Compiler : At a shot conversion high level language to machinery language

Interpreter: Line by line conversion high level language to machinery language

Java compiler: compiles the source code and produce byte code

When we compile .java file by java compiler which is platform dependent ,it is produce a byte code which is platform independent

JRE-> java runtime environment

JVM -> java virtual machine

Jdk=(JC+JRE)

Types of java

Java standard edition (j2se)

Java Enterprise edition(J2EE)

Java micro edition(J2ME)

Program is set of instruction

Structure of a java file

File🡪class🡪method🡪statement

Class is a key word to create a class

Creating more then one method having same name but different argument type is known as method over loading

Class demo

{

Public static void main(String[] args)

{

System.out.println();

}

}

Compile a java class javac demo.java

Execute ajava class java classname

**DataTypes:**

1 byte = 8 bit

Primitive Data type: int, float, double, char, long, Boolean, byte, sort

Non Primitive Data Type: String, array

**Bit wise operator<**Will implement only on integral type**>**

Int a = 8 // 1 0 0 0

Int b = a << 2 // 1 0 0 0 0 0 **left shift** operator will add two zeros

int b = a>>2 // 1 0 **right shift** operator will remove two element

4 & 5 // 100 & 110 o/p 4(100) if both are true **AND operator** will return true

4 | 5 // 100 & 110 o/p 5(110) if both one true **OR operator** will return true

4 ^ 5 // 100 & 110 o/p 2(10) if both are different **X- OR operator** will return true

(~4) compliment operator -5

4 by default Int type means 4 bytes means 32 bits so we have to perform compliment for 32 bit

000000……..0100 Most significant bit is sign bit 1 means –ve and 0 means +ve

Now perform negation

111111……1011 find twos compliment

-sign bit 000000……0100

1

* 000000……0101 🡺 -5

**Call by value and call by reference**

**Local Varible\Class Variable(Static variable)\ Instance Variable\Reference variable**

*Local Variable*

Variable declare inside the body of the method is called Local variable. We can use the local variable with in the method.

A local variable can’t be declare with **static keyword**

*Instance Variable*

These variables belong to the instance of a class, thus an object. And every instance of that class (object) has it's own copy of that variable. Changes made to the variable don't reflect in other instances of that class.

**public** **class** testJava {

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

instance i1 = **new** instance();

i1.data = 34;

instance i2 = **new** instance();

i2.data = 45;

System.***out***.println(i1.data); **O/P 34**

System.***out***.println(i2.data);  **O/P 45**

}

}

**class** instance{

**public** **int** data;

}

These are also known as static member variables and there's only one copy of that variable that is shared with all instances of that class. If changes are made to that variable, all other instances will see the effect of the changes.

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

instance i1 = **new** instance();

i1.*data* = 34;

instance i2 = **new** instance();

i2.*data* = 45;

System.***out***.println(i1.*data*); **O/P 45**

System.***out***.println(i2.*data*); **O/P 45**

}

}

**class** instance{

**public** **static** **int** *data*;

}

**This**

this is use to initialize no static variable/data member of a class

this is a keyword which represent current instance and it is used to refer non static members of current in

stance

this.memberName

**this keyword cannot be used inside *static* method**

*class rectangle{*

*int length;*

*int breadth;*

*void inti(int i,int j) {*

*this.length=i;*

*this.breadth=j;*

*}*

*void printDimention() {*

*System.out.println("length : "+length+" breadth: "+breadth);*

*}*

*}*

*class println{*

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

*{*

*rectangle r = new rectangle();*

*r.inti(10,20);*

*r.printDimention();*

*}*

*}*

Constructor

Accessmodifier name(Arguments)

{ ----------------------

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

}

Constructor is a special type of method which will execute immediately when object is created

* Constructor name should always same as class name
* Return type is not allowed for constructor not even void
* Static keyword is not allowed for constructor
* Allocate memory for non static members in heap , execute constructor , return the reference
* Constructor can be develop with or without arguments
* Constructor without argument is known as default constructor and constructor with argument is known as parameterized constructor
* Java implicitly add no argument constructor is a class does not have any constructor
* Constructor are use to initialize non static variable at the time of object creation

**public** **class** construct {

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

area a =**new** area(10,20);

//a.area(); //Wrong

area b = **new** area(); **O/P**

a.printdimention(); length 10 bth 20

} Area constructor is executing

} length 0bth 0

inside printdimention

**class** area{ length 10 bth 20

**int** length;

**int** bth;

area(**int** i,**int** b)

{

length=i;

bth=b;

System.*out*.println("length "+ length+" bth "+bth);

}

area()

{

System.*out*.println("Area constructor is executing ");

System.*out*.println("length "+ length+"bth "+bth);

}

**void** printdimention()

{

System.*out*.println("inside printdimention");

System.*out*.println("length "+ length+" bth "+bth);

}

}

Constructor Overloading

Creating multiple constructor in a class with different argument list is known as constructor overloading

When constructor is overloaded object can be created by calling any one overloaded constructor

this()

call to this is used to call current class constructor

this() involks a specific constructor based on parameter

call to this must be the first statement with in the constructor

**public** **class** construct {

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

area b = **new** area();

b.printdimention();

}

} **O/P**

Area pameterized cunstructor is executing

**class** area{ length 20 bth 40

**int** length; Area constructor is executing

**int** bth; length 20 bth 40

inside printdimention

area(**int** i,**int** b) length 20 bth 40

{

length=i;

bth=b;

System.*out*.println("Area pameterized cunstructor is executing");

System.*out*.println("length "+ length+" bth "+bth);

}

area()

{ **this**(20,40);

System.*out*.println("Area constructor is executing ");

System.*out*.println("length "+ length+" bth "+bth);

}

**void** printdimention()

{

System.*out*.println("inside printdimention");

System.*out*.println("length "+ length+" bth "+bth);

}

}

Inheritance

It is a process of accruing members of one class to another class. Java provides a key word *extends* for inheritance

*SubClass extends SuperClass*

Multiple inheritance is not possible in java

Class A extends Class B extends Class C

Object Class

* It is a built in class
* It is a default super most class
* If there is a class which does not extends any supper class explicitly than that class by default extends object class
* All class created in java are direct or indirect subclass to object class

Wrapper class methods are **equals, notify, notifyall, tostring, wait, finalize**

Constructor chain

* Calling super class constructor to sub class constructor is known as constructor chain
* Call to super(Super();) is used to call immediate super class constructor
* Call to super should be the 1st statement with in the constructor
* If a constructor does not contain call to super (super();)than java implicitly calls super in first line
* Because of constructor chain when a sub class object is created all constructor from super most class to sub class are executed

**class** mamal

{

mamal()

{

System.*out*.println("Mamal class is executing");

}

}

**class** animal **extends** mamal

{

animal(String s) //parameterized constructor

{

**super**(); [or // **super**();]

System.*out*.println(s+"Animal class is executing");

}

}

**class** food **extends** animal{

food()

{

**super**("cat"); // Mandatory

System.*out*.println("food is needed for animal");

}

}

**public** **class** Interf { **O/P**

Mamal class is executing

**public** **static** **void** main(String[] args) { catAnimal class is executing

food is needed for animal

food f = **new** food();

}

}

When explicit call to super is required?

When we need to call parameterized constructor of super class

How to use class of different package?

Either use fully qualified name or import a class using import keyword, A class name with package name is known as fully qualified name.

Import com.oce19.pac.b;

Import statement should written after package statement

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | Private | Default | Protected | Public |
| With in the class | Yes | Yes | Yes | Yes |
| Other class in side of package | No | Yes | Yes | Yes |
| Other class outside the package | No | Yes | Yes(If inherited in sub class) | Yes |

OverRiding

* Changing the implementation of inherited method in sub class is known as method overriding
* Writing new implementation for superclasss method signature according to subclass requirement is known as method over riding
* For over-riding inheritance is must
* While over riding methodName, argument list, return type should be same in sub class
* Access specifier can be changed but visibility cannot be decreased

|  |  |
| --- | --- |
| Super class | Subclass |
| Default | Default,Protected,Public |
| Protected | Protected,Public |
| Public | Public |

**class** A

{

**void** test()

{

System.*out*.println("Class A is executing");

}

}

**class** B **extends** A

{

**void** test(){

System.*out*.println("Class B is executing");

}

}

**class** overLoad{

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

{

System.*out*.println("main class is executing");

A a = **new** A();

a.test();

B b=**new** B();

b.test();

}

}

Final

* Final method cannot be override
* Final class cannot be inherited but final class can extends some other class
* Final variable cannot be changed

**final** **int** i = 10;

System.*out*.println(i);

**int** i = 20 //Compilelation error

* Final class can extends some other class

Super

Super is used to call immediate super class data member/member function

**class** A

{ **int** obj = 10;

**void** test()

{

System.*out*.println("Test in A");}

}

**class** B **extends** A

{ **int** obj = 20;

**void** test()

{

**super**.test(); //If two methods are over load ro call method of parent

class we use super

System.*out*.println(**super**.obj);

System.*out*.println("Test in B");

System.*out*.println(obj);

}

}

**class** overLoad{

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

{

B a = **new** B();

a.test();

}

}

|  |  |
| --- | --- |
| This | Super |
| this() is used to called current class constructor | Super() is used to call immediate super class constructor |
| This use to refer current instance of non static method and variable  this.variableNAame | Super is used to immediate super class nonstatic method and variable  Super.variableName |
| this.methodName() | Super.methodName() |

Abstract Method

*Public abstract void test();*

* A method without body is known as abstract method
* A abstract method must be end with semi column
* Method which contain body and head is known as concrete method
* Abstract method should be declare with a key word abstract

Abstract Class

* Abstract class declare with a key word abstract
* Abstract class can not be inherited/cannot be create an object
* If the class contain at least one abstract method then that class must be declare as abstract

Inheriting abstract class

When a class extends abstract class in sub class over riding abstract method is mandatory otherwise subclass became abstract

Abstract method forces subclass to implement a method without altering method signature

*abstract* ***final*** *void test();*

*private* ***abstract*** *void test();*

*abstract* ***static*** *void test(); (A*long with abstract method we cannot use static key word*)*

**abstract** **class** shape

{

**void** test()

{

System.*out*.println("Test method is executuing");

}

**abstract** **void** getArea();

}

**class** triangle **extends** shape

{

**void** getArea()

{

System.*out*.println("Triangle getArea is executing");

}

}

**class** overLoad{

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

{

//shape sh = new shape(); //CE (can not create an object of abstract class)

triangle tl = **new** triangle(); O/P

tl.getArea(); Triangle getArea is executing

tl.test(); Test method is executuing

}

}

Up casting

Class B extends A

A a = (A) new B()q

Type1 var = (Type1)new Type2

Assigning sub class object reference to super class variable is known as upcasting.

Up casting is automatic

Using super class variable only super class member can be referred

A a = (A) new B()

a.test1(); class A method

a.test2(); //CE class B method

**Polymorphism**

It a concept of one object /method call showing different behavior

1. Run time Polymorphism
2. Compile time Polymorphism

Runtime Polymorphism

Call to over ridden method is resolved at run time based on instance type is known as run time polymorphism

When over ridded method is called method implementation will be executed based on instance type not based on variable type

**abstract** **class** shape{

**void** getArea()

{

System.*out*.println("getArea method is executuing");

}

Void printArea()

{

System.*out*.println("printArea method is executuing");

}

}

**class** triangle **extends** shape{

**void** getArea()

{

System.*out*.println("Triangle getArea is executing");

}

}

**class** square **extends** shape{

**void** getArea()

{

System.*out*.println("square getArea is executing");

}

}

**class** overLoad{

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

{

shape sh ;

sh= **new** triangle(); O/P<Over riding Methods>

sh.getArea(); //Runtime Polymorphism Triangle getArea is executing

sh =**new** square();

sh.getArea(); square getArea is executing

shape sh1 = new triangle();

sh1.printArea(); print area method is executing

triangle tr = new shape();

tr.printArea(); //Error

}}

**package** FrameWorkTestNG;

**public** **class** testCase {

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

{

test2 tst = **new** test2();

tst.demo1();

}}

Inside test1 class

**class** test1 Inside test2 class

{

**public** **void** demo3()

{

System.***out***.println("Inside test1 class");

}}

**class** test2 **extends** test1

{

**public** **void** demo1()

{

**super**.demo3();

System.***out***.println("Inside test2 class");

}

}

**package** FrameWorkTestNG;

**public** **class** testCase {

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

{

test1 tst = **new** test2();

tst.demo3();

}} <Not Overriding methods>

Inside test1 class

**class** test1

{

**public** **void** demo3()

{

System.***out***.println("Inside test1 class");

}}

**class** test2 **extends** test1

{

**public** **void** demo1()

{

System.***out***.println("Inside test2 class");

}}

Compile time Polymorphism

It is a technique in which call to over loaded method is resolved at compile time based on argument

For RTP inheritance, overriding, upcasting is mandatory

Main method **can be overload** but cannot be override because **it is static**

**Binding**

Connecting a method call to a method implementation is known as **Binding**

1. Early binding(static binding)
2. Late binding(Dynamic binding)

In early binding method call is connected to a method body at compile time . For static, Private, final method call early binding takes place

In late binding JRE resolve method call at run time, for non-static, final and private late binding takes place

Up casting

Assigning sub class object reference to super class is known as up casting

A a1 = new B() 🡪 Auto up casting

<Super class> obj = new <Sub class>

Up casting is automatic

Down casting

Assigning superclass variable which contains subclass instance to subclass variable is known as down casting .It is always explicit

Animal a = new dog();

Dog d =(dog)a;

d.play();

Instance of

Is a relationship

B Is.A C

B🡪 Sub class, C🡪Super class

|  |  |
| --- | --- |
| OverLoading | OverRiding |
| Overloaded method must have same name and different argument list | Override method must have same name same argument list |
| OverLoad method can have any return type | OverRide method must have same return type |
| Over Loaded method can have any access specifier | OverRide method can have higher visibility or same visibility access specifier |
| Final, Private, Static method can be override | Final, Private, Static method cannot be overloaded |
| Constructor can be overloaded | Constructor cannot be overRided |
| Use to achieve compile time polymorphism | Use to achieve Run time polymorphism |

Interface

An interface is just a template that is implemented by a class. And to achieve the abstraction we use interface.

It specifies what fields and methods the class should have but without providing more details

* Java provide a key word *interface* to create interface
* Interface is 100% abstract **hence it cannot be instantiated**
* In interface all methods are by default public and abstract
* Since static and abstract is invalid combination static method is not allowed in interface
* Interface allow only constraints(All variable in interface are by default public static and final)

*Abstract interface name()*

*{*

*Public static final int i=10;*

*Public abstract void test();*

*}*

* Interface is also a type in java(Which means variable can be declare with interface name )
* Java provide keyword *implement* to inherit method from interface
* When a java class implement interface in sub class method should be override otherwise subclass became abstract
* While implementing interface method in class **public** keyword is must.
* An interface can **extends** another interface
* If a class extends a class and also implements an interface , extends keyword should come before implements

**interface** i1 {

**void** test1();

}

**interface** i2

{

**void** test2();

}

**class** opt{

**void** test3()

{

System.*out*.println("test3 in opt");

}

}

**class** demo **extends** opt **implements** i1,i2

{

**public** **void** test1()

{

System.*out*.println("test1 in demo");

}

**public** **void** test2()

{

System.*out*.println("test2 in demo");

}

}

**public** **class** Interf {

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

i1 intr = **new** demo(); O/P

intr.test1(); test1 in demo

i2 itr = **new** demo();

itr.test2(); test2 in demo

demo d =**new** demo();

d.test3(); test3 in opt

}

}

* Hiding implementation details and showing only functionality is known as abstraction
* We can achieve abstraction by using abstract class or by using interface
* Create an interface and declare abstract method
* Provide implementation for abstract method in sub class
* User write a program for interface type

interface B {...}

interface A extends B {...} // allowed

interface A implements B {...} // not allowed

implements means implementation, when interface is meant to declare just to provide interface not for implementation.

A 100% abstract class is functionally equivalent to an interface but it can also have implementation if you wish (in this case it won't remain 100% abstract), so from the JVM's perspect

ive they are different things.

Also the member variable in a 100% abstract class can have any access qualifier, where in an interface they are implicitly public static final.

[ANS](https://stackoverflow.com/questions/3921412/why-an-interface-can-not-implement-another-interface)

Advantages

Loose coupling (two program integrate in such a way that any changes in implementation layer will not effect users program)

Encapsulation

It is technique in which making fields (data member) private and providing an access to those private fields through public method

Public method which use to update private field is known as setter

Public field which use to read private field is known as getter

Encapsulation is also referred as data hiding because fields are declare as private

Advantages

* A class can have total control on fields
* A class can change its storage implementation without effecting other user program
* A class can may it’s fields readonly or readwriteonly
* Users of that class no need to learn how the data is organize

**class** printStream

{

**public** **void** println()

{ }

**public** **void** println(**int** i)

{ }

}

**class** system **extends** printStream

{

**public** **static** printStream *out* = **new** printStream();

}

**public** **class** Interf {

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

system.*out*.println(10);

}

}

**class** printStream

{ **int** k=20;

**int** j=30;

**public** **void** println()

{

}

**public** **void** println(**int** i)

{

}

}

**class** system

{

**int** i =10;

printStream out = **new** printStream();

**boolean** test()

{

System.*out*.println(out.j);

System.*out*.println(out.k);

**return** **false**;

} O/P 30 30 10 30 20 false

}

**public** **class** Interf {

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

system s = **new** system();

System.*out*.println(s.out.j);

System.*out*.println(s.out.j);

System.*out*.println(s.i);

System.*out*.println(s.test());

}

0

}

**private** String company = "Aricent";

**public** String getCompanyName()

{

**return** company;

}

**Encapsulation** is hiding the implementation details which may or may not be for generic or specialized behavior(s).

**Abstraction** is providing a generalization (say, over a set of behaviors)

Object class

String Class

String s1 = new String("Java");

System.out.println(s1); O/P----🡪Java

String s2 = "Java";

System.out.println(s2); O/P---🡪 Java

System.out.println(s1==s2); O/P--🡪False

System.out.println(s1.equals(s2));O/P--🡪 True

* String is a **final class**
* String class is used to store a group of character
* We can create string Object in two ways

1. Using new operator/Keyword

String s1 = new String("Java");

1. Direct assigning string literal

String s2 = "Java";

* In string class two string class is override to return content of string Object
* In string class equals method is override to compare to string instance based on characters

String s1 = “Aricent”

String s2 = “aricent”

System.Out.Println(s1.equals(s2)); O/P---🡪 False

System.Out.Println (s1.equalsIgnoreCase(s2)); O/P--🡪 True

Object x = new String(“Aricent”); Run time polymorphism

Object x =”Aricent”;

String s =”AricentA”;

s.**charAt**(0); O/P-> A

s.**indexOf**(“e”); O/P---🡪 4

s.**indexOf**(“x”); O/P--🡪 -1

s.**indexOf**(“Aricent”) O/P---🡪0

s.**indexOf**(“A”) O/P--🡪7

**public** **class** string {

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

String s= "CuttuCc";

String s2="" ;

System.*out*.println(s.length());

**for**(**int** i =s.length()-1;i>=0;i--){

s2 += s.charAt(i);

}

System.*out*.println(s2);

**if**(s.equalsIgnoreCase(s2)) {

System.*out*.println(s+" is a pallindrom");

}

**else**{

System.*out*.println(s+" Not a pallindrom");

} } }

String s= "derA123cef5gt77";

**int** s2;

**for**(**int** i = 0; i<=s.length()-1;i++)

{

s2 = s.charAt(i);

**if**(s2>=48 && s2<=57)

{

System.*out*.println(s2);

}

}

SubString

String s = “SachinTendulkar”

s.subString(6); o/p🡪Tendulkar

s.subString(0,6); o/p🡪 Sachin

Replace

String s= "AricentA";

System.*out*.println(s.replace("A", "D")); O/P-🡪DricentD

System.*out*.println(s.replace("r"," ")); O/P-🡪A icentA

System.*out*.println(s.replace("r","")); O/P-🡪AicentA

Counting String in a word

String s = “developer”;

System.out.println(s.length()-s.replace("e", "").length()); O/P-🡪 3

String s= "AricentA";

String st = "Aricent is a good company";

System.*out*.println(s.toLowerCase()); O/P-🡪aricenta

System.*out*.println(s.toUpperCase()); O/P-🡪ARICENTA

System.*out*.println(st.contains("Aricent")); O/P-🡪true

System.*out*.println(st.startsWith("Aricent")); O/P-🡪true

System.*out*.println(st.endsWith("Aricent")); O/P-🡪false

Replace **two pattern** at a time

String s = "aricent Technology";

System.***out***.println(s.replaceAll("[aeiou|\\s+|]","")); **o/P** rcntTchnlgy

Trim(Remove the space at the beginning and ending)

String s= " Aricent ";

System.*out*.println(s.trim()); O/P-🡪Aricent

**Immutable**

String is immutable. Which means once we create string object we cannot perform any changes. This non changeable behavior is called immutability

* String which are created *without new* keyword are created in *string pool*
* String pool does not allow to duplicate objects
* Before creating a new object in string pool JVM verifies whether a object with same content exit or not. If exit it returns reference of object instead of creating new one
* String which are created with new keyword are created in heap outside string pool
* Duplicate string object can be create using new key word

Why string is immutable?

A string instance might be shared reference with many variables if one variable modifies the string instance, that modification cannot affect other variable. That’s why string is made as immutable.

|  |  |
| --- | --- |
| String | String Buffer |
| In String once we create any string object we cannot perform any changes on exiting object | Once we create string buffer object we can perform any type of changes on exiting object |
| It is immutable | It is mutable |
| String S = “Java”;  S,concat(“Developer”);  S.O.P(S); O/P-🡪Java | StringBuffer sb = new StringBuffer(“Java”);  Sb.concat(“Developer”);  S.O.P(sb); O/P-🡪JavaDeveloper |

|  |  |
| --- | --- |
| String Buffer | String Builder |
| Every method present inside string buffer is synchronized | No method present inside string Builder is synchronized |
| At a time one thread is allowed to operate on string buffer hence it is thread safe | At a time multiple thread allow to operate on string Builder hence it is not thread safe |
| It increase waiting time of thread hence it’s performance is low | Threads are not required to wait to operate string buffer hence it’s performance is high |
| Introduce in 1.0 version | Introduce 1.5 version |

**Wrapper Class**

Wrapper class is used to perform boxing and un boxing.

<**Integer, Character, Short, Boolean, Long, Float, Double**>

Automatic conversion of primitive data type to equivalent wrapper type class object is known as **boxing.**

Extracting primitive value from wrapper class object is known as **un boxing.**

int i = 50;

integer i1 = new integer(i); //boxing

integer i1 = 100;

integer I = 10; //unboxing

int i1 =I;

From Java 1.5 version boxing and unboxing operation is automatic which means boxing and unboxing is required implicitly java perform. Wrapper class is coming from **java.lang** package

**SingleTon** **Class** in Java

SingleTon class creates only one object/instance of the class. We can call singleton method inside a static and non-static method.

**public** **class** exception{

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

SingleTon ob = SingleTon.*getInstance*();

SingleTon ob1 = SingleTon.*getInstance*();//This will create a same instance

}

}

**class** SingleTon

{

//create a static object

**static** SingleTon *obj* = **new** SingleTon();

//create a private method

**private** SingleTon()

{

}

//return static object within static method

**public** **static** SingleTon getInstance()

{

**return** *obj*;

}

}

**Array**

Array is group of similar type of element. Array is created in heap

Type[] var = new type[size]

int[] arr = new int[5]

arr[0] = 12;

arr[1]= 23;

arr[2]= 56;

for(int i:arr)

{System.out.println(i)} O/P: 12, 23, 56, 0, 0

**Find the Biggest Element of an Array**

**int**[] arr = {12, 34, 56};

**int** big =0 ;

**for**(**int** i :arr)

{

**if**(big<i)

{

big=i;

}

}System.*out*.println("biggest element :"+big);

Sorting of an array ascending order

**int**[] arr = {5,12, 45, 2,1234,33,1,0};

**for**(**int** i=0;i<=arr.length-1;i++)

{

**for**(**int** j=i+1;j<=arr.length-1;j++)

{

**if**(arr[i]>arr[j])

{

**int** temp = arr[i];

arr[i]=arr[j];

arr[j]=temp;

}

}

}

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

{System.*out*.println(x);}

User input of an array

Scanner scn = **new** Scanner(System.*in*);

System.*out*.println("enter length: ");

**int** n = scn.nextInt();

**int**[] arr = **new** **int**[n] ;

**for**(**int** i=0;i<=n-1;i++)

{

**int** ad = scn.nextInt();

arr[i]= ad;

}

**List :**

* List is index based, which means element are arranged W.R.T an index and index starts from Zero

|  |  |
| --- | --- |
| Array List | Linked List |
| Array list uses resizeable array to store element | Linked list uses linklist data structure technique to store element |
| Array list is fast when element ‘s are added and removed at the end | Linked list is first when element are added or removed in between the list |

|  |  |
| --- | --- |
| Vector | Array list |
| Vector uses resizeable array to store element | Array list uses resizeable array to store element |
| Vector is a legacy class | Array list is added in 1.2 version |
| Vector methods are synchronized hence it is thread safe | Array list method are not synchronized |
| It is thread safe | It is not thread safe |

A code is **thread safe** if it behaves correctly when accessed from multiple **threads**

**synchronized** is a **Java** keyword. It means that the method cannot be executed by two threads at the same time

ArrayList arr = **new** ArrayList();

arr.add("hi");

arr.add("World");

arr.add("Aricent");

System.*out*.println("size is "+arr.size());

**for**(Object x:arr)

{

System.*out*.println(x);

}

System.out.println(arr.get(2));

arr.add(3, "Change"); //over ride the element at 3

arr.set(3, "Stay"); //insert an element at 3

arr.remove(3);

List lst = **new** ArrayList<String>();

List vtr = **new** LinkedList<Integer>();

List vtr1 = **new** Vector<Character>();

List lst1 = **new** ArrayList<Double>();

**Queue:**

Queue implements fast in fast out. First element is called as head, without remove head element 2nd element cannot be retrieve

Peek() 🡪 Return the head element

Poll() 🡪Remove and return head element

Queue q = **new** PriorityQueue();

q.add("Aricent");

q.add("Amdoc");

q.add("CF");

q.add("Accenture");

/\*for(Object x:q)

{

System.out.println(x);

}\*/

**while**(q.peek()!=**null**) o/p- Accenture Amdoc Aricent CF

{

System.*out*.println(q.poll());

}

**Set:**

Hash set: Iterate element in random order

LinkedHash Set: Iterate element in insertion order

Tree set: Iterate element in shorting order

Set s1 = **new** HashSet();

set s2 = **new** LinkedHashSet();

Set s = **new** TreeSet();

**Iterator:**

Iterator is used to iterate element from a collection object

It has three methods

HashNext(): This method verifies weather the next element is exit or not for iteration. This method returns Boolean

Next(): This method returns next element in a collection object

Remove(): This method removes current element from the collection object

*Set s = new TreeSet();*

*s.add(“java”);*

*s.add(“demo”);*

*s.add(“Aricent”);*

*iteration itr = s.iteration();*

*While(s.hashNext())*

*{*

*System.out.println(itr.next());*

*}*

List Iterator(Only for list)

List iterator is used to iterate element in both forward and backward direction from the list object

List class provides a public method listIterator which returns an object of type ListIterator

Boolean hashNext();

Object next();

Boolean hashPrevious();

Object previous();

Void remove();

Arraylist arr = new ArrayList();

a.add(10); a.add(20); a.add(30);

ListIterator lst = a.ListIterator();

While(listItr.hasNext()) O/P- 10 20 30 30 20 10

{ System.out.println(lst.next()); }

While(lst.hasPrevious())

{ System.out.Println(lst.previous()) }

|  |  |
| --- | --- |
| Iteraor | ListIterator |
| Use to iterate the element only forward direction | Use to iterate the element from both forward and backward direction |
| It uses all the collection | Only uses list |
| It has three methods | It has five methods |

**Set, List** and **Map** are interfaces, which defines core contract e.g. a Set contract says that it can not contain duplicates. Based on our knowledge of List, Set and Map let's compare them on different metrics.

The main difference between List and Set interface in Java is that **List** **allows duplicates** while **Set doesn't allow** duplicates. All implementation of Set honor this contract.   
  
While a **Map** holds two objects per Entry e.g. a key and a value and It may contain duplicate values but **keys are always unique**

**Null elements**

The list allows null elements and you can have many null objects in a List because it also allowed duplicates. Set just allow one null element as there is no duplicate permitted while in Map you can have null values and at most one null key

When we need to retrieve element from list we use array List, while LinkedList is more suitable for frequently adding and removing elements from List.

[Hashtable](http://java.sun.com/javase/7/docs/api/java/util/Hashtable.html) is [synchronized](https://stackoverflow.com/questions/1085709/what-does-synchronized-mean), whereas [HashMap](http://java.sun.com/javase/7/docs/api/java/util/HashMap.html) is not. This makes [HashMap](http://java.sun.com/javase/7/docs/api/java/util/HashMap.html) better for non-threaded applications, as unsynchronized Objects typically perform better than synchronized ones.

1. [Hashtable](http://java.sun.com/javase/7/docs/api/java/util/Hashtable.html) does not allow null keys or values. [HashMap](http://java.sun.com/javase/7/docs/api/java/util/HashMap.html) allows one null key and any number of null values.

**Map**

Map store key value pair (hashMap, LinkedHashMap, treeMap)

Map m1 = **new** Hashtable();

Map<String,Integer> m2 = **new** HashMap<String,Integer>();

Map<String,String> m = **new** LinkedHashMap<String,String>();

Map<String,Integer> m1 = **new** TreeMap<String,Integer>();

Put(Key,value)

get(key)

size()

remove(key);

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

lm.put("Mango", 50);

lm.put("Banana",48);

lm.put("Apple", 120);

**for**(Map.Entry m:lm.entrySet())

{

System.*out*.println(m.getKey()+" --->"+m.getValue());

}

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

ArrayList<Integer> ls = **new** ArrayList<Integer>();

ls.add(45);

ls.add(23);

ls.add(67);

ArrayList<Integer> lss = **new** ArrayList<Integer>();

lss.add(78);

lss.add(89);

lss.add(01);

lm.put("mark",ls);

lm.put("att",lss);

**for**(Entry<String, List<Integer>> m:lm.entrySet())

{

System.*out*.println(m.getKey());

System.*out*.println(m.getValue());

}

Exception

It is an unexpected event that might occur at run time that might block the program execution

Exception occurs at runtime. Java provides try catch block to handle exception

When exception thrown it should be caught or handled otherwise execution will be terminated.

*Try*

*{ int a=5;*

*Int b=0;*

*Int c = a/b; (Risky Statement)*

*}*

*Catch(Arithmetic Exception e)*

*{*

*C=10/2;(Alternative Statement that will execute when exception occurs)*

*}*

Throwable:

* Error
* Exception

1. Checked Exception(Compiler checks at compile time)

Filenotfound exception, Interrupt Exception

1. Unchecked Exception/Runtime Exception(compiler does not check at compile time)

Arithmetic Exception, Null Pointer Exception, ArrayIndexBoundOfException

* For checked exception compiler checks whether the handler provide or not at compile time
* For unchecked exception compiler does not checked for an handler at compiler
* Checked exception must be handled using try/catch or throws block
* Unchecked exception are not mediatory to use try/catch block
* When there are multiple catch block for a try, catch block order should be from specific to generic type
* We can write multiple catch block for a try
* A try block can exist without catch block if finally is there

*try{-----*

*}*

*Catch{*

*}*

*Finally{*

*}*

Exception Propagation

* When there is an exception in a method and there is no handler in that method than that exception must be propagate to next method in stack
* If unchecked exception is not handled in a method than by default the exception propagates to next method
* If checked exception is not handled in a method than it must be propagate explicitly by declaring throws key word otherwise compilation error
* Throws is a key word use to declare unchecked exception. It is used in method signature

Using throws we can declare multiple checked exception

Void main() throws Exceptiontypr1,Exceptiontype2

**User Define Exception**

**public** **class** exception1{

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

**try** {

**throw** **new** invalidException("Aricent");

} **catch** (invalidException e) {

e.printStackTrace();

}

}

}

**class** invalidException **extends** Exception

{

invalidException(String s)

{

System.***out***.println("Enter a valid number");

}

}

**File Read Write append function**

FileReader fr =**null**;

BufferedReader br = **null**;

**try** {

br= **new** BufferedReader(**new** FileReader("E:\\STUDY\\read.txt"));

String line;

**try** {

**while**((line = br.readLine())!=**null**)

{

System.*out*.println(line);

}

} **catch** (IOException e) {

// **TODO** Auto-generated catch block

e.printStackTrace();

}

} **catch** (FileNotFoundException e) {

// **TODO** Auto-generated catch block

e.printStackTrace();

}

FileWriter fw ;

BufferedWriter bw;

bw = **new** BufferedWriter(**new** FileWriter("H:////TestData//Demo.txt"));

//for append data in a exiting file (Even if a file is not exit it will create a file and write the data)

//bw = **new** BufferedWriter(**new** FileWriter("H:////TestData//Demo.txt",true));

bw.write("Hi hello 400:08311998:Inprise Corporation:");

bw.newLine();

bw.write("Hi hello 400:08311998:Inprise Corporation:");

bw.flush(); // flush forces any buffered output bytes to writtern in file

bw.close();

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.

**An interface is an empty shell**. There are only the signatures of the methods, which implies that the methods do not have a body

|  |  |
| --- | --- |
| **Abstract class** | **Interface** |
| 1. Abstract class can **have abstract and non-abstract** methods. | Interface can have **only abstract** methods. Since Java 8, it can have **default and static methods** also. |
| 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 provide the implementation of interface**. | Interface **can't provide the implementation of abstract class**. |
| 5) The **abstract keyword** is used to declare abstract class.  Abstract class can have constructor in them | The **interface keyword** is used to declare interface.  Interface doesn’t have any constructor |

Can we use static with over ride method?

No It will be difficult to instantiated static method at run time

**Working with Folder’s in java**

File f = new File(“h:\\TestData\\CreateFolder”);

f.**mkdir**();

for(File listF: f){

if(!listF.**isDirectory**()){

file.**delete()**;

}

}

To find current directory path

String currentDirectory = System.getProperty("user.dir")

**Delete all the files and directory recursively**

public class DeletingFilesRecursively {

   static void deleteFolder(File file){

      for (File subFile : file.listFiles()) {

         if(subFile.isDirectory()) {

            deleteFolder(subFile);

         } else {

            subFile.delete();

         }

      }

      file.delete();

   }

   public static void main(String args[]) {

      String filePath = "E://ExampleDirectory//";

      //Creating the File object

      File file = new File(filePath);

      deleteFolder(file);

      System.out.println("Files deleted........");

   }

}

**Jagged array : Array contains different sized array**

**int**[][] arr = {{1,2,3,5},{3,4,6},{2,3,6,4}};

**for**(**int**[] ar:arr)

{

**for**(**int** i:ar)

{

System.***out***.print(i);

}System.***out***.println();

}

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

{

**for**(**int** j =0;j<arr[i].length;j++)

{

System.***out***.print(arr[i][j]);

} System.***out***.println();

}

**Array(metrix) Multiplication**

Two array only be multiply only if column and row of both Metrix are same

Int[][] arr1 = {{1,2,3},{3,4,5}};(2\***3**)

Int[][] arr2 = {{7,8},{9,10},{11,12}};(**3**\*2) out put of this matrix will 2\*2

**int**[][] arr = {{1,2,3,5},{3,4,6,8},{2,3,6,4}}; //3 4

**int**[][] ar ={{2,3,4},{5,43,6},{8,6,9},{6,8,6}};//4 3

**int** sum =0;

**for**(**int** i=0;i<3;i++)

{

**for**(**int** j =0;j<3;j++)

{

**for**(**int** k=0;k<4;k++)

{

sum = sum+arr[i][k]\*ar[k][j];

}System.***out***.print(sum+",");

sum=0;

}System.***out***.println();

}

**Transpose of a Array**

int[][] arr = {{1,2,3,5},{3,4,6,8},{2,3,6,4}};

for(int i = 0;i<arr[1].length;i++)

{

for(int j =0;j<arr.length;j++)

{

System.out.print(arr[j][i]+" ");

}System.out.println();

}

**Multi Threading**

Doing two task at a time is know multi tasking. Thread is unit of process . If you take a process and break into sub process is called thread.

Three different application of thread

Why we need thread

In todays we have octacore cpu . when we create java application java use one thread to run on a CPU. But at the same time we can run the java application on eight different cores which will decrease the execution time.

Even if we have not created any thread by default main thread is running.<Exception in thread main….>

If 5 uses wants the response from the server. Server creates multiple thread and sends the response. We can send Asynchronous request

For game development Thread are very important

Just saying by extends thread it is not actually calling the function. So we have to start the start function for that.

Start is method that belongs to thread class

Every time we execute start internally thread execute a method. Because start always call run method. Run is an internal method of Thread so we needs to override the run method at the time of execution.

**public** **class** thread {

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

Hi obj1 = **new** Hi();

obj1.start();

Hello obj2 = **new** Hello();

obj2.start();

}

}

**class** Hi **extends** Thread

{

**public** **void** run()

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

{ System.***out***.println("Hi");

**try** {

Thread.*sleep*(500);

} **catch** (InterruptedException e) {

}

}}}

**class** Hello **extends** Thread

{

**public** **void** run()

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

{ System.***out***.println("Hello");

}}}

Here both the process are reaching to scheduler at the same time. So scheduler well execute the thread based upon certain parameter’s. Criteria like less priority execution time thread got executed.

**Runnable Interface**

**public** **class** thread {

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

Runnable obj1 = **new** Hi();

Runnable obj2 = **new** Hello();

Thread t1 = **new** Thread(obj1);

Thread t2 = **new** Thread(obj2);

t1.run();

t2.run();

}

}

**class** Hi **implements** Runnable

{

**public** **void** run()

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

{ System.***out***.println("Hi");

**try** {

Thread.*sleep*(500);

} **catch** (InterruptedException e) {

}

}}}

**class** Hello **implements** Runnable

{

**public** **void** run()

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

{ System.***out***.println("Hello");

}}}

The run() method comes from the Runnable interface but the start() method is only declared in the [Thread class](http://javarevisited.blogspot.com/2011/02/how-to-implement-thread-in-java.html). Since java.lang.Thread class implements Runnable interface, you can access the run() method from an instance of Thread class.

Runnable obj = **new** Hello();

obj.run();

**Runnable obj = new Hello();**

**obj.start(); //Invalid Runnable can’t call start directly**

Runnable obj = **new** Hello();

Thread t = **new** Thread(obj);

t.start();

By executing start method at each time thread get started

But for run method thread continuous its running

**package** com.CISCO.rest.com.CISCO.rest;

**public** **class** thread{

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

Runnable obj1 = **new** Runnable()

{

**public** **void** run() **// Anonymous class**

{

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

{

System.***out***.println("Hi");

**try** {

Thread.*sleep*(500);

} **catch** (InterruptedException e) {

}

}}};

Runnable obj2 = **new** Runnable()

{

**public** **void** run()**// Anonymous class**

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

{ System.***out***.println("Hello");

}}};

Thread t1 = **new** Thread(obj1);

Thread t2 = **new** Thread(obj2);

t1.start();

t2.start();

}

}

If a class extends a class it cann’t extends any other class

If a class implements Runnable it can extends any other class

If we extends Thread class all the methods inside the class get inherited which makes the program over head. So at that case we can use Runnable inter face.

Extending Thread class introduces tight coupling as the class contains code of Thread class and also the job assigned to the thread

Implementing Runnable interface introduces loose coupling as the code of Thread is separate form the job of Threads.

Runnable is a functional expression so we can use **lambda expression** here

**package** com.CISCO.rest.com.CISCO.rest;

**public** **class** thread{

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

Runnable obj1 = ()->

{

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

{

System.***out***.println("Hi");

**try** {

Thread.*sleep*(500);

} **catch** (InterruptedException e) {

}

}};

Runnable obj2 = ()->

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

{ System.***out***.println("Hello");

}};

Thread t1 = **new** Thread(obj1);

Thread t2 = **new** Thread(obj2);

t1.start();

t2.start();

}

}

T1.join() //will asking for t1 to complete its job

T2.join() will wait for a particular time till it joins with main thread

T1.isAlive() is used to check whether the thread is alieve or not

Thread t1 = **new** Thread(obj1,"Hi Thread"); //giving thread name as “**Hi Thread**”

Thread t2=**new** Thread(obj2,"Hello Thread");//giving thread name as “**HelloThread**”

t1.start();

t2.start();

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

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

Thread t1 = **new** Thread(obj1,"Hi Thread");

Thread t2 = **new** Thread(obj2,"Hello Thread");

t1.start();

t2.start();

t1.**setPriority**(Thread.***MIN\_PRIORITY***);

t2.**setPriority**(Thread.***MAX\_PRIORITY***);

t2.**setPriority**(Thread.***NORM\_PRIORITY***);

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

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

Thread.*currentThread*().getName();

**public** **class** thread{

**public** **static** **void** main(String[] args) **throws** Exception {

Runnable obj1 = **new** Hello();

Runnable obj2 = **new** Hi();

Thread t1 = **new** Thread(obj1);

Thread t2 = **new** Thread(obj2);

**t1.join();**

**t2.join(); //with join out put**

t1.start();

t2.start();

}

}

**class** Hello **implements** Runnable

{

**public** **void** run()

{

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

{

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

**try**{

Thread.*sleep*(500);}

**catch**(Exception e)

{}

}

}

}

**class** Hi **implements** Runnable

{

**public** **void** run()

{

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

{

System.***out***.println("Hi");

}

}

}

Hi

Hi

Hi

Hi

Hi

Hello

Hello

Hello

Hello

Hello

**package** com.CISCO.rest.com.CISCO.rest;

**public** **class** thread{

**public** **static** **void** main(String[] args) **throws** Exception {

Runnable obj1 = **new** Hello();

Runnable obj2 = **new** Hi();

Thread t1 = **new** Thread(obj1);

Thread t2 = **new** Thread(obj2);

t1.start();

t2.start();

**//with out join out put**

}

}

**class** Hello **implements** Runnable

{

**public** **void** run()

{

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

{

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

**try**{

Thread.*sleep*(500);}

**catch**(Exception e)

{}

}

}

}

**class** Hi **implements** Runnable

{

**public** **void** run()

{

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

{

System.***out***.println("Hi");

}

}

}

Hello

Hi

Hi

Hi

Hi

Hi

Hello

Hello

Hello

Hello

**Multi Threading**

It is a process of executing multiple threads simultaneously.

Life cycle of a thread is controlled by JVM. It has five stages **new, runnable, not-running, Terminated**

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

**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.

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

**Not-Runnable**: This is the state when the thread is still alive, but is currently not eligible to run.

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

We can define the thread by two way’s : By extending the Thread class or by implementing the runnable interface.

When we define a thread by extending thread class we have to override the run method in thread class. When we define a thread by implementing a runnable interface we have to implement only the run method in runnable interface.

**Extending Thread class**

**public** **class** testException {

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

testThread1 t1 = **new** testThread1();

testThread2 t2 = **new** testThread2(

);

Thread th1 = **new** Thread(t1);

Thread th2 = **new** Thread(t2);

th1.start();

th2.start();

}

}

**class** testThread1 **extends** Thread {

**public** **void** run() {

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

System.***out***.println("I am testThread1 class " + i);

}

}

}

**class** testThread2 **extends** Thread {

**public** **void** run() {

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

System.***out***.println("I am testThread2 class " + i);

}

}

}

**Implementing Runnable Interface**

**public** **class** testException {

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

Runnable t1 = **new** testThread1();

Runnable t2 = **new** testThread2();

Thread th1 = **new** Thread(t1);

Thread th2 = **new** Thread(t2);

th1.start(); //th1.join();

th2.start(); //th1.join();

}

}

**class** testThread1 **implements** Runnable {

**public** **void** run() {

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

System.***out***.println("I am testThread1 class " + i);

}

}

}

**class** testThread2 **implements** Runnable {

**public** **void** run() {

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

System.***out***.println("I am testThread2 class " + i);

}

}

}

th1.**setPriority**(Thread.***MIN\_PRIORITY***);

th2.**setPriority**(Thread.***MAX\_PRIORITY***);

th2.**setPriority**(Thread.***NORM\_PRIORITY***);

System.***out***.println(th1.**getName**());

System.***out***.println(th2.**getName**());

th1.isAlive()

If a class extends a class it cann’t extends any other class

If a class implements Runnable it can extends any other class

If we extends Thread class all the methods inside the class get inherited which makes the program over head. So at that case we can use Runnable inter face.

Extending Thread class introduces tight coupling as the class contains code of Thread class and also the job assigned to the thread

Implementing Runnable interface introduces loose coupling as the code of Thread is separate form the job of Threads.

The run() method comes from the Runnable interface but the start() method is only declared in the [Thread class](http://javarevisited.blogspot.com/2011/02/how-to-implement-thread-in-java.html). Since java.lang.Thread class implements Runnable interface, you can access the run() method from an instance of Thread class.

By executing start method at each time thread get started But for run method thread continuous its running

**What is Daemon thread** ?

Daemon thread is consider as secondary thread. Jre does not wait for completion of daemon thread. If main thread execution completes jre automatically terminate Daemon thread.

Eg Suppose, we make a game and there is background music in our game. The music will be played in a separate thread. Now, if the game gets over i.e. the execution of main thread finishes, then we wish to stop the music too. The JRE should not wait for the music thread to finish its execution (i.e. play the entire sound clip), so we can make the music thread as a daemon thread.

public class TestDaemonThread1 extends Thread{

public void run(){

if(Thread.currentThread().isDaemon()){//checking for daemon thread

System.out.println("daemon thread work");

}

else{

System.out.println("user thread work");

}

}

public static void main(String[] args){

TestDaemonThread1 t1=new TestDaemonThread1();//creating thread

TestDaemonThread1 t2=new TestDaemonThread1();

TestDaemonThread1 t3=new TestDaemonThread1();

t1.setDaemon(true);//now t1 is daemon thread

t1.start();//starting threads

t2.start();

t3.start(); } }

Daemon thread is always calls before main thread

**Synchronization**

Synchronized block is used to lock an object for any shared resource.Scope of synchronized block is smaller than the method

No two thread can access the object at the same time of execution

Synchronization(Object reference){

}

**What is dead lock and how to resolve a deadlock ?**

**Garbage Collection**

public class TestGarbage1{

public void finalize(){

System.out.println("object is garbage collected");

}

public static void main(String args[]){

TestGarbage1 s1=new TestGarbage1(); O/P: object is garbage collected

TestGarbage1 s2=new TestGarbage1(); object is garbage collected

s1=null;

s2=null;

**System.gc()**;

}

}

Serialization: It a process where the state of the object is converted to stream of byte code byte code. It uses markers interface to convert the object into stream of byte code.

POJO: Plain Old Java Object

In rest assured Serialization is a process of converting a java object in to a resource body

Converting back the response body into java object is known as de-serialization