# Basics of Java

**Overview of Java**

Java is one of the world's most important and widely used computer languages, and it has held this distinction for many years. Unlike some other computer languages whose influence has weared with passage of time, while Java's has grown.

As of 2015, Java is one of the most popular programming languages in use, particularly for client-server web applications, with a reported 9 million developers using and working on it.

**Creation of Java**

Java was developed by James Ghosling, Patrick Naughton, Mike Sheridan at Sun Microsystems Inc. in 1991. It took 18 months to develop the first working version.

The initial name was **Oak** but it was renamed to **Java** in 1995 as OAK was a registered trademark of another Tech company.

**Evolution of Java**

Java was initially launched as Java 1.0 but soon after its initial release, Java 1.1 was launched. Java 1.1 redefined event handling, new library elements were added.

In **Java 1.2** Swing and Collection framework was added and suspend(), resume() and stop() methods were deprecated from **Thread** class.

No major changes were made into **Java 1.3** but the next release that was **Java 1.4** contained several important changes. Keyword assert, chained exceptions and channel based I/O System was introduced.

**Java 1.5** was called **J2SE 5**, it added following major new features :

* Generics
* Annotations
* Autoboxing and autounboxing
* Enumerations
* For-each Loop
* Varargs
* Static Import
* Formatted I/O
* Concurrency utilities

Next major release was **Java SE 7** which included many new changes, like :

* Now **String** can be used to control Switch statement.
* Multi Catch Exception
* *try-with-resource* statement
* Binary Integer Literals
* *Underscore* in numeric literals, etc.

And the latest addition to the lot is, **Java SE 8**, it was released on March 18, 2014. Some of the major new features introduced in JAVA 8 are,

* Lambda Expressions
* New Collection Package java.util.stream to provide Stream API.
* Enhanced Security
* Nashorn Javascript Engine included
* Parallel Array Sorting
* The JDBC-ODBC Bridge has been removed etc.

**Application of Java**

Java is widely used in every corner of world and of human life. Java is not only used in softwares but is also widely used in designing hardware controlling software components. There are more than 930 million JRE downloads each year and 3 billion mobile phones run java.

Following are some other usage of Java :

1. Developing Desktop Applications
2. Web Applications like Linkedin.com, Snapdeal.com etc
3. Mobile Operating System like Android
4. Embedded Systems
5. Robotics and games etc.

### Features of Java

The prime reason behind creation of Java was to bring portability and security feature into a computer language. Beside these two major features, there were many other features that played an important role in moulding out the final form of this outstanding language. Those features are :

#### 1) Simple

Java is easy to learn and its syntax is quite simple, clean and easy to understand.The confusing and ambiguous concepts of C++ are either left out in Java or they have been re-implemented in a cleaner way.

*Eg :* Pointers and Operator Overloading are not there in java but were an important part of C++.

#### 2) Object Oriented

In java everything is Object which has some data and behaviour. Java can be easily extended as it is based on Object Model.

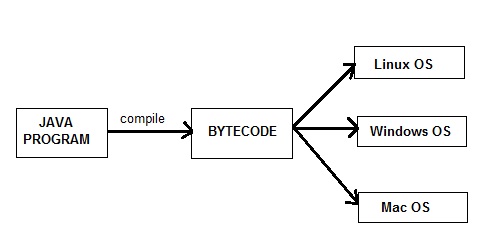
#### 3) Robust

Java makes an effort to eliminate error prone codes by emphasizing mainly on compile time error checking and runtime checking. But the main areas which Java improved were Memory Management and mishandled Exceptions by introducing automatic **Garbage Collector** and **Exception Handling**.

#### 4) Platform Independent

Unlike other programming languages such as C, C++ etc which are compiled into platform specific machines. Java is guaranteed to be write-once, run-anywhere language.

On compilation Java program is compiled into bytecode. This bytecode is platform independent and can be run on any machine, plus this bytecode format also provide security. Any machine with Java Runtime Environment can run Java Programs.



#### 5) Secure

When it comes to security, Java is always the first choice. With java secure features it enable us to develop virus free, temper free system. Java program always runs in Java runtime environment with almost null interaction with system OS, hence it is more secure.

#### 6) Multi Threading

Java multithreading feature makes it possible to write program that can do many tasks simultaneously. Benefit of multithreading is that it utilizes same memory and other resources to execute multiple threads at the same time, like While typing, grammatical errors are checked along.

#### 7) Architectural Neutral

Compiler generates bytecodes, which have nothing to do with a particular computer architecture, hence a Java program is easy to intrepret on any machine.

#### 8) Portable

Java Byte code can be carried to any platform. No implementation dependent features. Everything related to storage is predefined, example: size of primitive data types

#### 9) High Performance

Java is an interpreted language, so it will never be as fast as a compiled language like C or C++. But, Java enables high performance with the use of just-in-time compiler.

### New Features of JAVA 8

Below mentioned are some of the core upgrades done as a part of Java 8 release. Just go through them quickly, we will explore them in details later.

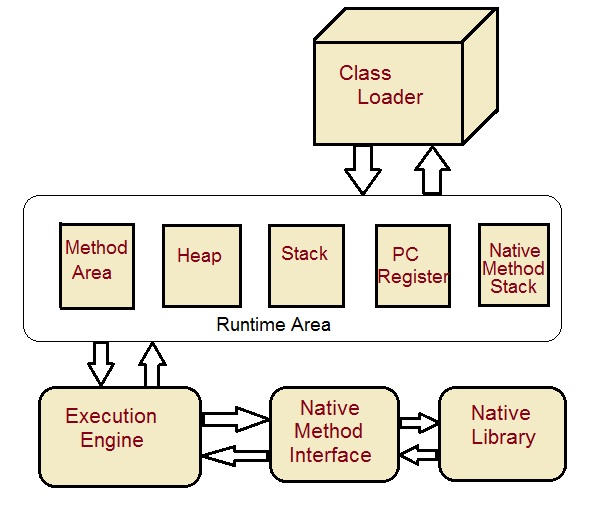
* Enhanced Productivity by providing Optional Classes feature, Lamda Expressions, Streams etc.
* Ease of Use
* Improved Polyglot programming. A **Polyglot** is a program or script, written in a form which is valid in multiple programming languages and it performs the same operations in multiple programming languages. So Java now supports such type of programming technique.
* Improved Security and performance.

### What is JVM?

Java virtual Machine(JVM) is a virtual Machine that provides runtime environment to execute java byte code. The JVM doesn't understand Java typo, that's why you compile your \*.java files to obtain \*.class files that contain the bytecodes understandable by the JVM.

JVM control execution of every Java program. It enables features such as automated exception handling, Garbage-collected heap.

### JVM Architecture



**Class Loader :** Class loader loads the Class for execution.

**Method area :** Stores pre-class structure as constant pool.

**Heap :** Heap is in which objects are allocated.

**Stack :** Local variables and partial results are store here. Each thread has a private JVM stack created when the thread is created.

**Program register :** Program register holds the address of JVM instruction currently being executed.

**Native method stack :** It contains all native used in application.

**Executive Engine :** Execution engine controls the execute of instructions contained in the methods of the classes.

**Native Method Interface :** Native method interface gives an interface between java code and native code during execution.

**Native Method Libraries :** Native Libraries consist of files required for the execution of native code.

#### Difference between JDK and JRE

**JRE** : The Java Runtime Environment (JRE) provides the libraries, the Java Virtual Machine, and other components to run applets and applications written in the Java programming language. JRE does not contain tools and utilities such as compilers or debuggers for developing applets and applications.



**JDK** : The JDK also called Java Development Kit is a superset of the JRE, and contains everything that is in the JRE, plus tools such as the compilers and debuggers necessary for developing applets and applications.



### First Java Program

Let us look at a simple java program.

**class** Hello

{

public static void **main**(String[] *args*)

{

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

}

}

**class** : class keyword is used to declare classes in Java

**public** : It is an access specifier. Public means this function is visible to all.

**static** : static is again a keyword used to make a function static. To execute a static function you do not have to create an Object of the class. The **main()** method here is called by JVM, without creating any object for class.

**void** : It is the return type, meaning this function will not return anything.

**main** : main() method is the most important method in a Java program. This is the method which is executed, hence all the logic must be inside the main() method. If a java class is not having a main() method, it causes compilation error.

**System.out.println** : This is used to print anything on the console like *printf* in C language.

#### Steps to Compile and Run your first Java program

**Step 1:** Open a text editor and write the code as above.

**Step 2:** Save the file as Hello.java

**Step 3:** Open command prompt and go to the directory where you saved your first java program assuming it is saved in C:\

**Step 4:** Type javac Hello.java and press Return to compile your code. This command will call the Java Compiler asking it to compile the specified file. If there are no errors in the code the command prompt will take you to the next line.

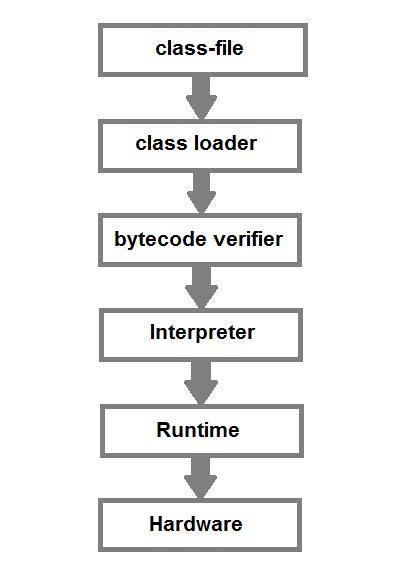
**Step 5:** Now type java Hello on command prompt to run your program.

**Step 6:** You will be able to see **Hello world program** printed on your command prompt.

#### Now let us see What happens at Runtime

After writing your Java program, when you will try to compile it. Compiler will perform some compilation operation on your program.

Once it is compiled successfully byte code(.class file) is generated by the compiler.



After compiling when you will try to run the byte code(.class file), the following steps are performed at runtime:-

1. Class loader loads the java class. It is subsystem of JVM Java Virtual machine.
2. Byte Code verifier checks the code fragments for illegal codes that can violate access right to the object.
3. Interpreter reads the byte code stream and then executes the instructions, step by step.

### Data Types in Java

Java language has a rich implementation of data types. Data types specify size and the type of values that can be stored in an identifier.

In java, data types are classified into two catagories :

1. Primitive Data type
2. Non-Primitive Data type

#### 1) Primitive Data type

A primitive data type can be of eight types :

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| **Primitive Data types** | | | | | | | |
| char | boolean | byte | short | int | long | float | double |

Once a primitive data type has been declared its type can never change, although in most cases its value can change. These eight primitive type can be put into four groups

#### Integer

This group includes byte, short, int, long

**byte :** It is 8 bit integer data type. Value range from -128 to 127. Default value zero. example: byte b=10;

**short :** It is 16 bit integer data type. Value range from -32768 to 32767. Default value zero. example: short s=11;

**int :** It is 32 bit integer data type. Value range from -2147483648 to 2147483647. Default value zero. example: int i=10;

**long :** It is 64 bit integer data type. Value range from -9,223,372,036,854,775,808 to 9,223,372,036,854,775,807. Default value zero. example: long l=100012;

#### Floating-Point Number

This group includes float, double

**float :** It is 32 bit float data type. Default value 0.0f. example: float ff=10.3f;

**double :** It is 64 bit float data type. Default value 0.0d. example: double db=11.123;

#### Characters

This group represent char, which represent symbols in a character set, like letters and numbers.

**char :** It is 16 bit unsigned unicode character. Range 0 to 65,535. example: char c='a';

#### Boolean

This group represent boolean, which is a special type for representing true/false values. They are defined constant of the language. example: boolean b=true;

#### 2) Non-Primitive(Reference) Data type

A reference data type is used to refer to an object. A reference variable is declare to be of specific and that type can never be change. We will talk a lot more about reference data type later in Classes and Object lesson.

### Identifiers in Java

All Java components require names. Name used for classes, methods, interfaces and variables are called **Identifier**. Identifier must follow some rules. Here are the rules:

* All identifiers must start with either a letter( a to z or A to Z ) or currency character($) or an underscore.
* After the first character, an identifier can have any combination of characters.
* A Java **keyword** cannot be used as an identifier.
* Identifiers in Java are case sensitive, foo and Foo are two different identifiers.

### Type Casting

Assigning a value of one type to a variable of another type is known as **Type Casting**.

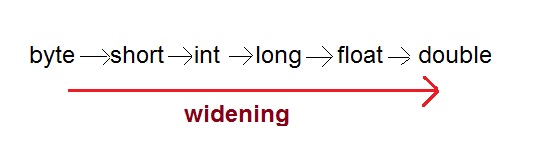
**Example :**

int x = 10;

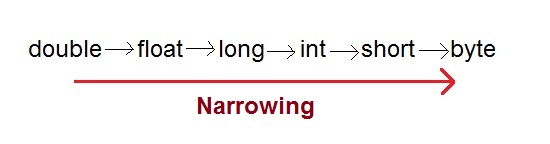
byte y = (byte)x;

In Java, type casting is classified into two types,

* Widening Casting(Implicit)



* Narrowing Casting(Explicitly done)



#### Widening or Automatic type converion

Automatic Type casting take place when,

* the two types are compatible
* the target type is larger than the source type

**Example :**

public class Test

{

public static void main(String[] args)

{

int i = 100;

long l = i; **//no explicit type casting required**

float f = l; **//no explicit type casting required**

System.out.println("Int value "+i);

System.out.println("Long value "+l);

System.out.println("Float value "+f);

}

}

**Output :**

Int value 100

Long value 100

Float value 100.0

#### Narrowing or Explicit type conversion

When you are assigning a larger type value to a variable of smaller type, then you need to perform explicit type casting.

**Example :**

public class Test

{

public static void main(String[] args)

{

double d = 100.04;

long l = (long)d; **//explicit type casting required**

int i = (int)l; **//explicit type casting required**

System.out.println("Double value "+d);

System.out.println("Long value "+l);

System.out.println("Int value "+i);

}

}

**Output :**

Double value 100.04

Long value 100

Int value 100

### Variable

Java Programming language defines mainly three kind of variables.

1. Instance variables
2. Static Variables
3. Local Variables

#### 1) Instance variables

Instance variables are variables that are declare inside a class but outside any method,constructor or block. Instance variable are also variable of object commonly known as field or property.

class Student

{

String name;

int age;

}

Here **name** and **age** are instance variable of Student class.

#### 2) Static variables

Static are class variables declared with static keyword. Static variables are initialized only once. Static variables are also used in declaring constant along with final keyword.

class Student

{

String name;

int age;

static int instituteCode=1101;

}

Here **instituteCode** is a static variable. Each object of Student class will share instituteCode property.

#### 3) Local variables

Local variables are declared in method constructor or blocks. Local variables are initialized when method or constructor block start and will be destroyed once its end. Local variable reside in stack. Access modifiers are not used for local variable.

float getDiscount(int price)

{

float discount;

discount=price\*(20/100);

return discount;

}

Here **discount** is a local variable.

### Concept of Array in Java

An array is a collection of similar data types. Array is a container object that hold values of homogenous type. It is also known as static data structure because size of an array must be specified at the time of its declaration.

An array can be either primitive or reference type. It gets memory in heap area. Index of array starts from zero to size-1.

#### Array Declaration

**Syntax :**

datatype[ ] *identifier*;

or

**datatype** identifier[ ];

Both are valid syntax for array declaration. But the former is more readable.

**Example :**

int[ ] arr;

char[ ] arr;

short[ ] arr;

long[ ] arr;

int[ ][ ] arr; // two dimensional array.

#### Initialization of Array

new operator is used to initialize an array.

**Example :**

int[ ] *arr* = new int[10]; //10 is the size of array.

or

int[ ] *arr* = {10,20,30,40,50};

#### Accessing array element

As mention ealier array index starts from 0. To access nth element of an array. Syntax

arrayname[n-1];

*Example :* To access 4th element of a given array

int[ ] arr = {10,20,30,40};

System.out.println("Element at 4th place" + **arr**[3]);

The above code will print the 4th element of array arr on console.

#### foreach or enhanced for loop

J2SE 5 introduces special type of for loop called foreach loop to access elements of array. Using foreach loop you can access complete array sequentially without using index of array. Let us see an example of foreach loop.

class **Test**

{

public static void main(String[] args)

{

int[] **arr** = {10, 20, 30, 40};

for(int *x* : **arr**)

{

System.out.println(x);

}

}

}

**Output :**

10

20

30

40

### Java Operators

Java provides a rich set of operators enviroment. Java operators can be devided into following categories

* Arithmetic operators
* Relation operators
* Logical operators
* Bitwise operators
* Assignment operators
* Conditional operators
* Misc operators

#### Arithmetic operators

Arithmetic operators are used in mathematical expression in the same way that are used in algebra.

|  |  |
| --- | --- |
| **operator** | **description** |
| + | adds two operands |
| - | subtract second operands from first |
| \* | multiply two operand |
| / | divide numerator by denumerator |
| % | remainder of division |
| ++ | Increment operator increases integer value by one |
| -- | Decrement operator decreases integer value by one |

#### Relation operators

The following table shows all relation operators supported by Java.

|  |  |
| --- | --- |
| **operator** | **description** |
| == | Check if two operand are equal |
| != | Check if two operand are not equal. |
| > | Check if operand on the left is greater than operand on the right |
| < | Check operand on the left is smaller than right operand |
| >= | check left operand is greater than or equal to right operand |
| <= | Check if operand on left is smaller than or equal to right operand |

#### Logical operators

Java supports following 3 logical operator. Suppose a=1 and b=0;

|  |  |  |
| --- | --- | --- |
| **operator** | **description** | **example** |
| && | Logical AND | (a && b) is false |
| || | Logical OR | (a || b) is true |
| ! | Logical NOT | (!a) is false |

#### Bitwise operators

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

|  |  |
| --- | --- |
| **operator** | **description** |
| & | Bitwise AND |
| | | Bitwise OR |
| ^ | Bitwise exclusive OR |
| << | left shift |
| >> | right shift |

Now lets see truth table for bitwise &, | and ^

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **a** | **b** | **a & b** | **a | b** | **a ^ b** |
| 0 | 0 | 0 | 0 | 0 |
| 0 | 1 | 0 | 1 | 1 |
| 1 | 0 | 0 | 1 | 1 |
| 1 | 1 | 1 | 1 | 0 |

The bitwise shift operators shifts the bit value. The left operand specifies the value to be shifted and the right operand specifies the number of positions that the bits in the value are to be shifted. Both operands have the same precedence.**Example**

a = 0001000

b= 2

a << b= 0100000

a >> b= 0000010

#### Assignment Operators

Assignment operator supported by Java are as follows

|  |  |  |
| --- | --- | --- |
| **operator** | **description** | **example** |
| = | assigns values from right side operands to left side operand | a=b |
| += | adds right operand to the left operand and assign the result to left | a+=b is same as a=a+b |
| -= | subtracts right operand from the left operand and assign the result to left operand | a-=b is same as a=a-b |
| \*= | mutiply left operand with the right operand and assign the result to left operand | a\*=b is same as a=a\*b |
| /= | divides left operand with the right operand and assign the result to left operand | a/=b is same as a=a/b |
| %= | calculate modulus using two operands and assign the result to left operand | a%=b is same as a=a%b |

#### Misc operator

There are few other operator supported by java language.

#### Conditional operator

It is also known as ternary operator and used to evaluate Boolean expression

epr1 ? expr2 : expr3

If **epr1**Condition is true? Then value **expr2** : Otherwise value **expr3**

#### instanceOf operator

This operator is used for object reference variables. The operator checks whether the object is of particular type (class type or interface type)

# OOPS Concepts

### Object and Classes

Since Java is an object oriented language, complete java language is build on classes and object. Java is also known as a strong **Object oriented programming language**(oops).

OOPS is a programming approach which provides solution to problems with the help of algorithms based on real world. It uses real world approach to solve a problem. So object oriented technique offers better and easy way to write program then procedural programming model such as C, ALGOL, PASCAL.

#### Main Features of OOPS

* Inheritence
* Polymorphism
* Encapsulation
* Abstraction

As an object oriented language Java supports all the features given above. We will discuss all these features in detail later.

### Class

In Java everything is encapsulated under classes. Class is the core of Java language. Class can be defined as a template/ blueprint that describe the behaviors /states of a particular entity. A class defines new data type. Once defined this new type can be used to create object of that type. Object is an instance of class. You may also call it as physical existence of a logical template class.

A class is declared using **class** keyword. A class contain both data and code that operate on that data. The data or variables defined within a **class** are called **instance variables** and the code that operates on this data is known as **methods**.

#### Rules for Java Class

* A class can have only public or default(no modifier) access specifier.
* It can be either abstract, final or concrete (normal class).
* It must have the class keyword, and class must be followed by a legal identifier.
* It may optionally extend one parent class. By default, it will extend java.lang.Object.
* It may optionally implement any number of comma-separated interfaces.
* The class's variables and methods are declared within a set of curly braces {}.
* Each **.java** source file may contain only one public class. A source file may contain any number of default visible classes.
* Finally, the source file name must match the public class name and it must have a .java suffix.

#### A simple class example

Suppose, Student is a **class** and student's name, roll number, age will be its property. Lets see this in Java syntax

class Student.

{

String name;

int rollno;

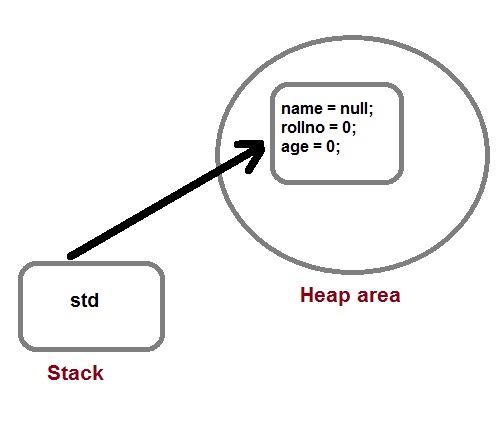
int age;

}

When a reference is made to a particular student with its property then it becomes an **object**, physical existence of Student class.

Student std=new Student();

After the above statement **std** is instance/object of Student class. Here the **new** keyword creates an actual physical copy of the object and assign it to the **std** variable. It will have physical existence and get memory in heap area. **The new operator dynamically allocates memory for an object**



#### Q. How a class is initialized in java?

A Class is initialized in Java when an instance of class is created using either **new** operator or using reflection using class.forName(). A class is also said to be initialized when a static method of **Class** is invoked or a static field of **Class** is assigned.

#### Q. How would you make a copy of an entire Java object with its state?

Make that class implement **Cloneable** interface and call **clone()** method on its object. **clone()** method is defined in **Object** class which is parent of all java class by default.

### Methods in Java

Method describe behavior of an object. A method is a collection of statements that are group together to perform an operation.

**Syntax :**

return-type methodName(parameter-list)

{

//body of method

}

#### Example of a Method

public String getName(String st)

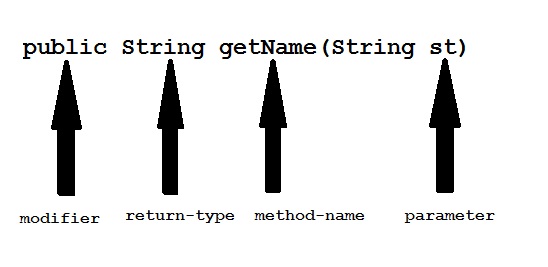
{

String name="StudyTonight";

name=name+st;

return name;

}



**Modifier :** Modifier are access type of method. We will discuss it in detail later.

**Return Type :** A method may return value. Data type of value return by a method is declare in method heading.

**Method name :** Actual name of the method.

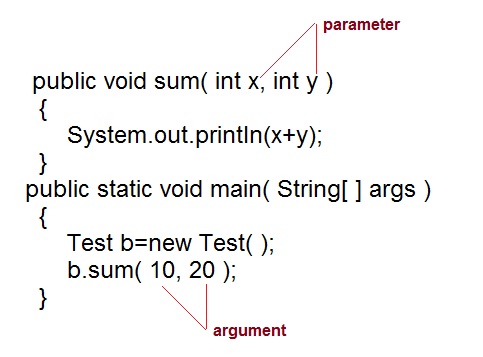
**Parameter :** Value passed to a method.

**Method body :** collection of statement that defines what method does.

#### Parameter Vs. Argument

While talking about method, it is important to know the difference between two terms **parameter** and **argument**.

**Parameter** is variable defined by a method that receives value when the method is called. Parameter are always local to the method they dont have scope outside the method. While **argument** is a value that is passed to a method when it is called.



#### call-by-value and call-by-reference

There are two ways to pass an argument to a method

1. **call-by-value :** In this approach copy of an argument value is pass to a method. Changes made to the argument value inside the method will have no effect on the arguments.
2. **call-by-reference :** In this reference of an argument is pass to a method. Any changes made inside the method will affect the agrument value.

**NOTE :** In Java, when you pass a primitive type to a method it is passed by value whereas when you pass an object of any type to a method it is passed as reference.

#### Example of call-by-value

public class Test

{

public void callByValue(int x)

{

x=100;

}

public static void main(String[] args)

{

int x=50;

Test t = new Test();

t.callByValue(x); //function call

System.out.println(x);

}

}

**Output :**

50

#### Example of call-by-reference

public class Test

{

int x=10;

int y=20;

public void callByReference(Test t)

{

t.x=100;

t.y=50;

}

public static void main(String[] args)

{

Test ts = new Test();

System.out.println("Before "+ts.x+" "+ts.y);

ts.callByReference(ts);

System.out.println("After "+ts.x+" "+ts.y);

}

}

**Output :**

Before 10 20

After 100 50

### Method overloading

If two or more method in a class have same name but different parameters, it is known as method overloading.

Method overloading is one of the ways through which java supports polymorphism. Method overloading can be done by changing number of arguments or by changing the data type of arguments. If two or more method have same name and same parameter list **but differs in return type are not** said to be overloaded method

#### Different ways of Method overloading

There are two different ways of method overloading

#### Method overloading by changing data type of Arguments

*Example :*

class Calculate

{

void sum (int a, int b)

{

System.out.println("sum is"+(a+b)) ;

}

void sum (float a, float b)

{

System.out.println("sum is"+(a+b));

}

Public static void main (String[] args)

{

Calculate cal = new Calculate();

cal.sum (8,5); **//sum(int a, int b) is method is called**.

cal.sum (4.6f, 3.8f); **//sum(float a, float b) is called**.

}

}

**Output :**

Sum is 13

Sum is 8.4

You can see that sum() method is overloaded two times. The first takes two integer arguments, the second takes two float arguments.

#### Method overloading by changing no. of argument.

*Example* :

class Area

{

void find(int l, int b)

{

System.out.println("Area is"+(l\*b)) ;

}

void find(int l, int b,int h)

{

System.out.println("Area is"+(l\*b\*h));

}

public static void main (String[] args)

{

Area ar = new Area();

ar.find(8,5); **//find(int l, int b) is method is called**.

ar.find(4,6,2); **//find(int l, int b,int h) is called**.

}

}

**Output :**

Area is 40

Area is 48

In this example the find() method is overloaded twice. The first takes two arguments to calculate area, and the second takes three arguments to calculate area.

When an overloaded method is called java look for match between the arguments to call the method and the method's parameters. This match need not always be exact, sometime when exact match is not found, Java automatic type conversion plays a vital role.

#### Example of Method overloading with type promotion.

class Area

{

void find(long l,long b)

{

System.out.println("Area is"+(l\*b)) ;

}

void find(int l, int b,int h)

{

System.out.println("Area is"+(l\*b\*h));

}

public static void main (String[] args)

{

Area ar = new Area();

ar.find(8,5); **//automatic type conversion from find(int,int) to find(long,long)** .

ar.find(2,4,6) **//find(int l, int b,int h) is called**.

}

}

**Output :**

Area is 40

Area is 48

### Constructors in Java

A constructor is a special method that is used to initialize an object.Every class has a constructor,if we don't explicitly declare a constructor for any java class the compiler builds a default constructor for that class. A constructor does not have any return type.

A constructor has same name as the class in which it resides. Constructor in Java can not be abstract, static, final or synchronized. These modifiers are not allowed for constructor.

class Car

{

String name ;

String model;

Car( ) **//Constructor**

{

name ="";

model="";

}

}

#### There are two types of Constructor

* Default Constructor
* Parameterized constructor

Each time a new object is created at least one constructor will be invoked.

Car c = new Car() **//Default constructor invoked**

Car c = new Car(name); **//Parameterized constructor invoked**

#### Constructor Overloading

Like methods, a constructor can also be overloaded. Overloaded constructors are differentiated on the basis of their type of parameters or number of parameters. Constructor overloading is not much different than method overloading. In case of method overloading you have multiple methods with same name but different signature, whereas in Constructor overloading you have multiple constructor with different signature but only difference is that Constructor doesn't have return type in Java.

#### Q. Why do we Overload constructors ?

Constuctor overloading is done to construct object in different ways.

#### Example of constructor overloading

class Cricketer

{

String name;

String team;

int age;

Cricketer () **//default constructor**.

{

name ="";

team ="";

age = 0;

}

Cricketer(String n, String t, int a) **//constructor overloaded**

{

name = n;

team = t;

age = a;

}

Cricketer (Cricketer ckt) **//constructor similar to copy constructor of c++**

{

name = ckt.name;

team = ckt.team;

age = ckt.age;

}

public String toString()

{

return "this is " + name + " of "+team;

}

}

Class test:

{

public static void main (String[] args)

{

Cricketer c1 = new Cricketer();

Cricketer c2 = new Cricketer("sachin", "India", 32);

Cricketer c3 = new Cricketer(c2 );

System.out.println(c2);

System.out.println(c3);

c1.name = "Virat";

c1.team= "India";

c1.age = 32;

System .out. print in (c1);

}

}

**output:**

this is sachin of india

this is sachin of india

this is virat of india

#### Q What's the difference between constructors and normal methods?

Constructors must have the same name as the class and can not return a value. They are only called once while regular methods could be called many times and it can return a value or can be void.

#### Q. What is constructor chaining in Java?

Constructor chaining is a phenomena of calling one constructor from another constructor of same class. Since constructor can only be called from another constructor in Java, constructor chaining is used for this purpose.

class Test

{

Test()

{

this(10);

}

Test(int x)

{

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

}

}

#### Q. Does constructors return any value?

Yes, constructors return current instant of a class. But yet constructor signature cannot have any return type.

### this keyword

* **this** keyword is used to refer to current object.
* **this** is always a reference to the object on which method was invoked.
* **this** can be used to invoke current class constructor.
* **this** can be passed as an argument to another method.

*Example* :

class Box

{

Double width, weight, dept;

Box (double w, double h, double d)

{

this.width = w;

this.height = h;

this.depth = d;

}

}

Here the **this** is used to initialize member of current object.

#### The this is used to call overloaded constructor in java

class Car

{

private String name;

public Car()

{

this("BMW"); //oveloaded constructor is called.

}

public Car(String n)

{

this.name=n; //member is initialized using this.

}

}

#### The this is also used to call Method of that class.

public void getName()

{

System.out.println("Studytonight");

}

public void display()

{

this.getName();

System.out.println();

}

#### this is used to return current Object

public Car getCar()

{

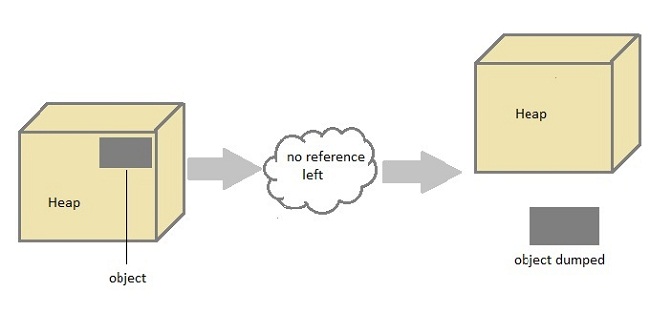
return this;

}

### Garbage Collection

In Java destruction of object from memory is done automatically by the JVM. When there is no reference to an object, then that object is assumed to be no longer needed and the memory occupied by the object are released. This technique is called **Garbage Collection**. This is accomplished by the JVM.

Unlike C++ there is no explicit need to destroy object.



#### Can the Garbage Collection be forced explicitly ?

No, the Garbage Collection can not be forced explicitly. We may request JVM for **garbage collection** by calling **System.gc()** method. But This does not guarantee that JVM will perform the garbage collection.

#### Advantages of Garbage Collection

1. Programmer doesn't need to worry about dereferencing an object.
2. It is done automatically by JVM.
3. Increases memory efficiency and decreases the chances for memory leak.

#### finalize() method

Sometime an object will need to perform some specific task before it is destroyed such as closing an open connection or releasing any resources held. To handle such situation **finalize()** method is used. **finalize()** method is called by garbage collection thread before collecting object. Its the last chance for any object to perform cleanup utility.

Signature of **finalize()** method

protected void finalize()

{

//finalize-code

}

#### Some Important Points to Remember

1. finalize() method is defined in **java.lang.Object** class, therefore it is available to all the classes.
2. finalize() method is declare as **proctected** inside Object class.
3. finalize() method gets called only once by GC threads.

#### gc() Method

**gc()** method is used to call garbage collector explicitly. However **gc()** method does not guarantee that JVM will perform the garbage collection. It only request the JVM for garbage collection. This method is present in **System** and **Runtime** class.

#### Example for gc() method

public class Test

{

public static void main(String[] args)

{

Test t = new Test();

t=null;

System.gc();

}

public void finalize()

{

System.out.println("Garbage Collected");

}

}

**Output :**

Garbage Collected

### Modifiers in Java

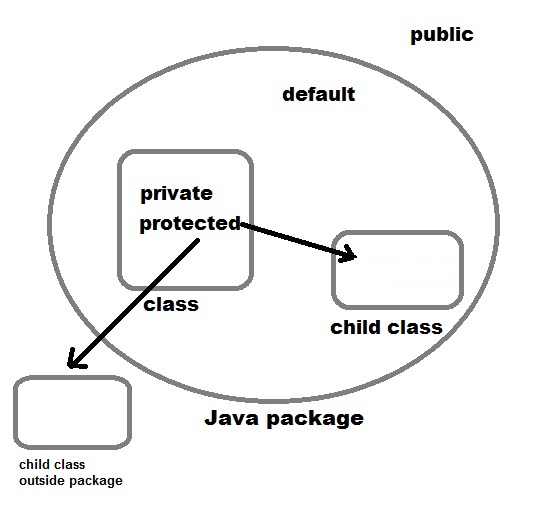
Modifiers are keywords that are added to change meaning of a definition. In Java, modifiers are catagorized into two types,

1. Access control modifier
2. Non Access Modifier

#### 1) Access control modifier

Java language has four access modifier to control access levels for classes, variable methods and constructor.

* **Default :** Default has scope only inside the same package
* **Public :** Public scope is visible everywhere
* **Protected :** Protected has scope within the package and all sub classes
* **Private :** Private has scope only within the classes



#### 2) Non-access Modifier

Non-access modifiers do not change the accessibility of variables and methods, but they do provide them special properties. Non-access modifiers are of 5 types,

1. Final
2. Static
3. Transient
4. Synchronized
5. Volatile

#### Final

Final modifier is used to declare a field as final i.e. it prevents its content from being modified. Final field must be initialized when it is declared.

*Example :*

class Cloth

{

final int MAX\_PRICE = 999; //final variable

final int MIN\_PRICE = 699;

final void display() //final method

{

System.out.println("Maxprice is" + MAX\_PRICE );

System.out.println("Minprice is" + MIN\_PRICE);

}

}

A class can also be declared as final. A class declared as final cannot be inherited. **String** class in java.lang package is a example of final class. Method declared as final can be inherited but you cannot override(redefine) it.

#### Static Modifier

Static Modifiers are used to create class variable and class methods which can be accessed without instance of a class. Lets study how it works with variables and member functions.

#### Static with Variables

Static variables are defined as a class member that can be accessed without any object of that class. Static variable has only one single storage. All the object of the class having static variable will have the same instance of static variable. Static variables are initialized only once.

Static variable are used to represent common property of a class. It saves memory. Suppose there are 100 employee in a company. All employee have its unique name and employee id but company name will be same all 100 employee. Here company name is the common property. So if you create a class to store employee detail, company\_name field will be mark as static.

**Example**

class Employee

{

int e\_id;

String name;

static String company\_name = "StudyTonight";

}

#### Example of static variable

class ST\_Employee

{

int eid;

String name;

static String company\_name ="StudyTonight";

public void show()

{

System.out.println(eid+" "+name+" "+company\_name);

}

public static void main( String[] args )

{

ST\_Employee se1 = new ST\_Employee();

se1.eid = 104;

se1.name = "Abhijit";

se1.show();

ST\_Employee se2 = new ST\_Employee();

se2.eid = 108;

se2.name = "ankit";

se2.show();

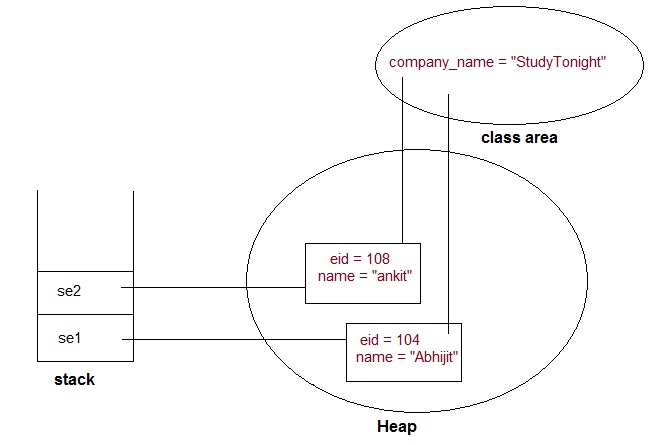
}

}

**Output :**

104 Abhijit StudyTonight

108 ankit StudyTonight



#### Static variable vs Instance Variable

| **Static variable** | **Instance Variable** |
| --- | --- |
| Represent common property | Represent unique property |
| Accessed using class name | Accessed using object |
| get memory only once | get new memory each time a new object is created |

**Example**

public class Test

{

static int x = 100;

int y = 100;

public void increment()

{

x++; y++;

}

public static void main( String[] args )

{

Test t1 = new Test();

Test t2 = new Test();

t1.increment();

t2.increment();

System.out.println(t2.y);

System.out.println(Test.x); **//accessed without any instance of class.**

}

}

**Output :**

101

102

See the difference in value of two variable. Static variable **y** shows the changes made to it by increment() method on the different object. While instance variable **x** show only the change made to it by increment() method on that particular instance.

#### Static Method

A method can also be declared as static. Static methods do not need instance of its class for being accessed. main() method is the most common example of static method. main() method is declared as static because it is called before any object of the class is created.

*Example :*

class Test

{

public static void square(int x)

{

System.out.println(x\*x);

}

public static void main (String[] arg)

{

square(8) **//static method square () is called without any instance of class.**

}

}

**Output:** 64

#### Static block

Static block is used to initialize static data member. Static block executes before main() method.

**Example**

class ST\_Employee

{

int eid;

String name;

static String company\_name;

static {

company\_name ="StudyTonight"; **//static block invoked before main() method**

}

public void show()

{

System.out.println(eid+" "+name+" "+company\_name);

}

public static void main( String[] args )

{

ST\_Employee se1 = new ST\_Employee();

se1.eid = 104;

se1.name = "Abhijit";

se1.show();

}

}

**Output :**

104 Abhijit StudyTonight

#### Q. Why a non-static variable cannot be referenced from a static context ?

When you try to access a non-static variable from a static context like main method, java compiler throws a message like *"a non-static variable cannot be referenced from a static context"*. This is because non-static variables are related with instance of class(object) and they get created when instance of a class is created by using **new** operator. So if you try to access a non-static variable without any instance compiler will complain because those variables are not yet created and they don't have any existence until an instance is created and associated with it.

**Example of accessing non-static variable from a static context**

class Test

{

int x;

public static void main(String[] args)

{

x=10;

}

}

**Output :**

compiler error: non-static variable count cannot be referenced from a static context

**Same example using instance of class**

class Test

{

int x;

public static void main(String[] args)

{

Test tt=new Test();

tt.x=10; //works fine with instance of class

}

}

#### Q. Why main() method is static in java ?

Because static methods can be called without any instance of a class and **main()** is called before any instance of a class is created.

#### Transient modifier

When an instance variable is declared as transient, then its value doesn't persist when an object is serialized

#### Synchronized modifier

When a method is synchronized it can be accessed by only one thread at a time. We will discuss it in detail in Thread.

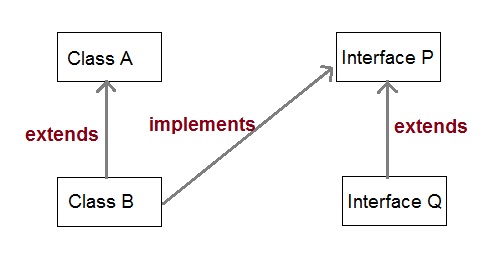
#### Volatile modifier

Volatile modifier tells the compiler that the volatile variable can be changed unexpectedly by other parts of your program. Volatile variables are used in case of multithreading program.

### Inheritance (IS-A)

Inheritance is one of the key features of Object Oriented Programming. Inheritance provided mechanism that allowed **a class to inherit property of another class**. When a Class extends another class it inherits all non-private members including fields and methods. Inheritance in Java can be best understood in terms of Parent and Child relationship, also known as **Super class**(Parent) and **Sub class**(child) in Java language.

Inheritance defines **is-a** relationship between a Super class and its Sub class. extends and implements keywords are used to describe inheritance in Java.



Let us see how **extend** keyword is used to achieve Inheritance.

class **Vehicle**.

{

......

}

class **Car** extends **Vehicle**

{

....... **//extends the property of vehicle class.**

}

Now based on above example. In OOPs term we can say that,

* **Vehicle** is super class of **Car**.
* **Car** is sub class of **Vehicle**.
* Car IS-A Vehicle.

#### Purpose of Inheritance

1. To promote code reuse.
2. To use Polymorphism.

#### Simple example of Inheritance

class **Parent**

{

public void p1()

{

System.out.println("Parent method");

}

}

public class **Child** extends **Parent** {

public void c1()

{

System.out.println("Child method");

}

public static void main(String[] args)

{

Child cobj = new Child();

cobj.c1(); **//method of Child class**

cobj.p1(); **//method of Parent class**

}

}

**Output :**

Child method

Parent method

#### Another example of Inheritance

class **Vehicle**

{

String vehicleType;

}

public class **Car** extends **Vehicle** {

String modelType;

public void showDetail()

{

vehicleType = "Car"; **//accessing Vehicle class member**

modelType = "sports";

System.out.println(modelType+" "+vehicleType);

}

public static void main(String[] args)

{

Car car =new Car();

car.showDetail();

}

}

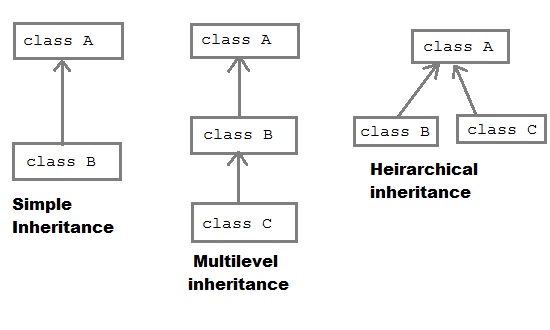
**Output :**

sports Car

#### Types of Inheritance

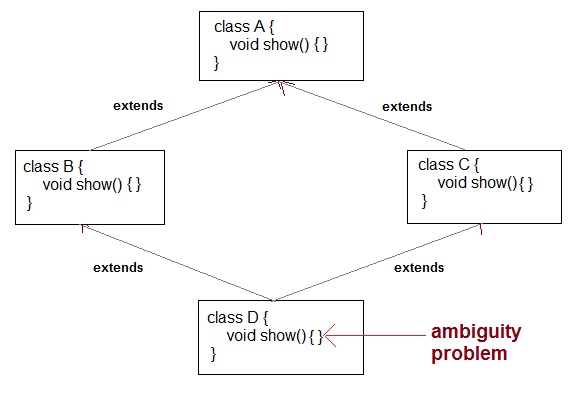
1. Single Inheritance
2. Multilevel Inheritance
3. Heirarchical Inheritance

**NOTE :**Multiple inheritance is not supported in java



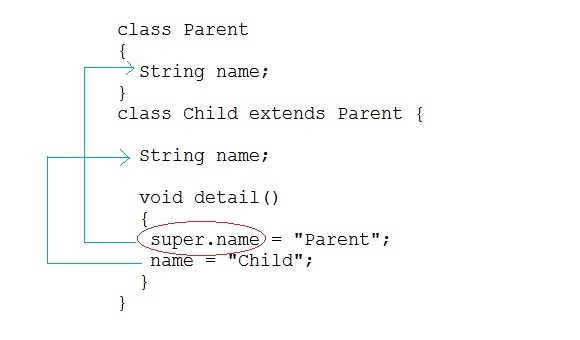
#### Why multiple inheritance is not supported in Java

* To remove ambiguity.
* To provide more maintainable and clear design.



#### super keyword

In Java, super keyword is used to refer to immediate parent class of a class. In other words **super** keyword is used by a subclass whenever it need to refer to its immediate super class.



#### Example of Child class refering Parent class property using super keyword

class **Parent**

{

String name;

}

public class **Child** extends **Parent** {

String name;

public void details()

{

super.name = "Parent"; **//refers to parent class member**

name = "Child";

System.out.println(super.name+" and "+name);

}

public static void main(String[] args)

{

Child cobj = new Child();

cobj.details();

}

}

**Output :**

Parent and Child

#### Example of Child class refering Parent class methods using super keyword

class **Parent**

{

String name;

public void details()

{

name = "Parent";

System.out.println(name);

}

}

public class **Child** extends **Parent** {

String name;

public void details()

{

super.details(); **//calling Parent class details() method**

name = "Child";

System.out.println(name);

}

public static void main(String[] args)

{

Child cobj = new Child();

cobj.details();

}

}

**Output :**

Parent

Child

#### Example of Child class calling Parent class constructor using super keyword

class **Parent**

{

String name;

public Parent(String n)

{

name = n;

}

}

public class **Child** extends **Parent** {

String name;

public Child(String n1, String n2)

{

super(n1); **//passing argument to parent class constructor**

this.name = n2;

}

public void details()

{

System.out.println(super.name+" and "+name);

}

public static void main(String[] args)

{

Child cobj = new Child("Parent","Child");

cobj.details();

}

}

**Output :**

Parent and Child

#### Super class reference pointing to Sub class object.

In context to above example where Class B extends class A.

A a=new B();

is legal syntax because of IS-A relationship is there between class A and Class B.

#### Q. Can you use both this() and super() in a Constructor?

NO, because both super() and this() must be first statement inside a constructor. Hence we cannot use them together.

### Aggregation (HAS-A)

HAS-A relationship is based on usage, rather than inheritance. In other words, class A *has-a* relationship with class B, if code in class A has a reference to an instance of class B.

#### Example

class Student

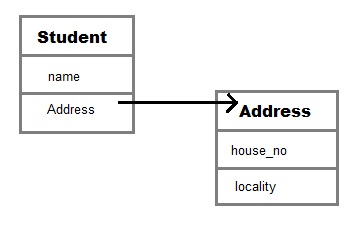
{

String name;

Address ad;

}

Here you can say that **Student** has-a **Address**.



**Student** class has an instance variable of type Address. Student code can use Address reference to invoke methods on the **Address**, and get **Address** behavior.

Aggregation allow you to design classes that follow good Object Oriented practices. It also provide code reusability.

#### Example of Aggregation

class Author

{

String authorName;

int age;

String place;

Author(String name,int age,String place)

{

this.authorName=name;

this.age=age;

this.place=place;

}

public String getAuthorName()

{

return authorName;

}

public int getAge()

{

return age;

}

public String getPlace()

{

return place;

}

}

class Book

{

String name;

int price;

Author auth;

Book(String n,int p,Author at)

{

this.name=n;

this.price=p;

this.auth=at;

}

public void showDetail()

{

System.out.println("Book is"+name);

System.out.println("price "+price);

System.out.println("Author is "+auth.getAuthorName());

}

}

class Test

{

public static void main(String args[])

{

Author ath=new Author("Me",22,"India");

Book b=new Book("Java",550,ath);

b.showDetail();

}

}

**Output :**

Book is Java.

price is 550.

Author is me.

#### Q. What is Composition in java?

Composition is restricted form of Aggregation. For example a class **Car** cannot exist without **Engine**.

class Car

{

private Engine engine;

Car(Engine en)

{

engine = en;

}

}

#### Q. When to use Inheritance and Aggregation?

When you need to use property and behaviour of a class without modifying it inside your class. In such case **Aggregation** is a better option. Whereas when you need to use and modify property and behaviour of a class inside your class, its best to use **Inheritance**.

### Method Overriding

When a method in a sub class has same name and type signature as a method in its super class, then the method is known as overridden method. Method overriding is also referred to as runtime polymorphism. The key benefit of overriding is the abitility to **define method that's specific to a particular subclass type**.

#### Example of Method Overriding

class Animal

{

public void eat()

{

System.out.println("Generic Animal eating");

}

}

class Dog extends Animal

{

public void eat() //eat() method overriden by Dog class.

{

System.out.println("Dog eat meat");

}

}

As you can see here Dog class gives it own implementation of eat() method. Method must have same name and same type signature.

**NOTE :** Static methods cannot be overridden because, a static method is bounded with class where as instance method is bounded with object.

#### Covariant return type

Since Java 5, it is possible to override a method by changing its return type. If subclass override any method by changing the return type of super class method, then the return type of overriden method must be **subtype of return type** declared in original method inside the super class. This is the only way by which method can be overriden by changing its return type.

*Example* :

class Animal

{

Animal myType()

{

return new Animal();

}

}

class Dog extends Animal

{

Dog myType() **//Legal override after Java5 onward**

{

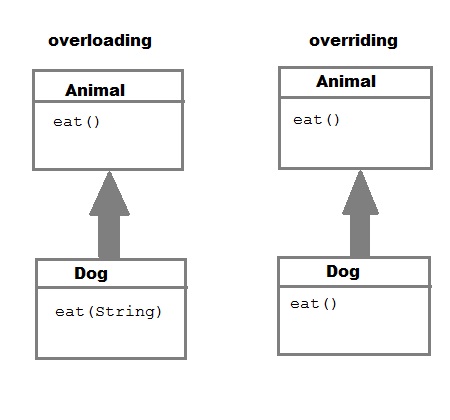
return new Dog();

}

}

#### Difference between Overloading and Overriding

| **Method Overloading** | **Method Overriding** |
| --- | --- |
| Parameter must be different and name must be same. | Both name and parameter must be same. |
| Compile time polymorphism. | Runtime polymorphism. |
| Increase readability of code. | Increase reusability of code. |
| Access specifier can be changed. | Access specifier most not be more restrictive than original method(can be less restrictive). |

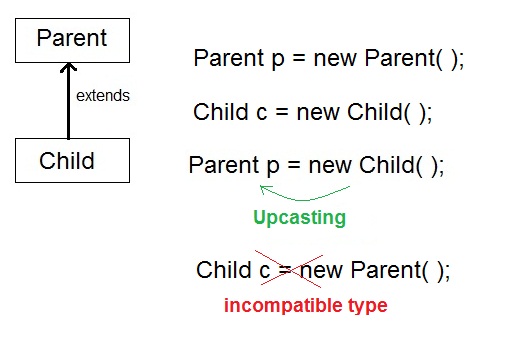


#### Q. Can we Override static method ? Explain with reasons ?

No, we cannot override static method. Because static method is bound to class whereas method overriding is associated with object i.e at runtime.

### Runtime Polymorphism or Dynamic method dispatch

Dynamic method dispatch is a mechanism by which a call to an overridden method is resolved at runtime. This is how java implements runtime polymorphism. When an overridden method is called by a reference, java determines which version of that method to execute based on the type of object it refer to. In simple words the type of object which it referred determines which version of overridden method will be called.



#### Upcasting

When **Parent** class reference variable refers to **Child** class object, it is known as **Upcasting**

#### Example

class **Game**

{

public void type()

{ System.out.println("Indoor & outdoor"); }

}

Class **Cricket** extends **Game**

{

public void type()

{ System.out.println("outdoor game"); }

public static void main(String[] args)

{

Game gm = new Game();

Cricket ck = new Cricket();

gm.type();

ck.type();

gm=ck; **//gm refers to Cricket object**

gm.type(); **//calls Cricket's version of type**

}

}

**Output :**

Indoor & outdoor

Outdoor game

Outdoor game

Notice the last output. This is because of **gm = ck**; Now gm.type() will call Cricket version of type method. Because here gm refers to cricket object.

#### Q. Difference between Static binding and Dynamic binding in java ?

Static binding in Java occurs during compile time while dynamic binding occurs during runtime. Static binding uses type(Class) information for binding while dynamic binding uses instance of class(Object) to resolve calling of method at run-time. Overloaded methods are bonded using static binding while overridden methods are bonded using dynamic binding at runtime.

### instanceof operator

In Java, instanceof operator is used to check the type of an object at runtime. It is the means by which your program can obtain run-time type information about an object. instanceof operator is also important in case of casting object at runtime. instanceof operator return boolean value, if an object reference is of specified type then it return **true** otherwise **false**.

#### Example of instanceOf

public class Test

{

public static void main(String[] args)

{

Test t= new Test();

System.out.println(t **instanceof** Test);

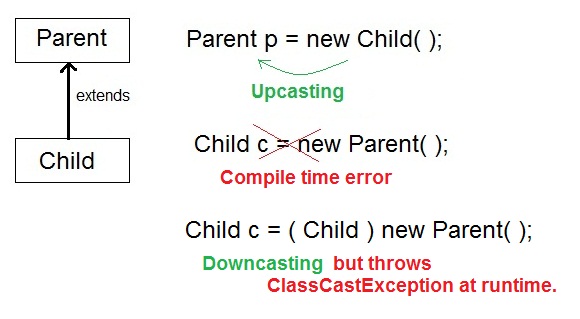
}

}

**Output :**

true

#### Downcasting



#### Example of downcasting with instanceof operator

**class Parent{ }**

public class **Child** extends **Parent**

{

public void check()

{

System.out.println("Sucessfull Casting");

}

public static void show(Parent p)

{

if(p instanceof Child)

{

Child b1=(Child)p;

b1.check();

}

}

public static void main(String[] args)

{

Parent p=new Child();

Child.show(p);

}

}

**Output :**

Sucessfull Casting

#### More example of instanceof operator

class Parent{}

class Child1 extends Parent{}

class Child2 extends Parent{}

class Test

{

public static void main(String[] args)

{

Parent p =new Parent();

Child1 c1 = new Child1();

Child2 c2 = new Child2();

System.out.println(**c1 instanceof Parent**); //true

System.out.println(**c2 instanceof Parent**); //true

System.out.println(**p instanceof Child1**); //false

System.out.println(**p instanceof Child2**); //false

p = c1;

System.out.println(**p instanceof Child1**); //true

System.out.println(**p instanceof Child2**); //false

p = c2;

System.out.println(**p instanceof Child1**); //false

System.out.println(**p instanceof Child2**); //true

}

}

**Output :**

true

true

false

false

true

false

false

true

**Command line argument in Java**

The command line argument is the argument passed to a program at the time when you run it. To access the command-line argument inside a java program is quite easy, they are stored as string in **String** array passed to the args parameter of main() method.

*Example*

class cmd

{

public static void main(String[] args)

{

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

{

System.out.println(args[i]);

}

}

}

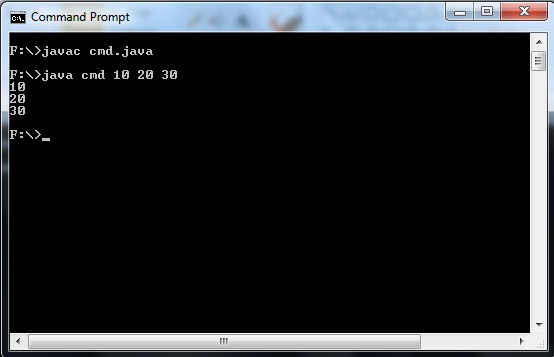
**Execute this program as java cmd 10 20 30**

**Output :**

10

20

30

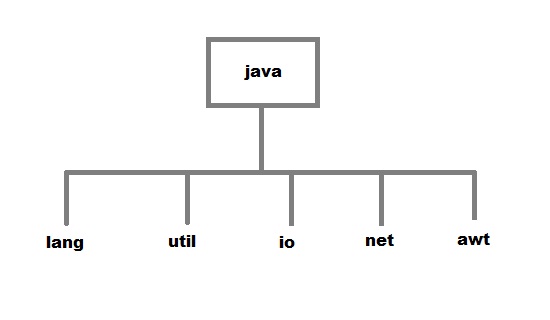


### Java Package

Package are used in Java, in-order to avoid name conflicts and to control access of class, interface and enumeration etc. A package can be defined as a group of similar types of classes, interface, enumeration and sub-package. Using package it becomes easier to locate the related classes.

#### Package are categorized into two forms

* Built-in Package:-Existing Java package for example java.lang, java.util etc.
* User-defined-package:- Java package created by user to categorized classes and interface



#### Creating a package

Creating a package in java is quite easy. Simply include a package command followed by name of the package as the first statement in java source file.

package mypack;

public class employee

{

...statement;

}

The above statement create a package called **mypack**.

Java uses file system directory to store package. For example the .class for any classes you to define to be part of **mypack** package must be stored in a directory called mypack

#### Example of package creation

package mypack

class Book

{

String bookname;

String author;

Book(String b, String c)

{

this.bookname = b;

this.author = c;

}

public void show()

{

System.out.println(bookname+" "+ author);

}

}

class test

{

public static void main(String[] args)

{

Book bk = new Book("java","Herbert");

bk.show();

}

}

##### To run this program :

* create a directory under your current working development directory(i.e. JDK directory), name it as **mypack**.
* compile the source file
* Put the class file into the directory you have created.
* Execute the program from development directory.

**NOTE :** Development directory is the directory where your JDK is install.

#### Uses of java package

Package is a way to organize files in java, it is used when a project consists of multiple modules. It also helps resolve naming conflicts. Package's access level also allows you to protect data from being used by the non-authorized classes.

#### import keyword

**import** keyword is used to import built-in and user-defined packages into your java source file. So that your class can refer to a class that is in another package by directly using its name.

There are 3 different ways to refer to class that is present in different package

1. **Using fully qualified name** (But this is not a good practice.)

*Example :*

class MyDate extends java.util.Date

{

//statement;

}

1. **import the only class you want to use.**

*Example :*

import java.util.Date;

class MyDate extends Date

{

//statement.

}

1. **import all the classes from the particular package**

*Example :*

import java.util.\*;

class MyDate extends Date

{

//statement;

}

#### import statement is kept after the package statement.

*Example :*

package mypack;

import java.util.\*;

But if you are not creating any package then import statement will be the first statement of your java source file.

#### Static import

**static import** is a feature that expands the capabilities of **import** keyword. It is used to import **static** member of a class. We all know that static member are referred in association with its class name outside the class. Using **static import**, it is possible to refer to the static member directly without its class name. There are two general form of static import statement.

* The first form of **static import** statement, import only a single static member of a class

**Syntax**

**import static *package.class-name.static-member-name;***

**Example**

import static java.lang.Math.sqrt; //importing static method **sqrt** of **Math** class

* The second form of **static import** statement,imports all the static member of a class

**Syntax**

**import static *package.class-type-name.\*;***

**Example**

import static java.lang.Math.\*; //importing all static member of **Math** class

#### Example without using static import

public class Test

{

public static void main(String[] args)

{

System.out.println(**Math.sqrt(144)**);

}

}

**Output :**

12

#### Example using static import

**import static java.lang.Math.\*;**

public class Test

{

public static void main(String[] args)

{

System.out.println(**sqrt(144)**);

}

}

**Output :**

12

### Abstract class

If a class contain any abstract method then the class is declared as abstract class. An abstract class is never instantiated. It is used to provide abstraction. Although it does not provide 100% abstraction because it can also have concrete method.

**Syntax :**

**abstract** class *class\_name* { }

#### Abstract method

Method that are declared without any body within an abstract class are called **abstract method**. The method body will be defined by its subclass. Abstract method can never be final and static. Any class that extends an abstract class must implement all the abstract methods declared by the super class.

**Syntax :**

**abstract** return\_type *function\_name* (); // No definition

#### Example of Abstract class

**abstract** class *A*

{

**abstract** void *callme*();

}

class *B* extends *A*

{

void callme()

{

System.out.println("this is callme.");

}

public static void main(String[] args)

{

B b = new B();

b.*callme*();

}

}

**Output :**

this is callme.

#### Abstract class with concrete(normal) method.

Abstract classes can also have normal methods with definitions, along with abstract methods.

**abstract** class *A*

{

**abstract** void *callme*();

public void normal()

{

System.out.println("this is concrete method");

}

}

class *B* extends *A*

{

void callme()

{

System.out.println("this is callme.");

}

public static void main(String[] args)

{

B b = new B();

b.*callme*();

b.*normal*();

}

}

**Output :**

this is callme.

this is concrete method.

#### Points to Remember

1. Abstract classes are not Interfaces. They are different, we will study this when we will study Interfaces.
2. An abstract class may or may not have an abstract method. But if any class has even a single abstract method, then it must be declared abstract.
3. Abstract classes can have Constructors, Member variables and Normal methods.
4. Abstract classes are never instantiated.
5. When you extend Abstract class with abstract method, you must define the abstract method in the child class, or make the child class abstract.

#### Abstraction using abstract class

Abstraction is an important feature of OOPS. It means hiding complexity. Abstract class is used to provide abstraction. Although it does not provide 100% abstraction because it can also have concrete method. Lets see how abstract class is used to provide abstraction.

**abstract** class *Vehicle*

{

public **abstract** void *engine*();

}

public class *Car* extends *Vehicle* {

public void *engine*()

{

System.out.println("Car engine");

//car engine implementation

}

public static void main(String[] args)

{

Vehicle v = new Car();

v.*engine*();

}

}

**Output :**

Car engine

Here by casting instance of **Car** type to **Vehicle** reference, we are hiding the complexity of **Car** type under **Vechicle**. Now the **Vehicle** reference can be used to provide the implementation but it will hide the actual implementation process.

#### When to use Abstract Methods & Abstract Class?

Abstract methods are usually declared where two or more subclasses are expected to do a similar thing in different ways through different implementations. These subclasses extend the same Abstract class and provide different implementations for the abstract methods.

Abstract classes are used to define generic types of behaviors at the top of an object-oriented programming class hierarchy, and use its subclasses to provide implementation details of the abstract class.

### Interface

Interface is a pure abstract class.They are syntactically similar to classes, but you cannot create instance of an **Interface** and their methods are declared without any body. Interface is used to achieve complete **abstraction** in Java. When you create an interface it defines what a class can do without saying anything about how the class will do it.

**Syntax :**

interface interface\_name { }

#### Example of Interface

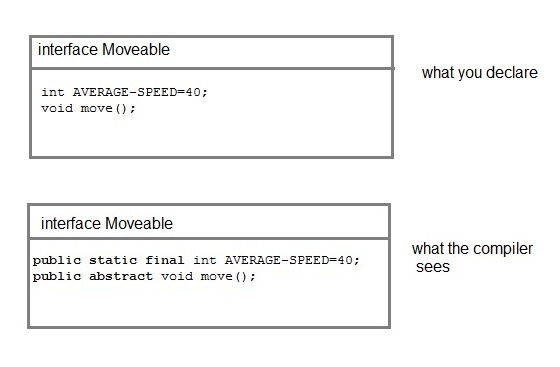
interface Moveable

{

int AVERAGE-SPEED=40;

void move();

}



**NOTE :** Compiler automatically converts methods of Interface as public and abstract, and the data members as public, static and final by default.

#### Rules for using Interface

* Methods inside Interface must not be static, final, native or strictfp.
* All variables declared inside interface are implicitly public static final variables(constants).
* All methods declared inside Java Interfaces are implicitly public and abstract, even if you don't use public or abstract keyword.
* Interface can extend one or more other interface.
* Interface cannot implement a class.
* Interface can be nested inside another interface.

#### Example of Interface implementation

interface Moveable

{

int AVG-SPEED = 40;

void move();

}

class Vehicle implements Moveable

{

public void move()

{

System .out. print in ("Average speed is"+AVG-SPEED");

}

public static void main (String[] arg)

{

Vehicle vc = new Vehicle();

vc.move();

}

}

**Output :**

Average speed is 40.

#### Interfaces supports Multiple Inheritance

Though classes in java doesn't suppost multiple inheritance, but a class can implement more than one interface.

interface Moveable

{

boolean isMoveable();

}

interface Rollable

{

boolean isRollable

}

class Tyre implements Moveable, Rollable

{

int width;

boolean isMoveable()

{

return true;

}

boolean isRollable()

{

return true;

}

public static void main(String args[])

{

Tyre tr=new Tyre();

System.out.println(tr.isMoveable());

System.out.println(tr.isRollable());

}

}

**Output :**

true

true

#### Interface extends other Interface

Classes implements interfaces, but an interface extends other interface.

interface NewsPaper

{

news();

}

interface Magazine extends NewsPaper

{

colorful();

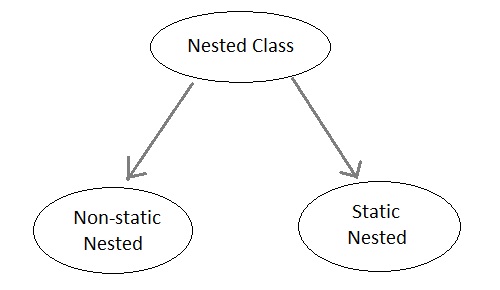
}

#### Difference between an interface and an abstract class?

| **Abstract class** | **Interface** |
| --- | --- |
| Abstract class is a class which contain one or more abstract methods, which has to be implemented by its sub classes. | Interface is a Java Object containing method declaration but no implementation. The classes which implement the Interfaces must provide the method definition for all the methods. |
| Abstract class is a Class prefix with an abstract keyword followed by Class definition. | Interface is a pure abstract class which starts with interface keyword. |
| Abstract class can also contain concrete methods. | Whereas, Interface contains all abstract methods and final variable declarations. |
| Abstract classes are useful in a situation that Some general methods should be implemented and specialization behavior should be implemented by child classes. | Interfaces are useful in a situation that all properties should be implemented. |

### Nested Class

A class within another class is known as Nested class. The scope of the nested is bounded by the scope of its enclosing class.



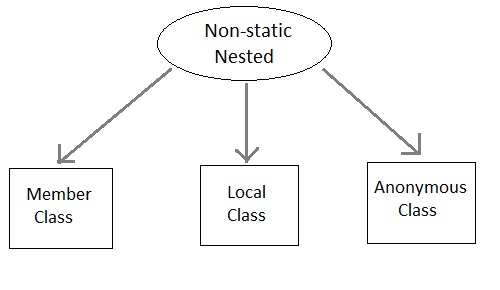
#### Static Nested Class

A satic nested class is the one that has **static** modifier applied. Because it is static it cannot refer to non-static members of its enclosing class directly. Because of this restriction static nested class is seldom used.

#### Non-static Nested class

Non-static Nested class is most important type of nested class. It is also known as **Inner** class. It has access to all variables and methods of **Outer** class and may refer to them directly. But the reverse is not true, that is, **Outer** class cannot directly access members of **Inner** class.

One more important thing to notice about an **Inner** class is that it can be created only within the scope of **Outer** class. Java compiler generates an error if any code outside **Outer** class attempts to instantiate **Inner** class.



#### Example of Inner class

class Outer

{

public void display()

{

Inner in=new Inner();

in.show();

}

class Inner

{

public void show()

{

System.out.println("Inside inner");

}

}

}

class Test

{

public static void main(String[] args)

{

Outer ot=new Outer();

ot.display();

}

}

**Output :**

Inside inner

#### Example of Inner class inside a method

class Outer

{

int count;

public void display()

{

for(int i=0;i<5;i++)

{

class Inner //Inner class defined inside for loop

{

public void show()

{

System.out.println("Inside inner "+(count++));

}

}

Inner in=new Inner();

in.show();

}

}

}

class Test

{

public static void main(String[] args)

{

Outer ot=new Outer();

ot.display();

}

}

**Output :**

Inside inner 0

Inside inner 1

Inside inner 2

Inside inner 3

Inside inner 4

#### Example of Inner class instantiated outside Outer class

class Outer

{

int count;

public void display()

{

Inner in=new Inner();

in.show();

}

class Inner

{

public void show()

{

System.out.println("Inside inner "+(++count));

}

}

}

class Test

{

public static void main(String[] args)

{

Outer ot=new Outer();

Outer.Inner in= ot.new Inner();

in.show();

}

}

**Output :**

Inside inner 1

#### Annonymous class

A class without any name is called Annonymous class.

interface Animal

{

void type();

}

public class ATest {

public static void main(String args[])

{

Animal an = new Animal(){ //Annonymous class created

public void type()

{

System.out.println("Annonymous animal");

}

};

an.type();

}

}

**Output :**

Annonymous animal

Here a class is created which implements **Animal** interace and its name will be decided by the compiler. This annonymous class will provide implementation of **type()** method.

# String Handling

### Introduction to String Handling

String is probably the most commonly used class in java library. String class is encapsulated under java.lang package. In java, every string that you create is actually an object of type **String**. One important thing to notice about string object is that string objects are immutable that means once a string object is created it cannot be altered.

#### What is an Immutable object?

An object whose state cannot be changed after it is created is known as an Immutable object. String, Integer, Byte, Short, Float, Double and all other wrapper class's objects are immutable.

#### Creating an Immutable class

public final class MyString

{

final String str;

MyString(String s)

{

this.str = s;

}

public String get()

{

return str;

}

}

In this example **MyString** is an immutable class. **MyString**'s state cannot be changed once it is created.

#### Creating a String object

String can be created in number of ways, here are a few ways of creating string object.

#### 1) Using a String literal

String literal is a simple string enclosed in double quotes " ". A string literal is treated as a String object.

String str1 = "Hello";

#### 2) Using another String object

String str2 = new String(str1);

#### 3) Using new Keyword

String str3 = new String("Java");

#### 4) Using + operator (Concatenation)

String str4 = str1 + str2;

or,

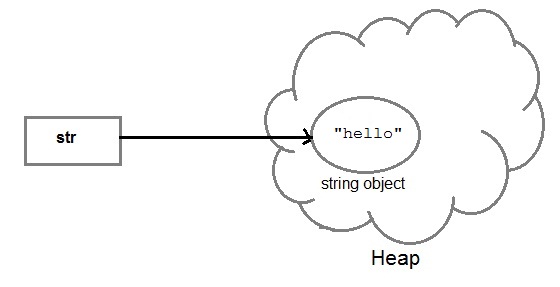
String str5 = "hello"+"Java";

Each time you create a String literal, the JVM checks the string pool first. If the string literal already exists in the pool, a reference to the pool instance is returned. If string does not exist in the pool, a new string object is created, and is placed in the pool. String objects are stored in a special memory area known as **string constant pool** inside the heap memory.

#### String object and How they are stored

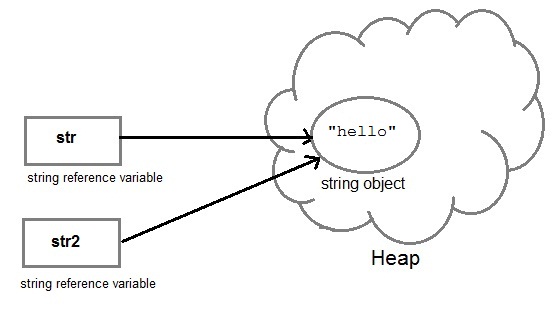
When we create a new string object using string literal, that string literal is added to the string pool, if it is not present there already.

String str= "Hello";



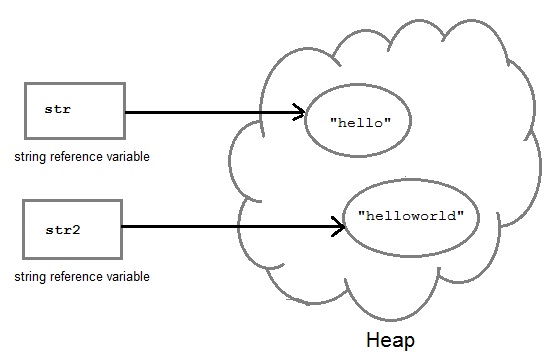
And, when we create another object with same string, then a reference of the string literal already present in string pool is returned.

String str2=str;



But if we change the new string, its reference gets modified.

str2=str2.concat("world");



#### Concatenating String

There are 2 methods to concatenate two or more string.

1. Using **concat()** method
2. Using + operator

#### 1) Using concat() method

string s = "Hello";

string str = "Java";

string str2 = s.concat(str);

String str1 = "Hello".concat("Java"); //works with string literals too.

#### 2) Using + operator

string str = "Rahul";

string str1 = "Dravid";

string str2 = str + str1;

string st = "Rahul"+"Dravid";

#### String Comparison

String comparison can be done in 3 ways.

1. Using **equals()** method
2. Using == operator
3. By **CompareTo()** method

#### Using equals() method

equals() method compares two strings for equality. Its general syntax is,

*boolean* **equals** (Object *str*)

It compares the content of the strings. It will return **true** if string matches, else returns **false**.

String s = "Hell";

String s1 = "Hello";

String s2 = "Hello";

s1.equals(s2); //true

s.equals(s1) ; //false

#### Using == operator

== operator compares two object references to check whether they refer to same instance. This also, will return **true** on successful match.

String s1 = "Java";

String s2 = "Java";

String s3 = new string ("Java");

test(Sl == s2) //true

test(s1 == s3) //false

#### By compareTo() method

compareTo() method compares values and returns an int which tells if the string compared is less than, equal to or greater than th other string. Its general syntax is,

*int* **compareTo**(String *str*)

To use this function you must implement the **Comparable Interface**. compareTo() is the only function in Comparable Interface.

String s1 = "Abhi";

String s2 = "Viraaj";

String s3 = "Abhi";

s1.compareTo(S2); //return -1 because s1 < s2

s1.compareTo(S3); //return 0 because s1 == s3

s2.compareTo(s1); //return 1 because s2 > s1

### String class function

The following methods are some of the most commonly used methods of String class.

#### charAt()

**charAt()** function returns the character located at the specified index.

String str = "studytonight";

System.out.println(str.charAt(2));

Output : u

#### equalsIgnoreCase()

**equalsIgnoreCase()** determines the equality of two Strings, ignoring thier case (upper or lower case doesn't matters with this fuction ).

String str = "java";

System.out.println(str.equalsIgnoreCase("JAVA"));

Output : true

#### length()

**length()** function returns the number of characters in a String.

String str = "Count me";

System.out.println(str.length());

Output : 8

#### replace()

**replace()** method replaces occurances of character with a specified new character.

String str = "Change me";

System.out.println(str.replace('m','M'));

Output : Change Me

#### substring()

**substring()** method returns a part of the string. **substring()** method has two forms,

public String substring(int begin);

public String substring(int begin, int end);

The first argument represents the starting point of the subtring. If the substring() method is called with only one argument, the subtring returned, will contain characters from specified starting point to the end of original string.

But, if the call to substring() method has two arguments, the second argument specify the end point of substring.

String str = "0123456789";

System.out.println(str.substring(4));

Output : 456789

System.out.println(str.substring(4,7));

Output : 456

#### toLowerCase()

**toLowerCase()** method returns string with all uppercase characters converted to lowercase.

String str = "ABCDEF";

System.out.println(str.toLowerCase());

Output : abcdef

#### valueOf()

Overloaded version of valueOf() method is present in String class for all primitive data types and for type Object.

**NOTE :** valueOf() function is used to convert **primitive data types** into Strings.

But for objects, valueOf() method calls **toString()** function.

#### toString()

**toString()** method returns the string representation of the object used to invoke this method. **toString()** is used to represent any Java Object into a meaningful string representation. It is declared in the **Object class**, hence can be overrided by any java class. (Object class is super class of all java classes.)

public class Car {

public static void main(String args[])

{

Car c=new Car();

System.out.println(c);

}

public String toString()

{

return "This is my car object";

}

}

Output : This is my car object

Whenever we will try to print any object of class Car, its toString() function will be called. toString() can also be used with normal string objects.

String str = "Hello World";

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

Output : Hello World

#### toString() with Concatenation

Whenever we concatenate any other primitive data type, or object of other classes with a String object, **toString()** function or **valueOf()** function is called automatically to change the other object or primitive type into string, for successful concatenation.

int age = 10;

String str = "He is" + **age** + "years old.";

In above case **10** will be automatically converted into string for concatenation using **valueOf()** function.

#### toUpperCase()

This method returns string with all lowercase character changed to uppercase.

String str = "abcdef";

System.out.println(str.toUpperCase());

Output : ABCDEF

#### trim()

This method returns a string from which any leading and trailing whitespaces has been removed.

String str = " hello ";

System.out.println(str.trim());

Output : hello

### StringBuffer class

StringBuffer class is used to create a **mutable** string object. It represents growable and writable character sequence. As we know that String objects are immutable, so if we do a lot of changes with **String** objects, we will end up with a lot of memory leak.

So **StringBuffer** class is used when we have to make lot of modifications to our string. It is also thread safe i.e multiple threads cannot access it simultaneously. StringBuffer defines 4 constructors. They are,

1. **StringBuffer** ( )
2. **StringBuffer** ( *int size* )
3. **StringBuffer** ( *String str* )
4. **StringBuffer** ( *charSequence [ ]ch* )

* StringBuffer() creates an empty string buffer and reserves room for 16 characters.
* stringBuffer(int size) creates an empty string and takes an integer argument to set capacity of the buffer.

#### Example showing difference between String and StringBuffer

class Test {

public static void main(String args[])

{

String str = "study";

str.concat("tonight");

System.out.println(str); // Output: study

StringBuffer strB = new StringBuffer("study");

strB.append("tonight");

System.out.println(strB); // Output: studytonight

}

}

#### Important methods of StringBuffer class

The following methods are some most commonly used methods of StringBuffer class.

#### append()

This method will concatenate the string representation of any type of data to the end of the invoking **StringBuffer** object. append() method has several overloaded forms.

*StringBuffer* **append**(*String str*)

*StringBuffer* **append**(*int n*)

*StringBuffer* **append**(*Object obj*)

The string representation of each parameter is appended to **StringBuffer** object.

StringBuffer str = new StringBuffer("test");

str.append(123);

System.out.println(str);

Output : test123

#### insert()

This method inserts one string into another. Here are few forms of insert() method.

*StringBuffer* **insert**(*int index*, *String str*)

*StringBuffer* **insert**(*int index*, *int num*)

*StringBuffer* **insert**(*int index*, *Object obj*)

Here the first parameter gives the index at which position the string will be inserted and string representation of second parameter is inserted into **StringBuffer** object.

StringBuffer str = new StringBuffer("test");

str.insert(4, 123);

System.out.println(str);

Output : test123

#### reverse()

This method reverses the characters within a **StringBuffer** object.

StringBuffer str = new StringBuffer("Hello");

str.reverse();

System.out.println(str);

Output : olleH

#### replace()

This method replaces the string from specified start index to the end index.

StringBuffer str = new StringBuffer("Hello World");

str.replace( 6, 11, "java");

System.out.println(str);

Output : Hello java

#### capacity()

This method returns the current capacity of **StringBuffer** object.

StringBuffer str = new StringBuffer();

System.out.println( str.capacity() );

Output : 16

#### ensureCapacity()

This method is used to ensure minimum capacity of **StringBuffer** object.

StringBuffer str = new StringBuffer("hello");

str.ensureCapacity(10);

### StringBuilder class

StringBuilder is identical to StringBuffer except for one important difference it is not synchronized, which means it is not thread safe. Its because StringBuilder methods are not synchronised.

#### StringBuilder Constructors

1. **StringBuilder** ( ), creates an empty StringBuilder and reserves room for 16 characters.
2. **StringBuilder** ( *int size* ), create an empty string and takes an integer argument to set capacity of the buffer.
3. **StringBuilder** ( *String str* ), create a StringBuilder object and initialize it with string str.

#### Difference between StringBuffer and StringBuilder class

|  |  |
| --- | --- |
| **StringBuffer class** | **StringBuilder class** |
| StringBuffer is synchronized. | StringBuilder is not synchronized. |
| Because of synchronisation, StringBuffer operation is slower than StringBuilder. | StringBuilder operates faster. |

#### Example of StringBuilder

class Test {

public static void main(String args[])

{

StringBuilder str = new StringBuilder("study");

str.append( "tonight" );

System.out.println(str);

str.replace( 6, 13, "today");

System.out.println(str);

str.reverse();

System.out.println(str);

str.replace( 6, 13, "today");

}

}

**Output :**

studytonight

studyttoday

yadottyduts

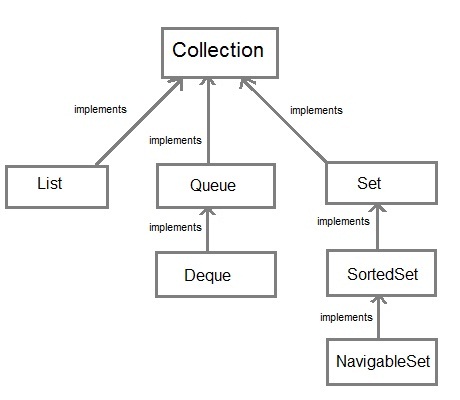
# Collection Framework

### Collection Framework

Collection framework was not part of original Java release. Collections was added to J2SE 1.2. Prior to Java 2, Java provided adhoc classes such as Dictionary, Vector, Stack and Properties to store and manipulate groups of objects. Collection framework provides many important classes and interfaces to collect and organize group of alike objects.

#### Important Interfaces of Collection API

|  |  |
| --- | --- |
| **Interface** | **Description** |
| **Collection** | Enables you to work with groups of object; it is at the top of collection hierarchy |
| **Deque** | Extends queue to handle double ended queue. |
| **List** | Extends collection to handle sequences list of object. |
| **Queue** | Extends collection to handle special kind of list in which element are removed only from the head. |
| **Set** | Extends collection to handle sets, which must contain unique element. |
| **SortedSet** | Extends sets to handle sorted set. |



All these Interfaces give several methods which are defined by collections classes which implement these interfaces.

#### Why Collections were made Generic ?

Generics added **type safety** to Collection framework. Earlier collections stored **Object class** references. Which means any collection could store any type of object. Hence there were chances of storing incompatible types in a collection, which could result in run time mismatch. Hence Generics was introduced, now you can explicitly state the type of object being stored.

#### Collections and Autoboxing

We have studied that Autoboxing converts primitive types into Wrapper class Objects. As collections doesn't store primitive data types(stores only refrences), hence Autoboxing facilitates the storing of primitive data types in collection by boxing it into its wrapper type.

#### Most Commonly thrown Exceptions in Collection Framework

|  |  |
| --- | --- |
| **Exception Name** | **Description** |
| UnSupportedOperationException | occurs if a Collection cannot be modified |
| ClassCastException | occurs when one object is incompatible with another |
| NullPointerException | occurs when you try to store null object in Collection |
| IllegalArgumentException | thrown if an invalid argument is used |
| IllegalStateException | thrown if you try to add an element to an already full Collection |

### Interfaces of Collection Framework

The collection framework has a lot of Interfaces, setting the fundamental nature of various collection classes. Lets study the most important Interfaces in the collection framework.

#### The Collection Interface

1. It is at the top of collection heirarchy and must be implemented by any class that defines a collection. Its general declaration is,
2. *interface* **Collection** < *E* >
3. Following are some of the commonly used methods in this interface.

|  |  |
| --- | --- |
| **Methods** | **Description** |
| add( E obj ) | Used to add objects to a collection. Doesn't add duplicate elements to the collection. |
| addAll( Collection C ) | Add all elements of collection C to the invoking collection |
| remove( Object obj ) | To remove an object from collection |
| removeAll( Collection C ) | Removes all element of collection C from the invoking collection |
| contains( Object obj ) | To determine whether an object is present in collection or not |
| isEmpty() | Returns true if collection is empty, else returns false |
| size() | returns number of elements present in collection |

#### The List Interface

1. It extends the **Collection** Interface, and defines storage as sequence of elements. Following is its general declaration,
2. *interface* **List** < *E* >
3. Allows random access and insertion, based on position.
4. It allows Duplicate elements.
5. Apart from methods of Collection Interface, it adds following methods of its own.

|  |  |
| --- | --- |
| **Methods** | **Description** |
| get( int index ) | Returns object stored at the specified index |
| set( int index, E obj) | Stores object at the specified index in the calling collection |
| indexOf( Object obj ) | Returns index of first occurence of obj in the collection |
| lastIndexOf( Object obj ) | Returns index of last occurence of obj in the collection |
| subList( int start, int end ) | Returns a list containing elements between start and end index in the collection |

#### The Set Interface

1. This interface defines a Set. It extends **Collection** interface and doesn't allow insertion of duplicate elements. It's general declaration is,
2. *interface* **Set** < *E* >
3. It doesn't define any method of its own. It has two sub interfaces, **SortedSet** and **NavigableSet**.
4. **SortedSet** interface extends **Set** interface and arranges added elements in an ascending order.
5. **NavigabeSet** interface extends **SortedSet** interface, and allows retrieval of elements based on the closest match to a given value or values.

#### The Queue Interface

1. It extends **collection** interface and defines behaviour of queue, that is first-in, first-out. It's general declaration is,
2. *interface* **Queue** < *E* >
3. There are couple of new and interestin methods added by this interface. Some of them are mentioned in below table.

|  |  |
| --- | --- |
| **Methods** | **Description** |
| poll() | removes element at the head of the queue and returns **null** if queue is empty |
| remove() | removes element at the head of the queue and throws **NoSuchElementException** if queue is empty |
| peek() | returns the element at the head of the queue without removing it. Returns **null** if queue is empty |
| element() | same as peek(), but throws **NoSuchElementException** if queue is empty |
| offer( E obj ) | Adds object to queue. |

#### The Dequeue Interface

1. It extends **Queue** interface and declares behaviour of a double-ended queue. Its general declaration is,
2. *interface* **Dequeue** < *E* >
3. Double ended queues can function as simple queues as well as like standard Stacks.

### The Collection classes

Java provides a set of Collection classes that implements Collection interface. Some of these classes provide full implementations that can be used as it is and other abstract classes provides skeletal implementations that can be used as starting points for creating concrete collections.

### ArrayList class

1. ArrayList class extends **AbstractList** class and implements the **List** interface.
2. ArrayList supports dynamic array that can grow as needed. ArrayList has three constructors.
3. **ArrayList**()
4. **ArrayList**( *Collection* C )
5. **ArrayList**( int *capacity* )
6. ArrayLists are created with an initial size, when this size is exceeded, it gets enlarged automatically.
7. It can contain Duplicate elements and maintains the insertion order.
8. ArrayLists are not synchronized.

#### Example of ArrayList

import java.util.\*

class Test

{

public static void main(String[] args)

{

**ArrayList< String> al = new ArrayList< String>()**;

al.add("ab");

al.add("bc");

al.add("cd");

system.out.println(al);

}

}

**Output :**

[ab,bc,cd]

#### Getting Array from an ArrayList

toArray() method is used to get an araay containing all the contents of the list. Following are the reasons why you must obtain array from your ArrayList whenever required.

* To obtain faster processing.
* To pass array to methods who do not accept Collectionn as arguments.
* To integrate and use collections with legacy code.

#### Storing User-Defined classes

In the above example we are storing only string object in ArrayList collection. But You can store any type of object, including object of class that you create in Collection classes.

#### Example of storing User-Defined object

**Contact class**

class Contact

{

String first\_name;

String last\_name;

String phone\_no;

public Contact(String fn,String ln,String pn)

{

first\_name = fn;

last\_name = ln;

phone\_no = pn;

}

public String toString()

{

return first\_name+" "+last\_name+"("+phone\_no+")";

}

}

**Storing Contact class**

public class PhoneBook

{

public static void main(String[] args)

{

Contact c1 = new Contact("Ricky", "Pointing","999100091");

Contact c2 = new Contact("David", "Beckham","998392819");

Contact c3 = new Contact("Virat", "Kohli","998131319");

**ArrayList< Contact> al = new ArrayList< Contact>();**

al.add(c1);

al.add(c2);

al.add(c3);

System.out.println(al);

}

}

**Output:**

[Ricky Pointing(999100091), David Beckham(998392819), Virat Kohli(998131319)]

### LinkedList class

1. LinkedList class extends **AbstractSequentialList** and implements **List**,**Deque** and **Queue** inteface.
2. It can be used as List, stack or Queue as it implements all the related interfaces.
3. It can contain duplicate elements and is not synchronized.

#### Example of LinkedList class

import java.util.\* ;

class Test

{

public static void main(String[] args)

{

**LinkedList< String> ll = new LinkedList< String>();**

ll.add("a");

ll.add("b");

ll.add("c");

ll.addLast("z");

ll.addFirst("A");

System.out.println(ll);

}

}

**Output:**

[A, a, b,c, z]

### HashSet class

1. HashSet extends **AbstractSet** class and implements the **Set** interface.
2. It creates a collection that uses hash table for storage.
3. HashSet does not maintain any order of elements.

#### Example of HashSet class

import java.util.\*;

class HashSetDemo

{

public static void main(String args[])

{

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

hs.add("B");

hs.add("A");

hs.add("D");

hs.add("E");

hs.add("C");

hs.add("F");

System.out.println(hs);

}

}

**Output:**

[D, E, F, A, B, C]

### LinkedHashSet Class

1. LinkedHashSet class extends **HashSet** class
2. LinkedHashSet maintains a linked list of entries in the set.
3. LinkedHashSet stores elements in the order in which elements are inserted.

#### Example of LinkedHashSet class

import java.util.\*;

class LinkedHashSetDemo

{

public static void main(String args[])

{

**LinkedHashSet< String> hs = new LinkedHashSet< String>()**;

hs.add("B");

hs.add("A");

hs.add("D");

hs.add("E");

hs.add("C");

hs.add("F");

System.out.println(hs);

}

}

**Output :**

[B, A, D, E, C, F]

### TreeSet Class

1. It extends **AbstractSet** class and implements the **NavigableSet** interface.
2. It stores elements sorted ascending order.
3. Uses a Tree structure to store elements.
4. Access and retrieval times are quite fast.
5. It has four Constructors.

**TreeSet**()

**TreeSet**( Collection *C* )

**TreeSet**( Comparator *comp* )

**TreeSet**( SortedSet *ss* )

### Accessing a Collection

To access, modify or remove any element from any collection we need to first find the element, for which we have to cycle throught the elements of the collection. There are three possible ways to cycle through the elements of any collection.

1. Using Iterator interface
2. Using ListIterator interface
3. Using for-each loop

#### Accessing elements using Iterator

Iterator Interface is used to traverse a list in forward direction, enabling you to remove or modify the elements of the collection. Each collection classes provide **iterator()** method to return an iterator.

import java.util.\*;

class Test\_Iterator

{

public static void main(String[] args)

{

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

ar.add("ab");

ar.add("bc");

ar.add("cd");

ar.add("de");

**Iterator it = ar.iterator()**; //Declaring Iterator

while(it.hasNext())

{

System.out.print(it.next()+" ");

}

}

}

**Output :**

ab bc cd de

#### Accessing element using ListIterator

ListIterator Interface is used to traverse a list in both forward and backward direction. It is available to only those collections that implement the **List** Interface.

import java.util.\*;

class Test\_Iterator

{

public static void main(String[] args)

{

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

ar.add("ab");

ar.add("bc");

ar.add("cd");

ar.add("de");

**ListIterator litr = ar.listIterator()**;

while(litr.hasNext()) //In forward direction

{

System.out.print(litr.next()+" ");

}

while(litr.hasPrevious()) //In backward direction

{

System.out.print(litr.next()+" ");

}

}

}

**Output :**

ab bc cd de

de cd bc ab

#### Using for-each loop

for-each version of for loop can also be used for traversing each element of a collection. But this can only be used if we don't want to modify the contents of a collection and we don't want any reverse access. for-each loop can cycle through any collection of object that implements Iterable interface.

import java.util.\*;

class ForEachDemo

{

public static void main(String[] args)

{

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

ls.add("a");

ls.add("b");

ls.add("c");

ls.add("d");

**for(String str : ls)**

{

System.out.print(str+" ");

}

}

}

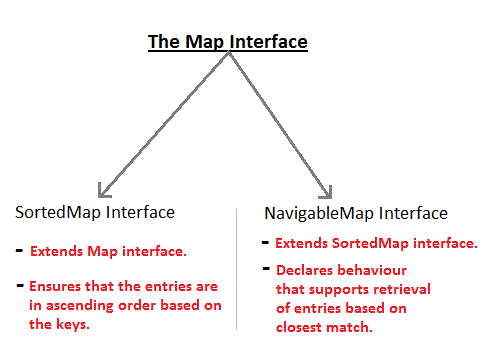
**Output :**

a b c d

### Map Interface

A Map stores data in key and value association. Both key and values are objects. The key must be unique but the values can be duplicate. Although Maps are a part of Collection Framework, they can not actually be called as collections because of some properties that they posses. However we can obtain a **collection-view** of maps.

|  |  |
| --- | --- |
| **Interface** | **Description** |
| **Map** | Maps unique key to value. |
| **Map.Entry** | Describe an element in key and value pair in a map. This is an inner class of map. |
| **NavigableMap** | Extends SortedMap to handle the retrienal of entries based on closest match searches |
| **SortedMap** | Extends Map so that key are maintained in an ascending order. |



#### Commonly used Methods defined by Map

* boolean **containsKey**(Object *k*): returns true if map contain *k* as key. Otherwise false.
* Object **get**(Object *k*) : returns values associated with the key *k*.
* Object **put**(Object *k*, Object *v*) : stores an entry in map.
* Object **putAll**(Map *m*) : put all entries from *m* in this map.
* Set **keySet**() : returns **Set** that contains the key in a map.
* Set **entrySet**() : returns **Set** that contains the entries in a map.

#### HashMap class

1. HashMap class extends **AbstractMap** and implements **Map** interface.
2. It uses a **hashtable** to store the map. This allows the execution time of get() and put() to remain same.
3. HashMap has four constructor.
4. HashMap()
5. HashMap(Map< ? extends k, ? extends V> m)
6. HashMap(int capacity)
7. HashMap(int capacity, float fillratio)
8. HashMap does not maintain order of its element.

**Example**

import java.util.\*;

class HashMapDemo

{

public static void main(String args[])

{

**HashMap< String,Integer> hm = new HashMap< String,Integer>()**;

hm.put("a",new Integer(100));

hm.put("b",new Integer(200));

hm.put("c",new Integer(300));

hm.put("d",new Integer(400));

**Set< Map.Entry< String,Integer> > st = hm.entrySet()**; //returns Set view

for(Map.Entry< String,Integer> me:st)

{

System.out.print(me.getKey()+":");

System.out.println(me.getValue());

}

}

}

**Output :**

c 300

a 100

d 400

b 200

#### TreeMap class

1. TreeMap class extends **AbstractMap** and implements **NavigableMap** interface.
2. It creates Map, stored in a tree structure.
3. A **TreeMap** provides an efficient means of storing key/value pair in efficient order.
4. It provides key/value pairs in sorted order and allows rapid retrieval.

**Example**

import java.util.\*;

class TreeMapDemo

{

public static void main(String args[])

{

**TreeMap< String,Integer> tm = new TreeMap< String,Integer>()**;

tm.put("a",new Integer(100));

tm.put("b",new Integer(200));

tm.put("c",new Integer(300));

tm.put("d",new Integer(400));

**Set< Map.Entry< String,Integer> > st = tm.entrySet()**;

for(Map.Entry me:st)

{

System.out.print(me.getKey()+":");

System.out.println(me.getValue());

}

}

}

**Output :**

a 100

b 200

c 300

d 400

#### LinkedHashMap class

1. **LinkedHashMap** extends **HashMap** class.
2. It maintains a linked list of entries in map in order in which they are inserted.
3. **LinkedHashMap** defines the following constructor
4. LinkedHashMap()
5. LinkedHashMap(Map< ? extends k, ? extends V> m)
6. LinkedHashMap(int capacity)
7. LinkedHashMap(int capacity, float fillratio)
8. LinkedHashMap(int capacity, float fillratio, boolean order)
9. It adds one new method removeEldestEntry(). This method is called by put() and putAll() By default this method does nothing. However we can override this method to remove oldest element in the map. **Syntax**
10. protected boolean removeEldestEntry(Map.Entry e)

#### EnumMap class

1. **EnumMap** extends **AbstractMap** and implements **Map** interface.
2. It is used for key as enum

### Comparator Interface

In Java, Comparator interface is used to order the object in your own way. It gives you ability to decide how element are stored within sorted collection and map.

Comparator Interface defines compare() method. This method compare two object and return 0 if two object are equal. It returns a positive value if object1 is greater than object2. Otherwise a negative value is return. The method can throw a **ClassCastException** if the type of object are not compatible for comparison.

#### Example

**Student class**

class Student

int roll;

String name;

Student(int r,String n)

{

roll = r;

name = n;

}

public String toString()

{

return roll+" "+name;

}

**MyComparator class**

This class defines the comparison logic for Student class based on their roll. Student object will be sotred in ascending order of their roll.

class MyComparator implements Comparator

{

public int compare(Student s1,Student s2)

{

if(s1.roll == s2.roll) return 0;

else if(s1.roll > s2.roll) return 1;

else return -1;

}

}

public class Test

{

public static void main(String[] args)

{

TreeSet< Student> ts = new TreeSet< Student>(**new MyComparator()**);

ts.add(new Student(45, "Rahul"));

ts.add(new Student(11, "Adam"));

ts.add(new Student(19, "Alex"));

System.out.println(ts);

}

}

**Output :**

[ 11 Adam, 19 Alex, 45 Rahul ]

As you can see in the ouput Student object are stored in ascending order of their **roll**.

### Legacy Classes

Early version of java did not include the **Collection** framework. It only defined several classes and interface that provide method for storing objects. When **Collection** framework were added in J2SE 1.2, the original classes were reengineered to support the collection interface. These classes are also known as Legacy classes. All legacy claases and interface were redesign by JDK 5 to support Generics.

The following are the legacy classes defined by **java.util** package

1. Dictionary
2. HashTable
3. Properties
4. Stack
5. Vector

There is only one legacy interface called **Enumeration**

**NOTE:** All the legacy classes are syncronized

#### Enumeration interface

1. **Enumeration** interface defines method to enumerate through collection of object.
2. This interface is suspended by **Iterator** interface.
3. However some legacy classes such as **Vector** and **Properties** defines several method in which **Enumeration** interface is used.
4. It specifies the following two methods

boolean hasMoreElements()

Object nextElement()

#### Vector class

1. **Vector** is similar to **ArrayList** which represents a dynamic array.
2. The only difference between **Vector** and **ArrayList** is that Vector is synchronised while Array is not.
3. Vector class has following four constructor
4. Vector()
5. Vector(int size)
6. Vector(int size, int incr)
7. Vector(Collection< ? extends E> c)

Vector defines several legacy method. Lets see some important legacy method define by **Vector** class.

|  |  |
| --- | --- |
| **Method** | **Description** |
| addElement() | add element to the Vector |
| elementAt() | return the element at specified index |
| elements | return an enumeration of element in vector |
| firstElement() | return first element in the Vector |
| lastElement() | return last element in the Vector |
| removeAllElement() | remove all element of the Vector |

#### Example of Vector

import java.util.\*;

public class Test

{

public static void main(String[] args)

{

Vector ve = new Vector();

ve.add(10);

ve.add(20);

ve.add(30);

ve.add(40);

ve.add(50);

ve.add(60);

Enumeration en = ve.elements();

while(en.hasMoreElements())

{

System.out.println(en.nextElement());

}

}

}

**Output :**

10

20

30

40

50

60

#### Hashtable class

1. Like HashMap, Hashtable also stores key/value pair in hashtable. However neither **keys** nor **values** can be **null**.
2. There is one more difference between **HashMap** and **Hashtable** that is Hashtable is synchronized while HashMap is not.
3. Hashtable has following four constructor
4. Hashtable()
5. Hashtable(int size)
6. Hashtable(int size, float fillratio)
7. Hashtable(Map< ? extends K, ? extends V> m)

#### Example of Hashtable

import java.util.\*;

class HashTableDemo

{

public static void main(String args[])

{

**Hashtable< String,Integer> ht = new Hashtable< String,Integer>()**;

ht.put("a",new Integer(100));

ht.put("b",new Integer(200));

ht.put("c",new Integer(300));

ht.put("d",new Integer(400));

Set st = ht.entrySet();

Iterator itr=st.iterator();

while(itr.hasNext())

{

Map.Entry m=(Map.Entry)itr.next();

System.out.println(itr.getKey()+" "+itr.getValue());

}

}

}

**Output:**

a 100

b 200

c 300

d 400

#### Difference between HashMap and Hashtable

|  |  |
| --- | --- |
| **Hashtable** | **HashMap** |
| Hashtable class is synchronized. | HastMap is not synchronize. |
| Because of Thread-safe, Hashtable is slower than HashMap | HashMap works faster. |
| Neither **key** nor **values** can be null | Both **key** and **values** can be null |
| Order of table remain constant over time. | does not guarantee that order of map remain constant over time. |

#### Properties class

1. **Properties** class extends **Hashtable** class.
2. It is used to maintain list of value in which both key and value are **String**
3. **Properties** class define two constructor
4. Properties()
5. Properties(Properties default)
6. One advantage of **Properties** over **Hashtable** is that we can specify a default property that will be useful when no value is associated with a certain key.

#### Example of Properties class

import java.util.\*;

public class Test

{

public static void main(String[] args)

{

Properties pr = new Properties();

pr.put("Java", "James Ghosling");

pr.put("C++", "Bjarne Stroustrup");

pr.put("C", "Dennis Ritchie");

pr.put("C#", "Microsoft Inc.");

Set< ?> creator = pr.keySet();

for(Object ob: creator)

{

System.out.println(ob+" was created by "+ pr.getProperty((String)ob) );

}

}

}

**Output :**

Java was created by James Ghosling

C++ was created by Bjarne Stroustrup

C was created by Dennis Ritchie

C# was created by Microsoft Inc

# Exception Handling

### Exception Handling

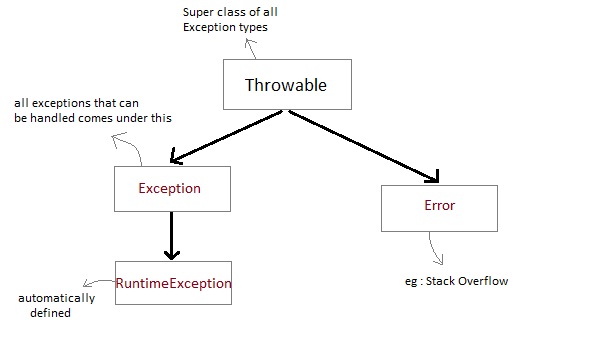
Exception Handling is the mechanism to handle runtime malfunctions. We need to handle such exceptions to prevent abrupt termination of program. The term exception means exceptional condition, it is a problem that may arise during the execution of program. A bunch of things can lead to exceptions, including programmer error, hardware failures, files that need to be opened cannot be found, resource exhaustion etc.

#### Exception

A Java Exception is an object that describes the exception that occurs in a program. When an exceptional events occurs in java, an exception is said to be thrown. The code that's responsible for doing something about the exception is called an **exception handler**.

#### Exception class Hierarchy

All exception types are subclasses of class **Throwable**, which is at the top of exception class hierarchy.



* **Exception** class is for exceptional conditions that program should catch. This class is extended to create user specific exception classes.
* **RuntimeException** is a subclass of Exception. Exceptions under this class are automatically defined for programs.

#### Exception are categorized into 3 category.

* **Checked Exception**

The exception that can be predicted by the programmer.*Example :* File that need to be opened is not found. These type of exceptions must be checked at compile time.

* **Unchecked Exception**

Unchecked exceptions are the class that extends RuntimeException. Unchecked exception are ignored at compile time. *Example :* ArithmeticException, NullPointerException, Array Index out of Bound exception. Unchecked exceptions are checked at runtime.

* **Error**

Errors are typically ignored in code because you can rarely do anything about an error. *Example :* if stack overflow occurs, an error will arise. This type of error is not possible handle in code.

#### Uncaught Exceptions

When we don't handle the exceptions, they lead to unexpected program termination. Lets take an example for better understanding.

class UncaughtException

{

public static void main(String args[])

{

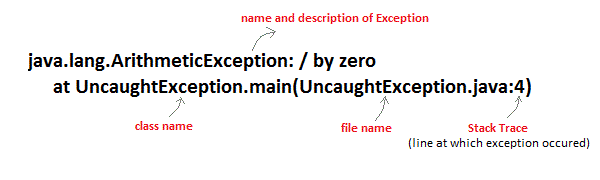
int a = 0;

int b = 7/a; // Divide by zero, will lead to exception

}

}

This will lead to an exception at runtime, hence the Java run-time system will construct an exception and then throw it. As we don't have any mechanism for handling exception in the above program, hence the default handler will handle the exception and will print the details of the exception on the terminal.



### Exception Handling Mechanism

In java, exception handling is done using five keywords,

1. **try**
2. **catch**
3. **throw**
4. **throws**
5. **finally**

Exception handling is done by transferring the execution of a program to an appropriate exception handler when exception occurs.

#### Using try and catch

Try is used to guard a block of code in which exception may occur. This block of code is called guarded region. A catch statement involves declaring the type of exception you are trying to catch. If an exception occurs in guarded code, the catch block that follows the try is checked, if the type of exception that occured is listed in the catch block then the exception is handed over to the catch block which then handles it.

#### Example using Try and catch

class Excp

{

public static void main(String args[])

{

int a,b,c;

try

{

a=0;

b=10;

c=b/a;

System.out.println("This line will not be executed");

}

catch(ArithmeticException e)

{

System.out.println("Divided by zero");

}

System.out.println("After exception is handled");

}

}

**Output :**

Divided by zero

After exception is handled

An exception will thrown by this program as we are trying to divide a number by zero inside **try** block. The program control is transfered outside **try** block. Thus the line "*This line will not be executed*" is never parsed by the compiler. The exception thrown is handle in **catch** block. Once the exception is handled the program controls continue with the next line in the program. Thus the line "*After exception is handled*" is printed.

#### Multiple catch blocks:

A try block can be followed by multiple catch blocks. You can have any number of catch blocks after a single try block.If an exception occurs in the guarded code the exception is passed to the first catch block in the list. If the exception type of exception, matches with the first catch block it gets caught, if not the exception is passed down to the next catch block. This continue until the exception is caught or falls through all catches.

#### Example for Multiple Catch blocks

class Excep

{

public static void main(String[] args)

{

try

{

int arr[]={1,2};

arr[2]=3/0;

}

catch(ArithmeticException ae)

{

System.out.println("divide by zero");

}

catch(ArrayIndexOutOfBoundsException e)

{

System.out.println("array index out of bound exception");

}

}

}

**Output :**

divide by zero

#### Example for Unreachable Catch block

While using multiple **catch** statements, it is important to remember that exception sub classes inside **catch** must come before any of their super classes otherwise it will lead to compile time error.

class Excep

{

public static void main(String[] args)

{

try

{

int arr[]={1,2};

arr[2]=3/0;

}

catch(Exception e) //This block handles all Exception

{

System.out.println("Generic exception");

}

catch(ArrayIndexOutOfBoundsException e) //This block is unreachable

{

System.out.println("array index out of bound exception");

}

}

}

#### Nested try statement

**try** statement can be **nested** inside another block of **try**. Nested try block is used when a part of a block may cause one error while entire block may cause another error. In case if inner **try** block does not have a **catch** handler for a particular exception then the outer **try** is checked for match.

class Excep

{

public static void main(String[] args)

{

try

{

int arr[]={5,0,1,2};

try

{

int x=arr[3]/arr[1];

}

catch(ArithmeticException ae)

{

System.out.println("divide by zero");

}

arr[4]=3;

}

catch(ArrayIndexOutOfBoundsException e)

{

System.out.println("array index out of bound exception");

}

}

}

#### Important points to Remember

1. If you do not explicitly use the try catch blocks in your program, java will provide a default exception handler, which will print the exception details on the terminal, whenever exception occurs.
2. Super class **Throwable** overrides **toString()** function, to display error message in form of string.
3. While using multiple catch block, always make sure that exception subclasses comes before any of their super classes. Else you will get compile time error.
4. In nested try catch, the inner try block, uses its own catch block as well as catch block of the outer try, if required.
5. Only the object of Throwable class or its subclasses can be thrown.

### Try with Resource Statement

JDK 7 introduces a new version of try statement known as **try-with-resources statement**. This feature add another way to exception handling with resources management,it is also referred to as **automatic resource management**.

**Syntax**

try(resource-specification)

{

//use the resource

}catch()

{...}

This **try statement** contains a paranthesis in which one or more resources is declare. Any object that implements java.lang.AutoCloseable or java.io.Closeable, can be passed as a parameter to **try statement**. A resource is an object that is used in program and must be closed after the program is finished. The **try-with-resources statement** ensures that each resource is closed at the end of the statement, you do not have to explicitly close the resources.

#### Example without using try with Resource Statement

import java.io.\*;

class Test

{

public static void main(String[] args)

{

try{

String str;

**//opening file in read mode using BufferedReader stream**

BufferedReader br=new BufferedReader(new FileReader("d:\\myfile.txt"));

while((str=br.readLine())!=null)

{

System.out.println(str);

}

br.close(); **//closing BufferedReader stream**

}catch(IOException ie)

{ System.out.println("exception"); }

}

}

#### Example using try with Resource Statement

import java.io.\*;

class Test

{

public static void main(String[] args)

{

**try(BufferedReader br=new BufferedReader(new FileReader("d:\\myfile.txt")))**

{

String str;

while((str=br.readLine())!=null)

{

System.out.println(str);

}

}catch(IOException ie)

{ System.out.println("exception"); }

}

}

**NOTE:** In the above example, we do not need to explicitly call close() method to close **BufferedReader stream.**

### throw Keyword

throw keyword is used to throw an exception explicitly. Only object of Throwable class or its sub classes can be thrown. Program execution stops on encountering **throw** statement, and the closest catch statement is checked for matching type of exception.

**Syntax :**

**throw** *ThrowableInstance*

#### Creating Instance of Throwable class

There are two possible ways to get an instance of class Throwable,

1. Using a parameter in catch block.
2. Creating instance with **new** operator.
3. **new** NullPointerException("test");

This constructs an instance of NullPointerException with name test.

#### Example demonstrating throw Keyword

class Test

{

static void avg()

{

try

{

throw new ArithmeticException("demo");

}

catch(ArithmeticException e)

{

System.out.println("Exception caught");

}

}

public static void main(String args[])

{

avg();

}

}

In the above example the avg() method throw an instance of ArithmeticException, which is successfully handled using the catch statement.

### throws Keyword

Any method capable of causing exceptions must list all the exceptions possible during its execution, so that anyone calling that method gets a prior knowledge about which exceptions to handle. A method can do so by using the **throws** keyword.

**Syntax :**

*type method\_name(parameter\_list)* **throws** *exception\_list*

{

//definition of method

}

**NOTE :** It is necessary for all exceptions, except the exceptions of type **Error** and **RuntimeException**, or any of their subclass.

#### Example demonstrating throws Keyword

class Test

{

static void check() throws ArithmeticException

{

System.out.println("Inside check function");

throw new ArithmeticException("demo");

}

public static void main(String args[])

{

try

{

check();

}

catch(ArithmeticException e)

{

System.out.println("caught" + e);

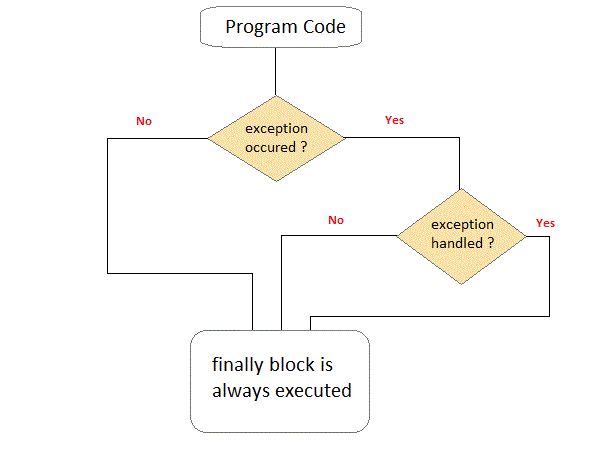
}

}

}

### finally clause

A finally keyword is used to create a block of code that follows a try block. A finally block of code always executes whether or not exception has occurred. Using a finally block, lets you run any cleanup type statements that you want to execute, no matter what happens in the protected code. A finally block appears at the end of catch block.



#### Example demonstrating finally Clause

Class ExceptionTest

{

public static void main(String[] args)

{

int a[]= new int[2];

System.out.println("out of try");

try

{

System.out.println("Access invalid element"+ a[3]);

**/\* the above statement will throw ArrayIndexOutOfBoundException \*/**

}

finally

{

System.out.println("finally is always executed.");

}

}

}

**Output :**

Out of try

finally is always executed.

Exception in thread main java. Lang. exception array Index out of bound exception.

You can see in above example even if exception is thrown by the program, which is not handled by catch block, still finally block will get executed.

### User defined Exception subclass

You can also create your own exception sub class simply by extending java **Exception** class. You can define a constructor for your Exception sub class (not compulsory) and you can override the **toString()** function to display your customized message on catch.

class MyException extends Exception

{

private int ex;

MyException(int a)

{

ex=a;

}

public String toString()

{

return "MyException[" + ex +"] is less than zero";

}

}

class Test

{

static void sum(int a,int b) throws MyException

{

if(a<0)

{

throw new MyException(a);

}

else

{

System.out.println(a+b);

}

}

public static void main(String[] args)

{

try

{

sum(-10, 10);

}

catch(MyException me)

{

System.out.println(me);

}

}

}

#### Points to Remember

1. Extend the Exception class to create your own ecxeption class.
2. You don't have to implement anything inside it, no methods are required.
3. You can have a Constructor if you want.
4. You can override the toString() function, to display customized message.

### Method Overriding with Exception Handling

There are few things to remember when overriding a method with exception handling. If super class method does not declare any exception, then sub class overriden method cannot declare checked exception but it can declare unchecked exceptions.

#### Example of Subclass overriden Method declaring Checked Exception

import java.io.\*;

class Super

{

void show() { System.out.println("parent class"); }

}

public class Sub extends Super

{

void show() throws IOException //Compile time error

{ System.out.println("parent class"); }

public static void main( String[] args )

{

Super s=new Sub();

s.show();

}

}

As the method **show()** doesn't throws any exception while in Super class, hence its overriden version can also not throw any checked exception.

#### Example of Subclass overriden Method declaring Unchecked Exception

import java.io.\*;

class Super

{

void show(){ System.out.println("parent class"); }

}

public class Sub extends Super

{

void show() throws ArrayIndexOutOfBoundsException //Correct

{ System.out.println("child class"); }

public static void main(String[] args)

{

Super s=new Sub();

s.show();

}

}

Output : child class

Because *ArrayIndexOutOfBoundsException* is an unchecked exception hence, overrided **show()** method can throw it.

#### More about Overriden Methods and Exceptions

If Super class method throws an exception, then Subclass overriden method can throw the same exception or no exception, but must not throw parent exception of the exception thrown by Super class method.

It means, if Super class method throws object of **NullPointerException** class, then Subclass method can either throw same exception, or can throw no exception, but it can never throw object of **Exception** class (parent of NullPointerException class).

#### Example of Subclass overriden method with same Exception

import java.io.\*;

class Super

{

void show() throws Exception

{ System.out.println("parent class"); }

}

public class Sub extends Super {

void show() throws Exception //Correct

{ System.out.println("child class"); }

public static void main(String[] args)

{

try {

Super s=new Sub();

s.show();

}

catch(Exception e){}

}

}

Output : child class

#### Example of Subclass overriden method with no Exception

import java.io.\*;

class Super

{

void show() throws Exception

{ System.out.println("parent class"); }

}

public class Sub extends Super {

void show() //Correct

{ System.out.println("child class"); }

public static void main(String[] args)

{

try {

Super s=new Sub();

s.show();

}

catch(Exception e){}

}

}

Output : child class

#### Example of Subclass overriden method with parent Exception

import java.io.\*;

class Super

{

void show() throws ArithmeticException

{ System.out.println("parent class"); }

}

public class Sub extends Super {

void show() throws Exception //Cmpile time Error

{ System.out.println("child class"); }

public static void main(String[] args)

{

try {

Super s=new Sub();

s.show();

}

catch(Exception e){}

}

}

### Chained Exception

Chained Exception was added to Java in JDK 1.4. This feature allow you to relate one exception with another exception, i.e one exception describes cause of another exception. For example, consider a situation in which a method throws an **ArithmeticException** because of an attempt to divide by zero but the actual cause of exception was an I/O error which caused the divisor to be zero. The method will throw only **ArithmeticException** to the caller. So the caller would not come to know about the actual cause of exception. Chained Exception is used in such type of situations.

Two new constructors and two methods were added to **Throwable** class to support chained exception.

1. **Throwable**( *Throwable cause* )
2. **Throwable**( *String str*, *Throwable cause* )

In the first form, the paramter **cause** specifies the actual cause of exception. In the second form, it allows us to add an exception description in string form with the actual cause of exception.

**getCause()** and **initCause()** are the two methods added to **Throwable** class.

* **getCause()** method returns the actual cause associated with current exception.
* **initCause()** set an underlying cause(exception) with invoking exception.

#### Example

import java.io.IOException;

public class ChainedException

{

public static void divide(int a, int b)

{

if(b==0)

{

ArithmeticException ae = new ArithmeticException("top layer");

ae.initCause( new IOException("cause") );

throw ae;

}

else

{

System.out.println(a/b);

}

}

public static void main(String[] args)

{

try {

divide(5, 0);

}

catch(ArithmeticException ae) {

System.out.println( "caught : " +ae);

System.out.println("actual cause: "+ae.getCause());

}

}

}

**Output :**

caught:java.lang.ArithmeticException: top layer

actual cause: java.io.IOException: cause

# Multithreading in Java

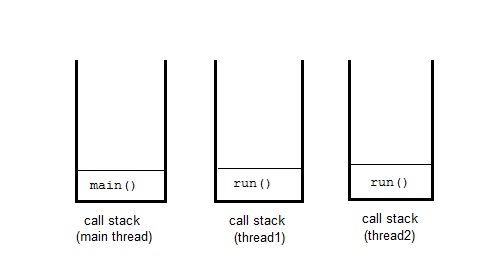
### Introduction to Multithreading

A program can be divided into a number of small processes. Each small process can be addressed as a single thread (a lightweight process). Multithreaded programs contain two or more threads that can run concurrently. This means that a single program can perform two or more tasks simultaneously. For example, one thread is writing content on a file at the same time another thread is performing spelling check.

In Java, the word **thread** means two different things.

* An instance of **Thread** class.
* or, A thread of execution.

An instance of **Thread** class is just an object, like any other object in java. But a thread of execution means an individual "lightweight" process that has its own call stack. In java each thread has its own call stack.



#### The main thread

Even if you don't create any thread in your program, a thread called **main** thread is still created. Although the **main** thread is automatically created, you can control it by obtaining a reference to it by calling **currentThread()** method.

Two important things to know about **main** thread are,

* It is the thread from which other threads will be produced.
* **main** thread must be always the last thread to finish execution.

class MainThread

{

public static void main(String[] args)

{

Thread t=Thread.currentThread();

t.setName("MainThread");

System.out.println("Name of thread is "+t);

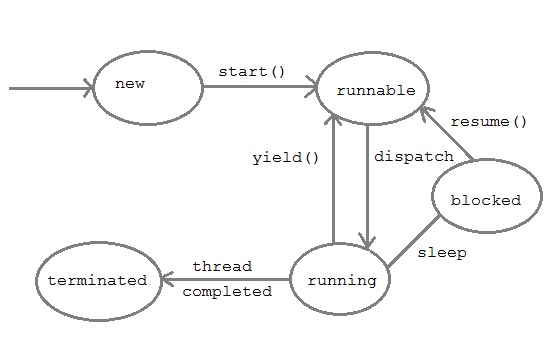
}

}

**Output :**

Name of thread is Thread[MainThread,5,main]

#### Life cycle of a Thread



1. **New :** A thread begins its life cycle in the new state. It remains in this state until the start() method is called on it.
2. **Runable :** After invocation of start() method on new thread, the thread becomes runable.
3. **Running :** A method is in running thread if the thread scheduler has selected it.
4. **Waiting :** A thread is waiting for another thread to perform a task. In this stage the thread is still alive.
5. **Terminated :** A thread enter the terminated state when it complete its task.

#### Thread Priorities

Every thread has a priority that helps the operating system determine the order in which threads are scheduled for execution. In java thread priority ranges between,

* MIN-PRIORITY (a constant of 1)
* MAX-PRIORITY (a constant of 10)

By default every thread is given a NORM-PRIORITY(5). The **main** thread always have NORM-PRIORITY.

### Thread Class

Thread class is the main class on which Java's Multithreading system is based. Thread class, along with its companion interface **Runnable** will be used to create and run threads for utilizing Multithreading feature of Java.

#### Constructors of Thread class

1. **Thread** ( )
2. **Thread** ( *String str* )
3. **Thread** ( *Runnable r* )
4. **Thread** ( *Runnable r*, *String str*)

You can create new thread, either by extending Thread class or by implementing Runnable interface. Thread class also defines many methods for managing threads. Some of them are,

| **Method** | **Description** |
| --- | --- |
| setName() | to give thread a name |
| getName() | return thread's name |
| getPriority() | return thread's priority |
| isAlive() | checks if thread is still running or not |
| join() | Wait for a thread to end |
| run() | Entry point for a thread |
| sleep() | suspend thread for a specified time |
| start() | start a thread by calling run() method |

#### Some Important points to Remember

1. When we extend Thread class, we cannot override **setName()** and **getName()** functions, because they are declared final in Thread class.
2. While using **sleep()**, always handle the exception it throws.

*static* void **sleep**(long *milliseconds*) throws **InterruptedException**

### Creating a thread

Java defines two ways by which a thread can be created.

* By implementing the **Runnable** interface.
* By extending the **Thread** class.

#### Implementing the Runnable Interface

The easiest way to create a thread is to create a class that implements the runnable interface. After implementing runnable interface , the class needs to implement the **run()** method, which is of form,

public *void* **run**()

* run() method introduces a concurrent thread into your program. This thread will end when run() returns.
* You must specify the code for your thread inside run() method.
* run() method can call other methods, can use other classes and declare variables just like any other normal method.

class MyThread **implements Runnable**

{

public void **run()**

{

System.out.println("concurrent thread started running..");

}

}

class MyThreadDemo

{

public static void main( String args[] )

{

MyThread mt = new MyThread();

Thread t = new Thread(mt);

**t.start()**;

}

}

**Output :**

concurrent thread started running..

To call the **run()** method, **start()** method is used. On calling start(), a new stack is provided to the thread and run() method is called to introduce the new thread into the program.

#### Extending Thread class

This is another way to create a thread by a new class that extends **Thread** class and create an instance of that class. The extending class must override **run()** method which is the entry point of new thread.

class MyThread **extends Thread**

{

public void **run()**

{

System.out.println("Concurrent thread started running..");

}

}

classMyThreadDemo

{

public static void main( String args[] )

{

MyThread mt = new MyThread();

**mt.start();**

}

}

**Output :**

concurrent thread started running..

In this case also, as we must override the **run()** and then use the **start()** method to start and run the thread. Also, when you create MyThread class object, Thread class constructor will also be invoked, as it is the super class, hence MyThread class object acts as Thread class object.

#### What if we call run() method directly without using start() method ?

In above program if we directly call **run()** method, without using **start()** method,

public static void main( String args[] )

{

MyThread mt = new MyThread();

**mt.run()**;

}

Doing so, the thread won't be allocated a new call stack, and it will start running in the current call stack, that is the call stack of the **main** thread. Hence Multithreading won't be there.



#### Can we Start a thread twice ?

No, a thread cannot be started twice. If you try to do so, **IllegalThreadStateException** will be thrown.

public static void main( String args[] )

{

MyThread mt = new MyThread();

**mt.start()**;

**mt.start()**; //Exception thrown

}

When a thread is in running state, and you try to start it again, or any method try to invoke that thread again using **start()** method, exception is thrown.

### Joining threads

Sometimes one thread needs to know when another thread is ending. In java, **isAlive()** and **join()** are two different methods to check whether a thread has finished its execution.

The **isAlive()** method returns **true** if the thread upon which it is called is still running otherwise it returns **false**.

*final* boolean **isAlive()**

But, **join()** method is used more commonly than **isAlive()**. This method waits until the thread on which it is called terminates.

*final* void **join()** throws **InterruptedException**

Using **join()** method, we tell our thread to wait until the specified thread completes its execution. There are overloaded versions of **join()** method, which allows us to specify time for which you want to wait for the specified thread to terminate.

*final* void **join**(long *milliseconds*) throws **InterruptedException**

#### Example of isAlive method

public class **MyThread** extends **Thread**

{

public void run()

{

System.out.println("r1 ");

try {

Thread.sleep(500);

}

catch(InterruptedException ie) { }

System.out.println("r2 ");

}

public static void main(String[] args)

{

MyThread t1=new MyThread();

MyThread t2=new MyThread();

t1.start();

t2.start();

System.out.println(**t1.isAlive()**);

System.out.println(**t2.isAlive()**);

}

}

**Output :**

r1

true

true

r1

r2

r2

#### Example of thread without join() method

public class **MyThread** extends **Thread**

{

public void run()

{

System.out.println("r1 ");

try {

Thread.sleep(500);

}

catch(InterruptedException ie){ }

System.out.println("r2 ");

}

public static void main(String[] args)

{

MyThread t1=new MyThread();

MyThread t2=new MyThread();

t1.start();

t2.start();

}

}

**Output :**

r1

r1

r2

r2

In this above program two thread t1 and t2 are created. t1 starts first and after printing "r1" on console thread t1 goes to sleep for 500 ms. At the same time Thread t2 will start its process and print "r1" on console and then go into sleep for 500 ms. Thread t1 will wake up from sleep and print "r2" on console similarly thread t2 will wake up from sleep and print "r2" on console. So you will get output like r1 r1 r2 r2

#### Example of thread with join() method

public class **MyThread** extends **Thread**

{

public void run()

{

System.out.println("r1 ");

try {

Thread.sleep(500);

}catch(InterruptedException ie){ }

System.out.println("r2 ");

}

public static void main(String[] args)

{

MyThread t1=new MyThread();

MyThread t2=new MyThread();

**t1.start();**

try{

**t1.join();** *//Waiting for t1 to finish*

}catch(InterruptedException ie){}

**t2.start();**

}

}

**Output :**

r1

r2

r1

r2

In this above program join() method on thread t1 ensures that t1 finishes it process before thread t2 starts.

#### Specifying time with join()

If in the above program, we specify time while using **join()** with **t1**, then **t1** will execute for that time, and then **t2** will join it.

**t1.join(1500);**

Doing so, initially t1 will execute for 1.5 seconds, after which t2 will join it.

### Synchronization

At times when more than one thread try to access a shared resource, we need to ensure that resource will be used by only one thread at a time. The process by which this is achieved is called **synchronization**. The synchronization keyword in java creates a block of code referred to as critical section.

Every Java object with a critical section of code gets a lock associated with the object. To enter critical section a thread need to obtain the corresponding object's lock.

**General Syntax :**

synchronized (object)

{

//statement to be synchronized

}

#### Why we use Syncronization ?

If we do not use syncronization, and let two or more threads access a shared resource at the same time, it will lead to distorted results.

Consider an example, Suppose we have two different threads **T1** and **T2**, T1 starts execution and save certain values in a file *temporary.txt* which will be used to calculate some result when T1 returns. Meanwhile, T2 starts and before T1 returns, T2 change the values saved by T1 in the file temporary.txt (temporary.txt is the shared resource). Now obviously T1 will return wrong result.

To prevent such problems, synchronization was introduced. With synchronization in above case, once T1 starts using *temporary.txt* file, this file will be **locked**(LOCK mode), and no other thread will be able to access or modify it until T1 returns.

#### Using Synchronized Methods

Using Synchronized methods is a way to accomplish synchronization. But lets first see what happens when we do not use synchronization in our program.

#### Example with no Synchronization

class First

{

public void display(String msg)

{

System.out.print ("["+msg);

try

{

Thread.sleep(1000);

}

catch(InterruptedException e)

{

e.printStackTrace();

}

System.out.println ("]");

}

}

class Second extends Thread

{

String msg;

First fobj;

Second (First fp,String str)

{

fobj = fp;

msg = str;

start();

}

public void run()

{

fobj.display(msg);

}

}

public class Syncro

{

public static void main (String[] args)

{

*First* ***fnew*** *= new First();*

Second ss = new second(**fnew**, "welcome");

Second ss1= new second (**fnew**,"new");

Second ss2 = new second(**fnew**, "programmer");

}

}

**Output :**

[welcome [ new [ programmer]

]

]

In the above program, object **fnew** of class First is shared by all the three running threads(ss, ss1 and ss2) to call the shared method(*void* **display**). Hence the result is unsynchronized and such situation is called **Race condition**.

#### Synchronized Keyword

To synchronize above program, we must *serialize* access to the shared **display()** method, making it available to only one thread at a time. This is done by using keyword **synchronized** with display() method.

**synchronized** *void* **display** (String msg)

#### Using Synchronised block

If you have to synchronize access to object of a class that has no synchronized methods, and you cannot modify the code. You can use synchronized block to use it.

class First

{

public void display(String msg)

{

System.out.print ("["+msg);

try

{

Thread.sleep(1000);

}

catch(InterruptedException e)

{

e.printStackTrace();

}

System.out.println ("]");

}

}

class Second extends Thread

{

String msg;

First fobj;

Second (First fp,String str)

{

fobj = fp;

msg = str;

start();

}

public void run()

{

**synchronized(fobj)** //Synchronized block

{

*fobj.display(msg)*;

}

}

}

public class Syncro

{

public static void main (String[] args)

{

First fnew = new First();

Second ss = new second(fnew, "welcome");

Second ss1= new second (fnew,"new");

Second ss2 = new second(fnew, "programmer");

}

}

**Output :**

[welcome]

[new]

[programmer]

Because of synchronized block this program gives the expected output.

### Interthread Communication

Java provide benefit of avoiding thread pooling using interthread communication. The **wait()**, **notify()**, **notifyAll()** of Object class. These method are implemented as **final** in Object. All three method can be called only from within a **synchronized** context.

* **wait()** tells calling thread to give up monitor and go to sleep until some other thread enters the same monitor and call notify.
* **notify()** wakes up a thread that called wait() on same object.
* **notifyAll()** wakes up all the thread that called wait() on same object.

#### Difference between wait() and sleep()

|  |  |
| --- | --- |
| **wait()** | **sleep()** |
| called from synchronised block | no such requirement |
| monitor is released | monitor is not released |
| awake when notify() or notifyAll() method is called. | not awake when notify() or notifyAll() method is called |
| not a static method | static method |
| wait() is generaly used on condition | sleep() method is simply used to put your thread on sleep. |

#### Thread Pooling

Pooling is usually implemented by loop i.e to check some condition repeatedly. Once condition is true appropriate action is taken. This waste CPU time.

#### Deadlock



Deadlock is a situation of complete Lock, when no thread can complete its execution because lack of resources. In the above picture, Thread 1 is holding a resource R1, and need another resource R2 to finish execution, but R2 is locked by Thread 2, which needs R3, which in turn is locked by Thread 3. Hence none of them can finish and are stuck in a deadlock.

#### Example of deadlock

class **Pen**{}

class **Paper**{}

public class **Write** {

public static void main(String[] args)

{

final Pen pn =new Pen();

final Paper pr =new Paper();

Thread t1 = new Thread(){

public void run()

{

**synchronized(pn)**

{

System.out.println("Thread1 is holding Pen");

try{

Thread.sleep(1000);

}catch(InterruptedException e){}

synchronized(pr)

{ System.out.println("Requesting for Paper"); }

}

}

};

Thread t2 = new Thread(){

public void run()

{

**synchronized(pr)**

{

System.out.println("Thread2 is holding Paper");

try{

Thread.sleep(1000);

}catch(InterruptedException e){}

synchronized(pn)

{ System.out.println("requesting for Pen"); }

}

}

};

t1.start();

t2.start();

}

}

**Output :**

Thread1 is holding Pen

Thread2 is holding Paper

# Advanced topics

### Enumerations

Enumerations was added to Java language in JDK5. **Enumeration** means a list of named constant. In Java, enumeration defines a class type. An Enumeration can have constructors, methods and instance variables. It is created using **enum** keyword. Each enumeration constant is *public*, *static* and *final* by default. Even though enumeration defines a class type and have constructors, you do not instantiate an **enum** using **new**. Enumeration variables are used and declared in much a same way as you do a primitive variable.

#### How to Define and Use an Enumeration

1. An enumeration can be defined simply by creating a list of enum variable. Let us take an example for list of Subject variable, with different subjects in the list.
2. enum **Subject** //Enumeration defined
3. {
4. *Java*, *Cpp*, *C*, *Dbms*
5. }
6. Identifiers Java, Cpp, C and Dbms are called **enumeration constants**. These are public, static final by default.
7. Variables of Enumeration can be defined directly without any **new** keyword.
8. *Subject* **sub**
9. Variables of Enumeration type can have only enumeration constants as value.
10. sub = Subject.Java;

#### Example of Enumeration

enum WeekDays

{ sun, mon, tues, wed, thurs, fri, sat }

class Test

{

public static void main(String args[])

{

WeekDays **wk**;

wk = WeekDays.sun;

System.out.println("Today is "+wk);

}

}

**Output :**

Today is sun

#### Values( ) and ValueOf( ) method

All the enumerations have **values()** and **valueOf()** methods in them. values() method returns an array of enum-type containing all the enumeration constants in it. Its general form is,

public **static** *enum-type[ ]* **values()**

valueOf() method is used to return the enumeration constant whose value is equal to the string passed in as argument while calling this method. It's general form is,

public **static** *enum-type* **valueOf** (String *str*)

#### Points to remember about Enumerations

1. Enumerations are of class type, and have all the capabilities that a Java class has.
2. Enumerations can have Constructors, instance Variables, methods and can even implement Interfaces.
3. Enumerations are not instantiated using **new** keyword.
4. All Enumerations by default inherit **java.lang.Enum** class.

#### Enumeration with Constructor, instance variable and Method

enum Student

{

John(11), Bella(10), Sam(13), Viraaj(9);

private int age; //age of students

int getage { return age; }

public Student(int age)

{

this.age= age;

}

}

class EnumDemo

{

public static void main( String args[] )

{

Student S;

System.out.println("Age of Viraaj is " +**Student.Viraaj.getage()**+ "years");

}

}

**Output :**

Age of Viraaj is 9 years

In this example as soon as we declare an enum variable(*Student S*), the constructor is called once, and it initializes age for every enumeration constant with values specified with them in parenthesis.

### type wrapper

Java uses primitive types such as int, double or float to hold the basic data types for the sake of performance. Despite the performance benefits offered by the primitive types, there are situation when you will need an object representation. For example, many data structures in Java operate on objects, so you cannot use primitive types with those data structures. To handle these situations Java provides **type Wrappers** which provide classes that encapsulate a primitive type within an object.

* **Character :** It encapsulates primitive type char within object.
* **Character** (char *ch*)
* **Boolean :** It encapsulates primitive type boolean within object.
* **Boolean** (boolean *boolValue*)
* **Numeric type wrappers :** It is the most commonly used type wrapper.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Byte** | **Short** | **Integer** | **Long** | **Float** | **Double** |

* Above mentioned Classes comes under Numeric type wrapper. These classes encapsulate byte, short, int, long, float, double primitive type.

#### Autoboxing and Unboxing

* Autoboxing and Unboxing features was added in Java5.
* **Autoboxing** is a process by which primitive type is automatically encapsulated(boxed) into its equivalent type wrapper
* **Auto-Unboxing** is a process by which the value of object is automatically extracted from a type wrapper.

#### Example of Autoboxing and Unboxing

class Test

{

public static void main(String[] args)

{

Integer iob = 100; //Autoboxing of int

int i = iob; //unboxing of Integer

System.out.println(i+" "+iob);

Character cob = 'a'; /Autoboxing of char

char ch = cob; //Auto-unboxing of Character

System.out.println(cob+" "+ch);

}

}

**Output :**

100 100

a a

#### Autoboxing / Unboxing in Expressions

Whenever we use object of Wrapper class in an expression, automatic unboxing and boxing is done by JVM.

Integer iOb;

iOb = 100; //Autoboxing of int

**++iOb**;

When we perform increment operation on Integer object, it is first unboxed, then incremented and then again reboxed into Integer type object.

This will happen always, when we will use Wrapper class objects in expressions or conditions etc.

#### Benefits of Autoboxing / Unboxing

1. Autoboxing / Unboxing lets us use primitive types and Wrapper class objects interchangeably.
2. We don't have to perform Explicit **typecasting**.
3. It helps prevent errors, but may lead to unexpected results sometimes. Hence must be used with care.

### IO Stream

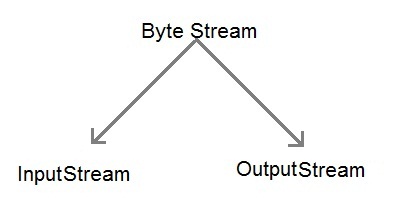
Java performs I/O through **Streams**. A Stream is linked to a physical layer by java I/O system to make input and output operation in java. In general, a stream means continuous flow of data. Streams are clean way to deal with input/output without having every part of your code understand the physical.

Java encapsulates Stream under **java.io** package. Java defines two types of streams. They are,

1. **Byte Stream :** It provides a convenient means for handling input and output of byte.
2. **Character Stream :** It provides a convenient means for handling input and output of characters. Character stream uses Unicode and therefore can be internationalized.

#### Byte Stream Classes

Byte stream is defined by using two abstract class at the top of hierarchy, they are InputStream and OutputStream.



These two abstract classes have several concrete classes that handle various devices such as disk files, network connection etc.

#### Some important Byte stream classes.

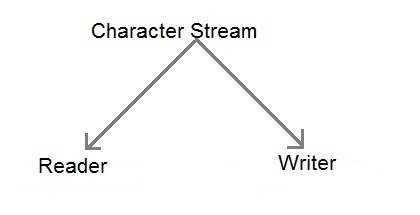
|  |  |
| --- | --- |
| **Stream class** | **Description** |
| **BufferedInputStream** | Used for Buffered Input Stream. |
| **BufferedOutputStream** | Used for Buffered Output Stream. |
| **DataInputStream** | Contains method for reading java standard datatype |
| **DataOutputStream** | An output stream that contain method for writing java standard data type |
| **FileInputStream** | Input stream that reads from a file |
| **FileOutputStream** | Output stream that write to a file. |
| **InputStream** | Abstract class that describe stream input. |
| **OutputStream** | Abstract class that describe stream output. |
| **PrintStream** | Output Stream that contain print() and println() method |

These classes define several key methods. Two most important are

1. read() : reads byte of data.
2. write() : Writes byte of data.

#### Character Stream Classes

Character stream is also defined by using two abstract class at the top of hierarchy, they are Reader and Writer.



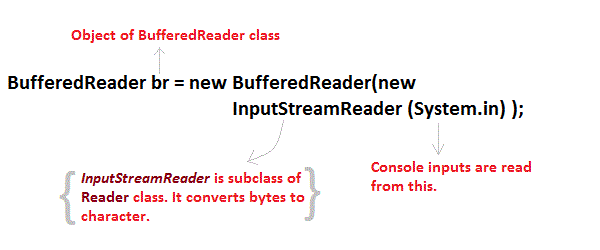
These two abstract classes have several concrete classes that handle unicode character.

#### Some important Charcter stream classes.

|  |  |
| --- | --- |
| **Stream class** | **Description** |
| **BufferedReader** | Handles buffered input stream. |
| **BufferedWriter** | Handles buffered output stream. |
| **FileReader** | Input stream that reads from file. |
| **FileWriter** | Output stream that writes to file. |
| **InputStreamReader** | Input stream that translate byte to character |
| **OutputStreamReader** | Output stream that translate character to byte. |
| **PrintWriter** | Output Stream that contain print() and println() method. |
| **Reader** | Abstract class that define character stream input |
| **Writer** | Abstract class that define character stream output |

#### Reading Console Input

We use the object of BufferedReader class to take inputs from the keyboard.



#### Reading Characters

read() method is used with BufferedReader object to read characters. As this function returns integer type value has we need to use typecasting to convert it into **char** type.

*int* **read()** throws **IOException**

Below is a simple example explaining character input.

class CharRead

{

public static void main( String args[])

{

BufferedReader br = new Bufferedreader(new InputstreamReader(System.in));

char c = **(char)br.read()**; //Reading character

}

}

#### Reading Strings

To read string we have to use readLine() function with BufferedReader class's object.

*String* **readLine()** throws **IOException**

#### Program to take String input from Keyboard in Java

import java.io.\*;

class MyInput

{

public static void main(String[] args)

{

String text;

**InputStreamReader** isr = new **InputStreamReader(System.in)**;

**BufferedReader** br = new **BufferedReader(isr)**;

text = br.readLine(); //Reading String

System.out.println(text);

}

}

#### Program to read from a file using BufferedReader class

import java. Io \*;

class ReadTest

{

public static void main(String[] args)

{

try

{

File fl = new File("d:/myfile.txt");

BufferedReader br = new BufferedReader(new FileReader(fl)) ;

String str;

while ((str=br.readLine())!=null)

{

System.out.println(str);

}

br.close();

fl.close();

}

catch (IOException e)

{ e.printStackTrace(); }

}

}

#### Program to write to a File using FileWriter class

import java. Io \*;

class WriteTest

{

public static void main(String[] args)

{

try

{

File fl = new File("d:/myfile.txt");

String str="Write this string to my file";

FileWriter fw = new FileWriter(fl) ;

fw.write(str);

fw.close();

fl.close();

}

catch (IOException e)

{ e.printStackTrace(); }

}

}

### Serialization and Deserialization in Java

**Serialization** is a process of converting an object into a sequence of bytes which can be persisted to a disk or database or can be sent through streams. The reverse process of creating object from sequence of bytes is called **deserialization**.

A class must implement **Serializable** interface present in **java.io** package in order to serialize its object successfully. **Serializable** is a **marker interface** that adds serializable behaviour to the class implementing it.

Java provides **Serializable** API encapsulated under *java.io* package for serializing and deserializing objects which include,

* *java.io.serializable*
* *java.io.Externalizable*
* *ObjectInputStream*
* and *ObjectOutputStream* etc.

#### Marker interface

Marker Interface is a special interface in Java without any field and method. Marker interface is used to inform compiler that the class implementing it has some special behaviour or meaning. Some example of Marker interface are,

* java.io.Serializable
* java.lang.Cloneable
* java.rmi.Remote
* java.util.RandomAccess

All these interfaces does not have any method and field. They only add special behavior to the classes implementing them. However marker interfaces have been deprecated since Java 5, they were replaced by **Annotations**. Annotations are used in place of Marker Interface that play the exact same role as marker interfaces did before.

#### Signature of writeObject() and readObject()

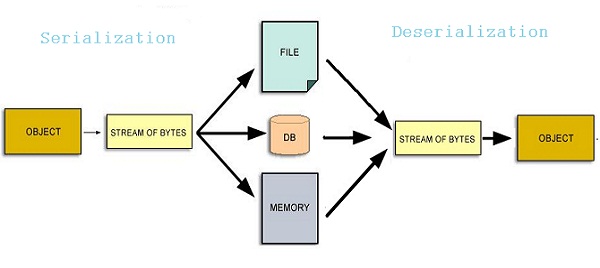
**writeObject()** method of *ObjectOutputStream* class serializes an object and send it to the output stream.

public *final* void **writeObject**(*object x*) throws **IOException**

**readObject()** method of *ObjectInputStream* class references object out of stream and deserialize it.

public final *Object* **readObject()** throws **IOException,ClassNotFoundException**

while serializing if you do not want any field to be part of object state then declare it either static or transient based on your need and it will not be included during java serialization process.



#### Serializing an Object

import java.io.\*;

class studentinfo implements Serializable

{

String name;

int rid;

static String contact;

studentinfo(string n, int r, string c)

{

this.name = n;

this.rid = r;

this.contact = c;

}

}

class Test

{

public static void main(String[] args)

{

try

{

Studentinfo si = new studentinfo("Abhi", 104, "110044");

FileOutputStream fos = new **FileOutputStream("student.ser")**;

Objectoutputstream oos = new ObjectOutputStream(fos);

oos.writeObject(si);

oos.close();

fos.close();

}

catch (Exception e)

{ e. printStackTrace(); }

}

}

Object of Studentinfo class is serialized using writeObject() method and written to **student.ser** file.

#### Deserialization of Object

import java.io \* ;

class DeserializationTest

{

public static void main(String[] args)

{

studentinfo si=null ;

try

{

FileInputStream fis = new FileInputStream("student.ser");

ObjectOutputStream ois = new ObjectOutputStream(fis);

si = (studentinfo)ois.readObject();

}

catch (Exception e)

{ e.printStackTrace(); }

System.out.println(si.name);

System.out. println(si.rid);

System.out.println(si.contact);

}

}

**Output :**

Abhi

104

null

Contact field is null because,it was marked as static and as we have discussed earlier static fields does not get serialized.

**NOTE :** Static members are never serialized because they are connected to class not object of class.

#### transient Keyword

While serializing an object, if we don't want certain data member of the object to be serialized we can mention it transient. transient keyword will prevent that data member from being serialized.

class studentinfo implements Serializable

{

String name;

**transient** int rid;

**static** String contact;

}

* Making a data member **transient** will prevent its serialization.
* In this example rid will not be serialized because it is **transient**, and contact will also remain unserialized because it is **static**.

### Networking in Java

Java is a premier language for network programming. **java.net** package encapsulate large number of classes and interface that provides an easy-to use means to access network resources. Here are some important classes and interfaces of java.net package.

#### Some Important Classes

|  |  |
| --- | --- |
| **CLASSES** | |
| CacheRequest | CookieHandler |
| CookieManager | Datagrampacket |
| Inet Address | ServerSocket |
| Socket | DatagramSocket |
| Proxy | URL |
| URLConnection |  |

#### Some Important Interfaces

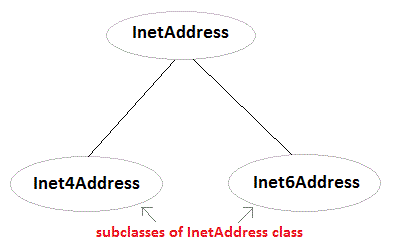
|  |  |
| --- | --- |
| **INTERFACES** | |
| CookiePolicy | CookieStore |
| FileNameMap | SocketOption |
| InetAddress | ServerSocket |
| SocketImplFactory | ProtocolFamily |

#### InetAddress

Inet Address encapsulates both numerical IP address and the domain name for that address. Inet address can handle both IPv4 and Ipv6 addresses. Inet Address class has no visible constructor. To create an inet Address object, you have to use **Factory methods**.

Three commonly used Inet Address factory methods are.

1. static *InetAddress* **getLocalHost()** throws **UnknownHostException**
2. static *InetAddress* **getByName** (*String hostname*) throws **UnknownHostException**
3. static *InetAddress[ ]* **getAllByName** (*String hostname*) throws **UnknownHostException**



#### Example using InetAddress class

import java.net.\*;

class Test

{

public static void main(String[] args)

{

**InetAddress address = InetAddress.getLocalHost()**;

System.out.println(address);

address = **InetAddress.getByName**("www.studytonight.com");

System.out.println(address);

InetAddress sw[] = **InetAddress.getAllByName**("www.google.com");

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

{

System.out.println(sw[i]);

}

}

}

**Output:**

Welcome-PC/59.161.87.227

www.studytonight.com/208.91.198.55

www.google.com/74.125.236.115

www.google.com/74.125.236.116

www.google.com/74.125.236.112

www.google.com/74.125.236.113

www.google.com/74.125.236.114

www.google.com/2404:6800:4009:802:0:0:0:1014

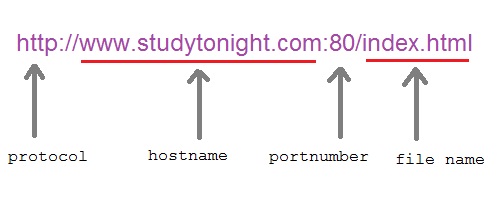
#### Socket and ServerSocket Class

Socket is foundation of modern networking, a socket allows single computer to serve many different clients at once. Socket establishes connection through the use of port, which is a numbered socket on a particular machine. Socket communication takes place via a protocol. Socket provides communication mechanism between two computers using TCP. There are two kind of TCP sockets in Java. One is for server and other is for client.

* **ServerSocket** is for servers.
* **Socket** class is for client.

#### URL class

Java URL Class present in java.net package, deals with URL (Uniform Resource Locator) which uniquely identify or locate resources on internet.



#### Important Methods of URL class

* **getProtocol() :** Returns protocol of URL
* **getHost() :** Returns hostname(domain name) of URL
* **getPort() :** Returns port number of URL
* **getFile() :** Returns filename of URL

#### Program using URL class

import java.net.\*;

class Test

{

public static void main(String[] arg) throws MalFormedURLException

{

**URL hp = New URL("http://www.studytonight.com/index");**

system.out.println(hp.getProtocol[]);

System.out.println(hp.getFile[]);

}

}

**Ouput:**

http

-1

www.studytonight.com

/index

### Generics

A class or interface that operates on parameterized type is called **Generic**. Generics was first introduced in Java5. Now it is one of the most profound feature of java programming language. It provides facility to write algorithm independent of any specific type of data. Generics also provide type safety.

Using **Generics**, it becomes possible to create a single class or method that automatically works with all types of data(Integer, String, Float etc). It expanded the ability to reuse code safely and easily.

#### Example of Generic class

class Gen **<T>**

{

**T** ob; //an object of type T is declared

Gen(T o) //constructor

{

ob = o;

}

public **T** getOb()

{

return ob;

}

}

class Test

{

public static void main (String[] args)

{

Gen **< Integer>** iob = new Gen(100); //instance of Integer type Gen Class.

int x = iob.getOb();

System.out.print in(x);

Gen **< String>** sob = new Gen("Hello"); //instance of String type Gen Class.

String str = sob.getOb();

}

}

**Output :**

100

Hello

#### Generics Work Only with Objects

You cannot use primitive datatype such as **int** or **char** etc with Generics type. It should always be an object.

Gen**< int>** iOb = new Gen**< int>**(07); //Error, can't use primitive type

#### Generics Types of different Type Arguments are never same

Reference of one generic type is never compatible with other generic type unless their type argument is same. In the example above we created two objects of class **Gen**, one of type **Integer**, and other of type **String**, hence,

iob = sob; //Absolutely Wrong

#### Generic Methods

You can also create generic methods that can be called with different types of arguments based on the type of arguments passed to generic method, the compiler handles each method.

#### Example of Generic method

class GenTest

{

static **< V, T>** void display (V v, T t)

{

System.out.println(v.getClass().getName()+" = " +v);

System.out.println(t.getClass().getName()+" = " +t);

}

public static void main(String[] args)

{

display(88,"This is string");

}

}

**Output :**

java lang.Integer = 88

Java lang.String = this is string

#### Generic Constructors

It is possible to create a generic constructor even if the class is not generic.

#### Example of Generic Constructor

class Gen

{

private double val;

**< T extends Number>** Gen(T ob)

{

val=ob.doubleValue();

}

void show()

{

System.out.println(val);

}

}

class Test

{

public static void main(String[] args)

{

Gen g = new Gen(100);

Gen g1 = new Gen(121.5f);

g.show();

g1.show();

}

}

**Output :**

100.0

121.5

#### Generic Interface

Like classes and methods, you can also create generic interfaces.

interface MyInterface**< T >**

{ .. }

#### Generic Bounded type Parameter

You can also set restriction on the type that will be allowed to pass to a type-parameter. This is done with the help of **extends** keyword when specifying the type parameter.

< T extends Number >

Here we have taken **Number** class, it can be any wrapper class name. This specifies that T can be only be replaced by **Number** class data itself or any of its subclass.

#### Generic Method with bounded type Parameters.

class Gen

{

static **< T, V extends number>** void display(T t, V v)

{

System.out.println(v.getClass().getName()+" = " +v);

System.out.println(t.getClass().getName()+" = " +t);

}

public static void main(String[] args)

{

// display(88,"This is string");

display ("this is string",99);

}

}

**Output :**

java.lang.String = This is string

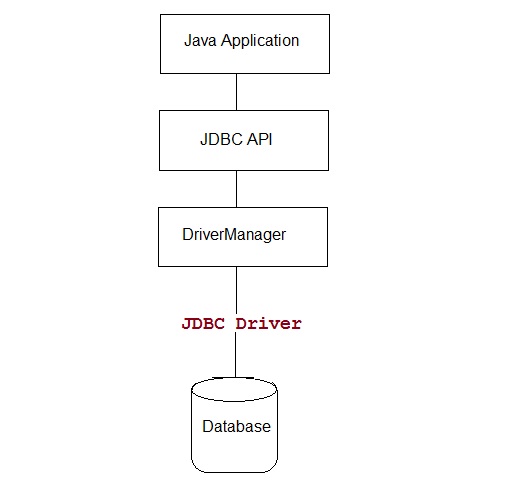
java.lang.Double = 99.O

* Type V is bounded to Number type and its subclass only.
* If display(88,"This is string") is uncommented, it will give an error of type incompatibility, as String is not a subclass of Number class.

# JDBC

### Introduction to JDBC

**Java Database Connectivity(JDBC)** is an **Application Programming Interface(API)** used to connect Java application with Database. JDBC is used to interact with various type of Database such as Oracle, MS Access, My SQL and SQL Server. JDBC can also be defined as the platform-independent interface between a relational database and Java programming. It allows java program to execute SQL statement and retrieve result from database.



#### What's new in JDBC 4.0

**JDBC 4.0** is new and advance specification of JDBC. It provides the following advance features

* Connection Management
* Auto loading of Driver Interface.
* Better exception handling
* Support for large object
* Annotation in SQL query.

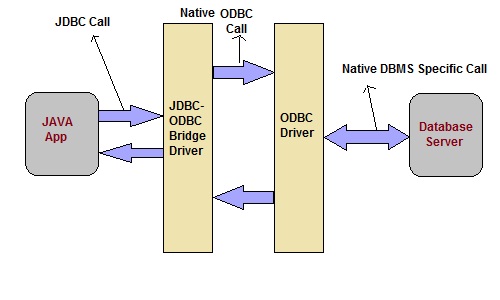
#### JDBC Driver

JDBC Driver is required to process SQL requests and generate result. The following are the different types of driver available in JDBC.

* **Type-1 Driver** or **JDBC-ODBC bridge**
* **Type-2 Driver** or **Native API Partly Java Driver**
* **Type-3 Driver** or **Network Protocol Driver**
* **Type-4 Driver** or **Thin Driver**

#### JDBC-ODBC bridge

**Type-1 Driver** act as a bridge between JDBC and other database connectivity mechanism(ODBC). This driver converts JDBC calls into ODBC calls and redirects the request to the ODBC driver.



**Advantage**

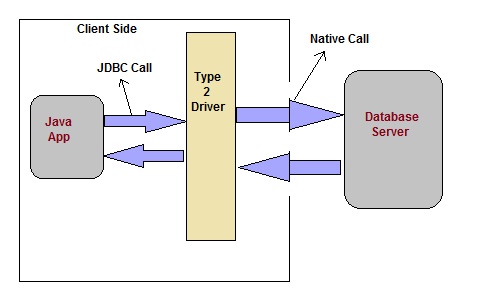
* Easy to use
* Allow easy connectivity to all database supported by the ODBC Driver.

**Disadvantage**

* Slow execution time
* Dependent on ODBC Driver.
* Uses Java Native Interface(JNI) to make ODBC call.

#### Native API Driver

This type of driver make use of Java Native Interface(JNI) call on database specific native client API. These native client API are usually written in C and C++.



**Advantage**

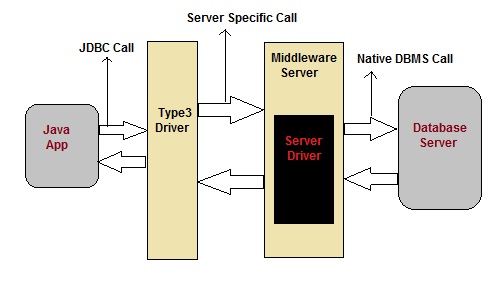
* faster as compared to **Type-1 Driver**
* Contains additional features.

**Disadvantage**

* Requires native library
* Increased cost of Application

#### Network Protocol Driver

This driver translate the JDBC calls into a database server independent and Middleware server-specific calls. Middleware server further translate JDBC calls into database specific calls.



**Advantage**

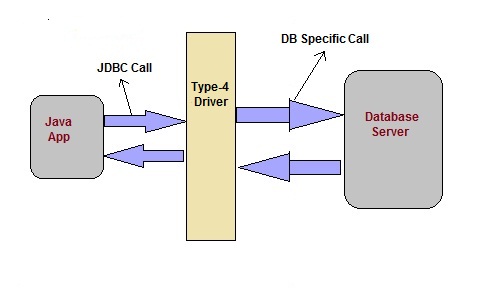
* Does not require any native library to be installed.
* Database Independency.
* Provide facility to switch over from one database to another database.

**Disadvantage**

* Slow due to increase number of network call.

#### Thin Driver

This is Driver called Pure Java Driver because. This driver interact directly with database. It does not require any native database library, that is why it is also known as Thin Driver.



**Advantage**

* Does not require any native library.
* Does not require any Middleware server.
* Better Performance than other driver.

**Disadvantage**

* Slow due to increase number of network call.

### JDBC 4.0 API

JDBC 4.0 API is mainly divided into two package

1. java.sql
2. javax.sql

#### java.sql package

This package include classes and interface to perform almost all JDBC operation such as creating and executing SQL Queries.

#### Important classes and interface of java.sql package

|  |  |
| --- | --- |
| **classes/interface** | **Description** |
| java.sql.BLOB | Provide support for BLOB(Binary Large Object) SQL type. |
| java.sql.Connection | creates a connection with specific database |
| java.sql.CallableStatement | Execute stored procedures |
| java.sql.CLOB | Provide support for CLOB(Character Large Object) SQL type. |
| java.sql.Date | Provide support for Date SQL type. |
| java.sql.Driver | create an instance of a driver with the DriverManager. |
| java.sql.DriverManager | This class manages database drivers. |
| java.sql.PreparedStatement | Used to create and execute parameterized query. |
| java.sql.ResultSet | It is an interface that provide methods to access the result row-by-row. |
| java.sql.Savepoint | Specify savepoint in transaction. |
| java.sql.SQLException | Encapsulate all JDBC related exception. |
| java.sql.Statement | This interface is used to execute SQL statements. |

#### javax.sql package

This package is also known as JDBC extension API. It provides classes and interface to access server-side data.

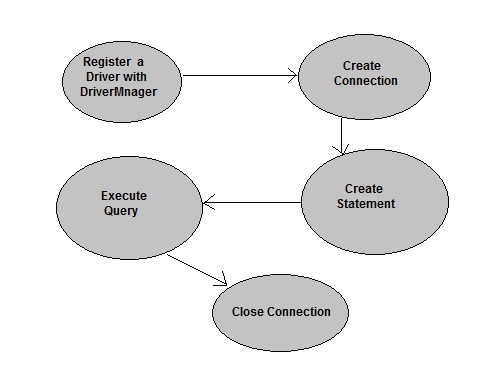
#### Important classes and interface of javax.sql package

|  |  |
| --- | --- |
| **classes/interface** | **Description** |
| javax.sql.ConnectionEvent | Provide information about occurence of event. |
| javax.sql.ConnectionEventListener | Used to register event generated by **PooledConnection** object. |
| javax.sql.DataSource | Represent the **DataSource** interface used in an application. |
| javax.sql.PooledConnection | provide object to manage connection pools. |

### Steps to connect a Java Application to Database

The following 5 steps are the basic steps involve in connecting a Java application with Database using JDBC.

1. Register the Driver
2. Create a Connection
3. Create SQL Statement
4. Execute SQL Statement
5. Closing the connection



#### Register the Driver

Class.forName() is used to load the driver class explicitly.

**Example to register with JDBC-ODBC Driver**

Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");

#### Create a Connection

getConnection() method of **DriverManager** class is used to create a connection.

**Syntax**

getConnection(String url)

getConnection(String url, String username, String password)

getConnection(String url, Properties info)

**Example establish connection with Oracle Driver**

Connection con = DriverManager.getConnection

("jdbc:oracle:thin:@localhost:1521:XE","username","password");

#### Create SQL Statement

createStatement() method is invoked on current **Connection** object to create a SQL Statement.

**Syntax**

public Statement createStatement() throws SQLException

**Example to create a SQL statement**

Statement s=con.createStatement();

#### Execute SQL Statement

executeQuery() method of **Statement** interface is used to execute SQL statements.

**Syntax**

public ResultSet executeQuery(String query) throws SQLException

**Example to execute a SQL statement**

ResultSet rs=s.executeQuery("select \* from user");

while(rs.next())

{

System.out.println(rs.getString(1)+" "+rs.getString(2));

}

#### Closing the connection

After executing SQL statement you need to close the connection and release the session. The close() method of **Connection** interface is used to close the connection.

**Syntax**

public void close() throws SQLException

**Example of closing a connection**

con.close();