Java

Introduction

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

As of December 2008, the latest release of the Java Standard Edition is 6 (J2SE). With the advancement of Java and its widespread popularity, multiple configurations were built to suite various types of platforms. Ex: J2EE for Enterprise Applications, J2ME for Mobile Applications.

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

Features

1. **Object Oriented:** In Java, everything is an Object. Java can be easily extended since it is based on the Object model.

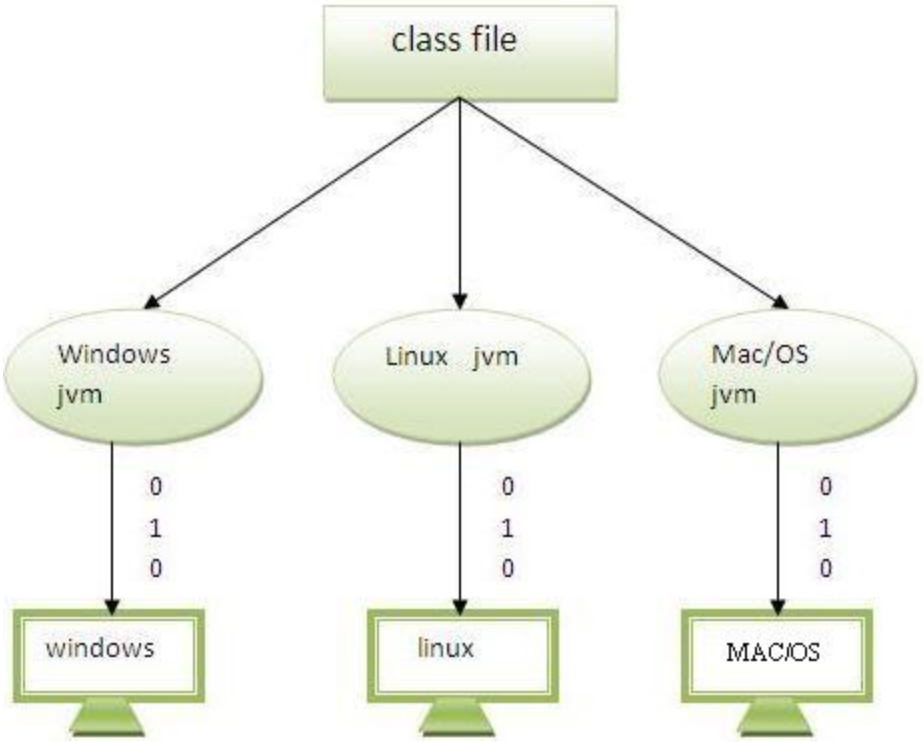
Object-oriented means we organize our software as a combination of different types of objects that incorporates both data and behaviour.

Object-oriented programming(OOPs) is a methodology that simplify software development and maintenance by providing some rules.

Basic concepts of OOPs are:

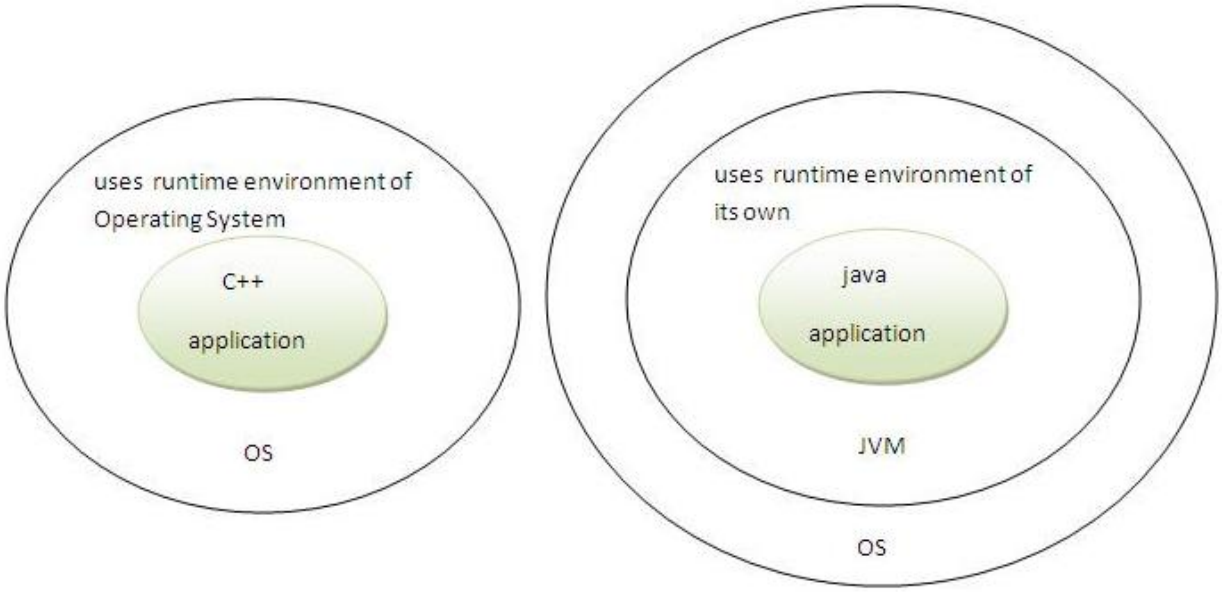
- Object
- Class
- Inheritance
- Polymorphism
- Abstraction
- Encapsulation

2. **Platform independent:** Unlike many other programming languages including C and C++, when Java is compiled, it is not compiled into platform specific machine, rather into platform independent byte code. This byte code is distributed over the web and interpreted by virtual Machine (JVM) on whichever platform it is being run.



3. **Simple:** Java is designed to be easy to learn. If you understand the basic concept of OOP Java would be easy to master.

4. **Secure:** With Java's secure feature it enables to develop virus-free, tamper-free systems. Authentication techniques are based on public-key encryption.



5. **Architectural-neutral:** Java compiler generates an architecture-neutral object file format which makes the compiled code to be executable on many processors, with the presence of Java runtime system.

6. **Portable:** Being architectural-neutral and having no implementation dependent aspects of the specification makes Java portable. Compiler in Java is written in ANSI C with a clean portability boundary which is a POSIX subset.

7. **Robust:** Java makes an effort to eliminate error prone situations by emphasizing mainly on compile time error checking and runtime checking.

8. **Multithreaded:** With Java's multithreaded feature it is possible to write programs that can do many tasks simultaneously. This design feature allows developers to construct smoothly running interactive applications.
9. **Interpreted:** Java byte code is translated on the fly to native machine instructions and is not stored anywhere. The development process is more rapid and analytical since the linking is an incremental and light weight process.

10. **High Performance:** With the use of Just-In-Time compilers, Java enables high performance.

11. **Distributed:** Java is designed for the distributed environment of the internet.

12. **Dynamic:** Java is considered to be more dynamic than C or C++ since it is designed to adapt to an evolving environment. Java programs can carry extensive amount of run-time information that can be used to verify and resolve accesses to objects on run-time.

History of Java:

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

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

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

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

Where it is used?

According to Sun, 3 billion devices run java. There are many devices where java is currently used. Some of them are as follows:

1. Desktop Applications such as acrobat reader, media player, antivirus etc.
2. Web Applications such as irctc.co.in, javatpoint.com etc.
3. Enterprise Applications such as banking applications.
4. Mobile
5. Embedded System
6. Smart Card
7. Robotics
8. Games etc.

Types of Java Applications

There are mainly 4 type of applications that can be created using java programming:

1) Standalone Application

It is also known as desktop application or window-based application. An application that we need to install on every machine such as media player, antivirus etc. AWT and Swing are used in java for creating standalone applications.

2) Web Application

An application that runs on the server side and creates dynamic page, is called web application. Currently, servlet, jsp, struts, jsf etc. technologies are used for creating web applications in java.

3) Enterprise Application

An application that is distributed in nature, such as banking applications etc. It has the advantage of high level security, load balancing and clustering. In java, EJB is used for creating enterprise applications.

4) Mobile Application

An application that is created for mobile devices. Currently Android and Java ME are used for creating mobile applications.

Why Java Name For Java Language?

1) **Why they choosed java name for java language?** The team gathered to choose a new name. The suggested words were "dynamic", "revolutionary", "Silk", "jolt", "DNA" etc. They wanted something that reflected the essence of the technology: revolutionary, dynamic, lively, cool, unique, and easy to spell and fun to say.

According to James Gosling "Java was one of the top choices along with **Silk**". Since java was so unique, most of the team members preferred java.

2) Java is an island of Indonesia where first coffee was produced (called java coffee).

3) Notice that Java is just a name not an acronym.

4) Originally developed by James Gosling at Sun Microsystems (which is now a subsidiary of Oracle Corporation) and released in 1995.

5) In 1995, Time magazine called **Java one of the Ten Best Products of 1995**.

6) JDK 1.0 released in(January 23, 1996).

Java Version History

There are many java versions that has been released. Current stable release of Java is Java SE 8.

1. JDK Alpha and Beta (1995)
2. JDK 1.0 (23rd Jan, 1996) (Originally called **Oak.** The first stable version, JDK 1.0.2, is called Java 1)
3. JDK 1.1 (19th Feb, 1997)
4. J2SE 1.2 (8th Dec, 1998) (Codename **Playground)**
5. J2SE 1.3 (8th May, 2000) (Codename **Kestrel)**
6. J2SE 1.4 (6th Feb, 2002) (Codename **Merlin**)
7. J2SE 5.0 (30th Sep, 2004) (Codename **Tiger**)
8. Java SE 6 (11th Dec, 2006) (Codename **Mustang**)
9. Java SE 7 (28th July, 2011) (Code name **Dolphin**)

10. Java SE 8 (18th March, 2014) (Code name **Lamda**)

Understanding First Java Program

Let's see what is the meaning of class, public, static, void, main, String[], System.out.println().

- **class** keyword is used to declare a class in java.
- **public** keyword is an access modifier which represents visibility, it means it is visible to all.
- **static** is a keyword, if we declare any method as static, it is known as static method. The core advantage of static method is that there is no need to create object to invoke the static method. The main method is executed by the JVM, so it doesn't require to create object to invoke the main method. So it saves memory.
- **void** is the return type of the method, it means it doesn't return any value.
- **main** represents startup of the program.
- **String[] args** is used for command line argument. We will learn it later.
- **System.out.println()** is used print statement.

Java Basic Syntax

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

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

Basic Syntax:

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

- **Case Sensitivity -** Java is case sensitive, which means identifier **Hello** and **hello** would have different meaning in Java.

- **Class Names –** For all class names the first letter should be in Upper Case.

If several words are used to form a name of the class, each inner word's first letter should be in UpperCase.
Example class MyFirstJavaClass

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

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

  Example public void myMethodName()

- **Program File Name -** Name of the program file should exactly match the class name.

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

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

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

Java Identifiers:

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

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

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

Java Modifiers:

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

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

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

Java Variables:

We would see following type of variables in Java:

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

Java Arrays:

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

Java Enums:

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

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

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

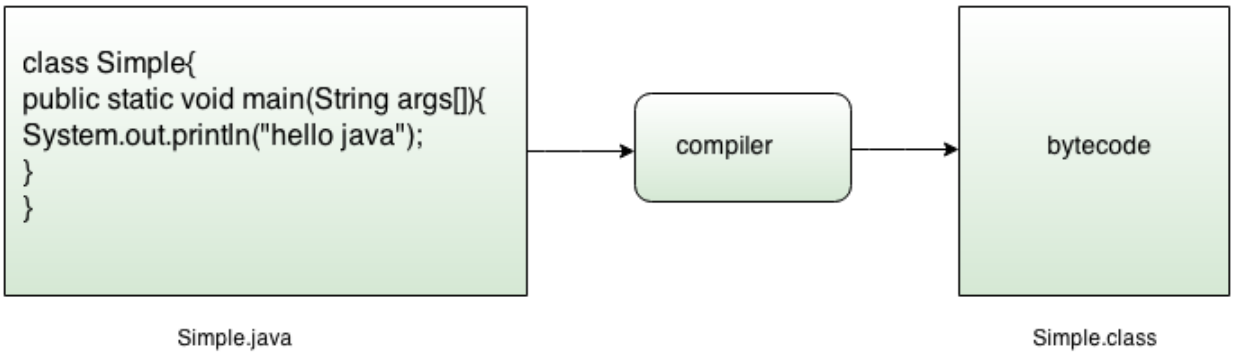
Java Keywords:

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

abstract	assert	boolean	break
byte	case	catch	char
class	const	continue	default
do	double	else	enum
extends	final	finally	float
for	goto	if	implements
import	instanceof	int	interface
long	native	new	package
private	protected	public	return
short	static	strictfp	super
switch	synchronized	this	throw
throws	transient	try	void
volatile	while		

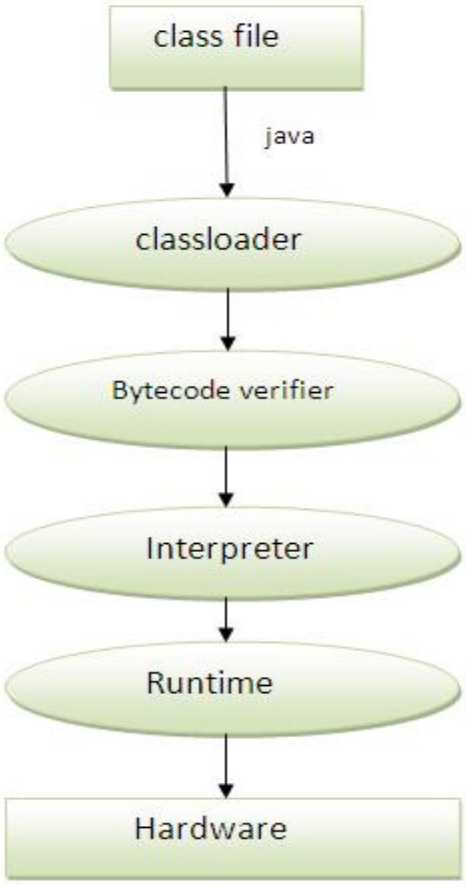
What happens at compile time?

At compile time, java file is compiled by Java Compiler (It does not interact with OS) and converts the java code into bytecode.



What happens at runtime?

At runtime, following steps are performed:



Classloader: is the subsystem of JVM that is used to load class files.

Bytecode Verifier: checks the code fragments for illegal code that can violate access right to objects.

Interpreter: read bytecode stream then execute the instructions.

Difference between JDK, JRE and JVM

JVM

JVM (Java Virtual Machine) is an abstract machine. It is a specification that provides runtime environment in which java bytecode can be executed.

JVMs are available for many hardware and software platforms. JVM, JRE and JDK are platform dependent because configuration of each OS differs. But, Java is platform

independent.

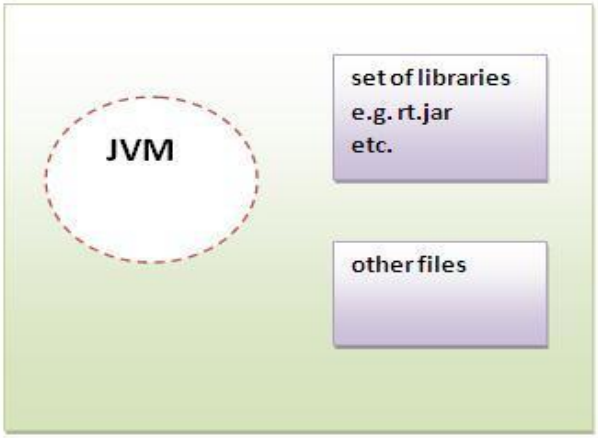
The JVM performs following main tasks:

- Loads code
- Verifies code
- Executes code
- Provides runtime environment

JRE

JRE is an acronym for Java Runtime Environment.It is used to provide runtime environment.It is the implementation of JVM.It physically exists.It contains set of libraries + other files that JVM uses at runtime.

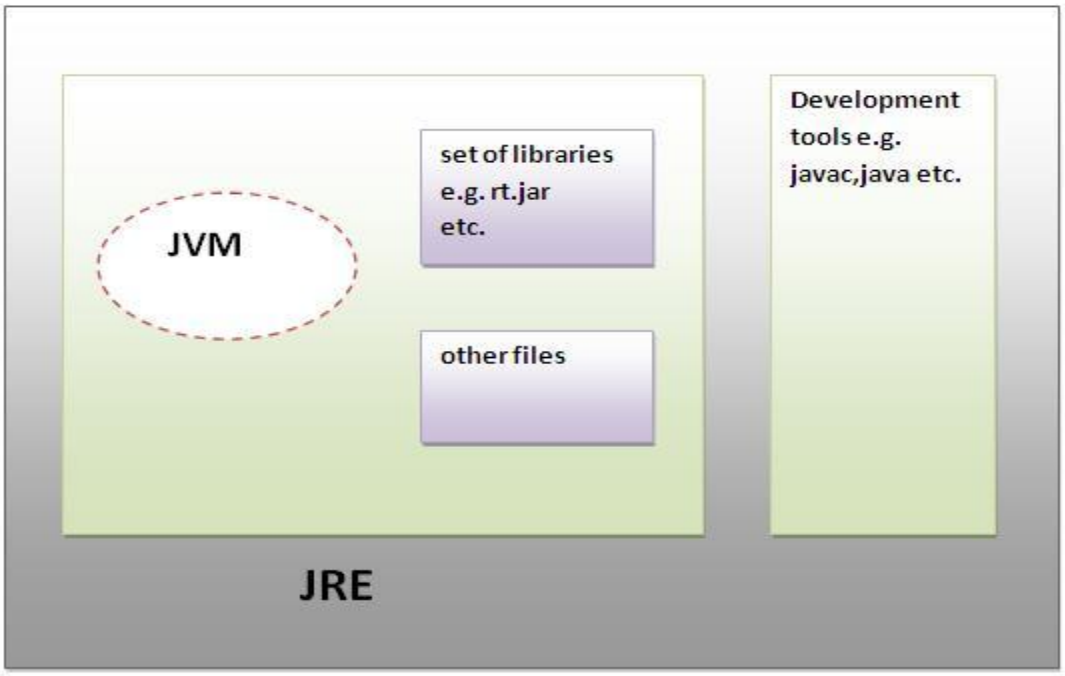
Implementation of JVMs are also actively released by other companies besides Sun Micro Systems.



JRE

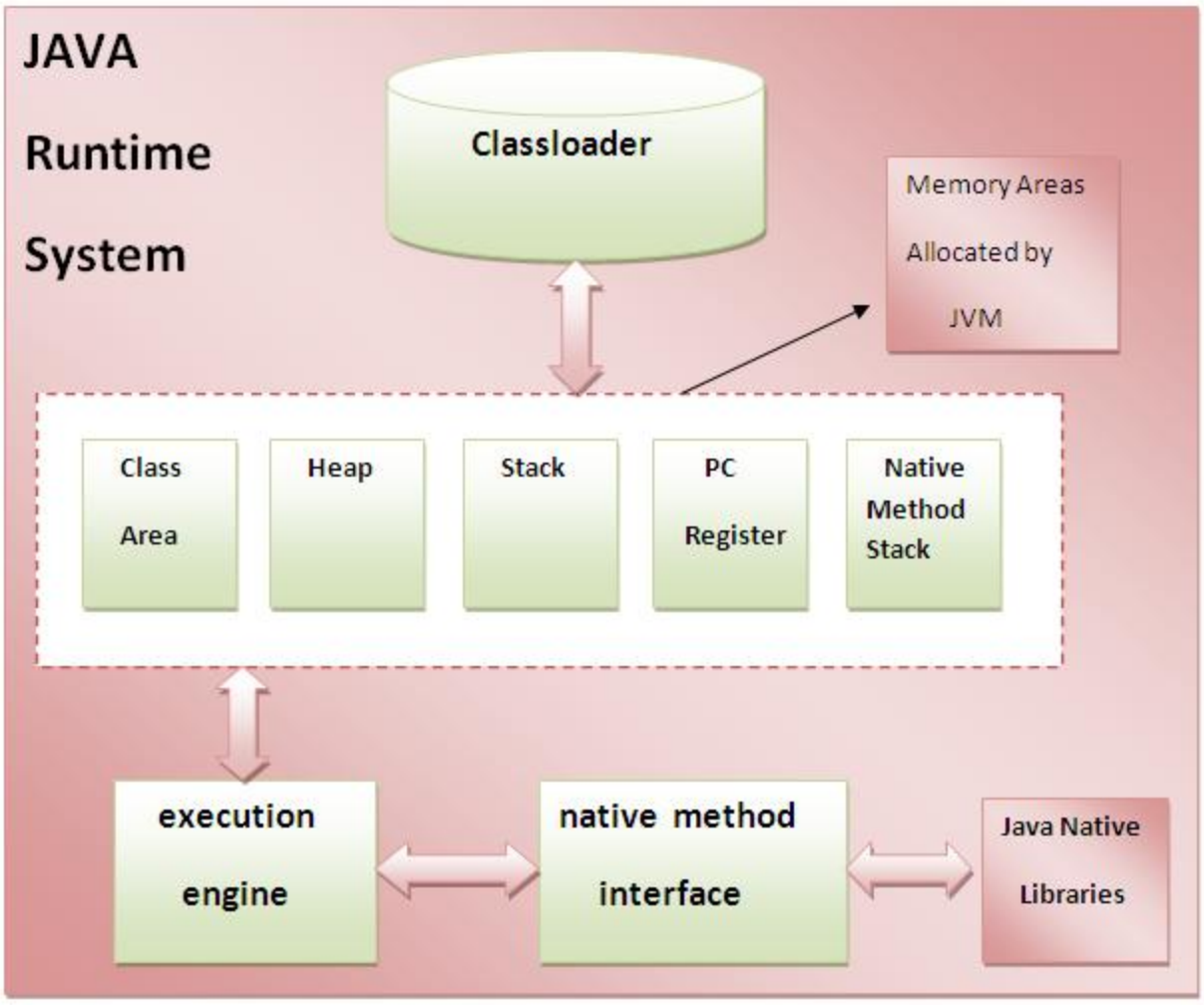
JDK

JDK is an acronym for Java Development Kit.It physically exists.It contains JRE + development tools.



Internal Architecture of JVM

Let's understand the internal architecture of JVM. It contains classloader, memory area, execution engine etc.



1) Classloader: **Classloader is a subsystem of JVM that is used to load class files.**

2) Class(Method) Area:**Class(Method) Area stores per-class structures such as the runtime constant pool, field and method data, the code for methods.**

3) Heap: **It is the runtime data area in which objects are allocated.**

4) Stack:
Java Stack stores frames. It holds local variables and partial results, and plays a part in method invocation and return.

Each thread has a private JVM stack, created at the same time as thread.

A new frame is created each time a method is invoked. A frame is destroyed when its method

invocation completes.

5) Program Counter Register : **PC (program counter) register. It contains the address of the Java virtual machine instruction currently being executed.**

6) Native Method Stack: **It contains all the native methods used in the application.**

7) Execution Engine:

It contains:

1) A virtual processor

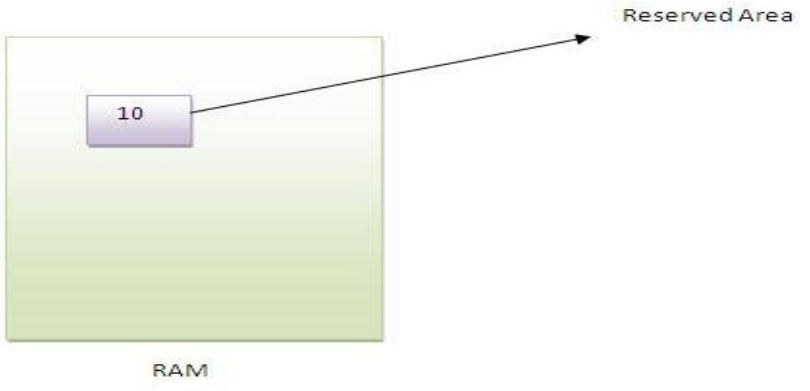
2) Interpreter: Read bytecode stream then execute the instructions.

3) Just-In-Time(JIT) compiler: It is used to improve the performance.JIT compiles parts of the byte code that have similar functionality at the same time, and hence reduces the amount of time needed for compilation.Here the term ?compiler? refers to a translator from the instruction set of a Java virtual machine (JVM) to the instruction set of a specific CPU.

Variable and Datatype in Java

Variable is a name of memory location. There are three types of variables: local, instance and static. There are two types of datatypes in java, primitive and non-primitive.

Variable : Variable is name of reserved area allocated in memory.

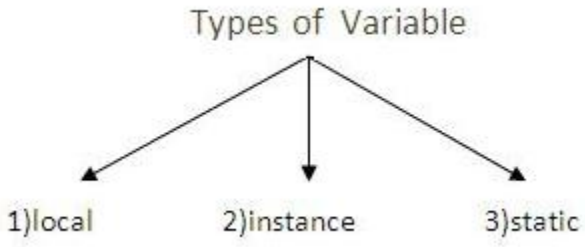



```
int data=50;//Here data is variable
```

Types of Variable

There are three types of variables in java

- local variable
- instance variable
- static variable



Local Variable

A variable that is declared inside the method is called local variable.

Instance Variable

A variable that is declared inside the class but outside the method is called instance variable . It is not declared as static.

Static variable

A variable that is declared as static is called static variable. It cannot be local.

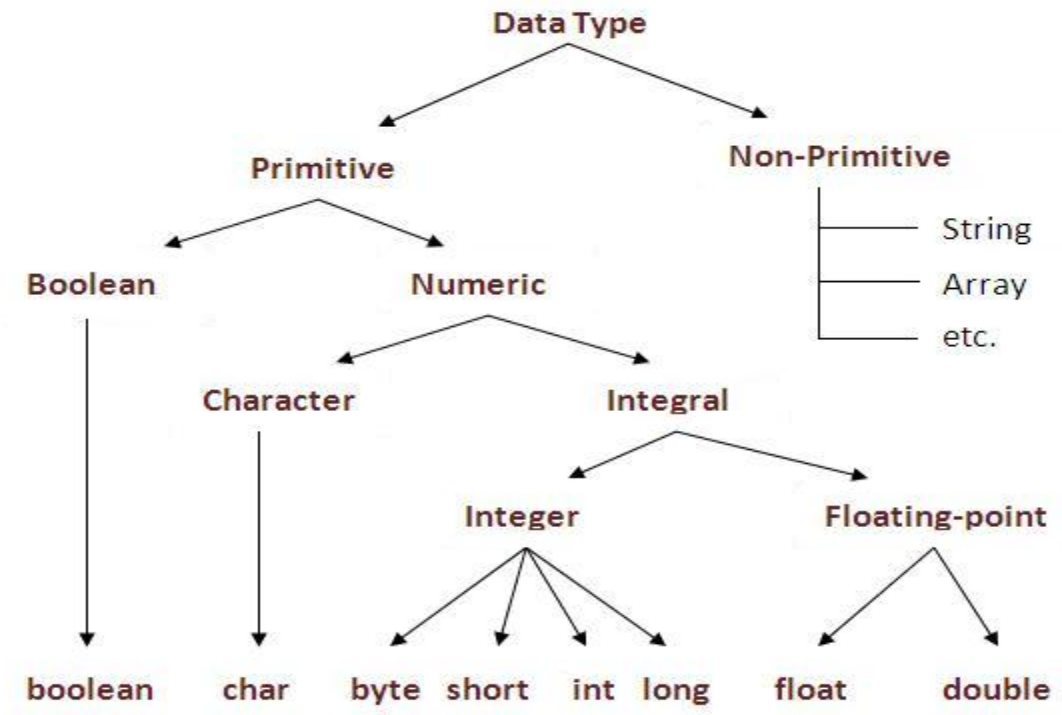
Example to understand the types of variables

```
class A{
int data=50;//instance variable
static int m=100;//static variable
void method(){
int n=90;//local variable
}
}//end of class
```

Data Types in Java

In java, there are two types of data types

- Primitive data type
- Reference data type (non-primitive data type)



Operators in java

Operator in java is a symbol that is used to perform operations. There are many types of operators in java such as unary operator, arithmetic operator, relational operator, shift operator, bitwise operator, ternary operator and assignment operator.

Operators	Precedence
postfix	*expr++ expr--*
unary	*++expr --expr +expr -expr ~ !*
multiplicative	* / %
additive	+ -
shift	<< >> >>>
relational	< > <= >= instanceof
equality	== !=
bitwise AND	&
bitwise exclusive OR	^
bitwise inclusive OR	|
logical AND	&&
logical OR	||
ternary	? :
assignment	= += -= *= /= %= &= ^= |= <<= >>= >>>=

Java Modifier Types

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

- Java Access Modifiers
- Non Access Modifiers

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

```
public class className {
  // ...
}
private boolean myFlag;
static final double weeks = 9.5;
```

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

Access Control Modifiers:

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

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

Non Access Modifiers:

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

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

Java Basic Operators

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

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

The Arithmetic Operators:

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

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

The Relational Operators:

There are following relational operators supported by Java language

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

The Bitwise Operators:

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

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

a = 0011 1100

b = 0000 1101

---

a&b = 0000 1100

a|b = 0011 1101

a^b = 0011 0001

~a = 1100 0011

The following table lists the bitwise operators:

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

The Logical Operators:

The following table lists the logical operators:

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

The Assignment Operators:

There are following assignment operators supported by Java language:

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

Misc Operators

There are few other operators supported by Java Language.

Conditional Operator (? :):

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

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

Following is the example:

```
public class Test {

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

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

instanceof Operator:

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

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

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

```
public class Test {

   public static void main(String args[]){
      String name = "James";
      // following will return true since name is type of String
      boolean result = name instanceof String;
      System.out.println( result );
   }
}
```

This would produce the following result:

```
true
```

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

```
class Vehicle {}

public class Car extends Vehicle {
   public static void main(String args[]){
      Vehicle a = new Car();
      boolean result =  a instanceof Car;
      System.out.println( result );
   }
}
```

This would produce the following result:

```
true
```

Precedence of Java Operators:

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

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

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

Category	Operator	Associativity
Postfix	() [] . (dot operator)	Left to right
Unary	++ - - ! ~	Right to left
Multiplicative	* / %	Left to right
Additive	+ -	Left to right
Shift	>> >>> <<	Left to right
Relational	> >= < <=	Left to right
Equality	== !=	Left to right
Bitwise AND	&	Left to right
Bitwise XOR	^	Left to right
Bitwise OR	|	Left to right
Logical AND	&&	Left to right
Logical OR	||	Left to right
Conditional	?:	Right to left
Assignment	= += -= *= /= %= >>= <<= &= ^= |=	Right to left
Comma	,	Left to right

Java Command Line Arguments

The java command-line argument is an argument i.e. passed at the time of running the java program.

The arguments passed from the console can be received in the java program and it can be used as an input.

So, it provides a convenient way to check the behavior of the program for the different values. You can pass **N** (1,2,3 and so on) numbers of arguments from the command prompt.

Simple example of command-line argument in java

In this example, we are receiving only one argument and printing it. To run this java program, you must pass at least one argument from the command prompt.

```
1.  class CommandLineExample{
2.  public static void main(String args[]){
3.  System.out.println("Your first argument is: "+args[0]);
4.  }
```

```
5.      }
1.      compile by > javac CommandLineExample.java
2.      run by > java CommandLineExample sonoo
```

Output: Your first argument is: sonoo

Example of command-line argument that prints all the values

In this example, we are printing all the arguments passed from the command-line. For this purpose, we have traversed the array using for loop.

```
1.      class A{
2.      public static void main(String args[]){
3.
4.      for(int i=0;i<args.length;i++)
5.      System.out.println(args[i]);
6.
7.      }
8.      }
1.      compile by > javac A.java
2.      run by > java A sonoo jaiswal 1 3 abc
```

Output: sonoo
jaiswal
1
3
abc

Java Loops - for, while and do...while

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

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

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

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

The while Loop:

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

Syntax:

The syntax of a while loop is:

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

When executing, if the boolean_expression result is true, then the actions inside the loop will be executed. This will continue as long as the expression result is true.
Here, key point of the while loop is that the loop might not ever run. When the expression is tested and the result is false, the loop body will be skipped and the first statement after the while loop will be executed.

Example:

```
public class Test {

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

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

This would produce the following result:

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

The do...while Loop:

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

Syntax:

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

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

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

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

Example:

```
public class Test {

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

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

This would produce the following result:

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

The for Loop:

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

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

Syntax:

The syntax of a for loop is:

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

Here is the flow of control in a for loop:

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

Example:

```
public class Test {

  public static void main(String args[]) {

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

This would produce the following result:

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

Enhanced for loop in Java:

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

Syntax:

The syntax of enhanced for loop is:

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

- **Declaration:** The newly declared block variable, which is of a type compatible with the elements of the array you are accessing. The variable will be available within the for block and its value would be the same as the current array element.
- **Expression:** This evaluates to the array you need to loop through. The expression can be an array variable or method call that returns an array.

Example:

```
public class Test {

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

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

```
    }
  }
}
```

This would produce the following result:

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

The break Keyword:

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

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

Syntax:

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

```
break;
```

Example:

```
public class Test {

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

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

This would produce the following result:

```
10
20
```

The continue Keyword:

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

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

Syntax:

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

```
continue;
```

Example:

```
public class Test {

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

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

This would produce the following result:

```
10
20
40
50
```

Java Decision Making

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

- if statements

- switch statements

The if Statement:

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

Syntax:

The syntax of an if statement is:

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

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

Example:

```
public class Test {

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

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

This would produce the following result:

```
This is if statement
```

The if...else Statement:

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

Syntax:

The syntax of an if...else is:

```
if(Boolean_expression){
```

```
  //Executes when the Boolean expression is true
}else{
  //Executes when the Boolean expression is false
}
```

Example:

```
public class Test {

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

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

This would produce the following result:

```
This is else statement
```

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

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

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

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

Syntax:

The syntax of an if...else is:

```
if(Boolean_expression 1){
  //Executes when the Boolean expression 1 is true
}else if(Boolean_expression 2){
  //Executes when the Boolean expression 2 is true
}else if(Boolean_expression 3){
```

```
  //Executes when the Boolean expression 3 is true
}else {
  //Executes when the none of the above condition is true.
}
```

Example:

```
public class Test {

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

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

This would produce the following result:

```
Value of X is 30
```

Nested if...else Statement:

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

Syntax:

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

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

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

Example:

```
public class Test {

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

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

This would produce the following result:

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

The switch Statement:

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

Syntax:

The syntax of enhanced for loop is:

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

The following rules apply to a switch statement:

- The variable used in a switch statement can only be a byte, short, int, or char.
- You can have any number of case statements within a switch. Each case is followed by the value to be compared to and a colon.

- The value for a case must be the same data type as the variable in the switch and it must be a constant or a literal.

- When the variable being switched on is equal to a case, the statements following that case will execute until a break statement is reached.
- When a break statement is reached, the switch terminates, and the flow of control jumps to the next line following the switch statement.
- Not every case needs to contain a break. If no break appears, the flow of control will fall through to subsequent cases until a break is reached.
- A switch statement can have an optional default case, which must appear at the end of the switch. The default case can be used for performing a task when none of the cases is true. No break is needed in the default case.

Example:

```
public class Test {

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

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

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

```
$ java Test
```

```
Well done
Your grade is a C
$
```

Java - Numbers Class

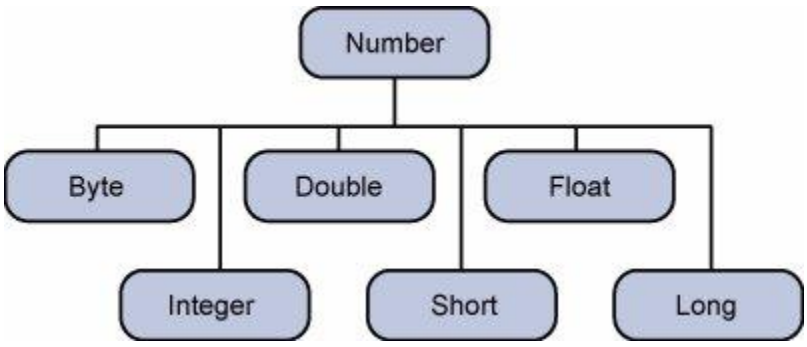
Normally, when we work with Numbers, we use primitive data types such as byte, int, long, double, etc.

Example:

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

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

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



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

Here is an example of boxing and unboxing:

```
public class Test{

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

```
    System.out.println(x);
  }
}
```

This would produce the following result:

```
15
```

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

Number Methods:

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

SN	Methods with Description
1	xxxValue() Converts the value of this Number object to the xxx data type and returned it.
2	compareTo() Compares this Number object to the argument.
3	equals() Determines whether this number object is equal to the argument.
4	valueOf() Returns an Integer object holding the value of the specified primitive.
5	toString() Returns a String object representing the value of specified int or Integer.
6	parseInt() This method is used to get the primitive data type of a certain String.
7	abs() Returns the absolute value of the argument.

8	ceil() Returns the smallest integer that is greater than or equal to the argument. Returned as a double.
9	floor() Returns the largest integer that is less than or equal to the argument. Returned as a double.
10	rint() Returns the integer that is closest in value to the argument. Returned as a double.
11	round() Returns the closest long or int, as indicated by the method's return type, to the argument.
12	min() Returns the smaller of the two arguments.
13	max() Returns the larger of the two arguments.
14	exp() Returns the base of the natural logarithms, e, to the power of the argument.
15	log() Returns the natural logarithm of the argument.
16	pow() Returns the value of the first argument raised to the power of the second argument.
17	sqrt() Returns the square root of the argument.
18	sin() Returns the sine of the specified double value.

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

Escape Sequences:

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

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

Following table shows the Java escape sequences:

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

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

Character Methods:

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

SN	Methods with Description
1	isLetter() Determines whether the specified char value is a letter.
2	isDigit() Determines whether the specified char value is a digit.
3	isWhitespace() Determines whether the specified char value is white space.
4	isUpperCase() Determines whether the specified char value is uppercase.
5	isLowerCase() Determines whether the specified char value is lowercase.

6	toUpperCase() Returns the uppercase form of the specified char value.
7	toLowerCase() Returns the lowercase form of the specified char value.
8	toString() Returns a String object representing the specified character valuethat is, a one-character string.

Package name: java.lang.Character API specification.

Java OOPs Concepts

Object Oriented Programming is a paradigm that provides many concepts such as **inheritance**, **data binding**, **polymorphism, encapsulation, abstraction** etc.

Simula is considered as the first object-oriented programming language. The programming paradigm where everything is represented as an object, is known as truly object-oriented programming language.

Smalltalk is considered as the first truly object-oriented programming language.

OOPs (Object Oriented Programming System)

Object means a real word entity such as pen, chair, table etc. **Object-Oriented Programming** is a methodology or paradigm to design a program using classes and objects. It simplifies the software development and maintenance by providing some concepts:

- Object
- Class
- Inheritance
- Polymorphism
- Abstraction
- Encapsulation

Object

Any entity that has state and behavior is known as an object. For example: chair, pen, table, keyboard, bike etc. It can be physical and logical.

Class

Collection of objects is called class. It is a logical entity.

Inheritance

When one object acquires all the properties and behaviours of parent object i.e. known as inheritance. It provides code reusability. It is used to achieve runtime polymorphism.

Polymorphism

When **one task is performed by different ways** i.e. known as polymorphism. For example: to convense the customer differently, to draw something e.g. shape or rectangle etc.

In java, we use method overloading and method overriding to achieve polymorphism.

Another example can be to speak something e.g. cat speaks meaw, dog barks woof etc.

Abstraction

Hiding internal details and showing functionality is known as abstraction. For example: phone call, we don't know the internal processing.

In java, we use abstract class and interface to achieve abstraction.


Capsule

Encapsulation

Binding (or wrapping) code and data together into a single unit is known as encapsulation. For example: capsule, it is wrapped with different medicines.

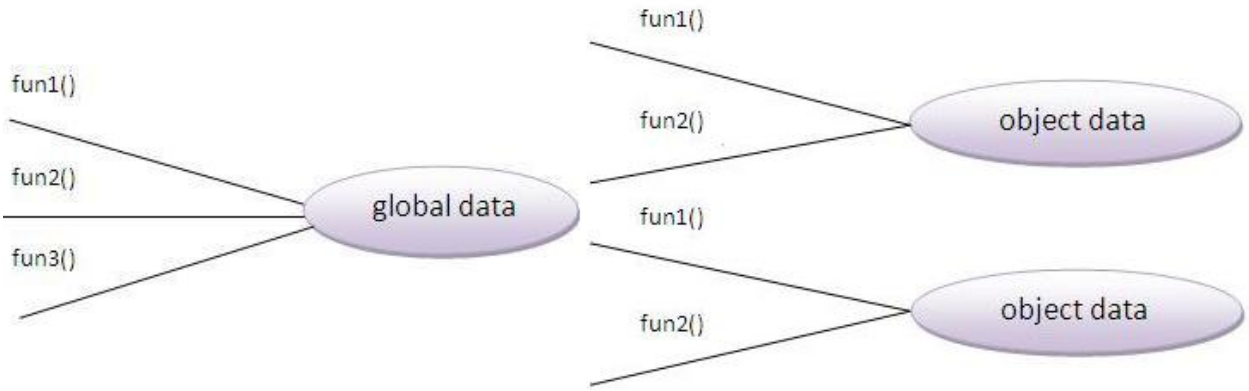
A java class is the example of encapsulation. Java bean is the fully encapsulated class because all the data members are private here.

Advantage of OOPs over Procedure-oriented programming language

1)OOPs makes development and maintenance easier where as in Procedure-oriented programming language it is not easy to manage if code grows as project size grows.

2)OOPs provides data hiding whereas in Procedure-oriented prgramming language a global data can be accessed from anywhere.

3)OOPs provides ability to simulate real-world event much more effectively. We can provide the solution of real word problem if we are using the Object-Oriented Programming language.



What is difference between object-oriented programming language and object-based programming language?

Object based programming language follows all the features of OOPs except Inheritance. JavaScript and VBScript are examples of object based programming languages.

Difference between object and class

There are many differences between object and class. A list of differences between object and class are given below:

No.	Object	Class
1)	Object is an **instance** of a class.	Class is a **blueprint or template** from which objects are created.
2)	Object is a **real world entity** such as pen, laptop, mobile, bed, keyboard, mouse, chair etc.	Class is a **group of similar objects**.
3)	Object is a **physical** entity.	Class is a **logical** entity.
4)	Object is created through **new keyword** mainly e.g. Student s1=new Student();	Class is declared using **class keyword** e.g. class Student{}
5)	Object is created **many times** as per requirement.	Class is declared **once**.
6)	Object **allocates memory when it is created**.	Class **doesn't allocated memory when it is created**.
7)	There are **many ways to create object** in java such as new keyword, newInstance() method, clone() method, factory method and deserialization.	There is only **one way to define class** in java using class keyword.

Java Naming conventions

Java **naming convention** is a rule to follow as you decide what to name your identifiers such as class, package, variable, constant, method etc.

But, it is not forced to follow. So, it is known as convention not rule.

All the classes, interfaces, packages, methods and fields of java programming language are given according to java naming convention.

Advantage of naming conventions in java

By using standard Java naming conventions, you make your code easier to read for yourself and for other programmers. Readability of Java program is very important. It indicates that **less time** is spent to figure out what the code does.

Name	Convention

class name	should start with uppercase letter and be a noun e.g. String, Color, Button, System, Thre
interface name	should start with uppercase letter and be an adjective e.g. Runnable, Remote, ActionList
method name	should start with lowercase letter and be a verb e.g. actionPerformed(), main(), print(), p
variable name	should start with lowercase letter e.g. firstName, orderNumber etc.
package name	should be in lowercase letter e.g. java, lang, sql, util etc.
constants name	should be in uppercase letter. e.g. RED, YELLOW, MAX_PRIORITY etc.

CamelCase in java naming conventions

Java follows camelcase syntax for naming the class, interface, method and variable.

If name is combined with two words, second word will start with uppercase letter always e.g. actionPerformed(), firstName, ActionEvent, ActionListener etc.

Object and Class in Java

In object-oriented programming technique, we design a program using objects and classes.

Object is the physical as well as logical entity whereas class is the logical entity only.

Object in Java

An entity that has state and behavior is known as an object e.g. chair, bike, marker, pen, table, car etc. It can be physical or logical (tengible and intengible). The example of integible object is banking system.

An object has three characteristics:

- **state:** represents data (value) of an object.
- **behavior:** represents the behavior (functionality) of an object such as deposit, withdraw etc.
- **identity:** Object identity is typically implemented via a unique ID. The value of the ID is not visible to the external user. But,it is used internally by the JVM to identify each object uniquely.

For Example: Pen is an object. Its name is Reynolds, color is white etc. known as its state. It is used to write, so writing is its behavior.

Object is an instance of a class. Class is a template or blueprint from which objects are created. So object is the instance(result) of a class.

Class in Java

A class is a group of objects that has common properties. It is a template or blueprint from which objects are created.

A class in java can contain:

- **data member**
- **method**
- **constructor**
- **block**
- **class and interface**

Syntax to declare a class:

```
1.  class <class_name>{
2.     data member;
3.     method;
4.  }
```

Simple Example of Object and Class

In this example, we have created a Student class that have two data members id and name. We are creating the object of the Student class by new keyword and printing the objects value.

```
1.  class Student1{
2.   int id;//data member (also instance variable)
3.   String name;//data member(also instance variable)
4.
5.   public static void main(String args[]){
6.    Student1 s1=new Student1();//creating an object of Student
7.    System.out.println(s1.id);
8.    System.out.println(s1.name);
9.   }
10. }
```

Output:0 null

Instance variable in Java

A variable that is created inside the class but outside the method, is known as instance variable. Instance variable doesn't get memory at compile time.It gets memory at runtime when object(instance) is created.That is why, it is known as instance variable.

Method in Java

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

Advantage of Method

- Code Reusability
- Code Optimization

new keyword

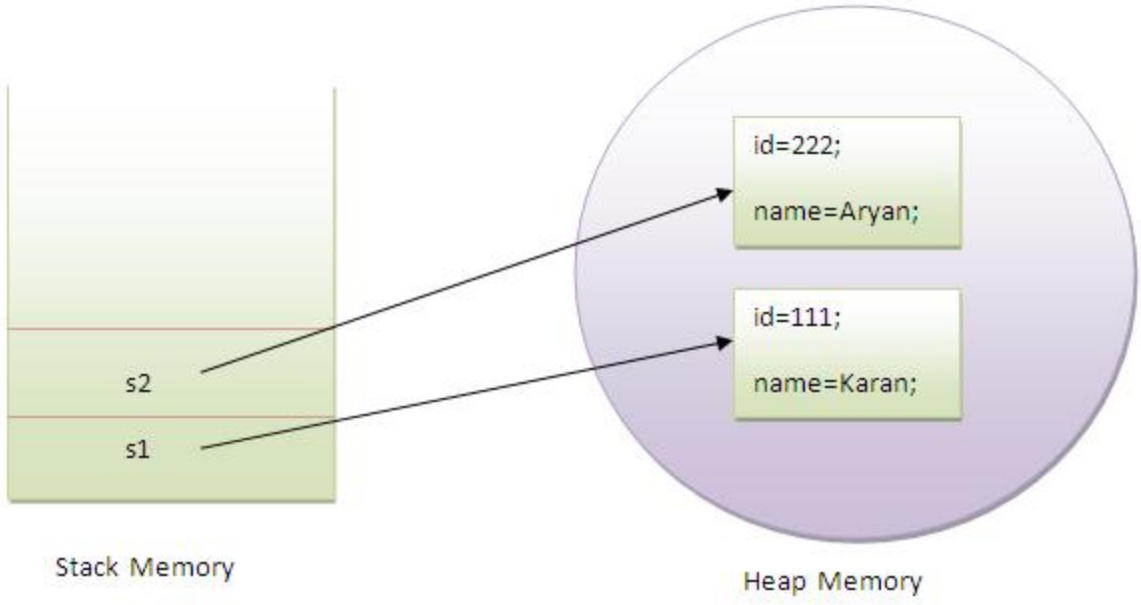
The new keyword is used to allocate memory at runtime.

Example of Object and class that maintains the records of students

In this example, we are creating the two objects of Student class and initializing the value to these objects by invoking the insert Record method on it. Here, we are displaying the state (data) of the objects by invoking the display Information method.

```
1.  class Student2{
2.   int rollno;
3.   String name;
4.
5.   void insertRecord(int r, String n){  //method
6.    rollno=r;
7.    name=n;
8.   }
9.
10.  void displayInformation(){System.out.println(rollno+" "+name);}//method
11.
12.  public static void main(String args[]){
13.   Student2 s1=new Student2();
14.   Student2 s2=new Student2();
15.
16.   s1.insertRecord(111,"Karan");
17.   s2.insertRecord(222,"Aryan");
18.
19.   s1.displayInformation();
20.   s2.displayInformation();
21.
22.  }
23. }
```

Output:111 Karan
222 Aryan



As you see in the above figure, object gets the memory in Heap area and reference variable refers to the object allocated in the Heap memory area. Here, s1 and s2 both are reference variables that refer to the objects allocated in memory.

Another Example of Object and Class

There is given another example that maintains the records of Rectangle class. Its exaplanation is same as in the above Student class example.

```
1.          class Rectangle{
2.           int length;
3.           int width;
4.
5.           void insert(int l,int w){
6.            length=l;
7.            width=w;
8.           }
9.
10.          void calculateArea(){System.out.println(length*width);}
11.
```

```
12.         public static void main(String args[]){
13.          Rectangle r1=new Rectangle();
14.          Rectangle r2=new Rectangle();
15.
16.          r1.insert(11,5);
17.          r2.insert(3,15);
18.
19.          r1.calculateArea();
20.          r2.calculateArea();
21.         }
22.         }
```

```
Output:55
       45
```

What are the different ways to create an object in Java?

There are many ways to create an object in java. They are:

- By new keyword
- By new Instance() method
- By clone() method
- By factory method etc.

We will learn, these ways to create the object later.

Annonymous object

Annonymous simply means nameless. An object that have no reference is known as annonymous object.

If you have to use an object only once, annonymous object is a good approach.

```
1.          class Calculation{
2.
3.           void fact(int  n){
4.            int fact=1;
5.            for(int i=1;i<=n;i++){
6.             fact=fact*i;
7.            }
8.           System.out.println("factorial is "+fact);
9.          }
10.
11.         public static void main(String args[]){
```

```
12.          new Calculation().fact(5);//calling method with annonymous object
13.         }
14.         }
```

Output:Factorial is 120

Creating multiple objects by one type only

We can create multiple objects by one type only as we do in case of primitives.

```
1.          Rectangle r1=new Rectangle(),r2=new Rectangle();//creating two objects
```

Let's see the example:

```
1.          class Rectangle{
2.           int length;
3.           int width;
4.
5.           void insert(int l,int w){
6.            length=l;
7.            width=w;
8.           }
9.
10.          void calculateArea(){System.out.println(length*width);}
11.
12.          public static void main(String args[]){
13.           Rectangle r1=new Rectangle(),r2=new Rectangle();//creating two objects
14.
15.           r1.insert(11,5);
16.           r2.insert(3,15);
17.
18.           r1.calculateArea();
19.           r2.calculateArea();
20.          }
21.          }
```

Output:55
45

Method Overloading in Java

If a class have multiple methods by same name but different parameters, it is known as **Method Overloading**.

If we have to perform only one operation, having same name of the methods increases the readability of the program.

Suppose you have to perform addition of the given numbers but there can be any number of arguments, if you write the method such as a(int,int) for two parameters, and b(int,int,int) for three parameters then it may be difficult for you as well as other programmers to understand the behaviour of the method because its name differs. So, we perform method overloading to figure out the program quickly.

Advantage of method overloading?

Method overloading **increases the readability of the program**.

Different ways to overload the method

There are two ways to overload the method in java

1. By changing number of arguments
2. By changing the data type

In java, Method Overloading is not possible by changing the return type of the method.

1)Example of Method Overloading by changing the no. of arguments

In this example, we have created two overloaded methods, first sum method performs addition of two numbers and second sum method performs addition of three numbers.

```
1.          class Calculation{
2.            void sum(int a,int b){System.out.println(a+b);}
3.            void sum(int a,int b,int c){System.out.println(a+b+c);}
4.
```

```
5.          public static void main(String args[]){
6.          Calculation obj=new Calculation();
7.          obj.sum(10,10,10);
8.          obj.sum(20,20);
9.
10.         }
11.       }
```

```
Output:30
       40
```

2) Example of Method Overloading by changing data type of argument

In this example, we have created two overloaded methods that differs in data type. The first sum method receives two integer arguments and second sum method receives two double arguments.

```
1.          class Calculation2{
2.          void sum(int a,int b){System.out.println(a+b);}
3.          void sum(double a,double b){System.out.println(a+b);}
4.
5.          public static void main(String args[]){
6.          Calculation2 obj=new Calculation2();
7.          obj.sum(10.5,10.5);
8.          obj.sum(20,20);
9.
10.         }
11.       }
```

```
Output:21.0
       40
```

Que) Why Method Overloading is not possible by changing the return type of method?

In java, method overloading is not possible by changing the return type of the method because there may occur ambiguity. Let's see how ambiguity may occur:

because there was problem:

```
1.          class Calculation3{
2.          int sum(int a,int b){System.out.println(a+b);}
3.          double sum(int a,int b){System.out.println(a+b);}
4.
5.          public static void main(String args[]){
```

```
6.          Calculation3 obj=new Calculation3();
7.          int result=obj.sum(20,20); //Compile Time Error
8.
9.          }
10.       }
```

int result=obj.sum(20,20); //Here how can java determine which sum() method should be called

Can we overload main() method?

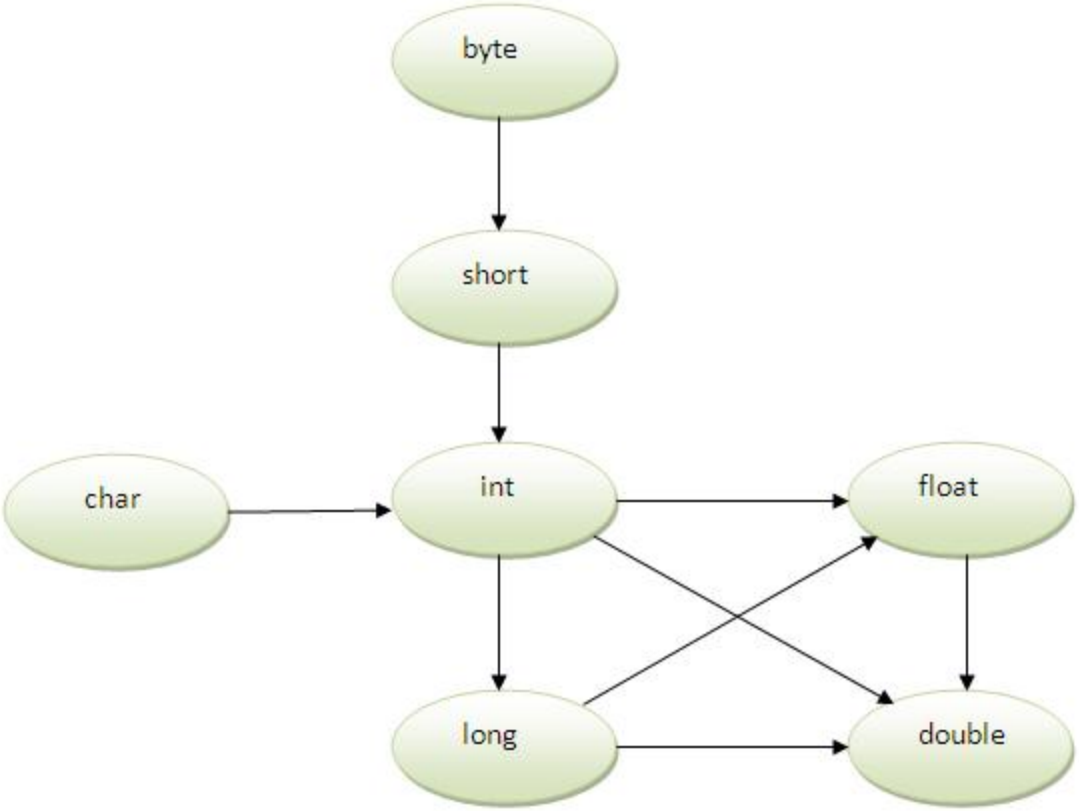
Yes, by method overloading. You can have any number of main methods in a class by method overloading. Let's see the simple example:

```
1.        class Overloading1{
2.          public static void main(int a){
3.          System.out.println(a);
4.          }
5.
6.          public static void main(String args[]){
7.          System.out.println("main() method invoked");
8.          main(10);
9.          }
10.       }
```

```
Output: main() method invoked
        10
```

Method Overloading and TypePromotion

One type is promoted to another implicitly if no matching datatype is found. Let's understand the concept by the figure given below:



As displayed in the above diagram, byte can be promoted to short, int, long, float or double. The short datatype can be promoted to int,long,float or double. The char datatype can be promoted to int,long,float or double and so on.

Example of Method Overloading with TypePromotion

```
1.          class OverloadingCalculation1{
2.            void sum(int a,long b){System.out.println(a+b);}
3.            void sum(int a,int b,int c){System.out.println(a+b+c);}
4.
5.            public static void main(String args[]){
6.            OverloadingCalculation1 obj=new OverloadingCalculation1();
7.            obj.sum(20,20);//now second int literal will be promoted to long
8.            obj.sum(20,20,20);
9.
10.           }
11.         }
```

Output:40

```
60
```

Example of Method Overloading with TypePromotion if matching found

If there are matching type arguments in the method, type promotion is not performed.

```
1.  class OverloadingCalculation2{
2.    void sum(int a,int b){System.out.println("int arg method invoked");}
3.    void sum(long a,long b){System.out.println("long arg method invoked");}
4.
5.    public static void main(String args[]){
6.    OverloadingCalculation2 obj=new OverloadingCalculation2();
7.    obj.sum(20,20);//now int arg sum() method gets invoked
8.    }
9.  }
```

Output: int arg method invoked

Example of Method Overloading with TypePromotion in case ambiguity

If there are no matching type arguments in the method, and each method promotes similar number of arguments, there will be ambiguity.

```
1.  class OverloadingCalculation3{
2.    void sum(int a,long b){System.out.println("a method invoked");}
3.    void sum(long a,int b){System.out.println("b method invoked");}
4.
5.    public static void main(String args[]){
6.    OverloadingCalculation3 obj=new OverloadingCalculation3();
7.    obj.sum(20,20);//now ambiguity
8.    }
9.  }
```

Output:Compile Time Error

Difference between method overloading and method overriding in java

There are many differences between method overloading and method overriding in java. A list of differences between method overloading and method overriding are given below:

No.	Method Overloading	Method Overriding
1)	Method overloading is used *to increase the readability* of the program.	Method overriding is used *to provide the specific implementation* of the method that is already provided by its super class.
2)	Method overloading is performed *within class*.	Method overriding occurs *in two classes* that have IS-A (inheritance) relationship.
3)	In case of method overloading, *parameter must be different*.	In case of method overriding, *parameter must be same*.
4)	Method overloading is the example of *compile time polymorphism*.	Method overriding is the example of *run time polymorphism*.
5)	In java, method overloading can't be performed by changing return type of the method only. *Return type can be same or different* in method overloading. But you must have to change the parameter.	*Return type must be same or covariant* in method overriding.

Java Method Overloading example

```
1.  class OverloadingExample{
2.  static int add(int a,int b){return a+b;}
3.  static int add(int a,int b,int c){return a+b+c;}
4.  }
```

Java Method Overriding example

```
1.  class Animal{
2.  void eat(){System.out.println("eating...");}
3.  }
4.  class Dog extends Animal{
5.  void eat(){System.out.println("eating bread...");}
6.  }
```

The Constructors:

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

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

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

Constructor in java is a special type of method that is used to initialize the object.

Java constructor is invoked at the time of object creation. It constructs the values i.e. provides data for the object that is why it is known as constructor.

Rules for creating java constructor

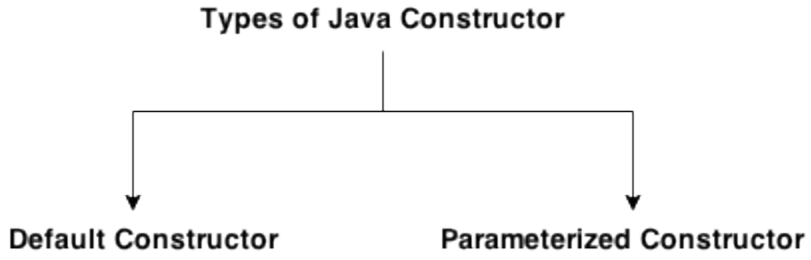
There are basically two rules defined for the constructor.

1. Constructor name must be same as its class name
2. Constructor must have no explicit return type

Types of java constructors

There are two types of constructors:

1. Default constructor (no-arg constructor)
2. Parameterized constructor



Java Default Constructor

A constructor that have no parameter is known as default constructor.

Syntax of default constructor:

```
1.  <class_name>(){}
```

Example of default constructor

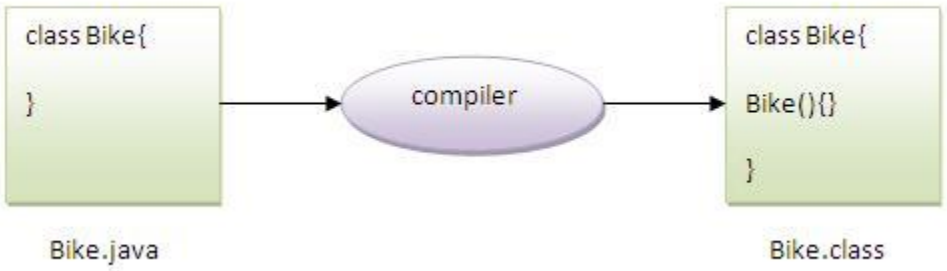
In this example, we are creating the no-arg constructor in the Bike class. It will be invoked at the time of object creation.

```
1.  class Bike1{
2.  Bike1(){System.out.println("Bike is created");}
3.  public static void main(String args[]){
4.  Bike1 b=new Bike1();
5.  }
6.  }
```

Output:

```
Bike is created
```

Rule: If there is no constructor in a class, compiler automatically creates a default constructor.



Q) What is the purpose of default constructor?

Default constructor provides the default values to the object like 0, null etc. depending on the type.

Example of default constructor that displays the default values

```
1.  class Student3{
2.  int id;
3.  String name;
```

```
4.
5.      void display(){System.out.println(id+" "+name);}
6.
7.      public static void main(String args[]){
8.      Student3 s1=new Student3();
9.      Student3 s2=new Student3();
10.     s1.display();
11.     s2.display();
12.     }
13.     }
```

Output:

0 null

0 null

Explanation: In the above class,you are not creating any constructor so compiler provides you a default constructor. Here 0 and null values are provided by default constructor.

Java parameterized constructor

A constructor that have parameters is known as parameterized constructor.

Why use parameterized constructor?

Parameterized constructor is used to provide different values to the distinct objects.

Example of parameterized constructor

In this example, we have created the constructor of Student class that have two parameters. We can have any number of parameters in the constructor.

```
1.      class Student4{
2.        int id;
3.        String name;
4.
5.        Student4(int i,String n){
6.        id = i;
7.        name = n;
8.        }
9.        void display(){System.out.println(id+" "+name);}
10.
11.       public static void main(String args[]){
12.       Student4 s1 = new Student4(111,"Karan");
```

```
13.         Student4 s2 = new Student4(222,"Aryan");
14.         s1.display();
15.         s2.display();
16.        }
17.     }
```

Output:

```
111 Karan
222 Aryan
```

Constructor Overloading in Java

Constructor overloading is a technique in Java in which a class can have any number of constructors that differ in parameter lists. The compiler differentiates these constructors by taking into account the number of parameters in the list and their type.

Example of Constructor Overloading

```
1.      class Student5{
2.         int id;
3.         String name;
4.         int age;
5.         Student5(int i,String n){
6.         id = i;
7.         name = n;
8.         }
9.         Student5(int i,String n,int a){
10.        id = i;
11.        name = n;
12.        age=a;
13.        }
14.        void display(){System.out.println(id+" "+name+" "+age);}
15.
16.        public static void main(String args[]){
17.        Student5 s1 = new Student5(111,"Karan");
18.        Student5 s2 = new Student5(222,"Aryan",25);
19.        s1.display();
20.        s2.display();
21.        }
22.     }
```

Output:

```
111 Karan 0
```

222 Aryan 25

Difference between constructor and method in java

There are many differences between constructors and methods. They are given below.

Java Constructor	Java Method
Constructor is used to initialize the state of an object.	Method is used to expose behaviour of an object.
Constructor must not have return type.	Method must have return type.
Constructor is invoked implicitly.	Method is invoked explicitly.
The java compiler provides a default constructor if you don't have any constructor.	Method is not provided by compiler in any case.
Constructor name must be same as the class name.	Method name may or may not be same as class name.

Java Copy Constructor

There is no copy constructor in java. But, we can copy the values of one object to another like copy constructor in C++.

There are many ways to copy the values of one object into another in java. They are:

- By constructor
- By assigning the values of one object into another
- By clone() method of Object class

In this example, we are going to copy the values of one object into another using java constructor.

```
1.  class Student6{
2.     int id;
3.     String name;
```

```
4.          Student6(int i,String n){
5.          id = i;
6.          name = n;
7.          }
8.
9.          Student6(Student6 s){
10.         id = s.id;
11.         name =s.name;
12.         }
13.         void display(){System.out.println(id+" "+name);}
14.
15.         public static void main(String args[]){
16.         Student6 s1 = new Student6(111,"Karan");
17.         Student6 s2 = new Student6(s1);
18.         s1.display();
19.         s2.display();
20.        }
21.     }
```

Output:

```
111 Karan
111 Karan
```

Copying values without constructor

We can copy the values of one object into another by assigning the objects values to another object. In this case, there is no need to create the constructor.

```
1.      class Student7{
2.         int id;
3.         String name;
4.         Student7(int i,String n){
5.         id = i;
6.         name = n;
7.         }
8.         Student7(){}
9.         void display(){System.out.println(id+" "+name);}
10.
11.        public static void main(String args[]){
12.        Student7 s1 = new Student7(111,"Karan");
13.        Student7 s2 = new Student7();
14.        s2.id=s1.id;
```

```
15.              s2.name=s1.name;
16.              s1.display();
17.              s2.display();
18.             }
19.          }
```

Output: 111 Karan

```
111 Karan
```

Q) Does constructor return any value?

Ans:yes, that is current class instance (You cannot use return type yet it returns a value).

Can constructor perform other tasks instead of initialization?

Yes, like object creation, starting a thread, calling method etc. You can perform any operation in the constructor as you perform in the method.

Example:

Here is a simple example that uses a constructor:

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

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

You would call constructor to initialize objects as follows:

```
public class ConsDemo {

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

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

Example:

Here is a simple example that uses a constructor:

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

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

You would call constructor to initialize objects as follows:

```
public class ConsDemo {

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

This would produce the following result:

```
10 20
```

Variable Arguments(var-args):

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

```
typeName... parameterName
```

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

Example:

```
public class VarargsDemo {

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

  public static void printMax( double... numbers) {
  if (numbers.length == 0) {
    System.out.println("No argument passed");
    return;
  }

  double result = numbers[0];

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

This would produce the following result:

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

The finalize( ) Method:

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

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

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

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

The finalize( ) method has this general form:

```
protected void finalize( )
{
  // finalization code here
}
```

Here, the keyword protected is a specifier that prevents access to finalize( ) by code defined outside its class.

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

Java – Inheritance

Inheritance in java is a mechanism in which one object acquires all the properties and behaviors of parent object.

The idea behind inheritance in java is that you can create new classes that are built upon existing classes. When you inherit from an existing class, you can reuse methods and fields of parent class, and you can add new methods and fields also.

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

Why use inheritance in java

- For Method Overriding (so runtime polymorphism can be achieved).
- For Code Reusability.

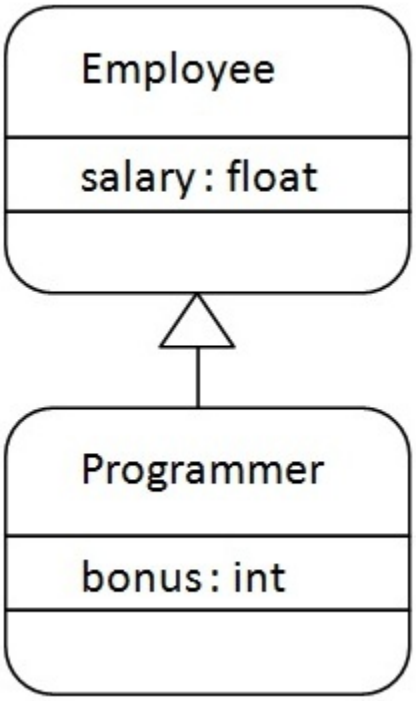
Syntax of Java Inheritance

```
1.  class Subclass-name extends Superclass-name
2.  {
3.    //methods and fields
4.  }
```

The **extends keyword** indicates that you are making a new class that derives from an existing class.

In the terminology of Java, a class that is inherited is called a super class. The new class is called a subclass.

Understanding the simple example of inheritance



As displayed in the above figure, Programmer is the subclass and Employee is the superclass. Relationship between two classes is **Programmer IS-A Employee**.It means that Programmer is a type of Employee.

```
1.  class Employee{
2.   float salary=40000;
3.  }
4.  class Programmer extends Employee{
5.   int bonus=10000;
6.   public static void main(String args[]){
7.     Programmer p=new Programmer();
8.     System.out.println("Programmer salary is:"+p.salary);
9.     System.out.println("Bonus of Programmer is:"+p.bonus);
10. }
11. }
```

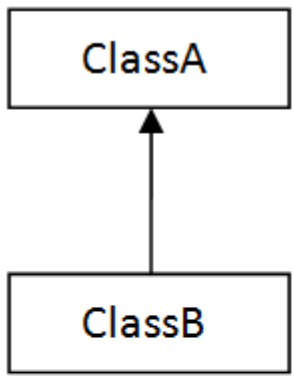
```
Programmer salary is:40000.0
Bonus of programmer is:10000
```

In the above example, Programmer object can access the field of own class as well as of Employee class i.e. code reusability.

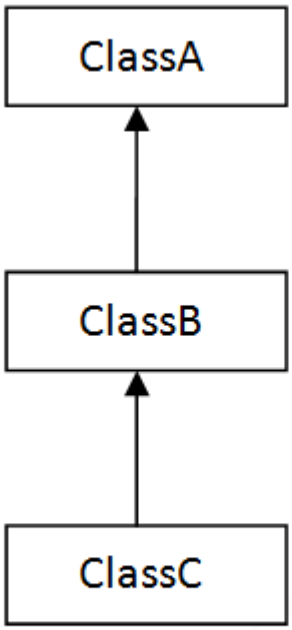
Types of inheritance in java

On the basis of class, there can be three types of inheritance in java: single, multilevel and hierarchical.

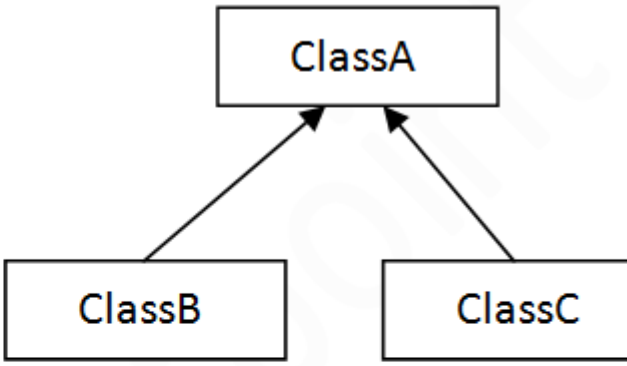
In java programming, multiple and hybrid inheritance is supported through interface only. We will learn about interfaces later.



1) Single



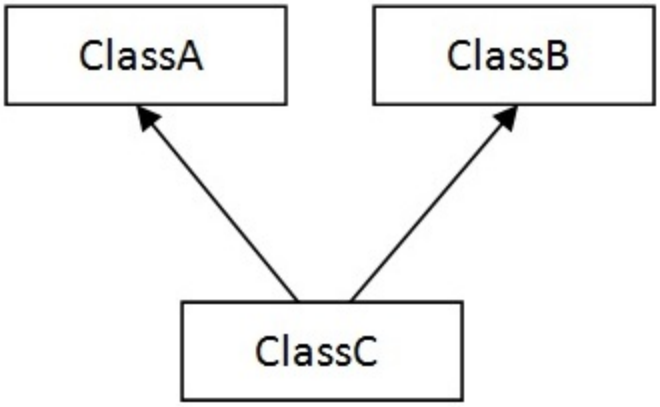
2) Multilevel



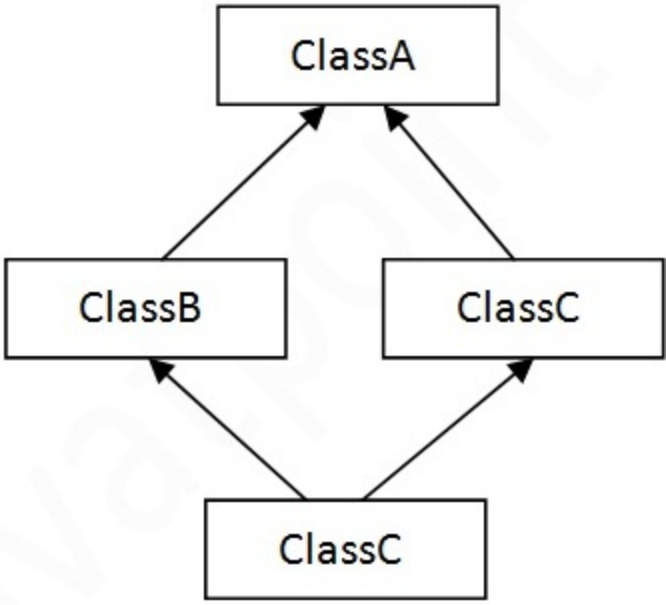
3) Hierarchical

Note: Multiple inheritance is not supported in java through class.

When a class extends multiple classes i.e. known as multiple inheritance. For Example:



4) Multiple



5) Hybrid

.

Q) Why multiple inheritance is not supported in java?

To reduce the complexity and simplify the language, multiple inheritance is not supported in java.

Consider a scenario where A, B and C are three classes. The C class inherits A and B classes. If A and B classes have same method and you call it from child class object, there will be ambiguity to call method of A or B class.

Since compile time errors are better than runtime errors, java renders compile time error if you inherit 2 classes. So whether you have same method or different, there will be compile time error now.

```
1.      class A{
2.      void msg(){System.out.println("Hello");}
3.      }
4.      class B{
5.      void msg(){System.out.println("Welcome");}
6.      }
```

```
7.          class C extends A,B{//suppose if it were
8.
9.           Public Static void main(String args[]){
10.            C obj=new C();
11.            obj.msg();//Now which msg() method would be invoked?
12.         }
13.         }
```

Output:

```
Compile Time Error
```

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

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

IS-A Relationship:

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

```
public class Animal{
}

public class Mammal extends Animal{
}

public class Reptile extends Animal{
}

public class Dog extends Mammal{
}
```

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

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

- Dog is the subclass of both Mammal and Animal classes.

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

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

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

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

Example:

```
public class Dog extends Mammal{

  public static void main(String args[]){

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

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

This would produce the following result:

```
true
true
true
```

Since we have a good understanding of the **extends** keyword let us look into how the **implements**keyword is used to get the IS-A relationship.
The **implements** keyword is used by classes by inherit from interfaces. Interfaces can never be extended by the classes.

Example:

```
public interface Animal {}
```

```
public class Mammal implements Animal{
}

public class Dog extends Mammal{
}
```

The instanceof Keyword:

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

```
interface Animal{}

class Mammal implements Animal{}

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

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

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

This would produce the following result:

```
true
true
true
```

HAS-A relationship:

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

Lets us look into an example:

```
public class Vehicle{}
```

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

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

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

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

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

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

Java - Overriding

If a class inherits a method from its super class, then there is a chance to override the method provided that it is not marked final.

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

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

Example:

Let us look at an example.

```
class Animal{

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

class Dog extends Animal{
```

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

public class TestDog{

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

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

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

This would produce the following result:

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

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

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

Consider the following example :

```
class Animal{

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

class Dog extends Animal{

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

```
    System.out.println("Dogs can bark");
  }
}

public class TestDog{

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

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

This would produce the following result:

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

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

Rules for method overriding:

- The argument list should be exactly the same as that of the overridden method.
- The return type should be the same or a subtype of the return type declared in the original overridden method in the superclass.
- The access level cannot be more restrictive than the overridden method's access level. For example: if the superclass method is declared public then the overridding method in the sub class cannot be either private or protected.
- Instance methods can be overridden only if they are inherited by the subclass.
- A method declared final cannot be overridden.
- A method declared static cannot be overridden but can be re-declared.
- If a method cannot be inherited, then it cannot be overridden.

- A subclass within the same package as the instance's superclass can override any superclass method that is not declared private or final.
- A subclass in a different package can only override the non-final methods declared public or protected.
- An overriding method can throw any uncheck exceptions, regardless of whether the overridden method throws exceptions or not. However the overriding method should not throw checked exceptions that are new or broader than the ones declared by the overridden method. The overriding method can throw narrower or fewer exceptions than the overridden method.
- Constructors cannot be overridden.

Using the super keyword:

The **super** keyword in java is a reference variable that is used to refer immediate parent class object.

Whenever you create the instance of subclass, an instance of parent class is created implicitly i.e. referred by super reference variable.

Usage of java super Keyword

1. super is used to refer immediate parent class instance variable.
2. super() is used to invoke immediate parent class constructor.
3. super is used to invoke immediate parent class method.

1) super is used to refer immediate parent class instance variable.

Problem without super keyword

```
1.  class Vehicle{
2.    int speed=50;
3.  }
4.  class Bike3 extends Vehicle{
5.    int speed=100;
6.    void display(){
7.     System.out.println(speed);//will print speed of Bike
8.    }
9.    public static void main(String args[]){
10.    Bike3 b=new Bike3();
11.    b.display();
12.  }
13.  }
```

Output:100

In the above example Vehicle and Bike both class have a common property speed. Instance variable of current class is refered by instance bydefault, but I have to refer parent class instance variable that is why we use super keyword to distinguish between parent class instance variable and current class instance variable.

Solution by super keyword

```
1.  //example of super keyword
2.
3.  class Vehicle{
4.    int speed=50;
5.  }
6.
7.  class Bike4 extends Vehicle{
8.    int speed=100;
9.
10.   void display(){
11.    System.out.println(super.speed);//will print speed of Vehicle now
12.   }
13.   public static void main(String args[]){
14.    Bike4 b=new Bike4();
15.    b.display();
16.
17. }
18. }
```

Output:50

2) super is used to invoke parent class constructor.

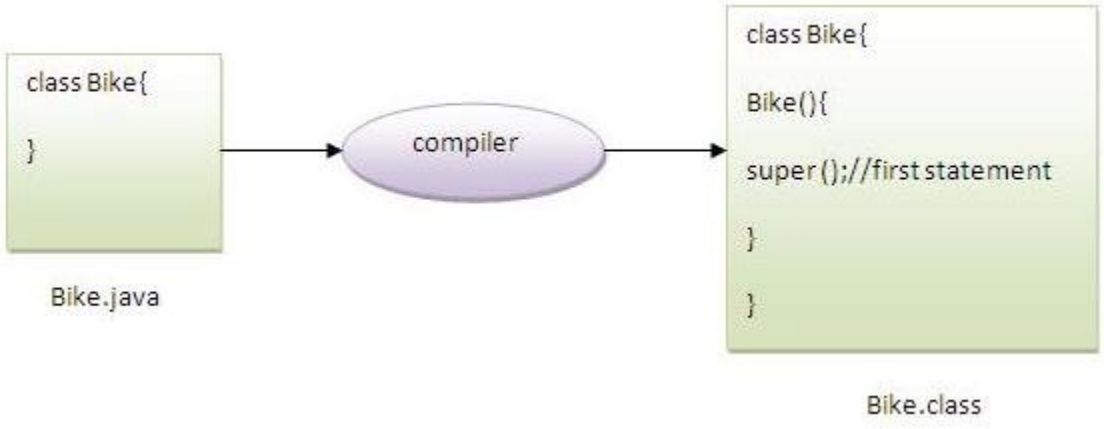
The super keyword can also be used to invoke the parent class constructor as given below:

```
1.  class Vehicle{
2.    Vehicle(){System.out.println("Vehicle is created");}
3.  }
4.
5.  class Bike5 extends Vehicle{
6.    Bike5(){
7.     super();//will invoke parent class constructor
8.     System.out.println("Bike is created");
9.    }
10.   public static void main(String args[]){
```

```
11.         Bike5 b=new Bike5();
12.
13.       }
14.       }
```

Output:Vehicle is created
Bike is created

Note: super() is added in each class constructor automatically by compiler.



As we know well that default constructor is provided by compiler automatically but it also adds super() for the first statement.If you are creating your own constructor and you don't have either this() or super() as the first statement, compiler will provide super() as the first statement of the constructor.

Another example of super keyword where super() is provided by the compiler implicitly.

```
1.        class Vehicle{
2.          Vehicle(){System.out.println("Vehicle is created");}
3.        }
4.
5.        class Bike6 extends Vehicle{
6.          int speed;
7.          Bike6(int speed){
8.            this.speed=speed;
9.            System.out.println(speed);
10.         }
11.         public static void main(String args[]){
12.          Bike6 b=new Bike6(10);
13.       }
14.       }
```

Output:Vehicle is created

```
10
```

3) super can be used to invoke parent class method

The super keyword can also be used to invoke parent class method. It should be used in case subclass contains the same method as parent class as in the example given below:

```
1.  class Person{
2.  void message(){System.out.println("welcome");}
3.  }
4.
5.  class Student16 extends Person{
6.  void message(){System.out.println("welcome to java");}
7.
8.  void display(){
9.  message();//will invoke current class message() method
10. super.message();//will invoke parent class message() method
11. }
12.
13. public static void main(String args[]){
14. Student16 s=new Student16();
15. s.display();
16. }
17. }
```

```
Output:welcome to java
       welcome
```

In the above example Student and Person both classes have message() method if we call message() method from Student class, it will call the message() method of Student class not of Person class because priority is given to local.

In case there is no method in subclass as parent, there is no need to use super. In the example given below message() method is invoked from Student class but Student class does not have message() method, so you can directly call message() method.

Program in case super is not required

```
1.  class Person{
2.  void message(){System.out.println("welcome");}
3.  }
4.
5.  class Student17 extends Person{
6.
7.  void display(){
```

```
8.          message();//will invoke parent class message() method
9.          }
10.
11.         public static void main(String args[]){
12.         Student17 s=new Student17();
13.         s.display();
14.         }
15.         }
```

Output:welcome

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

```
class Animal{

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

class Dog extends Animal{

  public void move(){
    super.move(); // invokes the super class method
    System.out.println("Dogs can walk and run");
  }
}

public class TestDog{

  public static void main(String args[]){

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

  }
}
```

This would produce the following result:

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

Java - Polymorphism

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

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

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

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

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

Example:

Let us look at an example.

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

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

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

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

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

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

Java – Abstraction

Abstract class in Java

A class that is declared with abstract keyword, is known as abstract class in java. It can have abstract and non-abstract methods (method with body).

Before learning java abstract class, let's understand the abstraction in java first.

Abstraction in Java

Abstraction is a process of hiding the implementation details and showing only functionality to the user.

Another way, it shows only important things to the user and hides the internal details for example sending sms, you just type the text and send the message. You don't know the internal processing about the message delivery.

Abstraction lets you focus on what the object does instead of how it does it.

Ways to achieve Abstaction

There are two ways to achieve abstraction in java

1. Abstract class (0 to 100%)
2. Interface (100%)

Abstract class in Java

A class that is declared as abstract is known as **abstract class**. It needs to be extended and its method implemented. It cannot be instantiated.

Example abstract class

```
1.  abstract class A{}
```

Abstract Method

A method that is declared as abstract and does not have implementation is known as abstract

method.

Example abstract method

```
1.  abstract void printStatus();//no body and abstract
```

Example of abstract class that has abstract method

In this example, Bike the abstract class that contains only one abstract method run. It implementation is provided by the Honda class.

```
1.  abstract class Bike{
2.    abstract void run();
3.  }
4.
5.  class Honda4 extends Bike{
6.  void run(){System.out.println("running safely..");}
7.
8.  public static void main(String args[]){
9.   Bike obj = new Honda4();
10.  obj.run();
11. }
12. }
```

Output:

```
running safely..
```

Understanding the real scenario of abstract class

In this example, Shape is the abstract class, its implementation is provided by the Rectangle and Circle classes. Mostly, we don't know about the implementation class (i.e. hidden to the end user) and object of the implementation class is provided by the **factory method**.

A **factory method** is the method that returns the instance of the class. We will learn about the factory method later.

In this example, if you create the instance of Rectangle class, draw() method of Rectangle class will be invoked.

File: TestAbstraction1.java

```
1.          abstract class Shape{
2.          abstract void draw();
3.          }
4.          //In real scenario, implementation is provided by others i.e. unknown by end user
5.          class Rectangle extends Shape{
6.          void draw(){System.out.println("drawing rectangle");}
7.          }
8.
9.          class Circle1 extends Shape{
10.         void draw(){System.out.println("drawing circle");}
11.         }
12.
13.         //In real scenario, method is called by programmer or user
14.         class TestAbstraction1{
15.         public static void main(String args[]){
16.         Shape s=new Circle1();//In real scenario, object is provided through method e.g. getShap
    e() method
17.         s.draw();
18.         }
19.         }
```

Output:

```
drawing circle
```

Another example of abstract class in java

File: TestBank.java

```
1.           abstract class Bank{
2.          abstract int getRateOfInterest();
3.          }
4.
5.          class SBI extends Bank{
6.          int getRateOfInterest(){return 7;}
7.          }
8.          class PNB extends Bank{
9.          int getRateOfInterest(){return 7;}
10.         }
11.
12.         class TestBank{
13.         public static void main(String args[]){
14.         Bank b=new SBI();//if object is PNB, method of PNB will be invoked
15.         int interest=b.getRateOfInterest();
16.         System.out.println("Rate of Interest is: "+interest+" %");
```

```
17.         }}
```

Output:

```
Rate of Interest is: 7 %
```

Abstract class having constructor, data member, methods etc.

An abstract class can have data member, abstract method, method body, constructor and even main() method.

File: TestAbstraction2.java

```
1.          //example of abstract class that have method body
2.          abstract class Bike{
3.            Bike(){System.out.println("bike is created");}
4.            abstract void run();
5.            void changeGear(){System.out.println("gear changed");}
6.          }
7.
8.          class Honda extends Bike{
9.          void run(){System.out.println("running safely..");}
10.         }
11.         class TestAbstraction2{
12.         public static void main(String args[]){
13.          Bike obj = new Honda();
14.          obj.run();
15.          obj.changeGear();
16.         }
17.        }
```

Output:

```
bike is created
running safely..
gear changed
```

Rule: If there is any abstract method in a class, that class must be abstract.

```
1.          class Bike12{
2.          abstract void run();
3.          }
```

Output:

compile time error

Rule: If you are extending any abstract class that have abstract method, you must either provide the implementation of the method or make this class abstract.

Another real scenario of abstract class

The abstract class can also be used to provide some implementation of the interface. In such case, the end user may not be forced to override all the methods of the interface.

```
1.  interface A{
2.  void a();
3.  void b();
4.  void c();
5.  void d();
6.  }
7.
8.  abstract class B implements A{
9.  public void c(){System.out.println("I am C");}
10. }
11.
12. class M extends B{
13. public void a(){System.out.println("I am a");}
14. public void b(){System.out.println("I am b");}
15. public void d(){System.out.println("I am d");}
16. }
17.
18. class Test5{
19. public static void main(String args[]){
20. A a=new M();
21. a.a();
22. a.b();
23. a.c();
24. a.d();
25. }}
```

Output:

```
Output:I am a
       I am b
       I am c
       I am d
```

Java – Interfaces

An **interface in java** is a blueprint of a class. It has static constants and abstract methods only.

The interface in java is **a mechanism to achieve fully abstraction**. There can be only abstract methods in the java interface not method body. It is used to achieve fully abstraction and multiple inheritance in Java.

Java Interface also **represents IS-A relationship**.

It cannot be instantiated just like abstract class.

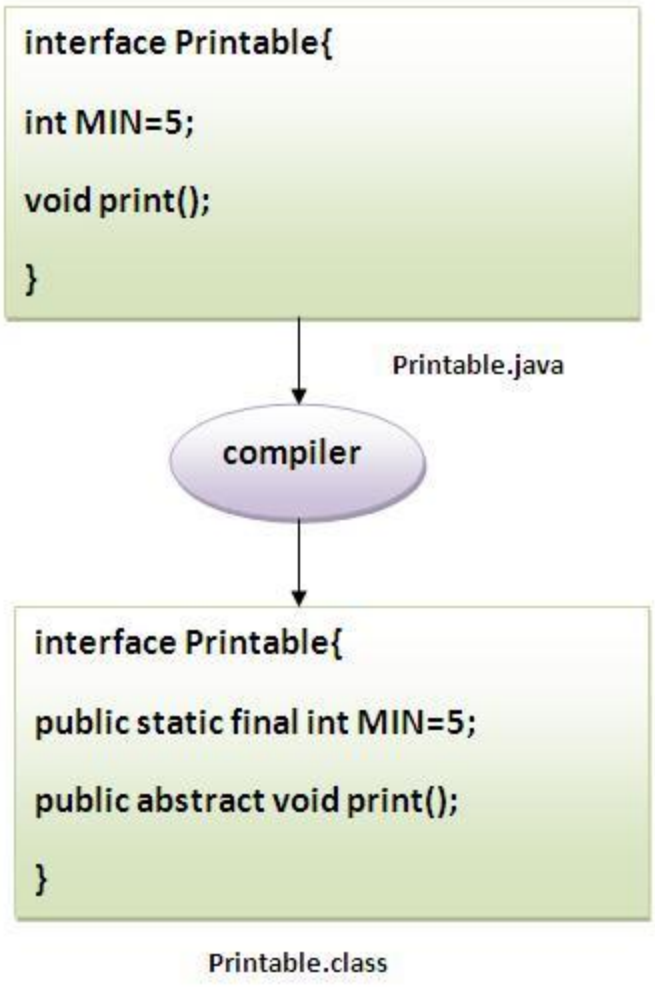
Why use Java interface?

There are mainly three reasons to use interface. They are given below.

- It is used to achieve fully abstraction.
- By interface, we can support the functionality of multiple inheritance.
- It can be used to achieve loose coupling.

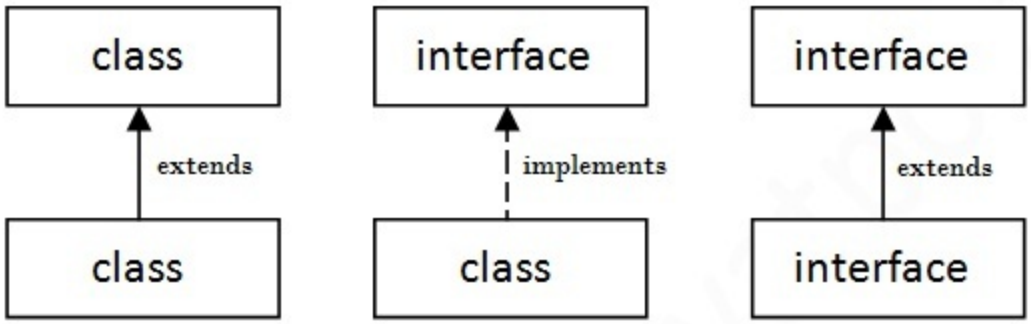
The java compiler adds public and abstract keywords before the interface method and public, static and final keywords before data members.

In other words, Interface fields are public, static and final bydefault, and methods are public and abstract.



Understanding relationship between classes and interfaces

As shown in the figure given below, a class extends another class, an interface extends another interface but a **class implements an interface**.



Simple example of Java interface

In this example, Printable interface have only one method, its implementation is provided in the A class.

```
1.  interface printable{
2.  void print();
3.  }
4.
5.  class A6 implements printable{
6.  public void print(){System.out.println("Hello");}
7.
8.  public static void main(String args[]){
9.  A6 obj = new A6();
10. obj.print();
11.  }
12. }
```

Output:Hello

Multiple inheritance in Java by interface

If a class implements multiple interfaces, or an interface extends multiple interfaces i.e. known as multiple inheritance.

Multiple Inheritance in Java

```
1.  interface Printable{
2.  void print();
3.  }
4.
5.  interface Showable{
6.  void show();
7.  }
8.
9.  class A7 implements Printable,Showable{
10.
11. public void print(){System.out.println("Hello");}
12. public void show(){System.out.println("Welcome");}
13.
14. public static void main(String args[]){
15. A7 obj = new A7();
16. obj.print();
17. obj.show();
18.  }
19. }
```

```
Output:Hello
    Welcome
```

Q) Multiple inheritance is not supported through class in java but it is possible by interface, why?

As we have explained in the inheritance chapter, multiple inheritance is not supported in case

of class. But it is supported in case of interface because there is no ambiguity as implementation is provided by the implementation class. For example:

```
1.  interface Printable{
2.  void print();
3.  }
4.
5.  interface Showable{
6.  void print();
7.  }
8.
9.  class testinterface1 implements Printable,Showable{
10.
11. public void print(){System.out.println("Hello");}
12.
13. public static void main(String args[]){
14. testinterface1 obj = new testinterface1();
15. obj.print();
16.  }
17. }
```

```
Hello
```

As you can see in the above example, Printable and Showable interface have same methods but its implementation is provided by class A, so there is no ambiguity.

Interface inheritance

A class implements interface but one interface extends another interface .

```
1.  interface Printable{
2.  void print();
3.  }
4.  interface Showable extends Printable{
5.  void show();
6.  }
7.  class Testinterface2 implements Showable{
8.
9.  public void print(){System.out.println("Hello");}
10. public void show(){System.out.println("Welcome");}
11.
12. public static void main(String args[]){
13. Testinterface2 obj = new Testinterface2();
14. obj.print();
```

```
15.         obj.show();
16.          }
17.         }
```

Output:

```
Hello
Welcome
```

Q) What is marker or tagged interface?

An interface that have no member is known as marker or tagged interface. For example: Serializable, Cloneable, Remote etc. They are used to provide some essential information to the JVM so that JVM may perform some useful operation.

```
1.          //How Serializable interface is written?
2.          public interface Serializable{
3.          }
```

Nested Interface in Java

Note: An interface can have another interface i.e. known as nested interface. For example:

```
1.          interface printable{
2.           void print();
3.           interface MessagePrintable{
4.             void msg();
5.           }
6.          }
```

An interface is a collection of abstract methods. A class implements an interface, thereby inheriting the abstract methods of the interface.

An interface is not a class. Writing an interface is similar to writing a class, but they are two different concepts. A class describes the attributes and behaviors of an object. An interface contains behaviors that a class implements.

Unless the class that implements the interface is abstract, all the methods of the interface need to be defined in the class.

An interface is similar to a class in the following ways:

- An interface can contain any number of methods.

- An interface is written in a file with a **.java** extension, with the name of the interface matching the name of the file.
- The bytecode of an interface appears in a **.class** file.
- Interfaces appear in packages, and their corresponding bytecode file must be in a directory structure that matches the package name.

However, an interface is different from a class in several ways, including:

- You cannot instantiate an interface.
- An interface does not contain any constructors.
- All of the methods in an interface are abstract.
- An interface cannot contain instance fields. The only fields that can appear in an interface must be declared both static and final.
- An interface is not extended by a class; it is implemented by a class.
- An interface can extend multiple interfaces.

Declaring Interfaces:

The **interface** keyword is used to declare an interface. Here is a simple example to declare an interface:

Example:

Let us look at an example that depicts encapsulation:

```
/* File name : NameOfInterface.java */
import java.lang.*;
//Any number of import statements

public interface NameOfInterface
{
  //Any number of final, static fields
  //Any number of abstract method declarations\
}
```

Interfaces have the following properties:

- An interface is implicitly abstract. You do not need to use the **abstract** keyword when declaring an interface.

- Each method in an interface is also implicitly abstract, so the abstract keyword is not needed.
- Methods in an interface are implicitly public.

Example:

```
/* File name : Animal.java */
interface Animal {

  public void eat();
  public void travel();
}
```

Implementing Interfaces:

When a class implements an interface, you can think of the class as signing a contract, agreeing to perform the specific behaviors of the interface. If a class does not perform all the behaviors of the interface, the class must declare itself as abstract.

A class uses the **implements** keyword to implement an interface. The implements keyword appears in the class declaration following the extends portion of the declaration.

```
/* File name : MammalInt.java */
public class MammalInt implements Animal{

  public void eat(){
    System.out.println("Mammal eats");
  }

  public void travel(){
    System.out.println("Mammal travels");
  }

  public int noOfLegs(){
    return 0;
  }

  public static void main(String args[]){
    MammalInt m = new MammalInt();
    m.eat();
    m.travel();
  }
}
```

This would produce the following result:

```
Mammal eats
Mammal travels
```

When overriding methods defined in interfaces there are several rules to be followed:

- Checked exceptions should not be declared on implementation methods other than the ones declared by the interface method or subclasses of those declared by the interface method.
- The signature of the interface method and the same return type or subtype should be maintained when overriding the methods.
- An implementation class itself can be abstract and if so interface methods need not be implemented.

When implementation interfaces there are several rules:

- A class can implement more than one interface at a time.
- A class can extend only one class, but implement many interfaces.
- An interface can extend another interface, similarly to the way that a class can extend another class.

Extending Interfaces:

An interface can extend another interface, similarly to the way that a class can extend another class. The **extends** keyword is used to extend an interface, and the child interface inherits the methods of the parent interface.

The following Sports interface is extended by Hockey and Football interfaces.

```
//Filename: Sports.java
public interface Sports
{
  public void setHomeTeam(String name);
  public void setVisitingTeam(String name);
}

//Filename: Football.java
public interface Football extends Sports
{
  public void homeTeamScored(int points);
  public void visitingTeamScored(int points);
  public void endOfQuarter(int quarter);
}
```

```
//Filename: Hockey.java
public interface Hockey extends Sports
{
  public void homeGoalScored();
  public void visitingGoalScored();
  public void endOfPeriod(int period);
  public void overtimePeriod(int ot);
}
```

The Hockey interface has four methods, but it inherits two from Sports; thus, a class that implements Hockey needs to implement all six methods. Similarly, a class that implements Football needs to define the three methods from Football and the two methods from Sports.

Extending Multiple Interfaces:

A Java class can only extend one parent class. Multiple inheritance is not allowed. Interfaces are not classes, however, and an interface can extend more than one parent interface.

The extends keyword is used once, and the parent interfaces are declared in a comma-separated list.

For example, if the Hockey interface extended both Sports and Event, it would be declared as:

```
public interface Hockey extends Sports, Event
```

Tagging Interfaces:

The most common use of extending interfaces occurs when the parent interface does not contain any methods. For example, the MouseListener interface in the java.awt.event package extended java.util.EventListener, which is defined as:

```
package java.util;
public interface EventListener
{}
```

An interface with no methods in it is referred to as a **tagging** interface. There are two basic design purposes of tagging interfaces:

Creates a common parent: As with the EventListener interface, which is extended by dozens of other interfaces in the Java API, you can use a tagging interface to create a common parent among a group of interfaces. For example, when an interface extends EventListener, the JVM knows that this particular interface is going to be used in an event delegation scenario.

Adds a data type to a class: This situation is where the term tagging comes from. A class that implements a tagging interface does not need to define any methods (since the interface does not have any), but the class becomes an interface type through polymorphism.

Difference between abstract class and interface

Abstract class and interface both are used to achieve abstraction where we can declare the abstract methods. Abstract class and interface both can't be instantiated.

But there are many differences between abstract class and interface that are given below.

Abstract class	Interface
1) Abstract class can **have abstract and non-abstract** methods.	Interface can have **only abstract** methods.
2) Abstract class **doesn't support multiple inheritance**.	Interface **supports multiple inheritance**.
3) Abstract class **can have final, non-final, static and non-static variables**.	Interface has **only static and final variables**.
4) Abstract class **can have static methods, main method and constructor**.	Interface **can't have static methods, main method or constructor**.
5) Abstract class **can provide the implementation of interface**.	Interface **can't provide the implementation of abstract class**.
6) The **abstract keyword** is used to declare abstract class.	The **interface keyword** is used to declare interface.

7) **Example:**
public class Shape{
public abstract void draw();
}
	Example:
	public interface Drawable{
	void draw();
	}

Simply, abstract class achieves partial abstraction (0 to 100%) whereas interface achieves fully abstraction (100%).

Example of abstract class and interface in Java

Let's see a simple example where we are using interface and abstract class both.

```
1.      //Creating interface that has 4 methods
2.      interface A{
3.      void a();//by default, public and abstract
4.      void b();
```

```
5.          void c();
6.          void d();
7.          }
8.
9.          //Creating abstract class that provides the implementation of one method of A interface
10.         abstract class B implements A{
11.         public void c(){System.out.println("I am C");}
12.         }
13.
14.         //Creating subclass of abstract class, now we need to provide the implementation of rest o
    f the methods
15.         class M extends B{
16.         public void a(){System.out.println("I am a");}
17.         public void b(){System.out.println("I am b");}
18.         public void d(){System.out.println("I am d");}
19.         }
20.
21.         //Creating a test class that calls the methods of A interface
22.         class Test5{
23.         public static void main(String args[]){
24.         A a=new M();
25.         a.a();
26.         a.b();
27.         a.c();
28.         a.d();
29.         }}
```

Output:

```
I am a
I am b
I am c
I am d
```

Java static keyword

The **static keyword** in java is used for memory management mainly. We can apply java static keyword with variables, methods, blocks and nested class. The static keyword belongs to the class than instance of the class.

The static can be:

1. variable (also known as class variable)
2. method (also known as class method)
3. block
4. nested class

1) Java static variable

If you declare any variable as static, it is known static variable.

- The static variable can be used to refer the common property of all objects (that is not unique for each object) e.g. company name of employees,college name of students etc.
- The static variable gets memory only once in class area at the time of class loading.

Advantage of static variable

It makes your program **memory efficient** (i.e it saves memory).

Understanding problem without static variable

```
1.  class Student{
2.      int rollno;
3.      String name;
4.      String college="ITS";
5.  }
```

Suppose there are 500 students in my college, now all instance data members will get memory each time when object is created.All student have its unique rollno and name so instance data member is good.Here, college refers to the common property of all objects.If we make it static,this field will get memory only once.

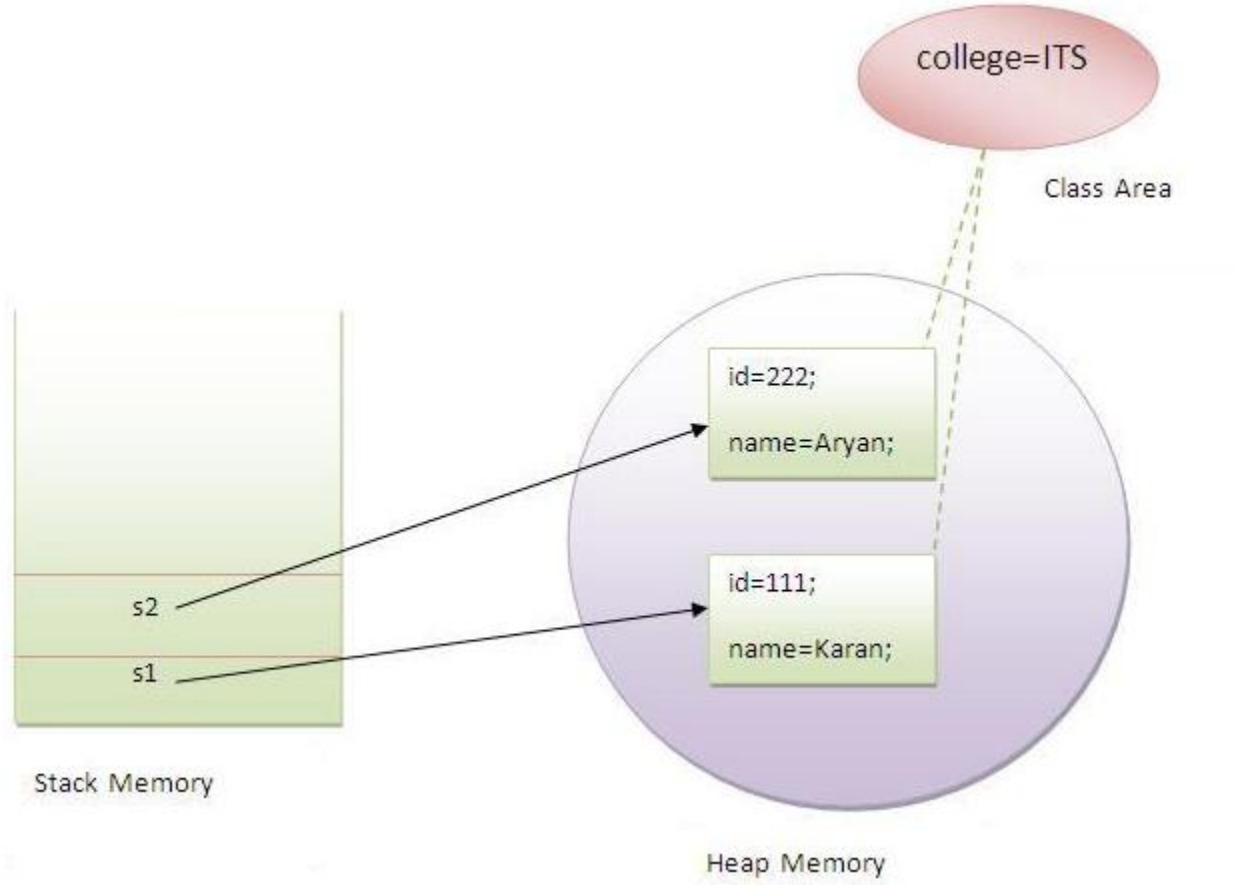
Java static property is shared to all objects.

Example of static variable

```
1.  //Program of static variable
2.
3.  class Student8{
4.    int rollno;
5.    String name;
6.    static String college ="ITS";
```

```
7.
8.          Student8(int r,String n){
9.          rollno = r;
10.         name = n;
11.         }
12.        void display (){System.out.println(rollno+" "+name+" "+college);}
13.
14.        public static void main(String args[]){
15.        Student8 s1 = new Student8(111,"Karan");
16.        Student8 s2 = new Student8(222,"Aryan");
17.
18.        s1.display();
19.        s2.display();
20.        }
21.       }
```

Output:111 Karan ITS
222 Aryan ITS



Program of counter without static variable

In this example, we have created an instance variable named count which is incremented in the constructor. Since instance variable gets the memory at the time of object creation, each object will have the copy of the instance variable, if it is incremented, it won't reflect to other objects. So each objects will have the value 1 in the count variable.

```
1.      class Counter{
2.      int count=0;//will get memory when instance is created
3.
4.      Counter(){
5.      count++;
6.      System.out.println(count);
7.      }
8.
9.      public static void main(String args[]){
10.
11.     Counter c1=new Counter();
12.     Counter c2=new Counter();
13.     Counter c3=new Counter();
14.
15.      }
16.     }
```

```
Output:1
       1
       1
```

Program of counter by static variable

As we have mentioned above, static variable will get the memory only once, if any object changes the value of the static variable, it will retain its value.

```
1.      class Counter2{
2.      static int count=0;//will get memory only once and retain its value
3.
4.      Counter2(){
5.      count++;
6.      System.out.println(count);
7.      }
8.
9.      public static void main(String args[]){
10.
11.     Counter2 c1=new Counter2();
```

```
12.         Counter2 c2=new Counter2();
13.         Counter2 c3=new Counter2();
14.
15.          }
16.         }
```

```
Output:1
       2
       3
```

2) Java static method

If you apply static keyword with any method, it is known as static method.

- A static method belongs to the class rather than object of a class.
- A static method can be invoked without the need for creating an instance of a class.
- static method can access static data member and can change the value of it.

Example of static method

```
1.          //Program of changing the common property of all objects(static field).
2.
3.          class Student9{
4.              int rollno;
5.              String name;
6.              static String college = "ITS";
7.
8.              static void change(){
9.              college = "BBDIT";
10.             }
11.
12.             Student9(int r, String n){
13.             rollno = r;
14.             name = n;
15.             }
16.
17.             void display (){System.out.println(rollno+" "+name+" "+college);}
18.
19.            public static void main(String args[]){
20.            Student9.change();
21.
```

```
22.             Student9 s1 = new Student9 (111,"Karan");
23.             Student9 s2 = new Student9 (222,"Aryan");
24.             Student9 s3 = new Student9 (333,"Sonoo");
25.
26.             s1.display();
27.             s2.display();
28.             s3.display();
29.             }
30.         }
```

Output:111 Karan BBDIT
222 Aryan BBDIT
333 Sonoo BBDIT

Another example of static method that performs normal calculation

```
1.          //Program to get cube of a given number by static method
2.
3.          class Calculate{
4.            static int cube(int x){
5.            return x*x*x;
6.            }
7.
8.            public static void main(String args[]){
9.            int result=Calculate.cube(5);
10.           System.out.println(result);
11.           }
12.         }
```

Output:125

Restrictions for static method

There are two main restrictions for the static method. They are:

1. The static method can not use non static data member or call non-static method directly.
2. this and super cannot be used in static context.

```
1.          class A{
2.           int a=40;//non static
3.
4.           public static void main(String args[]){
5.            System.out.println(a);
6.           }
7.          }
```

Output:Compile Time Error

Q) why java main method is static?

Ans) because object is not required to call static method if it were non-static method, jvm create object first then call main() method that will lead the problem of extra memory allocation.

3) Java static block

- Is used to initialize the static data member.
- It is executed before main method at the time of classloading.

Example of static block

```
1.  class A2{
2.    static{System.out.println("static block is invoked");}
3.    public static void main(String args[]){
4.     System.out.println("Hello main");
5.    }
6.  }
```

Output:static block is invoked
Hello main

Q) Can we execute a program without main() method?

Ans) Yes, one of the way is static block but in previous version of JDK not in JDK 1.7.

```
1.  class A3{
2.    static{
3.    System.out.println("static block is invoked");
4.    System.exit(0);
5.    }
6.  }
```

Output:static block is invoked (if not JDK7)

In JDK7 and above, output will be:

```
Output:Error: Main method not found in class A3, please define the main method as:
public static void main(String[] args)
```

this keyword in java

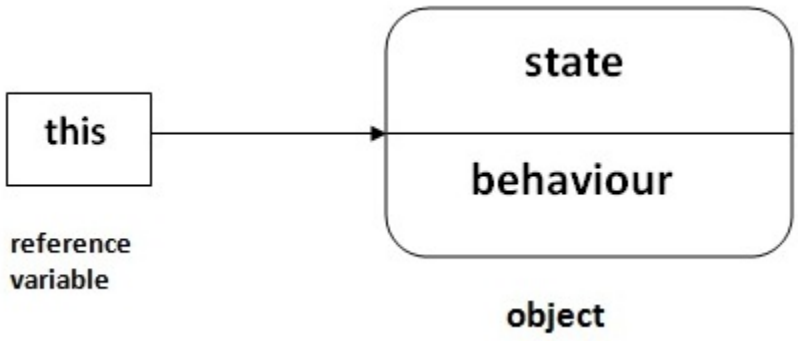
There can be a lot of usage of **java this keyword**. In java, this is a**reference variable** that refers to the current object.

Usage of java this keyword

Here is given the 6 usage of java this keyword.

1. this keyword can be used to refer current class instance variable.
2. this() can be used to invoke current class constructor.
3. this keyword can be used to invoke current class method (implicitly)
4. this can be passed as an argument in the method call.
5. this can be passed as argument in the constructor call.
6. this keyword can also be used to return the current class instance.

Suggestion: If you are beginner to java, lookup only two usage of this keyword.



1) The this keyword can be used to refer current class instance variable.

If there is ambiguity between the instance variable and parameter, this keyword resolves the problem of ambiguity.

Understanding the problem without this keyword

Let's understand the problem if we don't use this keyword by the example given below:

```
1.      class Student10{
2.         int id;
3.         String name;
4.
5.         Student10(int id,String name){
6.         id = id;
7.         name = name;
8.         }
9.         void display(){System.out.println(id+" "+name);}
10.
11.        public static void main(String args[]){
12.        Student10 s1 = new Student10(111,"Karan");
13.        Student10 s2 = new Student10(321,"Aryan");
14.        s1.display();
15.        s2.display();
16.        }
17.     }
```

```
Output:0 null
       0 null
```

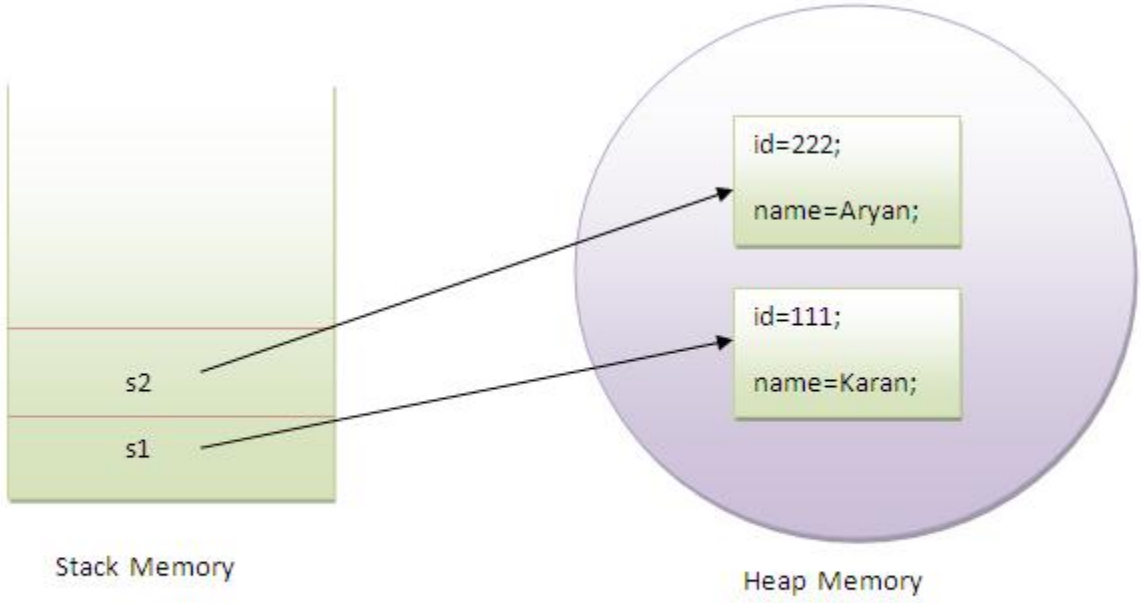
In the above example, parameter (formal arguments) and instance variables are same that is why we are using this keyword to distinguish between local variable and instance variable.

Solution of the above problem by this keyword

```
1.      //example of this keyword
2.      class Student11{
3.         int id;
4.         String name;
5.
6.         Student11(int id,String name){
7.         this.id = id;
8.         this.name = name;
9.         }
10.        void display(){System.out.println(id+" "+name);}
11.        public static void main(String args[]){
12.        Student11 s1 = new Student11(111,"Karan");
```

```
13.             Student11 s2 = new Student11(222,"Aryan");
14.             s1.display();
15.             s2.display();
16.         }
17.         }
```

Output111 Karan
222 Aryan



If local variables(formal arguments) and instance variables are different, there is no need to use this keyword like in the following program:

Program where this keyword is not required

```
1.          class Student12{
2.              int id;
3.              String name;
4.
5.              Student12(int i,String n){
6.              id = i;
7.              name = n;
8.              }
9.              void display(){System.out.println(id+" "+name);}
10.             public static void main(String args[]){
```

```
11.             Student12 e1 = new Student12(111,"karan");
12.             Student12 e2 = new Student12(222,"Aryan");
13.             e1.display();
14.             e2.display();
15.         }
16.         }
```

Output:111 Karan
222 Aryan

2) this() can be used to invoked current class constructor.

The this() constructor call can be used to invoke the current class constructor (constructor chaining). This approach is better if you have many constructors in the class and want to reuse that constructor.

```
1.          //Program of this() constructor call (constructor chaining)
2.
3.          class Student13{
4.             int id;
5.             String name;
6.             Student13(){System.out.println("default constructor is invoked");}
7.
8.             Student13(int id,String name){
9.             this ();//it is used to invoked current class constructor.
10.            this.id = id;
11.            this.name = name;
12.            }
13.            void display(){System.out.println(id+" "+name);}
14.
15.            public static void main(String args[]){
16.            Student13 e1 = new Student13(111,"karan");
17.            Student13 e2 = new Student13(222,"Aryan");
18.            e1.display();
19.            e2.display();
20.           }
21.         }
```

Output:
default constructor is invoked
default constructor is invoked
111 Karan
222 Aryan

Where to use this() constructor call?

The this() constructor call should be used to reuse the constructor in the constructor. It maintains the chain between the constructors i.e. it is used for constructor chaining. Let's see the example given below that displays the actual use of this keyword.

```
1.      class Student14{
2.         int id;
3.         String name;
4.         String city;
5.
6.         Student14(int id,String name){
7.         this.id = id;
8.         this.name = name;
9.         }
10.        Student14(int id,String name,String city){
11.        this(id,name);//now no need to initialize id and name
12.        this.city=city;
13.        }
14.        void display(){System.out.println(id+" "+name+" "+city);}
15.
16.        public static void main(String args[]){
17.        Student14 e1 = new Student14(111,"karan");
18.        Student14 e2 = new Student14(222,"Aryan","delhi");
19.        e1.display();
20.        e2.display();
21.       }
22.     }
```

Output:111 Karan null
222 Aryan delhi

Rule: Call to this() must be the first statement in constructor.

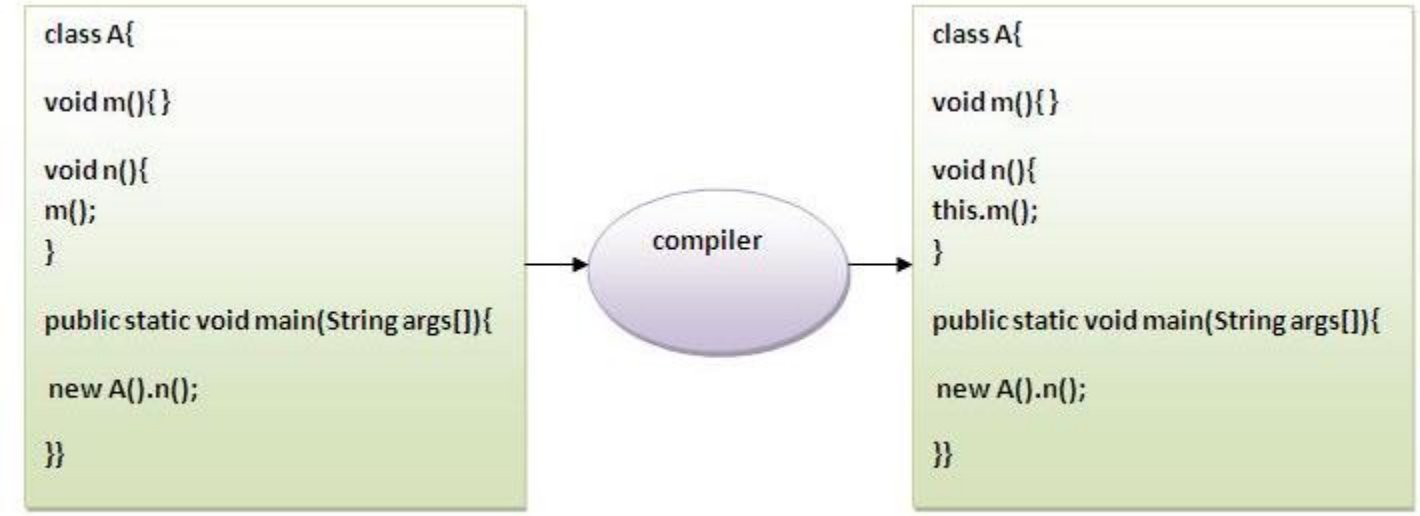
```
1.      class Student15{
2.         int id;
3.         String name;
4.         Student15(){System.out.println("default constructor is invoked");}
5.
6.         Student15(int id,String name){
7.         id = id;
8.         name = name;
9.         this ();//must be the first statement
10.        }
11.        void display(){System.out.println(id+" "+name);}
12.
```

```
13.         public static void main(String args[]){
14.         Student15 e1 = new Student15(111,"karan");
15.         Student15 e2 = new Student15(222,"Aryan");
16.         e1.display();
17.         e2.display();
18.        }
19.      }
```

Output: Compile Time Error

3)The this keyword can be used to invoke current class method (implicitly).

You may invoke the method of the current class by using the this keyword. If you don't use the this keyword, compiler automatically adds this keyword while invoking the method. Let's see the example




```
1.      class S{
2.        void m(){
3.        System.out.println("method is invoked");
4.        }
5.        void n(){
6.        this.m();//no need because compiler does it for you.
7.        }
8.        void p(){
9.        n();//complier will add this to invoke n() method as this.n()
10.       }
11.       public static void main(String args[]){
12.       S s1 = new S();
13.       s1.p();
```

```
14.         }
15.       }
```
Output: method is invoked

4) this keyword can be passed as an argument in the method.

The this keyword can also be passed as an argument in the method. It is mainly used in the event handling. Let's see the example:

```
1.        class S2{
2.          void m(S2 obj){
3.          System.out.println("method is invoked");
4.          }
5.          void p(){
6.          m(this);
7.          }
8.
9.          public static void main(String args[]){
10.         S2 s1 = new S2();
11.         s1.p();
12.         }
13.       }
```
Output: method is invoked

Application of this that can be passed as an argument:

In event handling (or) in a situation where we have to provide reference of a class to another one.

5) The this keyword can be passed as argument in the constructor call.

We can pass the this keyword in the constructor also. It is useful if we have to use one object in multiple classes. Let's see the example:

```
1.        class B{
2.          A4 obj;
3.          B(A4 obj){
4.            this.obj=obj;
5.          }
6.          void display(){
7.            System.out.println(obj.data);//using data member of A4 class
```

```
8.          }
9.        }
10.
11.       class A4{
12.        int data=10;
13.        A4(){
14.         B b=new B(this);
15.         b.display();
16.        }
17.        public static void main(String args[]){
18.         A4 a=new A4();
19.        }
20.       }
```

Output:10

6) The this keyword can be used to return current class instance.

We can return the this keyword as an statement from the method. In such case, return type of the method must be the class type (non-primitive). Let's see the example:

Syntax of this that can be returned as a statement

```
1.        return_type method_name(){
2.        return this;
3.        }
```

Example of this keyword that you return as a statement from the method

```
1.        class A{
2.        A getA(){
3.        return this;
4.        }
5.        void msg(){System.out.println("Hello java");}
6.        }
7.
8.        class Test1{
9.        public static void main(String args[]){
10.       new A().getA().msg();
11.       }
12.       }
```

Output:Hello java

Proving this keyword

Let's prove that this keyword refers to the current class instance variable. In this program, we

are printing the reference variable and this, output of both variables are same.

```
1.      class A5{
2.      void m(){
3.      System.out.println(this);//prints same reference ID
4.      }
5.
6.      public static void main(String args[]){
7.      A5 obj=new A5();
8.      System.out.println(obj);//prints the reference ID
9.
10.     obj.m();
11.     }
12.     }
```

```
Output:A5@22b3ea59
       A5@22b3ea59
```

- **Instance initializer block(init block)**

Instance Initializer block is used to initialize the instance data member. It run each time when object of the class is created.

The initialization of the instance variable can be directly but there can be performed extra operations while initializing the instance variable in the instance initializer block.

Que) What is the use of instance initializer block while we can directly assign a value in instance data member? For example:

```
1.      class Bike{
2.         int speed=100;
3.      }
```

Why use instance initializer block?

Suppose I have to perform some operations while assigning value to instance data member e.g. a for loop to fill a complex array or error handling etc.

Example of instance initializer block

Let's see the simple example of instance initializer block the performs initialization.

```
1.      class Bike7{
2.         int speed;
3.
4.         Bike7(){System.out.println("speed is "+speed);}
5.
6.         {speed=100;}
7.
8.         public static void main(String args[]){
9.         Bike7 b1=new Bike7();
10.        Bike7 b2=new Bike7();
11.        }
12.     }
```

```
Output:speed is 100
       speed is 100
```

There are three places in java where you can perform operations:

1. method
2. constructor
3. block

What is invoked firstly instance initializer block or constructor?

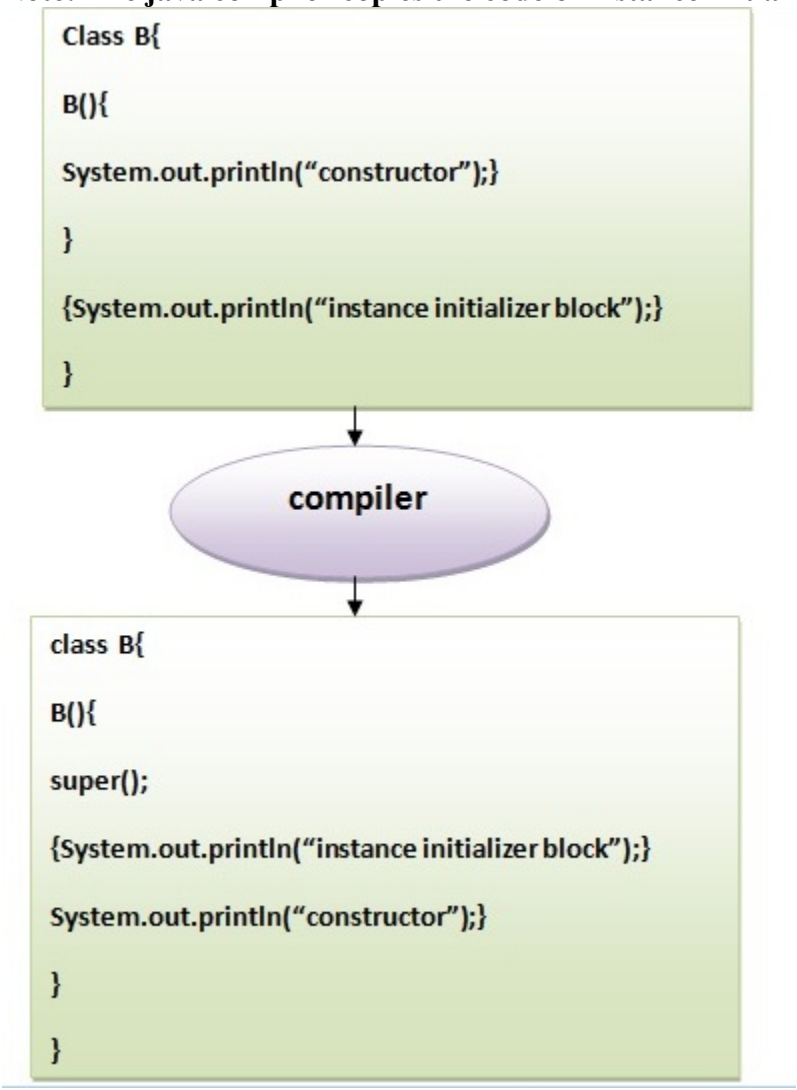
```
1.      class Bike8{
2.         int speed;
3.
4.         Bike8(){System.out.println("constructor is invoked");}
5.
6.         {System.out.println("instance initializer block invoked");}
7.
8.         public static void main(String args[]){
9.         Bike8 b1=new Bike8();
10.        Bike8 b2=new Bike8();
11.        }
12.     }
```

```
Output: instance initializer block invoked
        constructor is invoked
```

```
instance initializer block invoked
constructor is invoked
```

In the above example, it seems that instance initializer block is firstly invoked but NO. Instance intializer block is invoked at the time of object creation. The java compiler copies the instance initializer block in the constructor after the first statement super(). So firstly, constructor is invoked. Let's understand it by the figure given below:

Note: The java compiler copies the code of instance initializer block in every constructor.



Rules for instance initializer block :

There are mainly three rules for the instance initializer block. They are as follows:

1. The instance initializer block is created when instance of the class is created.
2. The instance initializer block is invoked after the parent class constructor is invoked (i.e. after super() constructor call).
3. The instance initializer block comes in the order in which they appear.

Program of instance initializer block that is invoked after super()

```
1.  class A{
2.  A(){
3.  System.out.println("parent class constructor invoked");
4.  }
5.  }
6.  class B2 extends A{
7.  B2(){
8.  super();
9.  System.out.println("child class constructor invoked");
10. }
11.
12. {System.out.println("instance initializer block is invoked");}
13.
14. public static void main(String args[]){
15. B2 b=new B2();
16. }
17. }
```

```
Output:parent class constructor invoked
       instance initializer block is invoked
       child class constructor invoked
```

Another example of instance block

```
1.  class A{
2.  A(){
3.  System.out.println("parent class constructor invoked");
4.  }
5.  }
6.
7.  class B3 extends A{
8.  B3(){
9.  super();
```

```
10.         System.out.println("child class constructor invoked");
11.         }
12.
13.         B3(int a){
14.         super();
15.         System.out.println("child class constructor invoked "+a);
16.         }
17.
18.         {System.out.println("instance initializer block is invoked");}
19.
20.         public static void main(String args[]){
21.         B3 b1=new B3();
22.         B3 b2=new B3(10);
23.         }
24.         }
```

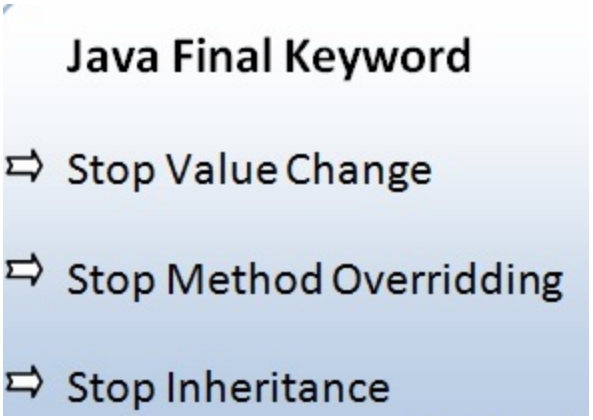
```
Output: parent class constructor invoked
        instance initializer block is invoked
        child class constructor invoked
        parent class constructor invoked
        instance initializer block is invoked
        child class constructor invoked 10
```

Final Keyword In Java

The **final keyword** in java is used to restrict the user. The java final keyword can be used in many context. Final can be:

1. variable
2. method
3. class

The final keyword can be applied with the variables, a final variable that have no value it is called blank final variable or uninitialized final variable. It can be initialized in the constructor only. The blank final variable can be static also which will be initialized in the static block only. We will have detailed learning of these. Let's first learn the basics of final keyword.



1) Java final variable

If you make any variable as final, you cannot change the value of final variable(It will be constant).

Example of final variable

There is a final variable speedlimit, we are going to change the value of this variable, but It can't be changed because final variable once assigned a value can never be changed.

```
1.  class Bike9{
2.   final int speedlimit=90;//final variable
3.   void run(){
4.    speedlimit=400;
5.   }
6.   public static void main(String args[]){
7.   Bike9 obj=new  Bike9();
8.   obj.run();
9.   }
10. }//end of class
```

Output:Compile Time Error

2) Java final method

If you make any method as final, you cannot override it.

Example of final method

```
1.  class Bike{
```

```
2.          final void run(){System.out.println("running");}
3.        }
4.
5.        class Honda extends Bike{
6.          void run(){System.out.println("running safely with 100kmph");}
7.
8.          public static void main(String args[]){
9.          Honda honda= new Honda();
10.         honda.run();
11.         }
12.       }
```

Output:Compile Time Error

3) Java final class

If you make any class as final, you cannot extend it.

Example of final class

```
1.        final class Bike{}
2.
3.        class Honda1 extends Bike{
4.          void run(){System.out.println("running safely with 100kmph");}
5.
6.          public static void main(String args[]){
7.          Honda1 honda= new Honda();
8.          honda.run();
9.          }
10.       }
```

Output:Compile Time Error

Q) Is final method inherited?

Ans) Yes, final method is inherited but you cannot override it. For Example:

```
1.        class Bike{
2.          final void run(){System.out.println("running...");}
3.        }
4.        class Honda2 extends Bike{
5.          public static void main(String args[]){
6.           new Honda2().run();
```

```
7.          }
8.      }
```
Output:running...

Q) What is blank or uninitialized final variable?

A final variable that is not initialized at the time of declaration is known as blank final variable.

If you want to create a variable that is initialized at the time of creating object and once initialized may not be changed, it is useful. For example PAN CARD number of an employee.

It can be initialized only in constructor.

Example of blank final variable

```
1.      class Student{
2.      int id;
3.      String name;
4.      final String PAN_CARD_NUMBER;
5.      ...
6.      }
```

Que) Can we initialize blank final variable?

Yes, but only in constructor. For example:

```
1.      class Bike10{
2.        final int speedlimit;//blank final variable
3.
4.        Bike10(){
5.        speedlimit=70;
6.        System.out.println(speedlimit);
7.        }
8.
9.        public static void main(String args[]){
10.         new Bike10();
11.       }
12.     }
```
Output:70

Static blank final variable

A static final variable that is not initialized at the time of declaration is known as static blank final variable. It can be initialized only in static block.

Example of static blank final variable

```
1.  class A{
2.    static final int data;//static blank final variable
3.    static{ data=50;}
4.    public static void main(String args[]){
5.      System.out.println(A.data);
6.    }
7.  }
```

Q) What is final parameter?

If you declare any parameter as final, you cannot change the value of it.

```
1.  class Bike11{
2.    int cube(final int n){
3.     n=n+2;//can't be changed as n is final
4.     n*n*n;
5.    }
6.    public static void main(String args[]){
7.      Bike11 b=new Bike11();
8.      b.cube(5);
9.    }
10. }
```

Output: Compile Time Error

Q) Can we declare a constructor final?

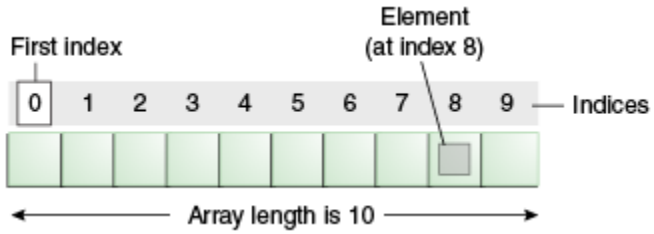
No, because constructor is never inherited.

Java Array

Array is a collection of similar type of elements that have contiguous memory location.

Java array is an object the contains elements of similar data type. It is a data structure where we store similar elements. We can store only fixed set of elements in a java array.

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



Advantage of Java Array

- **Code Optimization:** It makes the code optimized, we can retrieve or sort the data easily.
- **Random access:** We can get any data located at any index position.

Disadvantage of Java Array

- **Size Limit:** We can store only fixed size of elements in the array. It doesn't grow its size at runtime. To solve this problem, collection framework is used in java.

Types of Array in java

There are two types of array.

- Single Dimensional Array
- Multidimensional Array

Single Dimensional Array in java

Syntax to Declare an Array in java

```
1.  dataType[] arr; (or)
2.  dataType []arr; (or)
3.  dataType arr[];
```

Instantiation of an Array in java

```
1.  arrayRefVar=new datatype[size];
```

Example of single dimensional java array

Let's see the simple example of java array, where we are going to declare, instantiate, initialize and traverse an array.

```
1.      class Testarray{
2.      public static void main(String args[]){
3.
4.      int a[]=new int[5];//declaration and instantiation
5.      a[0]=10;//initialization
6.      a[1]=20;
7.      a[2]=70;
8.      a[3]=40;
9.      a[4]=50;
10.
11.     //printing array
12.     for(int i=0;i<a.length;i++)//length is the property of array
13.     System.out.println(a[i]);
14.
15.     }}
```

Output: 10
20
70
40
50

Declaration, Instantiation and Initialization of Java Array

We can declare, instantiate and initialize the java array together by:

```
1.      int a[]={33,3,4,5};//declaration, instantiation and initialization
```

Let's see the simple example to print this array.

```
1.      class Testarray1{
2.      public static void main(String args[]){
3.
4.      int a[]={33,3,4,5};//declaration, instantiation and initialization
5.
6.      //printing array
7.      for(int i=0;i<a.length;i++)//length is the property of array
8.      System.out.println(a[i]);
9.
10.     }}
```

Output:33

```
3
4
5
```

Passing Array to method in java

We can pass the java array to method so that we can reuse the same logic on any array.

Let's see the simple example to get minimum number of an array using method.

```
1.  class Testarray2{
2.  static void min(int arr[]){
3.  int min=arr[0];
4.  for(int i=1;i<arr.length;i++)
5.   if(min>arr[i])
6.    min=arr[i];
7.
8.  System.out.println(min);
9.  }
10.
11. public static void main(String args[]){
12.
13. int a[]={33,3,4,5};
14. min(a);//passing array to method
15.
16. }}
```

Output:3

Multidimensional Array In Java

In such case, data is stored in row and column based index (also known as matrix form).

Syntax to Declare Multidimensional Array in java

```
1.  dataType[][] arrayRefVar; (or)
2.  dataType [][]arrayRefVar; (or)
3.  dataType arrayRefVar[][]; (or)
4.  dataType []arrayRefVar[];
```

Example to instantiate Multidimensional Array in java

```
1.  int[][] arr=new int[3][3];//3 row and 3 column
```

Example to initialize Multidimensional Array in java

```
1.  arr[0][0]=1;
2.  arr[0][1]=2;
3.  arr[0][2]=3;
4.  arr[1][0]=4;
5.  arr[1][1]=5;
6.  arr[1][2]=6;
7.  arr[2][0]=7;
8.  arr[2][1]=8;
9.  arr[2][2]=9;
```

Example of Multidimensional java array

Let's see the simple example to declare, instantiate, initialize and print the 2Dimensional array.

```
1.  class Testarray3{
2.  public static void main(String args[]){
3.
4.  //declaring and initializing 2D array
5.  int arr[][]={{1,2,3},{2,4,5},{4,4,5}};
6.
7.  //printing 2D array
8.  for(int i=0;i<3;i++){
9.   for(int j=0;j<3;j++){
10.    System.out.print(arr[i][j]+" ");
11.   }
12.   System.out.println();
13.  }
14.
15.  }}
```

```
Output:1 2 3
       2 4 5
       4 4 5
```

What is class name of java array?

In java, array is an object. For array object, an proxy class is created whose name can be obtained by getClass().getName() method on the object.

```
1.  class Testarray4{
2.  public static void main(String args[]){
3.
4.  int arr[]={4,4,5};
```

```
5.
6.          Class c=arr.getClass();
7.          String name=c.getName();
8.
9.          System.out.println(name);
10.
11.         }}
```

Output:I

Copying a java array

We can copy an array to another by the arraycopy method of System class.

Syntax of arraycopy method

```
1.          public static void arraycopy(
2.          Object src, int srcPos,Object dest, int destPos, int length
3.          )
```

Example of arraycopy method

```
1.          class TestArrayCopyDemo {
2.            public static void main(String[] args) {
3.              char[] copyFrom = { 'd', 'e', 'c', 'a', 'f', 'f', 'e',
4.                    'i', 'n', 'a', 't', 'e', 'd' };
5.              char[] copyTo = new char[7];
6.
7.              System.arraycopy(copyFrom, 2, copyTo, 0, 7);
8.              System.out.println(new String(copyTo));
9.            }
10.         }
```

Output: caffein

Addition of 2 matrices in java

Let's see a simple example that adds two matrices.

```
1.          class Testarray5{
2.          public static void main(String args[]){
3.          //creating two matrices
4.          int a[][]={{1,3,4},{3,4,5}};
5.          int b[][]={{1,3,4},{3,4,5}};
6.
7.          //creating another matrix to store the sum of two matrices
```

```
8.          int c[][]=new int[2][3];
9.
10.         //adding and printing addition of 2 matrices
11.         for(int i=0;i<2;i++){
12.         for(int j=0;j<3;j++){
13.         c[i][j]=a[i][j]+b[i][j];
14.         System.out.print(c[i][j]+" ");
15.         }
16.         System.out.println();//new line
17.         }
18.
19.         }}
```

Output:2 6 8
6 8 10

Java Package

A **java package** is a group of similar types of classes, interfaces and sub-packages.

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

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

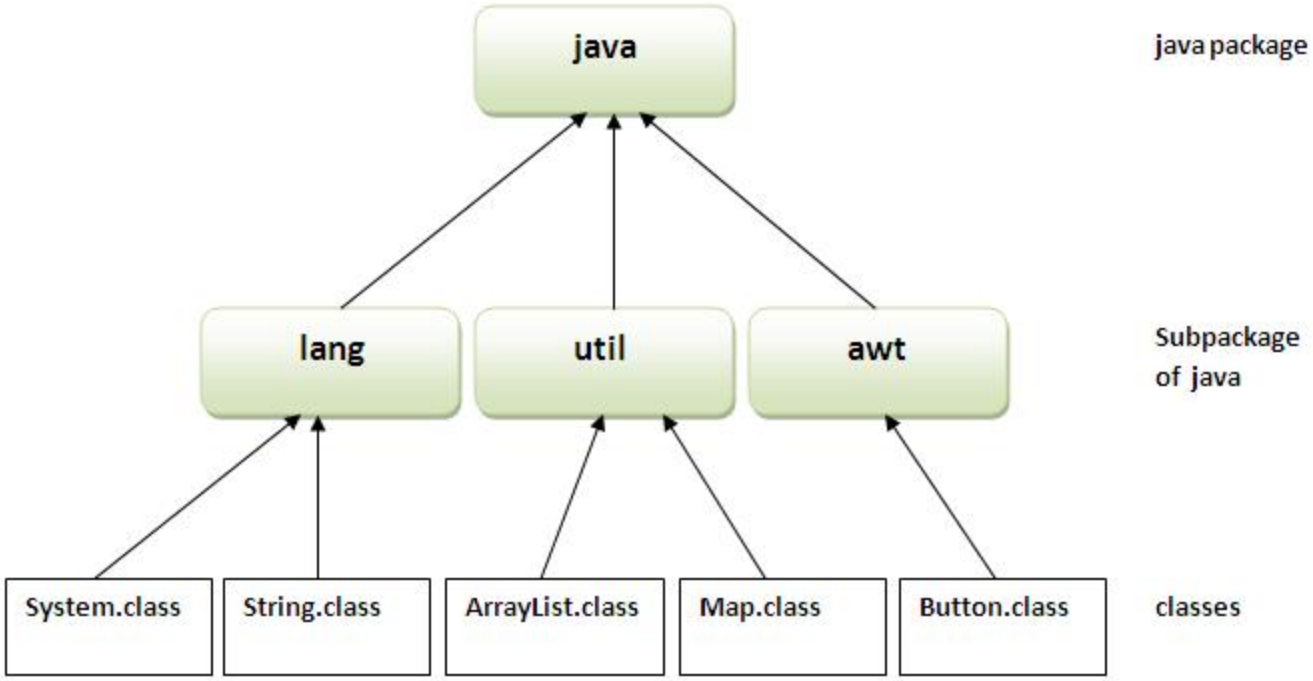
Here, we will have the detailed learning of creating and using user-defined packages.

Advantage of Java Package

1) Java package is used to categorize the classes and interfaces so that they can be easily maintained.

2) Java package provides access protection.

3) Java package removes naming collision.



Simple example of java package

The **package keyword** is used to create a package in java.

```
1.  //save as Simple.java
2.  package mypack;
3.  public class Simple{
4.   public static void main(String args[]){
5.     System.out.println("Welcome to package");
6.    }
7.  }
```

How to compile java package

If you are not using any IDE, you need to follow the **syntax** given below:

```
1.  javac -d directory javafilename
```

For **example**

```
1.  javac -d . Simple.java
```

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

How to run java package program

You need to use fully qualified name e.g. mypack.Simple etc to run the class.

To Compile: javac -d . Simple.java

To Run: java mypack.Simple

Output: Welcome to package

The -d is a switch that tells the compiler where to put the class file i.e. it represents destination. The . represents the current folder.

How to access package from another package?

There are three ways to access the package from outside the package.

1. import package.*;
2. import package.classname;
3. fully qualified name.

1) Using packagename.*

If you use package.* then all the classes and interfaces of this package will be accessible but not subpackages.

The import keyword is used to make the classes and interface of another package accessible to the current package.

Example of package that import the packagename.*

```
1.  //save by A.java
2.
3.  package pack;
4.  public class A{
```

```
5.          public void msg(){System.out.println("Hello");}
6.        }
1.        //save by B.java
2.
3.        package mypack;
4.        import pack.*;
5.
6.        class B{
7.          public static void main(String args[]){
8.           A obj = new A();
9.           obj.msg();
10.         }
11.       }
```

Output:Hello

2) Using packagename.classname

If you import package.classname then only declared class of this package will be accessible.

Example of package by import package.classname

```
1.        //save by A.java
2.
3.        package pack;
4.        public class A{
5.          public void msg(){System.out.println("Hello");}
6.        }
1.        //save by B.java
2.
3.        package mypack;
4.        import pack.A;
5.
6.        class B{
7.          public static void main(String args[]){
8.           A obj = new A();
9.           obj.msg();
10.         }
11.       }
```

Output:Hello

3) Using fully qualified name

If you use fully qualified name then only declared class of this package will be accessible. Now there is no need to import. But you need to use fully qualified name every time when you are accessing the class or interface.

It is generally used when two packages have same class name e.g. java.util and java.sql packages contain Date class.

Example of package by import fully qualified name

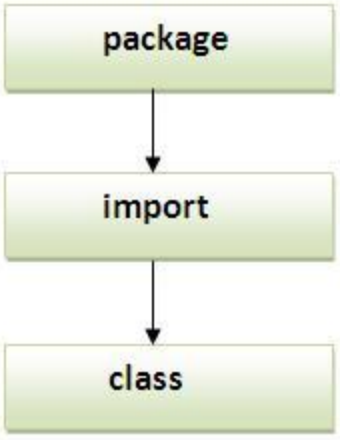
```
1.          //save by A.java
2.
3.          package pack;
4.          public class A{
5.            public void msg(){System.out.println("Hello");}
6.          }
1.          //save by B.java
2.
3.          package mypack;
4.          class B{
5.            public static void main(String args[]){
6.             pack.A obj = new pack.A();//using fully qualified name
7.             obj.msg();
8.            }
9.          }
```

Output:Hello

Note: If you import a package, subpackages will not be imported.

If you import a package, all the classes and interface of that package will be imported excluding the classes and interfaces of the subpackages. Hence, you need to import the subpackage as well.

Note: Sequence of the program must be package then import then class.



Subpackage in java

Package inside the package is called the **subpackage**. It should be created **to categorize the package further**.

Let's take an example, Sun Microsystem has definded a package named java that contains many classes like System, String, Reader, Writer, Socket etc. These classes represent a particular group e.g. Reader and Writer classes are for Input/Output operation, Socket and ServerSocket classes are for networking etc and so on. So, Sun has subcategorized the java package into subpackages such as lang, net, io etc. and put the Input/Output related classes in io package, Server and ServerSocket classes in net packages and so on.

The standard of defining package is domain.company.package e.g. com.javatpoint.bean or org.sssit.dao.

Example of Subpackage

```
1.          package com.brainmentors.core;
2.          class Simple{
3.            public static void main(String args[]){
4.             System.out.println("Hello subpackage");
5.            }
6.          }
```

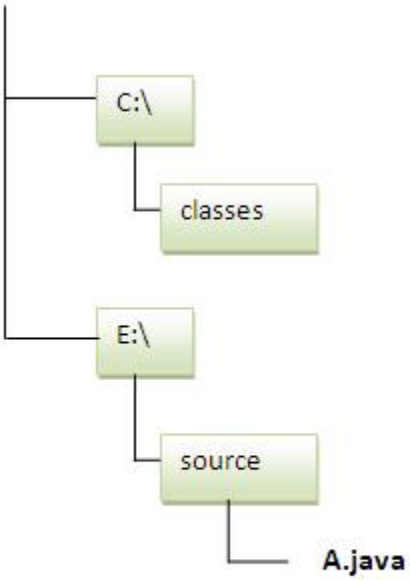
To Compile: javac -d . Simple.java

To Run: java com.brainmentors.core.Simple

Output:Hello subpackage

How to send the class file to another directory or drive?

There is a scenario, I want to put the class file of A.java source file in classes folder of c: drive. For example:




```
1.          //save as Simple.java
2.
3.          package mypack;
4.          public class Simple{
5.           public static void main(String args[]){
6.              System.out.println("Welcome to package");
7.             }
8.          }
```

To Compile:

e:\sources> javac -d c:\classes Simple.java

To Run:

To run this program from e:\source directory, you need to set classpath of the directory where the class file resides.

e:\sources> set classpath=c:\classes;.;

e:\sources> java mypack.Simple

Another way to run this program by -classpath switch of java:

The -classpath switch can be used with javac and java tool.

To run this program from e:\source directory, you can use -classpath switch of java that tells where to look for class file. For example:

e:\sources> java -classpath c:\classes mypack.Simple

Output:Welcome to package

Ways to load the class files or jar files

There are two ways to load the class files temporary and permanent.

- Temporary
  - By setting the classpath in the command prompt
  - By -classpath switch
- Permanent
  - By setting the classpath in the environment variables
  - By creating the jar file, that contains all the class files, and copying the jar file in the jre/lib/ext folder.

Rule: There can be only one public class in a java source file and it must be saved by the public class name.

```
1.  //save as C.java otherwise Compilte Time Error
2.
3.  class A{}
4.  class B{}
5.  public class C{}
```

How to put two public classes in a package?

If you want to put two public classes in a package, have two java source files containing one

public class, but keep the package name same. For example:

```
1.      //save as A.java
2.
3.      package javatpoint;
4.      public class A{}
1.      //save as B.java
2.
3.      package javatpoint;
4.      public class B{}
```

Java String

Java String provides a lot of concepts that can be performed on a string such as compare, concat, equals, split, length, replace, compareTo, intern, substring etc.

In java, string is basically an object that represents sequence of char values.

An array of characters works same as java string. For example:

```
1.      char[] ch={'j','a','v','a','t','p','o','i','n','t'};
2.      String s=new String(ch);
```

is same as:

```
1.      String s="javapython";
```

The java.lang.String class implements *Serializable*, *Comparable* and *CharSequence* interfaces.

The java String is immutable i.e. it cannot be changed but a new instance is created. For mutable class, you can use StringBuffer and StringBuilder class.

What is String in java

Generally, string is a sequence of characters. But in java, string is an object that represents a sequence of characters. String class is used to create string object.

How to create String object?

There are two ways to create String object:

1. By string literal

2. By new keyword

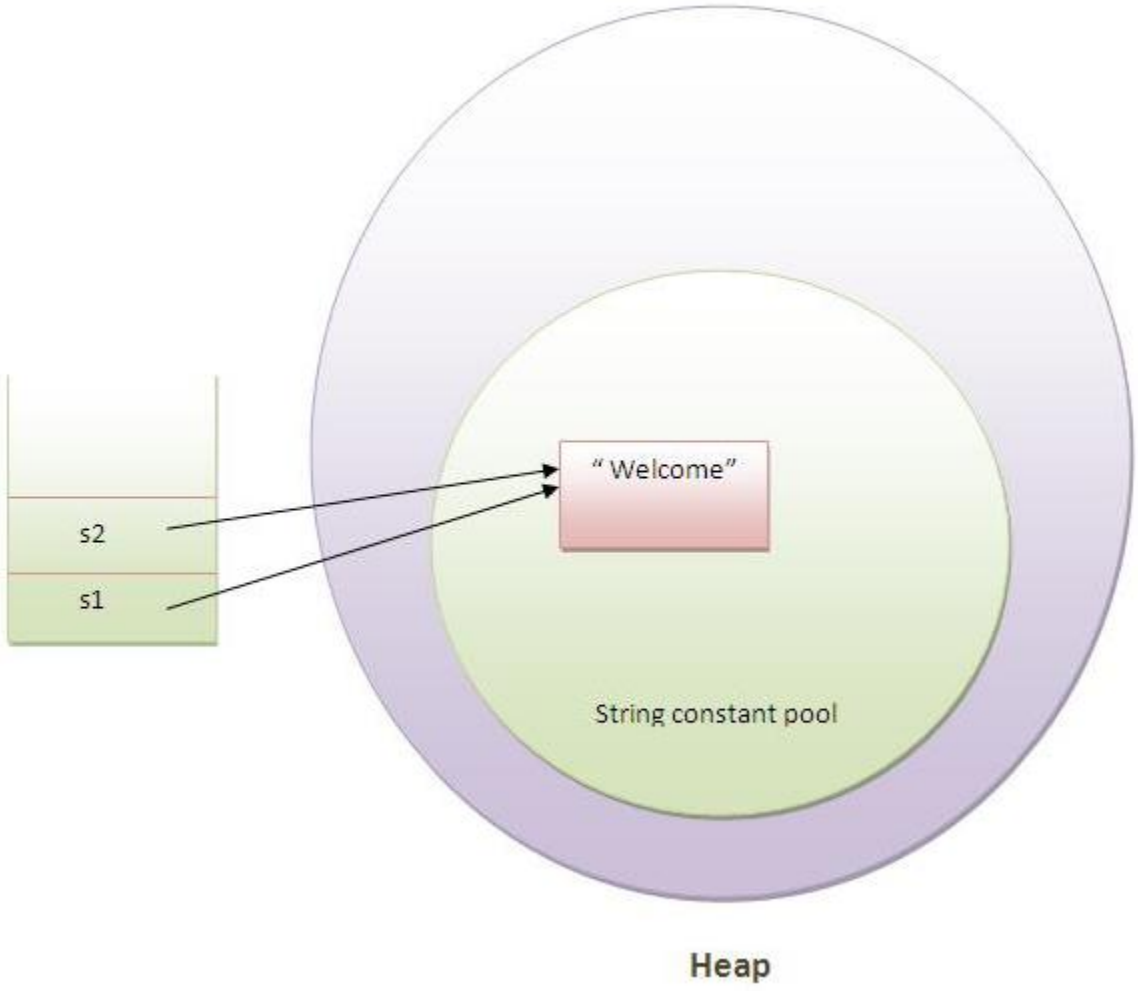
1) String Literal

Java String literal is created by using double quotes. For Example:

```
1.  String s="welcome";
```

Each time you create a string literal, the JVM checks the string constant pool first. If the string already exists in the pool, a reference to the pooled instance is returned. If string doesn't exist in the pool, a new string instance is created and placed in the pool. For example:

```
1.  String s1="Welcome";
2.  String s2="Welcome";//will not create new instance
```



In the above example only one object will be created. Firstly JVM will not find any string object with the value "Welcome" in string constant pool, so it will create a new object. After that it will find the string with the value "Welcome" in the pool, it will not create new object but will return the reference to the same instance.

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

Why java uses concept of string literal?

To make Java more memory efficient (because no new objects are created if it exists already in string constant pool).

2) By new keyword

```
1.  String s=new String("Welcome");//creates two objects and one reference variable
```

In such case, JVM will create a new string object in normal(non pool) heap memory and the literal "Welcome" will be placed in the string constant pool. The variable s will refer to the object in heap(non pool).

Java String Example

```
1.  public class StringExample{
2.  public static void main(String args[]){
3.  String s1="java";//creating string by java string literal
4.
5.  char ch[]={'s','t','r','i','n','g','s'};
6.  String s2=new String(ch);//converting char array to string
7.
8.  String s3=new String("example");//creating java string by new keyword
9.
10. System.out.println(s1);
11. System.out.println(s2);
12. System.out.println(s3);
13. }}
```

Output

```
java
strings
example
```

Java String class methods

The java.lang.String class provides many useful methods to perform operations on sequence of char values.

No.	Method	Description
1	char charAt(int index)	returns char value for the particular index
2	int length()	returns string length
3	static String format(String format, Object... args)	returns formatted string
4	static String format(Locale l, String format, Object... args)	returns formatted string with given locale
5	String substring(int beginIndex)	returns substring for given begin index
6	String substring(int beginIndex, int endIndex)	returns substring for given begin index and end index
7	boolean contains(CharSequence s)	returns true or false after matching the sequence of char value
8	static String join(CharSequence delimiter, CharSequence... elements)	returns a joined string
9	static String join(CharSequence delimiter, Iterable<? extends CharSequence> elements)	returns a joined string
10	boolean equals(Object another)	checks the equality of string with object
11	boolean isEmpty()	checks if string is empty
12	String concat(String str)	concatinates specified string

13	String replace(char old, char new)	replaces all occurrences of specified char value
14	String replace(CharSequence old, CharSequence new)	replaces all occurrences of specified CharSequence
15	String trim()	returns trimmed string omitting leading and trailing spaces
16	String split(String regex)	returns splitted string matching regex
17	String split(String regex, int limit)	returns splitted string matching regex and limit
18	String intern()	returns interned string
19	int indexOf(int ch)	returns specified char value index
20	int indexOf(int ch, int fromIndex)	returns specified char value index starting with given index
21	int indexOf(String substring)	returns specified substring index
22	int indexOf(String substring, int fromIndex)	returns specified substring index starting with given index
23	String toLowerCase()	returns string in lowercase.
24	String toLowerCase(Locale l)	returns string in lowercase using specified locale.
25	String toUpperCase()	returns string in uppercase.
26	String toUpperCase(Locale l)	returns string in uppercase using specified locale.

Immutable String in Java

In java, **string objects are immutable**. Immutable simply means unmodifiable or unchangeable.

Once string object is created its data or state can't be changed but a new string object is created.

Let's try to understand the immutability concept by the example given below:

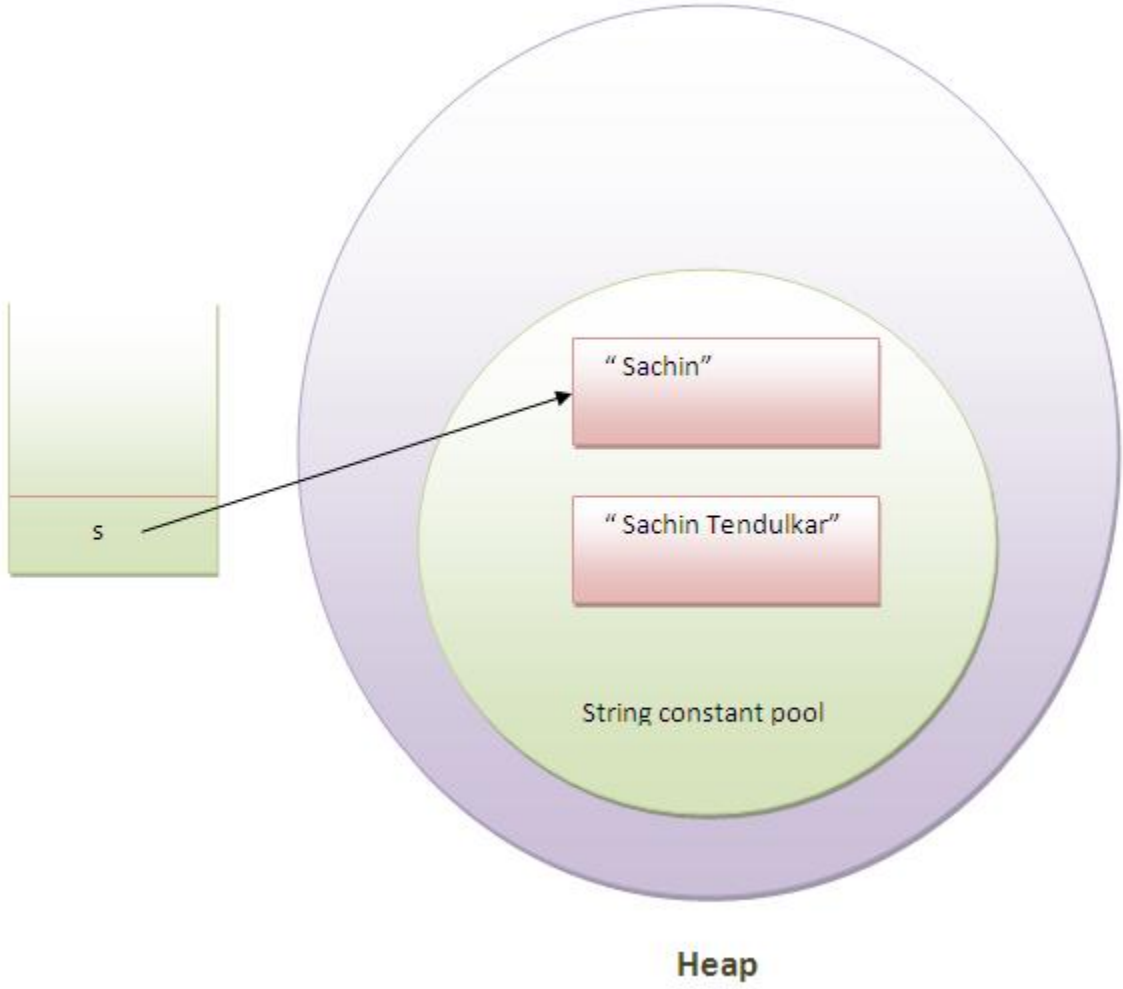
```
1.  class Testimmutablestring{
2.   public static void main(String args[]){
3.     String s="Sachin";
4.     s.concat(" Tendulkar");//concat() method appends the string at the end
```

```
5.              System.out.println(s);//will print Sachin because strings are immutable objects
6.          }
7.      }
```

Output

```
Output:Sachin
```

Now it can be understood by the diagram given below. Here Sachin is not changed but a new object is created with sachintendulkar. That is why string is known as immutable.



As you can see in the above figure that two objects are created but s reference variable still refers to "Sachin" not to "Sachin Tendulkar".

But if we explicitely assign it to the reference variable, it will refer to "Sachin Tendulkar" object.For example:

```
1.          class Testimmutablestring1{
2.           public static void main(String args[]){
3.             String s="Sachin";
4.             s=s.concat(" Tendulkar");
5.             System.out.println(s);
6.           }
7.          }
```

Output

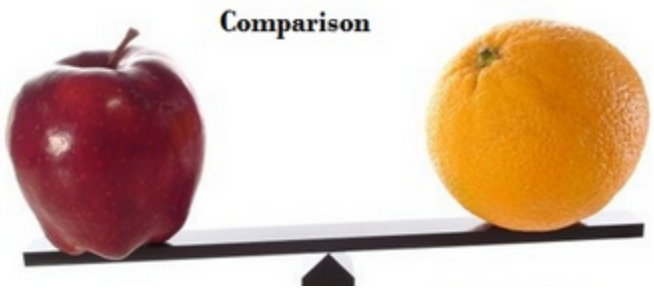
```
Output:Sachin Tendulkar
```

In such case, s points to the "Sachin Tendulkar". Please notice that still sachin object is not modified.

Why string objects are immutable in java?

Because java uses the concept of string literal.Suppose there are 5 reference variables,all referes to one object "sachin".If one reference variable changes the value of the object, it will be affected to all the reference variables. That is why string objects are immutable in java.

Java String compare



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

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

There are three ways to compare string in java:

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

1) String compare by equals() method

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

- **public boolean equals(Object another)** compares this string to the specified object.
- **public boolean equalsIgnoreCase(String another)** compares this String to another string, ignoring case.

```
1.  class Teststringcomparison1{
2.   public static void main(String args[]){
3.     String s1="Sachin";
4.     String s2="Sachin";
5.     String s3=new String("Sachin");
6.     String s4="Saurav";
7.     System.out.println(s1.equals(s2));//true
8.     System.out.println(s1.equals(s3));//true
9.     System.out.println(s1.equals(s4));//false
10.   }
11.  }
```

Output

```
Output:true
       true
       false
```

```
1.  class Teststringcomparison2{
2.   public static void main(String args[]){
3.     String s1="Sachin";
4.     String s2="SACHIN";
5.
6.     System.out.println(s1.equals(s2));//false
7.     System.out.println(s1.equalsIgnoreCase(s3));//true
8.   }
9.  }
```

Output

```
Output:false
       true
```

2) String compare by == operator

The = = operator compares references not values.

```
1.          class Teststringcomparison3{
2.           public static void main(String args[]){
3.             String s1="Sachin";
4.             String s2="Sachin";
5.             String s3=new String("Sachin");
6.             System.out.println(s1==s2);//true (because both refer to same instance)
7.             System.out.println(s1==s3);//false(because s3 refers to instance created in nonpool)
8.           }
9.          }
```

Output

```
Output:true
       false
```

3) String compare by compareTo() method

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

Suppose s1 and s2 are two string variables. If:

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

```
1.          class Teststringcomparison4{
2.           public static void main(String args[]){
3.             String s1="Sachin";
4.             String s2="Sachin";
5.             String s3="Ratan";
6.             System.out.println(s1.compareTo(s2));//0
7.             System.out.println(s1.compareTo(s3));//1(because s1>s3)
8.             System.out.println(s3.compareTo(s1));//-1(because s3 < s1 )
9.           }
10.         }
```

Output

```
Output:0
     1
     -1
```

String Concatenation in Java

In java, string concatenation forms a new string *that is* the combination of multiple strings. There are two ways to concat string in java:

1. By + (string concatenation) operator
2. By concat() method

1) String Concatenation by + (string concatenation) operator

Java string concatenation operator (+) is used to add strings. For Example:

```
1.  class TestStringConcatenation1{
2.   public static void main(String args[]){
3.     String s="Sachin"+" Tendulkar";
4.     System.out.println(s);//Sachin Tendulkar
5.   }
6.  }
```

Output

```
Output:Sachin Tendulkar
```

The **Java compiler transforms** above code to this:

```
1.  String s=(new StringBuilder()).append("Sachin").append(" Tendulkar).toString();
```

In java, String concatenation is implemented through the StringBuilder (or StringBuffer) class and its append method. String concatenation operator produces a new string by appending the second operand onto the end of the first operand. The string concatenation operator can concat not only string but primitive values also. For Example:

```
1.  class TestStringConcatenation2{
2.   public static void main(String args[]){
3.     String s=50+30+"Sachin"+40+40;
4.     System.out.println(s);//80Sachin4040
```

```
5.          }
6.        }
```

Output

```
80Sachin4040
```

Note: After a string literal, all the + will be treated as string concatenation operator.

2) String Concatenation by concat() method

The String concat() method concatenates the specified string to the end of current string. Syntax:

```
1.      public String concat(String another)
```

Let's see the example of String concat() method.

```
1.      class TestStringConcatenation3{
2.       public static void main(String args[]){
3.         String s1="Sachin ";
4.         String s2="Tendulkar";
5.         String s3=s1.concat(s2);
6.         System.out.println(s3);//Sachin Tendulkar
7.        }
8.      }
```

Output

```
Sachin Tendulkar
```

Substring in Java

A part of string is called **substring**. In other words, substring is a subset of another string. In case of substring startIndex is inclusive and endIndex is exclusive.

Note: Index starts from 0.

You can get substring from the given string object by one of the two methods:

1. **public String substring(int startIndex):** This method returns new String object containing the substring of the given string from specified startIndex (inclusive).
2. **public String substring(int startIndex, int endIndex):** This method returns new String object containing the substring of the given string from specified startIndex to endIndex.

In case of string:

- **startIndex:** inclusive
- **endIndex:** exclusive

Let's understand the startIndex and endIndex by the code given below.

```
1.      String s="hello";
2.      System.out.println(s.substring(0,2));//he
```

In the above substring, 0 points to h but 2 points to e (because end index is exclusive).

Example of java substring

```
1.      public class TestSubstring{
2.       public static void main(String args[]){
3.         String s="Sachin Tendulkar";
4.         System.out.println(s.substring(6));//Tendulkar
5.         System.out.println(s.substring(0,6));//Sachin
6.       }
7.      }
```

Output

```
Tendulkar
Sachin
```

Java String class methods

The java.lang.String class provides a lot of methods to work on string. By the help of these methods, we can perform operations on string such as trimming, concatenating, converting, comparing, replacing strings etc.

Java String is a powerful concept because everything is treated as a string if you submit any form in window based, web based or mobile application.

Let's see the important methods of String class.

Java String toUpperCase() and toLowerCase() method

The java string toUpperCase() method converts this string into uppercase letter and string toLowerCase() method into lowercase letter.

```
1.	String s="Sachin";
2.	System.out.println(s.toUpperCase());//SACHIN
3.	System.out.println(s.toLowerCase());//sachin
4.	System.out.println(s);//Sachin(no change in original)
```

Output

```
SACHIN
sachin
Sachin
```

Java String trim() method

The string trim() method eliminates white spaces before and after string.

```
1.	String s="  Sachin  ";
2.	System.out.println(s);//  Sachin
3.	System.out.println(s.trim());//Sachin
```

Output

```
 Sachin
Sachin
```

Java String startsWith() and endsWith() method

```
1.	String s="Sachin";
2.	 System.out.println(s.startsWith("Sa"));//true
3.	 System.out.println(s.endsWith("n"));//true
```

Output

```
true
true
```

Java String charAt() method

The string charAt() method returns a character at specified index.

```
1.	String s="Sachin";
2.	System.out.println(s.charAt(0));//S
```

```
3.      System.out.println(s.charAt(3));//h
```

Output

```
S
h
```

Java String length() method

The string length() method returns length of the string.

```
1.      String s="Sachin";
2.      System.out.println(s.length());//6
```

Output

```
6
```

Java String intern() method

A pool of strings, initially empty, is maintained privately by the class String.

When the intern method is invoked, if the pool already contains a string equal to this String object as determined by the equals(Object) method, then the string from the pool is returned. Otherwise, this String object is added to the pool and a reference to this String object is returned.

```
1.      String s=new String("Sachin");
2.      String s2=s.intern();
3.      System.out.println(s2);//Sachin
```

Output

```
Sachin
```

Java String valueOf() method

The string valueOf() method coverts given type such as int, long, float, double, boolean, char and char array into string.

```
1.      int a=10;
2.      String s=String.valueOf(a);
3.      System.out.println(s+10);
```

Output:

```
1010
```

Java String replace() method

The string replace() method replaces all occurrence of first sequence of character with second sequence of character.

```
1.  String s1="Java is a programming language. Java is a platform. Java is an Island.";
2.  String replaceString=s1.replace("Java","Kava");//replaces all occurrences of "Java" to "K
    ava"
3.  System.out.println(replaceString);
```

Output:

```
Kava is a programming language. Kava is a platform. Kava is an Island.
```

Java StringBuffer class

Java StringBuffer class is used to created mutable (modifiable) string. The StringBuffer class in java is same as String class except it is mutable i.e. it can be changed.

Note: Java StringBuffer class is thread-safe i.e. multiple threads cannot access it simultaneously. So it is safe and will result in an order.

Important Constructors of StringBuffer class

1. **StringBuffer():** creates an empty string buffer with the initial capacity of 16.
2. **StringBuffer(String str):** creates a string buffer with the specified string.
3. **StringBuffer(int capacity):** creates an empty string buffer with the specified capacity as length.

Important methods of StringBuffer class

1. **public synchronized StringBuffer append(String s):** is used to append the specified string with this string. The append() method is overloaded like append(char), append(boolean), append(int), append(float), append(double) etc.
2. **public synchronized StringBuffer insert(int offset, String s):** is used to insert the specified string with this string at the specified position. The insert() method is overloaded like insert(int, char), insert(int, boolean), insert(int, int), insert(int, float), insert(int, double) etc.
3. **public synchronized StringBuffer replace(int startIndex, int endIndex, String str):** is used to replace the string from specified startIndex and endIndex.
4. **public synchronized StringBuffer delete(int startIndex, int endIndex):** is used to delete the string from specified startIndex and endIndex.

5. **public synchronized StringBuffer reverse():** is used to reverse the string.
6. **public int capacity():** is used to return the current capacity.
7. **public void ensureCapacity(int minimumCapacity):** is used to ensure the capacity at least equal to the given minimum.
8. **public char charAt(int index):** is used to return the character at the specified position.
9. **public int length():** is used to return the length of the string i.e. total number of characters.
10. **public String substring(int beginIndex):** is used to return the substring from the specified beginIndex.
11. **public String substring(int beginIndex, int endIndex):** is used to return the substring from the specified beginIndex and endIndex.

What is mutable string

A string that can be modified or changed is known as mutable string. StringBuffer and StringBuilder classes are used for creating mutable string.

1) StringBuffer append() method

The append() method concatenates the given argument with this string.

```
1.  class A{
2.  public static void main(String args[]){
3.  StringBuffer sb=new StringBuffer("Hello ");
4.  sb.append("Java");//now original string is changed
5.  System.out.println(sb);//prints Hello Java
6.  }
7.  }
```

2) StringBuffer insert() method

The insert() method inserts the given string with this string at the given position.

```
1.  class A{
2.  public static void main(String args[]){
3.  StringBuffer sb=new StringBuffer("Hello ");
4.  sb.insert(1,"Java");//now original string is changed
5.  System.out.println(sb);//prints HJavaello
6.  }
7.  }
```

3) StringBuffer replace() method

The replace() method replaces the given string from the specified beginIndex and endIndex.

```
1.  class A{
2.  public static void main(String args[]){
3.  StringBuffer sb=new StringBuffer("Hello");
4.  sb.replace(1,3,"Java");
5.  System.out.println(sb);//prints HJavalo
6.  }
7.  }
```

4) StringBuffer delete() method

The delete() method of StringBuffer class deletes the string from the specified beginIndex to endIndex.

```
1.  class A{
2.  public static void main(String args[]){
3.  StringBuffer sb=new StringBuffer("Hello");
4.  sb.delete(1,3);
5.  System.out.println(sb);//prints Hlo
6.  }
7.  }
```

5) StringBuffer reverse() method

The reverse() method of StringBuilder class reverses the current string.

```
1.  class A{
2.  public static void main(String args[]){
3.  StringBuffer sb=new StringBuffer("Hello");
4.  sb.reverse();
5.  System.out.println(sb);//prints olleH
6.  }
7.  }
```

6) StringBuffer capacity() method

The capacity() method of StringBuffer class returns the current capacity of the buffer. The default capacity of the buffer is 16. If the number of character increases from its current capacity, it increases the capacity by (oldcapacity*2)+2. For example if your current capacity is 16, it will be (16*2)+2=34.

```
1.  class A{
2.  public static void main(String args[]){
3.  StringBuffer sb=new StringBuffer();
4.  System.out.println(sb.capacity());//default 16
5.  sb.append("Hello");
6.  System.out.println(sb.capacity());//now 16
7.  sb.append("java is my favourite language");
8.  System.out.println(sb.capacity());//now (16*2)+2=34 i.e (oldcapacity*2)+2
9.  }
10. }
```

7) StringBuffer ensureCapacity() method

The ensureCapacity() method of StringBuffer class ensures that the given capacity is the minimum to the current capacity. If it is greater than the current capacity, it increases the capacity by (oldcapacity*2)+2. For example if your current capacity is 16, it will be (16*2)+2=34.

```
1.  class A{
2.  public static void main(String args[]){
3.  StringBuffer sb=new StringBuffer();
4.  System.out.println(sb.capacity());//default 16
5.  sb.append("Hello");
6.  System.out.println(sb.capacity());//now 16
7.  sb.append("java is my favourite language");
8.  System.out.println(sb.capacity());//now (16*2)+2=34 i.e (oldcapacity*2)+2
9.  sb.ensureCapacity(10);//now no change
10. System.out.println(sb.capacity());//now 34
11. sb.ensureCapacity(50);//now (34*2)+2
12. System.out.println(sb.capacity());//now 70
13. }
14. }
```

Java StringBuilder class

Java StringBuilder class is used to create mutable (modifiable) string. The Java StringBuilder class is same as StringBuffer class except that it is non-synchronized. It is available since JDK 1.5.

Important Constructors of StringBuilder class

1. **StringBuilder():** creates an empty string Builder with the initial capacity of 16.

2. **StringBuilder(String str):** creates a string Builder with the specified string.
3. **StringBuilder(int length):** creates an empty string Builder with the specified capacity as length.

Important methods of StringBuilder class

Method	Description
public StringBuilder append(String s)	is used to append the specified string with this string. The append() method is overloaded like append(char), append(boolean), append(int), append(float), append(double) etc.
public StringBuilder insert(int offset, String s)	is used to insert the specified string with this string at the specified position. The insert() method is overloaded like insert(int, char), insert(int, boolean), insert(int, int), insert(int, float), insert(int, double) etc.
public StringBuilder replace(int startIndex, int endIndex, String str)	is used to replace the string from specified startIndex and endIndex.
public StringBuilder delete(int startIndex, int endIndex)	is used to delete the string from specified startIndex and endIndex.
public StringBuilder reverse()	is used to reverse the string.
public int capacity()	is used to return the current capacity.
public void ensureCapacity(int minimumCapacity)	is used to ensure the capacity at least equal to the given minimum.
public char charAt(int index)	is used to return the character at the specified position.
public int length()	is used to return the length of the string i.e. total number of characters.
public String substring(int beginIndex)	is used to return the substring from the specified beginIndex.
public String substring(int beginIndex, int endIndex)	is used to return the substring from the specified beginIndex and endIndex.

Java StringBuilder Examples

Let's see the examples of different methods of StringBuilder class.

1) StringBuilder append() method

The StringBuilder append() method concatenates the given argument with this string.

```
1.  class A{
2.  public static void main(String args[]){
3.  StringBuilder sb=new StringBuilder("Hello ");
4.  sb.append("Java");//now original string is changed
5.  System.out.println(sb);//prints Hello Java
6.  }
7.  }
```

2) StringBuilder insert() method

The StringBuilder insert() method inserts the given string with this string at the given position.

```
1.  class A{
2.  public static void main(String args[]){
3.  StringBuilder sb=new StringBuilder("Hello ");
4.  sb.insert(1,"Java");//now original string is changed
5.  System.out.println(sb);//prints HJavaello
6.  }
7.  }
```

3) StringBuilder replace() method

The StringBuilder replace() method replaces the given string from the specified beginIndex and endIndex.

```
1.  class A{
2.  public static void main(String args[]){
3.  StringBuilder sb=new StringBuilder("Hello");
4.  sb.replace(1,3,"Java");
5.  System.out.println(sb);//prints HJavalo
6.  }
7.  }
```

4) StringBuilder delete() method

The delete() method of StringBuilder class deletes the string from the specified beginIndex to endIndex.

```
1.  class A{
2.  public static void main(String args[]){
3.  StringBuilder sb=new StringBuilder("Hello");
4.  sb.delete(1,3);
5.  System.out.println(sb);//prints Hlo
6.  }
7.  }
```

5) StringBuilder reverse() method

The reverse() method of StringBuilder class reverses the current string.

```
1.  class A{
2.  public static void main(String args[]){
3.  StringBuilder sb=new StringBuilder("Hello");
4.  sb.reverse();
5.  System.out.println(sb);//prints olleH
6.  }
7.  }
```

6) StringBuilder capacity() method

The capacity() method of StringBuilder class returns the current capacity of the Builder. The default capacity of the Builder is 16. If the number of character increases from its current capacity, it increases the capacity by (oldcapacity*2)+2. For example if your current capacity is 16, it will be (16*2)+2=34.

```
1.  class A{
2.  public static void main(String args[]){
3.  StringBuilder sb=new StringBuilder();
4.  System.out.println(sb.capacity());//default 16
5.  sb.append("Hello");
6.  System.out.println(sb.capacity());//now 16
7.  sb.append("java is my favourite language");
8.  System.out.println(sb.capacity());//now (16*2)+2=34 i.e (oldcapacity*2)+2
9.  }
10. }
```

7) StringBuilder ensureCapacity() method

The ensureCapacity() method of StringBuilder class ensures that the given capacity is the minimum to the current capacity. If it is greater than the current capacity, it increases the capacity by (oldcapacity*2)+2. For example if your current capacity is 16, it will be (16*2)+2=34.

```
1.  class A{
2.  public static void main(String args[]){
3.  StringBuilder sb=new StringBuilder();
4.  System.out.println(sb.capacity());//default 16
5.  sb.append("Hello");
6.  System.out.println(sb.capacity());//now 16
7.  sb.append("java is my favourite language");
8.  System.out.println(sb.capacity());//now (16*2)+2=34 i.e (oldcapacity*2)+2
9.  sb.ensureCapacity(10);//now no change
10. System.out.println(sb.capacity());//now 34
11. sb.ensureCapacity(50);//now (34*2)+2
12. System.out.println(sb.capacity());//now 70
13. }
14. }
```

Difference between String and StringBuffer

There are many differences between String and StringBuffer. A list of differences between String and StringBuffer are given below:

No.	String	StringBuffer
1)	String class is immutable.	StringBuffer class is mutable.
2)	String is slow and consumes more memory when you concat too many strings because every time it creates new instance.	StringBuffer is fast and consumes less memory when you cancat strings.
3)	String class overrides the equals() method of Object class. So you can compare the contents of two strings by equals() method.	StringBuffer class doesn't override the equals() method of Object class.

Performance Test of String and StringBuffer

```
1.      public class ConcatTest{
2.         public static String concatWithString()    {
3.            String t = "Java";
4.            for (int i=0; i<10000; i++){
5.               t = t + "Python";
6.            }
7.            return t;
8.         }
9.         public static String concatWithStringBuffer(){
10.           StringBuffer sb = new StringBuffer("Java");
11.           for (int i=0; i<10000; i++){
12.              sb.append("Python");
13.           }
14.           return sb.toString();
15.        }
16.        public static void main(String[] args){
17.           long startTime = System.currentTimeMillis();
18.           concatWithString();
19.           System.out.println("Time taken by Concating with String: "+(System.currentTimeM
   illis()-startTime)+"ms");
20.           startTime = System.currentTimeMillis();
21.           concatWithStringBuffer();
22.           System.out.println("Time taken by Concating with  StringBuffer: "+(System.current
   TimeMillis()-startTime)+"ms");
23.        }
24.     }
```

```
Time taken by Concating with String: 578ms
Time taken by Concating with  StringBuffer: 0ms
```

String and StringBuffer HashCode Test

As you can see in the program given below, String returns new hashcode value when you concat string but StringBuffer returns same.

```
1.      public class InstanceTest{
2.         public static void main(String args[]){
3.            System.out.println("Hashcode test of String:");
4.            String str="java";
5.            System.out.println(str.hashCode());
6.            str=str+"python";
7.            System.out.println(str.hashCode());
8.
9.            System.out.println("Hashcode test of StringBuffer:");
```

```
10.             StringBuffer sb=new StringBuffer("java");
11.             System.out.println(sb.hashCode());
12.             sb.append("python");
13.             System.out.println(sb.hashCode());
14.         }
15.     }
```

```
Hashcode test of String:
3254818
229541438
Hashcode test of StringBuffer:
118352462
118352462
```

Difference between StringBuffer and StringBuilder

There are many differences between StringBuffer and StringBuilder. A list of differences between StringBuffer and StringBuilder are given below:

No.	StringBuffer	StringBuilder
1)	StringBuffer is *synchronized* i.e. thread safe. It means two threads can't call the methods of StringBuffer simultaneously.	StringBuilder is *non-synchronized* i.e. not thread safe. It means two threads can call the methods of StringBuilder simultaneously.
2)	StringBuffer is *less efficient* than StringBuilder.	StringBuilder is *more efficient* than StringBuffer.

StringBuffer Example

```
1.      public class BufferTest{
2.          public static void main(String[] args){
3.              StringBuffer buffer=new StringBuffer("hello");
4.              buffer.append("java");
5.              System.out.println(buffer);
6.          }
7.      }
```

```
hellojava
```

StringBuilder Example

```
1.          public class BuilderTest{
2.              public static void main(String[] args){
3.                  StringBuilder builder=new StringBuilder("hello");
4.                  builder.append("java");
5.                  System.out.println(builder);
6.              }
7.          }
```

```
hellojava
```

Performance Test of StringBuffer and StringBuilder

Let's see the code to check the performance of StringBuffer and StringBuilder classes.

```
1.          public class ConcatTest{
2.              public static void main(String[] args){
3.                  long startTime = System.currentTimeMillis();
4.                  StringBuffer sb = new StringBuffer("Java");
5.                  for (int i=0; i<10000; i++){
6.                      sb.append("Python");
7.                  }
8.                  System.out.println("Time taken by StringBuffer: " + (System.currentTimeMillis() -
    startTime) + "ms");
9.                  startTime = System.currentTimeMillis();
10.                 StringBuilder sb2 = new StringBuilder("Java");
11.                 for (int i=0; i<10000; i++){
12.                     sb2.append("Python");
13.                 }
14.                 System.out.println("Time taken by StringBuilder: " + (System.currentTimeMillis() -
    startTime) + "ms");
15.             }
16.         }
```

```
Time taken by StringBuffer: 16ms
Time taken by StringBuilder: 0ms
```

Java String compareTo

The **java string compareTo()** method compares the given string with current string lexicographically. It returns positive number, negative number or 0.

If first string is greater than second string, it returns positive number (difference of character value). If first string is less than second string, it returns negative number and if first string is equal to second string, it returns 0.

1. s1 > s2 => positive number
2. s1 < s2 => negative number
3. s1 == s2 => 0

Signature

1. **public int** compareTo(String anotherString)

Parameters

anotherString: represents string that is to be compared with current string

Returns

an integer value

Java String compareTo() method example

```
1.  public class LastIndexOfExample{
2.  public static void main(String args[]){
3.  String s1="hello";
4.  String s2="hello";
5.  String s3="meklo";
6.  String s4="hemlo";
7.  System.out.println(s1.compareTo(s2));
8.  System.out.println(s1.compareTo(s3));
9.  System.out.println(s1.compareTo(s4));
10. }}
```

Output:

```
0
-5
-1
```

Java String concat

The **java string concat()** method *combines specified string at the end of this string*. It returns combined string. It is like appending another string.

Signature

The signature of string concat() method is given below:

```
1.  public String concat(String anotherString)
```

Parameter

anotherString : another string i.e. to be combined at the end of this string.

Returns

combined string

Java String concat() method example

```
1.  public class ConcatExample{
2.  public static void main(String args[]){
3.  String s1="java string";
4.  s1.concat("is immutable");
5.  System.out.println(s1);
6.  s1=s1.concat(" is immutable so assign it explicitly");
7.  System.out.println(s1);
8.  }}
```

Output

```
java string
java string is immutable so assign it explicitly
```

Java String equals

The **java string equals()** method compares the two given strings based on the content of the string. If any character is not matched, it returns false. If all characters are matched, it returns true.

The String equals() method overrides the equals() method of Object class.

Signature

```
1.      public boolean equals(Object anotherObject)
```

Parameter

anotherObject : another object i.e. compared with this string.

Returns

true if characters of both strings are equal otherwise **false**.

Overrides

equals() method of java Object class.

Java String equals() method example

```
1.      public class EqualsExample{
2.      public static void main(String args[]){
3.      String s1="java";
4.      String s2="java";
5.      String s3="JAVA";
6.      String s4="python";
7.      System.out.println(s1.equals(s2));//true because content and case is same
8.      System.out.println(s1.equals(s3));//false because case is not same
9.      System.out.println(s1.equals(s4));//false because content is not same
10.     }}
```

Output

```
true
false
```

```
false
```

Java String length

The **java string length()** method length of the string. It returns count of total number of characters. The length of java string is same as the unicode code units of the string.

Signature

The signature of the string length() method is given below:

```
1.  public int length()
```

Specified by

CharSequence interface

Returns

length of characters

Java String length() method example

```
1.  public class LengthExample{
2.  public static void main(String args[]){
3.  String s1="java";
4.  String s2="python";
5.  System.out.println("string length is: "+s1.length());//4 is the length of java string
6.  System.out.println("string length is: "+s2.length());//6 is the length of python string
7.  }}
```

Output

```
string length is: 4
string length is: 6
```

Java String substring

The **java string substring()** method returns a part of the string.

We pass begin index and end index number position in the java substring method where start index is inclusive and end index is exclusive. In other words, start index starts from 0 whereas end index starts from 1.

There are two types of substring methods in java string.

Signature

```
1.  public String substring(int startIndex)
2.  and
3.  public String substring(int startIndex, int endIndex)
```

If you don't specify endIndex, java substring() method will return all the characters from startIndex.

Parameters

startIndex : starting index is inclusive

endIndex : ending index is exclusive

Returns

specified string

Throws

StringIndexOutOfBoundsException if start index is negative value or end index is lower than starting index.

Java String substring() method example

```
1.  public class SubstringExample{
2.  public static void main(String args[]){
3.  String s1="javapython";
4.  System.out.println(s1.substring(2,4));//returns va
5.  System.out.println(s1.substring(2));//returns vapython
6.  }}
```

Output

```
va
vapython
```

Java String split

The **java string split()** method splits this string against given regular expression and returns a char array.

Signature

There are two signature for split() method in java string.

1. **public** String split(String regex)
2. and,
3. **public** String split(String regex, **int** limit)

Parameter

regex : regular expression to be applied on string.

limit : limit for the number of strings in array. If it is zero, it will returns all the strings matching regex.

Returns

array of strings

Throws

PatternSyntaxException if pattern for regular expression is invalid

Since

1.4

Java String split() method example

The given example returns total number of words in a string excluding space only. It also includes special characters.

```
1.  public class SplitExample{
2.  public static void main(String args[]){
3.  String s1="java string split method by javatpoint";
4.  String[] words=s1.split("\\s");//splits the string based on string
5.  //using java foreach loop to print elements of string array
6.  for(String w:words){
7.  System.out.println(w);
8.  }
9.  }}
```

```
java
string
split
method
by
javatpoint
```

Java String split() method with regex and length example

```
1.  public class SplitExample2{
2.  public static void main(String args[]){
3.  String s1="welcome to split world";
4.  System.out.println("returning words:");
5.  for(String w:s1.split("\\s",0)){
6.  System.out.println(w);
7.  }
8.  System.out.println("returning words:");
9.  for(String w:s1.split("\\s",1)){
10. System.out.println(w);
11. }
12. System.out.println("returning words:");
13. for(String w:s1.split("\\s",2)){
14. System.out.println(w);
15. }
16.
17. }}
```

```
returning words:
welcome
to
split
world
returning words:
```

```
welcome to split world
returning words:
welcome
to split world
```

Java String toLowerCase()

The **java string toLowerCase()** method returns the string in lowercase letter. In other words, it converts all characters of the string into lower case letter.

The toLowerCase() method works same as toLowerCase(Locale.getDefault()) method. It internally uses the default locale.

Signature

There are two variant of toLowerCase() method. The signature or syntax of string toLowerCase() method is given below:

```
1.  public String toLowerCase()
2.  public String toLowerCase(Locale locale)
```

The second method variant of toLowerCase(), converts all the characters into lowercase using the rules of given Locale.

Returns

string in lowercase letter.

Java String toLowerCase() method example

```
1.  public class StringLowerExample{
2.  public static void main(String args[]){
3.  String s1="JAVATPOINT HELLO stRIng";
4.  String s1lower=s1.toLowerCase();
5.  System.out.println(s1lower);
6.  }}
```

Output:

javatpoint hello string

Java String toUpperCase

The **java string toUpperCase()** method returns the string in uppercase letter. In other words, it converts all characters of the string into upper case letter.

The toUpperCase() method works same as toUpperCase(Locale.getDefault()) method. It internally uses the default locale.

Signature

There are two variant of toUpperCase() method. The signature or syntax of string toUpperCase() method is given below:

```
1.  public String toUpperCase()
2.  public String toUpperCase(Locale locale)
```

The second method variant of toUpperCase(), converts all the characters into uppercase using the rules of given Locale.

Returns

string in uppercase letter.

Java String toUpperCase() method example

```
1.  public class StringUpperExample{
2.  public static void main(String args[]){
3.  String s1="hello string";
4.  String s1upper=s1.toUpperCase();
5.  System.out.println(s1upper);
6.  }}
```

Output:

HELLO STRING

Java String trim

The **java string trim()** method eliminates leading and trailing spaces. The unicode value of space character is '\u0020'. The trim() method in java string checks this unicode value before and after the string, if it exists then removes the spaces and returns the omitted string.

The string trim() method doesn't omits middle spaces.

Signature

The signature or syntax of string trim method is given below:

```
1.  public String trim()
```

Returns

string with omitted leading and trailing spaces

Java String trim() method example

```
1.  public class StringTrimExample{
2.  public static void main(String args[]){
3.  String s1="  hello string   ";
4.  System.out.println(s1+"java");//without trim()
5.  System.out.println(s1.trim()+"java");//with trim()
6.  }}
```

```
 hello string   java
hello stringjava
```

Java I/O (File Handling)

Java I/O (Input and Output) is used to process the input and produce the output based on the input.

Java uses the concept of stream to make I/O operation fast. The java.io package contains all the classes required for input and output operations.

We can perform **file handling in java** by java IO API.

Stream

A stream is a sequence of data.In Java a stream is composed of bytes. It's called a stream because it's like a stream of water that continues to flow.

In java, 3 streams are created for us automatically. All these streams are attached with console.

1) System.out: standard output stream

2) System.in: standard input stream

3) System.err: standard error stream

Let's see the code to print **output and error** message to the console.

```
1.  System.out.println("simple message");
2.  System.err.println("error message");
```

Let's see the code to get **input** from console.

```
1.  int i=System.in.read();//returns ASCII code of 1st character
2.  System.out.println((char)i);//will print the character
```

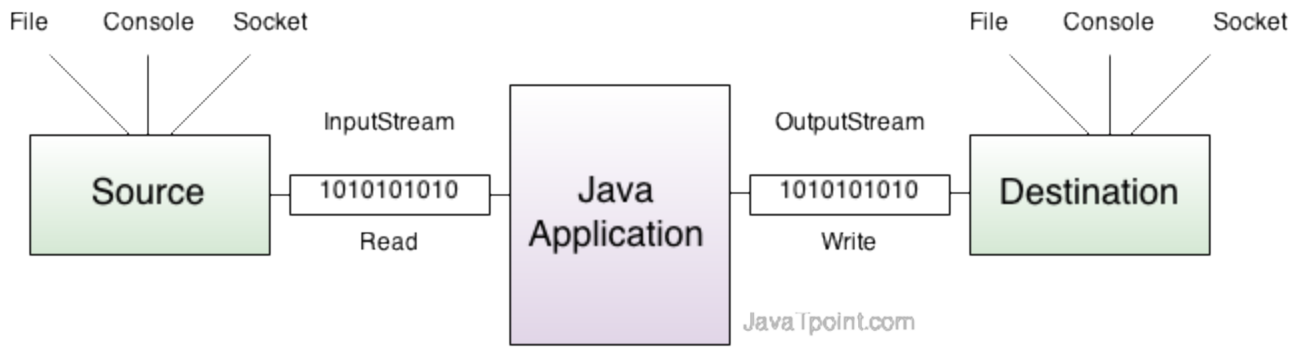
OutputStream

Java application uses an output stream to write data to a destination, it may be a file,an array,peripheral device or socket.

InputStream

Java application uses an input stream to read data from a source, it may be a file,an array,peripheral device or socket.

Let's understand working of Java OutputStream and InputStream by the figure given below.

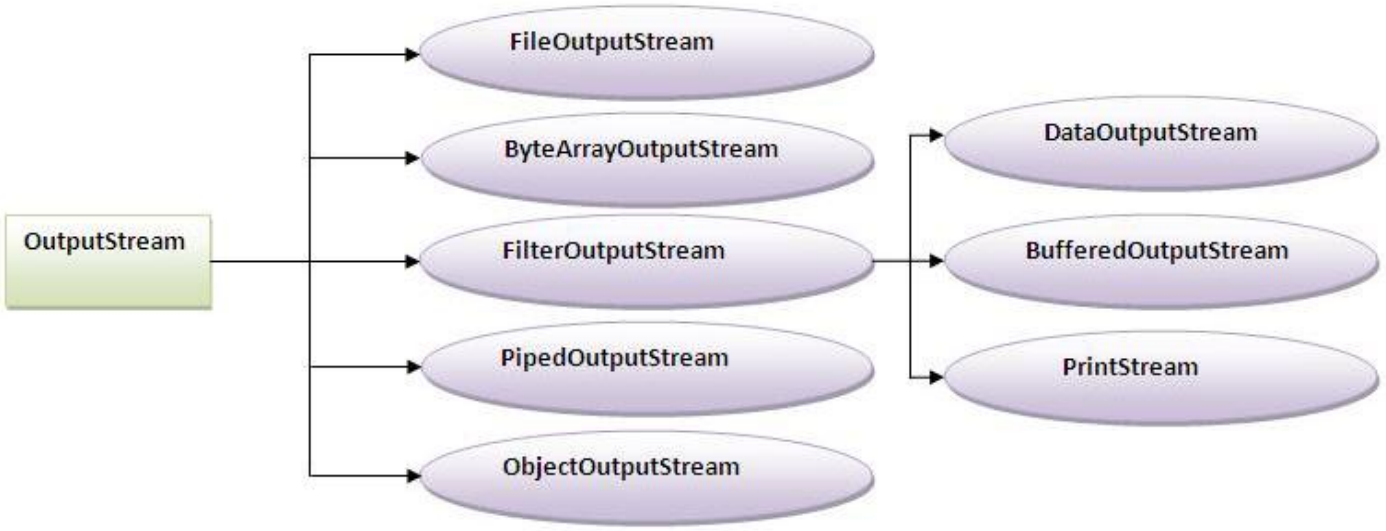


OutputStream class

OutputStream class is an abstract class.It is the superclass of all classes representing an output stream of bytes. An output stream accepts output bytes and sends them to some sink.

Commonly used methods of OutputStream class

Method	Description
1) public void write(int)throws IOException:	is used to write a byte to the current output stream.
2) public void write(byte[])throws IOException:	is used to write an array of byte to the current output stream.
3) public void flush()throws IOException:	flushes the current output stream.
4) public void close()throws IOException:	is used to close the current output stream.

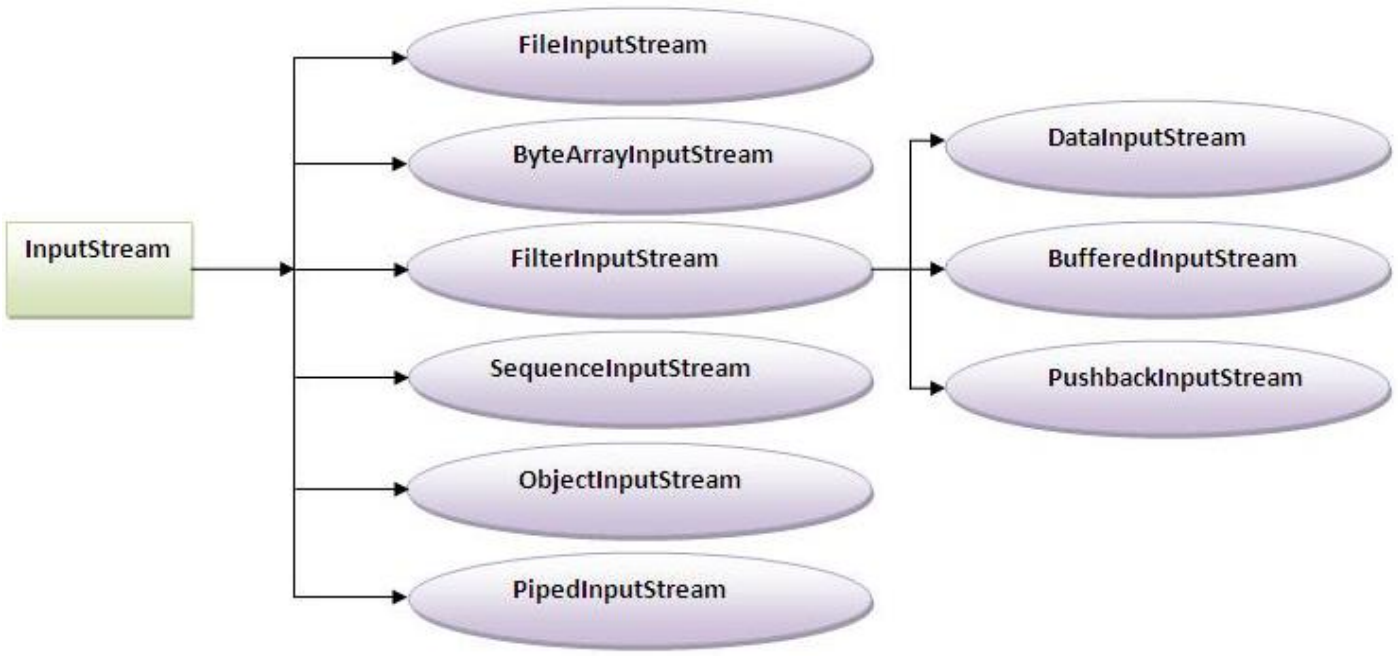


InputStream class

InputStream class is an abstract class.It is the superclass of all classes representing an input stream of bytes.

Commonly used methods of InputStream class

Method	Description
1) public abstract int read()throws IOException:	reads the next byte of data from the input stream.It returns -1 at the end of file.
2) public int available()throws IOException:	returns an estimate of the number of bytes that can be read from the current input stream.
3) public void close()throws IOException:	is used to close the current input stream.



FileInputStream and FileOutputStream (File Handling)

In Java, FileInputStream and FileOutputStream classes are used to read and write data in file. In another words, they are used for file handling in java.

Java FileOutputStream class

Java FileOutputStream is an output stream for writing data to a file.

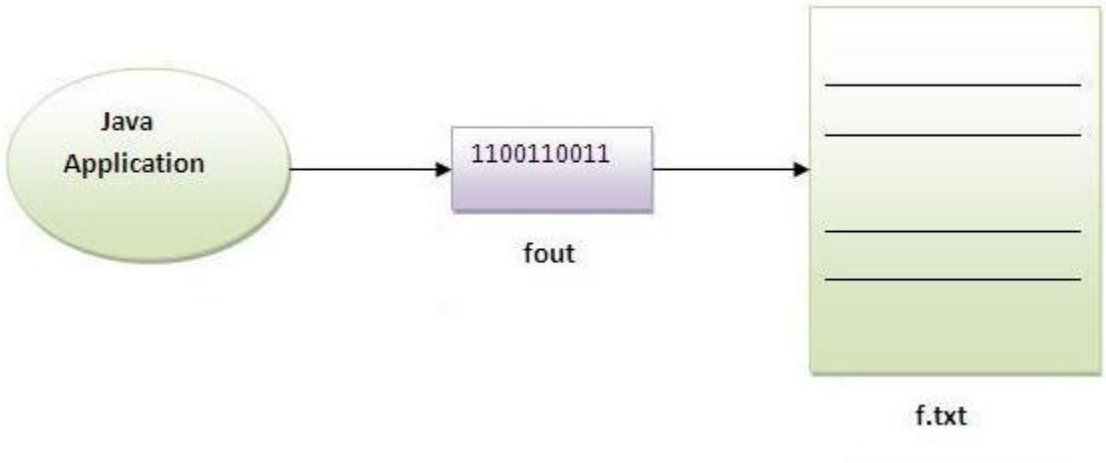
If you have to write primitive values then use FileOutputStream.Instead, for character-oriented data, prefer FileWriter.But you can write byte-oriented as well as character-oriented data.

Example of Java FileOutputStream class

```
1.  import java.io.*;
2.  class Test{
3.    public static void main(String args[]){
4.     try{
5.       FileOutputstream fout=new FileOutputStream("abc.txt");
6.       String s="Sachin Tendulkar is my favourite player";
7.       byte b[]=s.getBytes();//converting string into byte array
8.       fout.write(b);
9.       fout.close();
10.      System.out.println("success...");
11.     }catch(Exception e){system.out.println(e);}
```

```
12.         }
13.       }
```

Output:success...



Java FileInputStream class

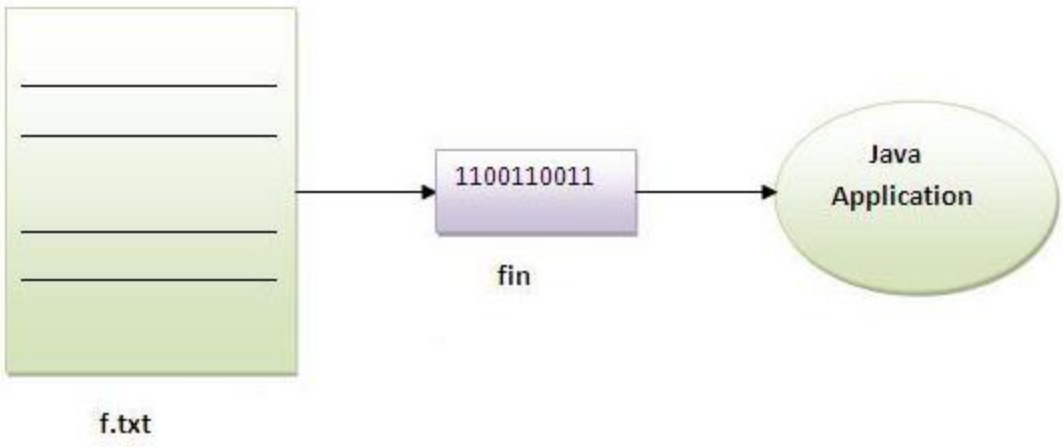
Java FileInputStream class obtains input bytes from a file.It is used for reading streams of raw bytes such as image data. For reading streams of characters, consider using FileReader.

It should be used to read byte-oriented data for example to read image, audio, video etc.

Example of FileInputStream class

```
1.        import java.io.*;
2.        class SimpleRead{
3.         public static void main(String args[]){
4.          try{
5.            FileInputStream fin=new FileInputStream("abc.txt");
6.            int i=0;
7.            while((i=fin.read())!=-1){
8.             System.out.println((char)i);
9.            }
10.           fin.close();
11.         }catch(Exception e){system.out.println(e);}
12.       }
13.       }
```

Output:Sachin is my favourite player.



Example of Reading the data of current java file and writing it into another file

We can read the data of any file using the FileInputStream class whether it is java file, image file, video file etc. In this example, we are reading the data of C.java file and writing it into another file M.java.

```
1.  import java.io.*;
2.  class C{
3.  public static void main(String args[])throws Exception{
4.  FileInputStream fin=new FileInputStream("C.java");
5.  FileOutputStream fout=new FileOutputStream("M.java");
6.  int i=0;
7.  while((i=fin.read())!=-1){
8.  fout.write((byte)i);
9.  }
10. fin.close();
11. }
12. }
```

Java ByteArrayOutputStream class

Java ByteArrayOutputStream class is used to write data into multiple files. In this stream, the data is written into a byte array that can be written to multiple stream.

The ByteArrayOutputStream holds a copy of data and forwards it to multiple streams.

The buffer of ByteArrayOutputStream automatically grows according to data.

Closing the ByteArrayOutputStream has no effect.

Constructors of ByteArrayOutputStream class

Constructor	Description
ByteArrayOutputStream()	creates a new byte array output stream with the initial capacity of 32 bytes, though its size increases if necessary.
ByteArrayOutputStream(int size)	creates a new byte array output stream, with a buffer capacity of the specified size, in bytes.

Methods of ByteArrayOutputStream class

Method	Description
1) public synchronized void writeTo(OutputStream out) throws IOException	writes the complete contents of this byte array output stream to the specified output stream.
2) public void write(byte b) throws IOException	writes byte into this stream.
3) public void write(byte[] b) throws IOException	writes byte array into this stream.
4) public void flush()	flushes this stream.
5) public void close()	has no affect, it doesn't closes the bytearrayoutputstream.

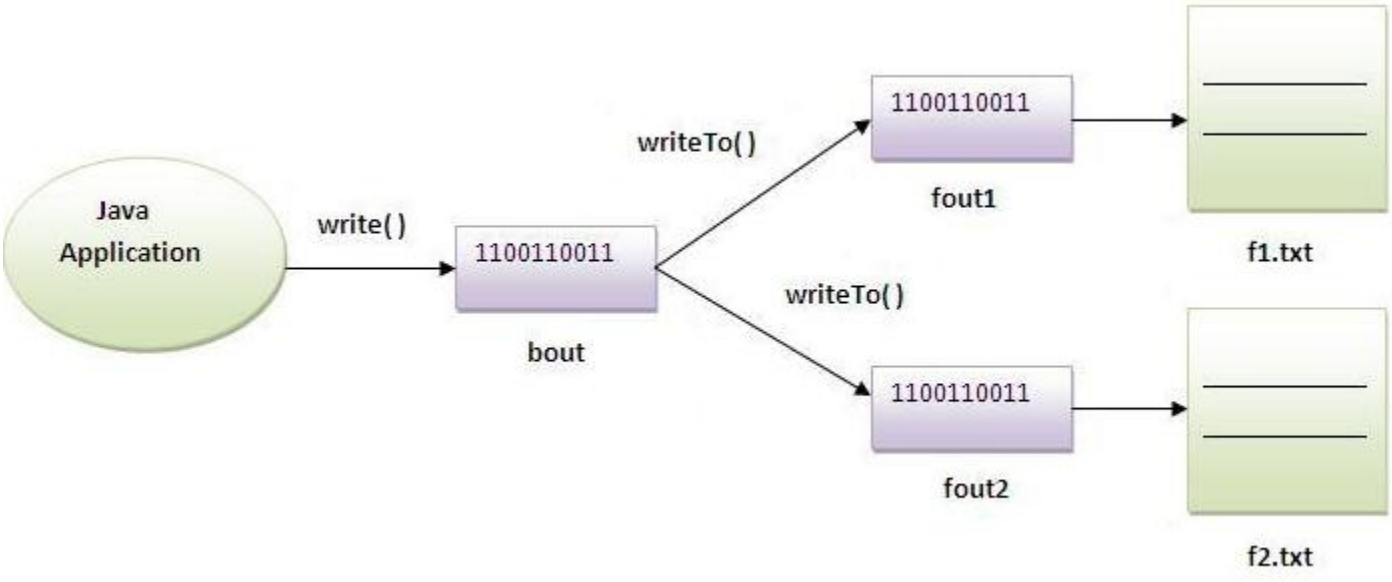
Java ByteArrayOutputStream Example

Let's see a simple example of java ByteArrayOutputStream class to write data into 2 files.

```
1.  import java.io.*;
2.  class S{
3.   public static void main(String args[])throws Exception{
4.    FileOutputStream fout1=new FileOutputStream("f1.txt");
5.    FileOutputStream fout2=new FileOutputStream("f2.txt");
6.
7.    ByteArrayOutputStream bout=new ByteArrayOutputStream();
8.    bout.write(139);
9.    bout.writeTo(fout1);
```

```
10.             bout.writeTo(fout2);
11.
12.             bout.flush();
13.             bout.close();//has no effect
14.             System.out.println("success...");
15.            }
16.           }
```

```
success...
```



Java BufferedOutputStream and BufferedInputStream

Java BufferedOutputStream class

Java BufferedOutputStream class uses an internal buffer to store data. It adds more efficiency than to write data directly into a stream. So, it makes the performance fast.

Example of BufferedOutputStream class:

In this example, we are writing the textual information in the BufferedOutputStream object which is connected to the FileOutputStream object. The flush() flushes the data of one stream and send it into another. It is required if you have connected the one stream with another.

```
1.          import java.io.*;
2.          class Test{
3.           public static void main(String args[])throws Exception{
```

```
4.              FileOutputStream fout=new FileOutputStream("f1.txt");
5.              BufferedOutputStream bout=new BufferedOutputStream(fout);
6.              String s="Sachin is my favourite player";
7.              byte b[]=s.getBytes();
8.              bout.write(b);
9.
10.             bout.flush();
11.             bout.close();
12.             fout.close();
13.             System.out.println("success");
14.           }
15.           }
```

Output:

```
success...
```

Java BufferedInputStream class

Java BufferedInputStream class is used to read information from stream. It internally uses buffer mechanism to make the performance fast.

Example of Java BufferedInputStream

Let's see the simple example to read data of file using BufferedInputStream.

```
1.            import java.io.*;
2.            class SimpleRead{
3.             public static void main(String args[]){
4.              try{
5.                FileInputStream fin=new FileInputStream("f1.txt");
6.                BufferedInputStream bin=new BufferedInputStream(fin);
7.                int i;
8.                while((i=bin.read())!=-1){
9.                 System.out.println((char)i);
10.               }
11.               bin.close();
12.               fin.close();
13.             }catch(Exception e){system.out.println(e);}
14.           }
15.           }
```

Output:

Sachin is my favourite player

Java FileWriter and FileReader (File Handling in java)

Java FileWriter and FileReader classes are used to write and read data from text files. These are character-oriented classes, used for file handling in java.

Java has suggested not to use the FileInputStream and FileOutputStream classes if you have to read and write the textual information.

Java FileWriter class

Java FileWriter class is used to write character-oriented data to the file.

Constructors of FileWriter class

Constructor	Description
FileWriter(String file)	creates a new file. It gets file name in string.
FileWriter(File file)	creates a new file. It gets file name in File object.

Methods of FileWriter class

Method	Description
1) public void write(String text)	writes the string into FileWriter.
2) public void write(char c)	writes the char into FileWriter.
3) public void write(char[] c)	writes char array into FileWriter.
4) public void flush()	flushes the data of FileWriter.
5) public void close()	closes FileWriter.

Java FileWriter Example

In this example, we are writing the data in the file abc.txt.

```
1.  import java.io.*;
2.  class Simple{
3.   public static void main(String args[]){
4.    try{
5.     FileWriter fw=new FileWriter("abc.txt");
6.     fw.write("my name is sachin");
7.     fw.close();
8.    }catch(Exception e){System.out.println(e);}
9.    System.out.println("success");
10.  }
11. }
```

Output:

```
success...
```

Java FileReader class

Java FileReader class is used to read data from the file. It returns data in byte format like FileInputStream class.

Constructors of FileWriter class

Constructor	Description
FileReader(String file)	It gets filename in string. It opens the given file in read mode. If file doesn't exist, it throws FileNotFoundException.
FileReader(File file)	It gets filename in file instance. It opens the given file in read mode. If file doesn't exist, it throws FileNotFoundException.

Methods of FileReader class

Method	Description
1) public int read()	returns a character in ASCII form. It returns -1 at the end of file.
2) public void close()	closes FileReader.

Java FileReader Example

In this example, we are reading the data from the file abc.txt file.

```
1.  import java.io.*;
2.  class Simple{
3.   public static void main(String args[])throws Exception{
4.    FileReader fr=new FileReader("abc.txt");
5.    int i;
6.    while((i=fr.read())!=-1)
7.    System.out.println((char)i);
8.
9.    fr.close();
10.  }
11. }
```

12. Output:
13. my name is sachin

CharArrayWriter class:

The CharArrayWriter class can be used to write data to multiple files. This class implements the Appendable interface. Its buffer automatically grows when data is written in this stream. Calling the close() method on this object has no effect.

Example of CharArrayWriter class:

In this example, we are writing a common data to 4 files a.txt, b.txt, c.txt and d.txt.

```
1.  import java.io.*;
2.  class Simple{
3.   public static void main(String args[])throws Exception{
4.
5.    CharArrayWriter out=new CharArrayWriter();
6.    out.write("my name is");
7.
8.    FileWriter f1=new FileWriter("a.txt");
9.    FileWriter f2=new FileWriter("b.txt");
10.   FileWriter f3=new FileWriter("c.txt");
11.   FileWriter f4=new FileWriter("d.txt");
12.
13.   out.writeTo(f1);
14.   out.writeTo(f2);
15.   out.writeTo(f3);
```

```
16.             out.writeTo(f4);
17.
18.
19.             f1.close();
20.             f2.close();
21.             f3.close();
22.             f4.close();
23.            }
24.            }
```

Reading data from keyboard:

There are many ways to read data from the keyboard. For example:

- InputStreamReader
- Console
- Scanner
- DataInputStream etc.

InputStreamReader class:

InputStreamReader class can be used to read data from keyboard.It performs two tasks:

- connects to input stream of keyboard
- converts the byte-oriented stream into character-oriented stream

BufferedReader class:

BufferedReader class can be used to read data line by line by readLine() method.

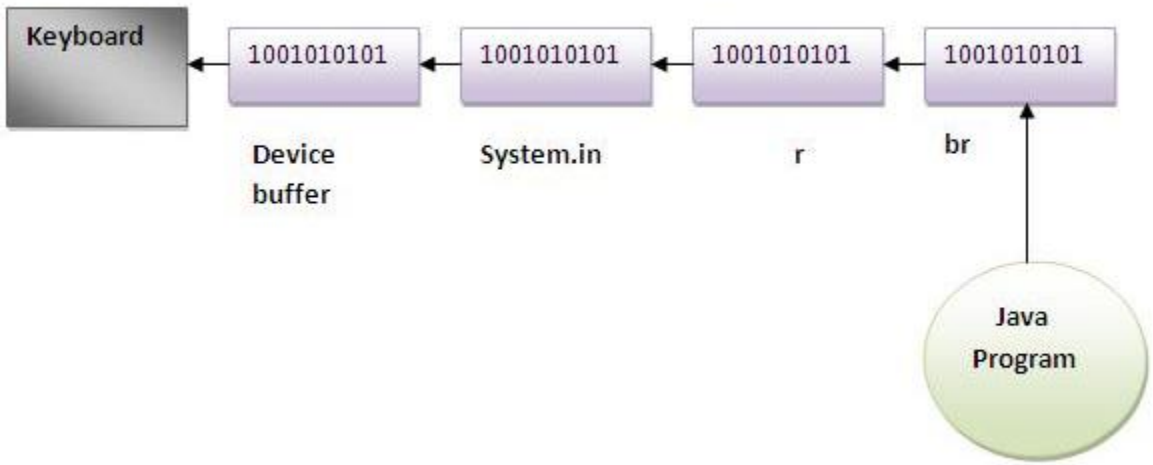
Example of reading data from keyboard by InputStreamReader and BufferdReader class:

In this example, we are connecting the BufferedReader stream with the InputStreamReader stream for reading the line by line data from the keyboard.

```
1.              //<b><i>Program of reading data</i></b>
2.
3.              import java.io.*;
4.              class G5{
5.              public static void main(String args[])throws Exception{
6.
7.              InputStreamReader r=new InputStreamReader(System.in);
```

```
8.          BufferedReader br=new BufferedReader(r);
9.
10.         System.out.println("Enter your name");
11.         String name=br.readLine();
12.         System.out.println("Welcome "+name);
13.          }
14.         }
```

Output:Enter your name
Amit
Welcome Amit



Another Example of reading data from keyboard by InputStreamReader and BufferdReader class until the user writes stop

In this example, we are reading and printing the data until the user prints stop.

```
1.          import java.io.*;
2.          class G5{
3.          public static void main(String args[])throws Exception{
4.
5.           InputStreamReader r=new InputStreamReader(System.in);
6.           BufferedReader br=new BufferedReader(r);
7.
8.           String name="";
9.
10.           while(name.equals("stop")){
11.            System.out.println("Enter data: ");
12.            name=br.readLine();
```

```
13.              System.out.println("data is: "+name);
14.             }
15.
16.            br.close();
17.            r.close();
18.            }
19.           }
```

```
Output:Enter data: Amit
       data is: Amit
       Enter data: 10
       data is: 10
       Enter data: stop
       data is: stop
```

java.io.PrintStream class:

The PrintStream class provides methods to write data to another stream. The PrintStream class automatically flushes the data so there is no need to call flush() method. Moreover, its methods don't throw IOException.

Commonly used methods of PrintStream class:

There are many methods in PrintStream class. Let's see commonly used methods of PrintStream class:

- **public void print(boolean b):** it prints the specified boolean value.
- **public void print(char c):** it prints the specified char value.
- **public void print(char[] c):** it prints the specified character array values.
- **public void print(int i):** it prints the specified int value.
- **public void print(long l):** it prints the specified long value.
- **public void print(float f):** it prints the specified float value.
- **public void print(double d):** it prints the specified double value.
- **public void print(String s):** it prints the specified string value.
- **public void print(Object obj):** it prints the specified object value.
- **public void println(boolean b):** it prints the specified boolean value and terminates the line.
- **public void println(char c):** it prints the specified char value and terminates the line.
- **public void println(char[] c):** it prints the specified character array values and terminates the line.
- **public void println(int i):** it prints the specified int value and terminates the line.
- **public void println(long l):** it prints the specified long value and terminates the line.
- **public void println(float f):** it prints the specified float value and terminates the line.

- **public void println(double d):** it prints the specified double value and terminates the line.
- **public void println(String s):** it prints the specified string value and terminates the line./li>
- **public void println(Object obj):** it prints the specified object value and terminates the line.
- **public void println():** it terminates the line only.
- **public void printf(Object format, Object... args):** it writes the formatted string to the current stream.
- **public void printf(Locale l, Object format, Object... args):** it writes the formatted string to the current stream.
- **public void format(Object format, Object... args):** it writes the formatted string to the current stream using specified format.
- **public void format(Locale l, Object format, Object... args):** it writes the formatted string to the current stream using specified format.

Example of java.io.PrintStream class:

In this example, we are simply printing integer and string values.

```
1.  import java.io.*;
2.  class PrintStreamTest{
3.   public static void main(String args[])throws Exception{
4.
5.     FileOutputStream fout=new FileOutputStream("mfile.txt");
6.     PrintStream pout=new PrintStream(fout);
7.     pout.println(1900);
8.     pout.println("Hello Java");
9.     pout.println("Welcome to Java");
10.    pout.close();
11.    fout.close();
12.
13.  }
14.  }
```

Exception Handling in Java

The **exception handling in java** is one of the powerful mechanism to handle the runtime errors so that normal flow of the application can be maintained.

What is exception

Dictionary Meaning: Exception is an abnormal condition.

In java, exception is an event that disrupts the normal flow of the program. It is an object which is thrown at runtime.

What is exception handling

Exception Handling is a mechanism to handle runtime errors such as ClassNotFound, IO, SQL, Remote etc.

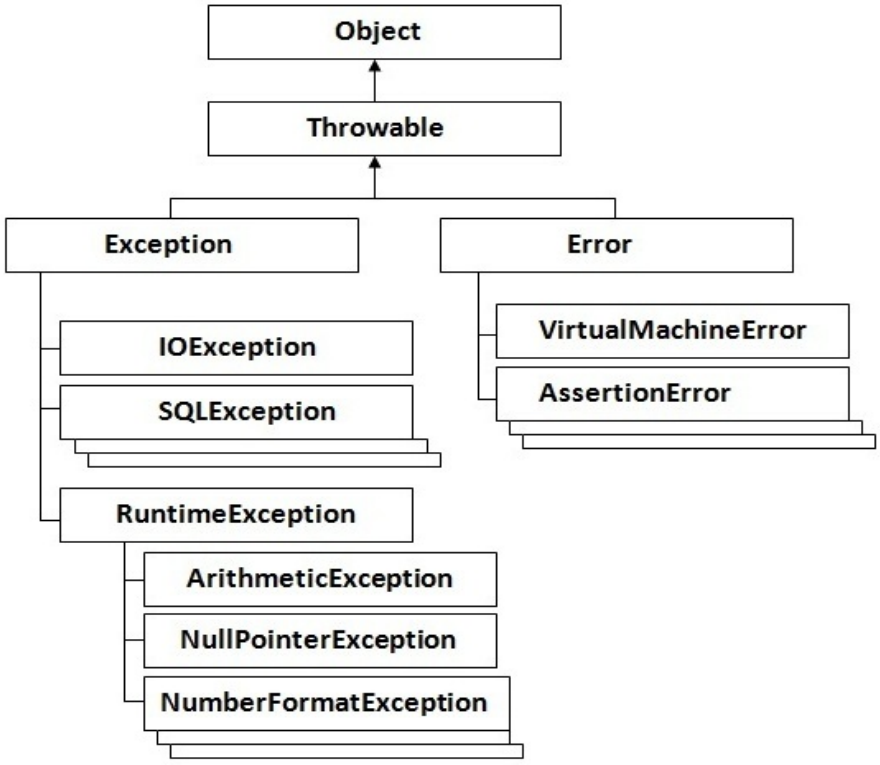
Advantage of Exception Handling

The core advantage of exception handling is **to maintain the normal flow of the application**. Exception normally disrupts the normal flow of the application that is why we use exception handling. Let's take a scenario:

```
1.  statement 1;
2.  statement 2;
3.  statement 3;
4.  statement 4;
5.  statement 5;//exception occurs
6.  statement 6;
7.  statement 7;
8.  statement 8;
9.  statement 9;
10. statement 10;
```

Suppose there is 10 statements in your program and there occurs an exception at statement 5, rest of the code will not be executed i.e. statement 6 to 10 will not run. If we perform exception handling, rest of the exception will be executed. That is why we use exception handling in java.

Hierarchy of Java Exception classes



Types of Exception

There are mainly two types of exceptions: checked and unchecked where error is considered as unchecked exception. The sun microsystem says there are three types of exceptions:

1. Checked Exception
2. Unchecked Exception
3. Error

Difference between checked and unchecked exceptions

1) Checked Exception

The classes that extend Throwable class except RuntimeException and Error are known as checked exceptions e.g.IOException, SQLException etc. Checked exceptions are checked at compile-time.

2) Unchecked Exception

The classes that extend RuntimeException are known as unchecked exceptions e.g. ArithmeticException, NullPointerException, ArrayIndexOutOfBoundsException etc. Unchecked exceptions are not checked at compile-time rather they are checked at runtime.

3) Error

Error is irrecoverable e.g. OutOfMemoryError, VirtualMachineError, AssertionError etc.

Common scenarios where exceptions may occur

There are given some scenarios where unchecked exceptions can occur. They are as follows:

1) Scenario where ArithmeticException occurs

If we divide any number by zero, there occurs an ArithmeticException.

```
1.  int a=50/0;//ArithmeticException
```

2) Scenario where NullPointerException occurs

If we have null value in any variable, performing any operation by the variable occurs an NullPointerException.

```
1.  String s=null;
2.  System.out.println(s.length());//NullPointerException
```

3) Scenario where NumberFormatException occurs

The wrong formatting of any value, may occur NumberFormatException. Suppose I have a string variable that have characters, converting this variable into digit will occur NumberFormatException.

```
1.  String s="abc";
2.  int i=Integer.parseInt(s);//NumberFormatException
```

4) Scenario where ArrayIndexOutOfBoundsException occurs

If you are inserting any value in the wrong index, it would result ArrayIndexOutOfBoundsException as shown below:

```
1.  int a[]=new int[5];
2.  a[10]=50; //ArrayIndexOutOfBoundsException
```

Java Exception Handling Keywords

There are 5 keywords used in java exception handling.

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

Java try-catch

Java try block

Java try block is used to enclose the code that might throw an exception. It must be used within the method.

Java try block must be followed by either catch or finally block.

Syntax of java try-catch

```
1.  try{
2.  //code that may throw exception
3.  }catch(Exception_class_Name ref){}
```

Syntax of try-finally block

```
1.  try{
2.  //code that may throw exception
3.  }finally{}
```

Java catch block

Java catch block is used to handle the Exception. It must be used after the try block only.

You can use multiple catch block with a single try.

Problem without exception handling

Let's try to understand the problem if we don't use try-catch block.

```
1.  public class Testtrycatch1{
2.    public static void main(String args[]){
3.       int data=50/0;//may throw exception
4.       System.out.println("rest of the code...");
5.  }
6.  }  Output:
```

```
Exception in thread main java.lang.ArithmeticException:/ by zero
```

As displayed in the above example, rest of the code is not executed (in such case, rest of the code... statement is not printed).

There can be 100 lines of code after exception. So all the code after exception will not be executed.

Solution by exception handling

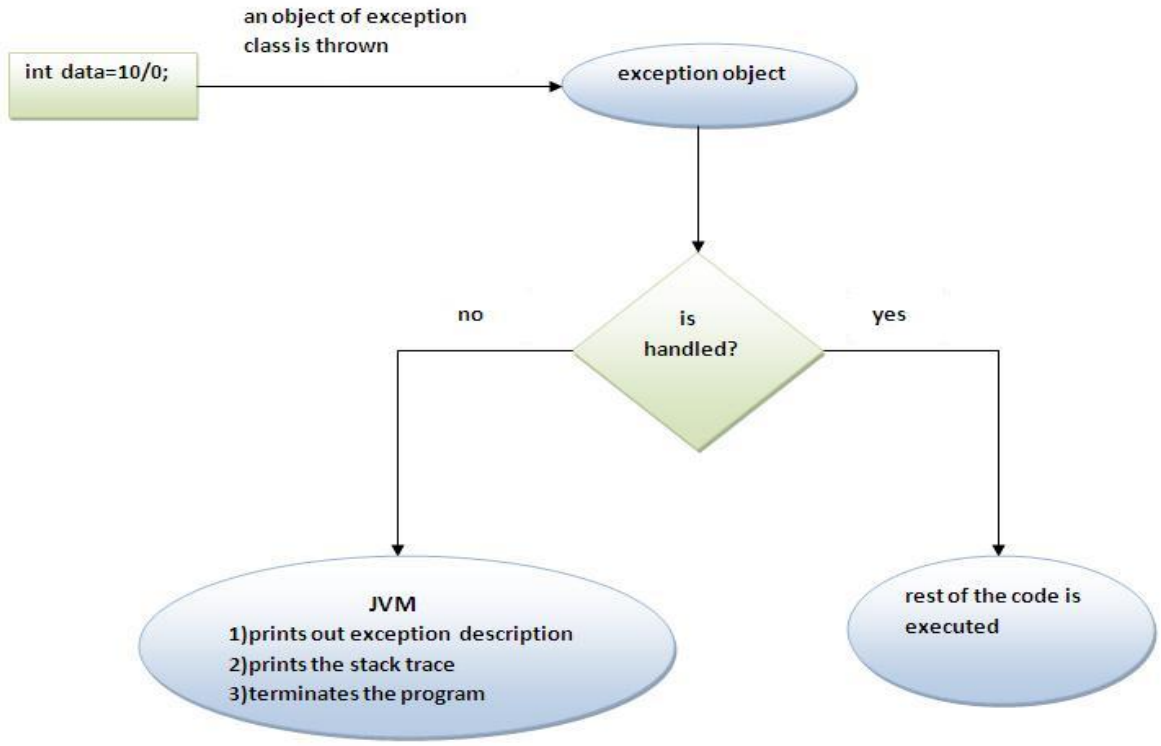
Let's see the solution of above problem by java try-catch block.

```
1.  public class Testtrycatch2{
2.    public static void main(String args[]){
3.     try{
4.       int data=50/0;
5.     }catch(ArithmeticException e){System.out.println(e);}
6.     System.out.println("rest of the code...");
7.  }
8.  }Output:
```

```
Exception in thread main java.lang.ArithmeticException:/ by zero
rest of the code...
```

Now, as displayed in the above example, rest of the code is executed i.e. rest of the code... statement is printed.

Internal working of java try-catch block



The JVM firstly checks whether the exception is handled or not. If exception is not handled, JVM provides a default exception handler that performs the following tasks:

- Prints out exception description.
- Prints the stack trace (Hierarchy of methods where the exception occurred).
- Causes the program to terminate.

But if exception is handled by the application programmer, normal flow of the application is maintained i.e. rest of the code is executed.

Java catch multiple exceptions

Java Multi catch block

If you have to perform different tasks at the occurrence of different Exceptions, use java multi catch block.

Let's see a simple example of java multi-catch block.

```
1.          public class TestMultipleCatchBlock{
2.            public static void main(String args[]){
3.             try{
4.              int a[]=new int[5];
5.              a[5]=30/0;
6.             }
7.             catch(ArithmeticException e){System.out.println("task1 is completed");}
8.             catch(ArrayIndexOutOfBoundsException e){System.out.println("task 2 completed");}

9.             catch(Exception e){System.out.println("common task completed");}
10.
11.            System.out.println("rest of the code...");
12.           }
13.          }
```

Output:

```
task1 completed
     rest of the code...
```

Rule: At a time only one Exception is occured and at a time only one catch block is executed.

Rule: All catch blocks must be ordered from most specific to most general i.e. catch for ArithmeticException must come before catch for Exception .

```
1.          class TestMultipleCatchBlock1{
2.            public static void main(String args[]){
3.             try{
4.              int a[]=new int[5];
5.              a[5]=30/0;
6.             }
7.             catch(Exception e){System.out.println("common task completed");}
8.             catch(ArithmeticException e){System.out.println("task1 is completed");}
9.             catch(ArrayIndexOutOfBoundsException e){System.out.println("task 2 completed");}

10.            System.out.println("rest of the code...");
11.           }
12.          }
```

Output:

```
Compile-time error
```

Java Nested try block

The try block within a try block is known as nested try block in java.

Why use nested try block

Sometimes a situation may arise where a part of a block may cause one error and the entire block itself may cause another error. In such cases, exception handlers have to be nested.

Syntax:

```
....
try
{
    statement 1;
    statement 2;
    try
    {
        statement 1;
        statement 2;
    }
    catch(Exception e)
    {
    }
}
catch(Exception e)
{
}
....
```

Java nested try example

Let's see a simple example of java nested try block.

```
class Excep6{
 public static void main(String args[]){
  try{
    try{
     System.out.println("going to divide");
     int b =39/0;
    }catch(ArithmeticException e){System.out.println(e);}

    try{
    int a[]=new int[5];
```

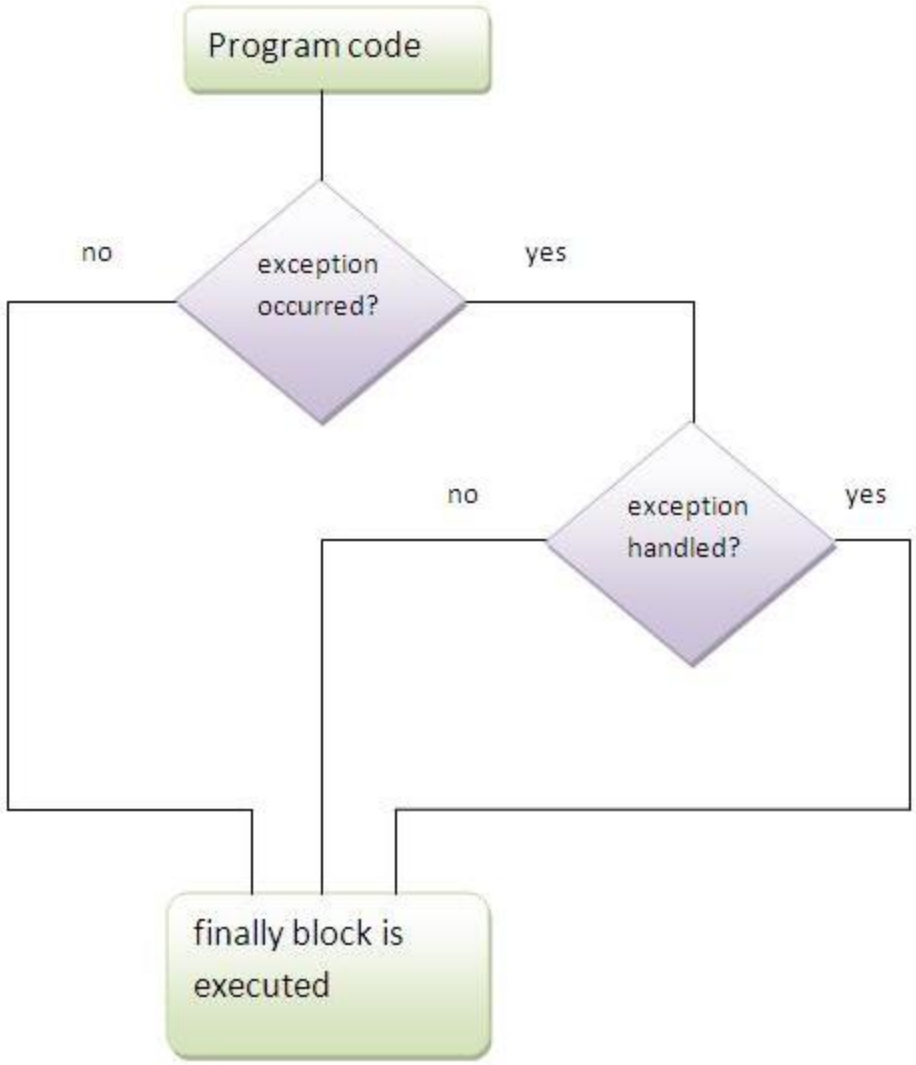
```
11.             a[5]=4;
12.             }catch(ArrayIndexOutOfBoundsException e){System.out.println(e);}
13.
14.             System.out.println("other statement);
15.            }catch(Exception e){System.out.println("handeled");}
16.
17.            System.out.println("normal flow..");
18.           }
19.          }
```

Java finally block

Java finally block is a block that is used to execute important code such as closing connection, stream etc.

Java finally block is always executed whether exception is handled or not.

Java finally block must be followed by try or catch block.



Note: If you don't handle exception, before terminating the program, JVM executes finally block(if any).

Why use java finally

- Finally block in java can be used to put "cleanup" code such as closing a file, closing connection etc.

Usage of Java finally

Let's see the different cases where java finally block can be used.

Case 1

Let's see the java finally example where **exception doesn't occur**.

```
1.  class TestFinallyBlock{
2.    public static void main(String args[]){
3.    try{
4.     int data=25/5;
5.     System.out.println(data);
6.    }
7.    catch(NullPointerException e){System.out.println(e);}
8.    finally{System.out.println("finally block is always executed");}
9.    System.out.println("rest of the code...");
10.   }
11. }
```

Output:5
finally block is always executed
rest of the code...

Case 2

Let's see the java finally example where **exception occurs and not handled**.

```
1.  class TestFinallyBlock1{
2.    public static void main(String args[]){
3.    try{
4.     int data=25/0;
5.     System.out.println(data);
6.    }
7.    catch(NullPointerException e){System.out.println(e);}
8.    finally{System.out.println("finally block is always executed");}
9.    System.out.println("rest of the code...");
10.   }
11. }
```

Output:finally block is always executed
Exception in thread main java.lang.ArithmeticException:/ by zero

Case 3

Let's see the java finally example where **exception occurs and handled**.

```
1.  public class TestFinallyBlock2{
2.    public static void main(String args[]){
3.    try{
4.     int data=25/0;
5.     System.out.println(data);
6.    }
7.    catch(ArithmeticException e){System.out.println(e);}
8.    finally{System.out.println("finally block is always executed");}
9.    System.out.println("rest of the code...");
10.   }
11. }
```

```
Output:Exception in thread main java.lang.ArithmeticException:/ by zero
      finally block is always executed
      rest of the code...
```

Rule: For each try block there can be zero or more catch blocks, but only one finally block.

Note: The finally block will not be executed if program exits(either by calling System.exit() or by causing a fatal error that causes the process to abort).

Java throw exception

Java throw keyword

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

We can throw either checked or uncheked exception in java by throw keyword. The throw keyword is mainly used to throw custom exception.

The syntax of java throw keyword is given below.

```
1.  throw exception;
```

Let's see the example of throw IOException.

```
1.  throw new IOException("sorry device error);
```

java throw keyword example

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

```
1.  public class TestThrow1{
2.    static void validate(int age){
3.      if(age<18)
4.        throw new ArithmeticException("not valid");
5.      else
6.        System.out.println("welcome to vote");
7.    }
8.    public static void main(String args[]){
9.      validate(13);
10.     System.out.println("rest of the code...");
11.   }
12. }
```

Output:

```
Exception in thread main java.lang.ArithmeticException:not valid
```

Java throws keyword

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

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

Syntax of java throws

```
1.  return_type method_name() throws exception_class_name{
2.  //method code
```

```
3.          }
```

Which exception should be declared

Ans) checked exception only, because:

- **unchecked Exception:** under your control so correct your code.
- **error:** beyond your control e.g. you are unable to do anything if there occurs VirtualMachineError or StackOverflowError.

Advantage of Java throws keyword

Now Checked Exception can be propagated (forwarded in call stack).

It provides information to the caller of the method about the exception.

Java throws example

Let's see the example of java throws clause which describes that checked exceptions can be propagated by throws keyword.

```
1.          import java.io.IOException;
2.          class Testthrows1{
3.            void m()throws IOException{
4.              throw new IOException("device error");//checked exception
5.            }
6.            void n()throws IOException{
7.              m();
8.            }
9.            void p(){
10.            try{
11.             n();
12.            }catch(Exception e){System.out.println("exception handled");}
13.           }
14.           public static void main(String args[]){
15.            Testthrows1 obj=new Testthrows1();
16.            obj.p();
17.            System.out.println("normal flow...");
18.           }
19.         }
```

Output:

```
exception handled
normal flow...
```

Rule: If you are calling a method that declares an exception, you must either caught or declare the exception.

There are two cases:

1. **Case1:**You caught the exception i.e. handle the exception using try/catch.
2. **Case2:**You declare the exception i.e. specifying throws with the method.

Case1: You handle the exception

- In case you handle the exception, the code will be executed fine whether exception occurs during the program or not.

```
1.  import java.io.*;
2.  class M{
3.   void method()throws IOException{
4.    throw new IOException("device error");
5.   }
6.  }
7.  public class Testthrows2{
8.    public static void main(String args[]){
9.     try{
10.     M m=new M();
11.     m.method();
12.     }catch(Exception e){System.out.println("exception handled");}
13.
14.     System.out.println("normal flow...");
15.   }
16. }
```

```
Output:exception handled
       normal flow...
```

Case2: You declare the exception

- A)In case you declare the exception, if exception does not occur, the code will be executed fine.
- B)In case you declare the exception if exception occures, an exception will be thrown at runtime because throws does not handle the exception.

A)Program if exception does not occur

```
1.  import java.io.*;
2.  class M{
3.   void method()throws IOException{
4.    System.out.println("device operation performed");
5.   }
6.  }
7.  class Testthrows3{
8.     public static void main(String args[])throws IOException{//declare exception
9.       M m=new M();
10.      m.method();
11.
12.     System.out.println("normal flow...");
13.   }
14. }
```

Output:device operation performed
normal flow...

B)Program if exception occurs

```
1.  import java.io.*;
2.  class M{
3.   void method()throws IOException{
4.    throw new IOException("device error");
5.   }
6.  }
7.  class Testthrows4{
8.     public static void main(String args[])throws IOException{//declare exception
9.       M m=new M();
10.      m.method();
11.
12.     System.out.println("normal flow...");
13.   }
14. }
```

Output:Runtime Exception

Difference between throw and throws

Que) Can we rethrow an exception?

Yes, by throwing same exception in catch block.

Difference between throw and throws in Java

There are many differences between throw and throws keywords. A list of differences between throw and throws are given below:

No.	throw	throws
1)	Java throw keyword is used to explicitly throw an exception.	Java throws keyword is used to declare an exception.
2)	Checked exception cannot be propagated using throw only.	Checked exception can be propagated with throws.
3)	Throw is followed by an instance.	Throws is followed by class.
4)	Throw is used within the method.	Throws is used with the method signature.
5)	You cannot throw multiple exceptions.	You can declare multiple exceptions e.g. public void method()throws IOException,SQLException.

Java throw example

```
1.  void m(){
2.  throw new ArithmeticException("sorry");
3.  }
```

Java throws example

```
1.  void m()throws ArithmeticException{
2.  //method code
3.  }
```

Java throw and throws example

```
1.  void m()throws ArithmeticException{
2.  throw new ArithmeticException("sorry");
3.  }
```

Difference between final, finally and finalize

There are many differences between final, finally and finalize. A list of differences between final, finally and finalize are given below:

No.	final	finally	finalize
1)	Final is used to apply restrictions on class, method and variable. Final class can't be inherited, final method can't be overridden and final variable value can't be changed.	Finally is used to place important code, it will be executed whether exception is handled or not.	Finalize is used to perform clean up processing just before object is garbage collected.
2)	Final is a keyword.	Finally is a block.	Finalize is a method.

Java final example

```
1.  class FinalExample{
2.  public static void main(String[] args){
3.  final int x=100;
4.  x=200;//Compile Time Error
5.  }}
```

Java finally example

```
1.  class FinallyExample{
2.  public static void main(String[] args){
3.  try{
4.  int x=300;
5.  }catch(Exception e){System.out.println(e);}
6.  finally{System.out.println("finally block is executed");}
7.  }}
```

Java finalize example

```
1.  class FinalizeExample{
2.  public void finalize(){System.out.println("finalize called");}
3.  public static void main(String[] args){
4.  FinalizeExample f1=new FinalizeExample();
5.  FinalizeExample f2=new FinalizeExample();
6.  f1=null;
7.  f2=null;
8.  System.gc();
9.  }}
```

Java Custom Exception

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

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

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

```
1.  class InvalidAgeException extends Exception{
2.   InvalidAgeException(String s){
3.    super(s);
4.   }
5.  }
1.  class TestCustomException1{
2.
3.    static void validate(int age)throws InvalidAgeException{
4.     if(age<18)
5.      throw new InvalidAgeException("not valid");
6.     else
7.      System.out.println("welcome to vote");
8.    }
9.
10.   public static void main(String args[]){
11.     try{
12.     validate(13);
13.     }catch(Exception m){System.out.println("Exception occured: "+m);}
14.
15.     System.out.println("rest of the code...");
16.   }
17. }
```

Output:Exception occured: InvalidAgeException:not valid
rest of the code...

Applet

Applet is a special type of program that is embedded in the webpage to generate the dynamic content. It runs inside the browser and works at client side.

An applet is a Java program that runs in a Web browser. An applet can be a fully functional Java application because it has the entire Java API at its disposal.

There are some important differences between an applet and a standalone Java application, including the following:

- An applet is a Java class that extends the java.applet.Applet class.
- A main() method is not invoked on an applet, and an applet class will not define main().
- Applets are designed to be embedded within an HTML page.
- When a user views an HTML page that contains an applet, the code for the applet is downloaded to the user's machine.
- A JVM is required to view an applet. The JVM can be either a plug-in of the Web browser or a separate runtime environment.
- The JVM on the user's machine creates an instance of the applet class and invokes various methods during the applet's lifetime.
- Applets have strict security rules that are enforced by the Web browser. The security of an applet is often referred to as sandbox security, comparing the applet to a child playing in a sandbox with various rules that must be followed.
- Other classes that the applet needs can be downloaded in a single Java Archive (JAR) file.

Advantages of Applet

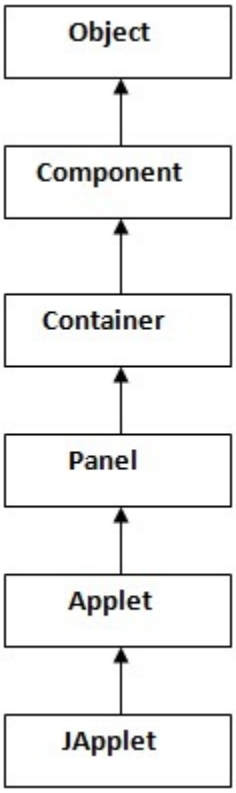
There are many advantages of applet. They are as follows:

- It works at client side so less response time.
- Secured
- It can be executed by browsers running under many plateforms, including Linux, Windows, Mac Os etc.

Drawback of Applet

- Plugin is required at client browser to execute applet.

Hierarchy of Applet



As displayed in the above diagram, Applet class extends Panel. Panel class extends Container which is the subclass of Component.

Lifecycle of Java Applet

1. Applet is initialized.
2. Applet is started.
3. Applet is painted.
4. Applet is stopped.
5. Applet is destroyed.

Lifecycle methods for Applet:

The java.applet.Applet class 4 life cycle methods and java.awt.Component class provides 1 life cycle methods for an applet.

java.applet.Applet class

For creating any applet java.applet.Applet class must be inherited. It provides 4 life cycle methods of applet.

1. **public void init():** is used to initialized the Applet. It is invoked only once.
2. **public void start():** is invoked after the init() method or browser is maximized. It is used to start the Applet.
3. **public void stop():** is used to stop the Applet. It is invoked when Applet is stop or browser is minimized.
4. **public void destroy():** is used to destroy the Applet. It is invoked only once.

Four methods in the Applet class give you the framework on which you build any serious applet:

- **init:** This method is intended for whatever initialization is needed for your applet. It is called after the param tags inside the applet tag have been processed.
- **start:** This method is automatically called after the browser calls the init method. It is also called whenever the user returns to the page containing the applet after having gone off to other pages.
- **stop:** This method is automatically called when the user moves off the page on which the applet sits. It can, therefore, be called repeatedly in the same applet.
- **destroy:** This method is only called when the browser shuts down normally. Because applets are meant to live on an HTML page, you should not normally leave resources behind after a user leaves the page that contains the applet.
- **paint:** Invoked immediately after the start() method, and also any time the applet needs to repaint itself in the browser. The paint() method is actually inherited from the java.awt.

Displaying Graphics in Applet

java.awt.Graphics class provides many methods for graphics programming.

Commonly used methods of Graphics class:

1. **public abstract void drawString(String str, int x, int y):** is used to draw the specified string.
2. **public void drawRect(int x, int y, int width, int height):** draws a rectangle with the specified width and height.
3. **public abstract void fillRect(int x, int y, int width, int height):** is used to fill rectangle with the default color and specified width and height.
4. **public abstract void drawOval(int x, int y, int width, int height):** is used to draw oval with the specified width and height.
5. **public abstract void fillOval(int x, int y, int width, int height):** is used to fill oval with the default color and specified width and height.
6. **public abstract void drawLine(int x1, int y1, int x2, int y2):** is used to draw line between the points(x1, y1) and (x2, y2).
7. **public abstract boolean drawImage(Image img, int x, int y, ImageObserver observer):** is used draw the specified image.
8. **public abstract void drawArc(int x, int y, int width, int height, int startAngle, int arcAngle):** is used draw a circular or elliptical arc.
9. **public abstract void fillArc(int x, int y, int width, int height, int startAngle, int arcAngle):** is used to fill a circular or elliptical arc.
10. **public abstract void setColor(Color c):** is used to set the graphics current color to the specified color.
11. **public abstract void setFont(Font font):** is used to set the graphics current font to the specified font.

Displaying Image in Applet

Applet is mostly used in games and animation. For this purpose image is required to be displayed. The java.awt.Graphics class provide a method drawImage() to display the image.

Syntax of drawImage() method:

1. **public abstract boolean drawImage(Image img, int x, int y, ImageObserver observer):** is used draw the specified image.

How to get the object of Image:

The java.applet.Applet class provides getImage() method that returns the object of Image.
Syntax:

1. **public** Image getImage(URL u, String image){}

Other required methods of Applet class to display image:

1. **public URL getDocumentBase():** is used to return the URL of the document in which applet is embedded.
2. **public URL getCodeBase():** is used to return the base URL.

```
1.  import java.awt.*;
2.  import java.applet.*;
3.
4.
5.  public class DisplayImage extends Applet {
6.
7.    Image picture;
8.
9.    public void init() {
10.     picture = getImage(getDocumentBase(),"java.jpg");
11.   }
12.
13.   public void paint(Graphics g) {
14.     g.drawImage(picture, 30,30, this);
15.   }
16.
17.   }
```

myapplet.html

```
1.  <html>
2.  <body>
3.  <applet code="DisplayImage.class" width="300" height="300">
4.  </applet>
5.  </body>
6.  </html>
```

Displaying Image in Applet

Applet is mostly used in games and animation. For this purpose image is required to be displayed. The java.awt.Graphics class provide a method drawImage() to display the image.

Syntax of drawImage() method:

1. **public abstract boolean drawImage(Image img, int x, int y, ImageObserver observer):** is used draw the specified image.

How to get the object of Image:

The java.applet.Applet class provides getImage() method that returns the object of Image.

Syntax:

```
1.  public Image getImage(URL u, String image){}
```

Other required methods of Applet class to display image:

1. **public URL getDocumentBase():** is used to return the URL of the document in which applet is embedded.
2. **public URL getCodeBase():** is used to return the base URL.

Example of displaying image in applet:

```
1.  import java.awt.*;
2.  import java.applet.*;
3.
4.
5.  public class DisplayImage extends Applet {
6.
7.    Image picture;
8.
9.    public void init() {
10.     picture = getImage(getDocumentBase(),"twitter.jpg");
11.   }
12.
13.   public void paint(Graphics g) {
14.     g.drawImage(picture, 30,30, this);
15.   }
16.   }
```

In the above example, drawImage() method of Graphics class is used to display the image. The 4th argument of drawImage() method of is ImageObserver object. The Component class implements ImageObserver interface. So current class object would also be treated as ImageObserver because Applet class indirectly extends the Component class.

myapplet.html

```
1.  <html>
2.  <body>
3.  <applet code="DisplayImage.class" width="300" height="300">
4.  </applet>
5.  </body>
```

6. </html>

Creating AudioClip from an Audio File

To play an audio file in an applet, first create an audio clip object for the audio file. The audio clip is created once and can be played repeatedly without reloading the file. To create an audio clip, use the static method newAudioClip() in the java.applet.Applet class:

AudioClip audioClip = Applet.newAudioClip(url);

Audio was originally used with Java applets. For this reason, the AudioClip interface is in the java.applet package.

The following statements, for example, create an AudioClip for the beep.au audio file in the same directory with the class you are running.

Class class = this.getClass();

URL url = class.getResource("beep.au");

AudioClip audioClip = Applet.newAudioClip(url);

Playing Audio

«interface» *java.applet.AudioClip*	
+play()	Starts playing this audio clip. Each time this method is called, the clip is restarted from the beginning.
+loop()	Plays the clip repeatedly.
+stop()	Stops playing the clip.

Here's an example to load audio file ("ding.wav") in Applet.

```
package com.mkyong.applet;

import java.applet.Applet;
import java.applet.AudioClip;
import java.awt.Button;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
```

```
public class LoadSoundApplet extends Applet implements ActionListener {

	Button play, stop;
	AudioClip audioClip;

	private static final String PLAY = "PLAY";
	private static final String STOP = "STOP";

	public void init(){
		play = new Button();
		play.setLabel(PLAY);
		play.setActionCommand(PLAY);
		play.addActionListener(this);
		add(play);

		stop = new Button();
		stop.setLabel(STOP);
		stop.setActionCommand(STOP);
		stop.addActionListener(this);
		add(stop);

		audioClip = getAudioClip(getCodeBase(), "ding.wav");
	}

	@Override
	public void actionPerformed(ActionEvent e) {

		if(e.getActionCommand().equals(PLAY)){
			audioClip.play();
		}else if(e.getActionCommand().equals(STOP)){
			audioClip.stop();
		}else{
			audioClip.stop();
		}

	}
}
```

2. Create a HTML

Create a HTML file to include the Applet.

```
<html>
<head><title>Testing</title></head>
<body>

<h1>Applet Load Sound</h1>
<applet width=300 height=100 code="com.mkyong.applet.LoadSoundApplet.class">
</applet>

</body>
</html>
```

3. Output

After you clicked on the Play button, Applet will start to play the "ding.wav"

Event and Listener (Java Event Handling)

Changing the state of an object is known as an event. For example, click on button, dragging mouse etc. The java.awt.event package provides many event classes and Listener interfaces for event handling.

Event classes and Listener interfaces:

Event Classes	Listener Interfaces
ActionEvent	ActionListener
MouseEvent	MouseListener and MouseMotionListener
MouseWheelEvent	MouseWheelListener
KeyEvent	KeyListener
ItemEvent	ItemListener
TextEvent	TextListener
AdjustmentEvent	AdjustmentListener
WindowEvent	WindowListener
ComponentEvent	ComponentListener
ContainerEvent	ContainerListener
FocusEvent	FocusListener

Steps to perform Event Handling

Following steps are required to perform event handling:

1. Implement the Listener interface and overrides its methods
2. Register the component with the Listener

For registering the component with the Listener, many classes provide the registration methods. For example:

- **Button**
  - public void addActionListener(ActionListener a){}
- **MenuItem**
  - public void addActionListener(ActionListener a){}
- **TextField**
  - public void addActionListener(ActionListener a){}
  - public void addTextListener(TextListener a){}
- **TextArea**
  - public void addTextListener(TextListener a){}
- **Checkbox**

- public void addItemListener(ItemListener a){}
- **Choice**
  - public void addItemListener(ItemListener a){}
- **List**
  - public void addActionListener(ActionListener a){}
  - public void addItemListener(ItemListener a){}

EventHandling Codes:

We can put the event handling code into one of the following places:

1. Same class
2. Other class
3. Annonymous class

Example of event handling within class:

```
1.  import java.awt.*;
2.  import java.awt.event.*;
3.
4.  class AEvent extends Frame implements ActionListener{
5.  TextField tf;
6.  AEvent(){
7.
8.  tf=new TextField();
9.  tf.setBounds(60,50,170,20);
10.
11. Button b=new Button("click me");
12. b.setBounds(100,120,80,30);
13.
14. b.addActionListener(this);
15.
16. add(b);add(tf);
17.
18. setSize(300,300);
19. setLayout(null);
20. setVisible(true);
21.
22. }
23.
24. public void actionPerformed(ActionEvent e){
25. tf.setText("Welcome");
26. }
27.
28. public static void main(String args[]){
29. new AEvent();
```

30. }
31. }

public void setBounds(int xaxis, int yaxis, int width, int height); have been used in the above example that sets the position of the component it may be button, textfield etc.

Example of Adding Listeners in Applet

```
import java.awt.*;
import java.applet.*;
import javax.swing.*;
import java.awt.event.*;
import java.awt.event.KeyListener;
import java.awt.event.KeyEvent;


public class guitarGame extends Applet implements ActionListener, KeyListener {
AudioClip brainStew;
Timer timer = new Timer (1000, this);

public void init()
{
   brainStew = getAudioClip(getDocumentBase(), "green day - brain stew.au");
   brainStew.play();

}

public void keyReleased(KeyEvent ae){}
```

```
public void keyPressed(KeyEvent ae){

   repaint();
}

public void keyTyped(KeyEvent ae){}

public void actionPerformed (ActionEvent ae){}
public void paint (Graphics g)
{
}
}
```

What is Event Handling?

Event Handling is the mechanism that controls the event and decides what should happen if an event occurs. This mechanism have the code which is known as event handler that is executed when an event occurs. Java Uses the Delegation Event Model to handle the events. This model defines the standard mechanism to generate and handle the events. Let's have a brief introduction to this model.

The Delegation Event Model has the following key participants namely:

- **Source** - The source is an object on which event occurs. Source is responsible for providing information of the occurred event to it's handler. Java provide as with classes for source object.

- **Listener** - It is also known as event handler. Listener is responsible for generating response to an event. From java implementation point of view the listener is also an object. Listener waits until it receives an event. Once the event is received , the listener process the event an then returns.

The benefit of this approach is that the user interface logic is completely separated from the logic that generates the event. The user interface element is able to delegate the processing of an event to the separate piece of code. In this model ,Listener needs to be registered with the source object so that the listener can receive the event notification. This is an efficient way of handling the event because the event notifications are sent only to those listener that want to receive them.

Steps involved in event handling

- The User clicks the button and the event is generated.
- Now the object of concerned event class is created automatically and information about the source and the event get populated with in same object.
- Event object is forwarded to the method of registered listener class.
- the method is now get executed and returns.

OR

Java's delegation event model

The event model is based on the Event Source and Event Listeners. Event Listener is an object that receives the messages / events. The Event Source is any object which creates the message / event. The Event Delegation model is based on – The Event Classes, The Event Listeners, Event Objects.

There are three participants in event delegation model in Java;

- Event Source – the class which broadcasts the events
- Event Listeners – the classes which receive notifications of events
- Event Object – the class object which describes the event.

An event occurs (like mouse click, key press, etc) which is followed by the event is broadcasted by the event source by invoking an agreed method on all event listeners. The event object is passed as argument to the agreed-upon method. Later the event listeners respond as they fit, like submit a form, displaying a message / alert etc.

Keyboard Event

```
import java.awt.*;
import java.awt.event.*;
import java.applet.*;
/*
<applet code="Key" width=300 height=400>
</applet>
*/
public class Key extends Applet
implements KeyListener
{
        int X=20,Y=30;
        String msg="KeyEvents--->";
```

```
	public void init()
	{
		addKeyListener(this);
		requestFocus();
		setBackground(Color.green);
		setForeground(Color.blue);
	}
	public void keyPressed(KeyEvent k)
	{
		showStatus("KeyDown");
		int key=k.getKeyCode();
		switch(key)
		{
			case KeyEvent.VK_UP:
				showStatus("Move to Up");
				break;
			case KeyEvent.VK_DOWN:
				showStatus("Move to Down");
				break;
			case KeyEvent.VK_LEFT:
				showStatus("Move to Left");
				break;
			case KeyEvent.VK_RIGHT:
				showStatus("Move to Right");
				break;
		}
		repaint();
	}
	public void keyReleased(KeyEvent k)
	{
		showStatus("Key Up");
	}
	public void keyTyped(KeyEvent k)
	{
		msg+=k.getKeyChar();
		repaint();
	}
	public void paint(Graphics g)
	{
		g.drawString(msg,X,Y);
	}
}
```

OR

```
import java.awt.*;
```

```
import java.awt.event.*;
import java.applet.*;
import java.applet.*;
import java.awt.event.*;
import java.awt.*;

public class Test extends Applet implements KeyListener
{
 String msg="";
 public void init()
 {
  addKeyListener(this);
 }
 public void keyPressed(KeyEvent k)
 {
  showStatus("KeyPressed");
 }
 public void keyReleased(KeyEvent k)
 {
  showStatus("KeyRealesed");
 }
 public void keyTyped(KeyEvent k)
 {
  msg = msg+k.getKeyChar();
  repaint();
 }
 public void paint(Graphics g)
 {
  g.drawString(msg, 20, 40);
 }
}
```

HTML code :

```
< applet code="Test" width=300, height=100 >
```

Mouse Event

```
class ColorChangeCanvas extends Canvas implements MouseListener {

        int currentColor = 0;  // 0 for red, 1 for blue, 2 for green

        ColorChangeCanvas() { // constructor
          setBackground(Color.red);
```

```
    addMouseListener(this);  // Canvas will listen for
                             //   its own mouse events.
  }

  public void mousePressed(MouseEvent evt) {
      // User has clicked on the canvas;
      // cycle its background color and repaint it.
    currentColor++;
    if (currentColor > 2)
      currentColor = 0;
    switch (currentColor) {
      case 0: setBackground(Color.red); break;
      case 1: setBackground(Color.blue); break;
      case 2: setBackground(Color.green); break;
    }
    repaint();
  }

  public void mouseReleased(MouseEvent evt) { }   // Definitions required
  public void mouseClicked(MouseEvent evt) { }    //   by the MouseListener
  public void mouseEntered(MouseEvent evt) { }    //   interface.
  public void mouseExited(MouseEvent evt) { }

}
```

Java Swing

Java Swing tutorial is a part of Java Foundation Classes (JFC) that is *used to create window-based applications*. It is built on the top of AWT (Abstract Windowing Toolkit) API and entirely written in java.

Unlike AWT, Java Swing provides platform-independent and lightweight components.

The javax.swing package provides classes for java swing API such as JButton, JTextField, JTextArea, JRadioButton, JCheckbox, JMenu, JColorChooser etc.

Difference between AWT and Swing

There are many differences between java awt and swing that are given below.

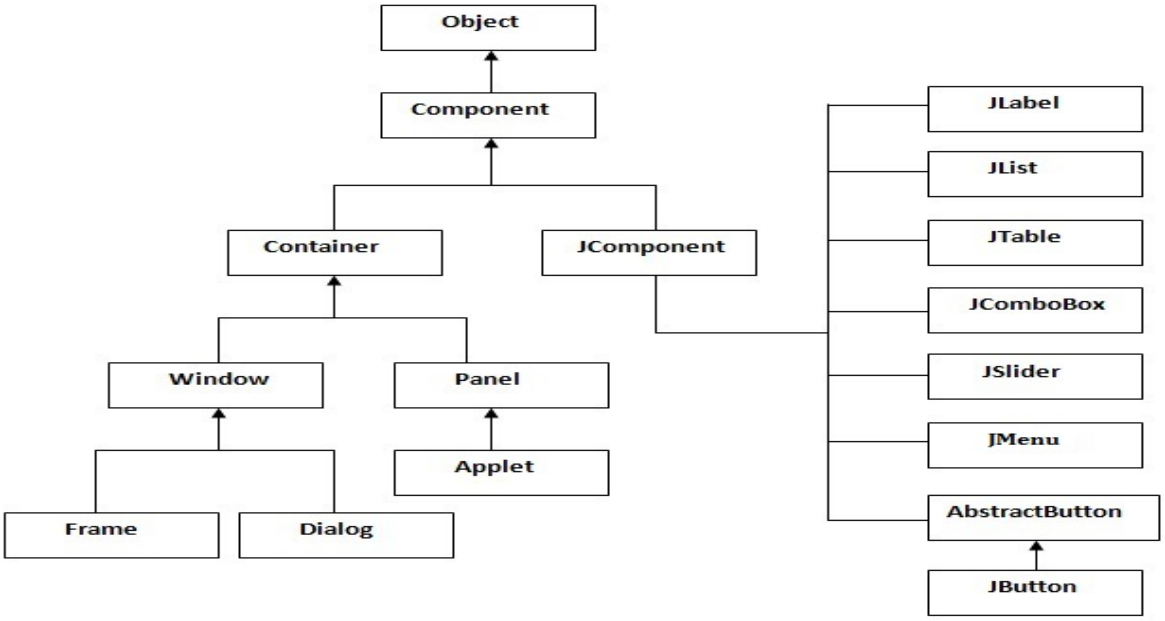
No.	Java AWT	Java Swing
1)	AWT components are **platform-dependent**.	Java swing components are **platform-independent**.
2)	AWT components are **heavyweight**.	Swing components are **lightweight**.
3)	AWT **doesn't support pluggable look and feel**.	Swing **supports pluggable look and feel**.
4)	AWT provides **less components** than Swing.	Swing provides **more powerful components** such as tables, lists, scrollpanes, colorchooser, tabbedpane etc.
5)	AWT **doesn't follows MVC**(Model View Controller) where model represents data, view represents presentation and controller acts as an interface between model and view.	Swing **follows MVC**.

What is JFC

The Java Foundation Classes (JFC) are a set of GUI components which simplify the development of desktop applications.

Hierarchy of Java Swing classes

The hierarchy of java swing API is given below.



Commonly used Methods of Component class

The methods of Component class are widely used in java swing that are given below.

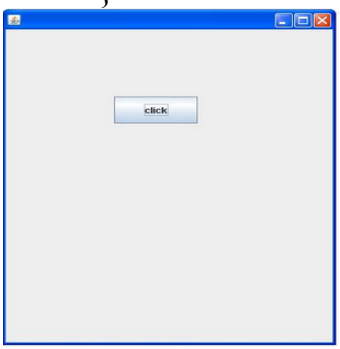
Method	Description
public void add(Component c)	add a component on another component.
public void setSize(int width,int height)	sets size of the component.
public void setLayout(LayoutManager m)	sets the layout manager for the component.
public void setVisible(boolean b)	sets the visibility of the component. It is by default false.

Simple Java Swing Example

Let's see a simple swing example where we are creating one button and adding it on the JFrame object inside the main() method.

File: FirstSwingExample.java

```
1.      import javax.swing.*;
2.      public class FirstSwingExample {
3.      public static void main(String[] args) {
4.      JFrame f=new JFrame();//creating instance of JFrame
5.
6.      JButton b=new JButton("click");//creating instance of JButton
7.      b.setBounds(130,100,100, 40);//x axis, y axis, width, height
8.
9.      f.add(b);//adding button in JFrame
10.
11.     f.setSize(400,500);//400 width and 500 height
12.     f.setLayout(null);//using no layout managers
13.     f.setVisible(true);//making the frame visible
14.     }
15.     }
```



Adapter Class

In java programming language, adapter class is used to implement an interface having a set of dummy methods. The developer can then further subclass the adapter class so that he can override to the methods he requires. Implementing an interface directly, requires to write all the dummy methods. In general an adapter class is used to rapidly construct your own Listener class to field events.

Java provides a special feature, called an adapter class, that can simplify the creation of event handlers in certain situations.An adapter class provides an empty implementation of all methods in an event listener interface i.e this class itself write definition for methods which are present in particular event listener interface. However these definitions does not affect program flow or meaning at all. Adapter classes are useful when you want to receive and process only some of the events that are handled by a particular event listener interface. You can define a new class to act as an event listener by extending one of the adapter classes and implementing only those events in which you are interested.
E.g. Suppose you want to use MouseClicked Event or method from MouseListener, if you do not

use adapter class then unnecessarily you have to define all other methods from MouseListener such as MouseReleased, MousePressed etc.
But If you use adapter class then you can only define MouseClicked method and don't worry about other method definition because class provides an empty implementation of all methods in an event listener interface.

- Java provides a special feature, called an *adapter class*, that can simplify the creation of event handlers in certain situations.
- An adapter class provides an empty implementation of all methods in an event listener interface i.e this class itself write definition for methods which are present in particular event listener interface. However these definitions does not affect program flow or meaning at all.
- Adapter classes are useful when you want to receive and process only some of the events that are handled by a particular event listener interface.
- You can define a new class to act as an event listener by extending one of the adapter classes and implementing only those events in which you are interested.
- E.g. Suppose you want to use MouseClicked Event or method from MouseListener, if you do not use adapter class then unnecessarily you have to define all other methods from MouseListener such as MouseReleased, MousePressed etc.
- But If you use adapter class then you can only define MouseClicked method and don't worry about other method definition because class provides an empty implementation of all methods in an event listener interface.
- Below Table Indicates Listener Interface with their respective adapter class.

Listener Interface	Adapter Class
ComponentListener	ComponentAdapter
ContainerListener	ContainerAdapter
FocousListener	FocousAdapter
KeyListener	KeyAdapter
MouseListener	MouseAdapter
MouseMotionListener	MouseMotionAdapter
WindowListener	WindowAdapter

Multithreading in Java

Multithreading in java is a process of executing multiple threads simultaneously.

Thread is basically a lightweight sub-process, a smallest unit of processing. Multiprocessing and multithreading, both are used to achieve multitasking.

But we use multithreading than multiprocessing because threads share a common memory area. They don't allocate separate memory area so saves memory, and context-switching between the threads takes less time than process.

Java Multithreading is mostly used in games, animation etc.

Advantage of Java Multithreading

1) It **doesn't block the user** because threads are independent and you can perform multiple operations at same time.

2) You **can perform many operations together so it saves time**.

3) Threads are **independent** so it doesn't affect other threads if exception occur in a single thread.

Multitasking

Multitasking is a process of executing multiple tasks simultaneously. We use multitasking to utilize the CPU. Multitasking can be achieved by two ways:

- Process-based Multitasking(Multiprocessing)
- Thread-based Multitasking(Multithreading)

1) Process-based Multitasking (Multiprocessing)

- Each process have its own address in memory i.e. each process allocates separate memory area.
- Process is heavyweight.
- Cost of communication between the process is high.
- Switching from one process to another require some time for saving and loading registers, memory maps, updating lists etc.

2) Thread-based Multitasking (Multithreading)

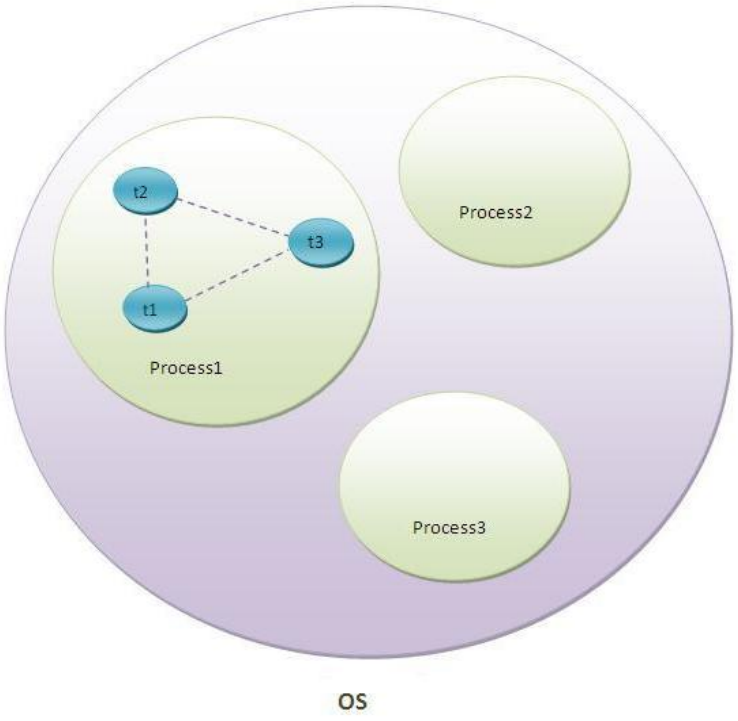
- Threads share the same address space.
- Thread is lightweight.
- Cost of communication between the thread is low.

Note: At least one process is required for each thread.

What is Thread in java

A thread is a lightweight sub process, a smallest unit of processing. It is a separate path of execution.

Threads are independent, if there occurs exception in one thread, it doesn't affect other threads. It shares a common memory area.



As shown in the above figure, thread is executed inside the process. There is context-switching between the threads. There can be multiple processes inside the OS and one process can have multiple threads.

Note: At a time one thread is executed only.

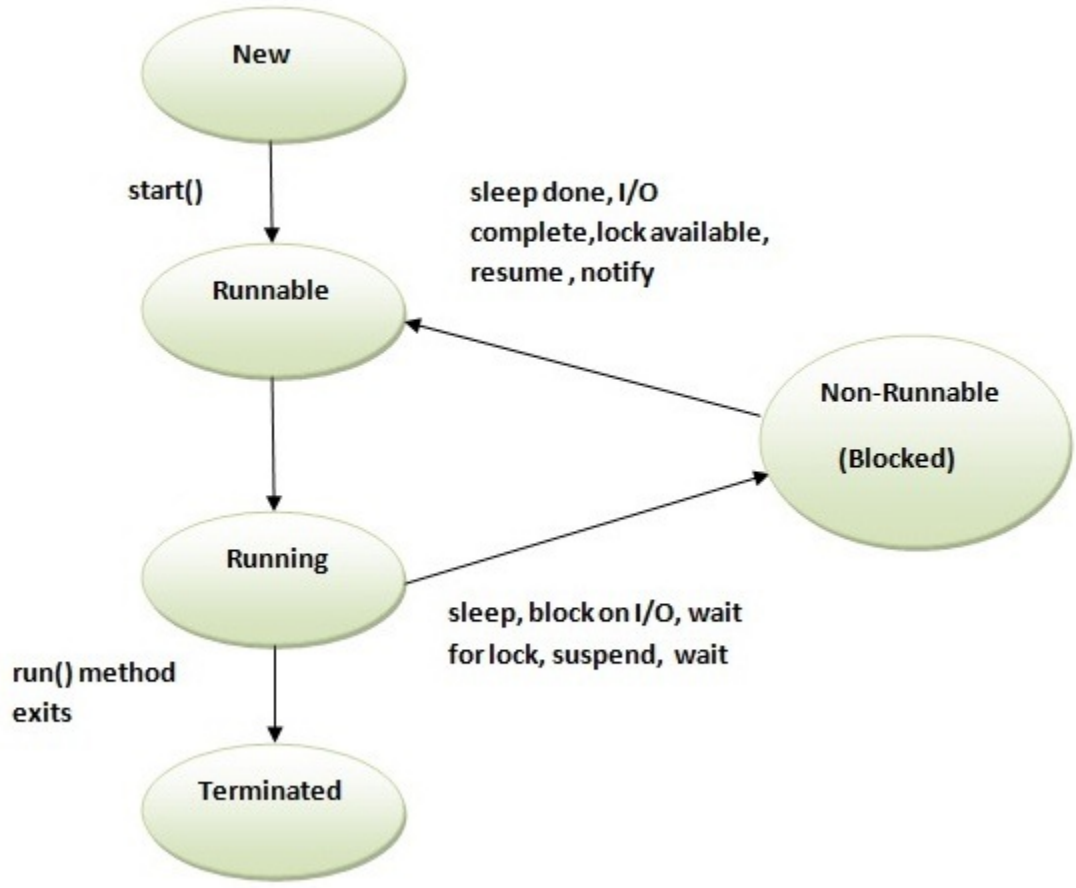
Life cycle of a Thread (Thread States)

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

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

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

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



1) New

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

2) Runnable

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

3) Running

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

4) Non-Runnable (Blocked)

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

5) Terminated

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

How to create thread

There are two ways to create a thread:

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

Thread class:

Thread class provide constructors and methods to create and perform operations on a thread.Thread class extends Object class and implements Runnable interface.

Commonly used Constructors of Thread class:

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

Commonly used methods of Thread class:

1. **public void run():** is used to perform action for a thread.
2. **public void start():** starts the execution of the thread.JVM calls the run() method on the thread.
3. **public void sleep(long miliseconds):** Causes the currently executing thread to sleep (temporarily cease execution) for the specified number of milliseconds.
4. **public void join():** waits for a thread to die.
5. **public void join(long miliseconds):** waits for a thread to die for the specified miliseconds.
6. **public int getPriority():** returns the priority of the thread.
7. **public int setPriority(int priority):** changes the priority of the thread.
8. **public String getName():** returns the name of the thread.
9. **public void setName(String name):** changes the name of the thread.
10. **public Thread currentThread():** returns the reference of currently executing thread.
11. **public int getId():** returns the id of the thread.
12. **public Thread.State getState():** returns the state of the thread.
13. **public boolean isAlive():** tests if the thread is alive.
14. **public void yield():** causes the currently executing thread object to temporarily pause and allow other threads to execute.
15. **public void suspend():** is used to suspend the thread(depricated).
16. **public void resume():** is used to resume the suspended thread(depricated).
17. **public void stop():** is used to stop the thread(depricated).
18. **public boolean isDaemon():** tests if the thread is a daemon thread.
19. **public void setDaemon(boolean b):** marks the thread as daemon or user thread.
20. **public void interrupt():** interrupts the thread.
21. **public boolean isInterrupted():** tests if the thread has been interrupted.
22. **public static boolean interrupted():** tests if the current thread has been interrupted.

Runnable Interface:

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

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

Starting A Thread:

start() method of Thread class is used to start a newly created thread. It performs following tasks:

- A new thread starts(with new callstack).
- The thread moves from New state to the Runnable state.
- When the thread gets a chance to execute, its target run() method will run.

1) **By extending Thread class:**

```
1.  class Multi extends Thread{
2.  public void run(){
3.  System.out.println("thread is running...");
4.  }
5.  public static void main(String args[]){
6.  Multi t1=new Multi();
7.  t1.start();
8.   }
9.  }
```

Output:thread is running...

Who makes your class object as thread object?

Thread class constructor allocates a new thread object.When you create object of Multi class,your class constructor is invoked(provided by Compiler) fromwhere Thread class constructor is invoked(by super() as first statement).So your Multi class object is thread object now.

2) **By implementing the Runnable interface:**

```
1.  class Multi3 implements Runnable{
2.  public void run(){
3.  System.out.println("thread is running...");
4.  }
5.
6.  public static void main(String args[]){
7.  Multi3 m1=new Multi3();
8.  Thread t1 =new Thread(m1);
9.  t1.start();
10.  }
11.  }
```

Output:thread is running...

If you are not extending the Thread class,your class object would not be treated as a thread object.So you need to explicitely create Thread class object.We are passing the object of your class that implements Runnable so that your class run() method may execute.

Sleep method in java

The sleep() method of Thread class is used to sleep a thread for the specified amount of time.

Syntax of sleep() method in java

The Thread class provides two methods for sleeping a thread:

- public static void sleep(long miliseconds)throws InterruptedException
- public static void sleep(long miliseconds, int nanos)throws InterruptedException

Example of sleep method in java

```
1.  class TestSleepMethod1 extends Thread{
2.   public void run(){
3.    for(int i=1;i<5;i++){
4.      try{Thread.sleep(500);}catch(InterruptedException e){System.out.println(e);}
5.      System.out.println(i);
6.    }
7.   }
8.   public static void main(String args[]){
9.    TestSleepMethod1 t1=new TestSleepMethod1();
10.   TestSleepMethod1 t2=new TestSleepMethod1();
11.
12.   t1.start();
13.   t2.start();
14.  }
15. }
```

Output:

1
1
2
2
3
3
4
4

As you know well that at a time only one thread is executed. If you sleep a thread for the specified time,the thread shedular picks up another thread and so on.

Can we start a thread twice

No. After starting a thread, it can never be started again. If you does so, an *IllegalThreadStateException* is thrown. In such case, thread will run once but for second time, it will throw exception.

Let's understand it by the example given below:

```
1.  public class TestThreadTwice1 extends Thread{
2.   public void run(){
3.    System.out.println("running...");
4.   }
5.   public static void main(String args[]){
6.    TestThreadTwice1 t1=new TestThreadTwice1();
7.    t1.start();
8.    t1.start();
9.   }
10.  }  Output:
```

running

Exception in thread "main" java.lang.IllegalThreadStateException

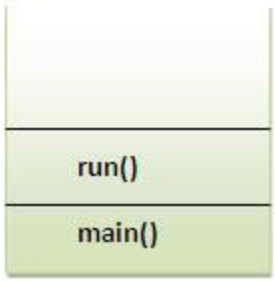
What if we call run() method directly instead start() method?

- Each thread starts in a separate call stack.
- Invoking the run() method from main thread, the run() method goes onto the current call stack rather than at the beginning of a new call stack.

```
1.  class TestCallRun1 extends Thread{
2.   public void run(){
3.    System.out.println("running...");
4.   }
5.   public static void main(String args[]){
6.    TestCallRun1 t1=new TestCallRun1();
7.    t1.run();//fine, but does not start a separate call stack
8.   }
9.  }
```

Output

Output:running...



Stack
(main thread)

Problem if you direct call run() method

```
1.  class TestCallRun2 extends Thread{
2.   public void run(){
3.    for(int i=1;i<5;i++){
4.      try{Thread.sleep(500);}catch(InterruptedException e){System.out.println(e);}
5.      System.out.println(i);
6.    }
7.   }
8.   public static void main(String args[]){
9.    TestCallRun2 t1=new TestCallRun2();
10.   TestCallRun2 t2=new TestCallRun2();
11.
12.   t1.run();
13.   t2.run();
14.  }
15. }
```

Output:1
2
3
4
5
1
2
3
4
5

As you can see in the above program that there is no context-switching because here t1 and t2 will be treated as normal object not thread object.

Priority of a Thread (Thread Priority):

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

3 constants defiend in Thread class:

1. public static int MIN_PRIORITY (1)
2. public static int NORM_PRIORITY(5)
3. public static int MAX_PRIORITY(10)

Default priority of a thread is 5 (NORM_PRIORITY). The value of MIN_PRIORITY is 1 and the value of MAX_PRIORITY is 10.

Example of priority of a Thread:

```
1.  class TestMultiPriority1 extends Thread{
2.   public void run(){
3.    System.out.println("running thread name is:"+Thread.currentThread().getName());
4.    System.out.println("running thread priority is:"+Thread.currentThread().getPriority());
5.
6.   }
7.   public static void main(String args[]){
8.    TestMultiPriority1 m1=new TestMultiPriority1();
9.    TestMultiPriority1 m2=new TestMultiPriority1();
10.   m1.setPriority(Thread.MIN_PRIORITY);
11.   m2.setPriority(Thread.MAX_PRIORITY);
12.   m1.start();
13.   m2.start();
14.
15.  }
16. }
```

```
Output:running thread name is:Thread-0
       running thread priority is:10
       running thread name is:Thread-1
```

```
running thread priority is:1
```

Java - Thread Synchronization

When we start two or more threads within a program, there may be a situation when multiple threads try to access the same resource and finally they can produce unforeseen result due to concurrency issue. For example if multiple threads try to write within a same file then they may corrupt the data because one of the threads can overrite data or while one thread is opening the same file at the same time another thread might be closing the same file.

So there is a need to synchronize the action of multiple threads and make sure that only one thread can access the resource at a given point in time. This is implemented using a concept called **monitors**. Each object in Java is associated with a monitor, which a thread can lock or unlock. Only one thread at a time may hold a lock on a monitor.

Java programming language provides a very handy way of creating threads and synchronizing their task by using **synchronized** blocks. You keep shared resources within this block. Following is the general form of the synchronized statement:

```
synchronized(objectidentifier) {
  // Access shared variables and other shared resources
}
```

Here, the **objectidentifier** is a reference to an object whose lock associates with the monitor that the synchronized statement represents. Now we are going to see two examples where we will print a counter using two different threads. When threads are not synchronized, they print counter value which is not in sequence, but when we print counter by putting inside synchronized() block, then it prints counter very much in sequence for both the threads.

Example:-

```
package com.brainmentors;

class sync extends Thread
{
```

```
        synchronized public void run()
        {
                System.out.println("hi");
}
        void show()
        {
                synchronized(this)
                {
                        System.out.println("synchronized block...");
                }}}

public class Synchronisation {
        public static void main(String[] args) {
                sync s1=new sync();
                sync s2=new sync();
                s1.show();
                s2.show();
                s1.start();
                s2.start();
}}
```

Copy Constructors in Java

WHAT ?

A copy constructor is a constructor that takes only one argument which is of the type as the class in which the copy constructor is implemented. For example, let us assume a class namely Car and it has a constructor called copy constructor which expects only one argument of type Car.

WHY ?

Copy constructors are widely used for creating a duplicates of objects known as cloned objects. Duplicate object in the sense the object will have the same characteristics of the original object from which duplicate object is created. But we have to ensure that both original and duplicate objects refer to different memory locations.

WHERE ?

It's our responsibility to implement a copy constructor in our class in the right way. As mentioned above, it's used to duplicate objects. So we are free to use copy constructors instead of **clone** method in java.

HOW ?

Copy constructors are implemented for **deep cloning** and **shallow cloning**. Both these cloning techniques can be achieved by overriding the clone method in java. Shallow cloning is the default cloning implementation in java and we just need to override the clone method. But deep cloning has to be implemented as per our need and logic. Explaining about cloning is beyond the scope of this post. So let's come back.

When you pass an instance of a particular class to copy constructor, it returns a new instance of the same class with all values copied from the parameter instance.

For example:-

```
Employee originalEmployee = new Employee(1, "Ram", department);
Employee clonedEmployee = new Employee(originalEmployee);
                                    OR

 Employee originalEmployee = new Employee(1, "Ram", department);
 Employee clonedEmployee = Employee.copy(originalEmployee);
```

Java Networking

Java Networking is a concept of connecting two or more computing devices together so that we can share resources.

Java socket programming provides facility to share data between different computing devices.

Advantage of Java Networking

1. sharing resources
2. centralize software management

Java Networking Terminology

The widely used java networking terminologies are given below:

1. IP Address
2. Protocol
3. Port Number
4. MAC Address
5. Connection-oriented and connection-less protocol
6. Socket

1) IP Address

IP address is a unique number assigned to a node of a network e.g. 192.168.0.1 . It is composed of octets that range from 0 to 255.

It is a logical address that can be changed.

2) Protocol

A protocol is a set of rules basically that is followed for communication. For example:

- TCP
- FTP
- Telnet
- SMTP
- POP etc.

3) Port Number

The port number is used to uniquely identify different applications. It acts as a communication endpoint between applications.

The port number is associated with the IP address for communication between two applications.

4) MAC Address

MAC (Media Access Control) Address is a unique identifier of NIC (Network Interface Controller). A network node can have multiple NIC but each with unique MAC.

5) Connection-oriented and connection-less protocol

In connection-oriented protocol, acknowledgement is sent by the receiver. So it is reliable but slow. The example of connection-oriented protocol is TCP.

But, in connection-less protocol, acknowledgement is not sent by the receiver. So it is not reliable but fast. The example of connection-less protocol is HTTP,UDP etc.

6) Socket

A socket is an endpoint between two way communication.

Java Socket Programming

Java Socket programming is used for communication between the applications running on different JRE.

Java Socket programming can be connection-oriented or connection-less.

Socket and ServerSocket classes are used for connection-oriented socket programming and DatagramSocket and DatagramPacket classes are used for connection-less socket programming.

The client in socket programming must know two information:

1. IP Address of Server, and
2. Port number.

Socket class

A socket is simply an endpoint for communications between the machines. The Socket class can be used to create a socket.

Important methods

Method	Description
1) public InputStream getInputStream()	returns the InputStream attached with this socket.
2) public OutputStream getOutputStream()	returns the OutputStream attached with this socket.
3) public synchronized void close()	closes this socket

ServerSocket class

The ServerSocket class can be used to create a server socket. This object is used to establish communication with the clients.

Important methods

Method	Description
1) public Socket accept()	returns the socket and establish a connection between server and client.
2) public synchronized void close()	closes the server socket.

Example of Java Socket Programming

Let's see a simple of java socket programming in which client sends a text and server receives it.

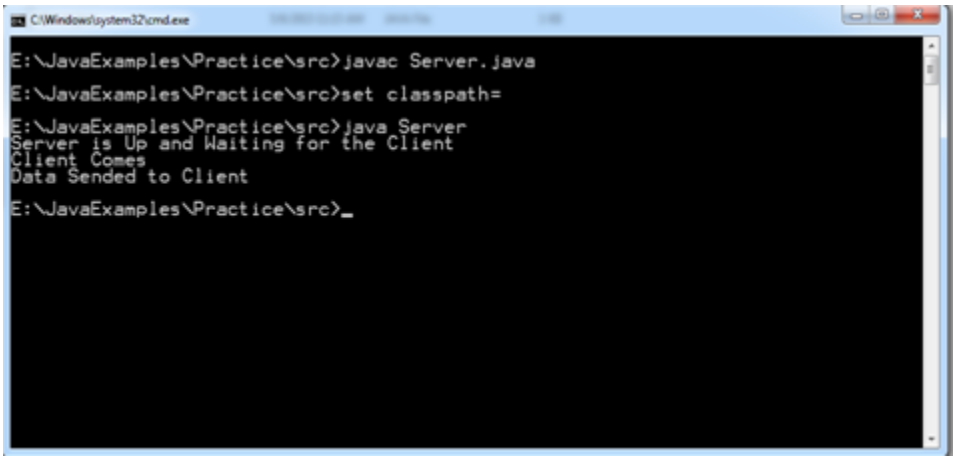
File: Server.java

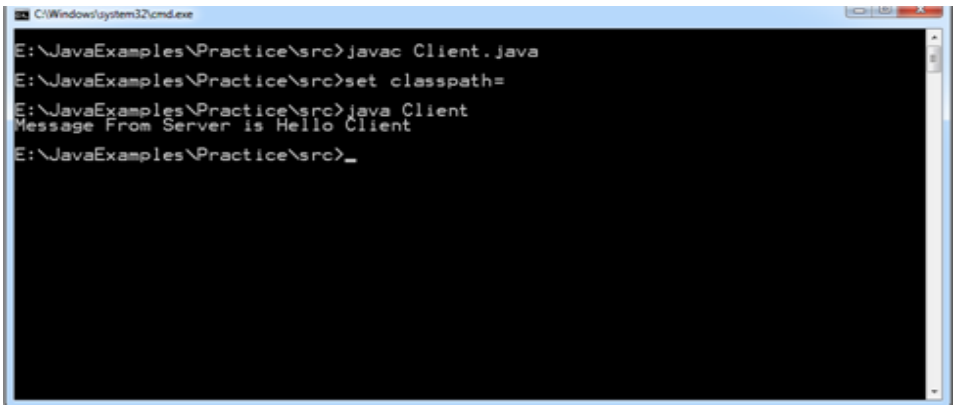
```
import java.io.IOException;
import java.io.OutputStream;
import java.io.PrintStream;
import java.net.ServerSocket;
import java.net.Socket;


public class Server
{
        public static void main(String[] args) throws IOException
{
 ServerSocket server = new ServerSocket(9001); // port value should be above 1024
        System.out.println("Server is Up and Waiting for the Client");
                Socket socket = server.accept();
                System.out.println("Client Comes");
                OutputStream os = socket.getOutputStream();
                PrintStream out = new PrintStream(os);
                out.println("Hello Client ");
                System.out.println("Data Sended to Client ");
                out.close();
                os.close();
                socket.close();
}}
```

File: MyClient.java

```
import java.io.IOException;
import java.io.InputStream;
import java.net.Socket;
```

```
import java.net.UnknownHostException;
import java.util.Scanner;

public class Client
{
	public static void main(String[] args) throws UnknownHostException, IOException {
		Socket socket = new Socket("localhost",9001);// port value should be above 1024
		InputStream is = socket.getInputStream();
		Scanner scanner = new Scanner(is);
		String message = scanner.nextLine();
		System.out.println("Message From Server is "+message);
		scanner.close();
		is.close();
		socket.close();
}}
```





Servlet

Servlet technology is used to create web application (resides at server side and generates dynamic web page).

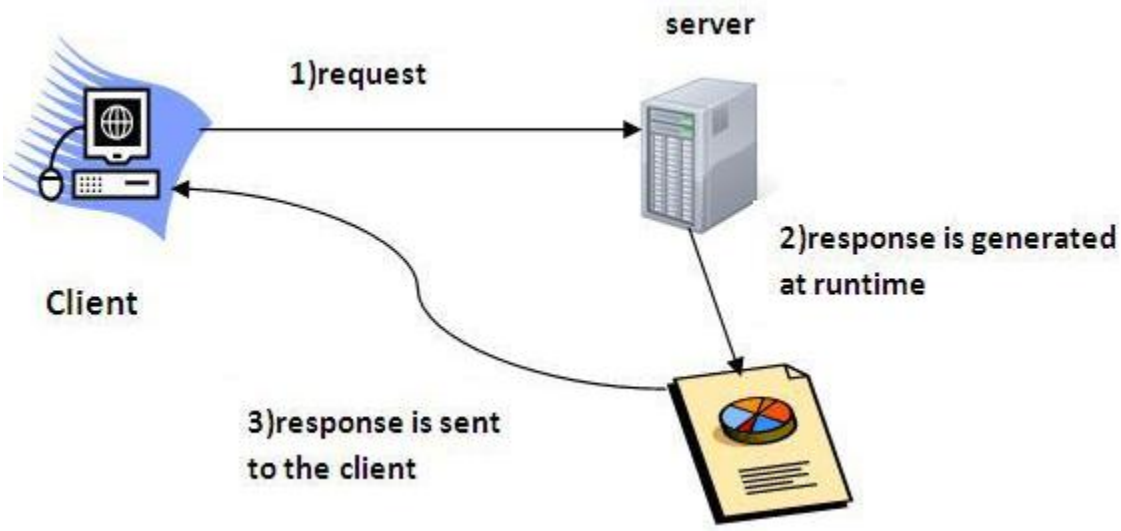
Servlet technology is robust and scalable because of java language. Before Servlet, CGI (Common Gateway Interface) scripting language was popular as a server-side programming language. But there was many disadvantages of this technology.

There are many interfaces and classes in the servlet API such as Servlet, GenericServlet, HttpServlet, ServletRequest, ServletResponse etc.

What is a Servlet?

Servlet can be described in many ways, depending on the context.

- Servlet is a technology i.e. used to create web application.
- Servlet is an API that provides many interfaces and classes including documentations.
- Servlet is an interface that must be implemented for creating any servlet.
- Servlet is a class that extend the capabilities of the servers and respond to the incoming request. It can respond to any type of requests.
- Servlet is a web component that is deployed on the server to create dynamic web page.

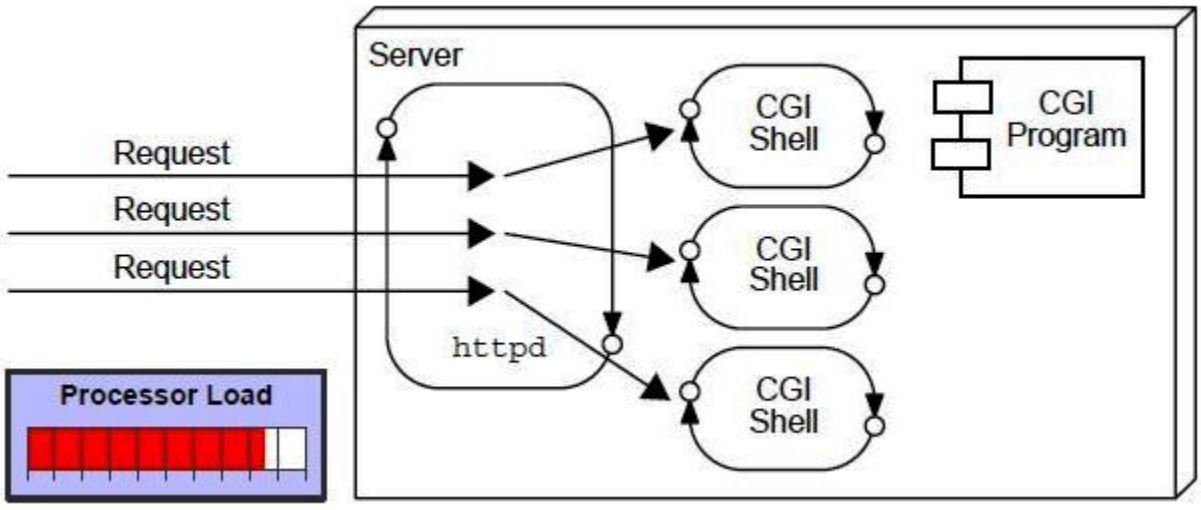


What is web application?

A web application is an application accessible from the web. A web application is composed of web components like Servlet, JSP, Filter etc. and other components such as HTML. The web components typically execute in Web Server and respond to HTTP request.

CGI(Commmon Gateway Interface)

CGI technology enables the web server to call an external program and pass HTTP request information to the external program to process the request. For each request, it starts a new process.

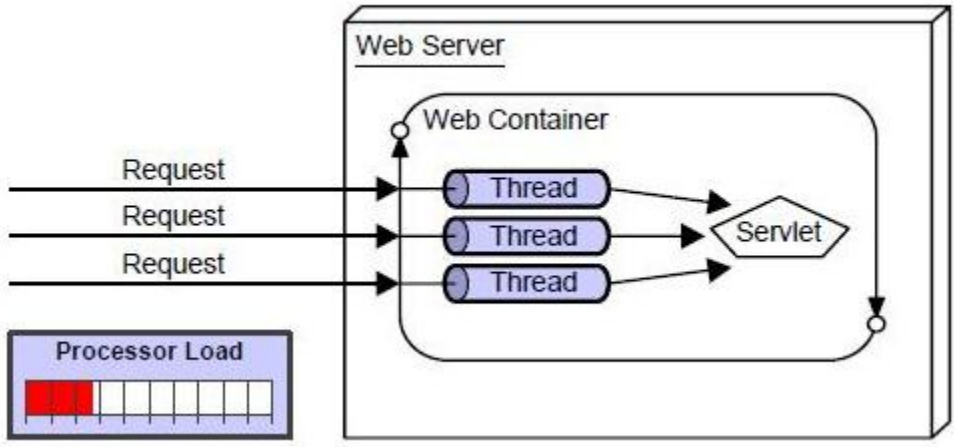


Disadvantages of CGI

There are many problems in CGI technology:

1. If number of clients increases, it takes more time for sending response.
2. For each request, it starts a process and Web server is limited to start processes.
3. It uses platform dependent language e.g. C, C++, perl.

Advantages of Servlet



There are many advantages of servlet over CGI. The web container creates threads for handling the multiple requests to the servlet. Threads have a lot of benefits over the Processes such as they share a common memory area, lightweight, cost of communication between the threads are low. The basic benefits of servlet are as follows:

1. **Better performance:** because it creates a thread for each request not process.
2. **Portability:** because it uses java language.
3. **Robust:** Servlets are managed by JVM so no need to worry about momory leak, garbage collection etc.
4. **Secure:** because it uses java language..

Servlet Terminology

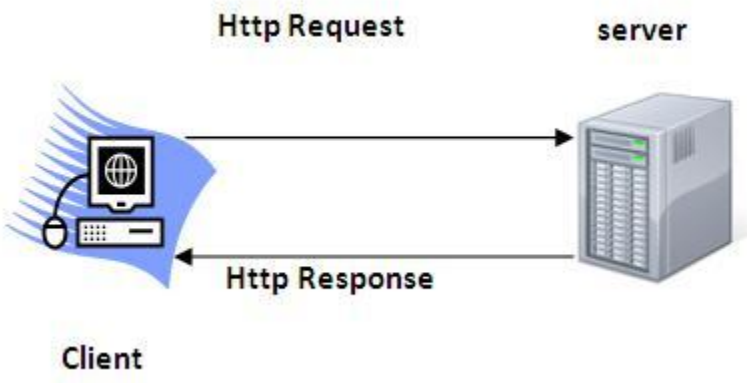
There are some key points that must be known by the servlet programmer like server, container, get request, post request etc. Let's first discuss these points before starting the servlet technology.

The basic **terminology used in servlet** are given below:

1. HTTP
2. HTTP Request Types
3. Difference between Get and Post method
4. Container
5. Server and Difference between web server and application server
6. Content Type
7. Introduction of XML
8. Deployment

HTTP (Hyper Text Transfer Protocol)

1. Http is the protocol that allows web servers and browsers to exchange data over the web.
2. It is a request response protocol.
3. Http uses reliable TCP connections bydefault on TCP port 80.
4. It is stateless means each request is considered as the new request. In other words, server doesn't recognize the user bydefault.



Http Request Methods

Every request has a header that tells the status of the client. There are many request methods. Get and Post requests are mostly used.

The http request methods are:

- GET
- POST
- HEAD
- PUT
- DELETE
- OPTIONS
- TRACE

HTTP Request	Description
GET	Asks to get the resource at the requested URL.
POST	Asks the server to accept the body info attached. It is like GET request with extra info sent with the request.
HEAD	Asks for only the header part of whatever a GET would return. Just like GET but with no body.
TRACE	Asks for the loopback of the request message, for testing or troubleshooting.
PUT	Says to put the enclosed info (the body) at the requested URL.
DELETE	Says to delete the resource at the requested URL.
OPTIONS	Asks for a list of the HTTP methods to which the thing at the request URL can respond

What is the difference between Get and Post?

There are many differences between the Get and Post request. Let's see these differences:

GET	POST
1) In case of Get request, only **limited amount of data** can be sent because data is sent in header.	In case of post request, **large amount of data** can be sent because data is sent in body.
2) Get request is **not secured** because data is exposed in URL bar.	Post request is **secured** because data is not exposed in URL bar.
3) Get request **can be bookmarked**	Post request **cannot be** bookmarked
4) Get request is **idempotent**. It means second request will be ignored until response of first request is delivered.	Post request is **non-idempotent**
5) Get request is **more efficient** and used more than Post	Post request is **less efficient** and used less than get.

Container

It provides runtime environment for JavaEE (j2ee) applications.

It performs many operations that are given below:

1. Life Cycle Management
2. Multithreaded support

3. Object Pooling
4. Security etc.

Server

It is a running program or software that provides services.

There are two types of servers:

1. Web Server
2. Application Server

Web Server

Web server contains only web or servlet container. It can be used for servlet, jsp, struts, jsf etc. It can't be used for EJB.

Example of Web Servers are: **Apache Tomcat** and **Resin**.

Application Server

Application server contains Web and EJB containers. It can be used for servlet, jsp, struts, jsf, ejb etc.

Example of Application Servers are:

1. **JBoss** Open-source server from JBoss community.
2. **Glassfish** provided by Sun Microsystem. Now acquired by Oracle.
3. **Weblogic** provided by Oracle. It more secured.
4. **Websphere** provided by IBM.

Content Type

Content Type is also known as MIME (Multipurpose internet Mail Extension) Type. It is a **HTTP header** that provides the description about what are you sending to the browser.

There are many content types:

- text/html

- text/plain
- application/msword
- application/vnd.ms-excel
- application/jar
- application/pdf
- application/octet-stream
- application/x-zip
- images/jpeg
- video/quicktime etc.

Servlet API

The javax.servlet and javax.servlet.http packages represent interfaces and classes for servlet api.

The **javax.servlet** package contains many interfaces and classes that are used by the servlet or web container. These are not specific to any protocol.

The **javax.servlet.http** package contains interfaces and classes that are responsible for http requests only.

Let's see what are the interfaces of javax.servlet package.

Interfaces in javax.servlet package

There are many interfaces in javax.servlet package. They are as follows:

1. Servlet
2. ServletRequest
3. ServletResponse
4. RequestDispatcher
5. ServletConfig
6. ServletContext
7. SingleThreadModel
8. Filter
9. FilterConfig
10. FilterChain
11. ServletRequestListener
12. ServletRequestAttributeListener
13. ServletContextListener
14. ServletContextAttributeListener

Classes in javax.servlet package

There are many classes in javax.servlet package. They are as follows:

1. GenericServlet
2. ServletInputStream
3. ServletOutputStream
4. ServletRequestWrapper
5. ServletResponseWrapper
6. ServletRequestEvent
7. ServletContextEvent
8. ServletRequestAttributeEvent
9. ServletContextAttributeEvent
10. ServletException
11. UnavailableException

Interfaces in javax.servlet.http package

There are many interfaces in javax.servlet.http package. They are as follows:

1. HttpServletRequest
2. HttpServletResponse
3. HttpSession
4. HttpSessionListener
5. HttpSessionAttributeListener
6. HttpSessionBindingListener
7. HttpSessionActivationListener
8. HttpSessionContext (deprecated now)

Classes in javax.servlet.http package

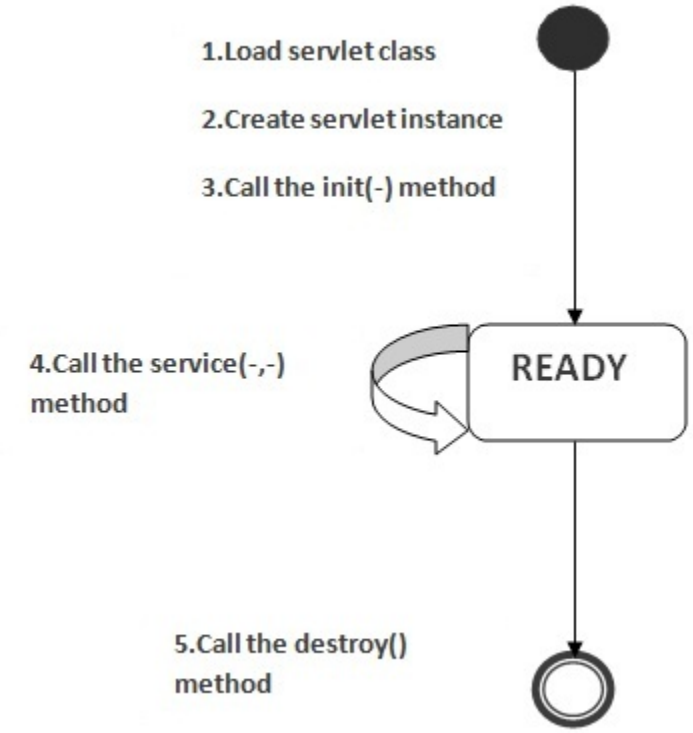
There are many classes in javax.servlet.http package. They are as follows:

1. HttpServlet
2. Cookie
3. HttpServletRequestWrapper
4. HttpServletResponseWrapper
5. HttpSessionEvent
6. HttpSessionBindingEvent
7. HttpUtils (deprecated now)

Life Cycle of a Servlet (Servlet Life Cycle)

The web container maintains the life cycle of a servlet instance. Let's see the life cycle of the servlet:

1. Servlet class is loaded.
2. Servlet instance is created.
3. init method is invoked.
4. service method is invoked.
5. destroy method is invoked.



As displayed in the above diagram, there are three states of a servlet: new, ready and end. The servlet is in new state if servlet instance is created. After invoking the init() method, Servlet comes in the ready state. In the ready state, servlet performs all the tasks. When the web container invokes the destroy() method, it shifts to the end state.

1) Servlet class is loaded

The classloader is responsible to load the servlet class. The servlet class is loaded when the first request for the servlet is received by the web container.

2) Servlet instance is created

The web container creates the instance of a servlet after loading the servlet class. The servlet instance is created only once in the servlet life cycle.

3) init method is invoked

The web container calls the init method only once after creating the servlet instance. The init method is used to initialize the servlet. It is the life cycle method of the javax.servlet.Servlet interface. Syntax of the init method is given below:

```
1.  public void init(ServletConfig config) throws ServletException
```

4) service method is invoked

The web container calls the service method each time when request for the servlet is received. If servlet is not initialized, it follows the first three steps as described above then calls the service method. If servlet is initialized, it calls the service method. Notice that servlet is initialized only once. The syntax of the service method of the Servlet interface is given below:

```
1.  public void service(ServletRequest request, ServletResponse response)
2.    throws ServletException, IOException
```

5) destroy method is invoked

The web container calls the destroy method before removing the servlet instance from the service. It gives the servlet an opportunity to clean up any resource for example memory, thread etc. The syntax of the destroy method of the Servlet interface is given below:

```
1.  public void destroy()
```

Servlet Interface

Servlet interface provides common behaviour to all the servlets.

Servlet interface needs to be implemented for creating any servlet (either directly or indirectly). It provides 3 life cycle methods that are used to initialize the servlet, to service the requests, and to destroy the servlet and 2 non-life cycle methods.

Methods of Servlet interface

There are 5 methods in Servlet interface. The init, service and destroy are the life cycle methods of servlet. These are invoked by the web container.

Method	Description
public void init(ServletConfig config)	initializes the servlet. It is the life cycle method of servlet and invoked by the web container only once.
public void service(ServletRequest request,ServletResponse response)	provides response for the incoming request. It is invoked at each request by the web container.
public void destroy()	is invoked only once and indicates that servlet is being destroyed.
public ServletConfig getServletConfig()	returns the object of ServletConfig.
public String getServletInfo()	returns information about servlet such as writer, copyright, version etc.

Servlet Example by implementing Servlet interface

Let's see the simple example of servlet by implementing the servlet interface.

File: First.java

```
1.  import java.io.*;
2.  import javax.servlet.*;
3.
4.  public class First implements Servlet{
5.  ServletConfig config=null;
6.
7.  public void init(ServletConfig config){
8.  this.config=config;
9.  System.out.println("servlet is initialized");
10. }
11.
12. public void service(ServletRequest req,ServletResponse res)
13. throws IOException,ServletException{
14.
```

```
15.         res.setContentType("text/html");
16.
17.         PrintWriter out=res.getWriter();
18.         out.print("<html><body>");
19.         out.print("<b>hello simple servlet</b>");
20.         out.print("</body></html>");
21.
22.         }
23.         public void destroy(){System.out.println("servlet is destroyed");}
24.         }
```

Java Inner Class

Java inner class or nested class is a class i.e. declared inside the class or interface.

We use inner classes to logically group classes and interfaces in one place so that it can be more readable and maintainable.

Additionally, it can access all the members of outer class including private data members and methods.

Syntax of Inner class

```
1.          class Java_Outer_class{
2.           //code
3.           class Java_Inner_class{
4.            //code
5.           }
6.          }
```

Advantage of Java Inner Classes

There are basically three advantages of inner classes in java. They are as follows:

1) Nested classes represent a special type of relationship that is **it can access all the members (data members and methods) of outer class** including private.

2) Nested classes are used **to develop more readable and maintainable code** because it logically group classes and interfaces in one place only.

3) **Code Optimization**: It requires less code to write.

Difference Between Nested Class And Inner Class In Java

Inner class is a part of nested class. Non-static nested classes are known as inner classes.

Types of Nested classes

There are two types of nested classes non-static and static nested classes.The non-static nested classes are also known as inner classes.

1. Non-static nested class(inner class)
    - a)Member inner class
    - b)Annomynous inner class
    - c)Local inner class

2. Static Nested Class

Type	Description
Member Inner Class	A class created within class and outside method.
Anonymous Inner Class	A class created for implementing interface or extending class. Its name is decided by the java compiler.
Local Inner Class	A class created within method.
Static Nested Class	A static class created within class.
Nested Interface	An interface created within class or interface.

Java Member Inner Class

A non-static class that is created inside a class but outside a method is called member inner class.

Syntax:

```
1.  class Outer{
2.   //code
3.   class Inner{
4.    //code
5.   }
6.  }
```

Java Member Inner Class Example

In this example, we are creating msg() method in member inner class that is accessing the private data member of outer class.

```
1.  class TestMemberOuter1{
2.   private int data=30;
3.   class Inner{
4.    void msg(){System.out.println("data is "+data);}
5.   }
6.   public static void main(String args[]){
7.    TestMemberOuter1 obj=new TestMemberOuter1();
8.    TestMemberOuter1.Inner in=obj.new Inner();
9.    in.msg();
10.  }
11. }
```

Internal working of Java member inner class

The java compiler creates two class files in case of inner class. The class file name of inner class is "Outer$Inner". If you want to instantiate inner class, you must have to create the instance of outer class. In such case, instance of inner class is created inside the instance of outer class.

Internal code generated by the compiler

The java compiler creates a class file named Outer$Inner in this case. The Member inner class have the reference of Outer class that is why it can access all the data members of Outer class including private.

```
1.  import java.io.PrintStream;
2.  class Outer$Inner
3.  {
4.     final Outer this$0;
5.     Outer$Inner()
6.     {   super();
7.        this$0 = Outer.this;
8.     }
9.     void msg()
10.    {
11.       System.out.println((new StringBuilder()).append("data is ")
12.                .append(Outer.access$000(Outer.this)).toString());
13.    }
```

```
14.     }
```

Java Anonymous Inner Class

A class that have no name is known as anonymous inner class in java. It should be used if you have to override method of class or interface. Java Anonymous inner class can be created by two ways:

1. Class (may be abstract or concrete).
2. Interface

Java anonymous inner class example using class

```
1.      abstract class Person{
2.        abstract void eat();
3.      }
4.      class TestAnonymousInner{
5.       public static void main(String args[]){
6.        Person p=new Person(){
7.        void eat(){System.out.println("nice fruits");}
8.        };
9.        p.eat();
10.     }
11.     }
```

Output:

```
nice fruits
```

Internal working of given code

```
1.      Person p=new Person(){
2.      void eat(){System.out.println("nice fruits");}
3.      };
```

1. A class is created but its name is decided by the compiler which extends the Person class and provides the implementation of the eat() method.
2. An object of Anonymous class is created that is referred by p reference variable of Person type.

Internal Class Generated By The Compiler

```
1.      import java.io.PrintStream;
2.      static class TestAnonymousInner$1 extends Person
3.      {
```

```
4.      TestAnonymousInner$1(){}
5.      void eat()
6.       {
7.          System.out.println("nice fruits");
8.       }
9.    }
```

Java Anonymous Inner Class Example Using Interface

```
1.    interface Eatable{
2.     void eat();
3.    }
4.    class TestAnnonymousInner1{
5.     public static void main(String args[]){
6.     Eatable e=new Eatable(){
7.      public void eat(){System.out.println("nice fruits");}
8.     };
9.     e.eat();
10.    }
11.   }
```

Output:

```
nice fruits
```

Internal working of given code

It performs two main tasks behind this code:

```
1.      Eatable p=new Eatable(){
2.      void eat(){System.out.println("nice fruits");}
3.      };
```

1. A class is created but its name is decided by the compiler which implements the Eatable interface and provides the implementation of the eat() method.
2. An object of Anonymous class is created that is referred by p reference variable of Eatable type.

Internal class generated by the compiler

```
1.      import java.io.PrintStream;
2.      static class TestAnonymousInner1$1 implements Eatable
3.      {
4.      TestAnonymousInner1$1(){}
```

```
5.      void eat(){System.out.println("nice fruits");}
6.      }
```

Java Local Inner Class

A class i.e. created inside a method is called local inner class in java. If you want to invoke the methods of local inner class, you must instantiate this class inside the method.

Java local inner class example

```
1.      public class localInner1{
2.       private int data=30;//instance variable
3.       void display(){
4.        class Local{
5.         void msg(){System.out.println(data);}
6.        }
7.        Local l=new Local();
8.        l.msg();
9.       }
10.      public static void main(String args[]){
11.       localInner1 obj=new localInner1();
12.       obj.display();
13.      }
14.     }
```

Output:

```
30
```

Internal class generated by the compiler

In such case, compiler creates a class named Simple$1Local that have the reference of the outer class.

```
1.      import java.io.PrintStream;
2.      class localInner1$Local
3.      {
4.         final localInner1 this$0;
5.         localInner1$Local()
6.         {
7.            super();
8.            this$0 = Simple.this;
9.         }
10.        void msg()
```

```
11.         {
12.             System.out.println(localInner1.access$000(localInner1.this));
13.         }
14.     }
```

Rule: Local variable can't be private, public or protected.

Rules for Java Local Inner class

1) Local inner class cannot be invoked from outside the method.

2) Local inner class cannot access non-final local variable till JDK 1.7. Since JDK 1.8, it is possible to access the non-final local variable in local inner class.

Example of local inner class with local variable

```
1.      class localInner2{
2.       private int data=30;//instance variable
3.       void display(){
4.        int value=50;//local variable must be final till jdk 1.7 only
5.        class Local{
6.         void msg(){System.out.println(value);}
7.        }
8.        Local l=new Local();
9.        l.msg();
10.      }
11.      public static void main(String args[]){
12.       localInner2 obj=new localInner2();
13.       obj.display();
14.      }
15.     }
```

Output:

```
50
```

Java static nested class

A static class i.e. created inside a class is called static nested class in java. It cannot access non-static data members and methods. It can be accessed by outer class name.

- It can access static data members of outer class including private.
- Static nested class cannot access non-static (instance) data member or method.

Java static nested class example with instance method

```
1.  class TestOuter1{
2.    static int data=30;
3.    static class Inner{
4.     void msg(){System.out.println("data is "+data);}
5.    }
6.    public static void main(String args[]){
7.    TestOuter1.Inner obj=new TestOuter1.Inner();
8.    obj.msg();
9.    }
10. }
```

Output:

```
data is 30
```

In this example, you need to create the instance of static nested class because it has instance method msg(). But you don't need to create the object of Outer class because nested class is static and static properties, methods or classes can be accessed without object.

Internal class generated by the compiler

```
1.  import java.io.PrintStream;
2.  static class TestOuter1$Inner
3.  {
4.  TestOuter1$Inner(){}
5.  void msg(){
6.  System.out.println((new StringBuilder()).append("data is ")
7.  .append(TestOuter1.data).toString());
8.  }
9.  }
```

Java static nested class example with static method

If you have the static member inside static nested class, you don't need to create instance of static nested class.

```
1.  class TestOuter2{
2.    static int data=30;
3.    static class Inner{
```

```
4.          static void msg(){System.out.println("data is "+data);}
5.         }
6.         public static void main(String args[]){
7.         TestOuter2.Inner.msg();//no need to create the instance of static nested class
8.         }
9.        }
```

Output:

```
data is 30
```

Java Nested Interface

An interface i.e. declared within another interface or class is known as nested interface. The nested interfaces are used to group related interfaces so that they can be easy to maintain. The nested interface must be referred by the outer interface or class. It can't be accessed directly.

Points to remember for nested interfaces

There are given some points that should be remembered by the java programmer.

- Nested interface must be public if it is declared inside the interface but it can have any access modifier if declared within the class.
- Nested interfaces are declared static implicitely.

Syntax of nested interface which is declared within the interface

```
1.      interface interface_name{
2.       ...
3.       interface nested_interface_name{
4.        ...
5.       }
6.      }
```

Syntax of nested interface which is declared within the class

```
1.      class class_name{
2.       ...
3.       interface nested_interface_name{
4.        ...
5.       }
6.      }
```

Example of nested interface which is declared within the interface

In this example, we are going to learn how to declare the nested interface and how we can

access it.

```
1.  interface Showable{
2.    void show();
3.    interface Message{
4.     void msg();
5.    }
6.  }
7.
8.  class TestNestedInterface1 implements Showable.Message{
9.   public void msg(){System.out.println("Hello nested interface");}
10.
11.  public static void main(String args[]){
12.   Showable.Message message=new TestNestedInterface1();//upcasting here
13.   message.msg();
14.  }
15. }
```

Output:hello nested interface

As you can see in the above example, we are acessing the Message interface by its outer interface Showable because it cannot be accessed directly. It is just like almirah inside the room, we cannot access the almirah directly because we must enter the room first. In collection frameword, sun microsystem has provided a nested interface Entry. Entry is the subinterface of Map i.e. accessed by Map.Entry.

Internal code generated by the java compiler for nested interface Message

The java compiler internally creates public and static interface as displayed below:.

```
1.  public static interface Showable$Message
2.  {
3.    public abstract void msg();
4.  }
```

Example of nested interface which is declared within the class

Let's see how can we define an interface inside the class and how can we access it.

```
1.  class A{
2.    interface Message{
3.     void msg();
4.    }
5.  }
6.
7.  class TestNestedInterface2 implements A.Message{
8.   public void msg(){System.out.println("Hello nested interface");}
```

```
9.
10.         public static void main(String args[]){
11.          A.Message message=new TestNestedInterface2();//upcasting here
12.          message.msg();
13.         }
14.         }
```

Output:hello nested interface

Can we define a class inside the interface?

Yes, If we define a class inside the interface, java compiler creates a static nested class.

```
1.          interface M{
2.           class A{}
3.
```

Wrapper classes

Java is an object-oriented language and can view everything as an object. A simple file can be treated as an object (with **java.io.File**), an address of a system can be seen as an object (with **java.util.URL**), an image can be treated as an object (with **java.awt.Image**) and a simple data type can be converted into an object (with **wrapper classes**). This tutorial discusses wrapper classes.

Wrapper classes are used to convert any data type into an object.

The primitive data types are not objects; they do not belong to any class; they are defined in the language itself. Sometimes, it is required to convert data types into objects in Java language. For example, upto JDK1.4, the data structures accept only objects to store. A data type is to be converted into an object and then added to a Stack or Vector etc. For this conversion, the designers introduced **wrapper classes**.

What are Wrapper classes?

As the name says, a wrapper class wraps (encloses) around a data type and gives it an object appearance. Wherever, the data type is required as an object, this object can be used. Wrapper classes include methods to unwrap the object and give back the data type. It can be compared with a chocolate. The manufacturer wraps the chocolate with some foil or paper to prevent from pollution. The user takes the chocolate, removes and throws the wrapper and eats it. For ex:-

int k = 100;
Integer it1 = new Integer(k);

The **int** data type **k** is converted into an object, **it1** using **Integer** class. The **it1** object can be used in Java programming wherever **k** is required an object.
The following code can be used to unwrap (getting back **int** from **Integer** object) the object **it1**.
int m = it1.intValue();
System.out.println(m*m); // prints 10000

intValue() is a method of **Integer** class that returns an **int** data type.

List of Wrapper classes

In the above code, **Integer** class is known as a wrapper class (because it wraps around int data type to give it an impression of object). To wrap (or to convert) each primitive data type, there comes a wrapper class. Eight wrapper classes exist in **java.lang** package that represent 8 data types. Following list gives.

Primitive data type	Wrapper class
byte	Byte
short	Short
int	Integer
long	Long
float	Float
double	Double
char	Character
boolean	Boolean

Following is the hierarchy of the above classes.

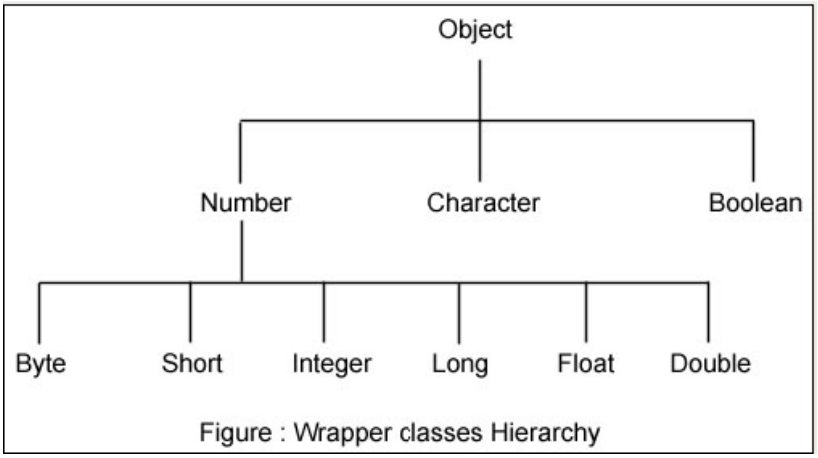


Figure : Wrapper classes Hierarchy

All the 8 wrapper classes are placed in **java.lang** package so that they are implicitly imported and made available to the programmer. As you can observe in the above hierarchy, the super class of all numeric wrapper classes is **Number** and the super class for **Character** and **Boolean** is Object. All the wrapper classes are defined as **final** and thus designers prevented them from inheritance.

Importance of Wrapper classes

There are mainly two uses with wrapper classes.

1. To convert simple data types into objects, that is, to give object form to a data type; here constructors are used.
2. To convert strings into data types (known as parsing operations), here methods of type parseXXX() are used.

The following program expresses the style of converting data type into an object and at the same time retrieving the data type from the object.

```
1  public class WrappingUnwrapping
2  {
3   public static void main(String args[])
4   {                                                    //  data types
5     byte grade = 2;
6     int marks = 50;
7     float price = 8.6f;          // observe a suffix of <strong>f</strong> for float
8     double rate = 50.5;
9                                          // data types to objects
10    Byte g1 = new Byte(grade);              // wrapping
```

```
11    Integer m1 = new Integer(marks);
12    Float f1 = new Float(price);
13    Double r1 = new Double(rate);
14                                  // let us print the values from objects
15    System.out.println("Values of Wrapper objects (printing as objects)");
16    System.out.println("Byte object g1:  " + g1);
17    System.out.println("Integer object m1:  " + m1);
18    System.out.println("Float object f1:  " + f1);
19    System.out.println("Double object r1:  " + r1);
20                    // objects to data types (retrieving data types from objects)
21    byte bv = g1.byteValue();               // unwrapping
22    int iv = m1.intValue();
23    float fv = f1.floatValue();
24    double dv = r1.doubleValue();
25                                  // let us print the values from data types
26    System.out.println("Unwrapped values (printing as data types)");
27    System.out.println("byte value, bv: " + bv);
28    System.out.println("int value, iv: " + iv);
29    System.out.println("float value, fv: " + fv);
30    System.out.println("double value, dv: " + dv);
31  }
32 }
```

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

Method	Purpose
parseInt(s)	returns a signed decimal integer value equivalent to string s
toString(i)	returns a new String object representing the integer i
byteValue()	returns the value of this Integer as a byte
doubleValue()	returns the value of this Integer as an double
floatValue()	returns the value of this Integer as a float
intValue()	returns the value of this Integer as an int
shortValue()	returns the value of this Integer as a short
longValue()	returns the value of this Integer as a long
int compareTo(int i)	Compares the numerical value of the invoking object with that of i. Returns 0 if the values are equal. Returns a negative value if the invoking object has a lower value. Returns a positive value if the invoking object has a greater value.
static int compare(int num1, int num2)	Compares the values of num1 and num2. Returns 0 if the values are equal. Returns a negative value if num1 is less than num2. Returns a positive value if num1 is greater than num2.
boolean equals(Object intObj)	Returns true if the invoking Integer object is equivalent to intObj. Otherwise, it returns false.

Java Package

A **java package** is a group of similar types of classes, interfaces and sub-packages.

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

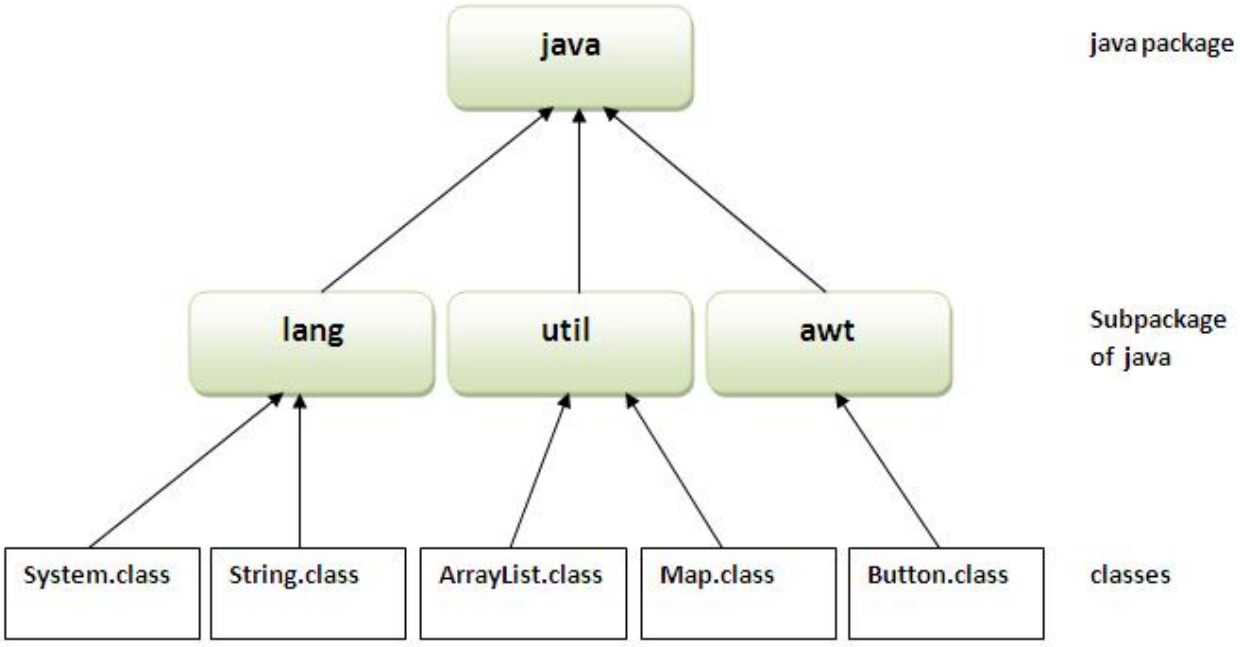
There are many built-in packages such as java, lang, awt, javax, swing, net, io, util, sql etc.

Advantage of Java Package

1) Java package is used to categorize the classes and interfaces so that they can be easily maintained.

2) Java package provides access protection.

3) Java package removes naming collision.

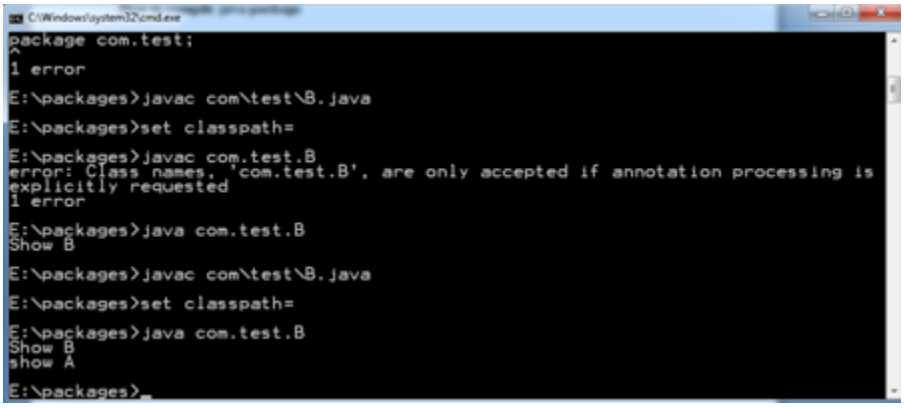


Simple example of java package

The **package keyword** is used to create a package in java.

```
1.  //save as Simple.java
2.  package mypack;
3.  public class Simple{
4.   public static void main(String args[]){
5.     System.out.println("Welcome to package");
6.    }
7.  }
```

How to compile java package



If you are not using any IDE, you need to follow the **syntax** given below:

Old Method to compile and run the package:-

1. javac -d directory javafilename

For **example**

1. javac -d . Simple.java

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

How to run java package program

You need to use fully qualified name e.g. mypack.Simple etc to run the class.

To Compile: javac -d . Simple.java

To Run: java mypack.Simple

Output:Welcome to package

The -d is a switch that tells the compiler where to put the class file i.e. it represents destination. The . represents the current folder.

How to access package from another package?

There are three ways to access the package from outside the package.

1. import package.*;
2. import package.classname;
3. fully qualified name.

1) Using packagename.*

If you use package.* then all the classes and interfaces of this package will be accessible but not subpackages.

The import keyword is used to make the classes and interface of another package accessible to

the current package.

Example of package that import the packagename.*

```
1.          //save by A.java
2.
3.          package pack;
4.          public class A{
5.            public void msg(){System.out.println("Hello");}
6.          }
1.          //save by B.java
2.
3.          package mypack;
4.          import pack.*;
5.
6.          class B{
7.            public static void main(String args[]){
8.             A obj = new A();
9.             obj.msg();
10.           }
11.         }
```

Output:Hello

2) Using packagename.classname

If you import package.classname then only declared class of this package will be accessible.

Example of package by import package.classname

```
1.          //save by A.java
2.
3.          package pack;
4.          public class A{
5.            public void msg(){System.out.println("Hello");}
6.          }
1.          //save by B.java
2.
3.          package mypack;
4.          import pack.A;
5.
6.          class B{
7.            public static void main(String args[]){
8.             A obj = new A();
```

```
9.              obj.msg();
10.            }
11.          }
```

Output:Hello

3) Using fully qualified name

If you use fully qualified name then only declared class of this package will be accessible. Now there is no need to import. But you need to use fully qualified name every time when you are accessing the class or interface.

It is generally used when two packages have same class name e.g. java.util and java.sql packages contain Date class.

Example of package by import fully qualified name

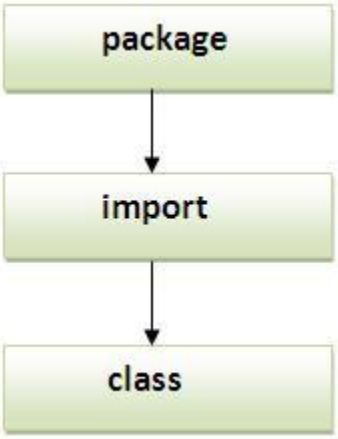
```
1.          //save by A.java
2.
3.          package pack;
4.          public class A{
5.            public void msg(){System.out.println("Hello");}
6.          }
1.          //save by B.java
2.
3.          package mypack;
4.          class B{
5.            public static void main(String args[]){
6.             pack.A obj = new pack.A();//using fully qualified name
7.             obj.msg();
8.            }
9.          }
```

Output:Hello

Note: If you import a package, subpackages will not be imported.

If you import a package, all the classes and interface of that package will be imported excluding the classes and interfaces of the subpackages. Hence, you need to import the subpackage as well.

Note: Sequence of the program must be package then import then class.



Subpackage in java

Package inside the package is called the **subpackage**. It should be created **to categorize the package further**.

Let's take an example, Sun Microsystem has definded a package named java that contains many classes like System, String, Reader, Writer, Socket etc. These classes represent a particular group e.g. Reader and Writer classes are for Input/Output operation, Socket and ServerSocket classes are for networking etc and so on. So, Sun has subcategorized the java package into subpackages such as lang, net, io etc. and put the Input/Output related classes in io package, Server and ServerSocket classes in net packages and so on.

The standard of defining package is domain.company.package e.g. com.javaxyz.bean or org.sssit.dao.

Example of Subpackage

```
1.  package com.javaxyz.core;
2.  class Simple{
3.    public static void main(String args[]){
4.     System.out.println("Hello subpackage");
5.    }
6.  }
```

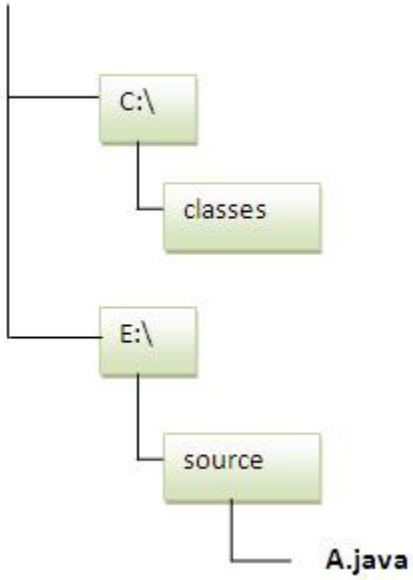
To Compile: javac -d . Simple.java

To Run: java com.javaxyz.core.Simple

Output:Hello subpackage

How to send the class file to another directory or drive?

There is a scenario, I want to put the class file of A.java source file in classes folder of c: drive. For example:




```
1.  //save as Simple.java
2.
3.  package mypack;
4.  public class Simple{
5.   public static void main(String args[]){
6.     System.out.println("Welcome to package");
7.    }
8.  }
```

To Compile:

e:\sources> javac -d c:\classes Simple.java

To Run:

To run this program from e:\source directory, you need to set classpath of the directory where the class file resides.

e:\sources> set classpath=c:\classes;.;

e:\sources> java mypack.Simple

Another way to run this program by -classpath switch of java:

The -classpath switch can be used with javac and java tool.

To run this program from e:\source directory, you can use -classpath switch of java that tells where to look for class file. For example:

e:\sources> java -classpath c:\classes mypack.Simple

Output:Welcome to package

Ways to load the class files or jar files

There are two ways to load the class files temporary and permanent.

- Temporary
  - By setting the classpath in the command prompt
  - By -classpath switch
- Permanent
  - By setting the classpath in the environment variables
  - By creating the jar file, that contains all the class files, and copying the jar file in the jre/lib/ext folder.

Rule: There can be only one public class in a java source file and it must be saved by the public class name.

```
1.  //save as C.java otherwise Compilte Time Error
2.
3.  class A{}
4.  class B{}
5.  public class C{}
```

How to put two public classes in a package?

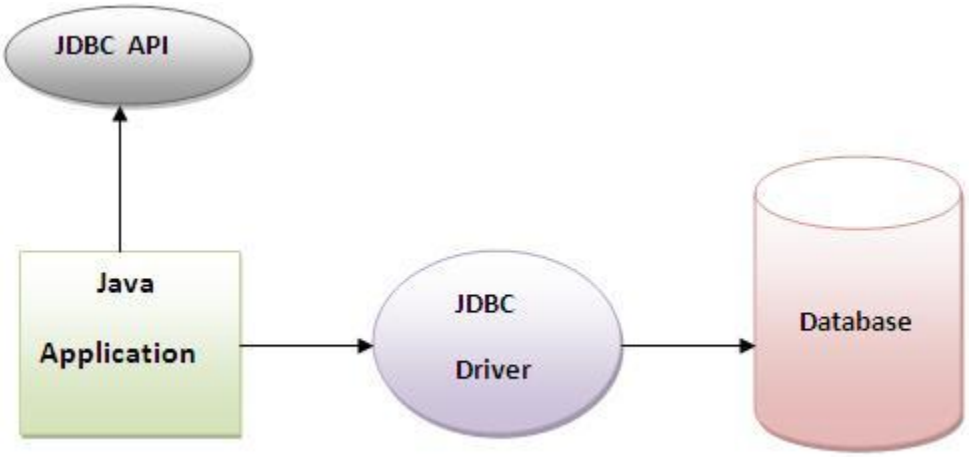
If you want to put two public classes in a package, have two java source files containing one public class, but keep the package name same. For example:

```
1.  //save as A.java
2.
3.  package javaxyz;
```

```
4.      public class A{}
1.      //save as B.java
2.
3.      package javapqr;
4.      public class B{}
```

Java JDBC(Java Database Connectivity)

Java JDBC is a java API to connect and execute query with the database. JDBC API uses jdbc drivers to connect with the database.



Why use JDBC

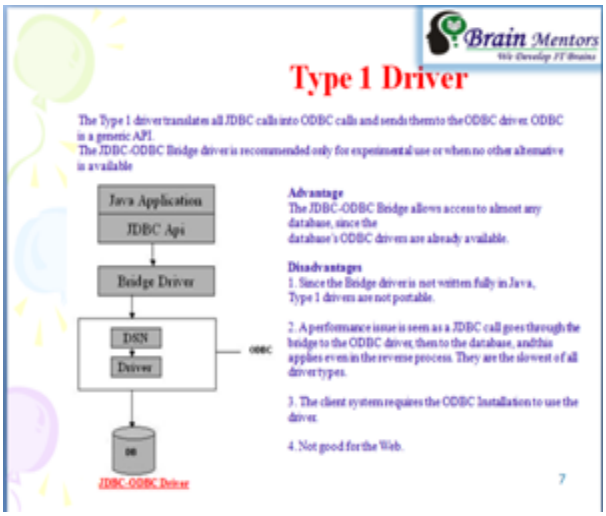
Before JDBC, ODBC API was the database API to connect and execute query with the database. But, ODBC API uses ODBC driver which is written in C language (i.e. platform dependent and unsecured). That is why Java has defined its own API (JDBC API) that uses JDBC drivers (written in Java language).

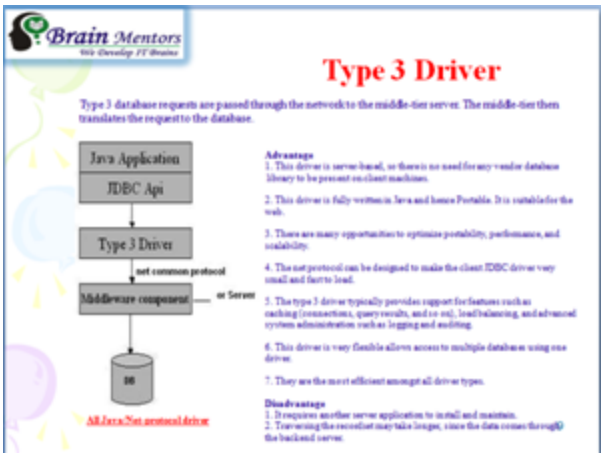
What is API

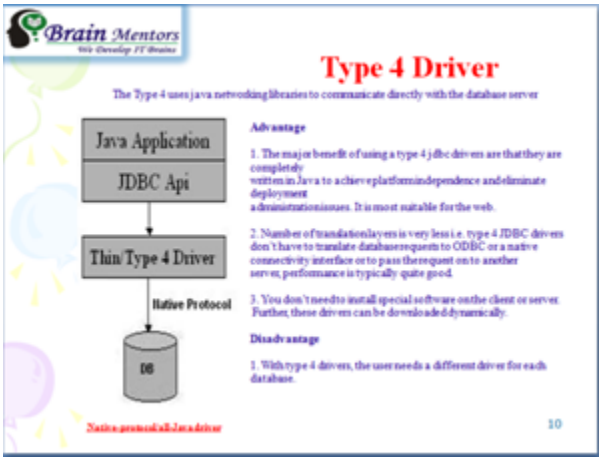
API (Application programming interface) is a document that contains description of all the features of a product or software. It represents classes and interfaces that software programs can follow to communicate with each other. An API can be created for applications, libraries, operating systems, etc

JDBC Driver

Kindly refer our ppt of JDBC for Type 1,2,3,4.







5 Steps to connect to the database in java

There are 5 steps to connect any java application with the database in java using JDBC. They are as follows:

1. Load the driver class
2. Creating connection
3. Creating statement
4. Executing queries
5. Closing connection

1) Load the driver class

The forName() method of Class class is used to register the driver class. This method is used to dynamically load the driver class.

Syntax of forName() method

```
Class.forName("com.mysql.jdbc.Driver");
```

2) Create the connection object

The getConnection() method of DriverManager class is used to establish connection with the database.

Syntax of getConnection() method

```
con=DriverManager.getConnection("jdbc:mysql://localhost:3306/demo","root","root");
```

3) Create the Statement object

The createStatement() method of Connection interface is used to create statement. The object of statement is responsible to execute queries with the database.

Syntax of createStatement() method

```
Statement stmt=con.createStatement();
```

4) Execute the query

The executeQuery() method of Statement interface is used to execute queries to the database. This method returns the object of ResultSet that can be used to get all the records of a table.

Syntax of executeQuery() method

```
ResultSet rs=stmt.executeQuery("select * from emp");
```

5) Close the connection object

By closing connection object statement and ResultSet will be closed automatically. The close() method of Connection interface is used to close the connection.

Syntax of close() method

```
con.close();
```

Example :-

```
import java.sql.*;
```

```
public class Select {

    public static void main(String[] args) throws ClassNotFoundException {
            try {

                    // 1. Load the Driver
                    Class.forName("com.mysql.jdbc.Driver");
                    // 2. Get Connection
                    Connection
con=DriverManager.getConnection("jdbc:mysql://localhost:3306/demo","root","root");
                // 3. Create a Statement
                    Statement stmt=con.createStatement();
                // 4. Execute SQL query
                    ResultSet rs=stmt.executeQuery("select * from emp");
                    // 5. Process Result Set
                while(rs.next())
                {
System.out.println(rs.getInt("id")+" "+rs.getString("name")+" "+rs.getDouble("sal"));
                }
            rs.close();
            stmt.close();
            con.close();
            } catch (SQLException e) {

                    System.out.println(e);
            }
             }}
```

ResultSet interface (Scrollable & Updateable)

The object of ResultSet maintains a cursor pointing to a particular row of data. Initially, cursor points to before the first row.

By default, ResultSet object can be moved forward only and it is not updatable.

But we can make this object to move forward and backward direction by passing either TYPE_SCROLL_INSENSITIVE or TYPE_SCROLL_SENSITIVE in createStatement(int,int) method as well as we can make this object as updatable by:

1. Statement stmt = con.createStatement(ResultSet.TYPE_SCROLL_INSENSITIVE, ResultSet.CONCUR_UPDATABLE);

Commonly used methods of ResultSet interface

1) public boolean next():	is used to move the cursor to the one row next from the current position.
2) public boolean previous():	is used to move the cursor to the one row previous from the current position.
3) public boolean first():	is used to move the cursor to the first row in result set object.
4) public boolean last():	is used to move the cursor to the last row in result set object.
5) public boolean absolute(int row):	is used to move the cursor to the specified row number in the ResultSet object.
6) public boolean relative(int row):	is used to move the cursor to the relative row number in the ResultSet object, it may be positive or negative.
7) public int getInt(int columnIndex):	is used to return the data of specified column index of the current row as int.
8) public int getInt(String columnName):	is used to return the data of specified column name of the current row as int.
9) public String getString(int columnIndex):	is used to return the data of specified column index of the current row as String.
10) public String getString(String columnName):	is used to return the data of specified column name of the current row as String.

Example of Scrollable ResultSet

Let's see the simple example of ResultSet interface to retrieve the data of 3rd row.

```
1.  import java.sql.*;
2.  class FetchRecord{
3.  public static void main(String args[])throws Exception{
4.
5.  Class.forName("oracle.jdbc.driver.OracleDriver");
6.  Connection con=DriverManager.getConnection("jdbc:oracle:thin:@localhost:1521:xe","system","oracle");
7.  Statement stmt=con.createStatement(ResultSet.TYPE_SCROLL_SENSITIVE,ResultSet.CONCUR_UPDATABLE);
8.  ResultSet rs=stmt.executeQuery("select * from emp765");
9.
10. //getting the record of 3rd row
11. rs.absolute(3);
12. System.out.println(rs.getString(1)+" "+rs.getString(2)+" "+rs.getString(3));
13.
14. con.close();
15. }}
```

ResultSetMetaData Interface (Metadata)

The metadata means data about data i.e. we can get further information from the data.

If you have to get metadata of a table like total number of column, column name, column type etc. , ResultSetMetaData interface is useful because it provides methods to get metadata from the ResultSet object.

Commonly used methods of ResultSetMetaData interface

Method	Description
public int getColumnCount()throws SQLException	it returns the total number of columns in the ResultSet object.
public String getColumnName(int index)throws SQLException	it returns the column name of the specified column index.
public String getColumnTypeName(int index)throws SQLException	it returns the column type name for the specified index.
public String getTableName(int index)throws SQLException	it returns the table name for the specified column index.

How to get the object of ResultSetMetaData:

The getMetaData() method of ResultSet interface returns the object of ResultSetMetaData.
Syntax:

```
1.  public ResultSetMetaData getMetaData()throws SQLException
```

Example of ResultSetMetaData interface :

```
1.  import java.sql.*;
2.  class Rsmd{
3.  public static void main(String args[]){
4.  try{
5.  Class.forName("oracle.jdbc.driver.OracleDriver");
6.
7.  Connection con=DriverManager.getConnection(
8.  "jdbc:oracle:thin:@localhost:1521:xe","system","oracle");
9.
10. PreparedStatement ps=con.prepareStatement("select * from emp");
11. ResultSet rs=ps.executeQuery();
12.
13. ResultSetMetaData rsmd=rs.getMetaData();
14.
15. System.out.println("Total columns: "+rsmd.getColumnCount());
```

```
16.         System.out.println("Column Name of 1st column: "+rsmd.getColumnName(1));
17.         System.out.println("Column Type Name of 1st column: "+rsmd.getColumnTypeName(1)
    );
18.
19.         con.close();
20.
21.         }catch(Exception e){ System.out.println(e);}
22.
23.         }
24.         }
```

```
Output:Total columns: 2
    Column Name of 1st column: ID
    Column Type Name of 1st column: NUMBER
```

JDBC RowSet

The instance of **RowSet** is the java bean component because it has properties and java bean notification mechanism. It is introduced since JDK 5.

It is the wrapper of ResultSet. It holds tabular data like ResultSet but it is easy and flexible to use.

The implementation classes of RowSet interface are as follows:

- JdbcRowSet
- CachedRowSet
- WebRowSet
- JoinRowSet
- FilteredRowSet

Let's see how to create and execute RowSet.

```
1.          JdbcRowSet rowSet = RowSetProvider.newFactory().createJdbcRowSet();
2.          rowSet.setUrl("jdbc:oracle:thin:@localhost:1521:xe");
3.          rowSet.setUsername("system");
4.          rowSet.setPassword("oracle");
5.
6.          rowSet.setCommand("select * from emp400");
7.          rowSet.execute();
```

It is the new way to get the instance of JdbcRowSet since JDK 7.

Advantage of RowSet

The advantages of using RowSet are given below:

1. It is easy and flexible to use
2. It is Scrollable and Updatable bydefault

Simple example of JdbcRowSet

Let's see the simple example of JdbcRowSet without event handling code.

```
1.  import java.sql.Connection;
2.  import java.sql.DriverManager;
3.  import java.sql.ResultSet;
4.  import java.sql.Statement;
5.  import javax.sql.RowSetEvent;
6.  import javax.sql.RowSetListener;
7.  import javax.sql.rowset.JdbcRowSet;
8.  import javax.sql.rowset.RowSetProvider;
9.
10. public class RowSetExample {
11.     public static void main(String[] args) throws Exception {
12.         Class.forName("oracle.jdbc.driver.OracleDriver");
13.
14.    //Creating and Executing RowSet
15.     JdbcRowSet rowSet = RowSetProvider.newFactory().createJdbcRowSet();
16.     rowSet.setUrl("jdbc:oracle:thin:@localhost:1521:xe");
17.     rowSet.setUsername("system");
18.     rowSet.setPassword("oracle");
19.
20.     rowSet.setCommand("select * from emp400");
21.     rowSet.execute();
22.
23.    while (rowSet.next()) {
24.             // Generating cursor Moved event
25.             System.out.println("Id: " + rowSet.getString(1));
26.             System.out.println("Name: " + rowSet.getString(2));
27.             System.out.println("Salary: " + rowSet.getString(3));
28.         }
29.
30.    }
```

31. }

The output is given below:

```
Id: 55
Name: Om Bhim
Salary: 70000
Id: 190
Name: abhi
Salary: 40000
Id: 191
Name: umesh
Salary: 50000
```

Transaction Management in JDBC

Transaction represents **a single unit of work**.

The ACID properties describes the transaction management well. ACID stands for Atomicity, Consistency, isolation and durability.

Atomicity means either all successful or none.

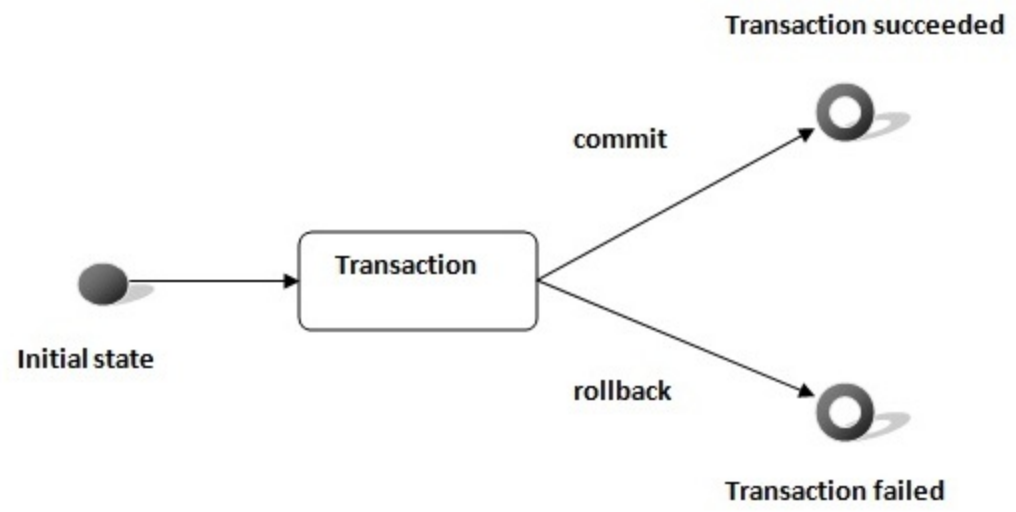
Consistency ensures bringing the database from one consistent state to another consistent state.

Isolation ensures that transaction is isolated from other transaction.

Durability means once a transaction has been committed, it will remain so, even in the event of errors, power loss etc.

Advantage of Transaction Mangaement

fast performance It makes the performance fast because database is hit at the time of commit.



In JDBC, **Connection interface** provides methods to manage transaction.

Method	Description
void setAutoCommit(boolean status)	It is true bydefault means each transaction is committed bydefault.
void commit()	commits the transaction.
void rollback()	cancels the transaction.

Simple example of transaction management in jdbc using Statement

Let's see the simple example of transaction management using Statement.

```
1.  import java.sql.*;
2.  class FetchRecords{
3.  public static void main(String args[])throws Exception{
4.  Class.forName("oracle.jdbc.driver.OracleDriver");
5.  Connection con=DriverManager.getConnection("jdbc:oracle:thin:@localhost:1521:xe","system","oracle");
6.  con.setAutoCommit(false);
7.
8.  Statement stmt=con.createStatement();
9.  stmt.executeUpdate("insert into user420 values(190,'abhi',40000)");
10. stmt.executeUpdate("insert into user420 values(191,'umesh',50000)");
11.
12. con.commit();
13. con.close();
14. }}
```

If you see the table emp400, you will see that 2 records has been added.