Map of Fruit and price of it

                Map<String, Integer> tMap = new TreeMap<String, Integer>();

                tMap.put("Orange", 12);

                tMap.put("Apple", 25);

                tMap.put("Mango", 45);

                tMap.put("Chicku", 10);

                tMap.put("Banana", 4);

                tMap.put("Strawberry", 90);

                System.out.println("Sorted Fruit by Name: "+tMap);

                tMap.put("Pinapple", 87);

                tMap.remove("Chicku");

                System.out.println("Updated Sorted Fruit by Name: "+tMap);

        }

}
```

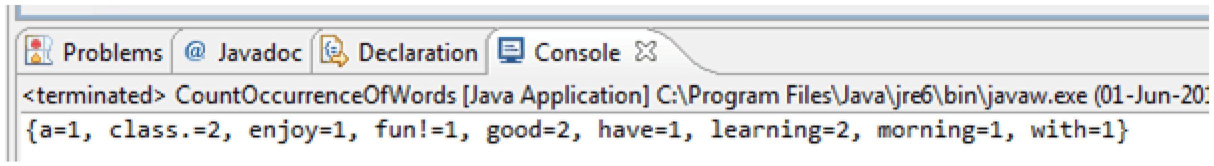
Output:



```
<terminated> TreeMapDemo [Java Application] C:\Program Files\Java\jre6\bin\javaw.exe (01-Jun-2013 6:29:53 PM)
Sorted Fruit by Name: {Apple=25, Banana=4, Chicku=10, Mango=45, Orange=12, Strawberry=90}
Updated Sorted Fruit by Name: {Apple=25, Banana=4, Mango=45, Orange=12, Pinapple=87, Strawberry=90}
```

```
import java.util.*;

public class CountOccurrenceOfWords {

        public static void main(String[] args) {

                // Set text in a string

                String text = "Good morning class. Have a good learning class. Enjoy learning
with fun!";

                // Create a TreeMap to hold words as key and count as value
                TreeMap<String, Integer> map = new TreeMap<String, Integer>();

                String[] words = text.split(" "); //Splitting sentance based on String

                for (int i = 0; i < words.length; i++) {

                        String key = words[i].toLowerCase();

                        if (key.length() > 0) {

                                if (map.get(key) == null) {

                                        map.put(key, 1);

                                } else {

                                        int value = map.get(key).intValue();

                                        value++;

                                        map.put(key, value);

                                }

                        }

                }

                System.out.println(map);

        }

}
```

Output:



```
<terminated> CountOccurrenceOfWords [Java Application] C:\Program Files\Java\jre6\bin\javaw.exe (01-Jun-20
{a=1, class.=2, enjoy=1, fun!=1, good=2, have=1, learning=2, morning=1, with=1}
```

Summary:

- Map is collection of key-value pair (associate) objects collection
- HashMap allows one null key but Hashtable does not allow any null keys.
- Values in HashMap can be null or duplicate but keys have to be unique.
- Iteration order is not constant in the case of HashMap.
- When we need to maintain insertion order while iterating we should use LinkedHashMap.

- LinkedHashMap provides all the methods same as HashMap.
- LinkedHashMap is not threaded safe.
- TreeMap has faster iteration speed compare to other map implementation.
- TreeMap is sorted order collection either natural ordering or custom ordering as per comparator.

Java Collections Utility Class

Introduction

The Collections utility class consists exclusively of static methods that operate on or return collections. It contains polymorphic algorithms that operate on collections, "wrappers", which return a new collection backed by a specified collection,

Some useful method in Collections class:

Method Signature	Description
Collections.sort(List myList)	Sort the myList (implementation of any List interface) provided an argument in natural ordering.
Collections.sort(List, comparator c)	Sort the myList(implementation of any List interface) as per comparator c ordering (c class should implement comparator interface)
Collections.shuffle(List myList)	Puts the elements of myList ((implementation of any List interface)in random order
Collections.reverse(List myList)	Reverses the elements of myList ((implementation of any List interface)

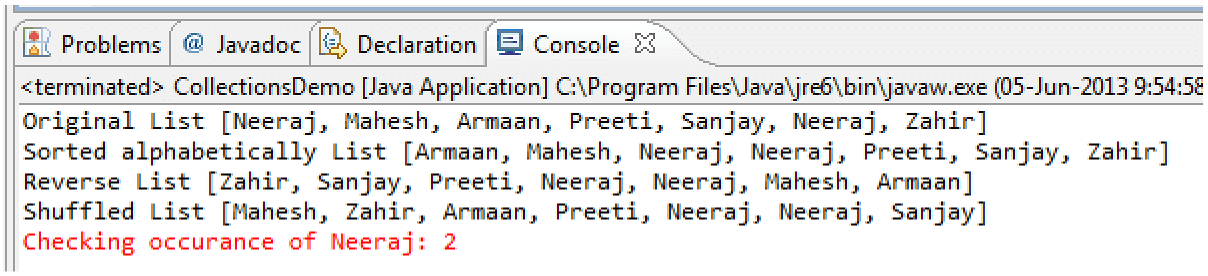
Collections.binarySearch(List mlist, T key)	Searches the mlist (implementation of any List interface) for the specified object using the binary search algorithm.
Collections.copy(List dest, List src)	Copy the source List into dest List.
Collections.frequency(Collection c, Object o)	Returns the number of elements in the specified collection class c (which implements Collection interface can be List, Set or Queue) equal to the specified object
Collections.synchronizedCollection(Collection c)	Returns a synchronized (thread-safe) collection backed by the specified collection.

Let's take the example of List sorting using Collection class. We can sort any Collection using “Collections” utility class. i.e.; ArrayList of Strings can be sorted alphabetically using this utility class. ArrayList class itself is not providing any methods to sort. We use Collections class static methods to do this. Below program shows use of reverse(), shuffle(), frequency() methods as well.

Java Code:

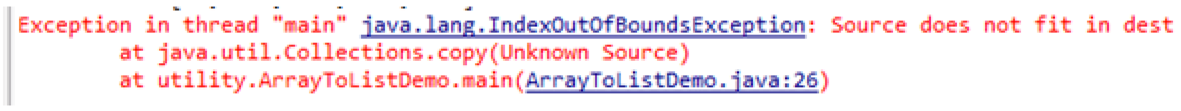
```
package utility;


import java.util.Collections;

import java.util.ArrayList;

import java.util.List;


public class CollectionsDemo {


        public static void main(String[] args) {

                List<String>student<String>List = new ArrayList();

                studentList.add("Neeraj");

                studentList.add("Mahesh");

                studentList.add("Armaan");

                studentList.add("Preeti");

                studentList.add("Sanjay");
                studentList.add("Neeraj");

                studentList.add("Zahir");


                System.out.println("Original List " + studentList);


                Collections.sort(studentList);

                System.out.println("Sorted alphabetically List " + studentList);


                Collections.reverse(studentList);

                System.out.println("Reverse List " + studentList);

                Collections.shuffle(studentList);

                System.out.println("Shuffled List " + studentList);

                System.out.println("Checking occurance of Neeraj: "

                                + Collections.frequency(studentList, "Neeraj"));

        }

}
```

Output:



```
<terminated> CollectionsDemo [Java Application] C:\Program Files\Java\jre6\bin\javaw.exe (05-Jun-2013 9:54:58
Original List [Neeraj, Mahesh, Armaan, Preeti, Sanjay, Neeraj, Zahir]
Sorted alphabetically List [Armaan, Mahesh, Neeraj, Neeraj, Preeti, Sanjay, Zahir]
Reverse List [Zahir, Sanjay, Preeti, Neeraj, Neeraj, Mahesh, Armaan]
Shuffled List [Mahesh, Zahir, Armaan, Preeti, Neeraj, Neeraj, Sanjay]
Checking occurance of Neeraj: 2
```

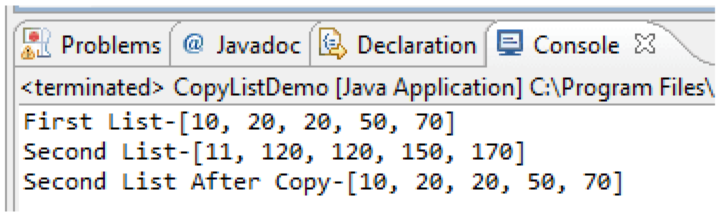
Using Collections class we can copy one type of collection to another type. Collections provide us copy method to copy all the elements from source to destination. Below program demonstrates the use of copy function. Here size of source collection and destination collection should be same else we will get following exception.



```
Exception in thread "main" java.lang.IndexOutOfBoundsException: Source does not fit in dest
	at java.util.Collections.copy(Unknown Source)
	at utility.ArrayToListDemo.main(ArrayToListDemo.java:26)
```

```java
import java.util.Collections;

import java.util.*;


public class CopyListDemo {


        public static void main(String[] args) {

                List <Integer>myFirstList = new ArrayList<Integer>();

                List <Integer> mySecondList = new ArrayList<Integer>();

                myFirstList.add(10);

                myFirstList.add(20);

                myFirstList.add(20);

                myFirstList.add(50);

                myFirstList.add(70);



                mySecondList.add(11);

                mySecondList.add(120);

                mySecondList.add(120);

                mySecondList.add(150);

                mySecondList.add(170);


                System.out.println("First List-"+ myFirstList);

                System.out.println("Second List-"+ mySecondList);

                Collections.copy(mySecondList, myFirstList );

                System.out.println("Second List After Copy-"+ mySecondList);

        }

}
```

Output:



```
<terminated> CopyListDemo [Java Application] C:\Program Files\
First List-[10, 20, 20, 50, 70]
Second List-[11, 120, 120, 150, 170]
Second List After Copy-[10, 20, 20, 50, 70]
```

Java Defining, Instantiating, and Starting Threads

Introduction

One of the most appealing features in Java is the support for easy thread programming. Java provides built-in support for multithreaded programming. A multithreaded program contains two or more parts that can run concurrently. Each part of such a program is called a thread, and each thread defines a separate path of execution. Thus we can say that multithreading is a specialized form of multitasking.

The formal definition of a thread is, A thread is a basic processing unit to which an operating system allocates processor time, and more than one thread can be executing code inside a process. A thread is sometimes called a lightweight process or an execution context

Imagine an online ticket reservation application with a lot of complex capabilities. One of its functions is "search for train/flight tickets from source and destination" another is "check for prices and availability," and a third time-consuming operation is "ticket booking for multiple clients at a time".

In a single-threaded runtime environment, these actions execute one after another. The next action can happen only when the previous one is finished. If a ticket booking takes 10 mins, then other users have to wait for their search operation or book operation. This kind of application will result into waste of time and clients. To avoid this kind of problems java provides multithreading features where multiple operations can take place simultaneously and faster response can be achieved for better user experience. Multithreading enables you to write very efficient programs that make maximum use of the CPU, because idle time can be kept to a minimum.

Defining a Thread

In the most general sense, you create a thread by instantiating an object of type Thread.

Java defines two ways in which this can be accomplished:

- You can implement the Runnable interface.
- You can extend the Thread class

Implementing Runnable

The easiest way to create a thread is to create a class that implements the Runnable interface. Runnable abstracts a unit of executable code. You can construct a thread on any object that implements Runnable. To implement Runnable, a class need only implements a single method called run( ), which is declared like this:

```
public void run( )
```

Inside run( ), you will define the code that constitutes the new thread. It is important to understand that run( ) can call other methods, use other classes, and declare variables, just like the main thread can. The only difference is that run( ) establishes the entry point for another, concurrent thread of execution within your program. This thread will end when run( ) returns.

```
class MyRunnableThread implements Runnable {
public void run() {
System.out.println("Important job running in MyRunnableThread");
}
}
```

Extending java.lang.Thread

The simplest way to define code to run in a separate thread is to

- Extend the java.lang.Thread class.
- Override the run() method.

It looks like this:

```
classMyThread extends Thread {
public void run() {
System.out.println("Important job running in MyThread");
}
}
```

The limitation with this approach (besides being a poor design choice in most cases) is that if you extend Thread, you can't extend anything else. And it's not as if you really need that inherited Thread class behavior because in order to use a thread you'll need to instantiate one anyway.

Instantiating a Thread

Remember, every thread of execution begins as an instance of class Thread. Regardless of whether your run() method is in a Thread subclass or a Runnable implementation class, you still need a Thread object to do the work.

If you have approach two (extending Thread class): Instantiation would be simple

```
MyThread thread = new MyThread();
```

If you implement Runnable, instantiation is only slightly less simple.

To instantiate your Runnable class:

```
MyRunnableThreadmyRunnable = new MyRunnableThread ();
Thread thread = new Thread(myRunnable); // Pass your Runnable to the
Thread
```

Giving the same target to multiple threads means that several threads of execution will be running the very same job (and that the same job will be done multiple times).

Thread Class Constructors

- Thread() :

Default constructor – To create thread with default name and priority

- Thread(Runnable target)

This constructor will createa thread from the runnable object.

- Thread(Runnable target, String name)

This constructor will create thread from runnable object with name as passed in the second argument

- Thread(String name)

This constructor will create a thread with the name as per argument passed.

So now we've made a Thread instance, and it knows which run() method to call. But nothing is happening yet. At this point, all we've got is a plain old Java object of type Thread. It is not yet a thread of execution. To get an actual thread—a new call stack—we still have to start the thread.

Starting a Thread

You've created a Thread object and it knows its target (either the passed-inRunnable or itself if you extended class Thread). Now it's time to get the whole thread thing happening—to launch a new call stack. It's so simple it hardly deserves its own subheading:

```
t.start();
```

Prior to calling start() on a Thread instance, the thread is said to be in the new state. There are various thread states which we will cover in next tutorial.

When we call t.start() method following things happens:

- A new thread of execution starts (with a new call stack).
- The thread moves from the new state to the runnable state.
- When the thread gets a chance to execute, its target run() method will run.

The following example demonstrates what we've covered so far—defining, instantiating, and starting a thread: In below Java program we are not implementing thread communication or synchronization, because of that output may depend on operating system's scheduling mechanism and JDK version.

We are creating two threads t1 and t2 of MyRunnable class object. Starting both threads, each thread is printing thread name in the loop.

Java Code (MyRunnable.java)

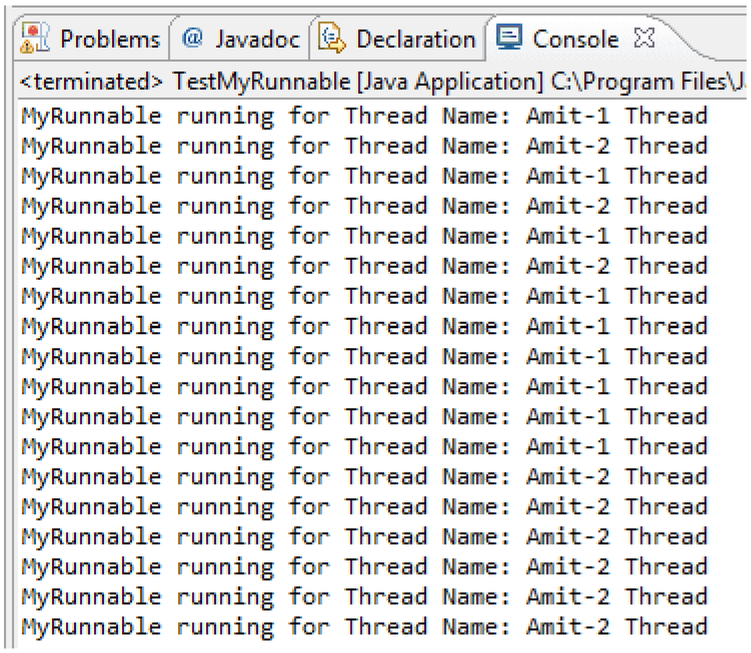
```
package mythreading;

public class MyRunnable implements Runnable{

        @Override

        public void run() {

                for(int x =1; x < 10; x++) {

                        System.out.println("MyRunnable running for Thread Name: " + 
Thread.currentThread().getName());

                        }

        }

}
```

Java Code (TestMyRunnable.java)

```
package mythreading;

public class TestMyRunnable {
```

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

                MyRunnable myrunnable = new MyRunnable();

                //Passing myrunnable object to Thread class constructor

                Thread t1 = new Thread(myrunnable);

                t1.setName("Amit-1 Thread");

                //Starting Thread t1

                t1.start();

                Thread t2 = new Thread(myrunnable);

                t2.setName("Amit-2 Thread");

                t2.start();

                }


}
```

Output:



Problems | Javadoc | Declaration | Console

```
<terminated> TestMyRunnable [Java Application] C:\Program Files\J
MyRunnable running for Thread Name: Amit-1 Thread
MyRunnable running for Thread Name: Amit-2 Thread
MyRunnable running for Thread Name: Amit-1 Thread
MyRunnable running for Thread Name: Amit-2 Thread
MyRunnable running for Thread Name: Amit-1 Thread
MyRunnable running for Thread Name: Amit-2 Thread
MyRunnable running for Thread Name: Amit-1 Thread
MyRunnable running for Thread Name: Amit-1 Thread
MyRunnable running for Thread Name: Amit-1 Thread
MyRunnable running for Thread Name: Amit-1 Thread
MyRunnable running for Thread Name: Amit-1 Thread
MyRunnable running for Thread Name: Amit-1 Thread
MyRunnable running for Thread Name: Amit-2 Thread
MyRunnable running for Thread Name: Amit-2 Thread
MyRunnable running for Thread Name: Amit-2 Thread
MyRunnable running for Thread Name: Amit-2 Thread
MyRunnable running for Thread Name: Amit-2 Thread
MyRunnable running for Thread Name: Amit-2 Thread
```

Summary:

- Threads can be created by extending Thread and overriding the public void run() method
- Thread objects can also be created by calling the Thread constructor that takes a Runnable argument. The Runnable object is said to be the target of the thread.

- You can call start() on a Thread object only once. If start() is called more than once on a Thread object, it will throw a Runtime Exception.
- Each thread has its own call stack which is storing state of thread execution
- When a Thread object is created, it does not become a thread of execution until its start() method is invoked. When a Thread object exists but hasn't been started, it is in the new state and is not considered alive.

Java Thread Methods and Thread States

Introduction

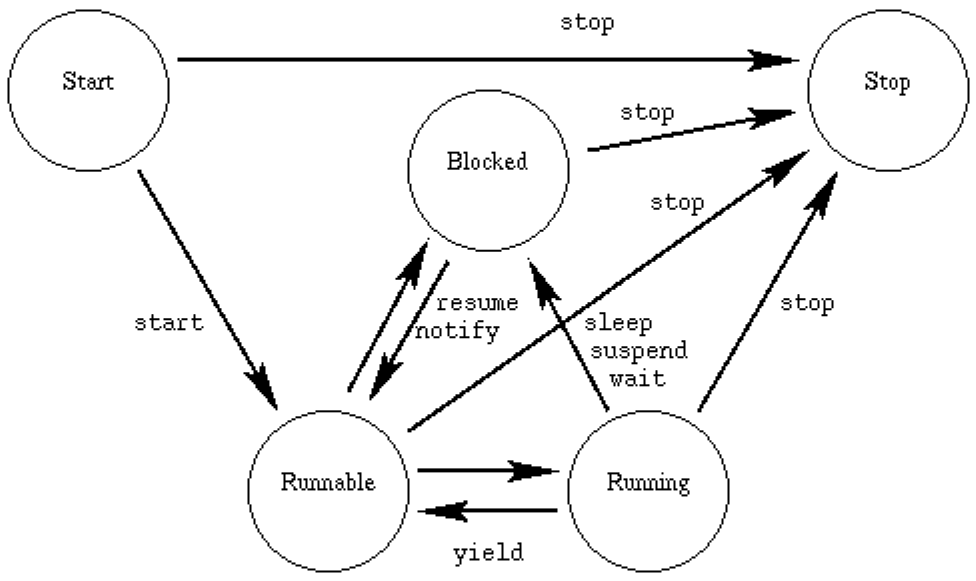
We have various methods which can be called on Thread class object. These methods are very useful when writing a multithreaded application. Thread class has following important methods. We will understand various thread states as well later in this tutorial.

Method Signature	Description
String getName()	Retrieves the name of running thread in the current context in String format
void start()	This method will start a new thread of execution by calling run() method of Thread/runnable object.
void run()	This method is the entry point of the thread. Execution of thread starts from this method.
void sleep(int sleeptime)	This method suspend the thread for mentioned time duration in argument (sleeptime in ms)
void yield()	By invoking this method the current thread pause its execution temporarily and allow other threads to execute.

void join()	This method used to queue up a thread in execution. Once called on thread, current thread will wait till calling thread completes its execution
boolean isAlive()	This method will check if thread is alive or dead

Thread States

The thread scheduler's job is to move threads in and out of the therunning state. While the thread scheduler can move a thread from the running state back to runnable, other factors can cause a thread to move out of running, but not back to runnable. One of these is when the thread's run()method completes, in which case the thread moves from the running state directly to the dead state.



New/Start:

This is the state the thread is in after the Thread instance has been created, but the start() method has not been invoked on the thread. It is a live Thread object, but not yet a thread of execution. At this point, the thread is considered not alive.

Runnable:

This means that a thread can be run when the time-slicing mechanism has CPU cycles available for the thread. Thus, the thread might or might not be running at any moment, but there’s nothing to prevent it from being run if the scheduler can arrange it. That is, it’s not dead or blocked.

Running:

This state is important state where the action is. This is the state a thread is in when the thread scheduler selects it (from the runnable pool) to be the currently executing process. A thread can transition out of a running state for several reasons, including because "the thread scheduler felt like it". There are several ways to get to the runnable state, but only one way to get to the running state: the scheduler chooses a thread from the runnable pool of thread.

Blocked:

The thread can be run, but something prevents it. While a thread is in the blocked state, the scheduler will simply skip it and not give it any CPU time. Until a thread reenters the runnable state, it won’t perform any operations. Blocked state has some sub-states as below,

- Blocked on I/O: The thread waits for completion of blocking operation. A thread can enter this state because of waiting I/O resource. In that case, the thread sends back to runnable state after the availability of resources.
- Blocked for join completion: The thread can come in this state because of waiting for the completion of another thread.
- Blocked for lock acquisition: The thread can come in this state because of waiting for acquire the lock of an object.

Dead:

A thread in the dead or terminated state is no longer schedulable and will not receive any CPU time. Its task is completed, and it is no longer runnable. One way for a task to die is by returning from its run( ) method, but a task’s thread can also be interrupted, as you’ll see shortly.

Let's take an example of Java program to demonstrate various thread state and methods of thread class.

Java Code (AnimalRunnable.java)

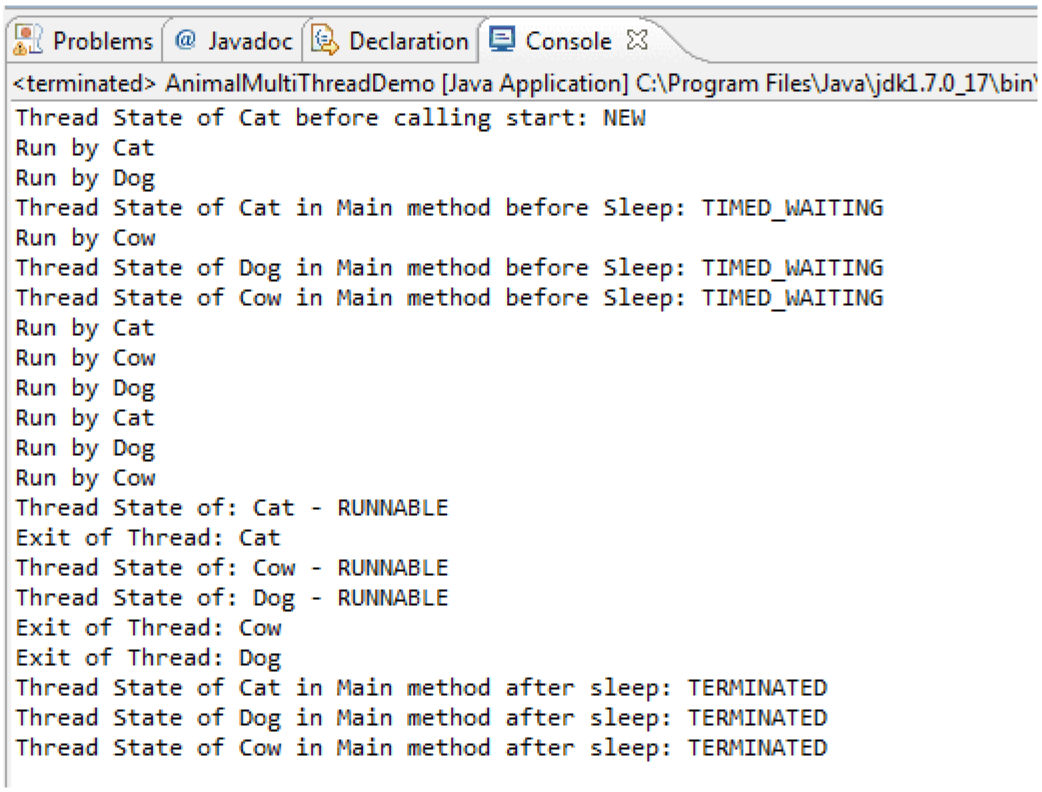
```
package threadstates;

public class AnimalRunnable implements Runnable {

        @Override

        public void run() {

                for (int x = 1; x < 4; x++) {

                        System.out.println("Run by " + Thread.currentThread().getName());

                        try {

                                Thread.sleep(1000);
                        } catch (InterruptedException ex) {

                                ex.printStackTrace();

                        }

                }

                System.out.println("Thread State of: "+ Thread.currentThread().getName()+ " -
"+Thread.currentThread().getState());

                System.out.println("Exit of Thread: "

                                + Thread.currentThread().getName());

        }

}
```

Java Code (AnimalMultiThreadDemo.java):

```
public class AnimalMultiThreadDemo {

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

                // Make object of  Runnable

                AnimalRunnable anr = new AnimalRunnable();

                Thread cat = new Thread(anr);

                cat.setName("Cat");

                Thread dog = new Thread(anr);

                dog.setName("Dog");

                Thread cow = new Thread(anr);

                cow.setName("Cow");

                System.out.println("Thread State of Cat before calling start: "+cat.getState());

                cat.start();

                dog.start();

                cow.start();

                System.out.println("Thread State of Cat in Main method before Sleep: " +
cat.getState());

                System.out.println("Thread State of Dog in Main method before Sleep: " +
dog.getState());

                System.out.println("Thread State of Cow in Main method before Sleep: " +
cow.getState());
```

```
                Thread.sleep(10000);

                System.out.println("Thread State of Cat in Main method after sleep: " +
cat.getState());

                System.out.println("Thread State of Dog in Main method after sleep: " +
dog.getState());

                System.out.println("Thread State of Cow in Main method after sleep: " +
cow.getState());

        }

}


class AnimalRunnable implements Runnable {

        @Override

        public void run() {

                for (int x = 1; x < 4; x++) {

                        System.out.println("Run by " + Thread.currentThread().getName());

                        try {

                                Thread.sleep(1000);

                        } catch (InterruptedException ex) {

                                ex.printStackTrace();

                        }

                }

                System.out.println("Thread State of: "+ Thread.currentThread().getName()+ " -
"+Thread.currentThread().getState());

                System.out.println("Exit of Thread: "

                                + Thread.currentThread().getName());

        }

}
```

Output:



```
<terminated> AnimalMultiThreadDemo [Java Application] C:\Program Files\Java\jdk1.7.0_17\bin\
Thread State of Cat before calling start: NEW
Run by Cat
Run by Dog
Thread State of Cat in Main method before Sleep: TIMED_WAITING
Run by Cow
Thread State of Dog in Main method before Sleep: TIMED_WAITING
Thread State of Cow in Main method before Sleep: TIMED_WAITING
Run by Cat
Run by Cow
Run by Dog
Run by Cat
Run by Dog
Run by Cow
Thread State of: Cat - RUNNABLE
Exit of Thread: Cat
Thread State of: Cow - RUNNABLE
Thread State of: Dog - RUNNABLE
Exit of Thread: Cow
Exit of Thread: Dog
Thread State of Cat in Main method after sleep: TERMINATED
Thread State of Dog in Main method after sleep: TERMINATED
Thread State of Cow in Main method after sleep: TERMINATED
```

Summary:

- Once a new thread is started, it will always enter the runnable state.
- The thread scheduler can move a thread back and forth between the runnable state and the running state.
- For a typical single-processor machine, only one thread can be running at a time, although many threads may be in the runnable state.
- There is no guarantee that the order in which threads were starteddetermines the order in which they'll run.There's no guarantee that threads will take turns in any fair way. It's upto the thread scheduler, as determined by the particular virtual machine implementation.
- A running thread may enter a blocked/waiting state by a wait(), sleep(),or join() call.

Java Thread Interaction

Introduction

The Object class has three methods, wait(), notify(), and notifyAll()that help threads communicate about the status of an event that the threads care about. In last tutorial, we have seen about synchronization on resources where one thread can acquire a lock which forces other threads to wait for lock availability. Object class has these three methods for inter-thread communication.

For example, we have two threads named car_owner and car_meachanic, if thread car_mechanicis busy, so car_owner thread has to wait and keep checking for the car_mechanic thread to get free. Using the wait and notify mechanism, the car_owner for service thread could check for car_mechanic, and if it doesn't find any it can say, “Hey, I’m not going to waste my time checking every few minutes. I’m going to go hang out, and when a car_mechanic gets free, have him notify me so I can go back to runnable and do some work.” In other words, using wait() and notify() lets one thread put itself into a “waiting room” until some another thread notifies it that there’s a reason to come back out.

Method Signature	Description
final void wait( ) throws InterruptedException	Calling this method will male calling thread to give up the monitor and go to sleep until some other thread enters the same monitor and calls notify( ) method.
final void notify( )	This method is similar to notify(), but it wakes up all the threads which are on waiting for state for object lock but only one thread will get access to object lock.

One key point to remember about wait/notify is this:

wait(), notify(), and notifyAll() must be called from within a synchronized context. A thread can't invoke a wait or notify method on an object unless it owns that object's lock.

Below program explains the concept of car service queue where car_owner and car_mechanic thread interact with each other in the loop.

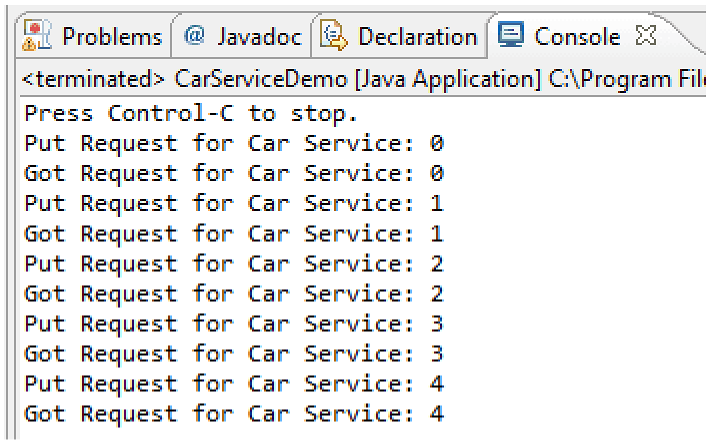
Java Code:

```
package threadcommunication;

public class CarOwner implements Runnable {

        CarQueueClass q;

        CarOwner(CarQueueClass queue){

                this.q=queue;
                new Thread(this, "CarOwner").start();
        }
        @Override
        public void run() {
                int count =0;
                try {
                        while(count< 5){
                                Thread.sleep(2000);
                                q.put(count++);
                        }
                } catch (InterruptedException e) {
                        // TODO Auto-generated catch block
                        e.printStackTrace();
                }
        }
}

package threadcommunication;
public class CarMechanic implements Runnable {
        CarQueueClass q;
        CarMechanic(CarQueueClass queue){
                this.q=queue;
                new Thread(this, "CarMechanic").start();
        }
        @Override
        public void run() {
                for(int i=0;i< 5;i++)
                q.get();
        }
}
```

```
package threadcommunication;

public class CarQueueClass {

                    int n;

                    boolean mechanic_available = false;

                    synchronized int get() {

                       if(!mechanic_available)

                       try {

                          wait(5000);

                       } catch(InterruptedException e) {

                          System.out.println("InterruptedException caught");

                       }

                       System.out.println("Got Request for Car Service: " + n);

                       mechanic_available = false;

                       notify();

                       return n;

                    }

                    synchronized void put(int n) {

                       if(mechanic_available)

                       try {

                          wait(5000);

                       } catch(InterruptedException e) {

                          System.out.println("InterruptedException caught");

                       }

                       this.n = n;

                       mechanic_available = true;

                       System.out.println("Put Request for Car Service: " + n);

                       notify();

                    }

}
```

```
package threadcommunication;

public class CarServiceDemo {

        /**

         * @param args

         */

        public static void main(String[] args) {

                 CarQueueClass q = new CarQueueClass();

              new CarOwner(q);

              new CarMechanic(q);

              System.out.println("Press Control-C to stop.");

        }

}
```

Output:



```
<terminated> CarServiceDemo [Java Application] C:\Program Fil
Press Control-C to stop.
Put Request for Car Service: 0
Got Request for Car Service: 0
Put Request for Car Service: 1
Got Request for Car Service: 1
Put Request for Car Service: 2
Got Request for Car Service: 2
Put Request for Car Service: 3
Got Request for Car Service: 3
Put Request for Car Service: 4
Got Request for Car Service: 4
```

Summary:

- The wait() method puts a thread in waiting for pool from running state.
- The notify() method is used to send a signal to one and only one of the threads that are waiting in that same object's waiting pool.
- The method notifyAll() works in the same way as for notify(), only it sends the signal to all of the threads waiting on the object.

- All three methods—wait(), notify(), and notifyAll()—must be called from within a synchronized context. A thread invokes wait() or notify() on a particular object, and the thread must currently hold the lock on that object.

Java Code Synchronization

Introduction

During the design stage of a multi-threaded application’s development, you should consider the possibility of a so-called race condition, which happens when multiple threads need to modify the same program resource at the same time (concurrently). The classic example is when a husband and wife are trying to withdraw cash from different ATMs at the same time.

In below example, we have a class called Account that represents a bank account. To keep the code short, this account starts with a balance of 50 and can be used only for withdrawals. The withdrawal will be accepted even if there isn't enough money in the account to cover it. The account simply reduces the balance by the amount you want to withdraw:

Java Code:

```
package synchronization;

public class Account {

        private int balance = 50;

        public int getBalance() {

        return balance;

        }

        public void withdraw(int amount) {

        balance = balance - amount;

        }
}
```

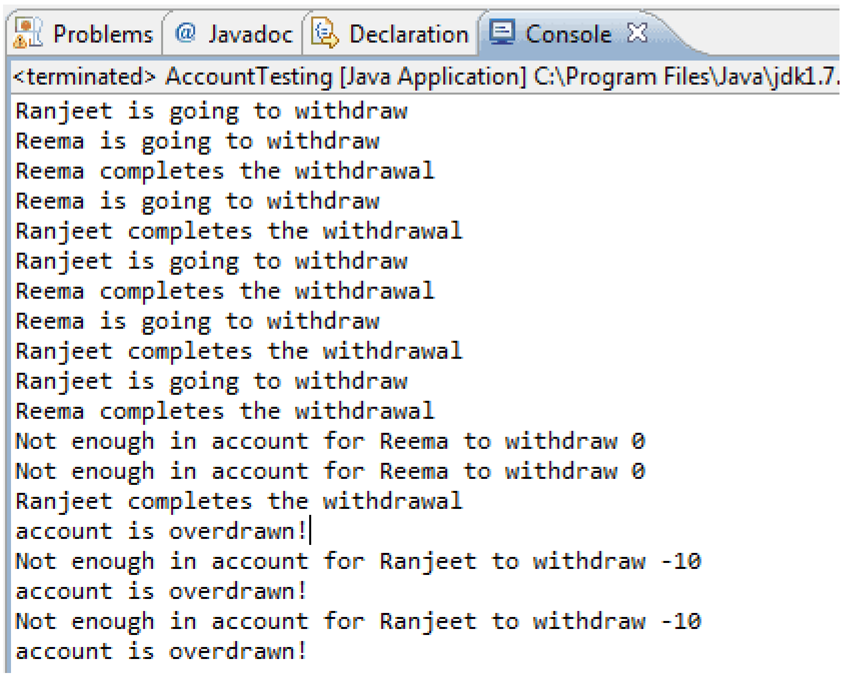
Imagine a couple, Ranjeet and Reema, who both have access to the account and want to make withdrawals. But they don't want the account to ever be overdrawn. Below AccountTesting.java class will start two threads and both thread trying to withdraw money from same account object in the loop. Withdrawal is two steps process :

1. Check the balance.

2. If there's enough in the account (withdraw10), make the withdrawal.

```
public class AccountTesting implements Runnable {

        private Account acct = new Account();

        public static void main(String[] args) {

                AccountTesting r = new AccountTesting();

                Thread one = new Thread(r);

                Thread two = new Thread(r);

                one.setName("Ranjeet");

                two.setName("Reema");

                one.start();

                two.start();

        }

        @Override

        public void run() {

                for (int x = 0; x < 5; x++) {

                        makeWithdrawal(10);

                        if (acct.getBalance() < 0) {

                                System.out.println("account is overdrawn!");

                        }

                }

        }

        private void makeWithdrawal(int amt) {

                if (acct.getBalance() >= amt) {
                        System.out.println(Thread.currentThread().getName() + " is going to
withdraw");

                        try {

                                Thread.sleep(100);

                        } catch (InterruptedException ex) {

                        }

                        acct.withdraw(amt);

                        System.out.println(Thread.currentThread().getName() + " completes the
withdrawal");

                } else {

                        System.out.println("Not enough in account for " +
Thread.currentThread().getName() + " to withdraw " + acct.getBalance());

                }

        }

}


class Account {

        private int balance = 50;

        public int getBalance() {

        return balance;

        }

        public void withdraw(int amount) {

        balance = balance - amount;

        }

}
```

Output:



```
<terminated> AccountTesting [Java Application] C:\Program Files\Java\jdk1.7.
Ranjeet is going to withdraw
Reema is going to withdraw
Reema completes the withdrawal
Reema is going to withdraw
Ranjeet completes the withdrawal
Ranjeet is going to withdraw
Reema completes the withdrawal
Reema is going to withdraw
Ranjeet completes the withdrawal
Ranjeet is going to withdraw
Reema completes the withdrawal
Not enough in account for Reema to withdraw 0
Not enough in account for Reema to withdraw 0
Ranjeet completes the withdrawal
account is overdrawn!
Not enough in account for Ranjeet to withdraw -10
account is overdrawn!
Not enough in account for Ranjeet to withdraw -10
account is overdrawn!
```

Although each time you run this code the output might be a little different, let's walk through this particular example using the numbered lines of output. For the first four attempts, everything is fine.This problem is known as a "race condition," where multiple threads can access the same resource (typically an object's instance variables) and can produce corrupted data if one thread "races in" too quickly before an operation that should be "atomic" has completed.

Synchronization

Synchronization is the solution for this problem. A special keyword, synchronized, prevents race conditions from happening. This keyword places a lock (a monitor) on an important object or piece of code to make sure that only one thread at a time will have access.

How do you protect the data? You must do two things:

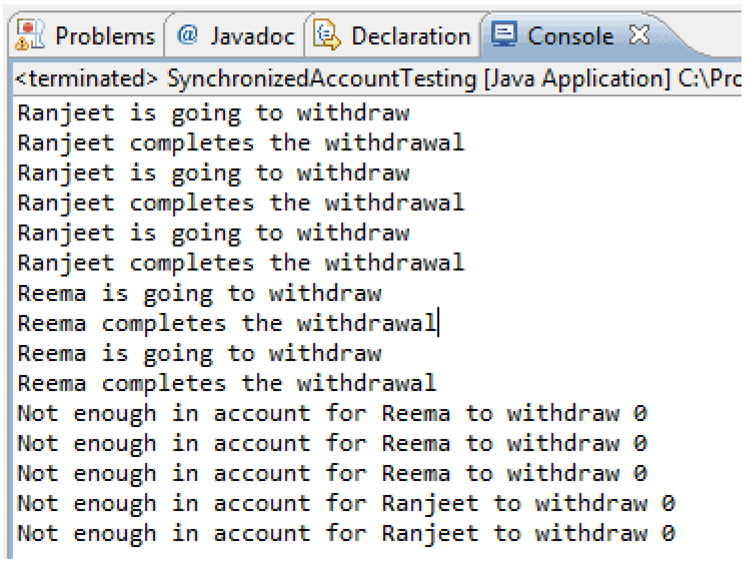
- Mark the variables private.
- Synchronize the code that modifies the variables.

We can solve all of Ranjeet and Reema's problems by adding one word to the code. We mark the makeWithdrawal() method synchronized as follows:

here is the modified Java Code

```
public class SynchronizedAccountTesting implements Runnable {

        private Account acct = new Account();

        public static void main(String[] args) {
                SynchronizedAccountTesting r = new SynchronizedAccountTesting();

                Thread one = new Thread(r);

                Thread two = new Thread(r);

                one.setName("Ranjeet");

                two.setName("Reema");

                one.start();

                two.start();

        }

        @Override

        public void run() {

                for (int x = 0; x < 5; x++) {

                        makeWithdrawal(10);

                        if (acct.getBalance() < 0) {

                                System.out.println("account is overdrawn!");

                        }

                }

        }

        private synchronized void makeWithdrawal(int amt) {

                if (acct.getBalance() >= amt) {

                        System.out.println(Thread.currentThread().getName() + " is going to 
withdraw");

                        try {

                                Thread.sleep(100);

                        } catch (InterruptedException ex) {

                        }

                        acct.withdraw(amt);

                        System.out.println(Thread.currentThread().getName() + " completes the 
withdrawal");

                } else {

                        System.out.println("Not enough in account for " + 
Thread.currentThread().getName() + " to withdraw " + acct.getBalance());

                }
        }
}
class Account {
        private int balance = 50;
        public int getBalance() {
        return balance;
        }
        public void withdraw(int amount) {
        balance = balance - amount;
        }
}
```

Output:



```
<terminated> SynchronizedAccountTesting [Java Application] C:\Prc
Ranjeet is going to withdraw
Ranjeet completes the withdrawal
Ranjeet is going to withdraw
Ranjeet completes the withdrawal
Ranjeet is going to withdraw
Ranjeet completes the withdrawal
Reema is going to withdraw
Reema completes the withdrawal
Reema is going to withdraw
Reema completes the withdrawal
Not enough in account for Reema to withdraw 0
Not enough in account for Reema to withdraw 0
Not enough in account for Reema to withdraw 0
Not enough in account for Ranjeet to withdraw 0
Not enough in account for Ranjeet to withdraw 0
```

Locks the entire method withdrawCash()so no other thread will get access to the specified portion of code until the current (locking) thread has finished its execution of withdrawCash().The locks should be placed for the shortest possible time to avoid slowing down the program: That’s why synchronizing short blocks of code is preferable to synchronizing whole methods.

Every object in Java has a built-in lock that only comes into play when the object has synchronized method code. When we enter a synchronized non-static method, we automatically acquire the lock associated with the current instance of the class whose code we're executing.

Summary:

- The synchronized methods prevent more than one thread from accessing an object's critical method code simultaneously.
- You can use the synchronized keyword as a method modifier, or to start a synchronized block of code.
- To synchronize a block of code (in other words, a scope smaller than the whole method), you must specify an argument that is the object whose lock you want to synchronize on.
- While only one thread can be accessing synchronized code of a particular instance, multiple threads can still access the same object's unsynchronized code.