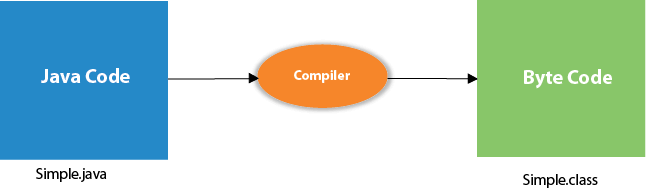
# Java

## **JDK JRE JVM:**

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.



**How to run through command prompt:**

**Javac test.java**

**Java test**

* Open a command prompt window and go to the directory where you saved the java program (test.java). Assume it's C:\.
* To Compile 🡪 **'javac test.java'** and press enter to compile your code. If there are no errors in your code, the command prompt will take you to the next line (Assumption: The path variable is set).
* To run 🡪 **' java test** ' to run your program.
* You will be able to see the result printed on the window.

JVM (Java Virtual Machine) is an abstract machine. It is called a virtual machine because it **doesn't physically exist**. It is a specification that provides a runtime environment in which Java bytecode can be executed. It can also run those programs which are written in other languages and compiled to Java bytecode.

 JVMs are available for many hardware and software platforms. **JVM, JRE, and JDK are platform dependent** because the configuration of each [OS](https://www.javatpoint.com/os-tutorial) is different from each other. However, **Java is platform independent**.

**The JVM performs the following main tasks:**

* Loads code : using **Classloader** is part of Java runtime environment, that is used to load class files(byte code) into the JVM (Java Virtual Machine).
* Verifies code :using **Bytecode Verifier**checks the code fragments for illegal code that can violate access right to objects
* Executes code : using **Interpreter** read bytecode stream then execute the instructions.
* Provides **runtime environment**.

JRE

JRE is an acronym for Java Runtime Environment

It is the implementation of JVM. It physically exists.

It is used to provide the runtime environment.

It contains a set of libraries + other files that JVM uses at runtime**.**

JDK

JDK is an acronym for Java Development Kit.

It physically exists.

It contains JRE + JVM

The JDK contains a private Java Virtual Machine (JVM) and a few other resources such as an interpreter/loader (java), a compiler (javac), an archiver (jar), a documentation generator (Javadoc), etc. to complete the development of a Java Application.

Difference between \*\*PATH\*\* and \*\*Classpath\*\* in Java?

PATH is used by the operating system while Classpath is used by JVM to locate Java binary, e.g. JAR files or Class files

what is class path in java?

C://program files/java/jdk/bin this is the java path we need to set **else "java command not recognise it will throw"**

How to set :

environment variable - system variable - edit path - add jdk bin path - Ok

## Memory management:

Whenever you trigger a java command, it allocates memory into two parts – **Stack and Heap.**Stack is used only for execution purpose. Heap is used for storage purpose.

**class** StackAndHeapMemory

{

**static** **void** methodOne()

{

System.out.println("From Method One");

methodTwo();

}

**static** **void** methodTwo()

{

System.out.println("From Method Two");

}

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

{

System.out.println("Main Method Started");

methodOne();

System.out.println("Main Method Ended");

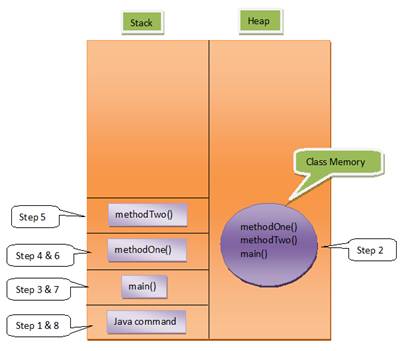
}

}

First java command enters the stack memory for execution. First it checks whether the class is loaded in heap memory or not. If it is not loaded, loading operation of class starts.

**Then** all static members are loaded into **class memory**. Then java commands calls main() method for execution.

**Step 8 :**java command also leaves stack memory and gives back the allocated memory to OS.



Summary:

1. Static var, static methods takes place in class memory

           2. Instance var, instance method takes place in heap .

           3. Stack is used only for execution purpose &  object variables

Primitive types declared locally will be on the stack

Eg:[**SIB – Static Initialization Block, Static Variables And Static Methods**](http://javaconceptoftheday.com/static-members-java/)

**class** StaticComponents

{

**static** **int** staticVariable;

**static**

{

System.out.println("StaticComponents SIB");

staticVariable = 10;

}

**static** **void** staticMethod()

{

System.out.println("From StaticMethod");

System.out.println(staticVariable);

}

}

**public** **class** MainClass

{

**static**

{

System.out.println("MainClass SIB");

}

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

{

//Static Members directly accessed with Class Name

StaticComponents.staticVariable = 20;

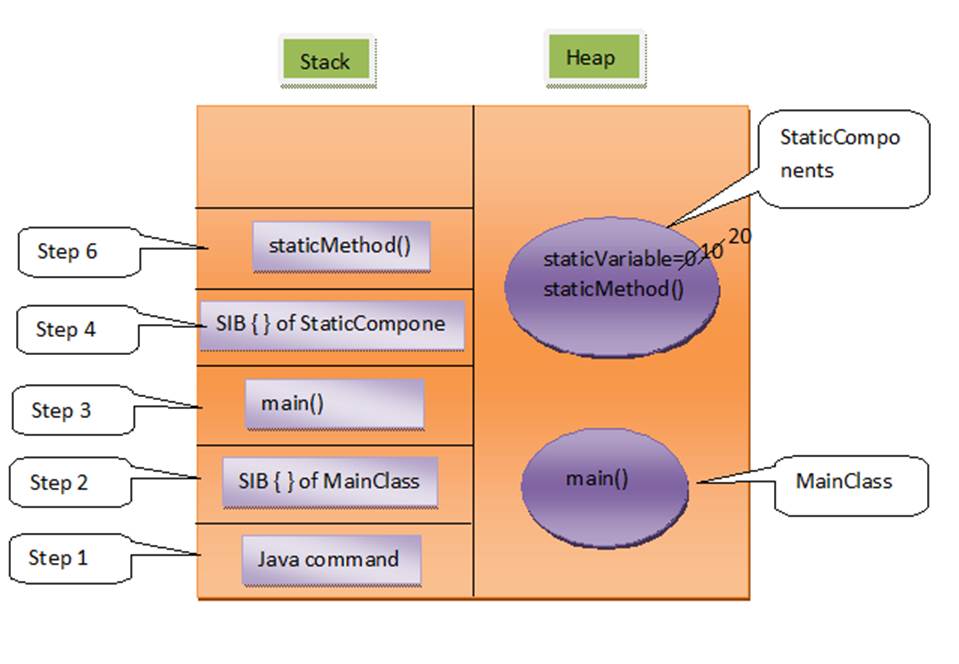
StaticComponents.staticMethod();

}

}

Output:

Main Class SIB  
StaticComponents SIB  
From StaticMethod  
20



Eg;[**Non-Static Members And Their Memory Management In Java**](http://javaconceptoftheday.com/non-static-members-in-java/)

**class** A

{

**int** nonStaticVariable;

**static** **int** staticVariable;

**static** **void** staticMethod()

{

System.out.println(staticVariable);

// System.out.println(nonStaticVariable);

}

**void** nonStaticMethod()

{

System.out.println(staticVariable);

System.out.println(nonStaticVariable);

}

}

**class** MainClass

{

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

{

A.staticVariable = 10;

// A.nonStaticVariable = 10;

A.staticMethod();

// A.nonStaticMethod();

A a1 = **new** A();

A a2 = **new** A();

System.out.println(a1.nonStaticVariable);

System.out.println(a1.staticVariable);

a1.nonStaticMethod();

a1.staticMethod();

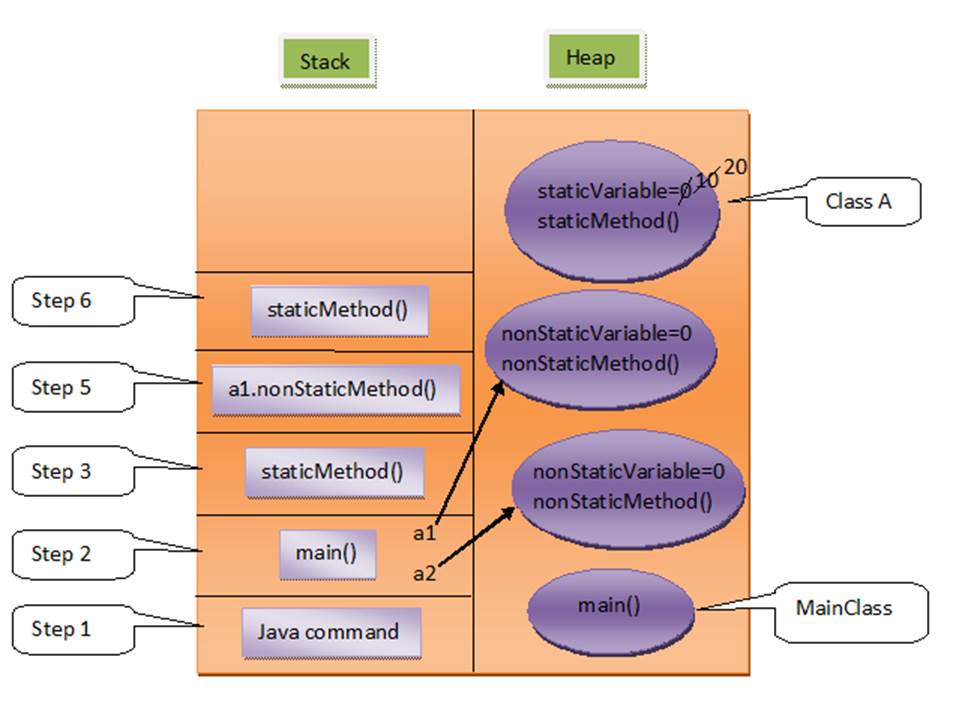
System.out.println(a2.staticVariable);

a1.staticVariable = 20;

System.out.println(a2.staticVariable);

}

}



 memory management

                                Inheritance

                                access modifiers

                                ploymorphism: overload n override

                                Type casting

                                nested classes

                                exception handling:

                                Arrays

                                Collection

**Primitive data types** : boolean, byte, short, int , long , float , double, char

**Non primitive type** : String , Array etc

 1. what are main pillars of oops?

     Encapsulation.

Abstraction.

Inheritance.

Polymorphism.

## **Access Modifier**

1). Private

**Usage of Private members :**

Private members of a class whether it is a field or method or constructor can not be accessed outside the class.

**Inheritance of Private Members :**

Private members will not be inherited to sub class.

**Important Note :**

1). Class can not be a private except inner classes. Inner classes are nothing but again members of outer class. So members of a class (field, method, constructor and inner class) can be private but not the class itself.

2). We can’t create sub classes to that class which has only private constructors.

**class** A

{

**private** **int** i;

**private** **void** methodOfClassA()

{

//Private Method

System.out.println(i); //Private field can be used within class

B b = **new** B(); //Private inner class can be used within class

}

**private** **class** B

{

//Private Inner Class

}

}

**class** C **extends** A

{

**void** methodOfClassC()

{

//System.out.println(i); Private member can not be inherited

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

//System.out.println(a.i); Private field can not be used outside the class

//a.methodOfClassA(); Private method can not be used outside the class

//A.B b = new A.B(); Private inner class can not be used outside the class

}

}

**private** **class** A

{

//Outer class Can not be private

}

**class** A

{

**private** A()

{

//Private Constructor

}

**private** A(**int** i)

{

//Private constructor

}

}

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

{

//Can't create subclass to the class

//which has only private constructors

}

**2). Default or Package or No-Access Modifiers**

**Usage of Default members :**

Default members or members with No-Access modifiers are accessed or visible within the package only. It applies to outer classes also.

**Inheritance Of Default Members :**

Default members can be inherited to sub classes within package.

**package** pack1;

**class** A

{

**int** i;

A()

{

//Constructor with default modifier

}

**void** methodOfClassA()

{

//Method with default access modifier

System.out.println(i);

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

}

**class** B

{

//Inner Class with default access modifier

}

}

**class** C **extends** A

{

**void** methodOfClassC()

{

System.out.println(i); //Default field can be inherited within package

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

System.out.println(a.i); //Default field can be used within the package

a.methodOfClassA(); //Default method can be used within the package

A.B b = **new** A.B(); //Default inner class can be used within the package

}

}

**package** pack2;

//import pack1.A; Class A with default access modifier not visible outside the package

/\*class D extends A Default Class can not have sub class outside the package

{

void methodOfClassD()

{

System.out.println(i); Default field can not be inherited outside package

A a = new A(); Can't use constructor with default access modifier outside the package

System.out.println(a.i); Default field can not be used outside the package

a.methodOfClassA(); Default method can not be used outside the package

A.B b = new A.B(); Default inner class can not be used outside the package

}

}\*/

**3). Protected**

**Usage of Protected Member :**

Protected member can be used within the package only.

**Inheritance Of Protected Member :**

Protected Member can be inherited to any sub classes.

**package** pack1;

**public** **class** A

{

**protected** **int** i;

**protected** **void** methodOfClassA()

{

//Protected method

System.out.println(i); //Protected field can be used within class

B b = **new** B(); //Protected Inner Class can be used within class.

}

**protected** **class** B

{

//Protected Inner Class

}

}

**class** C **extends** A

{

**void** methodOfClassC()

{

System.out.println(i); //Protected field can be inherited to any sub class

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

System.out.println(a.i); //Protected field can be used within the package

a.methodOfClassA(); //Protected method can be used within the package

A.B b = **new** A.B(); //Protected Inner Class can be used within the package

}

}

**package** pack2;

**import** pack1.A;

**class** D **extends** A

{

**void** methodOfClassD()

{

System.out.println(i); //Protected field can be inherited to any sub class

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

//System.out.println(a.i); Protected field can not be used outside the package

//a.methodOfClassA(); Protected method can not be used outside the package

//A.B b = new A.B(); Protected inner class can not be used outside the package

}

}

**Important Note :**

1). Outer class can not be protected.

2). We can create sub classes to a class which has only protected constructors but we can’t create objects to that class outside the package.

**4). Public**

**Usage of Public members :**

Public members can be used anywhere.

**Inheritance Of Public Members :**

Public members can be inherited to any sub class.

**package** pack1;

**public** **class** A

{

**public** **int** i;

**public** **void** methodOfClassA()

{

//public method

System.out.println(i); //public field can be used anywhere

B b = **new** B(); //public Inner Class can be used anywhere.

}

**public** **class** B

{

//public Inner Class

}

}

**class** C **extends** A

{

**void** methodOfClassC()

{

System.out.println(i); //public field can be inherited to any sub class

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

System.out.println(a.i); //public field can be used anywhere

a.methodOfClassA(); //public method can be used anywhere

A.B b = **new** A.B(); //public Inner Class can be used anywhere.

}

}

**package** pack2;

**import** pack1.A;

**class** D **extends** A

{

**void** methodOfClassD()

{

System.out.println(i); //public field can be inherited to any sub class

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

System.out.println(a.i); //Public field can be used anywhere

a.methodOfClassA(); //Public method can be used anywhere

A.B b = **new** A.B(); //Public inner class can be used anywhere

}

}

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| |  |  |  |  |  | | --- | --- | --- | --- | --- | |  | **Within class** | **Within pkg** | **Outside pkg by subclass** | **Outside pkg** | | **private** | Y | N | N | N | | **Default** | Y | Y | N | N | | **Protected** | Y | Y | Y | N | | **public** | Y | Y | Y | Y | |  |  |

## **overloading, overiding and inheritance program stategy**

|  |  |  |
| --- | --- | --- |
|  | Method overloading | Method overriding |
| 1 | **Def**: The class which is having multiple methods with same name but different method arguments | **Def:** |
| 2 |  |  |
| 3 |  |  |
| 4 |  |  |

a) Name of the method must be same as that of super class method.

b) Return type of overridden method must be compatible with the method being overridden. i.e if a method has primitive type as it’s return type then it must be overridden with primitive type only and if a method has derived type as it’s return type then it must be overridden with same type or it’s sub class types.

c) You must not reduce the visibility of a method while overriding.

Eg: in super calss if it is **protected** in sub class it can **public** but not declare with private

d) You must not change parameter list of a method while overriding.

e) You can not increase the **scope of exceptions** while overriding a method with throws clause.

f) **Constructors** and **initializers**(Static initializers and instance initializers) are not inherited to sub classes

**What is typecasting ? what is upcasting and downcasting ?**

A process of converting one data type to another is known as Typecasting

**Upcasting:** Upcasting is the [typecasting](https://www.geeksforgeeks.org/type-conversion-java-examples/) **of a child object to a parent object**. Upcasting can be done implicitly.

// Upcasting

Animal a=**new** Dog();

**Downcasting:** Similarly, downcasting means the typecasting of a **parent object to a child object**. Downcasting cannot be implicit.

// Trying to Downcasting Implicitly

// Dog d = new Animal(); - > compile time error

// Downcasting Explicitly

    Dog d = (Dog)a;

**Inheritance program stategy:**

**public** **class** Animal {

**static** **int** *i*=3;

**public** **void** m1(){

            System.***out***.println("animal method");

      }

**static** **void** staticMethod(){

            System.***out***.println("Animal static method");

      }

}

+++++++++++++++++++++++++

**public** **class** Dog **extends** Animal{

      @Override

**public** **void** m1() {

            System.***out***.println("Dog method");

      }

}

++++++++++++++++++++++++

**public** **class** Cat **extends** Animal {

      @Override

**public** **void** m1() {

            System.***out***.println("cat method");

      }

}

+++++++++++++++++++++++

/\* \*\*\*\*\*\* scenario 1 \*\*\*\*\*\*\*\*\*\*\*\*/

Animal a=**new** Animal();

a.m1();

Dog d=**new** Dog();

d.m1();

Cat c=**new** Cat();

c.m1(); \*/

**/\* output: ==============\*/**

**/\*    Animal method**

**Dog method**

**cat method\*/**

/\*/\*\*\*\*\*\*\* scenario 2 : upcasting \*\*\*\*\*\*\*\*\*\*\*\*/

Animal a2=**new** Dog();

a2.m1();

a2=**new** Cat();

a2.m1();

**/\* output:==============\*/**

**Dog method**

**cat method  \*/**

/\*/\*\*\*\*\*\*\* scenario 3 : downcasting compileTime Error \*\*\*\*\*\*\*\*\*\*\*\*/

Dog d2=**new** Animal(); // Type mismatch: cannot convert from Animal to Dog.

Dog d3=(Animal)**new** Dog(); // Type mismatch: cannot convert from Animal to Dog.

/\*\*\*\*\*\*\*\* scenario 4 : downcasting Runtime Exception \*\*\*\*/

Dog d2=(Dog) **new** Animal();

//Exception in thread "main" java.lang.ClassCastException: Animal cannot be cast to Dog

Animal a3=(Dog)**new** Animal();

//Exception in thread "main" java.lang.ClassCastException: Animal cannot be cast to Dog

/\* This is fine: \*/

Animal a2=(Animal)**new** Dog();

a2.m1();

**output: Dog method**

Dog d3=**new** Cat() //compileTime Error :: Type mismatch: cannot convert from Cat to Dog

Animal a3=(Cat)**new** Dog(); //compileTime Error :: Cannot cast from Dog to Cat

/\* \*\*\*\*\*\*\* scenario 6 : we can inherit static method in subclass \*/

Dog d=**new** Dog();

d.staticMethod();

Animal.staticMethod();

Dog.staticMethod();

**/\*output :**

**Animal static method**

**Animal static method**

**Animal static method\*/**

**System.out.println(Dog.i);          //output :: 3  //you can inherit static var as well**

We can inherit static var and static methods:

**Method overloading:**

**Compiler checks only method signature for overloading of methods** not an accees modifier, not an return type, not static or non static

2) What is method signature? What are the things it consist of?

Method signature is used by the **compiler** to differentiate the methods. Method signature consist of three things.

a) Method name

b) Number of arguments

c) Types of arguments

3) Can we declare one overloaded method as static and another one as non-static?

**Yes.** Overloaded methods can be either static or non static.

5) Is it possible to have two methods in a class with same method signature but different return types?

No, compiler will give duplicate method error. Compiler checks only method signature for duplication not the return types. If two methods have same method signature, straight away it gives compile time error.

6) In “MyClass” , there is a method called “myMethod” with four different overloaded forms. All four different forms have different visibility ( private, protected, public and default). Is “myMethod” properly overloaded?

Yes. **Compiler checks only method signature for overloading of methods** not the visibility of methods.

8) Can we overload main() method?

**Yes,** we can overload main() method. A class can have any number of main() methods but execution starts from **public static void main(String[] args)**only.

9) Can we declare overloaded methods as final?

**Yes,** we can declare overloaded methods as final.

11) Overloading is the best example of dynamic binding. True or false?

**False**. Overloading is the best example for static binding.

12) Can overloaded method be overrided?

**Yes**, we can override a method which is overloaded in super class.

Method overriding:

3) Can we override static methods?

**No**, Static methods can not be overridden. If we try to override them they will be hidden in the sub class.

4) What happens if we change the arguments of overriding method?

If we change the arguments of overriding method, then that method will be treated as overloaded not overridden.

5) Can we override protected method of super class as public method in the sub class?

**Yes**. You can increase the visibility of overriding methods but can’t reduce it.

6) Can we change the return type of overriding method from Number type to Integer type?

**Yes**. You can change as Integer is a sub class of Number type.

7) Can we override a super class method without throws clause as a method with throws clause in the sub class?

**Yes**, but only with unchecked type of exceptions.

8) Can we change an exception of a method with throws clause from SQLException to NumberFormatException while overriding it?

**Yes**. Overridden method may throw SQLException or it’s sub class exception or any unchecked type of exceptions.

9) Can we change an exception of a method with throws clause from unchecked to checked while overriding it?

**No**. We can’t change an exception of a method with throws clause from unchecked to checked.

10) How do you refer super class version of overridden method in the sub class?

Using **super** keyword, we can

11) Can we override private methods?

**No**, They are not at all inherited to sub class.

12) Can we remove throws clause of a method while overriding it?

**Yes**. You can remove throws clause of a method while overriding it.

13) Is it possible to override non-static methods as static?

**No**. You can’t override non-static methods as static.

14) Can we change an exception of a method with throws clause from checked to unchecked while overriding it?

**Yes**. We can change an exception from checked to unchecked but reverse is not possible.

15) Can we change the number of exceptions thrown by a method with throws clause while overriding it?

Yes, we can change. But, exceptions must be compatible with throws clause in the super class method.

**Inheritance:**

1) What do you mean by inheritance.?

2) What are the types of inheritance.?

3) Can a class extend more than one classes or does java support multiple inheritance? If not, why?  //No

4) How do you implement multiple inheritance in java?

5) You know that all classes in java are inherited from java.lang.Object class. Are interfaces also inherited from Object class.?

No, only classes in java are inherited from Object class. Interfaces in java are not inherited from Object class. But, classes which implement interfaces are inherited from Object class.

6) How do you restrict a member of a class from inheriting to it’s sub classes.?

By declaring that member as a private. Because, private members are not inherited to sub classes.

7) Can a class extend itself.?

No, A class can not extend itself.

8) Are constructors and initializers also inherited to sub classes.?

**No**, Constructors and initializers(Static initializers and instance initializers) are not inherited to sub classes. But, they are executed while instantiating a sub class.

9) What happens if both, super class and sub class, have a field with same name.?

Super class field will be hidden in the sub class. You can access hidden super class field in sub class using **super** keyword.

10) Are static members inherited to sub classes?

**Yes**, , Static members are also inherited to sub classes.

**class** A

{

**static** **int** *i* = 10;

**static** **void** method()

    {

        System.***out***.println("Static Method");

    }

}

**class** B **extends** A

{

}

**public** **class** StaticInitializers

{

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

    {

        B.*method*();       //Calling inherited static method

        System.***out***.println(B.*i*);    //printing inherited static field.

    }

}

**Why multiple inheritance not possible with  java/abstaraction?**

1. because we can’t extend more then one class as a java rule.
2. To keep java as simple and to avoid ambiguity and Dimond prob.

**What is Diamond Problem in java/inheritance?**

consider a class A has foo() method and then B and C derived from A and has there own foo() implementation and now class D derive from B and C using multiple [inheritance](http://javarevisited.blogspot.com/2012/10/what-is-inheritance-in-java-and-oops-programming.html) and if we refer just foo() compiler will not be able to decide which foo() it should invoke.   
  
This is also called Diamond problem because structure on this inheritance scenario is similar to 4 edge diamond, see below

           A foo()

           / \

          /   \

   foo() B     C foo()

          \   /

           \ /

            D

           foo()

In my opinion even if we remove the top head of diamond class A and allow multiple inheritances we will see this problem of ambiguity.

**We cant override the data member**

Eg:

**package** multiple\_inheritance;

**public** **class** Vehicle {

**int** speedLimit = 60;

}

**package** multiple\_inheritance;

**public** **class** Bike **extends** Vehicle {

**int** speedLimit = 80;

}

**package** multiple\_inheritance;

**public** **class** Test {

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

Vehicle v= **new** Bike();

System.***out***.println(v.speedLimit); //60

}

}

+++++++++++++++++++++++++++++++++

## **Polymorphism**

In generalpolymorphism means executing single action in multiple forms.

 Any entity which shows polymorphism during compile time is called **static polymorphism**. Because, they show polymorphism during compilation.

[**method overloading**](http://javaconceptoftheday.com/method-overloading-in-java/) are best examples of static polymorphism

Eg: using method overload:

 The compiler looks at the method signature and decides which method to invoke for a particular method call at compile time.

**class** DemoOverload{

**public** **int** add(**int** x, **int** y){  //method 1

**return** x+y;

    }

**public** **int** add(**int** x, **int** y, **int** z){ //method 2

**return** x+y+z;

    }

**public** **int** add(**double** x, **int** y){ //method 3

**return** (**int**)x+y;

    }

**public** **int** add(**int** x, **double** y){ //method 4

**return** x+(**int**)y;

    }

}

**class** Test{

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

    DemoOverload demo=**new** DemoOverload();

    System.***out***.println(demo.add(2,3));      //method 1 called

    System.***out***.println(demo.add(2,3,4));    //method 2 called

    System.***out***.println(demo.add(2,3.4));    //method 4 called

    System.***out***.println(demo.add(2.5,3));    //method 3 called

    }

}

Any entity which shows polymorphism during run time is called **dynamic polymorphism/run time polymorphism**. because type of the object used will be determined at run time only.

[**Method Overriding**](http://javaconceptoftheday.com/method-overriding-java/) is the best example of dynamic

Eg:

**class** SuperClass

{

**void** methodOfSuperClass()

{

System.out.println("From Super Class");

}

}

**class** SubClass **extends** SuperClass

{

//Super Class Method Overrided

**void** methodOfSuperClass()

{

System.out.println("From Sub Class");

}

}

**public** **class** Test

{

**static** **void** util(SuperClass superclass)

{

superclass.methodOfSuperClass();

//For each execution of this method, different objects will be passed to it.

//which Object will be used is determined during run time only.

//This shows dynamic polymorphism.

}

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

{

SuperClass superclass1 = **new** SuperClass();

SubClass subclass = **new** SubClass();

SuperClass superclass2 = **new** SubClass();

util(superclass1); //SuperClass object is passes to util() // From Super Class

util(subclass); //SubClass object is passed to util() // From Sub Class

util(superclass2); //SubClass object is passed to util() // From Sub Class

}

}

Output:

/\*

From Super Class

From Sub Class

From Sub Class

\*/

## **this / super / final keyword**

Def: this is used to refer current object.

It is used in constructor , constructor chaining.

In constructor chaining this keyword should be always 1st statement.

**Without this:**

**============**

**public** **class** Employee {

**int** id;

String name;

**public** Employee(**int** id, String name) {

id = id;

name = name;

}

**public** **void** display()

{

System.***out***.println(id+" "+name);

}

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

Employee e = **new** Employee(1, "ram");

e.display();

}

}

Output : 0, null

But **below will work without this:**

**============================**

**public** **class** Employee {

**int** id;

String name;

**public** Employee(**int** rollNum, String firstName) {

id = rollNum;

name = firstName;

}

**public** **void** display()

{

System.***out***.println(id+" "+name);

}

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

Employee e = **new** Employee(1, "ram");

e.display();

}

}

// 1 ram

**With this()**

**============**

**public** **class** Employee {

**int** id;

String name;

**public** Employee(**int** id, String name) {

**this**.id = id;

**this**.name = name;

}

**public** **void** display()

{

System.***out***.println(id+" "+name);

}

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

Employee e = **new** Employee(1, "ram");

e.display();

}

}

**Constructor chaining with this keyword**

**public** **class** Employee {

**int** id;

String name;

String company;

**public** Employee(**int** id, String name) {

**this**.id = id;

**this**.name = name;

}

**public** Employee(**int** id, String name, String company) {

**this**(id,name);

**this**.company = company;

}

**public** **void** display()

{

System.***out***.println(id+" "+name +" "+company);

}

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

Employee e1 = **new** Employee(1, "ram");

Employee e2 = **new** Employee(1, "ram", "hcl");

e1.display();

e2.display();

}

}

**Super() keyword**

**==============**

It is used to refer immediate parent class object

* 1. Used to refer parent calss **instance var**
  2. Used to refer parent calss **method**
  3. Used to refer parent calss **constructor**

**Eg 1: calling parent class instance var and methods using super in subclass**

**public** **class** Vehicle {

**int** speed =50;

**public** **void** msg()

{

System.***out***.println("vehicle class");

}

}

**public** **class** Bike **extends** Vehicle {

**int** speed = 100;

**public** **void** display()

{

**super.msg();**

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

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

}

}

**public** **class** Test

{

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

Bike b = **new** Bike();

b.display();

}

}

vehicle class

50

Java

**Eg 2: whenever parent class has default constructor, it calls implicitly in subclass**

**public** **class** Vehicle {

Vehicle()

{

System.***out***.println("default constructor");

}

}

**public** **class** Bike **extends** Vehicle {

}

**public** **class** Test {

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

Bike b = **new** Bike();

}

}

//default constructor

**Another example**

**public** **class** Vehicle {

Vehicle()

{

System.***out***.println("default constructor");

}

}

**public** **class** Bike **extends** Vehicle {

Bike()

{

System.***out***.println("Bike constructor");

}

}

**public** **class** Test {

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

Bike b = **new** Bike();

}

}

// default constructor

Bike constructor

**Eg 3: if parent class does not have default constructor, then its mandatory to subclass to declare constructor else compile time error.**

**public** **class** Vehicle {

Vehicle(String s)

{

System.***out***.println("vehicle constructor " + s);

}

}

**public** **class** Bike **extends** Vehicle {

//compiler error

}

**SO:**

**public** **class** Bike **extends** Vehicle {

Bike(String s) {

**super**(s);

}

}

**public** **class** Test {

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

Bike b = **new** Bike("honda");

}

}

// vehicle constructor honda

**Eg 4: if parent class does not have default constructor, and subclass also some explicit constructor, then use super as 1st statement in child constructor and also parent class constructor fields.**

**public** **class** Vehicle {

Vehicle(String s)

{

System.***out***.println("vehicle constructor " + s);

}

}

**public** **class** Bike **extends** Vehicle {

Bike(String s, **int** a) {

**super**(s);

System.***out***.println("bike constructor "+a);

}

}

**public** **class** Test {

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

Bike b = **new** Bike("honda", 100);

}

}

vehicle constructor honda

bike constructor 100

**Final keyword**

**==============**

* 1. Stop value change for variable
  2. Stop method overriding for method (but can inherit)
  3. Stop extending of class

Eg:

**What are static initializers and when would you use them?**

Answer:

A static initializer gives you the opportunity to run code during the initial loading of a class and it guarantees that this code will only run once and will finish running before your class can be accessed in any way.

They are useful for performing initialization of complex static objects

## **Abstract(8) and Interface(4)**

1. You can’t create objects to abstract class even though it does not contain any abstract methods. i.e Abstract class never instantiated.

2. If class extending abstract class it should implement that method else it should be declared itself as a abstract class.

3. It is not compulsory that abstract class must have abstract methods. It may or may not have abstract methods.

4. You **can’t** refer abstract class to concrete class.

AbstractClass  a=new ConcreteClass();           //compileTime Error

5. Inside abstract class, we can keep any number of constructors. If you are not keeping any constructors, then compiler will keep default constructor

1. **Abstract class** and methods should **not** declare with **final** keyword since Abstract class needs to extend and its methods needs to implement.
2. Abstract methods can not be private.

(Because, abstract methods must be implemented somehow in the sub classes. If you declare them as private, then you can’t use them outside the class.)

**abstract** **class** AbstractClass

{

**private** **abstract** **void** abstractMethodOne();

//Compile time error, abstract method can not be private.

}

1. Constructors and fields can not be declared as abstract.

**abstract** **class** AbstractClass

{

**abstract** **int** i;

//Compile time error, field can not be abstract

**abstract** AbstractClass()

{

//Compile time error, constructor can not be abstract

}

}

1. Abstract methods can not be static.

**abstract** **class** AbstractClass

{

**static** **abstract** **void** abstractMethod();

//Compile time error, abstract methods can not be static

}

/////////////// //////  Interface  ///////////////////////////////////////////

* Interface can have two types of members.

 1) Fields     2) Abstract Methods.

* By default, All methods of an interface are public and abstract. Interfaces should not contain a single concrete method.
* By default, Every field of an interface is public, static and final

**interface** InterfaceClass

{

**void** abstractMethodOne(); //Abstract method

**void** abstractMethodTwo(); //Abstract Method

//No need to use abstract keyword,

//by default methods are public and abstract

**int** i = 0;

//By default, field is public, static and final

//Following statements give compile time errors

**private** **double** d = 10;

**protected** **long** l = 15;

//You can't use any other modifiers other than public, static and final

}

o   You can’t change the value of a field once they are initialized. Because they are static and final.

**interface** InterfaceClass

{

**int** i = 0;

}

**class** AnyClass **implements** InterfaceClass

{

**void** methodOne()

{

//Following statement gives compile time error.

InterfaceClass.i = 10;

//final field can not be re-assigned

}

}

* While implementing any interface methods inside a class, that method must be declared as public. Because, **according to**[**method overriding**](https://javaconceptoftheday.com/method-overriding-java/)**rule, you can’t reduce visibility of super class method.** By default, every member of an interface is public and while implementing you should not reduce this visibility.

**interface** InterfaceClass

{

**void** methodOne();

}

**class** AnyClass **implements** InterfaceClass

{

**void** methodOne()

{

//It gives compile time error.

//Interface methods must be implemented as public

}

}

o   by default interface itself is abstract like below,

**abstract** **interface** InterfaceClass

{

//By default interface is abstract

//No need to use abstract keyword

}

* [SIB](https://javaconceptoftheday.com/static-members-java/) – Static Initialization Block and [IIB](https://javaconceptoftheday.com/instance-initialization-block-in-java/) – Instance Initialization Block are not allowed in interfaces.

**interface** InterfaceClassOne

{

**static**

{

//compile time error

//SIB's are not allowed in interfaces

}

{

//Here also compile time error.

//IIB's are not allowed in interfaces

}

**void** methodOne(); //abstract method

}

**Eg 1: implementing interface in class:**

**package** multiple\_inheritance;

**public** **interface** Printable {

**void** print();

}

**package** multiple\_inheritance;

**public** **interface** Showable {

**void** show();

}

**package** multiple\_inheritance;

**public** **class** Test **implements** Printable, Showable {

**public** **void** print()

{

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

}

**public** **void** show()

{

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

}

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

Test t = **new** Test();

t.print(); // printing

t.show(); // showing

}

}

**Eg 2: If two interface has same method:**

**package** multiple\_inheritance;

**public** **interface** Printable {

**void** print();

}

**package** multiple\_inheritance;

**public** **interface** Showable {

**void** print();

}

**package** multiple\_inheritance;

**public** **class** Test **implements** Printable, Showable {

**public** **void** print()

{

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

}

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

Test t = **new** Test();

t.print(); // printing

}

}

Conclusion : printable and showable have same method but its implementation is provided by class, So there is no ambiguity.

**Eg 3 : extending one interface by another**

**package** multiple\_inheritance;

**public** **interface** Printable {

**void** print();

}

**package** multiple\_inheritance;

**public** **interface** Showable **extends** Printable{

**void** show();

}

**package** multiple\_inheritance;

**public** **class** Test **implements** Showable {

**public** **void** print()

{

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

}

**public** **void** show()

{

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

}

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

Test t = **new** Test();

t.print(); // printing

t.show(); // showing

}

}

## [**Type Casting In Java**](https://javaconceptoftheday.com/type-casting-in-java/)

There are two types of casting,

1) Primitive Casting.

2) Derived Casting

There are two types in primitive casting. **1) Auto Widening   2) Explicit Narrowing**

**1) Auto Widening**

When you are converting data from small sized data type to big sized data type, i.e when you are converting data from left-placed data type to right-placed data type in the above order, auto widening will be used. For example, when you are converting byte to short or short to int, auto widening will be used.

**class** AutoWidening

{

**static** **float** methodOne(**int** i)

{

**long** j = i; //int is auto widened to long

**return** j; //long is auto widened to float

}

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

{

**byte** b = 10;

**short** s = b; //byte is auto widened to short

**double** d = *methodOne*(s); //short is auto widened to int and float to double

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

}

}

**2) Explicit Narrowing**

When you are converting data from big sized data type to small sized data type, i.e when you are converting data from right-placed data type to left-placed data type in the above order, explicit narrowing will be used. For example, when you are converting double to float or float to int, explicit narrowing will be used.

**class** ExplicitlyNarrowing

{

**static** **short** methodOne(**long** l)

{

**int** i = (**int**) l; //long is explicitly narrowed to int

**return** (**short**)i; //int is explicitly narrowed to short

}

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

{

**double** d = 10.25;

**float** f = (**float**) d; //double is explicitly narrowed to float

**byte** b = (**byte**) *methodOne*((**long**) f); //float is explicitly narrowed to long and short to byte

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

}

}

**2) Derived Casting**

Derived casting is used to change the type of object from one user defined data type to another user defined data type in the class hierarchy.

There are two types in derived casting. **1) Auto-up Casting  2) Explicit Down Casting.**

**1) Auto-Up Casting**

Auto-Up Casting is used to change the type of object from sub class type to super class type. i.e an object of sub class type is automatically converted to an object of super class type. For example,

**class** A

{

**int** i = 10;

}

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

{

**int** j = 20;

}

**class** C **extends** B

{

**int** k = 30;

}

**class** D **extends** C

{

**int** m = 40;

}

**public** **class** AutoUpCasting

{

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

{

D d = **new** D();

C c = d; // D type object is Auto-Up Casted to C type

B b = d; // D type object is Auto-Up Casted to B type

C c1 = **new** C();

A a = c1; // C type object is Auto-Up Casted to A type

A a1 = **new** B(); // B type object is Auto-Up Casted to A type

}

}

**2) Explicit Down Casting**

Explicit down Casting is used to change the type of object from super class type to sub class type. i.e you have to explicitly convert an object of super class type to an object of sub class type. For example,

**class** A

{

**int** i = 10;

}

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

{

**int** j = 20;

}

**class** C **extends** B

{

**int** k = 30;

}

**class** D **extends** C

{

**int** m = 40;

}

**public** **class** ExplicitDownCasting

{

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

{

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

B b = (B) a; //A type is explicitly downcasted to B type

C c = (C) a; //A type is explicitly downcasted to C type

D d = (D) a; //A type is explicitly downcasted to D type

B b1 = **new** B();

D d1 = (D) b1; //B type is explicitly downcasted to D type

d1 = (D) **new** C(); //C type is explicitly downcasted to D type

}

}

**6) What is ClassCastException.?**

ClassCastException is an exception which occurs at run time when an object of one type can not be casted to another type. (Click [here](https://javaconceptoftheday.com/classcastexception-in-java/) to see more on ClassCastException)

**7) How do you avoid ClassCastException in your code.?**

**By using generics**, you can avoid ClassCastException. (Click [here](https://javaconceptoftheday.com/why-we-need-generics-in-java/) to see how you can avoid ClassCastException Using generics.)

**8) What is boxing and unboxing.?**

Wrapping of primitive content into corresponding wrapper class object is called boxing. Unwrapping the wrapper class object into corresponding primitive content is called unboxing.

## [**Nested Classes In Java**](https://javaconceptoftheday.com/nested-classes-in-java/)

There are 2 types of Nested Classes.

1. Static Nested Classes (7)
2. Non-Static Nested Classes or Inner Classes

### **Static Nested Classes In Java :**

1. If nested class is declared as static, then that nested class is called as **static nested class**.

**class** OuterClass

{

**int** i; //Field as a member

**void** methodOne()

{

//method as a member

}

**static** **class** NestedClass

{

//class as a member which is declared as static

}

}

1. Static nested classes can contain **both** static and non-static members.

**class** OuterClass

{

//Some members of OuterClass

**static** **class** NestedClass

{

**static** **int** *i*; //Static Field

**int** j; //Non-static Field

**void** methodOne()

{

//Non-static method

}

**static** **void** methodTwo()

{

//Static Method

}

}

}

1. **We can access only static members** of outer class inside a static nested class. We can’t access non-static members of outer class inside a static nested class.

|  |
| --- |
|  |

**class** OuterClass

{

**static** **int** *i*; //static field of OuterClass

**int** j; //Non-static field of OuterClass

**void** methodOne()

{

//Non-static method of OuterClass

}

**static** **void** methodTwo()

{

//static method of OuterClass

}

**static** **class** NestedClass

{

**void** methodOfInnerClass()

{

System.***out***.println(*i*); //static field can be accessed

System.***out***.println(j); //This gives Compile time error

*methodTwo*(); //can access static method

methodOne(); //This gives Compile time error

}

}

}

1. We have seen that **static methods can’t be abstract but static nested classes can be abstract.**

**class** OuterClass

{

//static and abstract inner class

**abstract** **static** **class** NestedClass

{

**abstract** **void** methodOne(); //abstract method of NestedClass

**void** methodTwo()

{

//concrete method of NestedClass

}

}

}

1. **Static nested class can be final.**

**class** OuterClass

{

//final and static nested class

**final** **static** **class** NestedClass

{

**void** methodOne()

{

//concrete method of NestedClass

}

}

}

1. Below example shows how to refer Objects of the static nested class.

**class** OuterClass

{

**int** i = 10; //Non-static Field of OuterClass

**static** **void** methodOne()

{

System.***out***.println("Static method of OuterClass");

}

**static** **class** NestedClassOne

{

**int** i = 20; //Non-static Field of NestedClassOne

**static** **void** methodOne()

{

System.***out***.println("Static method of NestedClassOne");

}

}

**static** **class** NestedClassTwo

{

**int** i = 30; //Non-static Field of NestedClassTwo

**static** **void** methodOne()

{

System.***out***.println("static method of NestedClassTwo");

}

}

}

**public** **class** NestedClasses

{

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

{

OuterClass.*methodOne*(); //static member can be accessed directly through class name.

OuterClass outer = **new** OuterClass();

System.***out***.println(outer.i); //Instance member must be accessed through object reference

OuterClass.NestedClassOne.*methodOne*(); //static member can be accessed directly through class name.

OuterClass.NestedClassOne nestedOne = **new** OuterClass.NestedClassOne();

System.***out***.println(nestedOne.i); //Instance member must be accessed through object reference

OuterClass.NestedClassTwo.*methodOne*(); //static member can be accessed directly through class name.

OuterClass.NestedClassTwo nestedTwo = **new** OuterClass.NestedClassTwo();

System.***out***.println(nestedTwo.i); //Instance member must be accessed through object reference

}

}

* Constructors and methods of nested classes can be overloaded.

**class** OuterClass

{

**static** **class** NestedClass

{

NestedClass()

{

//First constructor

}

NestedClass(**int** i)

{

//Second Constructor

}

NestedClass(**int** i, **int** j)

{

//Third Constructor

}

**void** methodOne()

{

//Overloaded method

}

**void** methodOne(**int** i)

{

//Overloaded method

}

**void** methodOne(**int** i, **int** j)

{

//Overloaded method

}

}

}

1. Static Nested Classes can be chained. i.e Nested class may contain another nested class and that nested class may contain another nested class and so on.

**class** OuterClass

{

**static** **class** NestedClass

{

**static** **class** NestedClassOne

{

**static** **class** NestedClassTwo

{

**static** **class** NestedclassThree

{

**static** **void** methodOne()

{

System.***out***.println("Chain Of Nested Classes");

}

}

}

}

}

}

**public** **class** NestedClasses

{

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

{

OuterClass.NestedClass.NestedClassOne.NestedClassTwo.NestedclassThree.*methodOne*();

}

}

### **Non-Static Nested Classes In Java :**

Nested classes which are declared as non-static or nested classes which can be accessed only though instantiating it’s outer class are called non-static nested classes. Non-static nested classes are also called as **Inner Classes.**

They are 3 types of Inner Classes in java.

1. Member Inner Classes
2. Local Inner Classes
3. Anonymous Inner classes

**Member Inner Classes :**

Member Inner Classes are non-static nested classes which are declared as non-static members of outer class.

* Member inner classes must contain only non-static members. Static members are not allowed inside member inner classes.

**class** OuterClass

{

//Member Inner Class : Class As a Non-Static Member

**class** InnerClass

{

**int** i; //can contain non-static field

**static** **int** *j* = 10; //It gives compile time error

//Should not contain static field

**void** methodOne()

{

//can have non-static method

}

**static** **void** methodTwo()

{

//Compile time error

//should not contain static method

}

}

}

* But, here is the interesting point. You can declare a **static field with final**. And such field must be initialized at the time of declaration only. Remember, this rule is only for the fields not for the methods.

**class** OuterClass

{

**class** InnerClass

{

**int** i; //can contain non-static field

**static** **final** **int** ***j*** = 10; //can contain static and final field

//it must be initialized at the time of declaration.

}

}

* Member inner class may contain any number of IIB’s but should not contain any SIB’s.

**class** OuterClass

{

**class** InnerClass

{

**int** i;

{

System.***out***.println("First IIB");

}

{

System.***out***.println("Second IIB");

}

**static**

{

//compile time error

//Member Inner Class should not contain SIB

}

}

}

* We can **access** **both** static and non-static members of outer class inside a member inner class.

**class** OuterClass

{

**int** i; //Non-static field of OuterClass

**static** **int** *j*; //Static field of OuterClass

**void** methodOne()

{

System.***out***.println("Non-Static Method Of OuterClass");

}

**static** **void** methodTwo()

{

System.***out***.println("Static Method Of OuterClass");

}

**class** InnerClass

{

**void** methodOfInnerClass()

{

System.***out***.println(i); //can use non-static field of OuterClass

System.***out***.println(*j*); //can use static field of OuterClass

methodOne(); //can call non-static method of OuterClass

*methodTwo*(); //can call static method of OuterClass

}

}

}

* Below example shows how to instantiate member inner class and how to access it’s members.

**class** OuterClass

{

**class** InnerClass

{

**int** i; //Non-static field of InnerClass

**static** **final** **int** ***j*** = 10; //static and final field of InnerClass

**void** methodOne()

{

System.***out***.println("Non-static method of InnerClass");

}

}

}

**public** **class** InnerClasses

{

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

{

OuterClass outer = **new** OuterClass(); //creating an instance of OuterClass

OuterClass.InnerClass inner = outer.**new** InnerClass(); //creating an instance of InnerClass

System.***out***.println(inner.i); //accessing non-static field of InnerClass

System.***out***.println(OuterClass.InnerClass.***j***); //static field can be accessed directly through class name

inner.methodOne(); //accessing non-static method of InnerClass

}

}

* All members of outer class are accessible inside member inner class and all members of member inner class are accessible inside the outer class irrespective of their visibility.

**class** OuterClass

{

**private** **int** i; //private field of OuterClass

**int** j; //Default field of OuterClass

**protected** **int** k; //protected field of OuterClass

**public** **int** m; //public field of OuterClass

**void** methodOfOuterClass()

{

InnerClass inner = **new** InnerClass(); //creating instance of InnerClass

System.***out***.println(inner.a); //accessing private field of InnerClass

System.***out***.println(inner.b); //accessing default field of InnerClass

System.***out***.println(inner.c); //accessing protected field of InnerClass

System.***out***.println(inner.d); //accessing public field of InnerClass

}

**class** InnerClass

{

**private** **int** a; //private field of InnerClass

**int** b; //Default field of InnerClass

**protected** **int** c; //protected field of InnerClass

**public** **int** d; //public field of InnerClass

**void** methodOfInnerClass()

{

OuterClass outer = **new** OuterClass(); //creating an instance of OuterClass

System.***out***.println(outer.i); //accessing private field of OuterClass

System.***out***.println(outer.j); //accessing default field of OuterClass

System.***out***.println(outer.k); //accessing protected field of OuterClass

System.***out***.println(outer.m); //accessing public field of OuterClass

}

}

}

* Member inner classes can be abstract or can be final but not both.

**class** OuterClass

{

**abstract** **class** InnerClassOne

{

//abstract Inner Class

}

**final** **class** InnerClassTwo

{

//final inner class

}

}

**Local Inner Class In Java :**

* Local inner classes must be defined inside a method or a block.

**class** OuterClass

{

**static**

{

**class** LocalInnerClassOne

{

//Class defined inside Static Initialization Block

}

}

{

**class** LocalInnerClassTwo

{

//Class defined inside Instance Initialization Block

}

}

**void** methodOne()

{

**class** LocalInnerClassThree

{

//Class defined inside a non-static method

}

}

**static** **void** methodTwo()

{

**class** LocalInnerClassFour

{

//Class defined inside a static method

}

}

**void** methodThree()

{

**if**(**true**)

{

**class** LocalInnerClassFive

{

//Class defined inside if-statement

}

}

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

{

**class** LocalInnerClassSix

{

//Class defined inside a for loop

}

}

}

}

* Local Inner Classes can’t be static. Because, local inner classes are nothing but local variables and local variables can’t be static.

**class** OuterClass

{

**void** methodOne()

{

**static** **class** LocalInnerClass

{

//compile time error

//Local Inner class can't be static

}

}

}

* Local inner classes can’t have static members. Only non-static members are allowed inside local inner classes. But local inner classes can contain static and final field.

**class** OuterClass

{

**void** methodOne()

{

**class** LocalInnerClass

{

**int** i; //can contain Non-static field

**static** **final** **int** ***j*** = 10; //can contain static and final field

**static** **int** *k*; //Compile time error : can't have static field

{

//can contain instance initializer

}

**static**

{

//can't have static initializer

}

**void** methodOne()

{

//can contain non-static method

}

**static** **void** methodTwo()

{

//compile time error : can't have static method

}

}

}

}

* To access members of local inner class, you must create an instance of it.

**class** OuterClass

{

**void** methodOne()

{

**class** LocalInnerClass

{

**int** i; //Non-static field

**static** **final** **int** ***j*** = 10; //static and final field

**void** methodOne()

{

System.***out***.println("From LocalInnerClass");

}

}

System.***out***.println(LocalInnerClass.***j***); //static and final field can be accessed directly through class name

LocalInnerClass inner = **new** LocalInnerClass(); //Creatin an object to LocalInnerClass

System.***out***.println(inner.i); //accessing non-static field through object reference

inner.methodOne(); //calling non-static method through object reference

}

}

**public** **class** InnerClasses

{

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

{

OuterClass outer = **new** OuterClass();

outer.methodOne();

}

}

* Local inner classes are local to a method or a block in which they are defined. i.e you can’t use local inner classes outside the method or block in which they are defined.

**class** OuterClass

{

**void** methodOne()

{

**class** LocalInnerClass

{

**int** i; //Non-static field

**static** **final** **int** ***j*** = 10; //static and final field

**void** methodOne()

{

System.***out***.println("From LocalInnerClass");

}

}

LocalInnerClass inner = **new** LocalInnerClass();

//can use LocalInnerClass within the method in which it is defined

}

**void** methodTwo()

{

LocalInnerClass inner = **new** LocalInnerClass(); //compile time error

//can't use LocalInnerClass outside the methodOne().

}

}

* Only final local variables of methods or blocks containing local inner class can be used inside local inner class.

**class** OuterClass

{

**void** methodOne()

{

**int** i; //Non-final local variable

**final** **int** j; //final local variable

**class** LocalInnerClass

{

**void** methodOne()

{

System.***out***.println(i); //compile time error

//can't use non-final local variable

System.***out***.println(j); //can use final local variable

}

}

}

}

* Local inner classes can not be declared with access modifiers. i.e Local inner classes can not be private, protected and public. But they can have private, public, protected and default members in them.

**class** OuterClass

{

**void** methodOne()

{

**private** **class** LocalInnerClassOne

{

//Compile time error

//Local inner class can't be private

}

**protected** **class** LocalInnerClassTwo

{

//Compile time error

//Local inner class can't be protected

}

**public** **class** LocalInnerClassThree

{

//Compile time error

//Local inner class can't be public

}

**class** LocalInnerClassFour

{

**private** **int** i; //can have private member

**protected** **int** j; //can have protected member

**public** **int** k; //can have public member

**int** m; //can have default member

}

}

}

* Local inner classes can be abstract or can be final but not both.

class OuterClass

{

    void methodOne()

    {

        abstract class LocalInnerClassOne

        {

            //abstract local inner class

        }

        final class LocalInnerClassTwo

        {

            //final local inner class

        }

    }

}

[**Anonymous Inner Class In Java**](https://javaconceptoftheday.com/anonymous-inner-class-in-java/)

Anonymous inner class, the name itself suggest that it is a class without a name. Anonymous inner class in java is an inner class or non-static nested class without a name.

Consider the below **class** definition.

**class** SuperClass

{

**void** methodOne()

{

System.out.println("From SuperClass");

}

**void** methodTwo()

{

System.out.println("From SuperClass");

}

}

Let’s consider that we have a requirement in which the above **class** ‘SuperClass’ has to be re-used with little modification to the ‘methodOne()’. To **do** **this**, we have to create a subclass to ‘SuperClass’ and override the ‘methodOne()’ method. Let’s implement **this**.

**class** SubClass **extends** SuperClass

{

@Override

**void** methodOne()

{

System.out.println("From Sub Class");

}

}

To use the methodOne(), we have to create an object of ‘SubClass’ type and call ‘methodOne()’ from that object.

**public** **class** InnerClasses

{

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

{

SubClass subclass = **new** SubClass();

subclass.methodOne();

}

}

This method of implementing is little bit lengthy. There is one more method of implementing **this** requirement which takes less time than **this** and you need to write only few lines of code to implement **this** requirement. That is called Anonymous Inner Class.

Let’s implement **this** requirement with anonymous inner **class**.

**public** **class** InnerClasses

{

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

{

SuperClass superclass = **new** SuperClass()

{

@Override

**void** methodOne()

{

System.out.println("From Anonymous Inner Class");

}

};

superclass.methodOne();

}

}

That’s it. You just have to create an object reference variable of type ‘SuperClass’ and override the method which needs modification in the curly brackets and end with semicolon. No need to create sub **class** separately. This method is easier than the above method. isn’t it?.

Let’s discuss some of features of anonymous inner **class**.

Anonymous inner classes don’t have name. They are nameless.

You can create only one object to anonymous inner **class**. If you want to create another object, you have to write the whole **class** again.

**public** **class** InnerClasses

{

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

{

//First Object Creation

SuperClass firstObject = **new** SuperClass()

{

@Override

**void** methodOne()

{

System.out.println("From First Object");

}

};

//Second Object Creation

SuperClass secondObject = **new** SuperClass()

{

@Override

**void** methodOne()

{

System.out.println("From Second Object");

}

};

}

}

When you are creating an anonymous inner **class**, you are actually creating a sub **class** to a **class** which needs to be modified. This sub **class** doesn’t have name and it is declared in another **class**. That’s why it is called Anonymous Inner Class.

While creating an anonymous inner **class** you are also creating an object to that subclass and it is referenced by **super** **class** reference variable. This also shows the polymorphism. Because, Super **class** reference variable can refer to **super** **class** object and also it’s sub **class** object.

Using anonymous inner **class**, you can implement both **abstract** classes and interfaces.

**abstract** **class** AbstractClass

{

**abstract** **void** methodOne();

**abstract** **void** methodTwo();

}

**interface** InterfaceClass

{

**abstract** **void** methodOfInterface();

}

**public** **class** InnerClasses

{

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

{

//Implementing abstract class

AbstractClass a = **new** AbstractClass()

{

@Override

**void** methodOne()

{

System.out.println("From AbstractClass");

}

//You have to override second abstract method also,

//otherwise, you will get compile time error.

@Override

**void** methodTwo()

{

System.out.println("From AbstractClass");

}

};

//Implementing Interface

InterfaceClass i = **new** InterfaceClass()

{

@Override

**public** **void** methodOfInterface()

{

System.out.println("From Interface");

}

};

}

}

[Inheritance Of Inner Classes In Java](https://javaconceptoftheday.com/inheritance-inner-classes-java/)

One inner **class** can extend another inner **class** of the same **class**.

**class** OuterClass

{

**class** InnerClassOne

{

**int** x = 10;

**void** methodOfInnerClassOne()

{

System.***out***.println("From InnerClassOne");

}

}

**class** InnerClassTwo **extends** InnerClassOne

{

//One Inner Class can extend another inner class

}

}

**public** **class** InnerClasses

{

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

{

OuterClass outer = **new** OuterClass(); //Instantiating OuterClass

OuterClass.InnerClassTwo innerTwo = outer.**new** InnerClassTwo(); //Instantiating InnerClassTwo

System.***out***.println(innerTwo.x); //Accessing inherited field x from InnerClassOne

innerTwo.methodOfInnerClassOne(); //calling inherited method from InnerClassOne

}

}

An inner **class** can be extended by another **class** outside of it’s outer **class**. If you are extending **static** inner **class** (Static nested **class**), then it is a straight forward implementation. If you are extending non-**static** inner **class**, then sub **class** constructor must explicitly call **super** **class** constructor using an instance of outer **class**. Because, you can’t access non-**static** inner **class** without the instance of outer **class**.

**class** OuterClass

{

**static** **class** InnerClassOne

{

//Class as a static member

}

**class** InnerClassTwo

{

//Class as a non-static member

}

}

//Extending Static inner class or static nested class

**class** AnotherClassOne **extends** OuterClass.InnerClassOne

{

//static nested class can be referred by outer class name,

}

//Extending non-static inner class or member inner class

**class** AnotherClassTwo **extends** OuterClass.InnerClassTwo

{

**public** AnotherClassTwo()

{

**new** OuterClass().**super**(); //accessing super class constructor through OuterClass instance

}

}

When an outer **class** is extended by it’s sub **class**, Member inner classes will not be inherited to sub **class**. To use inner **class** properties inside the sub **class** of outer **class**, sub **class** must also have an inner **class** and that inner **class** must extend inner **class** of the outer **class**. For example,

**class** OuterClass

{

**int** x;

**void** methodOfOuterClass()

{

System.out.println("From OuterClass");

}

//Class as a member

**class** InnerClass

{

**int** y;

}

}

**class** AnotherClass **extends** OuterClass

{

//Only fields and methods are inherited.

// To use inner class properties,

//it's inner class must extend inner class of it's super class

**class** AnotherInnerClass **extends** InnerClass

{

//Inner Class of AnotherClass extends Inner Class of OuterClass

}

}

**public** **class** InnerClasses

{

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

{

AnotherClass anotherClass = **new** AnotherClass(); //creating AnotherClass Object

System.out.println(anotherClass.x); //accessing inherited field x from OuterClass

anotherClass.methodOfOuterClass(); //calling inherited method from OuterClass

//Using the properties of InnerClass

AnotherClass.AnotherInnerClass anotherInnerClass = anotherClass.**new** AnotherInnerClass();

//creating object to AnotherInnerClass

System.out.println(anotherInnerClass.y); //accessing inherited field y from InnerClass

}

}

Inner **class** can extend it’s outer **class**. But, it does not serve any meaning. Because, even the **private** members of outer **class** are available inside the inner **class**. Even though, When an inner **class** **extends** its outer **class**, only fields and methods are inherited but not inner **class** itself.

**class** OuterClass

{

**int** x;

**void** methodOfOuterClass()

{

System.out.println("From OuterClass");

}

//Class as a member

**class** InnerClass **extends** OuterClass

{

//only fields and methods are inherited, but not member Inner Classes

}

**class** InnerClassOne

{

//another class as a member

}

}

**public** **class** InnerClasses

{

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

{

OuterClass outer = **new** OuterClass();

//You have to create OuterClass object to access non-static inner class

OuterClass.InnerClass inner = outer.**new** InnerClass(); //creating object to InnerClass

System.out.println(inner.x); //accesiing inherited field x

inner.methodOfOuterClass(); //accessing inherited method

}

}

Questions:

**1) How many types of nested classes are there in java?**

Java supports 2 types of nested classes. They are,

a) [Static Nested Classes](https://javaconceptoftheday.com/nested-classes-in-java/)

b) Non-static Nested Classes OR Inner Classes

Non-static nested classes can be of 3 type,

a) [Member Inner Classes](https://javaconceptoftheday.com/inner-classes-in-java/)

b) [Local Inner Classes](https://javaconceptoftheday.com/local-inner-class-in-java/)

c) [Anonymous Inner Classes](https://javaconceptoftheday.com/anonymous-inner-class-in-java/)

**2) Can we access non-static members of outer class inside a static nested class?**

No, we can’t access non-static members of outer class inside a static nested class. We can access only static members of outer class inside a static nested class.

**3) What are member inner classes in java?**

Member inner classes are the classes which are declared as non-static members of another class. Member inner classes can be accessed only by instantiating the outer class.

**4) Can member inner classes have static members in them?**

No, member inner classes can’t have static members in them. They can have only non-static members. But, exception being the static and final field. i.e member inner class can have static and final field, but it must be initialized at the time of declaration only.

**5) Can we access all the members of outer class inside a member inner class?**

Yes, we can access all the members, both static and non-static, of outer class inside a member inner class.

**6) Can we declare local inner classes as static?**

No. Local inner classes can’t be static.

**7) Can we use local inner classes outside the method or block in which they are defined?**

No. Local inner classes are local to method or block in which they are defined. We can’t use them outside the method or block in which they are defined.

**8) Can we declare local inner classes as private or protected or public?**

No. Local inner classes can’t be declared with access modifiers.They can’t be private or protected or public.

**9) What is the condition to use local variables inside a local inner class?**

The condition is that local variables must be final. We can’t use non-final local variables inside a local inner class.

**10) What are anonymous inner classes in java?**

Anonymous inner classes are the inner classes without a name. You can instantiate an anonymous inner class only once. Click [here](https://javaconceptoftheday.com/anonymous-inner-class-in-java/) for more info on anonymous inner classes.

**11) What is the main difference between static and non-static nested classes?**

The main difference between static and non-static nested classes is that you need not to instantiate the outer class to access static nested classes. But, to access non-static nested classes, you have to instantiate the outer class.

## **String in java**

**1) java.lang.String                     (From JDK 1.0)**

**java.lang.StringBuffer            (From JDK 1.5) java.lang.StringBuilder           (From JDK 1.5)**

2) All these three classes are members of **java.lang** package and they are final classes. That means you can’t create subclasses to these three classes.

3) All three classes implement **Serializable** and **CharSequence** interface.

4) **java.lang.String** objects are **immutable**

**StringBuffer** and**java.lang.StringBuilder** objects are **mutable**.

🡪 Give one example with program

5) Only **String** and **StringBuffer** objects are thread safe bc its a synchronized. **StringBuilder** objects are not thread safe cz its non synchrnized.

What are the ways to declare string:

Using string literal : **String Constant Pool.**

Using new key word : **heap memory**.

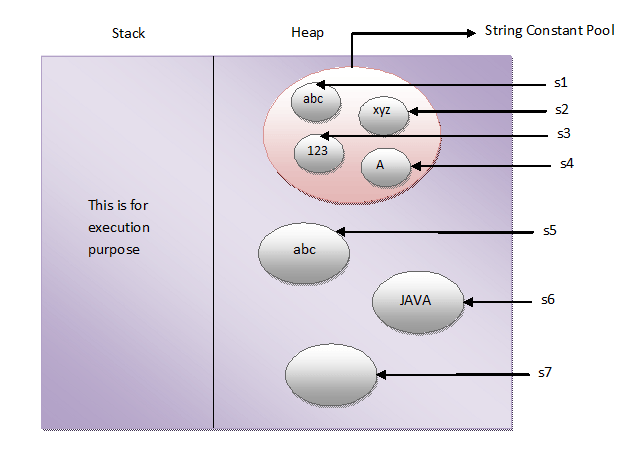
String s1 = "abc";

String s2 = "xyz";

String s3 = "123";

String s4 = "A";

|  |  |
| --- | --- |
|  | String s5 = new String("abc");  char[] c = {'J', 'A', 'V', 'A'};  String s6 = new String(c);  String s7 = new String(new StringBuffer()); |



[When To Use “==”, equals()](https://javaconceptoftheday.com/when-to-use-equals-hashcode-on-strings/) on String::

== is used to compare address

equals() is used to compare content.

###### 1 ##########

String s1="test";

String s2="test";

Integer i1=101;

Integer i2=101;

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

System.out.println(i1==i2); //true

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

System.out.println(i1.equals(i2)); //true

###### 2 ##########

String s1=**new** String("test");

String s2=**new** String("test");

Integer i1=**new** Integer(101);

Integer i2=**new** Integer(101);

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

System.out.println(i1==i2); //false

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

System.out.println(i1.equals(i2)); //true

############# 3 ################

**int** i1=101;

**int** i2=101;

**char** s1='f';

**char** s2='f';

System.out.println(i1==i2); //true

System.out.println(i1.equals(i2)); //compile error (Cannot invoke equals(int) on the primitive type int)

System.out.println(s1.equals(s2)); //compile error (Cannot invoke equals(char) on the primitive type char)

[**An Example To Prove Strings Are Immutable**](https://javaconceptoftheday.com/example-to-prove-strings-are-immutable/)**:**

**Immutability is the fundamental property of string objects. In whatever way you create the string objects, either using string literals or using new operator, they are immutable.**

String s1 = "JAVA";

String s2 = "JAVA";

System.out.println(s1 == s2); //Output : true

s1 = s1 + "J2EE";

System.out.println(s1 == s2); //Output : false

If the strings are mutable, both s1 and s2 should point to same object even after modification. That never happened here. That proves the string objects are immutable in java.

**new operator also immutable:**

**public** **class** StringExamples

{

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

{

String s1 = **new** String("JAVA");

System.out.println(s1); //Output : JAVA

s1.concat("J2EE");

System.out.println(s1); //Output : JAVA

}

}

**2) Is *String* a primitive type or derived type?**

*String* is a derived type.

**7) Which is the final class in these three classes – String, StringBuffer and StringBuilder?**

All three are final. (Interviewer will ask this type of questions to confuse you)

**11) How do you create mutable string objects?**

Using ***StringBuffer* and *StringBuilder*** classes. These classes provide mutable string objects.

**12) Which one will you prefer among “==” and equals() method to compare two string objects?**

I prefer *equals()* method because it compares two string objects based on their content.

**15) How many objects will be created in the following code and where they will be stored?**

|  |  |
| --- | --- |
|  | String s1 = new String("abc");    String s2 = "abc"; |

Two

**20) How many objects will be created in the following code and where they will be stored?**

|  |  |
| --- | --- |
| 1  2 | String s1 = new String("abc");    String s2 = new String("abc"); |

Two objects will be created and they will be stored in the **heap** memory.

**17) I am performing lots of string concatenation and string modification in my code. which class among string, StringBuffer and StringBuilder improves the performance of my code. Remember I also want thread safe code?**

***StringBuffer*** class gives better performance in this scenario. As *String* class is immutable, if you use this class, a new object will be created after every string concatenation or string modification. This will lower the performance of the code. You can use *StringBuilder* also, but it is not thread safe. So, *StringBuffer* will be optimal choice here.

**18) What is string intern?**

String object in the string constant pool is called as *String Intern*. You can create an exact copy of heap memory string object in string constant pool. This process of creating an exact copy of heap memory string object in the string constant pool is called interning

**public** String intern()

**public** **class** InternExample {

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

String s1 = **new** String("hello");

String s2 = "hello";

String s3 = s1.intern();// returns string from pool, now it will be same as s2

System.***out***.println(s1 == s2);// false because reference variables are pointing to different instance

System.***out***.println(s2 == s3);// true because reference variables are pointing to same instance

}

}

|  |  |
| --- | --- |
| **Methods** | **Description** |
| [substring()](https://www.programiz.com/java-programming/library/string/substring) | returns the substring of the string |
| [replace()](https://www.programiz.com/java-programming/library/string/replace) | replaces the specified old character with the specified new character |
| [charAt()](https://www.programiz.com/java-programming/library/string/charat) | returns the character present in the specified location |
| [getBytes()](https://www.programiz.com/java-programming/library/string/getbytes) | converts the string to an array of bytes |
| [indexOf()](https://www.programiz.com/java-programming/library/string/indexof) | returns the position of the specified character in the string |
| [compareTo()](https://www.programiz.com/java-programming/library/string/compareto) | compares two strings in the dictionary order |
| [trim()](https://www.programiz.com/java-programming/library/string/trim) | removes any leading and trailing whitespaces |
| [format()](https://www.programiz.com/java-programming/library/string/format) | returns a formatted string |
| [split()](https://www.programiz.com/java-programming/library/string/split) | breaks the string into an array of strings |
| [toLowerCase()](https://www.programiz.com/java-programming/library/string/tolowercase) | converts the string to lowercase |
| [toUpperCase()](https://www.programiz.com/java-programming/library/string/touppercase) | converts the string to uppercase |
| [valueOf()](https://www.programiz.com/java-programming/library/string/valueof) | returns the string representation of the specified argument |
| [toCharArray()](https://www.programiz.com/java-programming/library/string/tochararray) | converts the string to a char array |

**Append**:

public class Test {

public static void main(String args[]) {

StringBuffer sBuffer = new StringBuffer("test");

sBuffer.append(" String Buffer");

System.out.println(sBuffer); //test String Buffer

}

}

**Insert**:

StringBuffer sb = new StringBuffer("abcdefghijk"); sb.insert(3, "123");

System.out.println(sb); //abc123defghijk

**Reverse**:

public class Test {

public static void main(String args[]) {

StringBuffer buffer = new StringBuffer("Game Plan");

buffer.reverse();

System.out.println(buffer); // nalP emaG

}

}

**Delete:**

StringBuffer sb = new StringBuffer("abcdefghijk");

sb.delete(3, 7);

System.out.println(sb); // abchijk

Programs:

Which method will call?

**void** m1(**int** i, String s)

{

System.***out***.println("string method");

}

**void** m1(**int** i, Object m)

{

System.***out***.println("Object method");

}

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

Test t=**new** Test();

Test t2=**new** Test();

t.m1(3, "sudheer"); //string method

t.m1(3, t2); // object method

}

**26) How do you count the number of occurrences of each character in a string?**

[[Solution](https://javaconceptoftheday.com/how-to-count-occurrences-of-each-character-in-string-in-java/)]

**27) How do you remove all white spaces from a string in java?**

[[Solution](https://javaconceptoftheday.com/java-program-to-remove-all-white-spaces-from-a-string/)]

**28) How do you find duplicate characters in a string?**

[[Solution](https://javaconceptoftheday.com/duplicate-characters-in-a-string-in-java/)]

**29)  Write a java program to reverse a string?**

[[Solution](https://javaconceptoftheday.com/java-program-to-reverse-a-string/)]

**30) Write a java program to check whether two strings are anagram or not?**

[[Solution](https://javaconceptoftheday.com/anagram-program-in-java/)]

**31) Write a java program to reverse a given string with preserving the position of spaces?**

[[Solution](https://javaconceptoftheday.com/how-to-reverse-the-string-with-preserving-the-position-of-spaces/)]

**32) How do you convert string to integer and integer to string in java?**

[[Solution](https://javaconceptoftheday.com/string-to-integer-integer-to-string-conversion-in-java/)]

**33) Write a code to prove that strings are immutable in java?**

[[Solution](https://javaconceptoftheday.com/example-to-prove-strings-are-immutable/)]

**34) Write a code to check whether one string is a rotation of another?**

[[Solution](https://javaconceptoftheday.com/check-one-string-is-rotation-of-another/)]

**35)** **Write a java program to reverse each word of a given string?**

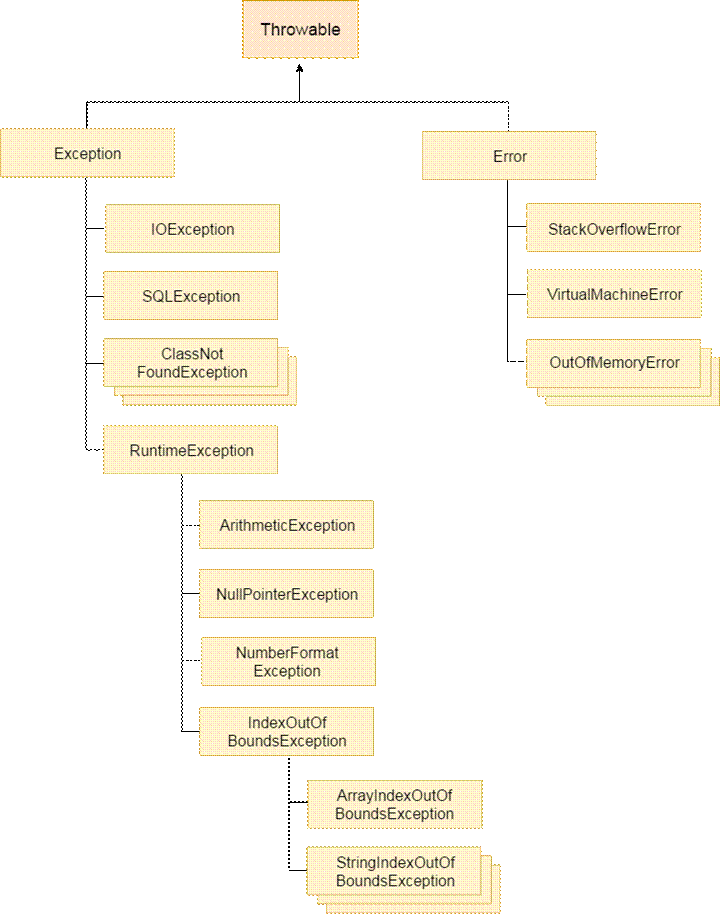
[[Solution](https://javaconceptoftheday.com/how-to-reverse-each-word-of-a-string-in-java/)]

## **Exception handling in java**

* Add points

**Hierarchy of java exception**:

The java.lang.Throwable class is the root class of Java Exception hierarchy which is inherited by two subclasses: Exception and Error.



Difference between Error and Exception:

|  |  |
| --- | --- |
| Errors | Exceptions |
| Errors in java are of type java.lang.Error. | Exceptions in java are of type java.lang.Exception. |
| All errors in java are **unchecked type**. | Exceptions include both checked as well as unchecked type. |
| Errors happen at run time. They will not be known to compiler. | Checked exceptions are known to compiler where as unchecked exceptions are not known to compiler because they occur at run time. |
| It is impossible to recover from errors. | You can recover from exceptions by handling them through try-catch blocks. |
| Errors are mostly caused by the environment in which application is running. | Exceptions are mainly caused by the application itself. |
| Examples : java.lang.StackOverflowError, java.lang.OutOfMemoryError | Examples : Checked Exceptions : SQLException, IOException Unchecked Exceptions : ArrayIndexOutOfBoundException, ClassCastException, NullPointerException |

Types and Difference

1) **Checked Exception**

The classes which directly inherit 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 which inherit RuntimeException are known as unchecked exceptions e.g. ArithmeticException, NullPointerException, ArrayIndexOutOfBoundsException etc. Unchecked exceptions are not checked at compile-time, but they are checked at runtime.

3) Error

Error is **irrecoverable** e.g. OutOfMemoryError, VirtualMachineError, AssertionError etc.

**5) Can we write only try block without catch and finally blocks?     imp**

**No**, It shows compilation error. The try block must be followed by either catch or finally block. You can remove either catch block or finally block but not both.

**try**

{

**int** i = 10/0; //This statement throws ArithmeticException

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

}

**catch**(Exception ex)

{

System.out.println("This block is executed immediately after an exception is thrown");

}

**finally**

{

System.out.println("This block is always executed");

}

From Java 7 onward, we can handle multiple exceptions using **pipe (|) operator.**

**public** **class** ExceptionHandling

{

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

{

String[] s = {"abc", "123", **null**, "xyz"};

//String array containing one null object

**for** (**int** i = 0; i < 6; i++)

{

**try**

{

**int** a = s[i].length() + Integer.parseInt(s[i]);

//This statement may throw NumberFormatException, NullPointerException and ArrayIndexOutOfBoundsException

}

**catch**(NumberFormatException | NullPointerException | ArrayIndexOutOfBoundsException ex)

{

System.out.println("Now, this block handles NumberFormatException, NullPointerException and ArrayIndexOutOfBoundsException");

}

}

}

}

The order of catch blocks should be from most specific to most general ones. Else compile time error  (**Unreachable Catch Block**.)

**try**

{

**int** i = Integer.parseInt("abc"); // NumberFormatException

}

**catch**(Exception ex)

{

System.out.println("This block handles all exception types");

}

**catch**(NumberFormatException ex)

{

//Compile time error

//This block becomes unreachable as

//exception is already handled by above catch block

}

Another eg:

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

{

**try**

{

**return**;

}

**int** invalid; //Compile Time Error

**catch** (Exception e)

{

**return**;

}

**finally**

{

**return**;

}

System.out.println("Unreachable code"); //Compile Time Error : Unreachable Code

}

**Below hierarchy is fine:**

**try**

{

**int** i = Integer.parseInt("abc"); // NumberFormatException

}

**catch**(NumberFormatException ex)

{

System.out.println("This block handles NumberFormatException");

}

**catch**(Exception ex)

{

System.out.println("This block handles all exception types");

}

**catch** (Throwable ex)

{

System.out.println("Throwable is super class of Exception");

}

Exception Propagation in Java

Unlike Unchecked Exceptions, the propagation of exception **does not happen** in case of Checked Exception and its mandatory to use [throw keyword](https://www.geeksforgeeks.org/throw-throws-java/) here. Only unchecked exceptions are propagated.**Checked exceptions throw compilation error.**

// Java program to illustrate

// unchecked exception propagation

// without using throws keyword

**class** Simple {

**void** m()

{

**int** data = 50 / 0; // unchecked exception occurred

// exception propagated to n()

}

**void** n()

{

m();

// exception propagated to p()

}

**void** p()

{

**try** {

n(); // exception handled

}

**catch** (Exception e) {

System.***out***.println("Exception handled");

}

}

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

{

Simple obj = **new** Simple();

obj.p();

System.***out***.println("Normal flow...");

}

}

/\*

Exception handled

Normal flow...

\*/

\*/

Eg 2: illustrate checked exceptions :

// Java program to illustrate exception propagation

// in checked exceptions and it can be propagated

// by throws keyword ONLY

**import** java.io.IOException;

**class** Simple {

// exception propagated to n()

**void** m() **throws** IOException

{

// checked exception occurred

**throw** **new** IOException("device error");

}

// exception propagated to p()

**void** n() **throws** IOException

{

m();

}

**void** p()

{

**try** {

// exception handled

n();

}

**catch** (Exception e) {

System.***out***.println("exception handled");

}

}

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

{

Simple obj = **new** Simple();

obj.p();

System.***out***.println("normal flow...");

}

}

Output:

exception handled

normal flow...

Nested **try**-**catch :**

**try** //Outer try block

{

//Some Statements

**try** //Inner try block

{

//Some Statements

}

**catch** (Exception ex) //Inner catch block

{

}

}

**catch**(Exception ex) //Outer catch block

{

}

[**Return Value From try-catch-finally Blocks**](https://javaconceptoftheday.com/return-value-from-try-catch-finally-blocks/)

If method returns a value and also has try, catch and finally blocks in it, then following two rules need to follow.

1) If finally block returns a value then try and catch blocks may or may not return a value.

2) If finally block does not return a value then both try and catch blocks must return a value.

**public** **class** ReturnValueFromTryCatchFinally

{

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

{

System.out.println(methodReturningValue());

}

**static** **int** methodReturningValue()

{

**try**

{

//This block may or may not return a value as finally block is returning a value

}

**catch** (Exception e)

{

//This block may or may not return a value as finally block is returning a value

}

**finally**

{

**return** 20;

}

}

}

**public** **class** ReturnValueFromTryCatchFinally

{

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

{

System.out.println(methodReturningValue());

}

**static** **int** methodReturningValue()

{

**try**

{

**return** 10;

}

**catch** (Exception e)

{

**return** 20;

}

**finally**

{

//Now, This block may or may not return a value

//as both try and catch blocks are returning a value

}

}

}

3) finally block overrides any return values from try and catch blocks.

**public** **class** ReturnValueFromTryCatchFinally

{

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

{

System.out.println(methodReturningValue()); //Output : 50

}

**static** **int** methodReturningValue()

{

**try**

{

**return** 10;

}

**catch** (Exception e)

{

**return** 20;

}

**finally**

{

**return** 50; //This method returns 50 not 10 or 20

}

}

}

**Analyse and give output**:

**public** **class** ReturnValueFromTryCatchFinally

{

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

{

System.out.println(methodReturningValue());

}

**static** **int** methodReturningValue()

{

**int** i = 0;

**try**

{

i = 1;

**return** i;

}

**catch** (Exception e)

{

i = 2;

**return** i;

}

**finally**

{

i = 3;

}

}

}

Output:1 //cz it wont throw any exception and catch block wont execute.

**Re-throwing An Exception :**

**public** **class** ReturnValueFromTryCatchFinally

{

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

{

**try**

{

methodWithThrow();

}

**catch**(NullPointerException ex)

{

System.out.println("NullPointerException Re-thrown in methodWithThrow() method will be handled here");

}

}

**static** **void** methodWithThrow()

{

**try**

{

String s = **null**;

System.out.println(s.length()); //This statement throws NullPointerException

}

**catch**(NullPointerException ex)

{

System.out.println("NullPointerException is caught here");

**throw** ex; //Re-throwing NullPointerException

}

}

}

Output:

NullPointerException is caught here

NullPointerException Re-thrown in methodWithThrow() method will be handled here

[**Method Overriding With throws Clause**](https://javaconceptoftheday.com/method-overriding-with-throws-clause/)**:**

1.     If super class method is not throwing any exceptions, then it can be overrided with any unchecked type of exceptions, but can not be overrided with checked type of exceptions.

2.     If a super class method is throwing unchecked exception, then it can be overrided in the sub class with same exception or any other unchecked exceptions but can not be overrided with checked exceptions.

**Eg:**

**class** SuperClass

{

**void** methodOfSuperClass()

{

System.out.println("Super class method is not throwing any exceptions");

}

}

**class** SubClass **extends** SuperClass

{

@Override

**void** methodOfSuperClass() **throws** ArrayIndexOutOfBoundsException

{

System.out.println("can be overrided with any unchecked Exception");

}

}

**class** SubClassOne **extends** SuperClass

{

@Override

**void** methodOfSuperClass() **throws** NumberFormatException, NullPointerException, RuntimeException

{

System.out.println("Can be overrided with any number of Unchecked Exceptions");

}

}

**class** SubClassTwo **extends** SuperClass

{

@Override

**void** methodOfSuperClass() **throws** SQLException

{

//Compile time error

//Can not be overrided with checked exception

}

}

1. If super class method is throwing checked type of exception, then it can be overrided with same exception or with it’s sub class exceptions i.e you can decrease the scope of the exception, but can not be overrided with it’s super class exceptions i.e you can not increase the scope of the exception.

Eg:

**class** SuperClass

{

**void** methodOfSuperClass() **throws** IOException

{

System.out.println("Super class method is throwing checked exception");

}

}

**class** SubClass **extends** SuperClass

{

@Override

**void** methodOfSuperClass() **throws** IOException

{

System.out.println("Can be Overrided with same checked exception");

}

}

**class** SubClassOne **extends** SuperClass

{

@Override

**void** methodOfSuperClass() **throws** FileNotFoundException

{

System.out.println("Can be overrided with checked Exception with lesser scope");

}

}

**class** SubClassTwo **extends** SuperClass

{

@Override

**void** methodOfSuperClass() **throws** NullPointerException, ArrayIndexOutOfBoundsException, FileNotFoundException

{

System.out.println("Can be overrided with any unchecked exceptions and checked exception with lesser scope");

}

}

**class** SubClassThree **extends** SuperClass

{

@Override

**void** methodOfSuperClass() **throws** Exception

{

//Compile time error

//Can not be overrided with checked exception with higher scope

}

}

One more eg:

**class** SuperClass

{

**void** methodOfSuperClass() **throws** IOException, ClassNotFoundException, NumberFormatException

{

System.out.println("Super class method is throwing both checked and unchecked exceptions");

}

}

**class** SubClass **extends** SuperClass

{

@Override

**void** methodOfSuperClass() **throws** IOException, ClassNotFoundException

{

System.out.println("Can be Overrided with same checked exceptions");

}

}

**class** SubClassOne **extends** SuperClass

{

@Override

**void** methodOfSuperClass() **throws** FileNotFoundException

{

System.out.println("Can be overrided with checked Exception with lesser scope");

}

}

**class** SubClassTwo **extends** SuperClass

{

@Override

**void** methodOfSuperClass() **throws** NullPointerException, ArrayIndexOutOfBoundsException, FileNotFoundException

{

System.out.println("Can be overrided with any other unchecked exceptions and checked exception with lesser scope");

}

}

**class** SubClassThree **extends** SuperClass

{

@Override

**void** methodOfSuperClass() **throws** Exception

{

//Compile time error

//Can not be overrided with checked exception with higher scope

}

}

custom exception or user-defined exception.

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

**class** InvalidAgeException **extends** Exception

{

InvalidAgeException(String s)

{

**super**(s);

}

}

**public** **class** Test {

**static** **void** validate(**int** age) **throws** InvalidAgeException

{

**if** (age < 18)

**throw** **new** InvalidAgeException("not valid age");

**else**

System.***out***.println("welcome to vote");

}

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

{

**try**

{

*validate*(13);

}

**catch** (Exception m)

{

m.printStackTrace();

}

System.***out***.println("rest of the code...");

}

}

/\*

com.hcl.InvalidAgeException: not valid age

at com.hcl.Test.validate(Test.java:15)

at com.hcl.Test.main(Test.java:24)

rest of the code...

\*/

**Difference between throw and throws:**

| **throw** | **throws** |
| --- | --- |
| Throw is a keyword which is used to throw an exception explicitly in the program **inside a function** or inside a block of code. | Throws is a keyword used in the **method signature** used to declare an exception which might get thrown by the function while executing the code. |
| Internally throw is implemented as it is allowed to **throw only single exception** at a time i.e we cannot throw multiple exception with throw keyword. | On other hand we can **declare multiple exceptions** with throws keyword that could get thrown by the function where throws keyword is used. |
| With throw keyword we can propagate only **unchecked exception** i.e checked exception cannot be propagated using throw. | On other hand with throws keyword **both** checked and unchecked exceptions can be declared and for the propagation checked exception must use throws keyword followed by specific exception class name. |

## **Comparable and Comparator interface:**

|  |  |
| --- | --- |
| **Comparable** | **comparator** |
| It has only one method called **compareTo**(Object) | It has 2 methods: **compare**(Obj 1, Obj 2) and **equals**(obj o) method |
| It provides only **single sorting** sequence | It provide **multiple sorting sequence**.(we can sort the elements based on any data member) |

**public** **class** Employee {

String name;

**int** age;

Employee (String name, **int** age)

{

**this**.name=name;

**this**.age=age;

}

}

**import** java.util.Comparator;

**public** **class** AgeComparator **implements** Comparator<Employee>{

@Override

**public** **int** compare(Employee s1, Employee s2) {

**if**(s1.age==s2.age)

{

**return** 0;

}

**else** **if**(s1.age>s2.age) {

**return** 1;

}

**else**

{

**return** -1;

}

}

}

**import** java.util.Comparator;

**public** **class** NameComparator **implements** Comparator<Employee>

{

@Override

**public** **int** compare(Employee s1, Employee s2)

{

**return** s1.name.compareTo(s2.name);

}

}

**import** java.util.ArrayList;

**import** java.util.Collections;

**import** java.util.Iterator;

**public** **final** **class** Test {

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

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

al.add(**new** Employee("vijay",23));

al.add(**new** Employee("ajay",27));

al.add(**new** Employee("jay",21));

Collections.*sort*(al,**new** AgeComparator());

System.***out***.println("sorting by age");

Iterator itr=al.iterator();

**while**(itr.hasNext())

{

Employee s=(Employee)itr.next();

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

}

Collections.*sort*(al,**new** NameComparator());

System.***out***.println("\nsorting by name");

Iterator itr2=al.iterator();

**while**(itr2.hasNext())

{

Employee s=(Employee)itr2.next();

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

}

}

}

Output:

sorting by age

21

23

27

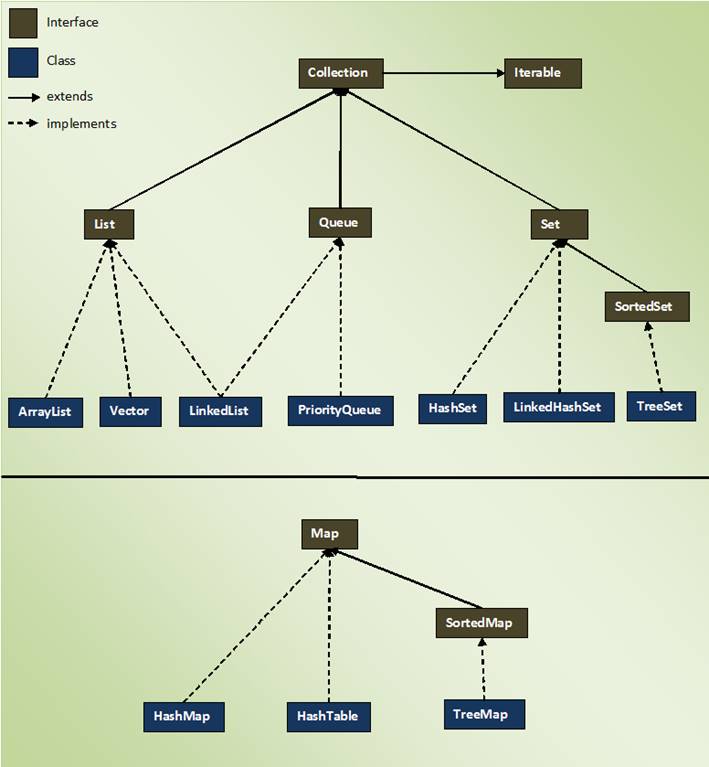
sorting by name

ajay

jay

vijay

## **Collection iteration n convertion**



**########    common file  ################**

**public** **class** Employee {

**private** **int** id;

**private** String name;

// constructor

**//setters n getters**

}

##########################################

**public** **class** ListIteration {

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

            //List<String> al = new ArrayList<String>();

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

            al.add("sudheer");

            al.add("sudheer2");

            al.add("sudheer2");

            System.***out***.println("### using forEach() method  ###");

**for**(String s:al){

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

            }

            System.***out***.println("\n### using iterator interface  ###");

            Iterator<String> itr=al.iterator();

**while**(itr.hasNext()){

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

            }

            System.***out***.println("\n### using ListIterator interface  ###");

            ListIterator<String> itr2=al.listIterator();

**while**(itr2.hasNext()){

                  System.***out***.println(itr2.next());

                  //itr2.next();

            }

            System.***out***.println("\n### reverse iteration  ###");

**while**(itr2.hasPrevious()){

                  System.***out***.println(itr2.previous());

                  System.***out***.println(itr2.hashCode());

            }

      }

}

/\*output:

      ### using forEach() method  ###

      sudheer

      sudheer2

      sudheer2

      ### using iterator interface  ###

      sudheer

      sudheer2

      sudheer2

      ### using ListIterator interface  ###

      sudheer

      sudheer2

      sudheer2

      ### reverse iteration  ###

      sudheer2

      190331520

      sudheer2

      190331520

      Sudheer

190331520\*/

################# EmployeeListIteration ################################

**public** **class** EmployeeListIteration {

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

            Employee e=**new** Employee(1, "sudheer");

            Employee e2=**new** Employee(2, "sudheer2");

            Employee e3=**new** Employee(2, "sudheer3");

            LinkedList<Employee> al=**new** LinkedList<Employee>();

            al.add(e);

            al.add(e2);

            al.add(e3);

            System.***out***.println("### using forEach() method  ###");

**for**(Employee emp:al){

                  System.***out***.println(emp.getId()+" "+emp.getName());

            }

            System.***out***.println("\n### using iterator interface  ###");

            Iterator<Employee> itr=al.iterator();

**while**(itr.hasNext()){

                  Employee emp=itr.next();

                  System.***out***.println(emp.getId()+" "+emp.getName());

            }

      }

}

/\*output:

      ### using forEach() method  ###

      1 sudheer

      2 sudheer2

      2 sudheer3

      ### using iterator interface  ###

      1 sudheer

      2 sudheer2

      2 sudheer3\*/

###############        MapIterator   ####################################

**public** **class** MapIterator{

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

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

        m.put(101,"sudheer");

        m.put(101,"sudheer2");

        m.put(102,"sudheer");

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

**for**(Entry<Integer, String> e:m.entrySet()){

            System.***out***.println(e.getKey()+" "+e.getValue());

        }

      }

}

//output:

/\*HashMap size is 2                       //cz does not allow duplicate keys

      102 sudheer

      101 sudheer2\*/

##############           EmployeeMapIteration    #############################

**public** **class** EmployeeMapIteration {

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

            Employee e1=**new** Employee(1, "sudheer");

            Employee e2=**new** Employee(2, "sudheer2");

            Employee e3=**new** Employee(3, "sudheer3");

        HashMap<Integer,Employee> m = **new** HashMap<Integer,Employee>();

        m.put(101,e1);

        m.put(102,e2);

        m.put(103,e3);

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

        System.***out***.println("\n### using forEach() method  ###");

**for**(Entry<Integer, Employee> e:m.entrySet()){

            Employee emp=e.getValue();

            System.***out***.println(e.getKey()+" "+emp.getId()+" "+emp.getName());

        }

      }

}

/\*output:

      HashMap size is 3

      ### using forEach() method  ###

      102 2 sudheer2

      103 3 sudheer3

      101 1 sudheer\*/

#############             MapToListConvertion     #################

**public** **class** MapToListConvertion {

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

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

        m.put(101,"sudheer");

        m.put(102,"sudheer2");

        m.put(103,"sudheer3");

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

        System.***out***.println("converting HashMap keys into arrayList");

            List<Integer> k=**new** ArrayList<Integer>(m.keySet());

            Collections.*sort*(k);

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

            System.***out***.println("\nconverting HashMap values into arrayList");

            List<String> values=**new** ArrayList<String>(m.values());

            Collections.*sort*(values);

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

            System.***out***.println("\nconverting HashMap keys & values into arrayList");

            List<Entry<Integer,String>> entry=**new** ArrayList<Entry<Integer,String>>(m.entrySet());

**for** (**int** i = 0; i < entry.size(); i++) {

                  System.***out***.println(entry.get(i));

            }

      }

}

/\*output:

      HashMap size is 3

      converting HashMap keys into arrayList

      [101, 102, 103]

      converting HashMap values into arrayList

      [sudheer, sudheer2, sudheer3]

      converting HashMap keys & values into arrayList

      102=sudheer2

      103=sudheer3

      101=sudheer

\*/

###########   MapCustomObjectToListConversion     ########################

**public** **class** MapCustomObjectToListConversion {

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

            Employee e1=**new** Employee(1, "sudheer");

            Employee e2=**new** Employee(2, "sudheer2");

            Employee e3=**new** Employee(3, "sudheer3");

            HashMap<Integer,Employee> m = **new** HashMap<Integer,Employee>();

            m.put(101,e1);

        m.put(102,e2);

        m.put(103,e3);

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

        System.***out***.println("converting HashMap keys into arrayList");

            List<Integer> k=**new** ArrayList<Integer>(m.keySet());

            Collections.*sort*(k);

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

            System.***out***.println("\nconverting HashMap values into arrayList");

            List<Employee> values=**new** ArrayList<Employee>(m.values());

**for** (**int** i = 0; i < values.size(); i++) {

                  Employee emp=values.get(i);

                  System.***out***.println(emp.getId()+" "+emp.getName());

            }

            System.***out***.println("\nconverting HashMap keys & values into arrayList");

            List<Entry<Integer,Employee>> entry=**new** ArrayList<Entry<Integer,Employee>>(m.entrySet());

**for** (**int** i = 0; i < entry.size(); i++) {

                  Entry<Integer, Employee> e=entry.get(i);

                  Employee emp=e.getValue();

                  System.***out***.println(e.getKey()+" "+emp.getId()+" "+emp.getName());

            }

      }

}

/\*    output:

      HashMap size is 3

      converting HashMap keys into arrayList

      [101, 102, 103]

      converting HashMap values into arrayList

      2 sudheer2

      3 sudheer3

      1 sudheer

      converting HashMap keys & values into arrayList

      102 2 sudheer2

      103 3 sudheer3

      101 1 sudheer

\*/

###############################

## **collection programs and some questions**

7. what is generics in java? Where it is used?

Generics was added in Java 5 to provide **compile-time type checking** and removing risk of ClassCastException that was common while working with collection classes

List list = **new** ArrayList();

list.add("abc");

list.add(**new** Integer(5)); //OK

**for**(Object obj : list)

{

//type casting leading to ClassCastException at runtime

String str=(String) obj;

}

Above code compiles fine but throws ClassCastException at runtime because we are trying to cast Object in the list to String whereas one of the element is of type Integer.

So to overcome this we used like below:

List<String> list1 = **new** ArrayList<String>(); // java 7 ? List<String> list1 = new ArrayList<>();

list1.add("abc");

//list1.add(new Integer(5)); //compiler error

**for**(String str : list1)

{

//no type casting needed, avoids ClassCastException

}

[**Does the capacity of ArrayList decrease when we remove elements?**](https://stackoverflow.com/questions/23831157/does-the-capacity-of-arraylist-decrease-when-we-remove-elements)

ArrayList have a default capacity of 10 objects. As the size exceeds 10 objects, an ArrayList will internally increase its capacity. Does the capacity decrease when we remove the object from ArrayList.

**It doesn't decrease this automatically. From the doc.**

**public void trimToSize()**

Trims the capacity of this ArrayList instance to be the list's current size. An application can use this operation to minimize the storage of an ArrayList instance.

+++++++++++++++++++++++

**What is the difference between size and capacity in Java List?**

**What is  size?**  The number of elements stored in the List .

**What is  capacity ?**  The current capacity of a List that means maximum number of elements it can hold currently.

+++++++++++++++++++++++

Is it possible to add mutable object as a key in HashMap?

++++++++++++++++++++

4. what is the differance between list/set/map

Duplicate Objects

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. See [here](http://java67.blogspot.sg/2012/08/difference-between-list-and-set-in-java.html) for more difference between List and Set data structure in Java.

 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.

|  |  |
| --- | --- |
| **1.**[**Differences Between Array Vs ArrayList In Java**](http://javaconceptoftheday.com/differences-between-array-vs-arraylist-in-java/)**?** 1.  Arrays are static in nature. Arrays are fixed length data structures. You can’t change their size once they are created. | 1.  ArrayList is dynamic in nature. Its size is automatically increased if you add elements beyond its capacity. |
| 2.     Arrays can hold both primitives as well as objects. | 2.  ArrayList can hold only objects. |
| 3.     Arrays can be iterated only through *for* loop or *for-each* loop. | 3.  ArrayList provides iterators to iterate through their elements. |
| 4.     The size of an array is checked using *length*attribute. | 4.  The size of an ArrayList can be checked using *size()* method. |

[**Differences Between HashMap Vs HashSet In Java**](http://javaconceptoftheday.com/differences-between-hashmap-vs-hashset-in-java/)**?**

|  |  |
| --- | --- |
| 1.  *HashSet* implements *Set* interface. | 1. *HashMap* implements *Map* interface. |
| 2.  *HashSet* stores the data as objects. | 2. *HashMap* stores the data as key-value pairs. |
| 3.  *HashSet* internally uses *HashMap*. | 3. *HashMap* internally uses an array of *Entry<K, V>* objects. |
| 4.  *HashSet* doesn’t allow duplicate elements. | 4. *HashMap* doesn’t allow duplicate keys, but allows duplicate values. |
| 5.  *HashSet* allows only one null element. | 5. *HashMap* allows one null key and multiple null values. |

**What is**[**Fail Fast Vs Fail Safe Iterators?   Examples**](http://javaconceptoftheday.com/fail-fast-and-fail-safe-iterators-in-java-with-examples/)**?**

|  |  |
| --- | --- |
| 1.  Fail-Fast iterators doesn’t allow modifications of a collection while iterating over it. | 1.     Fail-Safe iterators allow modifications of a collection while iterating over it. |
| 2.  These iterators throw ***ConcurrentModificationException*** if a collection is modified while iterating over it. | 2.  These iterators don’t throw any exceptions if a collection is modified while iterating over it. |
| 3.     They use original collection to traverse over the elements of the collection. | 3.     They use copy of the original collection to traverse over the elements of the collection. |
| 4.    These iterators don’t require extra memory. | 4.     These iterators require extra memory to clone the collection. |
| 5.     Ex : Iterators returned by *ArrayList*, *Vector*, *HashMap*. | 5.     Ex : Iterator returned by *ConcurrentHashMap.* |

*//Example for fail fast:*

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

        list.add(1452);

        list.add(6854);

        list.add(8741);

        list.add(6542);

        list.add(3845);

        Iterator<Integer> it = list.iterator();

**while** (it.hasNext())

        {

            Integer integer = (Integer) it.next();

            list.add(8457);      //This will throw ConcurrentModificationException

        }

*//Example for fail safe:*

ConcurrentHashMap<String, Integer> map = **new** ConcurrentHashMap<String, Integer>();

         map.put("ONE", 1);

         map.put("TWO", 2);

         map.put("THREE", 3);

         map.put("FOUR", 4);

         Iterator<String> it = map.keySet().iterator();

**while** (it.hasNext())

         {

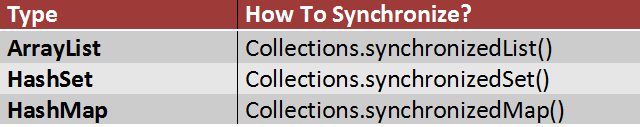
             String key = (String) it.next();

             System.***out***.println(key+" : "+map.get(key));

             map.put("FIVE", 5);     //This will not be reflected in the Iterator

         }

**[How To Synchronize ArrayList, HashSet And HashMap In Java?](http://javaconceptoftheday.com/synchronize-arraylist-hashset-and-hashmap-in-java/)**



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

        list.add("JAVA");

        list.add("JSP");

        list.add("ANDROID");

      List<String> synchronizedList = Collections.synchronizedList(list);

      //you must use synchronized block while iterating over synchronizedList

**synchronized** (synchronizedList)

        {

            Iterator<String> it = synchronizedList.iterator();

**while** (it.hasNext())

            {

                System.out.println(it.next());

            }

        }

*//doing hashSet*

       Set<String> synchronizedSet = Collections.synchronizedSet(set);

*//doing hashSet*

      Map<String, Integer> synchronizedMap = Collections.synchronizedMap(map);

      (before iterating u have to define that block as synchronized)

**Common methods used for ArrayList and LinkedList :**

**public** **class** Test7 {

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

            ArrayList<String> list2=**new** ArrayList<String>();

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

            list.add("puma0");

            list.add("puma1");

            list.add("puma1");

            list.indexOf("puma1");        //1

            list.lastIndexOf("puma1");    //2

            list.size();

            list.add("puma3");

            list.add(2,"puma2");    //adding another element at particular index.

            list.get(0);

            list.remove(1);               //based on index

            list.remove("puma2");         //based on element

            list.contains("puma2");

            list.isEmpty();

            list.addAll(list2);

            list.addAll(1, list2);        //adding another list at particular index.

            list.removeAll(list2);

            list.containsAll(list2);

            list.equals(list2);

            list.clear();                 //removing all elements

      }

}

**But in case of HashSet, LinkedHashSet and TreeSet we can’t use methods with who is working based on index like:**

list.indexOf("puma1");        //1

list.lastIndexOf("puma1");    //2

list.add(2,"puma2");

list.get(0);

list.addAll(1, list2);

**+++++++++++++++++++++++++++**

**Q) create an list and add elements to it, restrict list size only 5, as we keep on adding old elements should remove from list and new elements need to add. And list should contain only unique elements.**

**Points of requirment:**

**Unique elements   // so using contains() in if condition**

**need to maintain insertion order.**

**Need to call removing 1st element n needs to add element at last.**

**public** **class** DataStructure1 {

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

            LinkedList<Integer> list=**new** LinkedList<Integer>();

**int**[] ar={1,2,3,4,5,6,6,7,8,9};

**for**(**int** i=0;i<ar.length;i++){

**if**(!list.contains(ar[i])){

**if**(list.size()>4){

                                    list.removeFirst();

                                    list.addLast(ar[i]);

                              }

**else**{

                                    list.add(ar[i]);

                              }

                        }

                  }

                  System.***out***.println(list);           //[5,6,7,8,9]

            }

}

++++++++++++++++++++++++++++++++++++++++++

**// what is the expected output:**

**public** **class** DataStructure1 {

      {

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

      }

**Static**

{

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

      }

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

            DataStructure1 d1=**new** DataStructure1();

            DataStructure1 d2=**new** DataStructure1();

            DataStructure1 d3=**new** DataStructure1();

      }

}

Output:

/\*    A

      B

      B

      B\*/

++++++++++++++++++++++++++++++

//Create a list with 5 elements . And make a number of left shift based on the number passed in the method signature.

Eg:

[puma0, puma1, puma2, puma3, puma4]

*shiftLeftOper*(2);

 [puma2, puma3, puma4, puma0, puma1]

**public** **class** Test7 {

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

            ArrayList<String> list2=**new** ArrayList<String>();

*shiftLeftOper*(2);

      }

**public** **static** **void** shiftLeftOper(Integer inputNumber) {

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

            list.add("puma0");

            list.add("puma1");

            list.add("puma2");

            list.add("puma3");

            list.add("puma4");

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

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

                  list.add(list.size(), list.get(0));

                  list.remove(0);

            }

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

      }

}

//for right shift :

Eg:

/\*

[puma0, puma1, puma2, puma3, puma4]

[puma3, puma4, puma0, puma1, puma2]\*/

list.add(0,list.get(list.size()-1));

list.remove(list.size()-1);

++++++++++++++++++++++++++++++++++++++++

how will you find max n min element in hashSet/HashMap?

public String getMapKeyWithHighestValue(HashMap<String, Integer> map) {

    String keyWithHighestVal = "";

    // getting the maximum value in the Hashmap

    int maxValueInMap = (Collections.max(map.values()));

    //iterate through the map to get the key that corresponds to the maximum value in the Hashmap

    for (Map.Entry<String, Integer> entry : map.entrySet()) {  // Iterate through hashmap

        if (entry.getValue() == maxValueInMap) {

             keyWithHighestVal = entry.getKey();     // this is the key which has the max value

        }

     }

    return keyWithHighestVal;

}

+++++++++++++++++++++++++++++++

   find nth last element from the linked list.

  find pairs in an array whose sum is equal to given number.

+++++++++++++++++++++++++++++

## **hashcode and equals**

**########    common file  ################**

**public** **class** Employee {

**private** **int** id;

**private** String name;

//constructor

**//setter n getter**

            @Override equals

**public** **boolean** equals(Object obj) {

                  -----

                  -----

}

}

**public** **class** EqualsMethodWithCustomObject {

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

Employee e1=**new** Employee(101, "sudheer");

          Employee e2=**new** Employee(101, "sudheer");

        System.***out***.println("e1 hashcode = " + e1.hashCode());

        System.***out***.println("e2 hashcode = " + e2.hashCode());

        System.***out***.println("Checking equality between e1 and e2 = " + e1.equals(e2));

}

}

//before overiding of equals:

      e1 hashcode = 1534619972

      e2 hashcode = 1829164700

      Checking equality between e1 and e2 = false (cz no equals method is overrided)

//after overiding of equals:

          e1 hashcode = 1079268934

          e2 hashcode = 1829164700

          Checking equality between e1 and e2 = true

+++++++++ **#working with List** ++++++++++++++++

**public** **class** EqualsMethodWithArrayListObject {

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

             Student alex = **new** Student(1, "Alex");

            Student alex2 = **new** Student(1, "Alex");

              List < Student > studentsLst = **new** ArrayList < Student > ();

              studentsLst.add(alex);

              studentsLst.add(alex2);

              System.***out***.println("Arraylist size = " + studentsLst.size());

              System.***out***.println("Arraylist contains Alex = " + studentsLst.contains(**new** Student(1, "Alex")));

      }

}

//before overiding of equals:

      //Arraylist size = 2

      //Arraylist contains Alex = false

//after overiding of equals:

      //Arraylist size = 2

      //Arraylist contains Alex = true

+++++++++++ #**working with HashSet**  ++++++++++++++++++++++++++++++

//for primitive and non primitive data type : its not required to override hashcode n equals

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

hs.add("sudheer");

hs.add("sudheer3");

hs.add("sudheer");

System.***out***.println(hs);       //[sudheer, sudheer3]

++++++++++++++++++++

 //for custom objects

**public** **class** Student {

**private** **int** id;

**private** String name;

**//constructor**

**//setter n getter**

            @Override equals

**public** **boolean** equals(Object obj) {

                  -----

                  -----

}

@Override

**public** **int** hashCode() {

-----

                  -----

}

**public** **class** HashcodeEqualsWithHashSet {

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

             Student alex = **new** Student(1, "Alex");

            Student alex2 = **new** Student(1, "Alex");

HashSet < Student > studentsLst = **new** HashSet < Student > ();

              studentsLst.add(alex);

              studentsLst.add(alex2);

              System.***out***.println("hashCode is= "+alex.hashCode());

              System.***out***.println("hashCode is= "+alex2.hashCode());

              System.***out***.println("HashSet size = " + studentsLst.size());

              System.***out***.println("HashSet contains Alex = " + studentsLst.contains(**new** Student(1, "Alex")));

      }

}

**They can ask the output based on below scenarios:**

//before overiding of equals n hashcode:

/\*    hashCode is= 837503310

      hashCode is= 1310308732

      HashSet size = 2

      HashSet contains Alex = false\*/

//after overiding only equals:

/\*    hashCode is= 837503310

      hashCode is= 1310308732

      HashSet size = 2

      HashSet contains Alex = false\*/

//overiding only hashcode not equals

/\*    hashCode is= 2044446

      hashCode is= 2044446

      HashSet size = 2

      HashSet contains Alex = false\*/

//after overriding of equals n hashCode                     <--important

/\*    hashCode is= 2044446

      hashCode is= 2044446

      HashSet size = 1

      HashSet contains Alex = true\*/

Conclusion:

1. Overriding equals() and hashcode() is v much recommended in order to working with hashing data structures like: HashSet, HashMap, HashTable ... etc

2. If two objects are equal, they MUST have the same hash code. (but you need to override hashcode)

3. If two objects have the same hash code, it doesn't mean that they are equal.

4. Doesn’t maintaining insertion order

5. Overriding hashcode() alone doesn't force Java to ignore memory addresses when comparing two objects.

**#working with TreeSet**

Set<String> hs=**new** TreeSet<String>();

      hs.add("sudheer");

       hs.add("sudheer3");

       hs.add("sudheer");

       hs.add("sudheer5");

       System.***out***.println(hs);  //[sudheer, sudheer3, sudheer5]

++++++++++++++++++++

//for custom objects

Set<Employee> hs=**new** TreeSet<Employee>();

hs.add(**new** Employee(101, "sudheer"));

hs.add(**new** Employee(103, "sudheer"));

hs.add(**new** Employee(101, "sudheer"));

System.***out***.println(hs);       //**java.lang.ClassCastException**

Hence we should implement comparable/comparator in case of custom object for TreeSet.

**//after implementing comparable/comparator in employee class**

Set<Employee> hs=**new** TreeSet<Employee>();

hs.add(**new** Employee(101, "sudheea"));

hs.add(**new** Employee(103, "sudheer"));

hs.add(**new** Employee(101, "sudheer"));

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

Output will be depends on for which data member you implemented comparable/comparator.

Implemented comparable on id: (in this case, It takes only unique id employees n sorting based on id ascending)

101 Sudheer

103 sudheer

Implemented comparable on name: (in this case, It takes only unique name employees n sorting based on alphabetic ascending)

101 Sudheea

101 sudheer

Conclusion:

·       Doesn’t allow duplicate

·       Maintaining insertion order

·       we should implement comparable/comparator in case of custom object (else classCastExce)

## **interanal working of HashMap**

**What Is Entry Class?**

HashMap has an inner class called an Entry Class which holds the key and values. And there is also something called next, which you will get to know a bit later.

**static** **class** Entry<K,V> **implements** Map.Entry<K,V>

{

**final** K key;

V value;

Entry<K,V> next;

**final** **int** hash;

........

}

Inserting Key-Value Pair: Putting one key-value pair in above HashMap

map.put(new Key("vishal"), 20);

**Steps:**

1.      First, it checks if the key given is null or not. If the given key is null, it will be stored in the zero position, as the hashcode of null will be zero. else

2.    Then it applies the hashcode to the key .hashCode() by calling the hashcode method. In order to get the value within the limits of an array. Calculate hash code of Key {“vishal”}. It will be generated as 118.

3.    The indexFor() method is used to get the exact location to store the Entry object. So assume  by using index method it will be 6.

4.    Create a node object as :

          {

            int hash = 118

            // {"vishal"} is not a string but

            // an object of class Key

            Key key = {"vishal"}

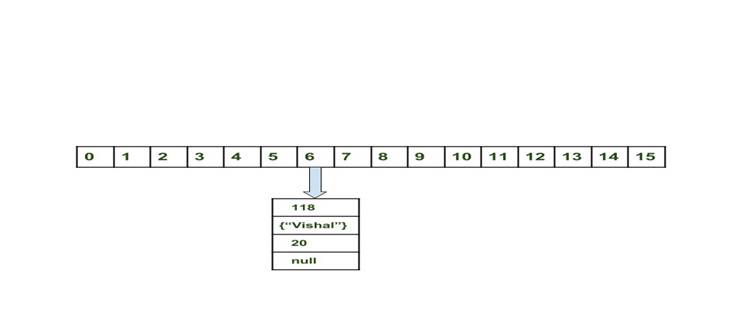
            Integer value = 20

            Node next = null

          }

5.     Place this object at index 6, if no other object is presented there.

Now HashMap becomes :



§  **Inserting another Key-Value Pair:** Now, putting other pair that is,

§  map.put(new Key("sachin"),30);

**Steps:**

1.      First, it checks if the key given is null or not. If the given key is null, it will be stored in the zero position, as the hashcode of null will be zero. else

2.    Then it applies the hashcode to the key .hashCode() by calling the hashcode method. In order to get the value within the limits of an array. Calculate hashCode of Key {“sachin”}. It will be generated as 115.

3.    The indexFor() method is used to get the exact location to store the Entry object. So assume  by using index method it will be 6.

4.    Create a node object as :

1.      {

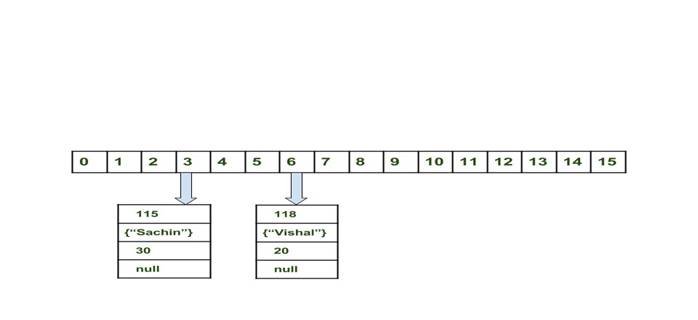
2.      int hash = 115

3.      Key key = {"sachin"}

4.      Integer value = 30

5.       Node next = null

}

Place this object at index 3 if no other object is presented there.  
Now Ha[](https://cdncontribute.geeksforgeeks.org/wp-content/uploads/Hashmap_working_2.jpg)shMap becomes :

§  **In Case of collision:** Now, putting another pair that is,

§  map.put(new Key("vaibhav"),40);

**Steps:**

1.      Then it applies the hashcode to the key .hashCode() by calling the hashcode method. In order to get the value within the limits of an array. Calculate hash code of Key {“vaibhav”}. It will be generated as 118.

2.    Calculate index by using index method it will be 6.

3.    Create a node object as :

4.     {

5.       int hash = 118

6.      Key key = {"vaibhav"}

7.       Integer value = 40

8.      Node next = null

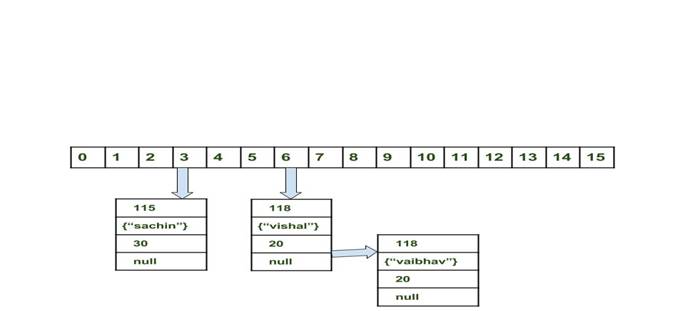
}

9.    Place this object at index 6 if no other object is presented there.

10.  In this case a node object is found at the index 6 – this is a case of collision.

11.  In that case, check via hashCode() and equals() method that if both the keys are same.

12.If keys are same, replace the value with current value.

13.  Otherwise connect this node object to the previous node object via linked list and both are stored at index 6.  
[](https://cdncontribute.geeksforgeeks.org/wp-content/uploads/Hashmap_working_3.jpg)Now HashMap becomes :

**Using get method()**

Now lets try some get method to get a value. get(K key) method is used to get a value by its key. If you don’t know the key then it is not possible to fetch a value.

Fetch the data for key sachin: Get method:

§  map.get(new Key("sachin"));

**Steps:**

1.      Calculate hash code of Key {“sachin”}. It will be generated as 115.

2.    Calculate index by using index method it will be 3.

3.    Go to index 3 of array and compare first element’s key with given key. If both are equals then return the value, otherwise check for next element if it exists.

4.    In our case it is found as first element and returned value is 30.

§  Fetch the data for key vaibahv:

§  map.get(new Key("vaibhav"));

**Steps:**

1.      Calculate hash code of Key {“vaibhav”}. It will be generated as 118.

2.    Calculate index by using index method it will be 6.

3.    Go to index 6 of array and compare first element’s key with given key. If both are equals then return the value, otherwise check for next element if it exists.

4.    In our case it is not found as first element and next of node object is not null.

5.     If next of node is null then return null.

6.    If next of node is not null traverse to the second element and repeat the process 3 until key is not found or next is not null.

How are Duplicate Keys Prevented in HashMap?

        Map map = new HashMap();

                  map.put(1,"sam");

                  map.put(1,"Ian");

                  map.put(1,"Scott");

                  map.put(null,"asdf");

                  System.out.println(map);

For the above code, you will get the output as  {null=asdf, 1=Scott} ,  as the values sam  and Ian  will be replaced by Scott. So, how does this happen?

All the Entry Objects in the LinkedList will have the same hashcode, but HashMap uses equals() . This method checks the equality, so if **key.equals(k)**is true, then it will replace the value object inside the Entry class and not the key. So this way it prevents the duplicate key from being inserted.

Important Points

7.     Time complexity is almost constant for put and get method until rehashing is not done.

8.    In case of collision, i.e. index of two or more nodes are same, nodes are joined by link list i.e. second node is referenced by first node and third by second and so on.

9.    If key given already exist in HashMap, the value is replaced with new value.

10.                              hash code of null key is 0.

11.  When getting an object with its key, the linked list is traversed until the key matches or null is found on next field.

## Internal working of HashSet

**HashSet** uses HashMap internally to store it’s objects. The elements you add into HashSet are stored as **keys** of this HashMap object. The value associated with those keys will be a **constant**.

Every constructor of HashSet class internally creates one HashMap object. You can check this in the source code of HashSet class in JDK installation directory.

You can notice that each and every constructor internally creates one new HashMap object.

//Constructor - 1

**public** HashSet()

{

map = **new** HashMap<>(); //Creating internally backing HashMap object

}

//Constructor - 2

**public** HashSet(Collection<? **extends** E> c)

{

map = **new** HashMap<>(Math.max((**int**) (c.size()/.75f) + 1, 16)); //Creating internally backing HashMap object

addAll(c);

}

//Constructor - 3

**public** HashSet(**int** initialCapacity, **float** loadFactor)

{

map = **new** HashMap<>(initialCapacity, loadFactor); //Creating internally backing HashMap object

}

//Constructor - 4

**public** HashSet(**int** initialCapacity)

{

map = **new** HashMap<>(initialCapacity); //Creating internally backing HashMap object

}

**How HashSet Works Internally In Java?**

Whenever you insert an element into HashSet using add() method, it actually creates an entry in the internally backing HashMap object with element you have specified as it’s key and constant called “PRESENT” as it’s value. This “PRESENT” is defined in the HashSet class as below.

// Dummy value to associate with an Object in the backing Map

**private** **static** **final** Object PRESENT = **new** Object();

Let’s have a look at **add()** method of HashSet class.

**public** **boolean** add(E e)

{

**return** map.put(e, PRESENT)==**null**;

}

You can notice that, add() method of HashSet class internally calls put() method of backing HashMap object by passing the element you have specified as a key and constant “PRESENT” as it’s value.

**remove()** method also works in the same manner.

|  |  |
| --- | --- |
|  | **public** **boolean** remove(Object o)  {  **return** map.remove(o)==PRESENT;  }  If map.put(key, value) returns null, then the statement “map.put(e, PRESENT) == null” will return true and  element is added to the HashSet(internally HashMap).  If map.put(key, value) returns old value of the key, then the statement “map.put(e, PRESENT) == null” will  return false and element is not added to the HashSet(internally HashMap). |

Let’s see one example of HashSet and how it maintains HashMap internally.

**public** **class** HashSetExample

{

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

{

//Creating One HashSet object

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

//Adding elements to HashSet

set.add("RED");

set.add("GREEN");

set.add("BLUE");

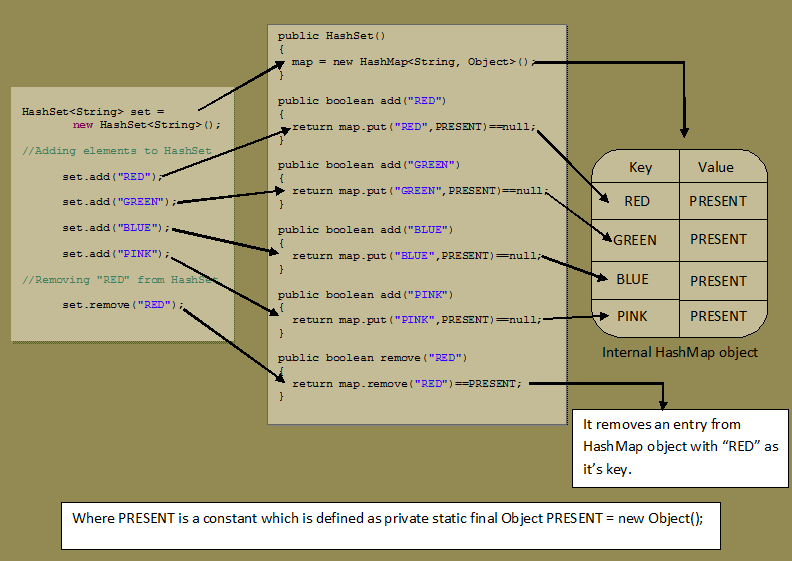
set.add("PINK");

//Removing "RED" from HashSet

set.remove("RED");

}

}



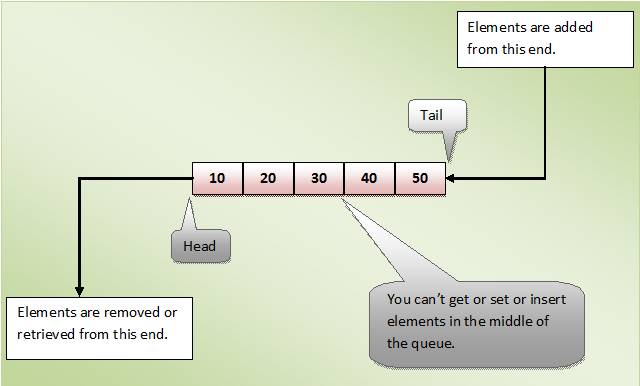
## **queue interface and ReadonlyCollection**

The Queue Interface:

The Queue Interface extends Collection interface. It defines queue data structure which is normally **First-In-First-Out**.

Queue is a data structure in which elements are added from one end and elements are deleted from another end.

But, exception being the Priority Queue in which elements are removed from one end, but elements are added according to the order defined by the specified comparator.



Properties Of Queue :

* **Null** elements are not allowed in the queue. If you try to insert null object into the queue, it throws NullPointerException.
* Queue can have **duplicate** elements.
* Unlike a normal list, queue is **not random access**. i.e you can’t set or insert or get elements at an arbitrary positions.
* In most of cases, elements are inserted at one end called **tail** of the queue and elements are removed or retrieved from another end called **head** of the queue.
* In the Queue Interface, there are two methods to obtain and remove the elements from the head of the queue. They are **poll()** and **remove()**. The difference between them is, poll() returns null if the queue is empty and remove() throws an exception if the queue is empty.
* There are two methods in the Queue interface to obtain the elements but don’t remove. They are **peek()** and **element()**. peek() returns null if the queue is empty and element() throws an exception if the queue is empty.

Methods Of Queue Interface:

|  |  |  |
| --- | --- | --- |
| **Operation** | **Throws An Exception If operation is not possible** | **Returns null or false if operation is not possible** |
| Add an element to the queue. | add() | offer() |
| Retrieve an element from the head of the queue without remove | element() | peek() |
| Retrieve And Remove an element from the head of the queue. | remove() | poll() |

[**The PriorityQueue Class**](https://javaconceptoftheday.com/java-collection-framework-priorityqueue-class/)

The **PriorityQueue** is a queue in which elements are ordered according to specified Comparator. You have to specify this Comparator while creating a PriorityQueue itsel. If no Comparator is specified, elements will be placed in their natural order. The PriorityQueue is a special type of queue because it is not a **First-In-First-Out** (FIFO) as in the normal queues. But, elements are placed according to supplied Comaparator.

Properties Of PriorityQueue Class :

* The PriorityQueue can have **duplicate** elements but can not have **null** elements.
* Elements in the PriorityQueue are ordered according to specified **Comparator**. If Comparator is not supplied, elements will be placed in their natural order.
* The default initial capacity of PriorityQueue is **11**. The PriorityQueue is **unbounded**. That means the capacity of the PriorityQueue increases automatically if the size exceeds capacity. But, how it grows is not specified.
* All elements of the PriorityQueue must be of **Comparable type**. Otherwise ClassCastException will be thrown at run time.
* The head element of the PriorityQueue is always the least element and tail element is always the largest element according to specified Comparator.
* You can retrieve the Comparator used to order the elements of the PriorityQueue using **comparator()** method.
* PriorityQueue is not a thread safe.

Examples to use Queue methods:

Eg 1:

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

            PriorityQueue<Integer> pQueue = **new** PriorityQueue<Integer>();

            System.out.println(pQueue.remove());      // Exception in thread "main" java.util.NoSuchElementException

             //System.out.println(pQueue. element());   // Exception in thread "main" java.util.NoSuchElementException

// System.out.println(pQueue.poll());                       //null

// System.out.println(pQueue.poll());                       //null

}

Eg 2: PriorityQueue Example With Default Comparator :

In this example, we create a PriorityQueue of Integers without supplying a Comparator So elements will be ordered in natural ascending order.

**public** **class** PriorityQueueExample {

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

            PriorityQueue<Integer> pQueue = **new** PriorityQueue<Integer>();

            pQueue.offer(21);       // either you can use add() also.

            pQueue.offer(17);

            pQueue.offer(37);

            pQueue.offer(41);

            pQueue.offer(9);

            pQueue.offer(67);

            pQueue.offer(31);

            //System.out.println(pQueue); [9, 17, 31, 41, 21, 67, 37]

            System.***out***.println(pQueue.poll());     //Output : 9

            System.***out***.println(pQueue.poll());     //Output : 17

            System.***out***.println(pQueue.poll());     //Output : 21

            System.***out***.println(pQueue.poll());     //Output : 31

            System.***out***.println(pQueue.poll());     //Output : 37

            System.***out***.println(pQueue.poll());     //Output : 41

            System.***out***.println(pQueue.poll());     //Output : 67

            //*poll() method removes the head of the queue.* peek() -> Retrieve an element from the head of the queue but not removes

            // *head element of the PriorityQueue always will be the least element.*

      }

}

**Eg 3: PriorityQueue Example With Customized Comparator :**

We try to create a PriorityQueue of ‘**Employee**‘ objects ordered in the ascending order of their salaries. That means head element always will be an ‘**Employee**‘ object with lowest salary.

//Employee Class

**class** Employee

{

String name;

**int** salary;

//Constructor Of Employee

**public** Employee(String name, **int** salary)

{

**this**.name = name;

**this**.salary = salary;

}

@Override

**public** String toString()

{

**return** name+" : "+salary;

}

}

//MyComparator Class

**class** MyComparator **implements** Comparator<Employee>

{

@Override

**public** **int** compare(Employee e1, Employee e2)

{

**return** e1.salary - e2.salary;

}

}

**public** **class** PriorityQueueExample

{

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

{

//Instantiating MyComaparator

MyComparator comparator = **new** MyComparator();

//Creating PriorityQueue of Employee objects with MyComparator as Comparator

PriorityQueue<Employee> pQueue = **new** PriorityQueue<Employee>(7, comparator);

//Adding Employee objects to pQueue

pQueue.offer(**new** Employee("AAA", 15000));

pQueue.offer(**new** Employee("BBB", 12000));

pQueue.offer(**new** Employee("CCC", 7500));

pQueue.offer(**new** Employee("DDD", 17500));

pQueue.offer(**new** Employee("EEE", 21500));

pQueue.offer(**new** Employee("FFF", 29000));

pQueue.offer(**new** Employee("GGG", 14300));

//Removing the head elements

System.out.println(pQueue.poll()); //Output --> CCC : 7500

System.out.println(pQueue.poll()); //Output --> BBB : 12000

System.out.println(pQueue.poll()); //Output --> GGG : 14300

System.out.println(pQueue.poll()); //Output --> AAA : 15000

System.out.println(pQueue.poll()); //Output --> DDD : 17500

System.out.println(pQueue.poll()); //Output --> EEE : 21500

System.out.println(pQueue.poll()); //Output --> FFF : 29000

}

}

+++++++++++++++++++++++++++++++++++++++++++++++

[**The Deque Interface**](https://javaconceptoftheday.com/collection-framework-deque-interface/)

The java.util.Deque interface is a subtype of the **[java.util.Queue](http://tutorials.jenkov.com/java-collections/queue.html)** interface

The Deque is the short name for “**Double Ended Queue**“, Means It allows  insertion and removal of elements from both the ends.

The main advantage of Deque is that you can use it as both **Queue** (FIFO) as well as **Stack** (LIFO).

examples of how to create a Deque instance:

Deque dequeA = new LinkedList();

Deque dequeB = new ArrayDeque();

Adding and Accessing Elements :

Deque dequeA = new LinkedList();

dequeA.add     ("element 1"); //add element at tail

dequeA.addFirst("element 2"); //add element at head

dequeA.addLast ("element 3"); //add element at tail

Eg 1:­­­­

## **immuatable class**

Immutable objects are like constants. You can’t modify them once they are created. The object’s fields are initialized only once inside the constructor and never change again. They are final in nature.

immutability helps in **concurrency**: **immutable objects can safely be shared among multiple threads**,

**Adv:**

* **Thread Safe** - because the state cannot be changed, no synchronization is required.
* Writing in an immutable style can lead to more **robust** **code**.

**Ref:**

https://dzone.com/articles/how-to-create-an-immutable-class-in-java

**Usage of Immutable Classes**

Nowadays, the *“must-have”* specification for every software application is to be **distributed and multi-threaded**—multi-threaded applications always cause headaches for developers since developers are required to protect the state of their objects from concurrent modifications of several threads at the same time, for this purpose, developers normally use the *Synchronized*blocks whenever they modify the state of an object.

With immutable classes, states are never modified; every modification of a state results in a new instance, hence each thread would use a different instance and developers wouldn’t worry about concurrent modifications.

Some Popular Immutable Classes:

**String,**wrapper classes like: **Integer, Float, Boolean** … these classes don’t modify their state, however they create a new instance each time you try to modify them.

**3. How Do We Create an Immutable Class?**

In order to create an immutable class, you should follow the below steps:

1.  Make your class **final,**so that no other classes can extend it.

2.  Make all your fields **private** and **final,**so that they’re initialized only once inside the constructor and never modified afterward.

3.  Don’t expose setter methods. **Only** **getters**.

4.  When exposing methods which modify the state of the class, you must always return a new instance of the class.

5.  If the class holds a mutable object:

o    Inside the constructor, make sure to use a **clone copy** of the passed argument and never set your mutable field to the real instance passed through constructor, this is to prevent the clients who pass the object from modifying it afterwards.

o    Make sure to always return a clone copy of the field and never return the real object instance.

 Eg:

public final class ImmutableStudent {

    private final int id;

    private final String name;

    public ImmutableStudent(int id, String name) {

        this.id = id;

        this.name = name;

    }

//only getters

    public int getId() {

        return id;

    }

    public String getName() {

        return name;

    }

}

**public** **class** TestImmutable {

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

ImmutableStudent t=**new** ImmutableStudent(101, "hcl");

System.***out***.println(t.getId()); //101

System.***out***.println(t.getName()); //hcl

}

}

Passing Mutable Objects to Immutable Class:

Now, let’s complicate our example a bit, we create a mutable class called **Age**and add it as a field to **ImmutableStudent:**

public class Age {

    private int day;

    private int month;

    private int year;

//setters and getters

public final class ImmutableStudent {

    private final int id;

    private final String name;

    private final Age age;

    public ImmutableStudent(int id, String name, Age age) {

    this.name = name;

    this.id = id;

    this.age = age;

    }

// declare only getters

    public int getId() {

    return id;

    }

    public String getName() {

    return name;

    }

    public Age getAge() {

    return age;

    }

}

Let’s create a simple test class and verify that **ImmutableStudent** is no more immutable:

 public static void main(String[] args) {

    Age age = new Age();

    age.setDay(1);

    age.setMonth(1);

    age.setYear(1992);

    ImmutableStudent student = new ImmutableStudent(1, "Alex", age);

    System.out.println("Alex age year before modification = " + student.getAge().getYear());

    age.setYear(1993);

    System.out.println("Alex age year after modification = " + student.getAge().getYear());

}

Output proved : Age class not immutable

Alex age year before modification = 1992

Alex age year after modification = 1993

In order to fix this and make our class again immutable, we follow step **#5** . So we modify the constructor in order to clone the passed argument of **Age** and use a clone instance of it.

public ImmutableStudent(int id, String name, Age age) {

    this.name = name;

    this.id = id;

    Age cloneAge = new Age();

    cloneAge.setDay(age.getDay());

    cloneAge.setMonth(age.getMonth());

    cloneAge.setYear(age.getYear());

    this.age = cloneAge;

}

Output:

Alex age year before modification = 1992

Alex age year after modification = 1992

As you see now, the age of **Alex** is never affected after construction and our class is back to immutable.

**#onemore twist here:**

However, our class still has a leak and is not fully immutable, let’s take the following test scenario:

public static void main(String[] args) {

    Age age = new Age();

    age.setDay(1);

    age.setMonth(1);

    age.setYear(1992);

    ImmutableStudent student = new ImmutableStudent(1, "Alex", age);

    System.out.println("Alex age year before modification = " + student.getAge().getYear());

    student.getAge().setYear(1993);                                                              ß

    System.out.println("Alex age year after modification = " + student.getAge().getYear());

}

Output:

Alex age year before modification = 1992

Alex age year after modification = 1993

Again according to step **#4**, when returning mutable fields from immutable object, you should return a clone instance of them and not the real instance of the field.

So we modify ***getAge()***in order to return a clone of the object’s age:

public Age getAge() {

    Age cloneAge = new Age();

    cloneAge.setDay(this.age.getDay());

    cloneAge.setMonth(this.age.getMonth());

    cloneAge.setYear(this.age.getYear());

    return cloneAge;

}

Now the class becomes fully immutable and provides no way or method for other objects to modify its state.

 Alex age year before modification = 1992

Alex age year after modification = 1992

## **clonning**

**What Is Cloning?**

**Cloning** is a process of creating an exact copy of an existing object in the memory. In java, **clone()** method of **java.lang.Object** class is used for cloning process. The objects which implement **Cloneable interface** are only eligible for cloning process. Cloneable interface is a [marker interface](https://javaconceptoftheday.com/marker-interface-java/) .

**Shallow Copy In Java :**

The default version of clone() method creates the shallow copy of an object. The shallow copy of an object will have exact copy of all the fields of original object. If original object has any references to other objects as fields, then only references of those objects are copied into clone object, copy of those objects are not created. That means any changes made to those objects through clone object will be reflected in original object or vice-versa. Shallow copy is not 100% independent of original object.

**class** Course

{

String subject1;

String subject2;

String subject3;

**public** Course(String sub1, String sub2, String sub3)

{

**this**.subject1 = sub1;

**this**.subject2 = sub2;

**this**.subject3 = sub3;

}

}

**class** Student **implements** Cloneable

{

**int** id;

String name;

Course course;

**public** Student(**int** id, String name, Course course)

{

**this**.id = id;

**this**.name = name;

**this**.course = course;

}

//Default version of clone() method. It creates shallow copy of an object.

**protected** Object clone() **throws** CloneNotSupportedException

{

**return** **super**.clone();

}

}

**public** **class** ShallowCopyInJava

{

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

{

Course science = **new** Course("Physics", "Chemistry", "Biology");

Student student1 = **new** Student(111, "John", science);

Student student2 = **null**;

**try**

{

//Creating a clone of student1 and assigning it to student2

student2 = (Student) student1.clone();

}

**catch** (CloneNotSupportedException e)

{

e.printStackTrace();

}

//Printing the subject3 of 'student1'

System.out.println(student1.course.subject3); //Output : Biology

//Changing the subject3 of 'student2'

student2.course.subject3 = "Maths";

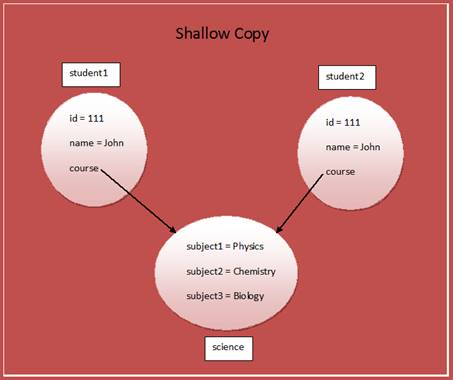
//This change will be reflected in original student 'student1'

System.out.println(student1.course.subject3); //Output : Maths

}

}

In the above example, ‘**student1**‘ is an object of ‘**Student**‘ class which has three fields – **id**, **name** and **course**. ‘**course**‘ is a reference variable pointing to a ‘**Course**‘ type object. Clone of ‘**student1**‘ is created by calling clone method on it and assigned it to ‘**student2**‘. As default version of clone method creates the shallow copy, the ‘**course**‘ field of both ‘**student1**‘ and ‘**student2**‘ will be pointing to same ‘**Course**‘ object. So, any changes made to this object through ‘**student2**‘ will be reflected in ‘**student1**‘ or vice-versa.



Deep Copy In Java**:**

Deep copy of an object will have exact copy of all the fields of original object just like shallow copy. But in additional, if original object has any references to other objects as fields, then copy of those objects are also created by calling clone() method on them. They will be 100% independent of each other. Any changes made to clone object will not be reflected in original object or vice-versa.

To create a deep copy of an object, you have to override the clone() method as demonstrated in the below example.

**class** Course **implements** Cloneable

{

String subject1;

String subject2;

String subject3;

**public** Course(String sub1, String sub2, String sub3)

{

**this**.subject1 = sub1;

**this**.subject2 = sub2;

**this**.subject3 = sub3;

}

**protected** Object clone() **throws** CloneNotSupportedException

{

**return** **super**.clone();

}

}

**class** Student **implements** Cloneable

{

**int** id;

String name;

Course course;

**public** Student(**int** id, String name, Course course)

{

**this**.id = id;

**this**.name = name;

**this**.course = course;

}

//Overriding clone() method to create a deep copy of an object.

**protected** Object clone() **throws** CloneNotSupportedException

{

Student student = (Student) **super**.clone();

student.course = (Course) course.clone();

**return** student;

}

}

**public** **class** DeepCopyInJava

{

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

{

Course science = **new** Course("Physics", "Chemistry", "Biology");

Student student1 = **new** Student(111, "John", science);

Student student2 = **null**;

**try**

{

//Creating a clone of student1 and assigning it to student2

student2 = (Student) student1.clone();

}

**catch** (CloneNotSupportedException e)

{

e.printStackTrace();

}

//Printing the subject3 of 'student1'

System.out.println(student1.course.subject3); //Output : Biology

//Changing the subject3 of 'student2'

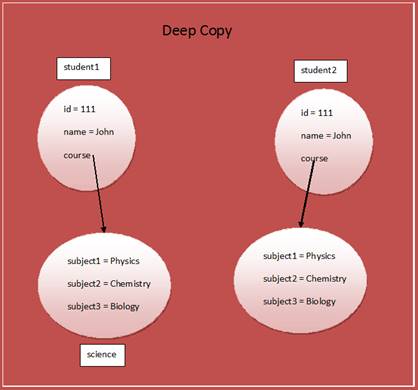
student2.course.subject3 = "Maths";

//This change will not be reflected in original student 'student1'

System.out.println(student1.course.subject3); //Output : Biology

}

}



icpdf

|  |  |
| --- | --- |
| **Shallow Copy** | **Deep Copy** |
| Shallow copy is not 100% **independent** of original object. | Deep  copy is  100% independent of original object. |
| Any **changes** made to cloned object will be reflected in original object or vice versa. | Any changes made to cloned object will not be reflected in original object or vice versa. |
| **Default** version of clone method creates the shallow copy of an object. | To create the deep copy of an object, you have to override clone method. |
| Shallow copy is preferred if an object has only **primitive** fields. | Deep copy is preferred if an object has references to other objects as fields. |
| Shallow copy is **fast** and also less expensive. | Deep copy is slow and very expensive. |

## **enum**

1)    Even though it is not mandatory to declare enum constants with **UPPERCASE** letters, but it is the best practice to do so.

enum Grade

{

    FIRST,    //Good One

    second;   //Bad One

}

* Duplicate enum constants are not allowed.

enum Directions

{

    NORTH, NORTH, SOUTH, EAST, WEST;  //Compile Time Error : Duplicate Constants

}

o    You can’t instantiate enum types using ‘new’ operator.

enum Directions

{

    NORTH, SOUTH, WEST, EAST;

    private Directions()

    {

        System.out.println(1);

    }

}

public class MainClass

{

    public static void main(String[] args)

    {

        Directions d1 = new Directions();       // Compile Time Error

    }

}

* Every constant of enum is public, static and final by default. As every constant is static, they can be accessed directly using enum name.

enum enums

{

    A, B, C;

}

public class EnumsExample

{

    public static void main(String[] args)

    {

        enums en = enums.A;   //accessing constant A through enum name

        enums en1 = enums.B;  //accessing constant B through enum name

        enums en2 = enums.C;  //accessing constant C through enum name

    }

}

2)    Enum types like classes can have **fields**, **constructors** and **methods** along with enum constants.

enum EnumClass

{

    A, B, C;       //Enum constants

    int i;      //Enums can have fields

    private EnumClass()

    {

        // Enums can have constructors

    }

    void methodOfEnum()

    {

        //Enums can have methods

    }

}

3)    Enum constructors are **private** by default. Only private constructors are allowed in enum types. That’s why you can’t instantiate enum types using new operator.

enum Directions

{

    SOUTH, NORTH, EAST, WEST;

    private Directions()

    {

        //Only private constructors are allowed

    }

    public Directions(int i)

    {

        //Compile time error, public constructors are not allowed

    }

}

5)    Enum constants must be declared first ahead of fields, constructors and methods (if any).

enum Enums

{

    int i;

    A, B, C;    //Compile time error, enum constants must be declared first.

}

6)    All enum types extend **java.lang.Enum** class by default. As multiple inheritance is not supported in java, they can’t extend any other classes.

7)    Enum types can implement any number of interfaces.

interface InterfaceOne

{

    abstract void methodOfInterfaceOne();

}

interface InterfaceTwo

{

    abstract void methodOfInterfaceTwo();

}

enum Enums implements InterfaceOne, InterfaceTwo

{

    A, B, C;

    public void methodOfInterfaceOne()

    {

        System.out.println("Method Of InterfaceOne");

    }

    public void methodOfInterfaceTwo()

    {

        System.out.println("Method Of InterfaceTwo");

    }

}

8)    Enum constants can have their own body called **Constant Specific Body**. In that body, you can define fields and methods. But, these methods and fields are visible within the Constant Specific Body in which they are defined.

enum Enums

{

    ONE

    {

        String s = "ONE";

        void methodOfONE()

        {

            System.out.println(s);

        }

    },

    TWO

    {

        String s = "TWO";

        void methodOfTWO()

        {

            System.out.println(s);

        }

    };

    void commonMethod()

    {

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

    }

}

public class MainClass

{

        public static void main(String[] args)

        {

            Enums.ONE.commonMethod();

            Enums.TWO.commonMethod();

            Enums.ONE.methodOfONE();     //Complie time error, methodOfONE() not visible outisde the ONE's body

            Enums.TWO.methodOfTWO();     //Complie time error, methodOfTWO() not visible outisde the TWO's body

        }

}

* Enum constants can override generalized method defined in the enum body.

enum enums

{

    FIRST

    {

        @Override

        void commonMethod()

        {

            System.out.println("Common method Overridden in FIRST");

        }

    },

    SECOND

    {

        @Override

        void commonMethod()

        {

            System.out.println("Common method Overridden in SECOND");

        }

    },

    THIRD

    {

        @Override

        void commonMethod()

        {

            System.out.println("Common method Overridden in THIRD");

        }

    };

    void commonMethod()

    {

        System.out.println("Generalized method, Common to all constants");

    }

}

public class EnumsExample

{

    public static void main(String[] args)

    {

        enums.FIRST.commonMethod();     //Output : Common method Overridden in FIRST

        enums.SECOND.commonMethod();    //Output : Common method Overridden in SECOND

        enums.THIRD.commonMethod();     //Output : Common method Overridden in THIRD

    }

}

* Enum can have abstract method declared in it’s body provided each enum constants must implement it.

enum enums

{

    FIRST

    {

        @Override

        void abstractMethod()

        {

            //Abstract Method Implemented

        }

    },

    SECOND

    {

        @Override

        void abstractMethod()

        {

            //Abstract Method Implemented

        }

    },

    THIRD

    {

        @Override

        void abstractMethod()

        {

            //Abstract Method Implemented

        }

    };

    abstract void abstractMethod();

}

11) Enum types can have any number of **static initialization blocks** as well as **instance initialization blocks**.

enum Enums

{

    A, B, C;

    static

    {

        System.out.println("Static Initialization Block");

    }

    {

        System.out.println("Instance Initialization Block");

    }

}

12) As java.lang.Enum class implements Comparable and Serializable interface, all enum types are **Comparable** and **Serializable** by default.

13) We can compare the enum constants using **“==”** operator.

enum Enums

{

    ONE, TWO, THREE, FOUR

}

public class MainClass

{

    public static void main(String[] args)

    {

        System.out.println(Enums.FOUR == Enums.FOUR);     //Output : true

        Enums myEnum = Enums.ONE;

        System.out.println(myEnum == Enums.ONE);        //Output : true

    }

}

14) You can retrieve the enum constants of any enum type using **values()** method. values() method returns an array of enum constants.

enum Directions

{

    SOUTH, NORTH, EAST, WEST

}

public class MainClass

{

    public static void main(String[] args)

    {

        Directions[] directions = Directions.values();

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

        {

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

        }

    }

}

15) Enums provide **type-safety** during compilation. That means you will get compile time error if you try to assign any other values other than the specified enum constants.

enum Directions

{

    SOUTH, NORTH, EAST, WEST

}

public class MainClass

{

    public static void main(String[] args)

    {

        Directions direction1 = RIGHT;    //Compile time error

        //You can't assign any other values other then SOUTH, NORTH, EAST, WEST

    }

}

16) You can define enum types outside a class or inside a class but not inside a method or block. (If declared inside a class, they are static by default and can be accessed directly by Class name.)

enum EnumOne

{

    A, B, C

    //Enum outside the class

}

public class MainClass

{

    enum EnumTwo

    {

        A, B, C

        //Enum inside the class

    }

    public static void main(String[] args)

    {

        enum EnumThree

        {

            A, B, C

            //Compile Time Error, can't define enum inside a method

        }

    }

}

17) **ordinal()** method is used get the order of an enum constant in an enum type.

enum Directions

{

    SOUTH, NORTH, EAST, WEST

}

public class MainClass

{

    public static void main(String[] args)

    {

        System.out.println(Directions.EAST.ordinal());    //Output : 2

        System.out.println(Directions.NORTH.ordinal());   //Output : 1

    }

}

4) Enum constants are created only once for the whole execution. All enum constants are created when you refer any enum constant first time in your code. While creating each enum constant, corresponding constructor is called.

public class EnumsExample {

      enum Enums

      {

          A, B(10), C("ccc", 20);

          //No-arg private constructor

          private Enums()

          {

              System.out.println(1);

          }

          //Private constructor taking one argument

          private Enums(int i)

          {

              System.out.println(2);

          }

          //Private constructor taking two arguments

          private Enums(String s, int j)

          {

              System.out.println(3);

          }

      }

      public static void main(String[] args)

    {

            Enums en = Enums.B;   //All enum constants are created while executing this statement.

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

              //While creating each enum constant, corresponding constructor is called

              Enums en2 = Enums.C;   //No enum constant is created here.

              System.out.println("en2 "+en2);

              Enums en3 = Enums.A;   //No enum constant is created here.

              System.out.println("en3 "+en3);

    }

}

                Output::

1

2

3

en B

en2 C

en3 A

## **Garbage Collection**

 Garbage Collection In **Java :**

It’s a process of destroying unused objects.

**Adv**: It make java as memory efficient.

Q) when object become eligible for garbage collection?

1.When object has null referance

2.When object is unreachable

Eg 1:

Employee e = **new** Employee();

E=**null**;

Eg 2:

Employee e1 = **new** Employee();

Employee e2 = **new** Employee();

e1=e2; //e1 become eligible for GC (unreachable)

1.  Whenever you run a java program, JVM creates three threads. 1) main thread   2) Thread Scheduler   3) Garbage Collector Thread.

     In these three threads, main thread is a user thread and remaining two are daemon threads which run in background. (what is daemon thread?)

2.  The task of garbage collector thread is to sweep out abandoned objects from the heap memory.

3.  Garbage collector thread before sweeping out an abandoned object, it calls finalize() method of that object.

4.  Garbage collector thread calls finalize() method only once for one object.

After finalize() method is executed, object is destroyed from the memory.

**finalize() method in Java:**

finalize() method is a **protected and non-static** method of **java.lang.Object** class.

This method will be available in all objects you create in java. This method is used to perform some final operations or **clean up operations** on an object before it is removed from the memory.

**protected** **void** finalize() **throws** Throwable

{

//Keep some resource closing operations here

}

## **SIB and IIB**

[**SIB – Static Initialization Block, Static Variables And Static Methods**](https://javaconceptoftheday.com/static-members-java/)

**·**Static variables, Static Initialization Block and Static Methods – these all are static components or static members of a class.

**·**Static Initialization Block is used to initialize only static variables. It is a block without a name enclosed within { }.

**·**These static members are stored inside the Class Memory. To access static members, you need not to create objects. Directly you can access them with class name.

**·       SIBs are not stored in the heap memory. They just come to stack, execute their tasks and leaves the memory.**

·       **SIB will execute before the main method.**

static

{

     //Set Of Statements

}

Eg:

**public** **class** StaticCompounent {

**static** **int** *staticVariable*;

**static**

     {

          System.***out***.println("StaticComponents SIB");

*staticVariable* = 10;

     }

**static** **void** staticMethod()

     {

          System.***out***.println("From StaticMethod");

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

     }

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

     {

         //Static Members directly accessed with Class Name

       StaticCompounent.*staticVariable* = 20;

       StaticCompounent.*staticMethod*();

     }

}

Output:

StaticComponents SIB

From StaticMethod

20

[**IIB – Instance Initialization Block**](https://javaconceptoftheday.com/instance-initialization-block-in-java/) :

**·**IIB is used to initialize instance variables or non-static variables.

**·**This is a block with no name and enclosed within {}. The syntax for IIB is,

{

      //Set Of Statements, mostly initialization statements

}

**Important Note :** IIB blocks will not be called from the constructor in which **this()** statement is written as a first statement, in such case they will be executed only in second constructor.

Eg:

class A

{

     int i;

     {

          System.out.println("First IIB Block");

     }

     {

          System.out.println("Second IIB Block");

     }

     A(int j)

     {

          this();

          System.out.println("First Constructor");

     }

     A()

     {

          System.out.println("Second Constructor");

     }

}

class MainClass

{

     public static void main(String[] args)

     {

          A a = new A(50);

     }

}

Output:

First IIB Block  
Second IIB Block  
Second Constructor  
First Constructor,

//if u comment ‘this keyword’, output would be ::

First IIB Block  
Second IIB Block  
First Constructor,

IIBs can also be written as,

class A

{

     int i = 10;

}

## **How to consume Rest API in java class**

**import** java.io.BufferedReader;

**import** java.io.DataOutputStream;

**import** java.io.InputStreamReader;

**import** java.net.HttpURLConnection;

**import** java.net.URL;

**public** **class** ConsumeRestApi {

**private** **final** String USER\_AGENT = "Mozilla/5.0";

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

            ConsumeRestApi http = **new** ConsumeRestApi();

            http.sendGet();

            //http.sendPost();

      }

      // HTTP GET request

**private** **void** sendGet() **throws** Exception {

            String url = "http://localhost:8080/orgnisation/all";

            URL u = **new** URL(url);

            HttpURLConnection con = (HttpURLConnection) u.openConnection();

            // optional default is GET

            con.setRequestMethod("GET");

            //add request header

            con.setRequestProperty("User-Agent", USER\_AGENT);

**int** responseCode = con.getResponseCode();

            String responseMessage = con.getResponseMessage();

            //String responseStatus = con.getResponseDtatus();

            System.***out***.println("Response Code : " + responseCode);

            System.***out***.println("Response message : " + responseMessage);

            //System.out.println("Response status : " + responseStatus);

            BufferedReader in = **new** BufferedReader(

**new** InputStreamReader(con.getInputStream()));

            String inputLine;

            StringBuffer response = **new** StringBuffer();

**while** ((inputLine = in.readLine()) != **null**) {

                  response.append(inputLine);

            }

            in.close();

            //print result

            System.***out***.println(response.toString());

      }

      /\*output::

      \*

       \* Response Code : 200

         Response message : null

        {

          "statusType": "OK",

          "entity": {

              "responseCode": 200,

              "message": "success",

              "orgResponses": [

                  {

                      "id": 1,

                      "org\_name": "brillio",

                      "employee": []

                  },

                  {

                      "id": 6,

                      "org\_name": "brillio5",

                      "employee": []

                  }

              ]

          },

          "entityType": "com.spring.boot.response.OrgResponseList",

          "status": 200,

          "metadata": {}

      }\*/

      // HTTP POST request

**private** **void** sendPost() **throws** Exception {

                        String url = "http://localhost:8080/orgnisation/findById";

                        URL u = **new** URL(url);

                        HttpURLConnection con = (HttpURLConnection) u.openConnection();

                        //add reuqest header

                        con.setRequestMethod("POST");

                        con.setRequestProperty("User-Agent", USER\_AGENT);

                        con.setRequestProperty("Accept-Language", "en-US,en;q=0.5");

                        String urlParameters ="id=1";

                        // Send post request

                        con.setDoOutput(**true**);

                        DataOutputStream wr = **new** DataOutputStream(con.getOutputStream());

                        wr.writeBytes(urlParameters);

                        wr.flush();

                        wr.close();

**int** responseCode = con.getResponseCode();

                        String responseMessage = con.getResponseMessage();

                        System.***out***.println("Response Code : " + responseCode);

                        System.***out***.println("Response Message : " + responseMessage);

                        BufferedReader in = **new** BufferedReader(

**new** InputStreamReader(con.getInputStream()));

                        String inputLine;

                        StringBuffer response = **new** StringBuffer();

**while** ((inputLine = in.readLine()) != **null**) {

                              response.append(inputLine);

                        }

                        in.close();

                        //print result

                        System.***out***.println(response.toString());

                  }

}

## **more java questions**

[**which collection should i use for faster search?**](https://stackoverflow.com/questions/30321169/performing-the-fastest-search-which-collection-should-i-use)

If you need fast access to elements using index, ArrayList should be choice.

If you need fast access to elements using a key, use HashMap.

what is serialization? Why we need it?

Ø  **Serialization in java** is a mechanism of writing the state of an object into a byte of stream. It is mainly used in Hibernate, RMI, JPA, EJB and JMS technologies.

The reverse operation of serialization is called **deserialization**.

**Serialization** is usually used When the **need** arises to send your data over network or stored in files.

**Serialization**is the translation of your **Java** object's values/states to bytes to send it over network or save it.

Why we use nested classes in java?

Ø  We use nested 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.

What is Enum?

Ø  **Enums** OR **Enum Types** OR **Enumeration types** are special data types which are used to represent similar kind of constants. For example, Months (JAN, FEB, MAR, APR . . . . . ), Days (MONDAY, TUESDAY, WEDNESDAY . . . .), Directions (NORTH, SOUTH, WEST, EAST).

What are the coding standards u followed in ur project?

                        1.1 Why Have Code Conventions

                          2 convinient file name

                          3. Beginning Comments

                          4. Package then Import Statements

                          5. Naming convention for class, constructor, var and methods.

                          6. single line comment n block comment

                          7. Simple Statements

                                                Each line should contain at most one statement. Example:

                                                argv++;         // Correct

                                                argc--;         // Correct

                                                argv++; argc--; // AVOID!

                          8.if (condition) //AVOID! THIS OMITS THE BRACES {}!

                                                statement;

                                                //avoid using above one.

                        9. use the braces properly

                                                if (a == b && c == d)     // AVOID!

                                                if ((a == b) && (c == d)) // RIGHT

                        10. put it frequently calling methods in a util class.n call whenever require.

                        11. Structure the packages properly. So that its eassy for identifying files n accessing

which rest your using?  //jersey

Which type Rest API you are using?

<!-- for Response Entity feature in controller class -->

<dependency>

<groupId>com.sun.jersey</groupId>

<artifactId>jersey-bundle</artifactId>

<version>1.17.1</version>

</dependency>

++++++++++++++++++++

What are the sorting technics do you know?  Know about each operation.   // bubble search, binary search, Lenear search etc

 Concurrency and parallelism?

What are the problems we face in concurrency application? Nope

Count down latch and cyclic barrier?

Locking APIs in Java?

how will u use join in hibernate?

how to improve performance issue?

how to implement third party resources?

which tool using for deployement? AWS , onestash

what is agile?  //it’s nothing but how to using JIRA only

++++++++++++++++++++++++

**Marker interface :**

An empty interface in Java is known as a marker interface i.e. it does not contain any methods or fields

**java.lang.Cloneable** and **java.io.Serializable** are examples of marker interfaces.

**package** com.hcl;

**import** java.util.Scanner;

**public** **class** Student **implements** Cloneable {

**int** age;

String name;

**public** Student (String name, **int** age){

**this**.age = age;

**this**.name = name;

}

**public** **void** display() {

System.***out***.println("Name of the student is: "+name);

System.***out***.println("Age of the student is: "+age);

}

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

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

System.***out***.println("Enter your name: ");

String name = sc.next();

System.***out***.println("Enter your age: ");

**int** age = sc.nextInt();

Student obj = **new** Student(name, age);

Student obj2 = (Student) obj.clone();

obj2.display();

}

}

/\*

Enter your name:

sudheer

Enter your age:

31

Name of the student is: sudheer

Age of the student is: 31

\*/

**Java Transient Keyword**

The **transient** keyword in **Java** is used to avoid serialization. If any object of a data structure is defined as a **transient** , then it will not be serialized.

Serialization is the ​process of converting an object into a byte stream.

Let's take an example, I have declared a class as Student, it has three data members id, name and age. If you serialize the object, all the values will be serialized but I don't want to serialize one value, e.g. age then we can declare the age data member as transient.

**//** Let's create a class with transient variable.

**import** java.io.Serializable;

**public** **class** Student **implements** Serializable {

**int** id;

String name;

**transient** **int** age;// Now it will not be serialized

**public** Student(**int** id, String name, **int** age) {

**this**.id = id;

**this**.name = name;

**this**.age = age;

}

}

//Now write the code to serialize the object.

**import** java.io.\*;

**class** PersistExample {

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

Student s1 = **new** Student(211, "ravi", 22);// creating object

// writing object into file

FileOutputStream f = **new** FileOutputStream("f.txt");

ObjectOutputStream out = **new** ObjectOutputStream(f);

out.writeObject(s1);

out.flush();

out.close();

f.close();

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

}

}

// success

// Now write the code for deserialization.

**import** java.io.\*;

**class** DePersist {

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

ObjectInputStream in = **new** ObjectInputStream(**new** FileInputStream("f.txt"));

Student s = (Student) in.readObject();

System.***out***.println(s.id + " " + s.name + " " + s.age);

in.close();

}

}

// 211 ravi 0

As you can see, printing age of the student returns 0 because value of age was not serialized.

ClassNotFoundException Vs NoClassDefFoundError

**ClassNotFoundException** and **NoClassDefFoundError** occur when a particular class is not found at runtime. However, they occur at different scenarios

**ClassNotFoundException** is an exception that occurs when you try to load a class at run time using **Class.forName()** or **loadClass()** methods and mentioned classes are not found in the classpath.

For example, the below program will throw ClassNotFoundException if the mentioned class **“oracle.jdbc.driver.OracleDriver”** is not found in the classpath.

**public** **class** MainClass {

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

{

**try**

{

Class.*forName*("oracle.jdbc.driver.OracleDriver");

}**catch** (ClassNotFoundException e)

{

e.printStackTrace();

}

}

}

**NoClassDefFoundError** is an error that occurs when a particular class is present at compile time, but was missing at run time.

**class** A

{

// some code

}

**public** **class** B

{

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

{

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

}

}

​ When you compile the above program, two .class files will be generated. One is **A.class** and another one is **B.class**. If you remove the **A.class** file and run the **B.class**file, Java Runtime System will throw NoClassDefFoundError.

In interface we can achieve multiple inheritance but in case abstrction not possible because we cant extend more then one class as a java rule.

++++++++++++++++++++

how will u find max n min element in hashSet/HashMap?

public String getMapKeyWithHighestValue(HashMap<String, Integer> map) {

    String keyWithHighestVal = "";

    // getting the maximum value in the Hashmap

    int maxValueInMap = (Collections.max(map.values()));

    //iterate through the map to get the key that corresponds to the maximum value in the Hashmap

    for (Map.Entry<String, Integer> entry : map.entrySet()) {  // Iterate through hashmap

        if (entry.getValue() == maxValueInMap) {

             keyWithHighestVal = entry.getKey();     // this is the key which has the max value

        }

    }

    return keyWithHighestVal;

}

++++++++++++++++++++

[Does the capacity of ArrayList decrease when we remove elements?](https://stackoverflow.com/questions/23831157/does-the-capacity-of-arraylist-decrease-when-we-remove-elements)

 It doesn't decrease this automatically. From the doc.

**public void trimToSize()**

**Size Vs Capacity in Java List**

What is  size?  The number of elements stored in the List .

What is  capacity ?  The current capacity of a List that means maximum number of elements it can hold currently.

+++++++++++++++++++++++

 Is it possible to add mutable object as a key in HashMap?

**What is volatile in java ? when to use? ? diff between volatile and synchronization?**

// <https://www.javatpoint.com/volatile-keyword-in-java#:~:text=Volatile%20keyword%20is%20used%20to,same%20time%20without%20any%20problem>.

Volatile keyword is used to modify the value of a variable by different threads. It is also used to make classes thread safe.

When to use:

* You can use a volatile variable if you want to read and write long and double variable automatically.
* It can be used as an alternative way of achieving synchronization in Java.

Diff :

|  |  |
| --- | --- |
| Volatile keyword is a field modifier. | Synchronized keyword modifies code blocks and methods. |
| The thread cannot be blocked for waiting in case of volatile. | Threads can be blocked for waiting in case of synchronized. |
| It improves thread performance. | Synchronized methods degrade the thread performance. |

**What is the meaning of Enum being type-safe in Java? (answer)**

hint: It means you cannot assign an instance of different Enum type to an Enum variable. e.g. if you have a variable like DayOfWeek day then you cannot assign it value from DayOfMonth enum.

27) **Difference between \*\*PATH\*\* and \*\*Classpath\*\* in Java? (answer)**

hint: PATH is used by the operating system while **Classpath is used by JVM to locate Java binary, e.g. JAR files or Class files**

 39) **Difference between Iterator and Enumeration in Java?**

Enumeration and Iterator are two interfaces in java.util package which are used to traverse over the elements of a Collection object

|  |  |
| --- | --- |
| **Enumeration** | **Iterator** |
| *Enumeration* is introduced in JDK 1.0 | *Iterator* is introduced from JDK 1.2 |
| *Enumeration* is used to traverse the legacy classes like *Vector*, *Stack* and *HashTable*. | *Iterator* is used to iterate most of the classes in the collection framework like *ArrayList*, *HashSet*, *HashMap*, *LinkedList* etc. |
| Methods : *hasMoreElements()*and *nextElement()* | Methods : *hasNext()*, *next()* and *remove()* |
| *Enumeration* is fail-safe in nature. | *Iterator* is fail-fast in nature. |
| *Enumeration* is not safe and secured due to it’s fail-safe nature. | *Iterator*is safer and secured than *Enumeration*.(because it does not allow other thread to modify the collection object while some thread is iterating over it and throws ConcurrentModificationException ) |

40)**Difference between java.util.Date and java.sql.Date in Java**

while java.util.Date represents both Date and Time information.

But java.sql.Date just represent DATE without time information

**Concurrency and parallelism?**

Concurrency is when two or more tasks can start, run, and complete in overlapping time periods. It doesn't necessarily mean they'll ever both be running at the same instant. For example, multitasking on a single-core machine.

Parallelism is when tasks literally run at the same time, e.g., on a multicore processor.

In simple words, concurrency is the ability to run several programs or several parts of a program in parallel. ... The backbone of java concurrency are threads.

**What is static binding and dynamic binding?**

When the type of object is determined at compile time is called static binding.

Eg: static polymorphism

When the type of object is determined at runtime time is called dynamic binding.

Eg: dynamic polymorphism

**what are object class methods in java?**

[**protected**](https://en.wikibooks.org/wiki/Java_Programming/Keywords/protected) [**void**](https://en.wikibooks.org/wiki/Java_Programming/Keywords/void) finalize() ,[**protected**](https://en.wikibooks.org/wiki/Java_Programming/Keywords/protected) Object clone() throws CloneNotSupportedException ;

[**int**](https://en.wikibooks.org/wiki/Java_Programming/Keywords/int) hashCode();[String](https://en.wikibooks.org/wiki/Java_Programming/API/java.lang.String) toString();[**void**](https://en.wikibooks.org/wiki/Java_Programming/Keywords/void) wait();[**void**](https://en.wikibooks.org/wiki/Java_Programming/Keywords/void) notify();[**void**](https://en.wikibooks.org/wiki/Java_Programming/Keywords/void) notifyAll();

## Association | Composition | Aggregation

**Association** refers to the relationship between multiple objects. It refers to how objects are related to each other and how they are using each other's functionality. Composition and aggregation are two types of association.

**Composition**

The composition is the strong type of association. An association is said to composition if an Object owns another object and another object cannot exist without the owner object. Consider the case of Human having a heart. Here Human object contains the heart and heart cannot exist without Human.

**Aggregation**

Aggregation is a weak association. An association is said to be aggregation if both Objects can exist independently. For example, Employee and Address

# Others :

## **Date API:**

### **DateFormatter:**

Date d = **new** Date(); // import java.util.Date;

SimpleDateFormat formatter = **new** SimpleDateFormat("yyyy-MM-dd"); //2021-10-10

//SimpleDateFormat formatter = new SimpleDateFormat("dd-MM-yyyy"); //10-10-2021

//SimpleDateFormat formatter = new SimpleDateFormat("yyyy-MM-dd HH:ss"); //2021-10-10 15:23

//SimpleDateFormat formatter = new SimpleDateFormat("yyyy/MM/dd HH:ss"); //2021/10/10 15:23

String formattedDate = formatter.format(d);

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

# Java 8 features:



<https://javaconceptoftheday.com/java-8-streams-beginners-guide/>

1) Lambda Expression

2) Functional Interfaces

3) Default methods and static methods

4) Predicates

5) Functions

6) Double colon operator (::) or method referance

7) Stream API

8) Date and Time API

Etc…..

## Lambda Expression :

is just an anonymous (nameless) function. That means the function which doesn’t have the name, return type and access modifiers.

Lambda Expression also known as anonymous functions or closures.

## functional interfaces :

If an interface contain only one abstract method, such type of interfaces are called functional interfaces and the method is called functional method or single abstract method (SAM).

Ex:

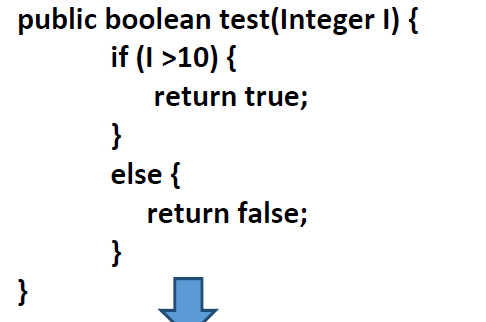
1) Runnable  It contains only run() method

2) Comparable  It contains only compareTo() method

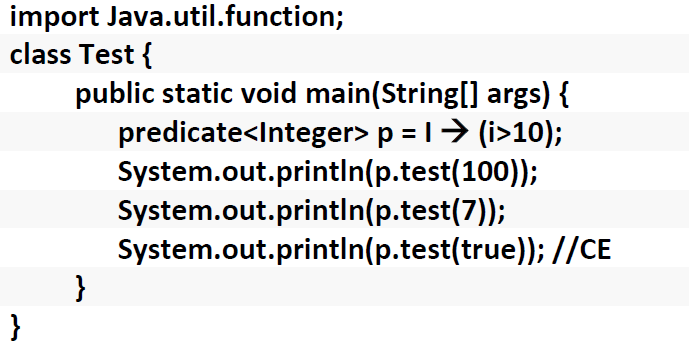
**Predicate function:**

* Predicate is a interface, containing only one method called **test**.
* It takes **1 arguments**, as input.
* And It always returns **Boolean** value.

Eg: without predicate:



Eg: with predicate:





**Function:**

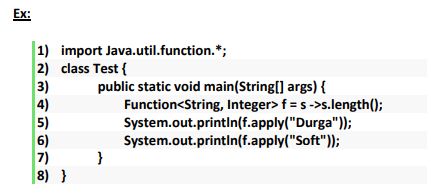
* Function is a interface, containing only one method called **apply**.
* It takes **2 arguments**, first one represent input type and second one represent return type
* And It returns any result type

Syntax :

Interface function(T,R){

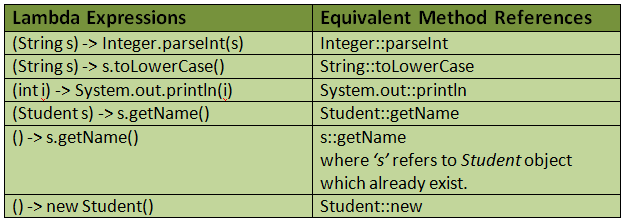
Public R apply (T t);

}



**Tell the Difference between predicate and function (min 3 points)?**

## Java 8 Method References :



+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++

## What is Streams? Why we need?

1. Streams are not the data structures:
2. We can’t add or remove elements from streams. Hence, they are not the data structures.
3. Stream consumes a source like collection or an array
4. And performs operations on it and produces the result

|  |  |
| --- | --- |
| Collections | Streams |
| Collections are mainly used to store and group the data. | Streams are mainly used to perform operations on data. |
| You can add or remove elements from collections. | You can’t add or remove elements from streams. |
| Collections have to be iterated externally. | Streams are internally iterated. |
| Collections can be traversed multiple times. | Streams are traversable only once. |
| Collections are eagerly constructed. | Streams are lazily constructed. |
| Ex : List, Set, Map… | Ex : filtering, mapping, matching… |

Collections are **eagerly** constructed i.e all the elements are computed at the beginning itself. But, streams are **lazily** constructed i.e intermediate operations are not evaluated until terminal operation is invoked.

List<Integer> numbers = Arrays.asList(4, 2, 8, 9, 5, 6, 7);

numbers.stream().filter(i -> i >= 5).limit(3).forEach(System.out::println);

//Here, not all numbers are evaluated.

//numbers are evaluated until 3 numbers >= 5 are found.

//Output :

//8

//9

//5

**Streams are traversable only once**

stream.forEach(System.out::println);

stream.forEach(System.out::println);

//Error : stream has already been operated upon or closed

**Short Circuiting Operations**

The operations which don’t need the whole stream to be processed to produce a result. For example – findFirst(), findAny(), limit() etc.

|  |  |  |
| --- | --- | --- |
|  | Intermediate Operations | Terminal Operations |
| 1 | Intermediate operations are declared between source and terminal operation | It’s declared at the end. |
| 2 | They can have pipeline of operations. | They can’t be pipelined of operation. |
| 3 | Intermediate operations are lazily loaded. | Terminal operations are eagerly loaded. |
| 4 | They don’t produce end result. | They produce end result. |
| 5 | Examples : filter(), map(), distinct(), sorted(), limit(), skip() | Examples : forEach(), toArray(), reduce(), collect(), min(), max(), count(), anyMatch(), allMatch(), noneMatch(), findFirst(), findAny() |

**Pipeline Of Operations**

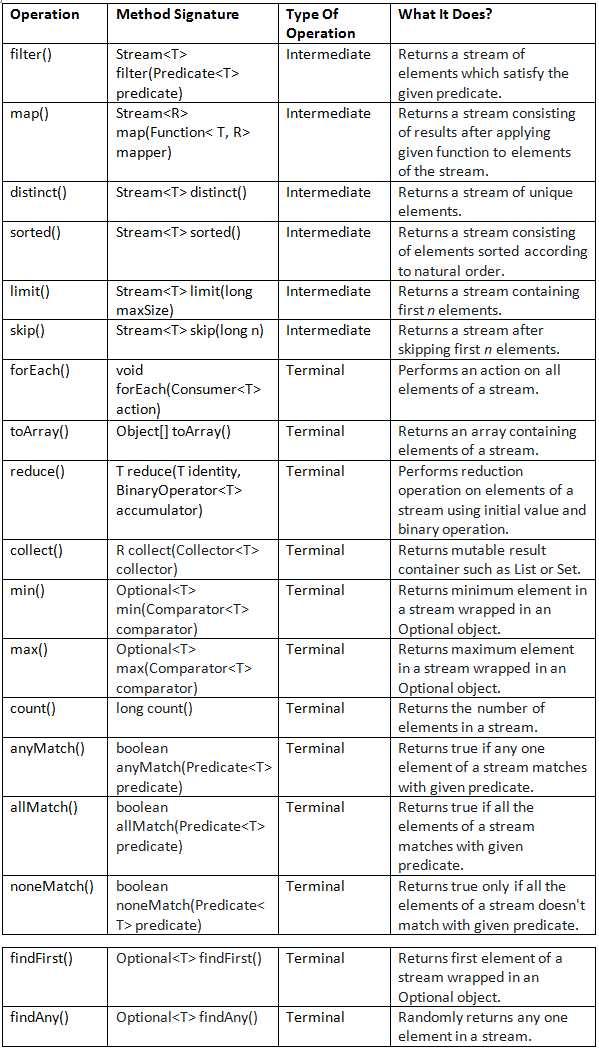
A pipeline of operations consists of three things – a source, one or more intermediate operations and a terminal operation.

Eg:

**long** l4=l1.stream().distinct().count();

l1.stream().filter((i) -> i > 10).distinct().forEach(System.***out***::println);

**Below table shows frequently used Stream methods with description.**

[](https://i0.wp.com/javaconceptoftheday.com/wp-content/uploads/2019/04/Java8Streams1.png?ssl=1)

ArrayList<Integer> l1=**new** ArrayList<Integer>();

l1.add(0);

l1.add(15);

l1.add(10);

l1.add(5);

l1.add(30);

l1.add(25);

l1.add(20);

l1.add(20);

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* Intermediate operations \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

// to print even numbers

**long** count=l1.stream().filter(i->i%2==0).count(); //to get the count number

l1.stream().filter(i->i%2==0).forEach(System.***out***::println);

//to save modified content in another list

l1.stream().map(i-> i+10).forEach(System.***out***::println);

List<Integer> l2=l1.stream().map(i-> i+10).collect(Collectors.*toList*());

//unique elements

l1.stream().distinct().forEach(System.***out***::println);

l1.stream().sorted().forEach(System.***out***::println);

//limit and skip

l1.stream().limit(2).forEach(System.***out***::println);

l1.stream().skip(2).forEach(System.***out***::println);

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* Terminal operations \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

l1.stream().forEach(System.***out***::println);

l1.stream().sorted().forEach(System.***out***::println); //natural sorting

List<Integer> l3 = l1.stream().sorted().collect(Collectors.*toList*());

System.***out***.println(l3);

//min and max only applicable with comparator interf

Comparator<Integer> comp=(i1,i2)->i1.compareTo(i2);

System.***out***.println(l3.stream().min(comp).get());

System.***out***.println(l3.stream().max(comp).get());

Long l=l1.stream().count();

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

//anyMatch(), allMatch(), noneMatch()

System.***out***.println(l1.stream().anyMatch((i)->i==5)); //true

System.***out***.println(l1.stream().allMatch((i)->i==5)); //false

System.***out***.println(l1.stream().noneMatch((i)->i==5)); //false

//findFirst(), findAny()

System.***out***.println(l1.stream().findFirst());

System.***out***.println(l1.stream().findAny());

+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++

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

names.add("David");

names.add("Johnson");

names.add("Samontika");

names.add("Brijesh");

names.add("John");

//Sorting the names according to natural order

names.stream().sorted().forEach(System.***out***::println);

// Output :

// Brijesh

// David

// John

// Johnson

// Samontika

//Sorting the names according to their length

names.stream().sorted((String s1, String s2) -> s1.length() - s2.length()).forEach(System.***out***::println);

// Output :

// John

// David

// Johnson

// Brijesh

// Samontika

//Returns length of each name

names.stream().map(String::length).forEach(System.***out***::println);

//Counting the names with length > 5

**long** noOfBigNames = names.stream().filter((String name) -> name.length() > 5).count();

System.***out***.println(noOfBigNames); // Output : 3

//Storing first 3 names in a mutable container

List<String> first3Names = names.stream().limit(3).collect(Collectors.*toList*());

System.***out***.println(first3Names);

// Output : [David, Johnson, Samontika]

**if**(names.stream().anyMatch((String name) -> name.length() == 5))

{

System.***out***.println("Yes... There is a name exist with 5 letters");

}

**if**(names.stream().allMatch((String name) -> name.length() > 3))

{

System.***out***.println("All are big names");

}

**if**(names.stream().noneMatch((String name) -> name.length() == 2))

{

System.***out***.println("There is no two letter name");

}

//output : There is no two letter name

**if**(names.stream().anyMatch((String name) -> name.equals(“John”)))

{

System.***out***.println("Yes... There is a name exist with John");

}

//Storing first 3 names in an array

Object[] streamArray = names.stream().limit(3).toArray();

System.***out***.println(Arrays.*toString*(streamArray));

// Output

// [David, Johnson, Samontika]

 Stream.collect()*Method*

collect() method is a terminal operation in Stream interface. It is a special case of reduction operation called mutable reduction operation because it returns mutable result container such as List, Set or Map according to supplied Collector.

Eg:

List<Integer> numbers = Arrays.*asList*(8, 2, 5, 7, 3, 6);

//collect() method returning List of OddNumbers

List<Integer> oddNumbers = numbers.stream().filter(i -> i%2 != 0).collect(Collectors.*toList*());

//OUTPUT : [5, 7, 3]

## stream.Collectors*Class*

java.util.stream.Collectors*Class*

accumulating elements into Collection, finding min, max, average, sum of elements etc. All the methods of Collectors class return Collector type which will be supplied to collect() method as an argument.



**Eg:**

**class** Student

{

**int** id;

String name;

String subject;

**double** percentage;

//setters getters

//constructor

//to string

}

List<Student> studentList = **new** ArrayList<Student>();

studentList.add(**new** Student("Paul", 11, "Economics", 78.9));

studentList.add(**new** Student("Zevin", 12, "Computer Science", 91.2));

studentList.add(**new** Student("Harish", 13, "History", 83.7));

studentList.add(**new** Student("Xiano", 14, "Literature", 71.5));

studentList.add(**new** Student("Soumya", 15, "Economics", 77.5));

studentList.add(**new** Student("Asif", 16, "Mathematics", 89.4));

studentList.add(**new** Student("Nihira", 17, "Computer Science", 84.6));

studentList.add(**new** Student("Mitshu", 18, "History", 73.5));

studentList.add(**new** Student("Vijay", 19, "Mathematics", 92.8));

studentList.add(**new** Student("Harry", 20, "History", 71.9));

//Collecting top 3 performing students into List

List<Student> top3Students = studentList.stream().sorted(Comparator.*comparingDouble*(Student::getPercentage).

reversed()).limit(3).collect(Collectors.*toList*());

System.***out***.println(top3Students);

//Output :[Vijay-19-Mathematics-92.8, Zevin-12-Computer Science-91.2, Asif-16-Mathematics-89.4]

//Collecting subjects and putting into Set.

Set<String> subjects = studentList.stream().map(Student::getSubject).collect(**Collectors.*toSet*()**);

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

//Output :

//[Economics, Literature, Computer Science, Mathematics, History]

// Collecting name and percentage of each student into a Map

Map<String, Double> namePercentageMap = studentList.stream().collect(**Collectors.*toMap***(Student::getName,

Student::getPercentage));

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

//Output :

//{Asif=89.4, Vijay=92.8, Zevin=91.2, Harry=71.9, Xiano=71.5, Nihira=84.6, Soumya=77.5, Mitshu=73.5, Harish=83.7, Paul=78.9}

// Collecting first 3 students into LinkedList

LinkedList<Student> studentLinkedList = studentList.stream().limit(3).collect(Collectors.*toCollection*(LinkedList::**new**));

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

//Output :

//[Paul-11-Economics-78.9, Zevin-12-Computer Science-91.2, Harish-13-History-83.7]

+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++

// Counting number of students.

Long studentCount = studentList.stream().collect(**Collectors.*counting*()**);

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

//Output : 10

//OR using **count()** also we can do

Long studentCount2 = studentList.stream().count();

System.***out***.println(studentCount2);

+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++

//Collecting highest percentage.

Optional<Double> highPercentage = studentList.stream().map(Student::getPercentage).collect(Collectors.*maxBy*(Comparator.*naturalOrder*()));

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

//Output : Optional[92.8]

**//OR**

Comparator<Double> comp = (Double d1, Double d2 ) -> d1.compareTo(d2);

System.***out***.println(studentList.stream().map(Student::getPercentage).max(comp).get());

//Collecting lowest percentage

Optional<Double> lowPercentage = studentList.stream().map(Student::getPercentage).collect(Collectors.*minBy*(Comparator.*naturalOrder*()));

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

//Output : Optional[71.5]

+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++

// Collecting sum of percentages

Double sumOfPercentages = studentList.stream().collect(Collectors.***summingDouble***(Student::getPercentage));

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

//Output : 815.0

//Collecting average percentage

Double averagePercentage = studentList.stream().collect(Collectors.***averagingDouble***(Student::getPercentage));

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

//Output : 81.5

//Extracting highest, lowest and average of percentage of students

**DoubleSummaryStatistics** studentStats = studentList.stream().collect(Collectors.*summarizingDouble*(Student::getPercentage));

System.***out***.println("Highest Percentage : "+studentStats.getMax());

System.***out***.println("Lowest Percentage : "+studentStats.getMin());

System.***out***.println("Average Percentage : "+studentStats.getAverage());

//Output :

//Highest Percentage : 92.8

//Lowest Percentage : 71.5

//Average Percentage : 81.5

+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++

[**Solving Real Time Queries Using Java 8 Features -Employee Management System**](https://javaconceptoftheday.com/solving-real-time-queries-using-java-8-features-employee-management-system/)

**Eg:**

**class** Employee

{

**int** id;

String name;

**int** age;

String gender;

String department;

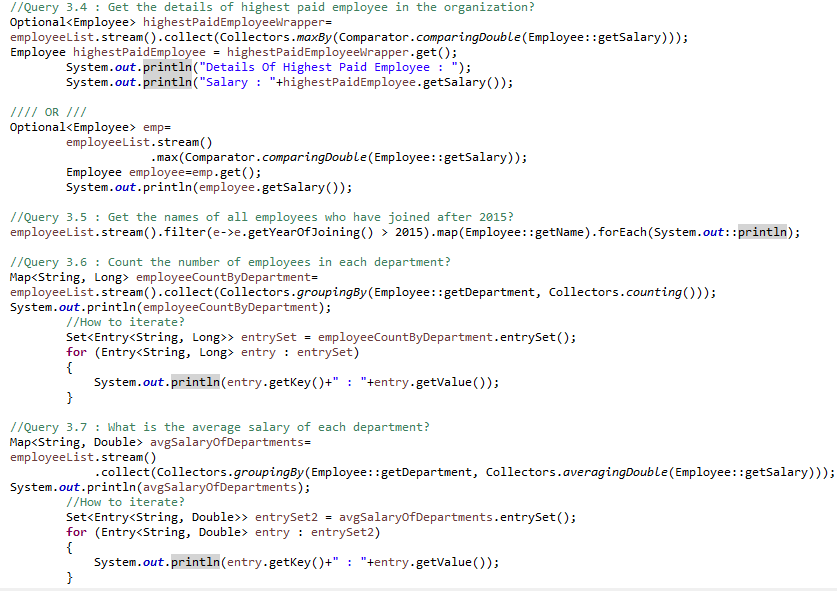
**int** yearOfJoining;

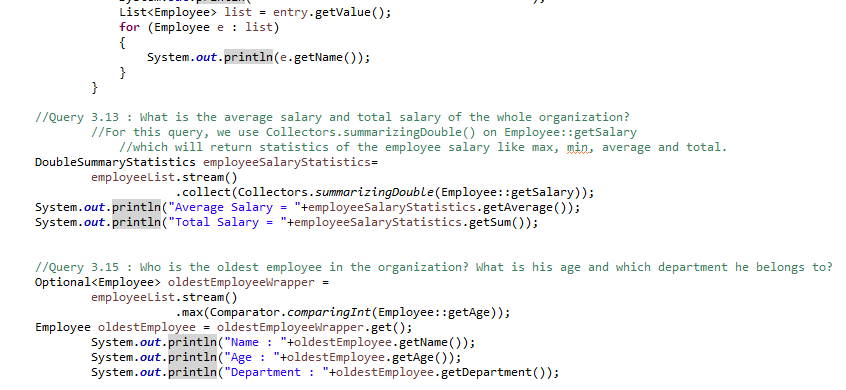
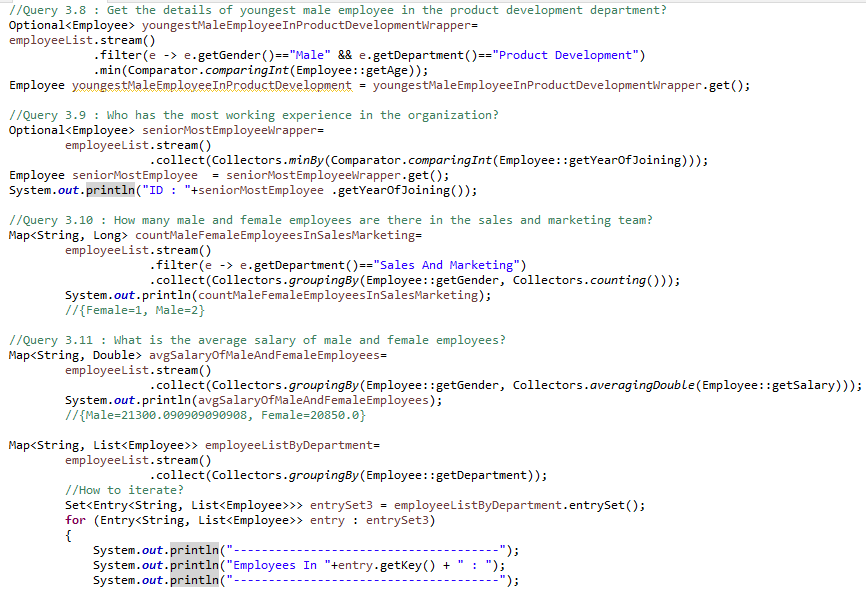
**double** salary;

//setters getters

//constructor

//to string

****



**Date and Time API: (Joda-Time API) :**

in the 1.8version oracle people introduced Joda-Time API.

**import** Java.time.\*;

**public** **class** DateTime {

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

LocalDate date = LocalDate.now();

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

LocalTime time = LocalTime.now();

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

}

}

O/p:

2015-11-23 12:39:26:587

Once we get LocalDate object we can call the following methods on that object to retrieve Day,month and year values separately.

**import** Java.time.\*;

**class** Test {

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

LocalDate date = LocalDate.now();

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

**int** dd = date.getDayOfMonth();

**int** mm = date.getMonthValue();

**int** yy = date.getYear();

System.***out***.println(dd + "..." + mm + "..." + yy);

System.***out***.printf("\n%d-%d-%d", dd, mm, yy);

}

}

Once we get LocalTime object we can call the following methods on that object.

Ex:

importJava.time.\*;

**class** Test {

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

{

LocalTime time = LocalTime.now();

**int** h = time.getHour();

**int** m = time.getMinute();

**int** s = time.getSecond();

**int** n = time.getNano();

System.out.printf("\n%d:%d:%d:%d",h,m,s,n);

}

}

If we want to represent both Date and Time then we should go for LocalDateTime object.

LocalDateTimedt = LocalDateTime.now();

System.out.println(dt);

O/p: 2015-11-23T12:57:24.531

We can represent a particular Date and Time by using LocalDateTime object as follows.

Ex:

LocalDateTime dt1 = LocalDateTime.of(1995,Month.APRIL,28,12,45);

sop(dt1);

Ex:

LocalDateTime dt1=LocalDateTime.of(1995,04,28,12,45);

Sop(dt1);

Sop(“After six months:”+dt.plusMonths(6));

Sop(“Before six months:”+dt.minusMonths(6));

To Represent Zone:

ZoneId object can be used to represent Zone.

Ex:

**import** Java.time.\*;

**class** ProgramOne {

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

ZoneId zone = ZoneId.systemDefault();

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

}

}

We can create ZoneId for a particular zone as follows

Ex:

ZoneId la = ZoneId.of("America/Los\_Angeles");

ZonedDateTimezt = ZonedDateTime.now(la);

System.out.println(zt);

Period Object:

Period object can be used to represent quantity of time

Ex:

LocalDate today = LocalDate.now();

LocalDate birthday = LocalDate.of(1989,06,15);

Period p = Period.between(birthday,today);

System.out.printf("age is %d year %d months %d days",p.getYears(),p.getMonths(),p.getDays());

#write a program to check the given year is leap year or not

**import** Java.time.\*;

**public** **class** Leapyear {

**int** n = Integer.*parseInt*(args[0]);

Year y = Year.of(n);

**if**(y.isLeap())

System.out.printf("%d is Leap year",n);

**else**

System.out.printf("%d is not Leap year",n);

}

# SQL:

## Qspider document



## Difference between Statement and PreparedStatement :

|  |  |
| --- | --- |
| Statement | PreparedStatement |
| 1. It is base interface. | It extends statement interface. |
| 1. It is used when SQL query is to be executed only once. | It is used when SQL query is to be executed multiple times. |
| 1. Used to execute normal SQL queries. | Used to execute dynamic SQL queries. |
| 1. You can’t pass parameters at runtime. | You can pass parameters at runtime. |
| 1. Used for CREATE, ALTER, DROP statements. | Used for the queries which are to be executed multiple times. |
| 1. Performance is very low. | Performance is better than Statement. |

**//Creating The Statement Object**

Statement GFG = con.createStatement();

//Executing The Statement

GFG.executeUpdate("CREATE TABLE STUDENT(ID NUMBER NOT NULL, NAME VARCHAR)");

**//Creating the PreparedStatement object**

PreparedStatement GFG = con.prepareStatement("update STUDENT set NAME = ? where ID = ?");

## **what is ACID property in database?**

The ACID properties describes the transaction management well by maintaining the data by consistency of data and availability of data in the database.

ACID stands for Atomicity, Consistency, isolation and durability.

1. **Atomicity** means either all successful or none.
2. **Consistency** ensures bringing the database from one consistent state to another consistent state.
3. **Isolation** ensures that transaction is isolated from other transaction, nothing but independent execution.
4. **Durability** means once a transaction has been committed, it will remain so, even in the event of errors, power loss etc.

Based on image you explain above points :

Eg: <https://www.javatpoint.com/acid-properties-in-dbms>

|  |
| --- |
|  |
|  |
| **Consistency** |
| **Isolation:** |

what is differance betweeen sysdate and now() and current\_date?

Current\_date() will give you the **only date**.

MariaDB [test]> select current\_date;

+--------------+

| current\_date |

+--------------+

| 2020-10-29 |

+--------------+

now() give you the **datetime**.  
sysdate() give you the current **datetime**.

NOW() returns a constant time that indicates the time at which the **statement began to execute.**

SYSDATE(), which returns the **exact time at which it executes**.

mysql> SELECT NOW(), SLEEP(2), NOW();

+---------------------+----------+---------------------+

| NOW() | SLEEP(2) | NOW() |

+---------------------+----------+---------------------+

| 2006-04-12 13:47:36 | 0 | 2006-04-12 13:47:36 |

+---------------------+----------+---------------------+

mysql> SELECT SYSDATE(), SLEEP(2), SYSDATE();

+---------------------+----------+---------------------+

| SYSDATE() | SLEEP(2) | SYSDATE() |

+---------------------+----------+---------------------+

| 2006-04-12 13:47:44 | 0 | 2006-04-12 13:47:46 |

+---------------------+----------+---------------------+

|  |  |  |
| --- | --- | --- |
| **Comparison Basis** | **MySQL** | **Oracle** |
| Introduction | It is an open-source, cross-platform relational database management system built by Swedish Company MYSQL AB and currently supported by the Oracle. | Oracle is a relational database system (RDBMS) that implements object-oriented features. It allows to store and retrieve data quickly and safely. It can handle a large amount of data. |
| Release | It was released in 1995. | It was released in 1980. |
| Cost | It is free and open-source. It is licensed under the GNU. | It is licensed for commercial purposes, but it provides the express edition for free. The express edition is recommended for students only. |
| Scalability | MySQL database is used for small and big businesses. | Oracle database is used for very large scale deployments. |
| Data Partitioning | It does not support data partitioning. | It supports data partitioning. |
| Security | It requires a username, password, and host to access the database. | It requires a username, password, and profile validation to access the database. |
| System Type | It only works with the static system. | It can work with both static and dynamic systems. |
| Null Value | MySQL supports the null value. | Oracle does not support the null value. |
| Character | MySQL support only two characters that are CHAR and VARCHAR. | Oracle supports four different characters that are CHAR, VARCHAR2, NCHAR, and NVARCHAR2. |
| Backup Mechanism | It offers only two backup mechanisms that are mysqlhotcopy and mysqldump. | It offers many backup mechanisms that are backup, hot backup, import, export, etc. |
| XML Support | It does not support XML. | It supports XML. |
| Storage Features | It contains only a few storage features like tablespace, synonym, packages, and many others. | It supports many storage features that are tablespace, synonym, packages, etc. |
| Locking facility | MySQL has only a table locking facility. | Oracle has table locking as well as a row locking facility. |
| Language support | MySQL support only SQL language. | Oracle supports both SQL and PL/SQL languages. |
| Operating System Support | It supports the following Operating System:   * Windows * Mac OS X * Linux * UNIX * z/OS * BSD * Symbian * AmigaOS | It supports the following Operating System:   * Windows * Mac OS X * Linux * UNIX * z/OS |

|  |  |  |
| --- | --- | --- |
| **Parameters** | **MySQL** | **MariaDB** |
| Definition | It is an open-source, cross-platform relational database management system built by Swedish Company MYSQL AB. Oracle Corporation develops and maintains at present. | It is a fork of the MySQL database management system that has data processing capabilities for small and enterprise tasks. The fork means to guarantees the MySQL codebase would be free forever. It is an improved and advanced version of MySQL. |
| Initial Release | The first release of MySQL is in May 1995. | The first release of MariaDB is in October 2009. |
| Current Release | The latest release of MySQL is 8.0.20 on 27 April 2020. | The latest release of MariaDB is 10.4.13 on 12 May 2020. |
| Protocols | MySQL | MySQL and MariaDB |
| Source Code | Open-source and Proprietary | Open-source |
| Replication | MySQL versions do not allow data to replicate from the MariaDB server to it. MySQL offers master-slave replication. | Different versions of MariaDB allow us to replicate data from MySQL server to the MariaDB server. MariaDB offers master-master and master-slave replication of the data. |
| Storage Engines | It has limited storage engines: InnoDB MyISAM BLACKHOLE CSV MEMORY ARCHIVE MERGE | It has extensive storage engine: MyISAM BLACKHOLE CSV MEMORY ARCHIVE MERGE ColumnStore MyRocks Aria SphinxSE TokuDB CONNECT SEQUENCE Spider Cassandra |
| Check Constraint | It does not have a check constraint. | It provides support for check constraints. |
| Speed | It has a slower speed in comparison to MariaDB. | It provides faster speed because it is the advanced version of MySQL. |
| Connection Pool | MySQL thread pool can support less than 200,000 connections per time. | MariaDB has an advanced thread pool that provides faster execution and can support more than 200,000 connections per time. |
| Development | Closed | Open |
| Document Maintenance | Oracle Corporation maintains the document. | MariaDB Foundation and other developers can also participate in development and maintenance. |
| Routing | It has MySQL Router. | It has a MariaDB MaxScale. |
| Data Masking | It does not allow data masking. | It allows data masking. |
| GUI | It provides MySQL Workbench. | It provides SQLyog. |
| Company Use | GitHub, US Navy, NASA, Tesla, Netflix, WeChat, Facebook, Twitter, YouTube, and Spotify. | Google, Craigslist, Wikipedia, archlinux, Red Hat, CentOS, and Fedora. |

++++++++++++++++++++

## Mysql query practice

Select substring(FIRST\_NAME,1,3) from Worker;

**-- Write An SQL Query That Fetches The Unique Values Of DEPARTMENT From Worker Table And Prints Its Length.**

Select distinct length(DEPARTMENT),DEPARTMENT from Worker;

**-- Write An SQL Query To Show Only Odd Rows From A Table.**

SELECT \* FROM Worker WHERE MOD (WORKER\_ID, 2) <> 0;

**--  Write An SQL Query To Show Only Even Rows From A Table.**

SELECT \* FROM Worker WHERE MOD (WORKER\_ID, 2) = 0;

**-- Write An SQL Query To Clone A New Table From Another Table.**

CREATE TABLE WorkerClone2 LIKE Worker;

INSERT INTO WorkerClone2 SELECT \* FROM Worker;

**-- Write An SQL Query To Show The Current Date And Time.**

SELECT CURDATE();                                     -- prints only date

select now();                                                                         --                     prints both date n time

**-- Write An SQL Query To Print The FIRST\_NAME From Worker Table After Replacing ‘a’ With ‘A’.**

Select REPLACE(FIRST\_NAME,'a','A') from Worker;

**-- Write An SQL Query To Print The FIRST\_NAME And LAST\_NAME From Worker Table Into A Single Column COMPLETE\_NAME. A Space Char Should Separate Them.**

Select CONCAT(FIRST\_NAME, ' ', LAST\_NAME) AS 'COMPLETE\_NAME' from Worker;

**-- Write An SQL Query To Print Details Of The Workers Who Have Joined In Feb’2014.**

Select \* from Worker where year(JOINING\_DATE) = 2014 and month(JOINING\_DATE) = 2;

### **group by queries**

==================

**--Write An SQL Query To Fetch The No. Of Workers For Each Department In The Descending Order.**

SELECT DEPARTMENT, count(WORKER\_ID) No\_Of\_Workers

FROM worker

GROUP BY DEPARTMENT

ORDER BY No\_Of\_Workers DESC;

**-- Write An SQL Query To Show All Departments Along With The Number Of People In There.**

SELECT DEPARTMENT, COUNT(DEPARTMENT) as 'Number of Workers' FROM Worker GROUP BY DEPARTMENT;

**-- Write An SQL Query To Fetch Departments Along With The Total Salaries Paid For Each Of Them.**

SELECT DEPARTMENT, sum(Salary) from worker group by DEPARTMENT;

having clause

=============

**-- Write An SQL Query To Fetch The Departments That Have Less Than Five People In It.**

SELECT DEPARTMENT, COUNT(WORKER\_ID) as 'Number of Workers' FROM Worker GROUP BY DEPARTMENT HAVING COUNT(WORKER\_ID) < 5;

**-- write a query to give a list of duplicate employee names:**

select name, count(name) as count from employee group by name having count > 1;

**Using self join**

select o1.org\_name, count((o1.org\_name)) from orgnisation o1, orgnisation o2 where o2.org\_name=o1.org\_name and o2.id!=o1.id;

Query to find the count of duplicate elements in table? (hint : use group by)

SELECT username, email, COUNT(\*)

FROM users

GROUP BY username, email

HAVING COUNT(\*) > 1

### Joins

====

**-- Write An SQL Query To Print Details Of The Workers Who Are Also Managers.**

SELECT DISTINCT W.FIRST\_NAME, T.WORKER\_TITLE

FROM onlinemysqlpractice.Worker W

INNER JOIN onlinemysqlpractice.Title T

ON W.WORKER\_ID = T.WORKER\_REF\_ID

AND T.WORKER\_TITLE in ('Manager');

### **limit condition**

===============

**-- Write An SQL Query To Show The Top 2  Records Of a Table**

SELECT \* FROM Worker  LIMIT 2;

**-- Write An SQL Query To Show The least 2  Records Of A Table**

SELECT \* FROM Worker order by worker\_id desc LIMIT 2;

**-- Write An SQL Query To Show The Top 5  Records Of a Table. and display result salary wise.**

SELECT \* FROM Worker ORDER BY Salary asc LIMIT 5;

**-- Write An SQL Query To Show The Last Record From A Table.**

Select \* from Worker where WORKER\_ID = (SELECT max(WORKER\_ID) from Worker);

**-- Write An SQL Query To Fetch The First Row Of A Table.**

Select \* from Worker where WORKER\_ID = (SELECT min(WORKER\_ID) from Worker);

**-- Write An SQL Query To Fetch Three Max Salaries From A Table.**

SELECT Salary FROM Worker order by salary asc limit 3;

**-- using sub query ::**

**--  Write An SQL Query To Show The Second Highest Salary From A Table.**

Select max(Salary) from Worker

where Salary not in (Select max(Salary) from Worker);

**-- Queries to find 2nd , 3rd , 4th ... highest salaries ::**

**-- using limit :: Finding 3rd highest salary**

Select min(salary) from (select salary from Employee order by salary desc limit 3)

**-- Without Using TOP Or Limit Method.**

SELECT Salary

FROM Worker W1

WHERE 3 = (

SELECT COUNT( DISTINCT ( W2.Salary ) )

FROM Worker W2

WHERE W2.Salary >= W1.Salary

);

### **self join**

========

**-- Write An SQL Query To Fetch The List Of Employees With The Same Salary.**

Select distinct W.WORKER\_ID, W.FIRST\_NAME, W.Salary

from Worker W, Worker W1

where W.Salary = W1.Salary

and W.WORKER\_ID != W1.WORKER\_ID;

+++++++++++++++++++++++++++++

Table : employee

|  |  |  |
| --- | --- | --- |
| id | name | mgr\_id |
| 1 | X | 4 |
| 2 | Y | 4 |
| 3 | Z | 4 |
| 4 | P | Null |

**-- write an SQL query to give list of employees wrt to their manager name.**

Expected:

|  |  |
| --- | --- |
| name | mgr\_name |
| 1 | P |
| 2 | P |
| 3 | P |

select A.name,B.name as mgr\_name from employee A, employee B

where A.mgr\_id=B.id;

+++++++++++++++++++++++++++++

# **Programs:**

****

Adding element into arrayList with infinite loop:

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

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

**for** (**int** i = 0; i >= 0; i++) {

al.add(**new** Student(i, "ram"));

System.***out***.println("size =" + al.size());

}

}

Output:

size =57293426

size =57293427

Exception in thread "main" java.lang.OutOfMemoryError: GC overhead limit exceeded

at com.hcl.Test.m1(Test.java:12)

at com.hcl.Test.main(Test.java:21)

How do you find the count of object creation?

**class** Test

{

**static** **int** *noOfObjects* = 0; //If you remove static, then count will always 1

**public** Test(**int** n) {

*noOfObjects* += 1;

}

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

**for** (**int** i = 0; i >= 0; i++) {

Test t = **new** Test(i);

System.***out***.println(Test.*noOfObjects*);

}

}

}

+++++++++++++++++++++++++++++++++++++++++++

What are Different ways to print Exception messages (3) ?

And what is difference between them?

Using **printStackTrace**() :

By using this method, we will get name(e.g. java.lang.ArithmeticException) and description(e.g. / by zero) of an exception separated by colon, and line of the exception

Using **toString**()

using this method, we will only get name and description of an exception

Using **getMessage**()

By using this method, we will only get description of an exception.

Eg:

**public** **class** Test{

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

**try**

{

**int** a = 20/0;

}

**catch** (Exception e)

{

// printStackTrace method

// prints line numbers + call stack

e.printStackTrace();

 //System.out.println(e.toString());

 //System.out.println(e.getMessage());

// Prints what exception has been thrown

System.***out***.println(e); //This wont help when deployed in production.

}

}

}

Output:

java.lang.ArithmeticException: / by zero

at com.hcl.Test.main(Test.java:8)

java.lang.ArithmeticException: / by zero //using A

https://www.geeksforgeeks.org/3-different-ways-print-exception-messages-java/?ref=rp

++++++++++++++++++++++++++++++++++++++++++

## Integer to String and String to Integer:

**int** i=101;

String s=Integer.*toString*(i);

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

Integer j=Integer.*parseInt*(s);

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

## Find the sum of given number and return :: 123 -> 6 12 -> 3

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

System.***out***.println("give input");

**int** input=sc.nextInt();

**int** sum=0;

**while**(input!=0){

**int** lastdigit=input%10;

sum=sum+ lastdigit;

input=input/10;

}

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

## Reverse string

String s="hcl";

String s2="";

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

{

s2=s2+s.charAt(i);

}

System.***out***.println(s2);

## What are the ways for commandLine inputs?

1.Using Scanner class

2.Using BufferedReader class

3.Using Console class

**public** **class** CommandLineInput {

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

//////////////////Using Scanner class//////////////////

Scanner sc1 = **new** Scanner(System.***in***);

// Scanner sc2 = new Scanner(System.in);

// Scanner sc3 = new Scanner(System.in);

System.***out***.println("please provide the input ");

**int** i = sc1.nextInt();

//double d=sc2.nextDouble();

//String s=sc3.next();

System.***out***.println("your given input is :"+i);

//////////////////Using BufferedReader class//////////////////

BufferedReader reader = **new** BufferedReader(**new** InputStreamReader(System.***in***));

System.***out***.print("Enter your name: ");

String name = reader.readLine();

System.***out***.println("Your name is: " + name);

}

}

## Java program to check Palindrome ## ex: madam, nan, gadag //

String s="gadag";

String reverse="";

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

{

reverse=reverse+s.charAt(i);

}

**if**(s.equals(reverse))

{

System.out.println("passed string "+s+" is palindrome");

}

## Find first repeated character in a String // or Java Program to find duplicate Characters in a String //

String c="analog";

**char** c2;

**int** i2=0;

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

{

c2=c.charAt(i);

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

{

**if**(c2==c.charAt(j)){

++i2;

}

}

**if**(i2==0)

{

System.out.println("non repeated char String”);

}

}

## Java program to find a length of string without using inbuilt length method //

String s="sudheer";

**char**[] strCharArray=s.toCharArray();

**int** k=0;

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

{

++k;

}

System.out.println("length of string is "+k);

## Java Program to find second largest number in array: //

## Fibonic series

**Int** t1=0,t2=1;

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

System.***out***.print(t1+" + ");

**int** sum=t1+t2;

t1=t2;

t2=sum;

}

0 + 1 + 1 + 2 + 3 + 5 + 8 + 13 + 21 + 34 +

## Check Prime number or not?

**int** checkPrime=23;

//int checkPrime=24;

**int** flag=0;

**int** halfValue=checkPrime/2;

**if**(checkPrime==0 || checkPrime==1)

{

System.***out***.println("not a prime");

}

**for** (**int** i = 2; i <=halfValue; i++)

{

**if**(checkPrime%i==0)

{

flag=1;

System.***out***.println("given num is not prime");

**break**;

}

}

**if**(flag==0)

{

System.***out***.println("given num is a prime");

}

## **swap 2 variables without 3rd variable.**

**class** Test2 {

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

{

**int** x = 10;

**int** y = 5;

x = x + y;

y = x - y;

x = x - y;

System.***out***.println("After swaping:"

+ " x = " + x + ", y = " + y);

}

}

//After swaping: x = 5, y = 10

## [**Array to ArrayList**](http://javaconceptoftheday.com/array-to-arraylist-in-java-with-examples/) **:**

String[] array = **new** String[] {"ANDROID", "JSP", "JAVA", "STRUTS", "HADOOP", "JSF"};

ArrayList<String> list = **new** ArrayList<String>(Arrays.asList(array));

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

**Output :** [ANDROID, JSP, JAVA, STRUTS, HADOOP, JSF]

## **\*ArrayList To Array :**

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

        list.add("JAVA");

        list.add("JSP");

        list.add("ANDROID");

        String[] array = **new** String[list.size()];

        list.toArray(array);

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

             System.***out***.println(array[i]);

      }

## How to find duplicate elements from the array

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

**int** i = 0;

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

**if** (arr[i] == arr[j]) {

System.***out***.println("found duplicate");

}

++i;

}

## [**How To Remove Duplicate Elements From ArrayList In Java?**](http://javaconceptoftheday.com/how-to-remove-duplicate-elements-from-arraylist-in-java/)

 //we can achieve this using hashSet/LinkedList

**public** **class** D

{

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

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

         listWithDuplicateElements.add("JAVA");

         listWithDuplicateElements.add("J2EE");

         listWithDuplicateElements.add("JAVA");

         //Printing listWithDuplicateElements

         System.***out***.print("ArrayList With Duplicate Elements :");

         System.***out***.println(listWithDuplicateElements);                       //[JAVA, J2EE, JAVA]

         //Constructing HashSet using listWithDuplicateElements

         HashSet<String> set = **new** HashSet<String>(listWithDuplicateElements);

         //Constructing listWithoutDuplicateElements using set

         ArrayList<String> listWithoutDuplicateElements = **new** ArrayList<String>(set);

         //Printing listWithoutDuplicateElements

         System.***out***.print("ArrayList After Removing Duplicate Elements :");

         System.***out***.println(set);                                       //[J2EE, JAVA] //(not maintaining insertion order)

      }

}

## **How To Remove Duplicate Elements From ArrayList & maintain the insertion order as it is?**

      //same as above just replace LinkedHashSet instead of HashSet

      LinkedHashSet<String> set = **new** LinkedHashSet<String>(listWithDuplicateElements);

## How to Convert map elements to List ?

#############             MapToListConvertion     #################

**public** **class** MapToListConvertion {

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

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

        m.put(101,"sudheer");

        m.put(102,"sudheer2");

        m.put(103,"sudheer3");

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

        System.***out***.println("converting HashMap keys into arrayList");

            List<Integer> k=**new** ArrayList<Integer>(m.keySet());

            Collections.*sort*(k);

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

            System.***out***.println("\nconverting HashMap values into arrayList");

            List<String> values=**new** ArrayList<String>(m.values());

            Collections.*sort*(values);

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

            System.***out***.println("\nconverting HashMap keys & values into arrayList");

            List<Entry<Integer,String>> entry=**new** ArrayList<Entry<Integer,String>>(m.entrySet());

**for** (**int** i = 0; i < entry.size(); i++) {

                  System.***out***.println(entry.get(i));

            }

      }

}

/\*output:

      HashMap size is 3

      converting HashMap keys into arrayList

      [101, 102, 103]

      converting HashMap values into arrayList

      [sudheer, sudheer2, sudheer3]

      converting HashMap keys & values into arrayList

      102=sudheer2

      103=sudheer3

      101=sudheer

\*/

###########   MapCustomObjectToListConversion     ########################

**public** **class** MapCustomObjectToListConversion {

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

            Employee e1=**new** Employee(1, "sudheer");

            Employee e2=**new** Employee(2, "sudheer2");

            Employee e3=**new** Employee(3, "sudheer3");

            HashMap<Integer,Employee> m = **new** HashMap<Integer,Employee>();

            m.put(101,e1);

         m.put(102,e2);

         m.put(103,e3);

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

        System.***out***.println("converting HashMap keys into arrayList");

            List<Integer> k=**new** ArrayList<Integer>(m.keySet());

            Collections.*sort*(k);

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

            System.***out***.println("\nconverting HashMap values into arrayList");

            List<Employee> values=**new** ArrayList<Employee>(m.values());

**for** (**int** i = 0; i < values.size(); i++) {

                  Employee emp=values.get(i);

                  System.***out***.println(emp.getId()+" "+emp.getName());

            }

            System.***out***.println("\nconverting HashMap keys & values into arrayList");

            List<Entry<Integer,Employee>> entry=**new** ArrayList<Entry<Integer,Employee>>(m.entrySet());

**for** (**int** i = 0; i < entry.size(); i++) {

                  Entry<Integer, Employee> e=entry.get(i);

                  Employee emp=e.getValue();

                  System.***out***.println(e.getKey()+" "+emp.getId()+" "+emp.getName());

            }

      }

}

/\*    output:

      HashMap size is 3

      converting HashMap keys into arrayList

      [101, 102, 103]

      converting HashMap values into arrayList

      2 sudheer2

      3 sudheer3

      1 sudheer

      converting HashMap keys & values into arrayList

      102 2 sudheer2

      103 3 sudheer3

      101 1 sudheer

\*/

## [How To Count Occurrences Of Each Character In String In Java?](https://javaconceptoftheday.com/how-to-count-occurrences-of-each-character-in-string-in-java/)

Java Program To Find Duplicate Characters In A String ?

**Java J2EE Java JSP J2EE è E=4, 2=2, v=2,  =4, P=1, S=1, a=4, J=5**.

//use HashMap<Character, Integer>

**static** **void** characterCount(String inputString)

    {

        HashMap<Character, Integer> map = **new** HashMap<Character, Integer>();

**char**[] strArray = inputString.toCharArray();

**for** (**char** c : strArray)

        {

**if**(map.containsKey(c))

            {

                map.put(c, map.get(c)+1);

            }

**else**

            {

                map.put(c, 1);

            }

        }

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

    }

problem-solving:

  find nth last element from the linked list .

  find pairs in an array whose sum is equal to given number.

# Spring MVC

## Advantages of Spring Framework

There are many advantages of Spring Framework. They are as follows:

**1) Predefined Templates**

Spring framework provides templates **for JDBC, Hibernate**, JPA etc. technologies. So there is no need to write too much code. It hides the basic steps of these technologies.

Let's take the example of JdbcTemplate, you don't need to write the code for exception handling, creating connection, creating statement, committing transaction, closing connection etc. You need to write the code of executing query only. Thus, it save a lot of JDBC code.

**2) Loose Coupling**

The Spring applications are loosely coupled because of **dependency injection**.

**3) Lightweight**

Spring framework is lightweight because of its **POJO implementation**

**4) Easy to test**

**5) Fast Development**

**6) MVC architecture:**

**Advantage of Dependency Injection**

* makes the code loosely coupled so **easy to maintain**
* makes the code **easy to test**

## **IoC container (Inversion Of Controller)**

**The IoC container is responsible for**

1. to instantiate the application,
2. to configure the objects and
3. to assemble dependencies between objects.
4. Manage the life cycle of bean.

The IoC container gets information from the XML file and works accordingly

There are two types of IoC containers. They are:

1. BeanFactory
2. ApplicationContext

The ApplicationContext interface is built on top of the BeanFactory interface. It adds some extra functionality than BeanFactory such as simple integration with Spring's AOP, message resource handling (for I18N), event propagation, application layer specific context (e.g. WebApplicationContext) for web application. So it is better to use ApplicationContext than BeanFactory.

**Spring framework provides two ways to inject dependency**

* By Constructor
* By Setter method

**Difference:**

1. **Partial dependency**: can be injected using setter injection but it is not possible by constructor. Suppose there are 3 properties in a class, having 3 arg constructor and setters methods. In such case, if you want to pass information for only one property, it is possible by setter method only.
2. **Overriding**: Setter injection overrides the constructor injection. If we use both constructor and setter injection, IOC container will use the setter injection.
3. **easily change the value**: We can easily change the value by setter injection. It doesn't create a new bean instance always like constructor. So, setter injection is flexible than constructor injection.

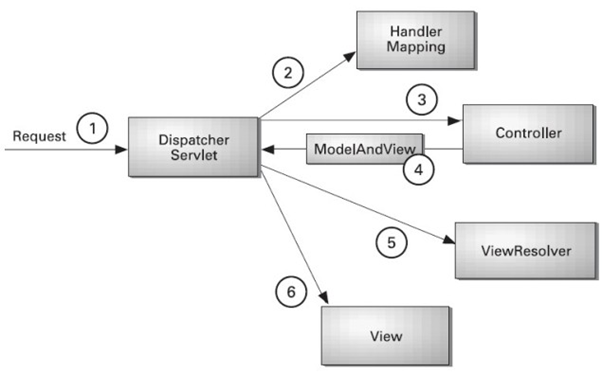
## **MVC:**

<https://www.javatpoint.com/spring-mvc-tutorial>

A Spring MVC is a Java framework which is used to build web applications. It follows the Model-View-Controller design pattern. It implements all the basic features of a core spring framework like Inversion of Control, Dependency Injection.

A Spring MVC provides an elegant solution to use MVC in spring framework by the help of **DispatcherServlet**. Here, **DispatcherServlet** is a class that receives the incoming request and maps it to the right resource such as controllers, models, and views.

* **Model** - A model contains the data of the application. A data can be a single object or a collection of objects.
* **Controller** - A controller contains the business logic of an application. Here, the @Controller annotation is used to mark the class as the controller.
* **View** - A view represents the provided information in a particular format. Generally, JSP+JSTL is used to create a view page. Although spring also supports other view technologies such as Apache Velocity, Thymeleaf and FreeMarker.
* **Front Controller** - In Spring Web MVC, the DispatcherServlet class works as the front controller. It is responsible to manage the flow of the Spring MVC application.



* As displayed in the figure, all the incoming request is intercepted by the DispatcherServlet that works as the front controller.
* The DispatcherServlet gets an entry of handler mapping from the XML file and forwards the request to the controller.
* The controller returns an object of ModelAndView.
* The DispatcherServlet checks the entry of view resolver in the XML file and invokes the specified view component.

**What is the front controller class of the Spring MVC?**

**OR dispatcher servlet?**

1&2

On receiving an HTTP request, the DispatcherServlet consults HandlerMapping (these are the configuration files) to call the appropriate Controller.

3&4

Then, the controller calls appropriate service methods to set the Model data. It also returns the view name to DispatcherServlet.

5&6

DispatcherServlet, with the help of ViewResolver, picks up the defined view for the request.

Once the view is finalized, the DispatcherServlet passes the Model data to View – where it is finally rendered on the browser.

**What is a Viewresolver pattern and how does it work in MVC?**

 View Resolver is a J2EE pattern which allows the applications **to dynamically choose technology for rendering the data on the browser** (View). Any technology like HTML, JSP, Tapestry, XSLT, JSF, or any other such technology can be used for View. The View Resolver pattern holds the mapping of different views. The Controller returns the name of the View which is then passed to View Resolver for selecting the appropriate technology.

**What do you mean by a “Bean” in the context of Spring framework?**

Any class that is initialised by the IoC container is known as a bean in Spring. The lifecycle of a Spring Bean is managed by Spring IoC Container.

## **Bean scopes in spring?**

There are 5 bean scopes in spring framework.

|  |  |  |
| --- | --- | --- |
| **No.** | **Scope** | **Description** |
| 1) | singleton | The bean instance will be only once and same instance will be returned by the IOC container. It is the default scope. |
| 2) | prototype | The bean instance will be created each time when requested. |
| 3) | request | The bean instance will be created per HTTP request. |
| 4) | session | The bean instance will be created per HTTP session. |
| 5) | globalsession | The bean instance will be created per HTTP global session. It can be used in portlet context only. |

### **Singleton Scope**:

<https://www.geeksforgeeks.org/singleton-and-prototype-bean-scopes-in-java-spring/>

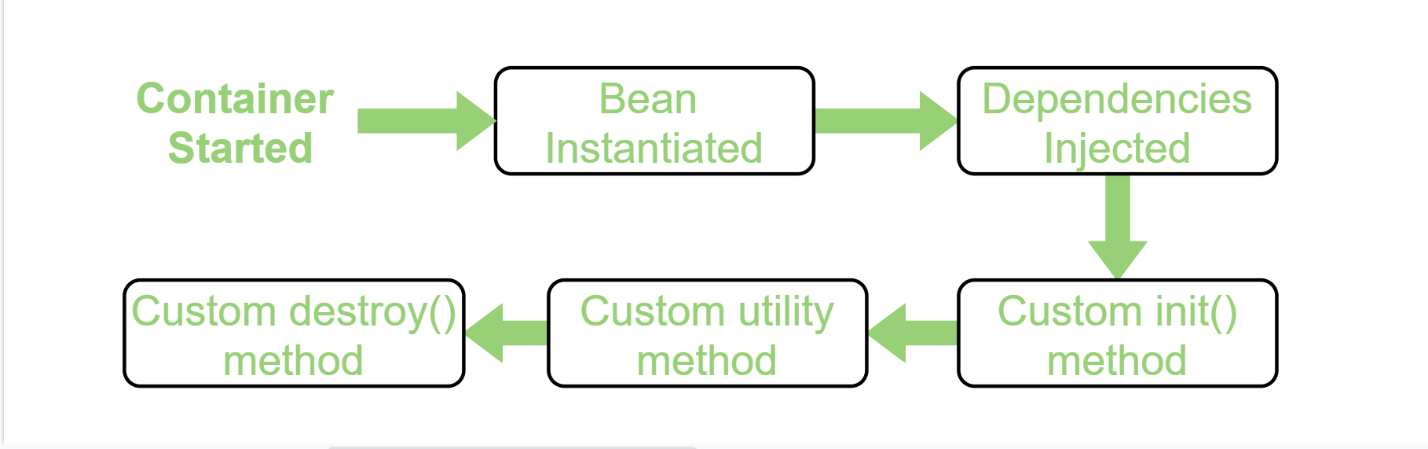
1. If the scope is a singleton, then only **one instance** of that bean will be instantiated per Spring IoC container and the same instance will be shared for each request.
2. That is when the scope of a bean is declared singleton, then whenever a new request is made for that bean, spring IOC container first checks whether an instance of that bean is already created or not. If it is already created, then the IOC container returns the same instance otherwise it creates a new instance of that bean only at the first request.
3. By **default**, the scope of a bean is a singleton.
4. singleton scope is used for the **stateless** beans

### **Prototype Scope:**

1. If the scope is declared **prototype**, then spring IOC container will create a new instance of that bean every time a request is made for that specific bean.
2. we use the prototype scope for all beans that are **stateful**.

## Bean life cycle:

<https://www.geeksforgeeks.org/bean-life-cycle-in-java-spring/>



There are different approaches for implementing bean life cycle:

**By XML**

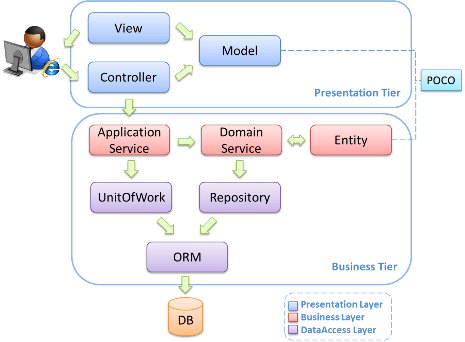
**By Programmatic Approach:**

**Using Annotation:**

In using Annotation, we need annotate **init()** method by **@PostConstruct** annotation and **destroy()** method by **@PreDestroy** annotation.

Note: To invoke the **destroy()** method we have to call the **close()** method of ConfigurableApplicationContext.

**How will you compare the MVC framework to the three-tier architecture?**

 A Three-tier architecture is an architecture style whereas MVC is a design pattern.  
  
Having said that, in larger applications, MVC forms the presentation tier of a three-tier architecture. The Model, View, and Controller are concerned only with the presentation – they use the middle tier to populate their models.

**1) In which scenario, you will use singleton and prototype scope?**

Singleton scope should be used with EJB **stateless session bean** and prototype scope with EJB **stateful session bean**.

7. What is auto wiring ? And different annotations and their usage.

Autowiring feature of spring framework enables you to inject the object dependency implicitly. It internally uses setter or constructor injection.

Autowiring can't be used to inject primitive and string values. It works with reference only.

Advantage of Autowiring

It requires the less code because we don't need to write the code to inject the dependency explicitly.

## **Spring AOP**

AOP breaks the program logic into distinct parts (called concerns). It is used to increase modularity by cross-cutting concerns.

A cross-cutting concern is a concern that can affect the whole application and should be centralized in one location in code as possible, such as transaction management, authentication, logging, security etc.

But the key unit of modularity is aspect than class.

**Join point**

Join point is any point in your program such as method execution, exception handling, field access etc. Spring supports only method execution join point.

**Advice**

Advice represents an action taken by an aspect at a particular join point. There are different types of advices:

o    Before Advice: it executes before a join point.

o    After Returning Advice: it executes after a joint point completes normally.

o    After Throwing Advice: it executes if method exits by throwing an exception.

o    After (finally) Advice: it executes after a join point regardless of join point exit whether normally or exceptional return.

o    Around Advice: It executes before and after a join point.

**Pointcut**

It is an expression language of AOP that matches join points.

**Introduction**

It means introduction of additional method and fields for a type. It allows you to introduce new interface to any advised object.

**Target Object**

It is the object i.e. being advised by one or more aspects. It is also known as proxied object in spring because Spring AOP is implemented using runtime proxies.

**Aspect**

It is a class that contains advices, joinpoints etc.

**Interceptor**

It is an aspect that contains only one advice.

**AOP Proxy**

It is used to implement aspect contracts, created by AOP framework. It will be a JDK dynamic proxy or CGLIB proxy in spring framework.

**Weaving**

It is the process of linking aspect with other application types or objects to create an advised object. Weaving can be done at compile time, load time or runtime. Spring AOP performs weaving at runtime

# **Multithreading**

**why thread pool?**

creating a new thread for every job may create performance issue with memory. to over come this we should go for thread pool.

Thread pool is a pool of already created threads ready to do our job.

introduces in java 1.5 version.

thread pool framework also known as executor framework.

**How to create thread pool?**

ExecutorService service = Executors.newFixedThreadPool(3);

**How I can assign runnable jobs to thread pool?**  //by using submit method

service.submit(job);

**we can shutdown ExecutorService by shutdown method**

service.shutdown();

**import** java.awt.Graphics;

**import** java.awt.print.PageFormat;

**import** java.awt.print.Printable;

**import** java.awt.print.PrinterException;

**public** **class** PrintJob **implements** Runnable {

String name;

PrintJob(String name)

{

**this**.name = name;

}

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

{

System.***out***.println(name + "... Job started by Thread: "+Thread.*currentThread*().getName());

**try**

{

Thread.*sleep*(5000);

}

**catch** (Exception e) {

}

System.***out***.println(name + "... Job completed by Thread: "+Thread.*currentThread*().getName());

}

}

**import** java.util.concurrent.ExecutorService;

**import** java.util.concurrent.Executors;

**public** **class** ExecutorDemo {

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

PrintJob[] jobs = {**new** PrintJob("durga1"),

**new** PrintJob("durga2"),

**new** PrintJob("durga3"),

**new** PrintJob("durga4"),

**new** PrintJob("durga5"),

**new** PrintJob("durga6")};

ExecutorService service = Executors.*newFixedThreadPool*(3);

**for**(PrintJob job:jobs)

{

service.submit(job);

}

service.shutdown();

}

}

/\*

durga1... Job started by Thread: pool-1-thread-1

durga2... Job started by Thread: pool-1-thread-2

durga3... Job started by Thread: pool-1-thread-3

durga2... Job completed by Thread: pool-1-thread-2

durga1... Job completed by Thread: pool-1-thread-1

durga3... Job completed by Thread: pool-1-thread-3

durga4... Job started by Thread: pool-1-thread-3

durga5... Job started by Thread: pool-1-thread-2

durga6... Job started by Thread: pool-1-thread-1

durga4... Job completed by Thread: pool-1-thread-3

durga5... Job completed by Thread: pool-1-thread-2

durga6... Job completed by Thread: pool-1-thread-1

\*/

in the above ex 3 threads are responsible for to execute 6 jobs. so that single thread can be reused for multiple jobs.

while designing web servers and application servers, we can use thread pool concept.

**callable and future:**

**===============**

in the case of runnable job thread won’t return anything after completing the job.

if a thread required to return some result after execution then we should go for callable.

callable interface contains only one method called call()

public Object call() throws Exception

{

}

Eg:

**public** **class** MyCallable {

**int** num;

MyCallable(**int** num)

{

**this**.num=num;

}

**public** Object call() **throws** Exception

{

System.***out***.println(Thread.*currentThread*().getName()+"...is responsible to find sum of first "+num+" numbers");

**int** sum=0;

**for** (**int** i = 1; i <= num; i++) {

sum=sum+i;

}

**return** sum;

}

}

**import** java.util.concurrent.Callable;

**import** java.util.concurrent.ExecutionException;

**import** java.util.concurrent.ExecutorService;

**import** java.util.concurrent.Executors;

**import** java.util.concurrent.Future;

**public** **class** CallableFutureDemo {

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

MyCallable[] jobs = {**new** MyCallable(10),

**new** MyCallable(20),

**new** MyCallable(30),

**new** MyCallable(40),

**new** MyCallable(50),

**new** MyCallable(60)};

ExecutorService service = Executors.*newFixedThreadPool*(3);

**for**(MyCallable job:jobs)

{

Future f=service.submit(job);

System.***out***.println(f.get());

}

service.shutdown();

}

}

If we submit callable object to executor then after completing the job, thread returns an object of the type Future. i.e **Future** object can be used to retrieve the result from callable job.

## Differance between runnable and callable:

|  |  |
| --- | --- |
| **runnable** | **callable** |
| 1. if a thread is not required to return anything after completing a job then we should go for runnable | 1. if a thread is required to return something after completing a job then we should go for callable |
| 1. it has only one method call run() | 1. it has only one method call call() |
| 1. return type is void, cz not return anything | 1. return type is Object |
| 1. within run() if there is a chance of raising checked exception, so compusory we shd handle by using try catch, cz we cant use throws keyword for run(() | 1. within call() if there is a chance of raising checked exception, we are not required to handle by using try catch, cz we call() already throws Exception |
| 1. introduced in java 1.0 version | 1. introduced in java 1.5 version |

# RestFull webservice

### **RestFull webservices**

What REST stands for?

REST stands for **REpresentational State Transfer**.

**What are webservices?**

Webservices are the api’s , can written with any programming lang and run on different platforms , used for

exchanging data between applications or systems..

**What are RESTful webservices? (4 points)**

|  |  |  |
| --- | --- | --- |
| **sNo.** | **SOAP** | **REST** |
| 1) | SOAP is a **protocol**. | REST is an **architectural style**. |
| 2) | SOAP stands for **Simple Object Access Protocol**. | REST stands for **REpresentational State Transfer**. |
| 3) | SOAP **can't use REST** because it is a protocol. | REST **can use SOAP** web services because it is a concept and can use any protocol like HTTP, SOAP. |
| 4) | SOAP uses **services interfaces** to expose the business logic. | REST uses **URI** to expose business logic. |
| 5) | **JAX-WS** is the java API for SOAP web services. | **JAX-RS** is the java API for RESTful web services. |
| 6) | SOAP **defines standards**to be strictly followed. | REST does not define too much standards like SOAP. |
| 7) | SOAP **requires more bandwidth** and resource than REST. | REST **requires less bandwidth** and resource than SOAP. |
| 8) | SOAP **defines its own security**. | RESTful web services **inherits security measures** from the underlying transport. |
| 9) | SOAP **permits XML** data format only. | REST **permits different** data format such as Plain text, HTML, XML, JSON etc. |
| 10) | SOAP is **less preferred** than REST. | REST **more preferred** than SOAP. |

Web services based on **REST Architecture** are known as RESTful web services. These web services use **HTTP methods** to implement the concept of REST architecture. A RESTful web service usually defines a **URI**, Uniform Resource Identifier a service, provides resource representation such as **JSON** and set of HTTP Methods.

**What are the differences between SOAP and RESTful services.**

**What is a Resource in REST? How to represent a resource in REST?**

REST architecture treats **every content as a resource**. REST uses various representations to represent a resource where **text, JSON, XML**. XML and JSON are the most popular representations of resources.

**Which protocol is used by RESTful webservices?**

RESTful web services make use of **HTTP** protocol as a medium of communication between client and server.

**What is messaging in RESTful webservices?**

A client sends a message in *form of a* ***HTTP Request*** and server responds in *form of a* ***HTTP Response***. This technique is termed as **Messaging**. These messages contain message data and metadata i.e. information about message itself.

**What are the core components of a HTTP Request?**

A HTTP Request has five major parts −

·        **Verb** − Indicate HTTP methods such as GET, POST, DELETE, PUT etc.

·        **URI** − Uniform Resource Identifier (URI) to identify the resource on server.

·        **HTTP** Version − Indicate HTTP version, for example HTTP v1.1 .

·        **Request Header** − Contains metadata for the HTTP Request message as key-value pairs. For example, client ( or browser) type, format supported by client, format of message body, cache settings etc.

·        **Request Body** − Message content or Resource representation.

**What are the core components of a HTTP response?**

A HTTP Response has four major parts −

·    **Status/Response Code** − Indicate Server status for the requested resource. For example 404 means resource not found and 200 means response is ok.

·   HTTP Version − Indicate HTTP version, for example HTTP v1.1 .

·    **Response Header** − Contains metadata for the HTTP Response message as key-value pairs. For example, content length, content type, response date, server type etc.

·    **Response Body** − Response message content or Resource representation.

**What is addressing in RESTful webservices? Or What is purpose of a URI in REST based webservices?**

Addressing refers to ***locating a resource or multiple resources lying on the server***. It is analogous to locate a postal address of a person.

**URI is used to define addressing.**

**What is format of a URI in REST architecture?**

A URI is of following format −

<protocol>://<service-name>/<ResourceType>/<ResourceID>

http://localhost:8082/getAllTopics

**What are the best practices to create a standard URI for a web service?**

Following are important points to be considered while designing a URI −

1. Use Plural Noun − Use plural noun to define resources. For example, we've used users to identify users as a resource.
2. Avoid using spaces − Use underscore(\_) or hyphen(-) when using a long resource name, for example, use authorized\_users instead of authorized%20users.
3. Use lowercase letters − Although URI is case-insensitive, it is good practice to keep url in lower case letters only.
4. Maintain Backward Compatibility − As Web Service is a public service, a URI once made public should always be available. In case, URI gets updated, redirect the older URI to new URI using HTTP Status code, 300.
5. Use HTTP Verb − Always use HTTP Verb like GET, PUT, and DELETE to do the operations on the resource. It is not good to use operations names in URI.

**What is statelessness in RESTful Webservices? OR**

**What are the advantages of statelessness in RESTful Webservices?**

As per REST architecture, a RESTful web service should not keep a client state on server. This restriction is called statelessness.

In simple words

·  Web services can treat each method request independently.

·  Web services need not to maintain client's previous interactions. **It simplifies application design.**

·  As HTTP is itself a statelessness protocol.

**What are the disadvantages of statelessness in RESTful Webservices?**

Web services need to get extra information in each request and then interpret to get the client's state in case client interactions are to be taken care of.

**What do you mean by idempotent operation?**

Idempotent operations means their **result will always same**, no matter how many times these operations are invoked.

**Which type of Webservices methods are to be idempotent?**

PUT and DELETE operations are idempotent.

**Which type of Webservices methods are to be read only?**

GET operations are read only and are safe.

**What is the difference between PUT and POST operations?**

PUT and POST operation are nearly same with the difference lying only in the result where PUT operation is idempotent and POST operation can cause different result(not idempotent). Simple rule of thumb: Use **POST to create**, use **PUT to update.**

What should be the purpose of **OPTIONS** **method** of RESTful web services?

It should list down the supported operations in a web service and should be read only.

What should be the purpose of **HEAD** **method** of RESTful web services?

It should return only HTTP Header, no Body and should be read only.

**What is caching?**

Caching refers to storing server response in client itself so that a client need not to make server request for same resource again and again. A server response should have information about how a caching is to be done so that a client caches response for a period of time or never caches the server response.

**Which header of HTTP response, provides the date and time of the resource when it was created?**

Date header provides the date and time of the resource when it was created.

**Which header of HTTP response, provides the date and time of the resource when it was last modified?**

Last Modified header provides the date and time of the resource when it was last modified.

**Which header of HTTP response provides control over caching?**

Cache-Control is the primary header to control caching.

W**hich header of HTTP response sets expiration date and time of caching?**

Expires header sets expiration date and time of caching.

**Which directive of Cache Control Header of HTTP response indicates that resource is cachable by any component?**

Public directive indicates that resource is cachable by any component.

**Which directive of Cache Control Header of HTTP response indicates that resource is cachable by only client and server, no intermediary can cache the resource?**

Private directive indicates that resource is cachable by only client and server, no intermediary can cache the resource.

**Which directive of Cache Control Header of HTTP response indicates that resource is not cachable?**

no-cache/no-store directive indicates that resource is not cachable.

**Which directive of Cache Control Header of HTTP response can set the time limit of caching?**

max-age directive indicates that the caching is valid up to max-age in seconds. After this, client has to make another request.

**Which directive of Cache Control Header of HTTP response provides indication to server to revalidate resource if max-age has passed?**

must-revalidate directive provides indication to server to revalidate resource if max-age has passed.

**What are the best practices for caching?**

**Always** keep static contents like images, css, JavaScript cacheable, with expiration date of 2 to 3 days. Never keep expiry date too high.

Dynamic contents should be cached for few hours only.

**What are the best practices to be followed while designing a secure RESTful web service?**

As RESTful web services work with HTTP URLs Paths so it is very important to safeguard a RESTful web service in the same manner as a website is be secured. Following are the best practices to be followed while designing a RESTful web service −

·        **Validation** − Validate all inputs on the server. Protect your server against SQL or NoSQL injection attacks.

·        Session based **authentication** − Use session based authentication to authenticate a user whenever a request is made to a Web Service method.

·        **No sensitive data in URL** − Never use username, password or session token in URL , these values should be passed to Web Service via POST method.

·        **Restriction** on Method execution − Allow restricted use of methods like GET, POST, DELETE. GET method should not be able to delete data.

·        Validate Malformed XML/JSON − Check for well formed input passed to a web service method.

·        Throw generic Error Messages − A web service method should use HTTP error messages like 403 to show access forbidden etc.

### HTTP Status Code?

HTTP Status code are standard codes and refers to predefined status of task done at server. For example, HTTP Status 404 states that requested resource is not present on server.

What HTTP Status Code **200** states?

It means, OK, shows **success**.

What HTTP Status Code **201** states?

It means, **CREATED**, when a resource is successful created using **POST or PUT** request. Return link to newly created resource using location header.

What HTTP Status Code **204** states?

It means, **NO CONTENT**, when response body is empty for example, a **DELETE** request.

What HTTP Status Code **304** states?

It means, **NOT MODIFIED**, used to reduce network bandwidth usage in case of conditional GET requests. Response body should be empty. Headers should have date, location etc.

What HTTP Status Code **400** states?

It means, **BAD REQUEST**, states that invalid input is provided e.g. validation error, missing data.

What HTTP Status Code **401** states?

It means, **FORBIDDEN**, states that user is not having access to method being used for example, delete access without admin rights.

What HTTP Status Code **404** states?

It means, **Resource** **NOT FOUND**, states that method is not available.

What HTTP Status Code 409 states?

It means, **CONFLICT**, states conflict situation while executing the method for example, adding duplicate entry.

What HTTP Status Code **500** states?

It means, **INTERNAL SERVER ERROR**, states that server has thrown some exception while executing the method.

**What is JAX-RS?**

JAX-RS stands for JAVA API for RESTful Web Services. JAX-RS is a JAVA based programming language API and specification to provide support for created RESTful Webservices. Its 2.0 version was released in 24 May 2013. JAX-RS makes heavy use of annotations available from Java SE 5 to simplify development of JAVA based web services creation and deployment. It also provides supports for creating clients for RESTful web services.

2. **What are the annotations present in Restful web services? Name** **some of the commonly used HTTP methods used in REST based architecture?**

The @GET HTTP Method Annotation

Methods annotation with the @GET annotation respond to HTTP get requests.

The @POST HTTP Method Annotation

Methods annotated @POST respond to POST method requests.

The @PUT HTTP Method Annotation

The @PUT annotation is used for updating a record and method annotated this way respond to an HTTP PUT request.

The @DELETE HTTP Method Annotation

Methods annotated @DELETE are expected to delete a resource.

6. **How to output JSON format of data in Restful services as producer.**

@POST

@Consumes(MediaType.APPLICATION\_JSON)

@Produces(MediaType.APPLICATION\_JSON)

public Response saveBook(Book book) {

    book = bookRepository.saveBook(book);

    return Response.ok(book).build();

}

@Path

The @Path annotation is used to specify the URI through which a resource and an API can be accessed.

·  Resource at the class level would have @Path("/"). This is the default annotation at the class level.

·  Resource at the API level would have @Path("Countries") annotation.

Media Type Annotations

There are two media type annotations: @Consumes and @Produces.

@Consumes

The @Consumes() annotation specifies the list of media types consumed by a particular API or class.

Examples:

·         @Consumes("application/json")

·         @Consumes("application/xml")

@Produces

The @Produces annotation specifies the list of the media types produced by a particular API or class

Examples:

·         @Produces("application/json")

·         @Produces("application/xml")

4. **Difference between query param and path param.**

# **Spring boot**

## File Upload using Spring Boot

<https://www.youtube.com/watch?v=7svczM5Xj9o>

## Spring boot data jpa: mapping strategy

FYI: <https://www.youtube.com/watch?v=f5bdUjEIbrg>

Knowledge: <https://hellokoding.com/deleting-data-with-jpa-hibernate/>

## How to preload database using CommandLineRunner

FYR: <https://spring.io/guides/tutorials/rest/>

What happens when it gets loaded?

* Create below **PreLoadDatabase** class in the same package structure of springboot main class
* Spring Boot will run ALL CommandLineRunner beans once the application context is loaded.
* This runner will request a copy of the **UserRepository**  you just created.
* Using it, it will create two entities and store them in db.
* Check related API, whether user present in db or not => <http://localhost:8082/getAllUser>

Eg:

**import** org.springframework.boot.CommandLineRunner;

**import** org.springframework.context.annotation.Bean;

**import** org.springframework.context.annotation.Configuration;

**import** com.spring.boot.entity.User;

**import** com.spring.boot.repo.UserRepo;

@Configuration

**public** **class** PreLoadDatabase {

@Bean

CommandLineRunner initDatabase(UserRepo repository) {

**return** args -> {

repository.save(**new** User(801, "sham801", "china1", 801));

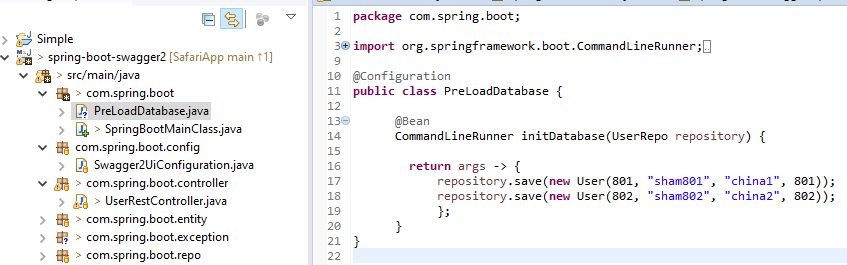
repository.save(**new** User(802, "sham802", "china2", 802));

};

}

}

FYR:



## Swagger implementation:

<https://howtodoinjava.com/swagger2/swagger-spring-mvc-rest-example/>

## **implimenting sb features (security, actuator, auto-reload,log4j, base-64, sb with mongoDB, )**

**Link to explore**:

https://www.codejava.net/frameworks/spring-boot/spring-boot-restful-crud-api-examples-with-mysql-database

·       [Spring Boot Hello World Example](https://www.codejava.net/frameworks/spring-boot/spring-boot-hello-world-example)

·       [Spring Boot automatic restart using Spring Boot DevTools](https://www.codejava.net/frameworks/spring-boot/spring-boot-automatic-restart-using-spring-boot-devtools)

·       [Spring Boot Form Handling Tutorial with Spring Form Tags and JSP](https://www.codejava.net/frameworks/spring-boot/spring-boot-form-handling-tutorial-with-spring-form-tags-and-jsp)

·       [Spring Boot Hello World RESTful Web Services Tutorial](https://www.codejava.net/frameworks/spring-boot/spring-boot-hello-world-restful-web-services-tutorial)

·       [How to create a Spring Boot Web Application (Spring MVC with JSP/ThymeLeaf)](https://www.codejava.net/frameworks/spring-boot/how-to-create-a-spring-boot-web-application-spring-mvc-with-jsp-thymeleaf)

·       [Spring Boot - Spring Data JPA - MySQL Example](https://www.codejava.net/frameworks/spring-boot/spring-boot-spring-data-jpa-mysql-example)

·       [Spring Boot CRUD Example with Spring MVC – Spring Data JPA – ThymeLeaf - Hibernate - MySQL](https://www.codejava.net/frameworks/spring-boot/spring-boot-crud-example-with-spring-mvc-spring-data-jpa-thymeleaf-hibernate-mysql)

·       [How to use JDBC with Spring Boot](https://www.codejava.net/frameworks/spring-boot/how-to-use-jdbc-with-spring-boot)

·       [Spring Boot CRUD Web Application with JDBC - Thymeleaf - Oracle](https://www.codejava.net/frameworks/spring-boot/spring-boot-crud-web-application-with-jdbc-thymeleaf-oracle)

·       [How to package Spring Boot application to JAR and WAR](https://www.codejava.net/frameworks/spring-boot/how-to-package-spring-boot-application-to-jar-and-war)

++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++

**Error: Could not find or load main class com.spring.boot.SpringBootApp**

***<!--in order to avoid above error: add below tag in pom.xml*** ***-->***

<properties>

             <start-class>com.spring.boot.SpringBootApp</start-class>

</properties>

+++++++++++++++++++++++++++++++++++++

**<!-- How to produce response in xml format : then hit api in browser n check -->**

          <dependency>

                  <groupId>com.fasterxml.jackson.dataformat</groupId>

                  <artifactId>jackson-dataformat-xml</artifactId>

            </dependency>

++++++++++++++++++++++++++++

## How to add Log4j in spring boot:

**1. add dependency to pom.xml**

<!-- logging -->

            <dependency>

            <groupId>org.springframework.boot</groupId>

            <artifactId>spring-boot-starter</artifactId>

            <exclusions>

              <exclusion>

                <groupId>org.springframework.boot</groupId>

                <artifactId>spring-boot-starter-logging</artifactId>

              </exclusion>

            </exclusions>

          </dependency>

          <dependency>

            <groupId>org.springframework.boot</groupId>

            <artifactId>spring-boot-starter-log4j2</artifactId>

        </dependency>

**2. create "log4j2.xml" under resource folder:**

<?xml version=*"1.0"* encoding=*"UTF-8"*?>

<Configuration>

  <Appenders>

    <!-- Console Appender -->

    <Console name=*"Console"* target=*"SYSTEM\_OUT"*>

      <PatternLayout pattern=*"%d{yyyy-MMM-dd HH:mm:ss a} [%t] %-5level %logger{36} - %msg%n"* />

    </Console>

    <!-- File Appender : path to access log file (file will auto generate in specified path)       important -->

    <File name=*"File"* fileName=*"d:/New folder/app.log"*>

      <PatternLayout pattern=*"%d{yyyy-MMM-dd HH:mm:ss a} [%t] %-5level %logger{36} - %msg%n"* />

    </File>

  </Appenders>

  <Loggers>

    <!-- Log everything in custom package : mention ur base package path           important -->  -->

    <Logger name=*"com.spring.boot"* level=*"debug"* additivity=*"false"*>

      <AppenderRef ref=*"Console"* />

      <AppenderRef ref=*"File"* />

    </Logger>

    <!-- Log everything in Spring Boot -->

    <Logger name=*"org.springframework.boot"* level=*"debug"* additivity=*"false"*>

      <AppenderRef ref=*"Console"* />

      <AppenderRef ref=*"File"* />

    </Logger>

    <!-- Log everything in Spring Core -->

    <Logger name=*"org.springframework.core"* level=*"debug"* additivity=*"false"*>

      <AppenderRef ref=*"Console"* />

      <AppenderRef ref=*"File"* />

    </Logger>

    <Root level=*"error"*>

      <AppenderRef ref=*"Console"* />

      <AppenderRef ref=*"File"* />

    </Root>

  </Loggers>

</Configuration>

**3.Start using inside classes wherever you want:**

Eg:

      //import

**import** org.apache.logging.log4j.LogManager;

**import** org.apache.logging.log4j.Logger;

@RestController

@RequestMapping(value="/employee")

**public** **class** EmployeeController {

**private** **static** **final** Logger ***logger*** = LogManager.*getLogger*(EmployeeController.**class**);

      @Autowired

**private** EmployeeService employeeService;

      @RequestMapping(value ="/all", method = RequestMethod.***GET***)

**public** Response getAllEmployeeList(){

        //use appropriate log message

***logger***.debug("Debugging log in controller");

***logger***.info("Info log in controller");

***logger***.warn("Hey, This is a warning! in controller ");

***logger***.error("Oops! We have an Error. OK in controller");

***logger***.fatal("Damn! Fatal error. Please fix me in controller.");

**try**{

                  //some logic

            }**catch**(Exception e){

**//some logic**

            }

      }

}

*Now ur able to see log message in : d:/New Folder/app.log*

## Package Spring Boot application to JAR file

The default packaging type of a Spring Boot project is jar if the <packaging> attribute is not present in the pom.xml file. It’s better to check your project’s pom.xml file to make sure that the packaging type is jar:

<project ...>

     ...

     <packaging>jar</packaging>

     ...

 </project>

And add a plugin:



Final name tag is nothing but giving custome name to our jar.

1. Right click on appl -> Run as -> maven build -> give a command as  "package" -> click on Run so jar will create in target folder of ur source code

2. Now open cmd : go to target folder path and give below command.

    java -jar <generated jar name>.jar

**### Package Spring Boot application to WAR file  ###**

To package a Spring Boot project to a deployable WAR file, you must do the following 2 steps first:

- Specify packaging type is **war** in the pom.xml file:

<project>

     ...

     <packaging>war</packaging>

     ...

 </project>

- Make the Spring Boot main class to extend the SpringBootServletInitializer class, for example:

@SpringBootApplication

public class HelloSpringBootApplication extends SpringBootServletInitializer {

    public static void main(String[] args) {

        SpringApplication.run(HelloSpringBootApplication.class, args);

    }

 }

Then you can package the project using the **mvn package** command in the command line, or **Run As > Maven build… > package** in Eclipse/Spring Tool Suite IDE – same as described above.

##### how to implement actuator ####

1.     in order to implement actuator add below line in : application.properties

***management.security.enabled=false***

*2.     add below dependency in pom.xml:*

*<!-- https://mvnrepository.com/artifact/org.springframework.boot/spring-boot-starter-actuator -->*

*<dependency>*

*<groupId>org.springframework.boot</groupId>*

*<artifactId>spring-boot-starter-actuator</artifactId>*

*<version>1.4.2.RELEASE</version>*

*</dependency>*

 Below are the some actuator endpoints, check it in postman/browser:

http://localhost:8080/**health**

{

    "status": "UP",

    "diskSpace": {

        "status": "UP",

        "total": 106310897664,

        "free": 24107868160,

        "threshold": 10485760

    },

    "db": {

        "status": "UP",

        "database": "MySQL",

        "hello": 1

    }

}

http://localhost:8080/**trace**

 http://localhost:8080/**beans**

        "beans": [

            {

                "bean": "springBootApp",

                "aliases": [],

                "scope": "singleton",

                "type": "com.spring.boot.SpringBootApp$$EnhancerBySpringCGLIB$$fe7efc7a",

                "resource": "null",

                "dependencies": []

            },

            {

                "bean": "employeeController",

                "aliases": [],

                "scope": "singleton",

                "type": "com.spring.boot.controller.EmployeeController",

                "resource": "file [C:/Users/sudheer.bijjaragi/workspace-mars/SpringBootRestSecurity/target/classes/com/spring/boot/controller/EmployeeController.class]",

                "dependencies": [

                    "employeeServiceImpl"

                ]

            }

http://localhost:8080/**loggers**

http://localhost:8080/**env**        ::  tells all class file path, port number, on which OS its running, what java version, which db etc

##### How to add Auto-reload configuration ####

1.    In order to implement auto-reload add below dependency into pom.xml

every changes you made to your project requires complete application restart to take effect. That really slows down your productivity as your time wasted on waiting for the application restart.

To overcome that bottleneck in Spring Boot development, the Spring team created Spring DevTools that can automatically restart the application so you don’t have to do restart manually – thus you will regain your productivity.

By default, any changes (create, update or delete) you made to any files in the project’s classpath (typically files under src/main directory) will trigger Spring DevTools to restart the application to update the changes. Actually, Spring Boot DevTools monitors changes for all entries under the project’s classpath.

When you make changes to Java code, Spring Boot DevTools will restart the embedded server to reflect the changes. It is full, automatic restart but quick enough

   <dependency>

             <groupId>org.springframework.boot</groupId>

             <artifactId>spring-boot-devtools</artifactId>

             <scope>runtime</scope>

        </dependency>

2.      Add auto-reload plugin to ur respective browser(chrome/firefox/anything) from <http://livereload.com/extensions/>

Advantages:

·      It’ll give scope for faster development.

·      No need to restart application after doing any changes in your code.

## How to add Basic spring boot security ####

Pom.xml:

<!-- springBoot security -->

           <dependency>

                <groupId>org.springframework.boot</groupId>

                <artifactId>spring-boot-starter-security</artifactId>

          </dependency>

Note 1 :

Then it will ask username & password before accessing any rest api in postman/webapplication :

By default  user name is  “**user**” and password will  “**auto generate in console**”  whenever u start the application.

Something like below :

Using generated security password: *28c2e93f-3bf3-43b4-a203-3558b1631d4f*

Note 2:

To make it specific username & password mention it in **application.properties** as below.

#security

security.user.name=sudheer

security.user.password=sudheer

Note 3:

To make Restriction based on specific API & specific role. Do the following changes

1.     Comment out specific user name & password mentioned in application.properties.

2.     Add the below class in base package(com.spring.boot.config) :

**@EnableWebSecurity**

**public** **class** SecurityConfig **extends** **WebSecurityConfigurerAdapter** {

   @Override

**protected** **void** **configure**(HttpSecurity http) **throws** Exception {

      http.csrf().disable().authorizeRequests().antMatchers("/employee/all/\*\*").hasAnyRole("admin","user").and().formLogin();

      http.csrf().disable().authorizeRequests().antMatchers("/org/all/\*\*").hasAnyRole("admin").and().formLogin();

         //”employee” & “org” are classlevel path in EmployeeController n OrgController. “all” is method level path in both controller

   }

   @Autowired

**public** **void** configureGlobal(AuthenticationManagerBuilder auth) **throws** Exception{

      auth.inMemoryAuthentication().withUser("sean").password("pass@123").roles("user");

      auth.inMemoryAuthentication().withUser("martin").password("pass@123").roles("user","admin");

   }

}

// SecurityConfig        :: This method specifies which api/methods and roles is involved for authentication .

// configureGlobal      :: This method specifies username & password wrt roles.

++++++++++++++++++++++++++++++

## How to encrypt and decrypt password

Simple implementation : <https://www.yawintutor.com/encode-decode-using-bcryptpasswordencoder-in-spring-boot-security/>

**BCryptPasswordEncoder** is a class provided by the **spring security**

There are multiple ways to implement password encrypt and decrypt:

Step 1 : add spring security dependency in pom:

<!-- spring boot security -->

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-security</artifactId>

</dependency>

Step 2:

@SpringBootApplication

**public** **class** SpringBootSwagger2Application {

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

SpringApplication.*run*(SpringBootSwagger2Application.**class**, args);

BCryptPasswordEncoder passwordEncoder = **new** BCryptPasswordEncoder();

String password = "yawinpassword";

String encodedPassword = passwordEncoder.encode(password);

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

System.***out***.println("Password is : " + password);

System.***out***.println("Encoded Password is : " + encodedPassword);

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

**boolean** isPasswordMatch = passwordEncoder.matches(password, encodedPassword);

System.***out***.println("Password : " + password + " isPasswordMatch : " + isPasswordMatch);

password = "yawin";

isPasswordMatch = passwordEncoder.matches(password, encodedPassword);

System.***out***.println("Password : " + password + " isPasswordMatch : " + isPasswordMatch);

}

}

++++++++++++++++++++++++++++++++++++

## How to convert image and store it in file  n how access same n represent as a image  using base64

SOAP with binary data process

SOAP with binary data process

**import** java.io.File;

**import** java.io.FileInputStream;

**import** java.io.FileNotFoundException;

**import** java.io.FileOutputStream;

**import** java.io.IOException;

**import** java.util.Base64;

**public** **class** Base64WithImage {

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

String imagePath = "D:\\base64\\image.jpg";

System.***out***.println("=================Encoder Image to Base 64!=================");

String base64ImageString = *encoder*(imagePath);

System.***out***.println("Base64ImageString = " + base64ImageString);

System.***out***.println("=================Decoder Base64ImageString to Image!=================");

*decoder*(base64ImageString, "D:\\base64\\decoderimage.jpg");

System.***out***.println("DONE!");

}

**public** **static** String encoder(String imagePath) {

String base64Image = "";

File file = **new** File(imagePath);

**try** (FileInputStream imageInFile = **new** FileInputStream(file))

{

// Reading a Image file from file system

**byte** imageData[] = **new** **byte**[(**int**) file.length()];

imageInFile.read(imageData);

base64Image = Base64.*getEncoder*().encodeToString(imageData);

}**catch** (FileNotFoundException e)

{

System.***out***.println("Image not found" + e);

} **catch** (IOException ioe)

{

System.***out***.println("Exception while reading the Image " + ioe);

}

**return** base64Image;

}

**public** **static** **void** decoder(String base64Image, String pathFile) {

**try** (FileOutputStream imageOutFile = **new** FileOutputStream(pathFile)) {

// Converting a Base64 String into Image byte array

**byte**[] imageByteArray = Base64.*getDecoder*().decode(base64Image);

imageOutFile.write(imageByteArray);

} **catch** (FileNotFoundException e)

{

System.***out***.println("Image not found" + e);

}

**catch** (IOException ioe) {

System.***out***.println("Exception while reading the Image " + ioe);

}

}

}

+++++++++++++++++++++++

Implimenting SpringBoot with MongoDB:

***Application.properties:***

spring.data.mongodb.host=localhost

spring.data.mongodb.database=Test1

spring.data.mongodb.port=27017

spring.data.username=bolt

spring.data.password=bolt

***pom.xml:***

<project xmlns=*"http://maven.apache.org/POM/4.0.0"* xmlns:xsi=*"http://www.w3.org/2001/XMLSchema-instance"* xsi:schemaLocation=*"http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd"*>

  <modelVersion>4.0.0</modelVersion>

  <groupId>com.brillio.in2</groupId>

  <artifactId>SpringBootAssociateMapping</artifactId>

  <version>0.0.1-SNAPSHOT</version>

       <parent>

           <groupId>org.springframework.boot</groupId>

           <artifactId>spring-boot-starter-parent</artifactId>

           <version>1.5.6.RELEASE</version>

       </parent>

        <dependencies>

           <dependency>

               <groupId>org.springframework.boot</groupId>

               <artifactId>spring-boot-starter-web</artifactId>

           </dependency>

          <!-- for Response Entity feature in controller class -->

         <dependency>

             <groupId>com.sun.jersey</groupId>

                        <artifactId>jersey-bundle</artifactId>

                        <version>1.17.1</version>

                  </dependency>

           <dependency>

                <groupId>org.springframework.boot</groupId>

                <artifactId>spring-boot-starter-data-mongodb</artifactId>

                <version>1.5.6.RELEASE</version>

             </dependency>

  </dependencies>

        <properties>

       <java.version>1.8</java.version>

       </properties>

</project>

***//Entity class :***

**import** org.springframework.data.mongodb.core.mapping.Document;

@Document(collection = "Employee")

**public** **class** Employee {

**private** Integer emp\_id;

**private** String emp\_name;

      //setters n getters

+++++++++++++++++++++++++++++++++++++++++

How to use JDBC with Spring Boot

For simple database operations, using JDBC is easier and more efficiently than using Hibernate.

Spring framework provides the JdbcTemplate class that simplifies coding with JDBC

**Create Database:**

CREATE TABLE `books` (

  `id` int(11) NOT NULL AUTO\_INCREMENT,

  `title` varchar(45) NOT NULL,

  `author` varchar(45) NOT NULL,

  `price` float NOT NULL,

  PRIMARY KEY (`id`)

);

**Add below 2 dependency in pom.xml:**

<dependencies>

        <dependency>

            <groupId>org.springframework.boot</groupId>

            <artifactId>spring-boot-starter-jdbc</artifactId>

        </dependency>

        <dependency>

            <groupId>mysql</groupId>

            <artifactId>mysql-connector-java</artifactId>

            <scope>runtime</scope>

        </dependency>

    </dependencies>

**Configure database in application.properies file:**

spring.datasource.url=jdbc:mysql://localhost:3306/bookshop

spring.datasource.username=root

spring.datasource.password=password

logging.level.root=INFO

**Create a model class:**

**Note** : this model class has field names exactly match the column names in the corresponding table.

public class Book {

    private int id;

    private String title;

    private String author;

    private float price;

    protected Book(String title, String author, float price) {

        this.title = title;

        this.author = author;

        this.price = price;

   }

    protected Book() {

    }

    @Override

    public String toString() {

        return "Book [" + title + ", " + author + ", $" + price + "]";

    }

    // getters and setters...

}

**Code Spring Boot JDBC Application :**

import org.springframework.beans.factory.annotation.Autowired;

import org.springframework.boot.CommandLineRunner;

import org.springframework.boot.SpringApplication;

import org.springframework.boot.autoconfigure.SpringBootApplication;

import org.springframework.jdbc.core.JdbcTemplate;

@SpringBootApplication

public class SpringBootJdbcExample implements CommandLineRunner {

    @Autowired

    JdbcTemplate jdbcTemplate;

    public static void main(String[] args) {

        SpringApplication.run(SpringBootJdbcExample.class, args);

    }

    @Override

    public void run(String... args) throws Exception {

        String sql = "INSERT INTO books (title, author, price) VALUES (?, ?, ?)";

        int result = jdbcTemplate.update(sql, "Head First Java", "Kathy Sierra", 18.55f);

        if (result > 0) {

            System.out.println("Insert successfully.");

        }

    }

}

**Note** : As you can see, we specify that an instance of the JdbcTemplate class will be automatically injected by the framework:

This is the key point to use JDBC with Spring Boot. And in the run() method, we use the jdbcTemplate object to insert a new row into the database

You can also use SimpleJdbcInsert class for SQL Insert statement like this:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11 | SimpleJdbcInsert insertActor = new SimpleJdbcInsert(jdbcTemplate);  insertActor.withTableName("books");    Book book = new Book("Effective Java", "Joshua Bloch", 29.99f);  BeanPropertySqlParameterSource paramSource = new BeanPropertySqlParameterSource(book);    int result = insertActor.execute(paramSource);    if (result > 0) {      System.out.println("Insert successfully.");  } |

As you can see, you don’t have to write SQL statement! Learn more about how to use the SimpleJdbcInsert class here: [Spring SimpleJdbcInsert Examples](https://www.codejava.net/frameworks/spring/spring-simplejdbcinsert-examples).

And here’s the sample code for executing Update operation:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | void testUpdate() {      String sql = "UPDATE books SET price=? WHERE title=?";      Object[] params = {9.99f, "Effective Java"};      int result = jdbcTemplate.update(sql, params);        if (result > 0) {          System.out.println("Update successfully.");      }  } |

Code example for Delete operation:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | String sql = "DELETE FROM books WHERE title=?";  Object[] params = {"Effective Java"};  int result = jdbcTemplate.update(sql, params);    if (result > 0) {      System.out.println("Delete successfully.");  } |

Code example for Retrieve operation (list all):

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10 | void testListAll() {      String sql = "SELECT \* FROM books";        List<Book> listBooks = jdbcTemplate.query(sql,                  BeanPropertyRowMapper.newInstance(Book.class));        for (Book book : listBooks) {          System.out.println(book);      }  } |

Code example for Retrieve operation (find):

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | void testFind() {      String sql = "SELECT \* FROM books WHERE id = 1";      Book book = jdbcTemplate.queryForObject(sql,          BeanPropertyRowMapper.newInstance(Book.class));        System.out.println(book);  } |

+++++++++++++++++++++++++++++++++++++++++

**How to read request parameter**

Update the hello() method as below:

|  |  |
| --- | --- |
|  | @RequestMapping("/hello")  public String hello(@RequestParam(name = "name", defaultValue = "World") String name) {      return "Hello " + name;  } |

The @RequestParam annotation binds the value of the request parameter named “name” to a parameter of this method. If the query parameter is absent, then the default value “world” is used.

You must stop the application and run again. Type the following command to test:

|  |  |
| --- | --- |
| 1 | curl http://localhost:8080/hello?name=Nam |

The server responds:

**How to handle different HTTP methods**

Basically you can specify a specific HTTP method using the method attribute of the @RequestMapping annotation (default is GET), for example:

|  |  |
| --- | --- |
| 1 | @RequestMapping(value = "/hello", method = RequestMethod.POST) |

To simplify RESTful APIs development, Spring provides the annotations @GetMapping, @PostMapping, @PutMapping and @DeleteMapping corresponding to HTTP methods GET, POST, PUT and DELETE. So a typical REST controller will look like this:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18 | @RestController  public class ResourceController {        @GetMapping("/resources")      // method that returns all items        @GetMapping("/resources/{id}")      // method that returns a specific item        @PostMapping("/resources")      // method that creates a new item        @PutMapping("/resources/{id}")      // method that updates an item        @DeleteMapping("/resources/{id}")      // method that deletes an item  } |

+++++++++++++++++++++++++++++++++++++++++

## **springboot interview que**

**1) What is Spring Boot?**

Spring Boot is a Spring module which provides RAD (Rapid Application Development) feature to Spring framework.

It is used to create stand alone spring based application that you can just run because it needs very little spring configuration.

**2) What are the advantages OR features of Spring Boot?**

1. Spring CLI – Spring Boot CLI allows you to Groovy for writing Spring boot application and avoids boilerplate code.
2. Starter Dependency – With the help of this feature, Spring Boot aggregates common dependencies together and eventually improves productivity
3. Auto-Configuration – The auto-configuration feature of Spring Boot helps in loading the default configurations according to the project you are working on. In this way, you can avoid any unnecessary WAR files.
4. Spring Initializer – This is basically a web application, which can create an internal project structure for you. So, you do not have to manually set up the structure of the project, instead, you can use this feature.
5. Spring Actuator –  This feature provides help while running Spring Boot applications.
6. Logging and Security – The logging and security feature of Spring Boot, ensures that all the applications made using Spring Boot are properly secured without any hassle.
7. Integrates with Spring Ecosystem like Spring [JDBC](https://www.edureka.co/blog/connect-mysql-database-in-java), Spring ORM, Spring Data, Spring Security easily by avoiding boilerplate code.

**4) How to create Spring Boot application using Maven?**

* Spring Boot CLI
* Spring Starter Project Wizard (STS)
* Spring Initializr
* Spring Maven Project

**8) What are the Spring Boot Annotations?**

**9) What is Spring Boot dependency management?**

Spring Boot manages dependencies and configuration automatically. You don't need to specify version for any of that dependencies.

Spring Boot upgrades all dependencies automatically when you upgrade Spring Boot.

**10) What are the Spring Boot properties?**

Spring Boot provides various properties which can be specified inside our project's **application.properties** file. These properties have default values and you can set that inside the properties file. Properties are used to set values like: **server-port number, database connection configuration** etc.

**Can you explain what happens in the background when a Spring Boot Application is “Run as Java Application”?**

When a Spring Boot application is executed as “Run as Java application”, then it automatically launches up the tomcat server as soon as it sees, that you are developing a web application.

**Mention the possible sources of external configuration.**

* **Application Properties –** By default, Spring Boot searches for the application properties file or its YAML file in the current directory, classpath root or config directory to load the properties.
* **Command-line properties –** Spring Boot provides command-line arguments and converts these arguments to properties. Then it adds them to the set of environment properties.

**11) What are the Spring Boot Starters? OR What is Spring Boot dependency management?**

Spring Boot starters are a set of convenient dependency management providers, which makes development easier and rapid. For example, if we want to get started using Spring and JPA for database access, just include the **spring-boot-starter-data-jpa** dependency in your project. Similarly

* spring-boot-starter: – This is the core starter and includes logging, auto-configuration support, and YAML.
* spring-boot-starter-jdbc – This starter is used for HikariCP connection pool with JDBC
* spring-boot-starter-web – Is the starter for building web applications, including RESTful, applications using Spring MVC
* spring-boot-starter-data-jpa – Is the starter to use Spring Data JPA with Hibernate
* spring-boot-starter-security – Is the starter used for Spring Security
* spring-boot-starter-aop: This starter is used for aspect-oriented programming with AspectJ and  Spring AOP
* spring-boot-starter-test: Is the starter for testing Spring Boot applications

Mention the minimum requirements for a Spring boot System.

* Java 8 +
* Spring Framework 5.1.9 +

**Explicit build support**

* Maven 3.3+
* Gradle 4.4+
* Tomcat 9.0 – Servlet Version 4.0

**12) What is Spring Boot Actuator?**

Spring Boot provides actuator to **monitor and manage** our application. Actuator is a tool which has HTTP endpoints. when application is pushed to production, you can choose to manage and monitor your application using HTTP endpoints.

Such as Spring Actuator provides easy access to features such as identifying beans, CPU usage, etc.

**13) What is thymeleaf?**

It is a **server side Java template engine for web application**. It's main goal is to bring elegant natural templates to your web application.

It can be integrate with Spring Framework and ideal for HTML5 Java web applications.

**14) How to use thymeleaf?**

In order to use Thymeleaf we must add it into our pom.xml file like:

**<dependency>**

**<groupId>**org.springframework.boot**</groupId>**

**<artifactId>**spring-boot-starter-thymeleaf**</artifactId>**

**</dependency>**

Mention the steps to create a Spring Boot project using Spring  Initializer.

Spring Initializr is a web tool provided by Spring. With the help of this tool, you can create Spring Boot projects by just providing project details. The following steps need to be followed to create a Spring Boot project using Spring Initializer:

* Choose the maven project and the required dependencies. Then, fill in the other required details like Group, Artifact, and then click on Generate Project.
* Once the project is downloaded, extract the project onto your system
* Next, you have to import this project using the import option on the Spring Tool Suite IDE
  + While importing the project, remember that you have to choose the project type to be Maven and the source project should contain the pom.xml file.

Once, all the above steps are followed you will see that the Spring Boot project is created with all the required dependencies.

**15) How to connect Spring Boot to the database using JPA?**

Spring Boot provides **spring-boot-starter-data-jpa** starter to connect Spring application with relational database efficiently. You can use it into project POM (Project Object Model) file.

For more information [click here.](https://www.javatpoint.com/spring-boot-jpa)

**16) How to connect Spring Boot application to database using JDBC?**

* + Create the database and table
  + Will add database configuration info in property file
  + Will add related dependency in pom.xml
  + create a controller to handle the HTTP requests,

How to enable HTTP/2 support in Spring Boot?

You can enable the HTTP/2 support in Spring Boot by: server.http2.enabled=true

**18) What is @GetMapping @RequestMapping annotation in Spring Boot?**

Primarily both annotation is used to provide **routing information**. And both are method level annotations.

@GetMapping we can use with only Get method

@RequestMapping can use with @GET | @POST |@PUT |Delete

**+++++++++++++++++      spring      +++++++++++++++++++++++++**

What happen if you not use  @RequestBody while passing object in method signature?

public Response encryptPassword(Register register){

you will get below error:

*"status": 500,*

*"error": "Internal Server Error",*

*"exception": "java.lang.NullPointerException",*

use like this:

public Response encryptPassword(@RequestBody Register register){

+++++++++++++++++

## **Difference between @RequestParam and @PathVariable:**

 @**RequestParams** extract values from the query string,

*@RequestMapping(value ="/decodePassword/login", method = RequestMethod.GET)*

*public Response decryptPassword(@RequestParam("userName") String userName, @RequestParam("password") String password ){*

url will be like this:

          http://localhost:8080/register/decodePassword/login?userName=sudheer5

@**PathVariable** is used to handle dynamic value changes in URI

*@RequestMapping(path="/getOrgnisationByPathVariable/{id}", method = RequestMethod.GET)*

*public Response getOrgnisationByPathVariable(@PathVariable("id") Integer org\_id)*

*{*

url will be like this:

http://localhost:8080/orgnisation/getOrgnisationByPathVariable/1

Note: we have to pass "@RequestParam value same as respective entity class property name" else it  throws bad request.

 (cz we are doing operations like findById,findByUserName in respective entity  class service, dao layers)

What are the @RequestMapping  and @RestController annotation in Spring Boot used for?

|  |  |
| --- | --- |
| **@RequestMapping** | **@RestController** |
| This annotation is used to provide the **routing information** and tells to Spring that any HTTP request must be mapped to the respective method. | This annotation is used to add the @ResponseBody and @Controller annotation to the class |
| To use this annotation, you have to import org.springframework.web.  bind.annotation.RequestMapping; | To use this annotation, you have to import org.springframework.web.  bind.annotation.RestController; |

**How can we create a custom endpoint in Spring Boot Actuator?**

To create a custom endpoint in Spring Boot 2.x, you can use the **@Endpoint annotation**. Spring Boot also exposes endpoints using @WebEndpointor, @WebEndpointExtension over HTTP with the help of [Spring MVC](https://www.edureka.co/blog/spring-mvc-tutorial/), [Jersey](https://www.edureka.co/blog/java-web-services-tutorial/), etc.

Q22. **Explain Spring Data.**

Spring Data aims to make it easy for the developers **to use relational and non-relational databases, cloud-based data services**, and other data access technologies. So, basically, it makes it easy for data access and still retains the underlying data.

Q23**. What do you understand by auto-configuration in Spring Boot and how to disable the auto-configuration?**

Auto-configuration is used to automatically configure the required configuration for the application. For example, if you have a data source bean present in the classpath of the application, then it automatically configures the [JDBC template](https://www.edureka.co/blog/connect-mysql-database-in-java). With the help of auto-configuration, you can create a Java application in an easy way, as it automatically configures the required beans, controllers, etc.

To disable the auto-configuration property, you have to exclude attribute of @EnableAutoConfiguration, in the scenario where you do not want it to be applied.

|  |  |
| --- | --- |
| 1 | @EnableAutoConfiguration(exclude={DataSourceAutoConfiguration.class}) |

If the class is not on the classpath, then to exclude the auto-configuration, you have to mention the following code:

|  |  |
| --- | --- |
| 1 | @EnableAutoConfiguration(excludeName={Sample.class}) |

Apart from this, Spring Boot also provides the facility to exclude list of auto-configuration classes by using the spring.autoconfigure.exclude property. You can go forward, and add it either in the application.properties or add multiple classes with comma-separated.

What are the differences between @SpringBootApplication and @EnableAutoConfiguration annotation?

|  |  |
| --- | --- |
| **@SpringBootApplication** | **@EnableAutoConfiguration** |
| Used in the main class or bootstrap class | Used to enable auto-configuration  and component scanning in your project |
| It is a combination of @Configuration, @ComponentScan and @EnableAutoConfiguration annotations. | It is a combination of @Configuration and @ComponentScan annotations |

**diff between @Bean @Component ?**

|  |  |  |  |
| --- | --- | --- | --- |
| 1 | Class/Method  Level Annotation | It is a method level annotation | It is a class level annotation. |
| 2 | Auto detection | It is used to explicitly declare a single bean, rather than letting Spring do it automatically. | If any class is annotated with @Component it will be automatically detect by using classpath scan. |
| 3 | Spring Container | Bean can be created even class is outside the spring container | We can’t create bean if class is outside spring container. |
| 4 | @Configuration | It works only when class is also annotated with @Configuration | It works without @Configuration annotation. |
| 5 | Use Case | We should use @bean, if you want specific implementation based on dynamic condition.  (we use to preload the database using CommanLineRunner, to have some data in db, after launching application ) | We can’t write specific implementation based on dynamic condition |

Q) Related for point 3 and 5

Lets take one example, we have one external jar, we need to configute into our spring application. At that time which annotation we can go for it?

Ans: @Bean

Some eg screen shots of using @Bean:





* @**Configuration** marks one or more beans defined inside the class.
* We can still use @**component** in place of @**Service** annotation in service layer. But still using @Service use actual intent of service better.

**What are the steps to deploy Spring Boot web applications as JAR and WAR files?**

Add plugin in pom.xml

Add jar/war as packaging

**Can you explain how to deploy to a different server with Spring Boot?**

To deploy a different server with Spring Boot, follow the below steps:

* Generate a WAR from the project
* Then, deploy the WAR file onto your favorite server : tomcat : webapp folder

**Mention the advantages of the YAML file than Properties file and the different ways to load YAML file in Spring boot.**

The advantages of the YAML file than a properties file is that the data is stored in a **hierarchical format**. So, it becomes very easy for the developers to debug if there is an issue.

How is Hibernate chosen as the default implementation for JPA without any configuration?

When we use the Spring Boot Auto Configuration, automatically the spring**-boot-starter-data-jpa**dependency gets added to the pom.xml file. Now, since this dependency has a transitive dependency on JPA and Hibernate, Spring Boot automatically auto-configures Hibernate as the default implementation for JPA, whenever it sees Hibernate in the classpath.

What do you understand by Spring Data REST?

Spring Data REST is used to expose the RESTful resources around Spring Data repositories

What is the error you see if  H2 is not in the classpath?

If H2 is not present in the classpath, then you see the following error:

Cannot determine embedded database driver class for database type NONE

To resolve this error, add H2 to the pom.xml file, and restart your server.  
The following code snippet can be added to add the dependency:

<dependency>

<groupId>com.h2database</groupId>

<artifactId>h2</artifactId>

<scope>runtime</scope>

</dependency>

**What is the way to use profiles to configure the environment-specific configuration with Spring Boot?**

Ref: <https://www.youtube.com/watch?v=vHnOwhSUTAo>

<https://www.youtube.com/watch?v=Op4SRM3t0nw>

Profile is nothing but a key to identify an environment lets consider the following two profiles in the example:

* dev
* prod
* Consider the following properties present in the application properties file:

example.number: 100  
example.value: true  
example.message: Dynamic Message

Now, say you want to customize the application.properties for dev profile, then you need to create a file with name application-dev.properties and override the properties that you want to customize. You can mention the following code:

example.message: Dynamic Message in Dev

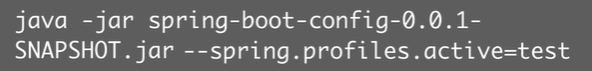
Similarly, if you want to customize the application.properties for prod profile, then you can mention the following code snippet:

example.message: Dynamic Message in Prod

Once you are done with the profile-specific configuration, you have to set the active profile in an environment. To do that, either you can

* Use -Dspring.profiles.active=prod in  arguments
* Use **spring.profiles.active=prod** in application.properties file

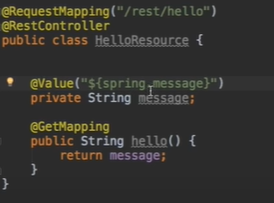
//How to point property file into jar :





How to access property file value:

//Rest method



//property file



//browser



Where is the database connection information specified and how does it automatically connect to H2?

Well, the answer to this question is very simple. It is because of the Spring Boot auto-configuration that, configures the dependencies of the application. So, the database connection information, and automatically connecting the database to H2 is done by the **auto-configuration property.**

Q49. What is the name of the default H2 database configured by Spring Boot?

The name of the default H2 database is **testdb.**

spring.datasource.name=testdb # Name of the datasource.

Do you think, you can use jetty instead of tomcat in spring-boot-starter-web?

Yes, we can use jetty instead of tomcat in spring-boot-starter-web, by removing the existing dependency and including the following:

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-web</artifactId>

<exclusions>

<exclusion>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-tomcat</artifactId>

</exclusion>

</exclusions>

</dependency>

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-jetty</artifactId>

</dependency>

## List of Annotations

* **@GetMapping:** It maps the **HTTP GET** requests on the specific handler method. It is used to create a web service endpoint that **fetches** It is used instead of using: **@RequestMapping(method = RequestMethod.GET)**
* **@PostMapping:** It maps the **HTTP POST**requests on the specific handler method. It is used to create a web service endpoint that **creates** It is used instead of using: **@RequestMapping(method = RequestMethod.POST)**
* **@PutMapping:** It maps the **HTTP PUT** requests on the specific handler method. It is used to create a web service endpoint that **creates** or **updates** It is used instead of using: **@RequestMapping(method = RequestMethod.PUT)**
* **@DeleteMapping:** It maps the **HTTP DELETE** requests on the specific handler method. It is used to create a web service endpoint that **deletes**a resource. It is used instead of using: **@RequestMapping(method = RequestMethod.DELETE)**
* **@PatchMapping:** It maps the **HTTP PATCH**requests on the specific handler method. It is used instead of using: **@RequestMapping(method = RequestMethod.PATCH)**
* **@RequestBody:** It is used to **bind** HTTP request with an object in a method parameter. Internally it uses **HTTP MessageConverters** to convert the body of the request. When we annotate a method parameter with **@RequestBody,** the Spring framework binds the incoming HTTP request body to that parameter.
* **@ResponseBody:** It binds the method return value to the response body. It tells the Spring Boot Framework to serialize a return an object into JSON and XML format.
* **@PathVariable:** It is used to extract the values from the URI. It is most suitable for the RESTful web service, where the URL contains a path variable. We can define multiple @PathVariable in a method.
* **@RequestParam:** It is used to extract the query parameters form the URL. It is also known as a **query parameter**. It is most suitable for web applications. It can specify default values if the query parameter is not present in the URL.
* **@RequestHeader:** It is used to get the details about the HTTP request headers. We use this annotation as a **method parameter**. The optional elements of the annotation are **name, required, value, defaultValue.**For each detail in the header, we should specify separate annotations. We can use it multiple time in a method
* **@RestController:** It can be considered as a combination of **@Controller** and **@ResponseBody**annotations**.** The @RestController annotation is itself annotated with the @ResponseBody annotation. It eliminates the need for annotating each method with @ResponseBody.
* **@RequestAttribute:** It binds a method parameter to request attribute. It provides convenient access to the request attributes from a controller method. With the help of @RequestAttribute annotation, we can access objects that are populated on the server-side.

+++++++++++++++++++++++++++++++++++++++

how to handle image request in spring boot

https://grokonez.com/java-integration/transfer-image-restfulapi-image-restfulapi

++++++++++++++++++

**How to implement custom queries with spring boot.**

//Using @Query, Criteria, Named query

**Note**: spring boot we’ll declare custom query in DAO interface  only not in DAOImpl class.

Eg:

**public** **interface** OrgServiceDao **extends** CrudRepository<Org,Integer> {

      /\*    custom queries  ::

       \* fetching org list in descending order

      \* \*/

      @Query("select o from Org o order by o.id desc")

      List<Org> findAllOrgListDecreasingOrder();

      /\*    custom queries  ::

      \* fetching based on like matches \*/

      @Query("select o from Org o where o.org\_name like %:name%")

      List<Org> getOrgListBasedOnMatchingName(@Param("name")String name);

      /\*

      fetching name based on where clause \*/

      @Query("select o from Org o where o.org\_name =:name")

List<Org> getOrgListBasedOnMatchingName(@Param("name")String name);

Link for reference:

https://www.petrikainulainen.net/programming/spring-framework/spring-data-jpa-tutorial-three-custom-queries-with-query-methods/

https://docs.spring.io/spring-data/jpa/docs/current/reference/html/#jpa.query-methods.at-query

 ++++++++++++++++

How to handle JSON data in java?

          It’s required to handle JSON data whenever ur going to consume other REST API in ur java class.

Eg:

Test obj1 = **new** Test();

Obj1.setId(101);

Obj1.setName("puma");

Gson gson = **new** Gson();     **//Gson is a class, comes under "com.google.gson" package**

//convert java object to JSON format

String json = gson.toJson(obj1);

System.***out***.println("java object into JSON format \n"+json);

//convert JSON format to java object

Test obj2=gson.fromJson(json, Test.**class**);

System.***out***.println("\n JSON to Java Object conversion \n"+obj2.getData1()+" "+obj2.getData2());

++++++++++++++++++++++

Rest call session management:

+++++++++++++++++++++++++++

## Difference between spring and spring boot

|  |  |
| --- | --- |
| **Spring** | **Spring Boot** |
| **Spring Framework** is a widely used Java EE framework for building applications. | **Spring Boot Framework** is widely used to develop **REST APIs**. |
| To test the Spring project, we need to set up the sever explicitly. | Spring Boot offers **embedded server** such as **Jetty** and **Tomcat**, etc. |
| It does not provide support for an in-memory database. | It offers several plugins for working with an embedded and **in-memory** database such as **H2**. |
| Developers manually define dependencies for the Spring project in **pom.xml**. | Spring Boot comes with the concept of **starter** in pom.xml file that internally takes care of downloading the dependencies **JARs** based on Spring Boot Requirement. |
| The developer writes a lot of code (boilerplate code) to do the minimal task. | It reduces boilerplate code. |

## Difference between @controller and @restcontroller

* 1. @Controller annotation typicaly used in **spring MVC** where as @RestController used with developing **RestFul web services** in springboot

* 1. In spring MVC we use explicitly **@ResponseBody** on every handler method but in springboot @RestController is a combination of the @Controller and the @ResponseBody annotation. So no need to define explicitly @ResponseBody

@Controller

public class Book{

@RequestMapping(value={"/book"})

@ResponseBody

public Book getBook(){

//...

return book;

}

}

@RestController

public class Book{

@RequestMapping(value={"/book"})

public Book getBook(){

//...

return book;

}

}

1. The job of the @Controller is to create a Map of model object and find a view but the @RestController simply returns the object and object data is directly written into HTTP response as JSON or XML.
2. @Controller is an old annotation, added on Spring 2.5 version.The @RestController is relatively new, added only on Spring 4.0

# Hibernate

## Theory :

Java JDBC is a java API to connect and execute query with the database. JDBC API uses jdbc drivers to connect with the database

Java Application ---🡪 jdbc driver ------🡪 database

JDBC Driver

|  |
| --- |
| JDBC Driver is a software component that enables java application to interact with the database.There are 4 types of JDBC drivers:   1. JDBC-ODBC bridge driver 2. Native-API driver (partially java driver) 3. Network Protocol driver (fully java driver) 4. Thin driver (fully java driver)   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. Register the driver class
2. Creating connection
3. Creating statement
4. Executing queries
5. Closing connection
6. forName() method of Class class is used to register the driver class
7. getConnection() method of DriverManager class is used to establish connection with the database.
8. createStatement() method of Connection interface is used to create statement
9. 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.
10. The close() method of Connection interface is used to close the connection.

Example:

**import** java.sql.\*;

**class** OracleCon{

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

**try**{

//step1 load the driver class

Class.forName("oracle.jdbc.driver.OracleDriver");

//step2 create  the connection object

Connection con=DriverManager.getConnection(

"jdbc:oracle:thin:@localhost:1521:xe","system","oracle");

//step3 create the statement object

Statement stmt=con.createStatement();

//step4 execute query

ResultSet rs=stmt.executeQuery("select \* from emp");

**while**(rs.next())

System.out.println(rs.getInt(1)+"  "+rs.getString(2)+"  "+rs.getString(3));

//step5 close the connection object

con.close();

}**catch**(Exception e){ System.out.println(e);}

}

}

The PreparedStatement interface is a subinterface of Statement. It is used to execute parameterized query.

Let's see the example of parameterized query:

String sql="insert into emp values(?,?,?)";

1) What is hibernate?

Hibernate is an open-source and lightweight ORM tool that is used to store, manipulate and retrieve data from the database.

2) What is ORM?

ORM is an acronym for Object/Relational mapping. It is a programming strategy to map object with the data stored in the database. It simplifies data creation, data manipulation and data access.

3. Name some of the databases that hibernate supports.

Hibernate supports almost all the major RDBMS. Following is list of few of the database

HSQL Database Engine , DB2/NT , MySQL , PostgreSQL , FrontBase , Oracle

Microsoft SQL Server Database , Sybase SQL Server , Informix Dynamic Server

4) What are the core interfaces(key components) of Hibernate?

* Configuration
* SessionFactory
* Session
* Query
* Criteria
* Transaction

What are persistent classes in hibernate?

Java classes whose objects or instances will be stored in database tables are called persistent classes in Hibernate.

What are the advantages of using Hibernate?

What is HQL (Hibernate Query Language)?

Hibernate Query Language is known as an object oriented query language. It is like structured query language (SQL).

1. You don't need to learn SQL
2. Database independent
3. Simple to write query
4. Automatic table creation (hbm.2ddl.auto)

What are the two key components of a hibernate configuration object?

**Database Connection** − This is handled through one or more configuration files supported by Hibernate. These files are **hibernate.properties** and **hibernate.cfg.xml**.

**Class Mapping Setup**

This component creates the connection between the Java classes and database tables.

What is a configuration object in hibernate?

The Configuration object is the first Hibernate object you create in any Hibernate application and usually created only once during application initialization. It represents a configuration or properties file required by the Hibernate.

What is a SessionFactory in hibernate?

Configuration object is used to create a SessionFactory object which inturn configures Hibernate for the application using the supplied configuration file and allows for a Session object to be instantiated. The SessionFactory is a thread safe object and used by all the threads of an application.

The SessionFactory is heavyweight object so usually it is created during application start up and kept for later use. You would need one SessionFactory object per database using a separate configuration file. So if you are using multiple databases then you would have to create multiple SessionFactory objects.

What is Session in hibernate?

A Session is used to get a physical connection with a database. The Session object is lightweight and designed to be instantiated each time an interaction is needed with the database. Persistent objects are saved and retrieved through a Session object.

The session objects should not be kept open for a long time because they are not usually thread safe and they should be created and destroyed them as needed.

What is Transaction in hibernate?

A Transaction represents **a unit of work with the database** and most of the RDBMS supports transaction functionality. Transactions in Hibernate are handled by an underlying transaction manager and transaction (from JDBC or JTA).

This is an optional object and Hibernate applications may choose not to use this interface, instead managing transactions in their own application code.

Name some of the properties you would require to configure for a databases in a standalone situation.

|  |  |
| --- | --- |
| **Sr.No.** | **Properties & Description** |
| 1 | **hibernate.dialect**  This property makes Hibernate generate the appropriate SQL for the chosen database. |
| 2 | **hibernate.connection.driver\_class**  The JDBC driver class. |
| 3 | **hibernate.connection.url**  The JDBC URL to the database instance. |
| 4 | **hibernate.connection.username**  The database username. |
| 5 | **hibernate.connection.password**  The database password. |
| 6 | **hibernate.connection.pool\_size**  Limits the number of connections waiting in the Hibernate database connection pool. |
| 7 | **hibernate.connection.autocommit**  Allows autocommit mode to be used for the JDBC connection. |

<?xml version = "1.0" encoding = "utf-8"?>

<!DOCTYPE hibernate-configuration SYSTEM

"http://www.hibernate.org/dtd/hibernate-configuration-3.0.dtd">

<hibernate-configuration>

<session-factory>

<property name = "hibernate.dialect">

org.hibernate.dialect.MySQLDialect

</property>

<property name = "hibernate.connection.driver\_class">

com.mysql.jdbc.Driver

</property>

<!-- Assume test is the database name -->

<property name = "hibernate.connection.url">

jdbc:mysql://localhost/test

</property>

<property name = "hibernate.connection.username">

root

</property>

<property name = "hibernate.connection.password">

root123

</property>

<!-- List of XML mapping files -->

<mapping resource = "Employee.hbm.xml"/>

</session-factory>

</hibernate-configuration>

//in property file

# Hibernate ddl auto (create, create-drop, validate, update)

spring.jpa.hibernate.ddl-auto = update

What are the three states of a persistent entity at a given point in time?

Instances may exist in one of the following three states at a given point in time −

**transient** − A new instance of a a persistent class which is not associated with a Session and has no representation in the database and no identifier value is considered transient by Hibernate.

**persistent** − You can make a transient instance persistent by associating it with a Session. A persistent instance has a representation in the database, an identifier value and is associated with a Session.

**detached** − Once we close the Hibernate Session, the persistent instance will become a detached instance.

What is the purpose of Session.beginTransaction() method?

Session.beginTransaction method begins a unit of work and returns the associated Transaction object.

Which method is used to add a criteria to a query?

Session.createCriteria creates a new Criteria instance, for the given entity class, or a superclass of an entity class.

Which method is used to create a HQL query?

Session.createQuery creates a new instance of Query for the given HQL query string.

Which method is used to create a SQL query?

Session.createSQLQuery creates a new instance of SQLQuery for the given SQL query string.

Which method is used to remove a persistent instance from the datastore?

Session.delete removes a persistent instance from the datastore.

Which method is used to get a persistent instance from the datastore?

Session.get returns the persistent instance of the given named entity with the given identifier, or null if there is no such persistent instance.

Which method is used to re-read the state of the given instance from the underlying database?

Session.refresh re-reads the state of the given instance from the underlying database.

Which method is used to save the state of the given instance from the underlying database?

Session.save saves the state of the given instance from the underlying database.

Which method is used to update the state of the given instance from the underlying database?

Session.update updates the state of the given instance from the underlying database.

Which method is used to save or update the state of the given instance from the underlying database?

Session.saveOrUpdate either saves(Object) or updates(Object) the given instance.

What are the best practices that hibernate recommends for persistent classes.

1. All Java classes that will be persisted need a **default constructor**.
2. All classes should contain an **ID** in order to allow easy identification of your objects within Hibernate and the database. This property maps to the primary key column of a database table.
3. All attributes that will be persisted should be declared private and have **getXXX** and **setXXX** methods defined in the JavaBean style.
4. All classes that do not extend or implement some specialized classes and interfaces required by the EJB framework

Where Object/relational mappings are defined in hibernate?

An Object/relational mappings are usually defined in an XML document. This mapping file instructs Hibernate how to map the defined class or classes to the database tables. We should save the mapping document in a file with the format <classname>.hbm.xml.

Which element of hbm.xml defines a specific mappings from a Java classes to the database tables?

The **<class>** elements are used to define specific mappings from a Java classes to the database tables. The Java class name is specified using the **name** attribute of the class element and the database table name is specified using the **table** attribute.

Which element of hbm.xml defines maps the unique ID attribute in class to the primary key of the database table?

The **<id>** element maps the unique ID attribute in class to the primary key of the database table. The **name** attribute of the id element refers to the property in the class and the **column** attribute refers to the column in the database table. The **type** attribute holds the hibernate mapping type, this mapping types will convert from Java to SQL data type.

Which element of hbm.xml is used to automatically generate the primary key values?

The **<generator>** element within the id element is used to automatically generate the primary key values. Set the **class** attribute of the generator element is set to **native** to let hibernate pick up either **identity, sequence** or **hilo** algorithm to create primary key depending upon the capabilities of the underlying database.

Which element of hbm.xml is used to map a Java class property to a column in the database table?

The **<property>** element is used to map a Java class property to a column in the database table. The **name** attribute of the element refers to the property in the class and the **column** attribute refers to the column in the database table. The **type** attribute holds the hibernate mapping type, this mapping types will convert from Java to SQL data type.

Which element of hbm.xml is used to map a java.util.Set property in hibernate?

This is mapped with a <set> element and initialized with java.util.HashSet.

Which element of hbm.xml is used to map a java.util.SortedSet property in hibernate?

This is mapped with a <set> element and initialized with java.util.TreeSet. The sort attribute can be set to either a comparator or natural ordering.

Which element of hbm.xml is used to map a java.util.List property in hibernate?

This is mapped with a <list> element and initialized with java.util.ArrayList.

Which element of hbm.xml is used to map a java.util.Collection property in hibernate?

This is mapped with a <bag> or <ibag> element and initialized with java.util.ArrayList.

Which element of hbm.xml is used to map a java.util.Map property in hibernate?

This is mapped with a <map> element and initialized with java.util.HashMap.

Which element of hbm.xml is used to map a java.util.SortedMap property in hibernate?

This is mapped with a <map> element and initialized with java.util.TreeMap. The sort attribute can be set to either a comparator or natural ordering.

What is lazy loading?

Lazy loading is a technique in which objects are loaded on demand basis.

Lazy loading in hibernate improves the performance. It loads the child objects on demand.

Since Hibernate 3**, lazy loading is by default**, enabled so that child objects are not loaded when parent is loaded.

13) What are the inheritance mapping strategies?

There are 3 ways of inheritance mapping in hibernate.

* Table per hierarchy
* Table per concrete class
* Table per subclass

How many types of association mapping are possible in hibernate?

1. One to One
2. One to Many
3. Many to One
4. Many to Many

What is the difference between first level cache and second level cache?

|  |  |  |
| --- | --- | --- |
| **No.** | **First Level Cache** | **Second Level Cache** |
| 1) | First Level Cache is **associated with Session**. | Second Level Cache is associated with **SessionFactory**. |
| 2) | It is **enabled** by default. | It is **not enabled** by default. |

create table EMPLOYEE (

id INT NOT NULL auto\_increment,

first\_name VARCHAR(20) default NULL,

last\_name VARCHAR(20) default NULL,

salary INT default NULL,

PRIMARY KEY (id)

);

Following is the mapping of Employee class with annotations to map objects with the defined EMPLOYEE table −

import javax.persistence.\*;

@Entity

@Table(name = "EMPLOYEE")

public class Employee {

@Id @GeneratedValue

@Column(name = "id")

private int id;

@Column(name = "first\_name")

private String firstName;

@Column(name = "last\_name")

private String lastName;

@Column(name = "salary")

private int salary;

public Employee() {}

//setters n getteers

Check “Tutorials point” for detailed example.

**One : Many**

 //in Orgnisation class

@OneToMany(cascade = CascadeType.***ALL***)

**private** Set<Employee> employee;

//in Employee class

@ManyToOne

@JoinColumn(name="id")

**private** Orgnisation orgnisation;

Ø The meaning of CascadeType.ALL is that the persistence will propagate (cascade) all EntityManager operations (PERSIST, REMOVE, REFRESH, MERGE, DETACH) to the relating entities.

Ø But it's bad practice to use CascadeType.ALL in @ManyToOne means It should propagate from Parent entities to Child ones always.

Ø Below are some other cascadetype

1.      **CascadeType.PERSIST** : means that save() or persist() operations cascade to related entities.

2.      **CascadeType.MERGE** : means that related entities are merged when the owning entity is merged.

3.      **CascadeType.REFRESH** : does the same thing for the refresh() operation.

4.      **CascadeType.REMOVE** : removes all related entities association with this setting when the owning entity is deleted.

5.      **CascadeType.DETACH** : detaches all related entities if a “manual detach” occurs.

6.      **CascadeType.ALL** : is shorthand for all of the above cascade operations.

++++++++++++++++++++++++++

@OneToMany(cascade = CascadeType.***ALL***, fetch=FetchType.***EAGER***)

**private** Set<Employee> employee;

@OneToMany(cascade = CascadeType.ALL, fetch=FetchType.LAZY)

**private** Set<Employee> employee;

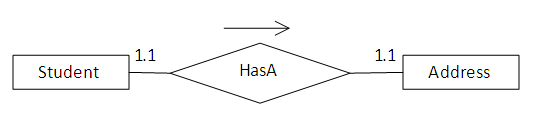
·         In Hibernate, FetchType.EAGER and FetchType.LAZY is used for collection.

·         If we set FetchType.LAZY, then until we fetch the collection, the collection will not be loaded. If we set FetchType.EAGER, then collection will be loaded at the same time when the parent entity is loaded.

·         FetchType is useful for the performance of system.

+++++++++++++++++++++++

**One : One**



## **hibernate methods**

**package** com.javatpoint.mypackage;

**import** javax.persistence.\*;

@Entity

@Table(name = "emp501")

**public** **class** Employee {

@Id

@Column(name = "id")

**private** **int** id;

@Column(name = "first\_name")

**private** String firstName;

@Column(name = "last\_name")

**private** String lastName;

**//setters n getters**

}

<!--hibernate.cfg.xml -->

<?xml version=*'1.0'* encoding=*'UTF-8'*?>

<!DOCTYPE hibernate-configuration PUBLIC

          "-//Hibernate/Hibernate Configuration DTD 3.0//EN"

          "http://hibernate.sourceforge.net/hibernate-configuration-3.0.dtd">

<hibernate-configuration>

    <session-factory>

        <property name=*"hbm2ddl.auto"*>update</property>

        <property name=*"dialect"*>org.hibernate.dialect.MySQL5Dialect</property>

        <property name=*"connection.url"*>jdbc:mysql://localhost:3306/test2</property>

        <property name=*"connection.username"*>root</property>

        <property name=*"connection.password"*>admin</property>

        <property name=*"connection.driver\_class"*>com.mysql.jdbc.Driver</property>

      <mapping class=*"com.javatpoint.mypackage.Employee"*/>

    </session-factory>

</hibernate-configuration>

+++++++++++++++++++

[contains() and isConnected methods in Hibernate](https://www.connect2java.com/tutorials/hibernate/contains-and-isconnected-methods-in-hibernate/)

**contains(Object object): Used to check whether the instance is associated with the session or not.**

**isConnected(): Used to check whether session is currently connected or not.**

**package** com.javatpoint.mypackage;

**import** org.hibernate.\*;

**import** org.hibernate.cfg.\*;

**public** **class** StoreData {

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

     Session session=**new** AnnotationConfiguration()

              .configure().buildSessionFactory().openSession();

           System.***out***.println("after open the session method ");

**if**(session.isConnected())

              System.***out***.println("Session is conncted");

**else**

              System.***out***.println("Session is not conncted");

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

         Transaction t=session.beginTransaction();

         Employee e1=**new** Employee();

         e1.setId(101);

         e1.setFirstName("sonoo7");

         e1.setLastName("jaiswal7");

         session.persist(e1);

**if**(session.contains(e1)){

            System.***out***.println("emp instance is associated with session");

        }

**else**{

            System.***out***.println("emp instance is NOT associated with session");

        }

         System.***out***.println();

        session.clear();

        System.***out***.println("After calling session.clear() method");

**if**(session.contains(e1)){

            System.***out***.println("emp instance is associated with session");

        }

**else**{

            System.***out***.println("emp instance is NOT associated with session");

        }

         t.commit();

         System.***out***.println();

         session.close();

         System.***out***.println("after session.close() method ");

**if**(session.isConnected())

            System.***out***.println("Session is conncted");

**else**

            System.***out***.println("Session is not conncted");

}

}

Output:

after open the session method

Session is conncted

emp instance is associated with session

After calling session.clear() method

emp instance is NOT associated with session

after session.close() method

Session is not conncted

+++++++++++++++++++++++++++

**/\* you can call transaction by 2 ways\*/**

**Transaction t=session.beginTransaction();**

**OR**

**Transaction t = session.getTransaction();**

**t.begin();**

++++++++++++++++++++++++++++++

[clear(), evict() and close() methods in Hibernate](https://www.connect2java.com/tutorials/hibernate/clear-evict-and-close-methods-in-hibernate/)

**clear():**Completely clear the session and is used to dissociate/disconnect all the objects from the session.

public void clear()

*Database before execution:*

mysql> select \* from emp501;

+----+------------+-----------+

| id | first\_name | last\_name |

+----+------------+-----------+

|  1 | sonoo      | jaiswal   |

|  2 | sonoo      | jaiswal   |

+----+------------+-----------+

2 rows in set (0.00 sec)

**public** **class** StoreData {

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

      Session session=**new** AnnotationConfiguration()

              .configure().buildSessionFactory().openSession();

         Transaction t=session.beginTransaction();

         Employee e1=**new** Employee();

         e1.setId(3);

         e1.setFirstName("sonoo");

         e1.setLastName("jaiswal");

         Employee e2=**new** Employee();

         e2.setId(4);

         e2.setFirstName("sonoo");

         e2.setLastName("jaiswal");

         session.persist(e1);

         session.persist(e2);

**session.clear();**

         t.commit();

         session.close();

         }

}

*Database after execution:*

mysql> select \* from emp501;

+----+------------+-----------+

| id | first\_name | last\_name |

+----+------------+-----------+

|  1 | sonoo      | jaiswal   |

|  2 | sonoo      | jaiswal   |

+----+------------+-----------+

2 rows in set (0.00 sec)

//suppose if u execute the above code once again, you will get : //Caused by: java.sql.BatchUpdateException: Duplicate entry ‘1’ and ‘2’ for key 'PRIMARY'

//to avoid above exception u can use below method

session.saveOrUpdate(e1);

session.saveOrUpdate(e2);

**evict():** Removes the object from the session. This method is used to dissociate/disconnect the specified object from the session

public void evict(Object object) throws HibernateException

*Database before execution:*

mysql> select \* from emp501;

+----+------------+-----------+

| id | first\_name | last\_name |

+----+------------+-----------+

|  1 | sonoo      | jaiswal   |

|  2 | sonoo      | jaiswal   |

+----+------------+-----------+

2 rows in set (0.00 sec)

**public** **class** StoreData {

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

          Session session=**new** AnnotationConfiguration()

              .configure().buildSessionFactory().openSession();

    Transaction t=session.beginTransaction();

**try**{

           Employee e1 = (Employee)session.get(Employee.**class**, 1);

            Employee e2 = (Employee)session.get(Employee.**class**, 2);

            //e1 and e2 are in persistent state.

            e1.setFirstName("sonoo Agarwal");

            e2.setFirstName("sonoo sham");

**session.evict(e1);**

            //e1 is in detached state and e2 is in persistent state.

               t.commit();

               session.close();

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

           }**catch**(HibernateException e){

               e.printStackTrace();

           }

}

}

*Database after execution:*

mysql> select \* from emp501;

+----+------------+-----------+

| id | first\_name | last\_name |

+----+------------+-----------+

|  1 | sonoo      | jaiswal   |

|  2 | sonoo sham | jaiswal   |

+----+------------+-----------+

2 rows in set (0.00 sec)

//if u add evict method after commit then there is no impact of evict()

**close():**

public Connection close() throws HibernateException

close the session by calling session.close() after transaction is completed.All the associated objects will be dissociated after calling session.close().It is not  strictly necessary to close the session but you must at least using disconnct it using session.disconnect()

++++++++++++++++++++++++

[saveOrUpdate() method in Hibernate Session](https://www.connect2java.com/tutorials/hibernate/saveorupdate-in-hibernate-session/)

public void saveOrUpdate(Object object) throws HibernateException

§  If the record is not present in the database, it will call save() method and inserts the record in the database.

§  If the record is present in the database, it will call update() method and updates the record in the database.

*Database before execution:*

mysql> select \* from emp501;

+----+------------+-----------+

| id | first\_name | last\_name |

+----+------------+-----------+

|  1 | sonoo      | jaiswal   |

+----+------------+-----------+

1 rows in set (0.00 sec)

**public** **class** StoreData {

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

      Session session=**new** AnnotationConfiguration()

              .configure().buildSessionFactory().openSession();

    Transaction t=session.beginTransaction();

**try**{

               Employee e1=**new** Employee();

               e1.setId(1);

               e1.setFirstName("sonoo modified");

               e1.setLastName("jaiswal modified");

               Employee e2=**new** Employee();

               e2.setId(2);

               e2.setFirstName("sonoo");

               e2.setLastName("jaiswal");

**session.saveOrUpdate(e1);**

               /\*

             \* Here employee object with the id 1 is already present

             \* in database. So it calls update() method and updates the

             \* record in database

             \*

            \*/

**session.saveOrUpdate(e2);**

               /\*

             \* Here employee object with the id 2 is not there

             \* in database, So it calls save() method and inserts the

             \* record in database

             \*

            \*/

               t.commit();

               session.close();

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

           }**catch**(HibernateException e){

               e.printStackTrace();

           }

}

}

Database after execution:

mysql> select \* from emp501;

+----+----------------+------------------+

| id | first\_name     | last\_name        |

+----+----------------+------------------+

|  1 | sonoo modified | jaiswal modified |

|  2 | sonoo          | jaiswal          |

+----+----------------+------------------+

2 rows in set (0.00 sec)

[flush() method in Hibernate](https://www.connect2java.com/tutorials/hibernate/flush-method-in-hibernate/) :

public void flush() throws HibernateException

When you call session.flush(), the statements are executed in database but it will not committed.

If you dont call session.flush() and if you call session.commit() , internally commit() method executes the statement and commits.

So commit()= flush+commit.

Database before execution:

mysql> select \* from emp501;

+----+----------------+------------------+

| id | first\_name     | last\_name        |

+----+----------------+------------------+

|  1 | sonoo          | jaiswal          |

+----+----------------+------------------+

1 rows in set (0.00 sec)

**public** **class** StoreData {

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

      Session session=new AnnotationConfiguration()

              .configure().buildSessionFactory().openSession();

    Transaction t=session.beginTransaction();

     try{

               Employee e1=new Employee();

               e1.setId(1);

               e1.setFirstName("sonoo mod");

               e1.setLastName("jaiswal");

               session.saveOrUpdate(e1);

               session.flush();

               t.commit();

               session.close();

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

           }catch(HibernateException e){

               e.printStackTrace();

           }

}

}

Database after execution:

mysql> select \* from emp501;

+----+----------------+------------------+

| id | first\_name     | last\_name        |

+----+----------------+------------------+

|  1 | sonoo mod      | jaiswal          |

+----+----------------+------------------+

1 rows in set (0.00 sec)

**//session.flush()**must be called before committing the transaction and closing the session

## Batch Processing with flush() method

Consider a requirement when you want to insert a large number of records in database using Hibernate. The code looks like as below

Session session = SessionFactory.openSession();

     Transaction tx = session.beginTransaction();

**for** ( **int** i=0; i<100000; i++ ) {

         Employee emp = **new** Employee(.....);

         session.save(emp);

     }

     tx.commit();

     session.close();

This code may throw **OutOfMemoryError**somewhere around 50,000th row. Because Hibernate caches all the newly inserted Employee objects in  the session level cache. We can solve this problem using **hibernate batch processing.**

We need to set **hibernate.jdbc.batch\_size**in **hibernate.cfg.xml**as below

<property name="hibernate.jdbc.batch\_size">40</property>

So, Hibernate executes every 40 rows as a batch.

And the above code has to be changed to :

Session session = SessionFactory.openSession();

     Transaction tx = session.beginTransaction();

**for** ( **int** i=0; i<100000; i++ ) {

         Employee emp = **new** Employee(.....);

         session.save(emp);

**if**( i % 40 == 0 ) { // Same as the JDBC batch size

             //flush a batch of inserts and release memory:

             session.flush();

             session.clear();

         }

     }

     tx.commit();

     session.close();

Batch processing helps to avoid **OutOfMemoryError**

**+++++++++++++++++++++++++++++++++++++++++++++**

[load() and get() methods of Hibernate Session](https://www.connect2java.com/tutorials/hibernate/load-and-get-methods-of-hibernate-session/)

**load():**

§  load() method throws **hibernate.ObjectNotFoundException**(which is Un-checked exception) if object not found in cache as well as in database.

§  load() method is **lazy loading i.e.**when you call session.load(Class,identifier) method, it will return **Proxy object**but not **entity**

§  Use load() method when you are sure that the object exists in database.

§  load() method is equivalent to **getReference()** method of JPA.

§  Since load() method returns proxy object, so it is not fully available in any future detached state. Detached state means Object is not associated with Session, but present in Database. So if you are not working with detached objects **load() or getReference()**methods can be use to have **better performance**.

**get():**

§  get() method returns **NULL**, if object is not found in cache as well as in database.

§  get() method is **early loading** **e.** when you call session.get(Class,identifier) method, it will hit the database immediately ,loads the entity object and returns the entity object.

§  Use get() method if you are not sure that the object exists in database.

§  It is equivalent to EntityManager.find() method of JPA.

§  Since get() returns a fully initialized object, so it is fully available in any future detached state.

§

**How to call load() method:**

employee = (Employee)session.load(Employee.**class**, 2);

        /\*

         \* Put break point here and see console. You will not be

         \* seeing any select statement and employee object is not

         \* initialized (see in variables window).

        \* At this time employee is just a proxy object.

         \*

         \*/

        System.out.println(employee.getEmpName());

        /\*

         \* Now Observe the console at this line,you can find the

         \* select statement and employee object is initialized

         \* (see in variables window.)

         \*

         \*/

        session.getTransaction().commit();

**How to call get() method:**

employee = (Employee)session.get(Employee.**class**, 2);

     /\*

      \* Put break point here and see console. You will be

      \* seeing select statement and employee object is

      \* initialized (see in variables window).

      \* At this time employee object is initialized with

      \* database data

      \*

      \*/

     System.***out***.println(employee.getEmpName());

     session.getTransaction().commit();

++++++++++++++++++++++++++++++++++++++

**What's the advantage/Differance of persist() vs save() in Hibernate?**

save:

1.      will return the id/identifier when the object is saved to the database.

2.      will also save when the object is tried to do the same by opening a new session after it is detached.

Persist:

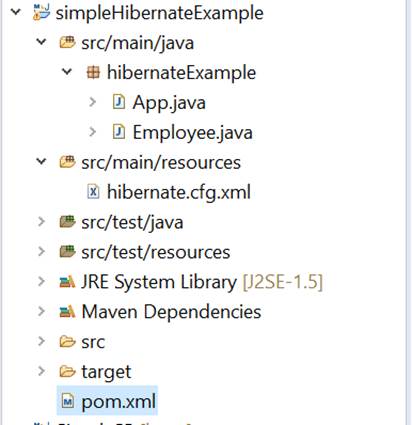
3.      will return void when the object is saved to the database.

4.      will throw PersistentObjectException when tried to save the detached object through a new session.

## **simple Hibernate annotation Example**:

Link :

<https://examples.javacodegeeks.com/enterprise-java/hibernate/hibernate-annotations-example/>



## **HQL and Criteria**

**HQL**:

It’s same as SQL, but it doesn’t depend on table. Here it uses class name instead of table name.

String hql = “select E.firstName from Employee E”

Query q= session.createQuery(hql);

List results = q.list();

String hql = “select E.firstName from Employee E where E.id=10”

Query q= session.createQuery(hql);

List results = q.list();

String hql = “select E.firstName from Employee E order by E.sal desc”

Query q= session.createQuery(hql);

List results = q.list();

String hql = “select E.firstName from Employee E group by E.name ”

Query q= session.createQuery(hql);

List results = q.list();

**Pagination:**

String hql = “from Employee E”

Query q= session.createQuery(hql);

q.setFirstResult(1);

q.setMaxResult(10);

List results = q.list();

**Criteria :**

Criteria cr = session.createCriteria(Employee.class);

List results = cr.list();

Criteria cr = session.createCriteria(Employee.class);

cr.add(Restrictions.eq(“salary”,2000));

List results = cr.list();

cr.add(Restrictions.gt(“salary”,2000));

cr.add(Restrictions.lt(“salary”,2000));

cr.add(Restrictions.like(“sname”, “zara%”));

cr.add(Restrictions.between(“salary”,1000,2000));

Criteria cr = session.createCriteria(Employee.class);

Criterion sal = Restrictions.eq(“salary”,2000);

Criterion name = Restrictions.like(“sname”, “zara%”);

LogicalExpression le = Restriction.or(sal,name);

cr.add(le);

List results = cr.list();

## differences between JPA and **Hibernate ?**

|  |  |
| --- | --- |
| **JPA** | **Hibernate** |
| Java Persistence API (JPA) defines the management of relational data in the Java applications. | Hibernate is an Object-Relational Mapping (ORM) tool which is used to save the state of Java object into the database. |
| It is just a specification. Various ORM tools implement it for data persistence. | It is one of the most frequently used JPA implementation. |
| It is defined in **javax.persistence** package. | It is defined in **org.hibernate** package. |
| The **EntityManagerFactory** interface is used to interact with the entity manager factory for the persistence unit. Thus, it provides an entity manager. | It uses **SessionFactory** interface to create Session instances. |
| It uses **EntityManager** interface to create, read, and delete operations for instances of mapped entity classes. This interface interacts with the persistence context. | It uses **Session** interface to create, read, and delete operations for instances of mapped entity classes. It behaves as a runtime interface between a Java application and Hibernate. |
| It uses **Java Persistence Query Language** (JPQL) as an object-oriented query language to perform database operations. | It uses **Hibernate Query Language** (HQL) as an object-oriented query language to perform database operations. |

# Microservices:

<https://codingnconcepts.com/top-microservices-interview-questions/>

## Difference between monolithic and microservices architecture ?

|  |  |  |
| --- | --- | --- |
| 1 | Monolithic architecture is built as **one code-base** | Microservices architecture is built as **small independent module** based on business functionality |
| 2 | It is not easy to **scale** based on demand | It is easy to scale based on demand. |
| 3 | It has shared **database** | Each project and module has their own database |
| 4 | It extremely difficult to change technology or language or **framework** because everything is tightly coupled and depend on each other | Easy to change technology or framework because every module and project is independent |
| 5 | Any **process** failure will bring whole system down | 6 Any process failure will not bring whole system down |

**Def:** microservices are method of developing software applications, which are **collection of small autonomous services** and which can be independently deployable.

## **Advantages of microservice architecture?**

OR**Why Would You Opt For Microservices Architecture?**

1. We can use any technologies/framework to build modules.
2. Deployment is separate and fast and independent.
3. Eassy to maintain, because smaller code base
4. Any process failuare will not bring whole system down.
5. Team size is less

**Dis adv :**

* Difficult end to end testing because its distributed system
* Heavy architecture set up – The system is distributed, the architecture is heavily involved.

**3) What is Monolithic Architecture?**  //ref above diff only for ans

**4. Name three commonly used tools for Microservices**

Wiremock, 2.) Docker and 3.) Hysrix are important Microservices tool.

**5) What is Spring Cloud?**

**6) Discuss uses of reports and dashboards in the environment of Microservices?**

Reports and dashboards help in **monitoring and upkeep** of Microservices.

**13) Explain three types of Tests for Microservices?**

* bottom level test :we can perform a general test like **performance and unit tests**. These kinds of tests are entirely automated.
* middle level: we can perform exploratory tests like the **stress tests and** **usability tests.**
* top level : we can conduct **acceptance** tests which are mostly fewer in numbers. It also helps stakeholders to know about different software features.

**14) What are Client certificates?**

Client certificates is a **digital certificate** used to make **authenticated requests to a remote server**. It is termed as a client certificate.

**16) What is the meaning of OAuth?**

OAuth means **open authorization protocol**. This protocol allows you to access the client applications on HTTP for third-party providers GitHub, Facebook, etc. It helps you to **share resources stored on one site with another site without the need for their credentials.**

**24) How independent micro-services communicate with each other?**

Microservices often communicate with each other using RESTful APIs over HTTP. The communication can be broadly divided into two categories:-

**RestTemplate, WebClient, FeignClient** can be used for synchronous communication between microservices

**ActiveMQ, RabbitMQ, Kafka** can be used for asynchronous communication across microservices.

Diff between synchronous and asynchronous?

Q6. How do you troubleshoot an issue using logs in microservices based application?

* We should have a centralized logging system where each microservice push their logs to **Splunk** or ELK (Elastic Logstash **Kibana**) and we can use their built in dashboards to look at the logs for debugging.
* We can generate a **requestId** for each external request, which is passed to all the microservices which are involved in handling the request. Include this **requestId** in all log messages pushed to splunk or ELK. We can troubleshoot any request end to end using this requestId if something goes wrong.

How do you manage authentication and authorization in microservices based application ?

https://www.youtube.com/watch?v=EW91te0s36Q

**Session based authentication** works well with **stateful** monolith applications but

**token based authentication** and authorization is recommended for microservices based application to maintain the **statelessness**. A typical flow of token based authentication is as follows:-

we have **AuthenticationManager** and **JwtTokenUtil** class provided by the spring security in order to authenticate and generate token.

1. User sends a login request with username and password.
2. If you are using Api gateway then it is responsible for generating the token and hence authentication. It achieves this by communicating with authorization and users service.
3. User receives a token on successful login which is typically stored in browser cookies. Token holds the user’s information in the encrypted format.
4. When user make any request of resources, this token is sent in Authorization header of each request.
5. Microservices decrypts the token and evaluate user information to authorize for resource access and send the response accordingly.

**JWT**

Hands on Ref: https://www.youtube.com/watch?v=lA18U8dGKF8

JWT (Json Web Token) is widely used token based authentication mechanism. JWT consist of three parts:

* **header** contains type, fixed value JWT and the hashing algorithm used by JWT

{

* "typ": "JWT",
* "alg": "HS256"
* }
* **payload** typically contains user authorization related information such as id, name, roles, permissions etc. It also contains the expiry period of token.

{

* "id": 12345,
* "name": "admin\_user",
* "email": "admin\_user@organization.com",
* "roles": ["admin"],
* "permissions": ["can\_access\_resource\_1", "can\_access\_resource\_2"]
* }
* **signature** is required to verify the authenticity of token. It consists of the encoded header, the payload and the secret key.
* HMACSHA256(
* base64UrlEncode(header) + "." +
* base64UrlEncode(payload),
* secret
* )

All microservices can verify the token based on the signature so there are no further calls to the authorization server after login.

What is circuit breaker pattern in microservices?

Services sometimes call other services to handle requests. There is always a possibility that the other service is **unavailable** or taking **longer time to respond**.

What to do when circuit breaks?  
● throw an error ***or***  
● return a fallback “default” response ***or***  
● save previous responses from cache

 How do you handle distributed transaction across microservices?

18) Why are Container used in Microservices? what is containerize?

Docker offers container environment, which can be used to host our application. the software application and the dependency which it support are tightly package together.

what are the best practice with microservice?

we can dockerize it. So that we will get all dependency and software in single image.

To manage a microservice based application, containers are the easiest alternative. It helps the user to individually deploy and develop. You can also use Docker to encapsulate the microservice in the image of a container. Without any additional dependencies or effort, microservices can use these elements.

## **challenges with microservices:**

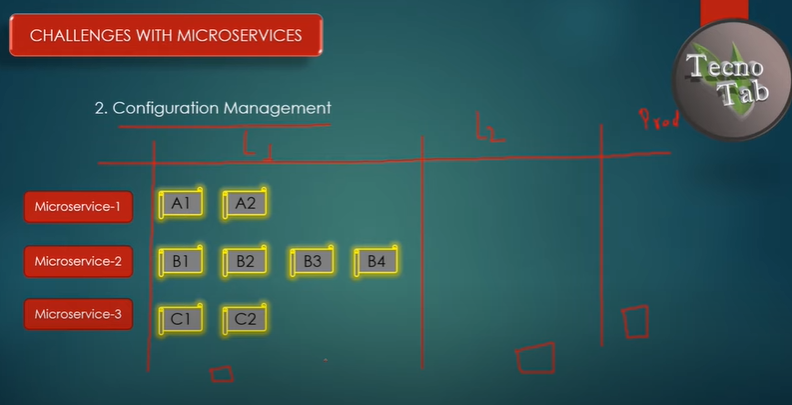
1. **Managing Microservices**

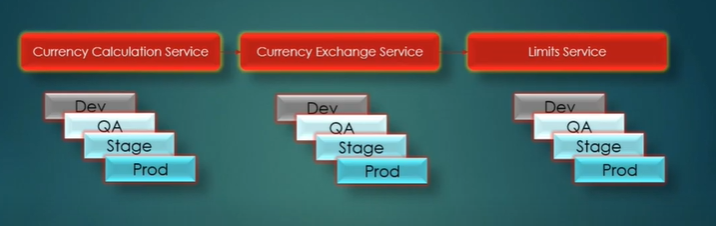
means, initially we start developing with targeting there are 16 modules(microservice) we are going to develop, but on going few more modules will add. So it is a evolving process which we'll figure out once we start developing.



1. **configuration management**

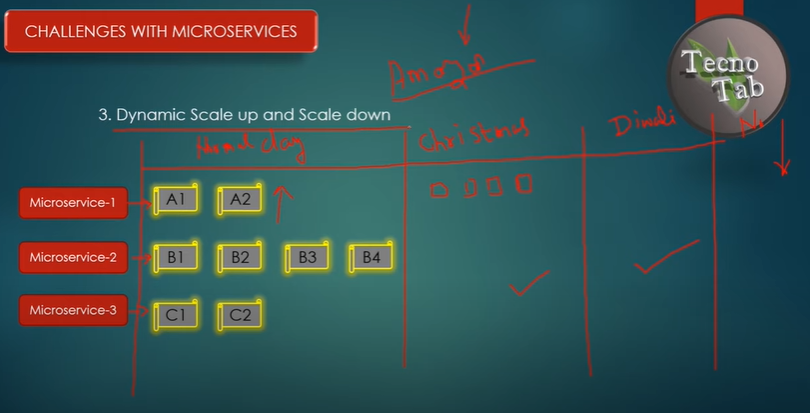
Suppose if we have 3 microservices and each microservice have its differant instances running. and our projects have differant environments like dev, testing and production. and each environment has its its seperate url, database configuration and other external configuration we have to maintain.





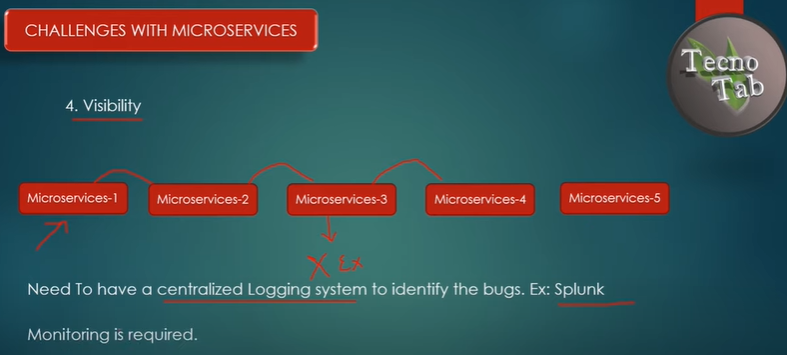
1. **Dynamic scaleup and scaledown**

Suppose if we have 3 microservices and each microservice have its differant instances running. But load on these instances will be depends on the occassions, For example if we have some e\_commerce application, during festival season the the number of users will be increases and load will be increases on application in that case we have to scaleup the application and in normal days we have to scaledown.



1. **Monitoring (Visibility)**

Suppose if we have 5 microservices communicating each other and in case exception occurred in one of microservice, how to identify where exception occurs? we cant go each microservice and check. So we Need to have centralized logging system to identify the bugs. Eg: Splunk or kibana



1. **Fault tolerance**

## **Feign client**

**==========**

Feign client is really convenient tool to use. And **Rest-Template is going to be deprecated** and will be replaced by WebClient, and Feign Client internally uses Rest-Template.

using "feign" instead of using RestTemplate, it reduces lines of code.

* 1. add below dependency in "currency calculation" application

<dependency>

<groupId>org.springframework.cloud</groupId>

<artifactId>spring-cloud-starter-openfeign</artifactId>

</dependency>

* 1. in "currency calculation" application, create a proxy interface, add @FeignClient annotation and define currency-exchange-application name in it.

@FeignClient(name = "currency-exchange-service")

* 1. and call the proxy interface method in controller.

## **Ribbon Load Balancer**

**====================**

"currency calculation" will call different instance of "currency exchange service" based on the ribbon load balancer.

//add dependency for ribbon load balancer in "currency calculation" application.

<dependency>

<groupId>org.springframework.cloud</groupId>

<artifactId>spring-cloud-starter-netflix-ribbon</artifactId>

</dependency>

in "currency calculation" application, in proxy interface, add @RibbonClient annotaion and define currency-exchange-application name in it.

@RibbonClient(name = "currency-exchange-service")

and define "currency exchange service" instance url's in application.properties of "currency calculation" application

## **eureka naming server**

**why eureka naming server?**

Eureka Server is an application that holds the information about all client-service applications. Every Micro service will register into the Eureka server and Eureka server knows all the client applications running on each port and IP address

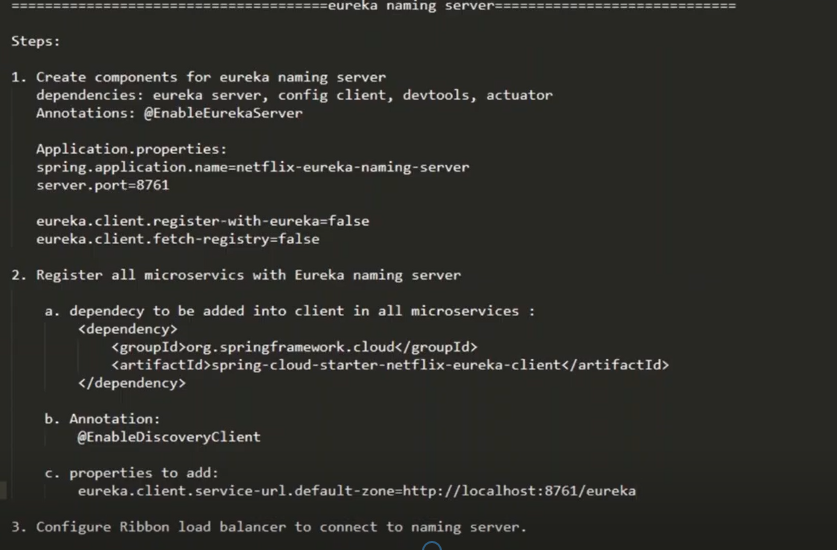
* 1. **service registration** : whenever new microservice is created it needs to register with "eureka naming server"
  2. **service discovery** : whenever any microservice wants to communicate with other microservice, 1st flow will go to eureka naming server then it will discover that how many instances service are available based on that it distribute the load. This process is called service discovery.

**What is Netflix Eureka server?**

What Is Netflix Eureka? This is a tool provided by Netflix to provide a solution to the above problem. It consists of the Eureka Server and Eureka clients. Eureka Server is in itself a microservice to which all other microservices registers. Eureka Clients are the independent microservices.

**How Eureka Server works internally?**

When a client registers with Eureka, it provides meta-data about itself ,such as host and port, health indicator URL, home page, etc. Eureka receives heartbeat messages from each instance belonging to a service. If the heartbeat fails over a configurable timetable, the instance is normally removed from the registry



## Configuring ZUUL API Gateway

**========================**

Zuul Server is an API Gateway application provided by Netflix. It handles all the requests and performs the dynamic routing of microservice applications. It works as a front door for all the requests. It is also known as Edge Server.

Zuul is built to enable dynamic routing, monitoring, resiliency, and security. It can also route the requests to multiple Amazon Auto Scaling Groups.

The volume and variety of Netflix API traffic sometimes result in production issues that arises quickly and without warning. So we need a system that allows us to rapidly change behavior in order to react to these situations.

Zuul provides a range of different types of filters that allows us to quickly and nimbly apply functionality to our edge service. The filters perform the following functions:

* 1. **Authentication and Security**: It provides authentication requirements for each resource.
  2. **Insights and Monitoring**: It tracks meaningful data and statistics that give us an accurate view of production.
  3. **Dynamic Routing**: It dynamically routes the requests to different backed clusters as needed.
  4. **Stress Testing**: It increases the traffic to a cluster in order to test performance.
  5. **Load Shedding**: It allocates capacity for each type of request and drops a request that goes over the limit.
  6. **Static Response Handling**: It builds some responses directly at the edge instead of forwarding them to an internal cluster.
  7. **Multi-region Resiliency**: It routes requests across AWS regions in order to diversify our ELB usage.

1. Create new microservice with name something NetflixZuulApiGatewayServerApplication
2. Add @EnableZuulProxy to main class
3. Add application name and port number in property file.
4. Create filter component :
   1. Creating class called ZuulLoggingFilter extends ZuulFilter
   2. It has some set of methods needs to implement
      1. boolean shouldFilter()
      2. Object Run()
      3. String filterType()
      4. Int filterOrder()
5. How to hit other microservice through zuul API Gateway?
   1. [http://localhost:8765/<application-name>/<uri](http://localhost:8765/%3capplication-name%3e/%3curi)>

**eg**:http://localhost:8765/currency-exchange-service/from/USD/to/INR

**FYI:**

**ZuulFilter** class has four abstract methods that are listed below:

* **shouldFilter():**The shouldFilter() method checks the request and decides whether filter to be executed or not.
* **run():**The run() method invokes, if both **!isFilterDisabled()** and **shouldFilter()**methods returns **true**.
* **filterType():**The filterType() method classify a filter by type. There are four types of standard filters in Zuul: **pre** for **pre-routing filtering**, **route** for **routing to an origin**, **post** for **post-routing filters**, and **error** for **error handling**. Zuul also supports a **static** type for **static responses.** Any filter type can be created or added and run by calling the method **runFilters(type).**
* **filterOrder()**: The filter order must be defined for a filter. Filters may have the same filter order if the precedence is not important for the filters. The filter order does not need to be sequential.

what is your thought process for building a new microservice architecture ?

The first thing is, let me take the

1. overall requirement gathering and analysis.
2. what and relationship mapping between class and interfaces? and designing the database.
3. what and all third party resource required? For example if its e-commerce service, we might need Google Map API and we may need SMS service. SO will think about how to implement third party resources.
4. Analysing the requirement and divide it into small modules.
5. Then come up with High Level Design Document with details like :
   * + Architecture design
     + Finalised Modules
     + Planned API in each modules and its inputs and expected output.
     + Hardware requirments : Like Environment details, server details
     + Expected Test cases

what is your thought process for migrating monolithic architecture to microservice architecture ?

1. The first thing is need to understand overall functionality and call flow of existing architecture

2. Need to identify the different modules in it.

eg: If you have a e-commerce application, there will be differant modules like Items, Customers, Payment, Stock, Address

3. Then based on identified seperate module we have to create seperate microservice for it.

In a ground level, It's a long term process. But I'm giving a just heads up now.

How do you restrict the accessing of API?

How to handle fault tolerance in microservice?

Hystrix

How to do distributed tracking in microservice?

Spring cloud sleuth

Eureka

what is entry point for microservice?

Zuul gateway

what is client side load balancing?

Ribbon

# Design patterns :

<https://www.youtube.com/watch?v=JFvSCobD7JU> : youtube link

<https://www.javatpoint.com/abstract-factory-pattern>

## **Factory method pattern:**

**define an interface or abstract class for creating an object but let the subclasses decide which class to instantiate.** In other words, subclasses are responsible to create the instance of the class.

Adv:

* Factory Method Pattern allows the sub-classes to choose the type of objects to create.
* It promotes the **loose-coupling**

Eg: Bank and rateOfIntrest()

## **Abstract factory pattern**

## **Singleton pattern**

<https://www.journaldev.com/1377/java-singleton-design-pattern-best-practices-examples>

### **What is singleton design pattern? How we can create it?**

* Singleton pattern restricts the instantiation of a class and ensures that only one instance of the class exists in the JVM.
* The singleton class must provide a global access point to get the instance of the class.

1. **Private constructor** to restrict instantiation of the class from other classes.
2. **Private** **static variable** of the same class that is the only instance of the class.
3. **Public** **static method** that returns the instance of the class, this is the global access point for outer world to get the instance of the singleton class.

**Advantage of Singleton design pattern**

* **Saves memory** because object is not created at each request. Only single instance is reused again and again.

**Usage of Singleton design pattern**

* Singleton pattern is mostly used in **multi-threaded** and database applications. It is used in logging, caching, thread pools, configuration settings etc.

### **what are the ways to declare singleton design pattern?**

* 1. Eager initialization
  2. Static block initialization
  3. Lazy Initialization
  4. Thread Safe Singleton
  5. singleton with double locking check

## **Prototype design pattern:**

Prototype Pattern says that **cloning of an existing object instead of creating new one and can also be customized as per the requirement**.

Adv/Usage:

* When the classes are instantiated at runtime.
* When the cost of creating an object is expensive or complicated.
* When you want to keep the number of classes in an application minimum.
* When the client application needs to be unaware of object creation and representation.

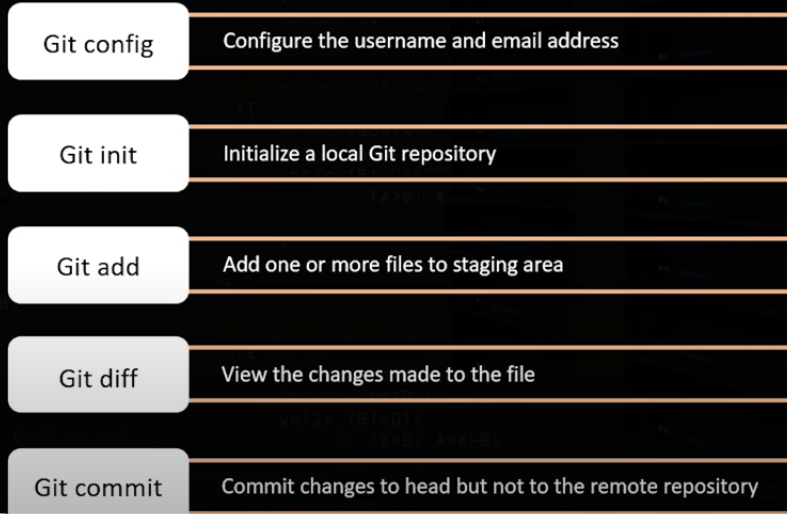
Eg:

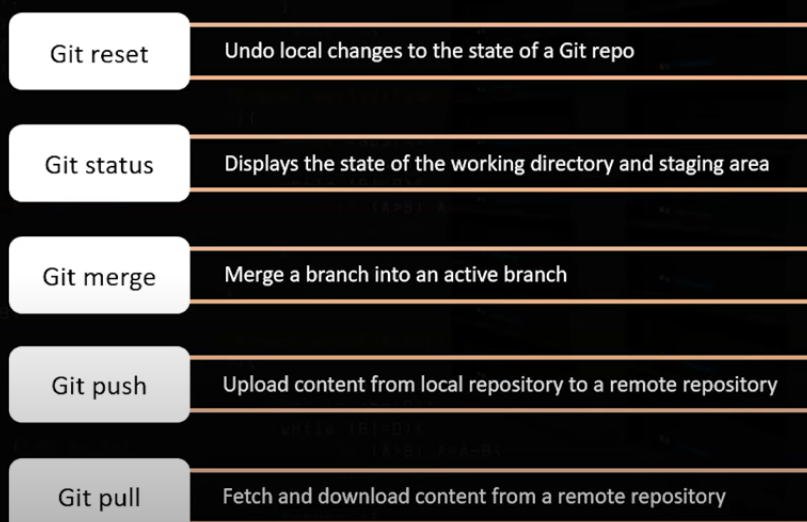
# Projects

how to configure git with eclipse <https://www.youtube.com/watch?v=LPT7v69guVY&t=627s>

some of your git projects: <https://github.com/sudheer436/SafariApp>

## **Git basic commands**





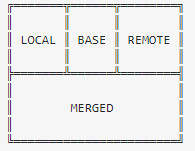
**how do u resolve merge conflicts in git?**

git pull origin master

OR

git mergetool

So it open up 3 windows, then we can make changes manually and save file.



LOCAL – this is file from the current branch

BASE – common ancestor, how file looked before both changes

REMOTE – file you are merging into your branch

MERGED – merge result, this is what gets saved in the repo

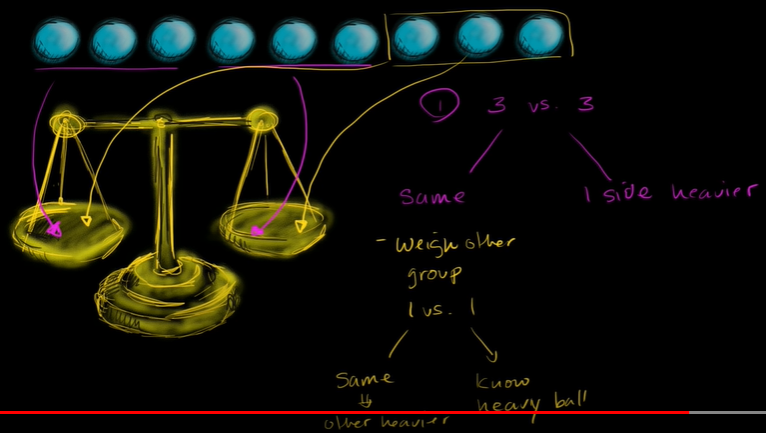
# Logical Questions:

## Weighing balls

* 1. You have **9 balls** that look identical, but 1 is heavier than the rest. What is the minimum number of weighings on a scale to determine the heavy ball?

Approach: **3 on one left side and 3 on right side of weighing machine**

**and 3 are in basket**



**Question:** You are given **8 identical looking balls**. One of them is heavier than the rest of the 7 (all the others weigh exactly the same). You a provided with a simple mechanical balance and you are restricted to only 2 uses. Find the heavier ball.

**Answer :**

**Approach : 3 on one left side and 3 on right side of weighing machine**

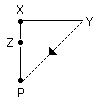
**and 2 are in basket**

## Direction related questions:

=======================

Y is in the East of X which is in the North of Z. If P is in the South of Z, then in which direction of Y, is P?

(which is means 'X' is north of 'Z')



P is in South-West of Y.

## water lilies

there are water lillies in lake, and each day it will doubles, after 20 days lake is full of lillies. on what day the lake will covered with half of lillies.

//Ans : 19 (cz every day it doubles from previous day)

5 machines produces 5 plates in 5 hours . how much do 3 machines need 3 plates?

//Ans: 5 hours only

## Three light bulbs

In this logic question, you are standing in a room with three light switches. The switches all correspond to three different light bulbs in an adjacent room that you cannot see into. With all the light switches starting in the off position, how can you find out which switch connects to which light bulb?

**Answer:** "I would turn on the first switch and let it stay on for a few minutes. Then, I'd turn the first switch off and quickly turn on the second switch. Then, checking the room, I'd see the second bulb turned on and feel the other two bulbs to see which one is warmer. The warmer bulb is the one I just turned off, so that belongs to the first switch, while the bulb that's on belongs to the second switch. The third switch would belong to the bulb that is off and coolest to the touch."

## Manhole covers

This logical question can be presented as a riddle, and there are several variations on the format of this puzzle. Here is the question:

Why are manhole covers round?

**Example answer:** "Manhole covers are round so they never need to be rotated to place the cover back on. Also, circles are the only geometric shape that won't fall in on itself in a similarly-shaped hole, so it should never be able to fall in. For instance, a square manhole cover might fall in a square hole if it were inserted diagonally."

## The farmer, the fox and the chicken

In this riddle, you are presented with the farmer's problem of transporting his animals and crops across a river. Here is the question:

A farmer needs to cross the river with his fox, his chicken and a bag of corn. However, the boat can only fit the farmer and one other thing at a time. The problem is, the fox and the chicken are both hungry, so if he leaves the fox and chicken together, the fox might eat the chicken. If he leaves the chicken and corn together, the chicken might eat the corn.

So how can the farmer get everyone across the river safely?

**Example answer:** "First, the farmer should take the chicken across. Then, he can go back for the fox. When he arrives at the other side to drop off the fox, he can take the chicken back with him to get the corn. The farmer drops off the chicken and transports the corn to the other side of the river. Finally, the farmer goes back for the chicken and continues to the other side of the river with nothing eaten."

## Four names

This question is a fairly simple riddle and it really only depends on your active listening skills. Here is the question:

Jane's mother has four children. The first child's name is Spring, the second child's name is Summer and the third child's name is Autumn. What is the fourth child's name?

**Example answer:** "The fourth child's name is **Jane** because she was named at the very beginning."

## Filling the jugs

This question can require some mathematical computation:

**3:5 | 4**

If you have a three-gallon jug and a five-gallon jug, how can you measure out exactly four gallons of water?

1. 3 of 3 to 0 of 5 = becomes 3 of 5
2. 3 of 3 to 5 = 5 of 5 but in 3C remains 1 of 3
3. Pour 5C full
4. Put remain 1 of 3 to 0 of 5 : becomes 1 of 5
5. Put 3 of 3 to 1 of 5: becomes 4 of 5 : means it has 4 litter now

Q) There are 2 jugs with 4 litres and 5 litres of water respectively. The objective is to pour exactly 7 litres of water **in a bucket**. How can it be accomplished?

**4:5 | 7**

Ans: 1. From 4 litter jug put 4 litter of water to 5 litter jug

* 1. Again from 4 litter jug to 5 litter jug: 5 litter jug has full 4 litter jug has 3 litter remain
  2. From 4 litter capacity jug, Put those remaining 3 litter to bucket
  3. Then again fill 4 litter jug fully and pour on bucket. Its become 7 litter totally.

## Climbing a wall

This numeracy question focuses on number patterns and how you process information to calculate an outcome. Here is the question:

Imagine you are standing at the bottom of a 30-foot wall. You start climbing, and each hour you climb three feet but slide down two feet. Assuming you keep this pattern of climbing, how long will it take you to climb to the top of the wall?

**// 28 hours**

## **Equal cake cutting :**

A birthday cake has to be equally divided into **8 equal pieces in exactly 3 cuts**. Determine the way to make this division possible



## **Focus on Questions**

*Tracy’s mother has four children. One child is named April. The second one is May. The third is June. What‘s the fourth one’s name?*

*Ans: tracy*

A doctor’s son’s father was not a doctor. How is this possible?

Ans:The doctor is the mother.

A woman and daughter walked into a restaurant. A man walked past and the women both said “Hello, Father”. How is this possible?

Ans: The man is a priest, or his name is Father.

A horse jumps over a castle, then lands on a man. The man disappears. What’s happening?

Ans: playing chess

A man was born in 1945, but he’s only 30 years old now. How is this possible?

Ans: 1945 was the number of the hospital room.

An explorer found a silver coin marked 7 BC. He was told it was a forgery. Why?

Ans: You can’t have “Before-Christ” in the Before-Christ years, because nobody knew who this Christ was.

## How strong EGG is

**Question:** You have two identical eggs. Standing in front of a 100 floor building, you wonder what is the maximum number of floors from which the egg can be dropped without breaking it. What is the minimum number of tries needed to find out the solution?

Ans : 51

Explain : Let’s start at the second floor. If the egg breaks, then we can use the second egg to go back to the first floor and try again. If it does not break, then we can go ahead and try on the 4th floor (in multiples of 2).

# Others

## **unit testing**

==============

<https://www.youtube.com/watch?v=kXhYu939_5s>

**junit and mockito**

To perform unit testing, we need to create test cases. **The unit test case is a code which ensures that the program logic works as expected.**

**Mockito is mainly used to mock the service data or repository data in order to avoid hit the db for testing,**

@Test annotation specifies that method is the test method.

@Test(timeout=1000) annotation specifies that method will be failed if it takes longer than 1000 milliseconds (1 second).

@BeforeClass annotation specifies that method will be invoked only once, before starting all the tests.

@Before annotation specifies that method will be invoked before each test.

@After annotation specifies that method will be invoked after each test.

@AfterClass annotation specifies that method will be invoked only once, after finishing all the tests.

**The common methods of Assert class are as follows:**

void assertEquals(boolean expected,boolean actual): checks that two primitives/objects are equal. It is overloaded.

void assertTrue(boolean condition): checks that a condition is true.

void assertFalse(boolean condition): checks that a condition is false.

void assertNull(Object obj): checks that object is null.

void assertNotNull(Object obj): checks that object is not null.

**Below are annotations we commonly used with junit testing:**

@RestController

public class UserController {

@Autowired

private UserService service;

@PostMapping(value = "/save")

public User saveUser(@RequestBody User user) {

return service.addUser(user);

}

@GetMapping("/getUsers")

public List<User> findAllUsers() {

return service.getUsers();

}

@GetMapping("/getUserByAddress/{address}")

public List<User> findUserByAddress(@PathVariable String address) {

return service.getUserbyAddress(address);

}

@DeleteMapping(value="/remove")

public User removeUser(@RequestBody User user) {

service.deleteUser(user);

return user;

}

}

@RunWith(SpringRunner.class)

@SpringBootTest

public class SpringBootMockitoApplicationTests {

@Autowired

private UserService service;

@MockBean

private UserRepository repository;

@Test

public void getUsersTest() {

when(repository.findAll()).thenReturn(Stream

.of(new User(376, "Danile", 31, "USA"), new User(958, "Huy", 35, "UK")).collect(Collectors.toList()));

assertEquals(2, service.getUsers().size());

}

@Test

public void getUserbyAddressTest() {

String address = "Bangalore";

when(repository.findByAddress(address))

.thenReturn(Stream.of(new User(376, "Danile", 31, "USA")).collect(Collectors.toList()));

assertEquals(1, service.getUserbyAddress(address).size());

}

@Test

public void saveUserTest() {

User user = new User(999, "Pranya", 33, "Pune");

when(repository.save(user)).thenReturn(user);

assertEquals(user, service.addUser(user));

}

@Test

public void deleteUserTest() {

User user = new User(999, "Pranya", 33, "Pune");

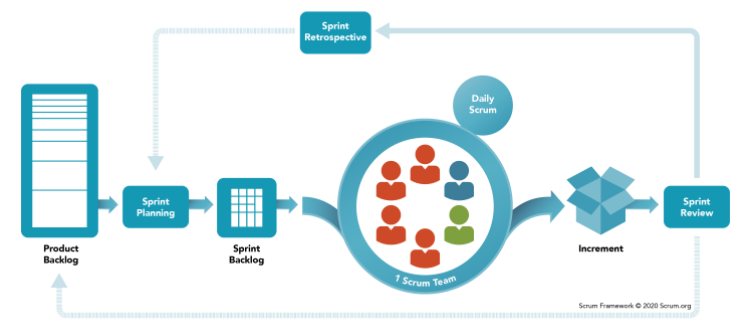
service.deleteUser(user);

verify(repository, times(1)).delete(user);

}

}

## Agile methodology:



**What is scrum/Agile methodology?**  
Scrum is a framework used to manage product development.

Scrum has three roles: **product owner, scrum master and the development team** members.

The **Product Owner** (PO) is a member of the Agile Team responsible for defining Stories and prioritizing the Team Backlog.

**Responsibilities of scrum master:**

* ensuring the team uses agile values and principles and follows the processes and practices
* Hosts Daily Stand-up Meetings, Sprint Planning Meetings, Retrospectives and Reviews. ...
* Helps the Product Owner with the Product Backlog. ...
* Helps team to Clearing obstacles.
* Protects the Team from outside interference. ...

**What is sprint and sprint palnning?**

Sprints are time-boxed periods of one week or 2 week or max 1 month, wherein a development team works to complete specific tasks, milestones, or deliverables.

A new sprint starts immediately after the current sprint ends.

The purpose of sprint planning is to define what can be delivered in the sprint and how that work will be achieved.

**What is product backlog and sprint backlog?**

The Product Backlog is specific to the entire goal of the product. The Sprint Backlog is specific only to the Sprint goal in a particular Sprint.

**What is scrum board?**

A Scrum Board is a tool that helps Teams make Sprint Backlog items visible. And helps us to track the status of each task (Story/todo/in progress/ verify/ done)

**What is retrospective in agile?**

The sprint retrospective is focused on how the team worked together during the sprint.

What went well, what we can improve etc

## jenkins :

(As a developer just know about CI CD pipeline brief)

**Best DevOps tools for Continuous Integration**

* Jenkins.
* TeamCity.
* Bamboo.
* Buddy.

What is CI CD pipeline? Explain?

Which tool you have used for CI CD pipeline in your project?

What is version control?

**Goals:**

1. understand jenkins: https://www.youtube.com/watch?v=jkJgS3zDv9g&list=PLVz2XdJiJQxwS0BZUHX34ocLTJtRGSQzN&index=5
2. create jenkins account:
3. spring boot + git + jenkins + docker image :
4. AWS DevOps | Build a CI/CD in AWS using CodePipeline :

https://www.youtube.com/watch?v=aHuCeLn72Lg

Ref:

<https://www.youtube.com/watch?v=k2aNsQKwyOo> //def and explain

<https://www.youtube.com/watch?v=m0a2CzgLNsc> // hands on with jenkins

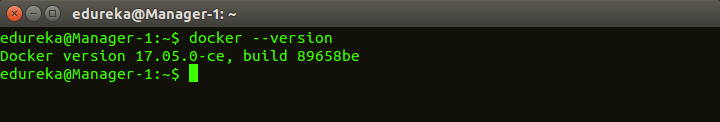
## Docker

### Basic docker commands:

* [**docker –version**](https://www.edureka.co/blog/docker-commands/#version)
* [**docker pull**](https://www.edureka.co/blog/docker-commands/#pull)
* [**docker run**](https://www.edureka.co/blog/docker-commands/#run)
* [**docker ps**](https://www.edureka.co/blog/docker-commands/#ps)
* [**docker ps -a**](https://www.edureka.co/blog/docker-commands/#psa)
* [**docker exec**](https://www.edureka.co/blog/docker-commands/#exec)
* [**docker stop**](https://www.edureka.co/blog/docker-commands/#stop)
* [**docker kill**](https://www.edureka.co/blog/docker-commands/#kill)
* [**docker commit**](https://www.edureka.co/blog/docker-commands/#commit)
* [**docker login**](https://www.edureka.co/blog/docker-commands/#login)
* [**docker push**](https://www.edureka.co/blog/docker-commands/#push)
* [**docker images**](https://www.edureka.co/blog/docker-commands/#images)
* [**docker rm**](https://www.edureka.co/blog/docker-commands/#rm)
* [**docker rmi**](https://www.edureka.co/blog/docker-commands/#rmi)
* [**docker build**](https://www.edureka.co/blog/docker-commands/#build)

1.**docker –version**

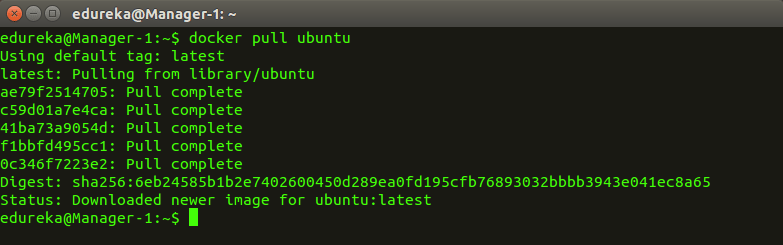
This command is used to get the currently installed version of docker



 2.**docker pull**

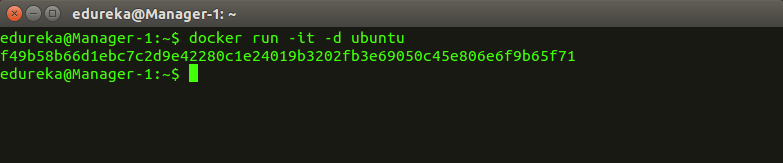
**Usage: docker pull <image name>**

This command is used to pull images from the **docker repository**(hub.docker.com)

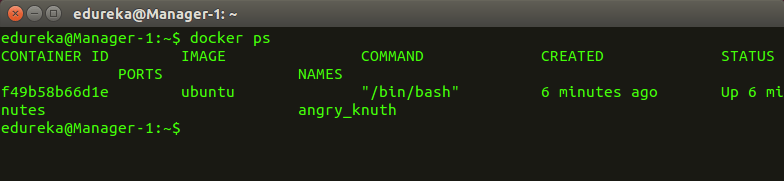
  
  
3. **docker run**

**Usage: docker run -it -d <image name>**

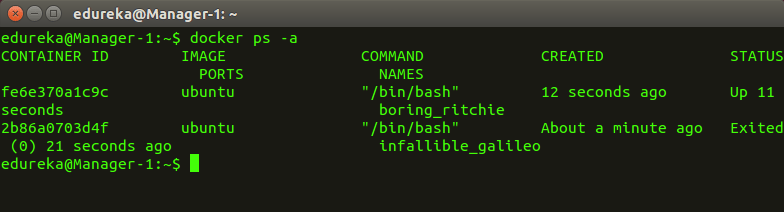
This command is used to create a container from an image

  
  
4. **docker ps**

This command is used to list the running containers

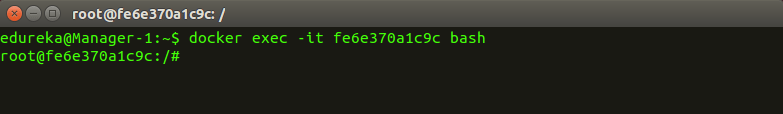
  
  
5. **docker ps -a**

This command is used to show all the running and exited containers

  
  
6. **docker exec**

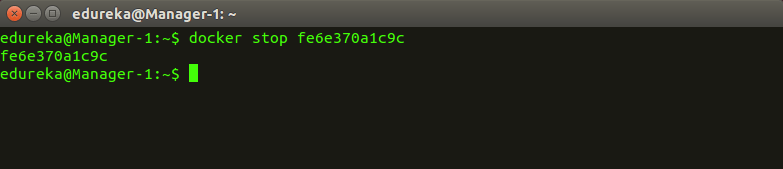
**Usage: docker exec -it <container id> bash**

This command is used to access the running container

  
  
7. **docker stop**

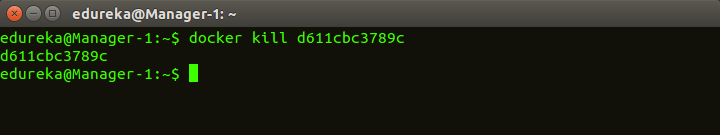
**Usage: docker stop <container id>**

This command stops a running container

  
  
8. **docker kill**

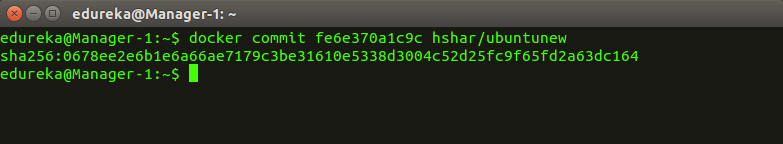
**Usage: docker kill <container id>**

This command kills the container by stopping its execution immediately. The difference between ‘docker kill’ and ‘docker stop’ is that ‘docker stop’ gives the container time to shutdown gracefully, in situations when it is taking too much time for getting the container to stop, one can opt to kill it

  
  
9. **docker commit**

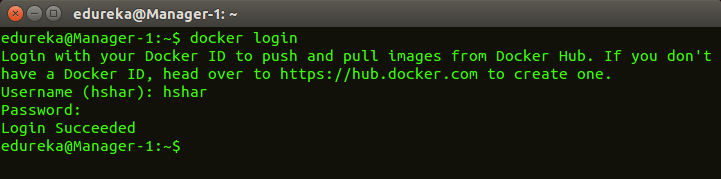
**Usage: docker commit <conatainer id> <username/imagename>**

This command creates a new image of an edited container on the local system

  
  
10. **docker login**

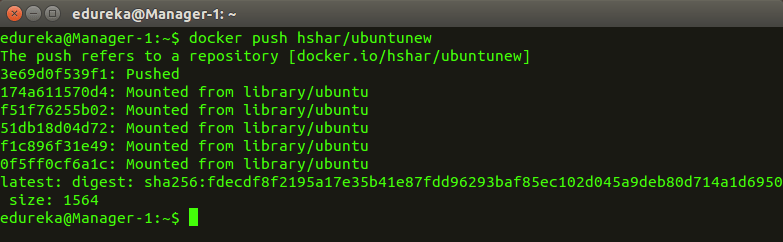
This command is used to login to the docker hub repository

Next

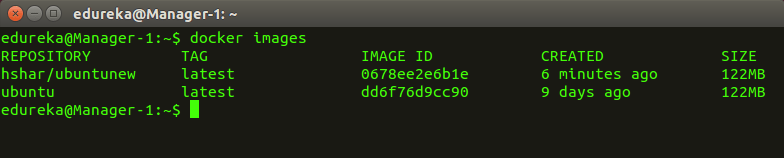
  
  
11.**docker push**

**Usage: docker push <username/image name>**

This command is used to push an image to the docker hub repository

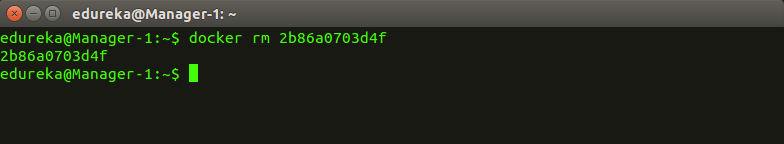
  
  
12. **docker images**

This command lists all the locally stored docker images

  
  
13. **docker rm**

**Usage: docker rm <container id>**

This command is used to delete a stopped container

  
  
14. **docker rmi**

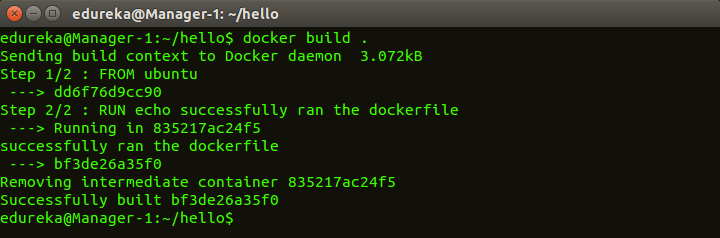
**Usage: docker rmi <image-id>**

This command is used to delete an image from local storage

  
  
15. **docker build**

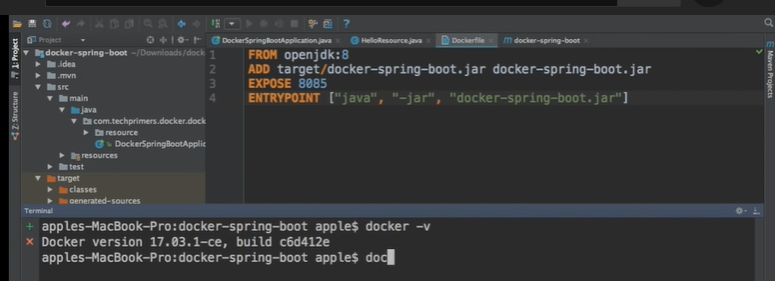
**Usage: docker build <path to docker file>**

This command is used to build an image from a specified docker file



### how do you craete a docker image for your java application?

<https://www.youtube.com/watch?v=FlSup_eelYE&t=651s>

* 

//Dockerfile:

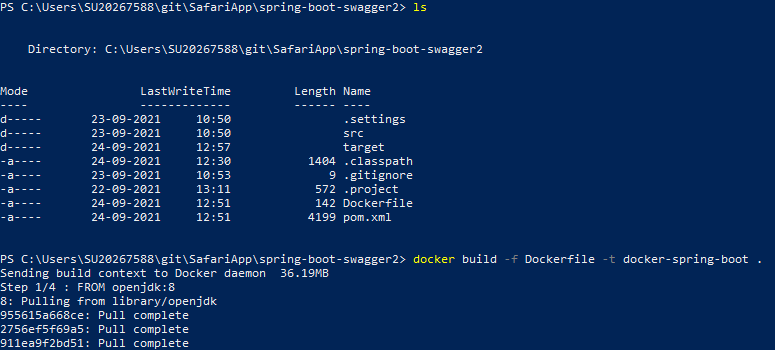
FROM openjdk:8

ADD target/docker-spring-boot.jar docker-spring-boot.jar

EXPOSE 8085

ENTRYPOINT ["java", "-jar", "docker-spring-boot.jar"]

* 1. create DockerFile
     + mention java version i.e openjdk:8
     + specify the jar path and image name.
     + mention the port number for container
     + mention the entry point (nothing but genrated jar)
  2. Then login into docker, go to the project folder.
* click on docker icon, takes few seconds to open,
* then open windows power shell.
* Check docker version using **docker --version**
* once it show docker vrsion properly then go ahead for next steps



* 1. Go to your project directory, then give docker build command.

***docker build -f Dockerfile -t docker-spring-boot .***

* it building an image and we gave image name as docker-spring-boot
* (.) dot represent directory (in my case im in current directory where my project is present)
* now it will check Dockerfile and downloads openJDK(typicaly 10 mins for 1st time)
* finally it adds the jar to container which we generated in local (target/<jar>)

4. give **docker images** command to check image is created or not

5. then give docker run command.

**docker run -p 8085:8085 docker-spring-boot**

* I’m going to publish application on 8085 port corresponding to the application which running inside the container.
* docker-spring-boot : is the image name specified
* now it creates a container .

6. any modify in code and need to push to docker?

* + Stop container 1st by clicking control C
  + Rebuild the code in local
  + Rebuild the docker image
  + Give docker run

Summary:

1. **docker –version**
2. ***docker build -f Dockerfile -t docker-spring-boot .***
3. **docker images**
4. **docker run -p 8085:8085 docker-spring-boot**

## **questions on internet**

Difference between HTTP and HTTPS ?

Ø  HTTP is considered to be unsecure and HTTPS is secure

Ø  HTTP uses port number 80 for communication and HTTPS uses 443

Ø  HTTP Works at Application Layer and HTTPS works at Transport Layer

Ø  In HTTP, Encryption is absent and Encryption is present in HTTPS through SSL.

Ø  HTTP does not require any certificates and HTTPS needs SSL Certificates

What is SSL? Why it is required?

The primary reason why SSL is used is to keep sensitive information sent across the Internet encrypted so that only the intended recipient can understand it. This is important because the information you send on the Internet is passed from computer to computer to get to the destination server. Any computer in between you and the server can see your credit card numbers, usernames and passwords, and other sensitive information if it is not encrypted with an SSL certificate. When an SSL certificate is used, the information becomes unreadable to everyone except for the server you are sending the information to. This protects it from hackers and identity thieves.



++++++++++++++++++++

What are different error status definitions?

2.WinScp   - (Windows Secure Copy) is a free and open-source SFTP, FTP. Its main function is secure file transfer between a local  and a remote computer

3.Putty       -        PuTTY is used to interact with the server directly. Putty is just a command line interface to your server.

++++++++++++++++++++++++++++++

What are common log file in server and how do you find it?

Go to /var/log directory using the following cd command:  
# cd /var/log

To list files use the following ls command:  
# ls

To view a common log file called /var/log/messages use any one of the following command:  
# less /var/log/messages

Common Linux log files names and usage:

·         /var/log/messages : General message and system related stuff

·         /var/log/auth.log : Authenication logs

·         /var/log/kern.log : Kernel logs

·         /var/log/cron.log : Crond logs (cron job)

·         /var/log/maillog : Mail server logs

·         /var/log/qmail/ : Qmail log directory (more files inside this directory)

·         /var/log/httpd/ : Apache access and error logs directory

·         /var/log/lighttpd/ : Lighttpd access and error logs directory

·         /var/log/boot.log : System boot log

·         /var/log/mysqld.log : MySQL database server log file

·         /var/log/utmp or /var/log/wtmp : Login records file

·         /var/log/yum.log : Yum command log file.

+++++++++++++++++++

[How do you return a JSON object from a Java Servlet](https://stackoverflow.com/questions/2010990/how-do-you-return-a-json-object-from-a-java-servlet)**?**

protected void doXxx(HttpServletRequest request, HttpServletResponse response) {

    // ...

    String json = new Gson().toJson(someObject);

    response.setContentType("application/json");

    response.setCharacterEncoding("UTF-8");

    response.getWriter().write(json);

}

++++++++++++

1) How do you differentiate between Core Java and Enterprise Java?

**Expected Ans**: Core Java is something that provides the API's like regular expression, String handling, collections. But enterprise java involves writing scalable, secure and per-formant applications which can have large user base.

2) What do you generally do after you have resolved a problem?

**Expected Ans**: Perform the Root Cause Analysis and make sure the changes done have not effected any other module.

3) What is JSON? Can you represent JSON as Java Object

**Expected Ans:**JSON stands for Javascript object notation and is used to initialize Javascript objects. Yes it can be used as Java object also. Provide more info here

4) What kind of HTTP request does the <a href="url">text</a> generate?

**Expected Ans**: It will generate HTTP GET request

5) What are the common browser issues you will keep in mind while creating a web application?

**Expected Ans**:   
1) User pressing back/refresh button  
2) Browser crashing  
3) Session issues  
4) Compatibility across web browsers

6) What steps will you take for ensuring the proper security of an web application?

**Expected Ans**: Stuff like Encryption, Authentication and Authorization

7) The Server and Database are working fine at your end but not on customer machine. What will you do?

**Expected Ans**:   
  
1) Check if the customer has not done any customizations  
2) Provide a test build same as running at my end (ask customer to take a backup of their  app)  
3) Check out how the customer is using the application

8) A web application is running but pages are loading slow. How will you figure out what the problem is?

**Expected Ans**: Look for threading, database, caching issues.

9) What is the difference between frameworks like Jquery/DOJO and AJAX?

**Expected Ans**: Jquery and DOJO are java script frameworks for writing rich web applications but AJAX is a server communication mechanism which can issue requests without page reload.

10) What are the reasons for a page not found error and how will you sort it out?

**Expected Ans**:   
1) The URL being sent is wrong  
2) The web.xml mapping is wrong  
3) The web server is down   
4) The application has not been deployed

11) How will you know whether a Java file is a servlet or not? **Expected Ans**: It will extend from HttpServlet class

12) When will you use Servlet and JSP or MVC framework? **Expected Ans:**While framework provides a number of components and allows one to concentrate more on the business logic but Servlets and JSP are used for controller and view layer respectively.

13) What are the common issues you have faced in web applications and how did you resolve them? **Expected Ans**:   
1) Server not starting up. Proper heap size not set  
2) Migrating from JBoss to Weblogic. Wrote a number of XML configurations  
Any other generic problems faced during development/support

14) How do you keep yourself updated about the latest web technologies? **Expected Ans**: Whatever websites/blogs/forums/authors you follow.

## **servlet and jsp**

The **javax.servlet and javax.servlet.http packages** represent interfaces and classes for servlet api.

Interfaces in javax.servlet package

1. Servlet
2. ServletRequest
3. ServletResponse
4. RequestDispatcher
5. ServletConfig
6. ServletContext
7. SingleThreadModel
8. Filter
9. FilterConfig
10. FilterChain  etc

1.   [What is different between web server and application server?](https://www.journaldev.com/2015/servlet-interview-questions-and-answers#web-server-vs-application-server)

o   A server which is capable of handling HTTP requests, sent by a client and respond back with a HTTP response..

o   Application server is a web server with additional functionalities such as Enterprise JavaBeans support, JMS Messaging support, Transaction Management etc  to help developers with enterprise applications.

2.   [Which HTTP method is non-idempotent?](https://www.journaldev.com/2015/servlet-interview-questions-and-answers#http-idempotent-methods)

o   GET, PUT, DELETE, HEAD, and OPTIONS are idempotent method

o   POST is non-idempotent

3.   [What is the difference between GET and POST method?](https://www.journaldev.com/2015/servlet-interview-questions-and-answers#http-get-vs-post)

4.   [What is MIME Type?](https://www.journaldev.com/2015/servlet-interview-questions-and-answers#mime-type)

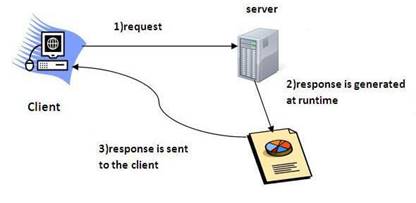
o   The “Content-Type” response header is known as MIME Type.  Some of the mostly used mime types are text/html, text/xml, application/xml etc.

5.   [What is a web application and what is it’s directory structure?](https://www.journaldev.com/2015/servlet-interview-questions-and-answers#web-application-directory)

o   Web Applications are modules that run on server to provide both static and dynamic content to the client browser.

6.   [What is a servlet?](https://www.journaldev.com/2015/servlet-interview-questions-and-answers#what-is-servlet)

**Servlet** technology is used to create a web application (resides at server side and generates a dynamic web page).



7.   [What are the advantages of Servlet over CGI?](https://www.journaldev.com/2015/servlet-interview-questions-and-answers#servlet-vs-cgi)

o    **Better performance:** because it creates a thread for each request, not process.

o    **Portability:** because it uses Java language.

o    **Robust:** JVM manages Servlets, so we don't need to worry about the memory leak, garbage collection, etc.

o    **Secure:** because it uses java language.

Steps to create a servlet example:

By implementing Servlet interface,

By inheriting GenericServlet class, (or)

By inheriting HttpServlet class

What is difference between GenericServlet and HttpServlet?

The GenericServlet is protocol independent whereas HttpServlet is HTTP protocol specific. HttpServlet provides additional functionalities such as state management etc

directory structures :

Compile the servlet

|  |  |
| --- | --- |
| servlet-api.jar | Apache Tomcat |

deployment descriptor (web.xml file)

The deployment descriptor is an xml file, from which Web Container gets the information about the servet to be invoked.

1.    <web-app>

2.

3.    <servlet>

4.    <servlet-name>sonoojaiswal</servlet-name>

5.    <servlet-class>DemoServlet</servlet-class>

6.    </servlet>

7.

8.    <servlet-mapping>

9.    <servlet-name>sonoojaiswal</servlet-name>

<url-pattern>/welcome</url-pattern>

</servlet-mapping>

</web-app>

war file?

The war file combines all the files into a single unit. So it takes less time while transferring file from client to server.

welcome-file-list in web.xml

A welcome file is the file that is invoked automatically by the server, if you don't specify any file name.

1.    <web-app>

2.     ....

3.

4.      <welcome-file-list>

5.       <welcome-file>home.html</welcome-file>

6.       <welcome-file>default.html</welcome-file>

7.      </welcome-file-list>

8.    </web-app>

9.

load on startup in web.xml : How to make sure a servlet is loaded at the application startup?

If you specify the load-on-startup in web.xml, servlet will be loaded at project deployment time or server start. So, it will take less time for responding to first request.

The 0 value will be loaded first then 1, 2, 3 and so on.

1.    <web-app>

2.     ....

3.      <servlet>

4.       <servlet-name>servlet1</servlet-name>

5.       <servlet-class>com.javatpoint.FirstServlet</servlet-class>

6.       <load-on-startup>0</load-on-startup>

7.      </servlet>

8.

9.      <servlet>

   <servlet-name>servlet2</servlet-name>

   <servlet-class>com.javatpoint.SecondServlet</servlet-class>

   <load-on-startup>1</load-on-startup>

  </servlet>

 ...

</web-app>

There are defined 2 servlets, both servlets will be loaded at the time of project deployment or server start. But, servlet1 will be loaded first then servlet2.

If you pass the negative value, servlet will be loaded at request time, at first request.

[**What are common tasks performed by Servlet Container/ Life cycle of Servlet?**](https://www.journaldev.com/2015/servlet-interview-questions-and-answers#servlet-container)

web container/servlet container same only

The server checks if the servlet is requested **for the first time**. **If yes,** web container does the following tasks:

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()

Are Servlets Thread Safe? How to achieve thread safety in servlets?

HttpServlet init() method and destroy() method are called only once in servlet life cycle, so we don’t need to worry about their synchronization. But service methods such as doGet() or doPost() are getting called in every client request and since servlet uses multithreading, we should provide thread safety in these methods.

If there are any local variables in service methods, we don’t need to worry about their thread safety because they are specific to each thread but if we have a shared resource then we can use synchronization to achieve thread safety in servlets when working with shared resources.

ServletRequest Interface:

ServletRequest is used to provide the client request information to a servlet such as content type, content length, parameter names and values, header informations, attributes etc.

//index.html

<form action="welcome" method="get">

Enter your name<input type="text" name="name"><br>

1.    <input type="submit" value="login">

2.    </form>

3.

4.    // DemoServ.java

1.    public class DemoServ extends HttpServlet{

2.    public void doGet(HttpServletRequest req,HttpServletResponse res)

3.    throws ServletException,IOException

4.    {

5.    res.setContentType("text/html");

6.    PrintWriter pw=res.getWriter();

7.

8.    String name=req.getParameter("name");  //will return value

9.    pw.println("Welcome "+name);

10.

11. pw.close();

12. }}

What is servlet collaboration? What is the inter-servlet communication?

When one servlet communicates to another servlet, it is known as servlet collaboration. There are many ways of servlet collaboration:

RequestDispacher interface

sendRedirect() method etc.

RequestDispatcher:

The RequestDispatcher interface provides the facility of dispatching the request to another resource it may be html, servlet or jsp.

[Methods of RequestDispatcher interface](https://www.javatpoint.com/requestdispatcher-in-servlet#rdmethod)

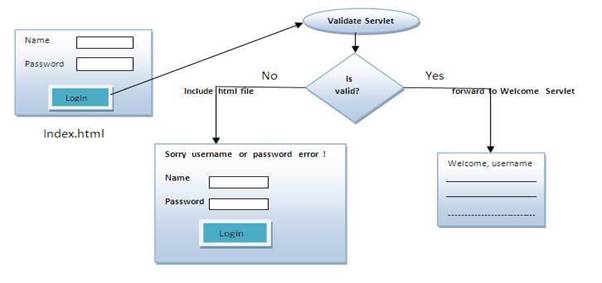
1.    [forward method](https://www.javatpoint.com/requestdispatcher-in-servlet#rdforward) : Forwards a request from a servlet to another resource (servlet, JSP file, or HTML file) on the server.

2.    [include method](https://www.javatpoint.com/requestdispatcher-in-servlet#rdinclude) : Includes the content of a resource (servlet, JSP page, or HTML file) in the response.

How to get the object of RequestDispatcher

The getRequestDispatcher() method of ServletRequest interface returns the object of RequestDispatcher

eg:



1.    //index.html

2.    <form action="servlet1" method="post">

3.    Name:<input type="text" name="userName"/><br/>

4.    Password:<input type="password" name="userPass"/><br/>

5.    <input type="submit" value="login"/>

6.    </form>

// Login .java

1.    public class Login extends HttpServlet {

2.      public void doPost(HttpServletRequest request, HttpServletResponse response)

3.            throws ServletException, IOException {

4.

5.        response.setContentType("text/html");

6.        PrintWriter out = response.getWriter();

7.

8.        String n=request.getParameter("userName");

    String p=request.getParameter("userPass");

    if(p.equals("servlet"){

        RequestDispatcher rd=request.getRequestDispatcher("servlet2");

        rd.forward(request, response);

    }

    else{

        out.print("Sorry UserName or Password Error!");

        RequestDispatcher rd=request.getRequestDispatcher("/index.html");

        rd.include(request, response);

        }

    }

  }

// WelcomeServlet .java

public class WelcomeServlet extends HttpServlet {

public void doPost(HttpServletRequest request, HttpServletResponse response)  throws ServletException, IOException {

1.        response.setContentType("text/html");

2.        PrintWriter out = response.getWriter();

3.

4.        String n=request.getParameter("userName");

5.        out.print("Welcome "+n);

6.        }

7.    }

Can you call a jsp from the servlet?

Yes, one of the way is RequestDispatcher interface for example:

1.    RequestDispatcher rd=request.getRequestDispatcher("/login.jsp");

2.    rd.forward(request,response);

SendRedirect method:

The sendRedirect() method of HttpServletResponse interface can be used to redirect response to another resource, it may be servlet, jsp or html file.

Eg:

1.    <form action="MySearcher">

2.    <input type="text" name="name">

3.    <input type="submit" value="Google Search">

4.    </form>

5.

// MySearcher .java

1.    public class MySearcher extends HttpServlet {

2.        protected void doGet(HttpServletRequest request, HttpServletResponse response)

3.                throws ServletException, IOException {

4.

5.            String name=request.getParameter("name");

6.            response.sendRedirect("https://www.google.co.in/#q="+name);

7.        }

8.    }

Difference between forward() and sendRedirect() method:

9.

|  |  |
| --- | --- |
| forward() method | sendRedirect() method |
| The forward() method works at server side. | The sendRedirect() method works at client side. |
| It sends the same request and response objects to another servlet. | It always sends a new request. |

ServletConfig Interface

·         An object of ServletConfig is created by the web container for each servlet.

·         This object can be used to get configuration information from web.xml file.

Advantage/Purpose of use:  The core advantage of ServletConfig is that you don't need to edit the servlet file if information is modified from the web.xml file.

Methods of ServletConfig interface

**public String getInitParameter(String name):**Returns the parameter value for the specified parameter name.

**public Enumeration getInitParameterNames():**Returns an enumeration of all the initialization parameter names.

**public String getServletName():**Returns the name of the servlet.

**public ServletContext getServletContext():**Returns an object of ServletContext.

Eg:

1.    public class DemoServlet extends HttpServlet {

2.    public void doGet(HttpServletRequest request, HttpServletResponse response)

3.        throws ServletException, IOException {

4.

5.        response.setContentType("text/html");

6.        PrintWriter out = response.getWriter();

7.

8.        ServletConfig config=getServletConfig();

9.        String driver=config.getInitParameter("driver");

    out.print("Driver is: "+driver);

    out.close();

    }

}

//web.xml

1.    <web-app>

2.

3.    <servlet>

4.    <servlet-name>DemoServlet</servlet-name>

5.    <servlet-class>DemoServlet</servlet-class>

6.

7.    <init-param>

8.    <param-name>driver</param-name>

9.    <param-value>sun.jdbc.odbc.JdbcOdbcDriver</param-value>

</init-param>

</servlet>

</web-app>

Example of ServletConfig to get all the initialization parameters

1.    public class DemoServlet extends HttpServlet {

2.    public void doGet(HttpServletRequest request, HttpServletResponse response) throws ServletException, IOException {

3.

4.        response.setContentType("text/html");

5.        PrintWriter out = response.getWriter();

6.

7.        ServletConfig config=getServletConfig();

8.        Enumeration<String> e=config.getInitParameterNames();

9.

    String str="";

    while(e.hasMoreElements()){

    str=e.nextElement();

    out.print("<br>Name: "+str);

    out.print(" value: "+config.getInitParameter(str));

    }

    out.close();

}

//web.xml

1.    <web-app>

2.

3.    <servlet>

4.    <servlet-name>DemoServlet</servlet-name>

5.    <servlet-class>DemoServlet</servlet-class>

6.

7.    <init-param>

8.    <param-name>username</param-name>

9.    <param-value>system</param-value>

10. </init-param>

11.

12. <init-param>

13. <param-name>password</param-name>

14. <param-value>oracle</param-value>

15. </init-param>

16.

17. </servlet>

18. </web-app>

ServletContext Interface

·         An object of ServletContext is created by the web container at time of deploying the project.

·         This object can be used to get configuration information from web.xml file.

·         There is only one ServletContext object per web application.

Adv/purpose: what is **<context-param>**  ?

If any information is shared to many servlet, it is better to provide it from the web.xml file using the **<context-param>** element.

Eg:

1.    public class DemoServlet extends HttpServlet{

2.    public void doGet(HttpServletRequest req,HttpServletResponse res)  throws ServletException,IOException

3.    {

4.    res.setContentType("text/html");

5.    PrintWriter pw=res.getWriter();

6.

7.    //creating ServletContext object

8.    ServletContext context=getServletContext();

//Getting the value of the initialization parameter and printing it

String driverName=context.getInitParameter("driver");

pw.println("driver name is="+driverName);

pw.close();

}}

//web.xml

1.    <web-app>

2.

3.    <servlet>

4.    <servlet-name>sonoojaiswal</servlet-name>

5.    <servlet-class>DemoServlet</servlet-class>

6.    </servlet>

7.

8.    <context-param>

9.    <param-name> driver </param-name>

<param-value>sun.jdbc.odbc.JdbcOdbcDriver</param-value>

</context-param>

10. </web-app>

Difference between ServletConfig and ServletContext

|  |
| --- |
| The servletconfig object refers to the single servlet whereas servletcontext object refers to the whole web application. |

Attribute in Servlet and their scope?

Attributes are object that can be set, get or removed from one of the following scopes:

request scope

session scope

application scope

purpose: It is mainly used to share information between one servlet to another.

Some of methods: **setAttribute(String name,Object object) :**sets the given object in the application scope.

**getAttribute(String name) :**Returns the attribute for the specified name.

**removeAttribute(String name) :**Removes the attribute with the given name from the servlet context.

Eg:

public class DemoServlet1 extends HttpServlet{

1.    public void doGet(HttpServletRequest req,HttpServletResponse res)

2.    {

3.    try{

4.

5.    res.setContentType("text/html");

6.    PrintWriter out=res.getWriter();

7.

8.    ServletContext context=getServletContext();

9.    context.setAttribute("company","IBM");

out.println("Welcome to first servlet");

out.println("<a href='servlet2'>visit</a>");

out.close();

}catch(Exception e){out.println(e);}

}}

// DemoServlet2.java

1.    public class DemoServlet2 extends HttpServlet{

2.    public void doGet(HttpServletRequest req,HttpServletResponse res)

3.    {

4.    try{

5.

6.    res.setContentType("text/html");

7.    PrintWriter out=res.getWriter();

8.

9.    ServletContext context=getServletContext();

10. String n=(String)context.getAttribute("company");

11.

12. out.println("Welcome to "+n);

13. out.close();

14.

15. }catch(Exception e){out.println(e);}

16. }}

Session Tracking/ session management  :

Session Tracking is a way to maintain state (data) of an user. It is also known as session management in servlet.

Http protocol is a stateless so we need to maintain state using session tracking techniques. Each time user requests to the server, server treats the request as the new request.

There are four techniques used in Session tracking:

Cookies

Hidden Form Field

URL Rewriting

HttpSession

Cookies : A cookie is a small piece of information that is persisted between the multiple client requests.

How Cookie works : By default, each request is considered as a new request. In cookies technique, we add cookie with response from the servlet. So cookie is stored in the cache of the browser. After that if request is sent by the user, cookie is added with request by default. Thus, we recognize the user as the old user.

How to create Cookie?

1.    Cookie ck=new Cookie("user","sonoo jaiswal");//creating cookie object

2.    response.addCookie(ck);//adding cookie in the response

What is difference between Cookies and HttpSession?

Cookie works at client side whereas HttpSession works at server side.

Servlet Filter

A filter is an object that is invoked at the preprocessing and postprocessing of a request.

Purpose: It is mainly used to perform filtering tasks such as conversion, logging, compression, encryption and decryption, input validation etc.

Usage of Filter

recording all incoming requests

logs the IP addresses of the computers from which the requests originate

conversion

data compression

encryption and decryption

input validation etc.

The servlet filter is pluggable, i.e. its entry is defined in the web.xml file, if we remove the entry of filter from the web.xml file, filter will be removed automatically and we don't need to change the servlet.

|  |  |
| --- | --- |
| public void init(FilterConfig config) | init() method is invoked only once. It is used to initialize the filter. |
| public void doFilter(HttpServletRequest request,HttpServletResponse response, FilterChain chain) | doFilter() method is invoked every time when user request to any resource, to which the filter is mapped.It is used to perform filtering tasks. |
| public void destroy() | This is invoked only once when filter is taken out of the service. |

How to count the total number of visitors and whole response time for a request?  -> using Filter

1.    public class MyFilter implements Filter{

2.        static int count=0;

public void doFilter(ServletRequest req, ServletResponse res,

            FilterChain chain) throws IOException, ServletException {

        PrintWriter out=res.getWriter();

        chain.doFilter(request,response);

        out.print("<br/>Total visitors "+(++count));

        out.close();

    }

// response time

public void doFilter(ServletRequest req, ServletResponse res,

1.                FilterChain chain) throws IOException, ServletException {

2.

3.            PrintWriter out=res.getWriter();

4.            long before=System.currentTimeMillis();

5.

6.            chain.doFilter(request,response);

7.

8.            long after=System.currentTimeMillis();

        out.print("<br/>Total response time "+(after-before)+" miliseconds");

        out.close();

}

1.  How to get the actual path of servlet in server?

We can use following code snippet to get the actual path of the servlet in file system.

getServletContext().getRealPath(request.getServletPath())

2.  How to get the server information in a servlet?

We can use below code snippet to get the servlet information in a servlet through servlet context object.

getServletContext().getServerInfo()

How to get the IP address of client in servlet?

We can use request.getRemoteAddr() to get the client IP address in servlet.

[How can we create deadlock situation in servlet?](https://www.journaldev.com/2015/servlet-interview-questions-and-answers#servlet-deadlock)

We can create deadlock in servlet by making a loop of method invocation, just call doPost() method from doGet() method and doGet() method to doPost() method to create deadlock situation in servlet.

[What is SingleThreadModel interface?](https://www.journaldev.com/2015/servlet-interview-questions-and-answers#singlethreadmodel-interface)

SingleThreadModel interface was provided for thread safety and it guarantees that no two threads will execute concurrently in the servlet’s service method.

[Do we need to override service() method?](https://www.journaldev.com/2015/servlet-interview-questions-and-answers#override-service-method)

When servlet container receives client request, it invokes the service() method which in turn invokes the doGet(), doPost() methods based on the HTTP method of request.

[Is it good idea to create servlet constructor?](https://www.journaldev.com/2015/servlet-interview-questions-and-answers#servlet-constructor)

We can define a constructor for servlet but I don’t think its of any use

[How can we invoke another servlet in a different application?](https://www.journaldev.com/2015/servlet-interview-questions-and-answers#servlet-sendRedirect)

we can use ServletResponse sendRedirect() method and provide complete URL of another servlet. This sends the response to client with response code as 302 to forward the request to another URL

[why we should override only no-agrs init() method.](https://www.journaldev.com/2015/servlet-interview-questions-and-answers#override-init)

If we have to initialize some resource before we want our servlet to process client requests, we should override init() method

JSP Tutorial

**JSP** technology is used to create web application just like Servlet technology.  A JSP page consists of HTML tags and JSP tags.

Advantages of JSP over Servlet

*1.Easy to maintain*

JSP can be easily managed because we can easily separate our business logic with presentation logic. In Servlet technology, we mix our business logic with the presentation logic.

*2.) Fast Development: No need to recompile and redeploy*

If JSP page is modified, we don't need to recompile and redeploy the project. The Servlet code needs to be updated and recompiled if we have to change the look and feel of the application.

The Lifecycle of a JSP Page

The JSP pages follow these phases:

Translation of JSP Page

Compilation of JSP Page

Classloading (the classloader loads class file)

Instantiation (Object of the Generated Servlet is created).

Initialization ( the container invokes jspInit() method).

Request processing ( the container invokes \_jspService() method).

Destroy ( the container invokes jspDestroy() method).

JSP Scripting elements

The scripting elements provides the ability to insert java code inside the jsp. There are three types of scripting elements:

scriptlet tag

expression tag

declaration tag