# OOPs concept

* Object oriented programing concept consist of 4 basic principle. Inheritance, Polymorphism, Encapsulation and Abstraction.
* **Inheritance**–when one object/class acquires its all property from parent object/class it is called Inheritance. It helps us in case for code reusability.
* **Polymorphism** –when one task is performed by many ways called polymorphism. In java it can be achieved through ‘method overloading’ and ‘method overriding’.
* **Encapsulation**- Binding code and data together into a single unit is known as Encapsulation. Like a wrapper. Example – Class is a classic example of encapsulation.
* **Abstraction** –Hiding Implementation and showing only functionality to end user is known as abstraction. It can be achieved through ‘abstract class’ and ‘interface’.

# Method Overloading

* Method overloading through changing not of argument6.
* Method overloading through changing data type of the arguments.
* Method overloading is not possible through changing return type of the method.

# Constructor

Constructors are used for initialize object.

* Constructor invoked at the time of object creation.
* Constructor does not have any return type.

Two type of constructor –

* Default constructor – it has no argument. Use to initialize default 0 and null values.
* Parameterized constructor – use to initialize object with parameter as argument.

Constructor can be overloaded.

Constructor does not have any return type.

Constructor can have access modifier.

# Static

Static keyword. Static used for memory management purpose. Static can be used with – variable, method, block, nested class.

* Static variable – static variable gets its memory location only once at the time of class loading.

Example –

/\* counter program without static variable \*/

Class Counter{

Int count = 0;

Counter (){

Count ++;

Syso (count);

}

Public static void main (string [] args){

Counter c1 = new counter ();

Counter c2 = new counter ();

Counter c3 = new counter ();

}

}

O/p –

1

1

1

/\* Counter program with static variable \*/

Class Counter{

Static Int count = 0;

Counter (){

Count ++;

Syso (count);

}

Public static void main (string [] args){

Counter c1 = new counter ();

Counter c2 = new counter ();

Counter c3 = new counter ();

}

}

O/p –

1

2

3

* Static Method –

Java static method belongs to its class rather than object of class. Static method is called without any object of class. Static method can access static variable and can change its value.

Restrictions – The static method cannot use non static member directly. This and supar key word cannot be used in static context.

Example –

Class st {

Int a = 40;

Public static void main(string [] args){

Syso (a);

}

}

O/p – compile time error.

* Static Block

Static block is used for initialized static variables. Static block executed before main method.

Can we use Static constructor ?

No we can not use.

Can constructor initialize static variable?

Yes, we can. But we should avoid doing this. Better to use static block.

Class st{

Static int a;

String b;

St (int a, string b){

This.a = a;

This.b = b;

}

St s1 = new St (10,”s”);

St s2 = new St(20,”a”);

Now, 10 become irrelevant.

# Inheritance in JAVA

Class child extends parents {}

# Method Overriding

If subclass has the same method as parent class known as method overriding. In other words if child class has specific implementation of method of its parent class known as overriding.

It is used for run time polymorphism.

Rules :

* Child class Method must have same name as parent.
* Must have same argument.
* Must be IS-A relation(inheritance).
* The argument list should be exactly the same as that of the overridden method.
* The return type should be the same or a subtype of the return type declared in the original overridden method in the superclass.
* The access level cannot be more restrictive than the overridden method's access level. For example: If the superclass method is declared public then the overridding method in the sub class cannot be either private or protected.
* Instance methods can be overridden only if they are inherited by the subclass.
* A method declared final cannot be overridden.
* A method declared static cannot be overridden but can be re-declared.
* If a method cannot be inherited, then it cannot be overridden.
* A subclass within the same package as the instance's superclass can override any superclass method that is not declared private or final.
* A subclass in a different package can only override the non-final methods declared public or protected.
* An overriding method can throw any uncheck exceptions, regardless of whether the overridden method throws exceptions or not. However, the overriding method should not throw checked exceptions that are new or broader than the ones declared by the overridden method. The overriding method can throw narrower or fewer exceptions than the overridden method.
* Constructors cannot be overridden.

Can we override static method?

No, Static is bound to class and other instance method bound to object which are in heap memory. (Since main method is static Can't override)

**Method Overloading** is compile time polymorphism**Method Overriding** is run time polymorphism.

**Method overriding with access modifier**:

We have 4 access-specifier.The following sequence represents the heirarchy from weakest/less restrictive access specifier to strongest/more restrictive access specifier

* public
* protected
* no-access or default
* private

**If super class method is public,while overriding it in child class,it should be public,Since it is the weakest access specifier or least restrictive access specifier**  
  
***OverrideTest.java***

example

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15 | classDemo  {      publicvoidsayHello()      {          System.out.println("Demo:sayHello()");      }  }    publicclassOverrideTest extendsDemo  {      publicvoidsayHello()//cannot be declared private,protected,no access      {          System.out.println("OverrideTest:sayHello()");      }  } |

**If super class method is protected,while overriding it in child class,it can be public or protected but not anything else**

***OverrideTest.java***

example

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15 | classDemo  {   protectedvoidsayHello()   {    System.out.println("Demo:sayHello()");   }  }    publicclassOverrideTest extendsDemo  {   publicvoidsayHello()//can be public,protected But not anything else   {    System.out.println("OverrideTest:sayHello()");   }  } |

**If superclass method is default,then while overriding it in child class,it can be public,protected or no access,but cannot be private.**

***OverrideTest.java***

example

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15 | classDemo  {   voidsayHello() //default access specifier   {    System.out.println("Demo:sayHello()");   }  }    publicclassOverrideTest extendsDemo  {   protectedvoidsayHello()//can be public,protected,default But not private   {    System.out.println("OverrideTest:sayHello()");   }  } |

**If superclass method is private,then while overriding it in child class,it can be anything. Since private method cannot be overrided.**

***OverrideTest.java***

example

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15 | classDemo  {   privatevoidsayHello() //cannot be overrided   {    System.out.println("Demo:sayHello()");   }  }    publicclassOverrideTest extendsDemo  {   publicvoidsayHello()//this is separate method,which can be overrided later by its subclass   {    System.out.println("OverrideTest:sayHello()");   }  } |

**Method overriding with exception modifier:**

**Rule1 :** if super class method does not declare any exception subclass method Can't declare checked exception. But can declare unchecked exception.

**Rule2 :** if super class method declare any exception subclass method can declare no exception, childexception. But can't declare super exception.

**Can Over loaded method be overridden and over redden method be overloaded?**

**Yes, both can be possible**

**Example-**

**public** **class** ReservBank {

**void** InterestRate(){

System.***out***.println("Basic Interest Rate = 6%");

}

**void** IntersetRate(**int** rate){

System.***out***.println("Basic Interest Rate = "+rate+"%" );

}

}

**public** **class** IndianBank **extends** ReservBank{

**void** InterestRate(){

System.***out***.println("Basic Interset Rate = 6% - call from child class");

}

**void** IntersetRate(**int** rate){

System.***out***.println("Basic Interest Rate = "+rate+"%"+" - call from child class" );

}

**void** IntersetRate(String rate){

System.***out***.println("Basic Interest Rate = "+rate+"%" );

}

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

// **TODO** Auto-generated method stub

ReservBank rb = **new** IndianBank();

IndianBank ib = **new** IndianBank();

ReservBank rb1 = **new** ReservBank();

ib.InterestRate();

ib.IntersetRate(8);

ib.IntersetRate("nine");

rb.InterestRate();

rb.IntersetRate(8);

rb1.InterestRate();

rb1.IntersetRate(9);

}

}

**Output –**

Basic Interset Rate = 6% - call from child class

Basic Interest Rate = 8% - call from child class

Basic Interest Rate = nine%

Basic Interset Rate = 6% - call from child class

Basic Interest Rate = 8% - call from child class

Basic Interest Rate = 6%

Basic Interest Rate = 9%

# Super

Use: super keyword use to refer immediate parent class object.

Super.method() – for parent class method.

Super.variable – for parent class instance variable.

Super () – invoke parent class constructor.

# Final

Final keyword use with variables, method, class.

Used for:

* Stop value change.
* Stop overriding.
* Stop inheritance.

Points to remember-

* Final method can be inherited.
* Blank final variable can be initialized in constructor.
* Static final variable can be initialized in static blog.
* Constructor cannot be final.

# Instance initializer block

At the time of compilation, compiler copy this to constructor after super.

Example constructor coder at compilation time-

Cons (){

Super ();

{ block code}

Constructor code;

}

Rules:

* It is created at the time of object creation.
* It is invoked after super()
* It invoked is same order they created.

# Polymorphism

Can be achieved through **method overloading** and **method overriding.**

Static polymorphism : method overloading

Dynamic polymorphism : method overriding

What is upcusting?

Class a {}

Class b extends a {

A obj = new b();

}

When object of super class refers to child class.

Java run time polymorphism can't be achieved by data member.

Example:

Class a{

Int I = 10;

}

Class b extends a{

Int I = 100;

Public static void main(string args[]){

A obj = new b ();

Syso (i);

}}

O/p – 10

# Primitive &Non primitive data type

Primitive data types are pre defined. Like int float etc.

Non primitives are user defined like reference variable.

# Static & Dynamic binding

Connecting a method call to its body is called binding.

Static – early binding. When the type is determined at compilation time known as static binding. Any method in a class is private or final or static is early binding.

Dynamic – see below example

Class a{

Void f (){

Syso (“a”);

}

}

Class b extends a {

Void f (){

Syso (“b”);

}

Public static void main (string args []){

A obj = new b ();

Obj.f ();

}

}

Type of object can not be determined at time of Compilation sinceinstance of a is also instance of b

# Abstract class

* Abstract class must have at least one abstract method.
* Abstract method cannot be final.
* Implementing class must implement abstract method.
* Abstract class can’t have object. Means instance cannot be created.
* Abstract class can have implemented method, constructor, member variable.

Example –

**public** **abstract** **class** ClassAbstract {

ClassAbstract(){

System.***out***.println("Constructor of class - ClassAbstract");

}

**abstract** **void** AbstractPrint();

**void** Print(){

System.***out***.println("Print method in class - AbstractPrint ");

};

}

**public** **class** Print **extends** ClassAbstract {

@Override

**void** AbstractPrint() {

System.***out***.println("Implemented Method in class - Print");

}

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

Print obj = **new** Print();

obj.AbstractPrint();

obj.Print();

}

}

Output –

Constructor of class - ClassAbstract

Implemented Method in class - Print

Print method in class – AbstractPrint

# Tight coupling & Loose Coupling

## ****Tight-Coupling:-****

1.    While creating complex application in java, the logic of one class will call the logic of another class to provide same service to the clients.  
  
2.    If one class calling another class logic then it is called collaboration.  
  
3.    When one class is collaborating with another class then there exists tight-coupling between the two classes.  
  
4.    If one class wants to call the logic of a second class then they first class need an object of second class it means the first class create an object of second class.  
  
5.    For example, if we have two classes called traveller and car, traveller class is calling logic of car class; in this case traveller class creates an object of car class.  
  
6.      In the above traveller class and car classes, car class object of dependency for traveller object.  
  
**Example:-**

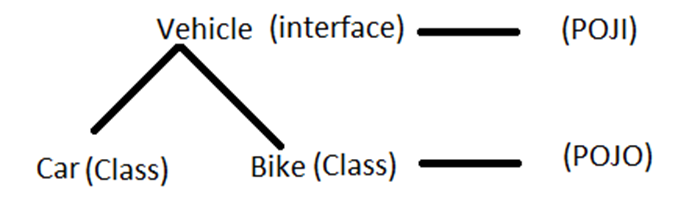
|  |  |
| --- | --- |
| Picture | Picture |

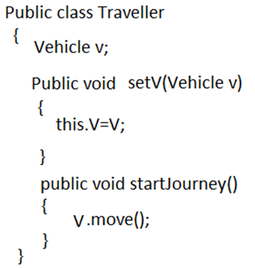
7.    In the above example traveller object is tightly coupled with car object because in place car object  if you want to use bike object then, we need to make changes in Traveller class  
  
    Example :-

|  |  |
| --- | --- |
| Picture | Picture |

## ****Loose-Coupling:-****

1.    In Loose-Coupling, when one object is depending on another class object, some external entity will provide that dependency object to the main object that external object we call as a Container.  
  
2.    In order to get loose-coupling between objects the following two rules are required  
  
1.    The classes should follow POJI/POJO model.  
  
2.    Apply dependency injection mechanism.  
  
 For example:-





3.    In the above traveler class, an external entity injects either car (or) Bike object.  
  
4.    In traveler, these are no changes required we are shifting the dependency from car to a Bike.  
  
5.    In the above traveler class, we are token vehicle reference, so that an external object (Container) can injects either car object (or) Bike object, depends on requirement if a traveler.  
  
6.    In spring frame work, spring container follows dependency injection mechanism and injects the dependency objects required for a main object.  
  
7.    Spring frame work is much success because of one of the main reason is it promotes Loose-Coupling between the objects.

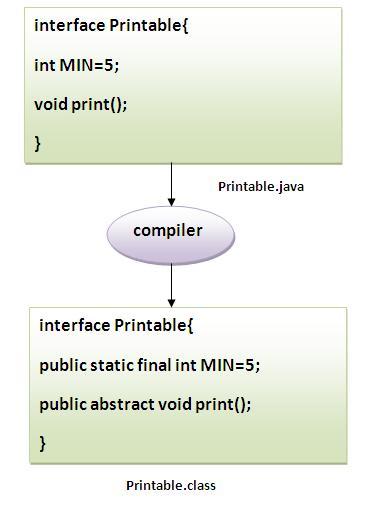
# Interface

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

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

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

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



**In Java 8 we can have static and default implemented method in interface.**

**Example:**

**interface** javaInterface {

**int** ***test*** =10;

**void** print();

**default** **void** print1(){

System.***out***.println("interface default method");

}

**static** **void** print2(){

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

}

}

**public** **class** ClassCall **implements** javaInterface {

@Override

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

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

}

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

ClassCall obj = **new** ClassCall();

javaInterface obj1 = **new** ClassCall();

//javaInterface obj2 = new javaInterface();

obj.print();

obj.print1();

obj1.print();

obj1.print1();

javaInterface.*print2*();

}

}

# **Difference between abstract class and interface**

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

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

|  |  |
| --- | --- |
| **Abstract class** | **Interface** |
| 1) Abstract class can **have abstract and non-abstract** methods. | Interface can have **only abstract** methods. |
| 2) Abstract class **doesn't support multiple inheritance**. | Interface **supports multiple inheritance**. |
| 3) Abstract class **can have final, non-final, static and non-static variables**. | Interface has **only static and final variables**. |
| 4) Abstract class **can have static methods, main method and constructor**. | Interface **can't have static methods, main method or constructor**. |
| 5) Abstract class **can provide the implementation of interface**. | Interface **can't provide the implementation of abstract class**. |
| 6) The **abstract keyword** is used to declare abstract class. | The **interface keyword** is used to declare interface. |
| 7) **Example:** public abstract class Shape{ public abstract void draw(); } | **Example:** public interface Drawable{ void draw(); } |

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

# Access Modifiers in JAVA

# **Access Modifiers in java**

1. [private access modifier](http://www.javatpoint.com/access-modifiers#accessprivate)
2. [Role of private constructor](http://www.javatpoint.com/access-modifiers#accessprivatecons)
3. [default access modifier](http://www.javatpoint.com/access-modifiers#accessdefault)
4. [protected access modifier](http://www.javatpoint.com/access-modifiers#accessprotected)
5. [public access modifier](http://www.javatpoint.com/access-modifiers#accesspublic)
6. [Applying access modifier with method overriding](http://www.javatpoint.com/access-modifiers#accessoverriding)

There are two types of modifiers in java: **access modifiers** and **non-access modifiers**.

The access modifiers in java specifies accessibility (scope) of a data member, method, constructor or class.

There are 4 types of java access modifiers:

1. private
2. default
3. protected
4. public

There are many non-access modifiers such as static, abstract, synchronized, native, volatile, transient etc. Here, we will learn access modifiers.

### **1) private access modifier**

|  |
| --- |
| The private access modifier is accessible only within class. |

### **Simple example of private access modifier**

|  |
| --- |
| In this example, we have created two classes A and Simple. A class contains private data member and private method. We are accessing these private members from outside the class, so there is compile time error. |

1. **class** A{
2. **private** **int** data=40;
3. **private** **void** msg(){System.out.println("Hello java");}
4. }
6. **public** **class** Simple{
7. **public** **static** **void** main(String args[]){
8. A obj=**new** A();
9. System.out.println(obj.data);//Compile Time Error
10. obj.msg();//Compile Time Error
11. }
12. }

### **Role of Private Constructor**

|  |
| --- |
| If you make any class constructor private, you cannot create the instance of that class from outside the class. For example: |

1. **class** A{
2. **private** A(){}//private constructor
3. **void** msg(){System.out.println("Hello java");}
4. }
5. **public** **class** Simple{
6. **public** **static** **void** main(String args[]){
7. A obj=**new** A();//Compile Time Error
8. }
9. }

#### Note: A class cannot be private or protected except nested class.

### **2) default access modifier**

|  |
| --- |
| If you don't use any modifier, it is treated as **default** bydefault. The default modifier is accessible only within package. |

### **Example of default access modifier**

|  |
| --- |
| In this example, we have created two packages pack and mypack. We are accessing the A class from outside its package, since A class is not public, so it cannot be accessed from outside the package. |

1. //save by A.java
2. **package** pack;
3. **class** A{
4. **void** msg(){System.out.println("Hello");}
5. }
6. //save by B.java
7. **package** mypack;
8. **import** pack.\*;
9. **class** B{
10. **public** **static** **void** main(String args[]){
11. A obj = **new** A();//Compile Time Error
12. obj.msg();//Compile Time Error
13. }
14. }

In the above example, the scope of class A and its method msg() is default so it cannot be accessed from outside the package.

### **3) protected access modifier**

The **protected access modifier** is accessible within package and outside the package but through inheritance only.

The protected access modifier can be applied on the data member, method and constructor. It can't be applied on the class.

### **Example of protected access modifier**

In this example, we have created the two packages pack and mypack. The A class of pack package is public, so can be accessed from outside the package. But msg method of this package is declared as protected, so it can be accessed from outside the class only through inheritance.

1. //save by A.java
2. **package** pack;
3. **public** **class** A{
4. **protected** **void** msg(){System.out.println("Hello");}
5. }
6. //save by B.java
7. **package** mypack;
8. **import** pack.\*;
10. **class** B **extends** A{
11. **public** **static** **void** main(String args[]){
12. B obj = **new** B();
13. obj.msg();
14. }
15. }

Output:Hello

### **4) public access modifier**

|  |
| --- |
| The **public access modifier** is accessible everywhere. It has the widest scope among all other modifiers. |

### **Example of public access modifier**

1. //save by A.java
3. **package** pack;
4. **public** **class** A{
5. **public** **void** msg(){System.out.println("Hello");}
6. }
7. //save by B.java
9. **package** mypack;
10. **import** pack.\*;
12. **class** B{
13. **public** **static** **void** main(String args[]){
14. A obj = **new** A();
15. obj.msg();
16. }
17. }

Output:Hello

### **Understanding all java access modifiers**

Let's understand the access modifiers by a simple table.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Access Modifier** | **within class** | **within package** | **outside package by subclass only** | **outside package** |
| **Private** | Y | N | N | N |
| **Default** | Y | Y | N | N |
| **Protected** | Y | Y | Y | N |
| **Public** | Y | Y | Y | Y |

# Difference between Method Overloading & Overriding

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

# Encapsulation in Java

**Encapsulation in java** is a process of wrapping code and data together into a single unit, for example capsule i.e. mixed of several medicines.



We can create a fully encapsulated class in java by making all the data members of the class private. Now we can use setter and getter methods to set and get the data in it.

The **Java Bean** class is the example of fully encapsulated class.

### **Advantage of Encapsulation in java**

By providing only setter or getter method, you can make the class **read-only or write-only**.

It provides you the **control over the data**. Suppose you want to set the value of id i.e. greater than 100 only, you can write the logic inside the setter method.

### **Simple example of encapsulation in java**

Let's see the simple example of encapsulation that has only one field with its setter and getter methods.

1. //save as Student.java
2. **package** com.javatpoint;
3. **public** **class** Student{
4. **private** String name;
6. **public** String getName(){
7. **return** name;
8. }
9. **public** **void** setName(String name){
10. **this**.name=name
11. }
12. }
13. //save as Test.java
14. **package** com.javatpoint;
15. **class** Test{
16. **public** **static** **void** main(String[] args){
17. Student s=**new** Student();
18. s.setName("vijay");
19. System.out.println(s.getName());
20. }
21. }

Compile By: javac -d . Test.java

Run By: java com.javatpoint.Test

Output: vijay

# String

Palindrome Code:

**public** **class** Palindrome {

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

{

String chr="madam";

String reverse="";

**int** len=chr.length();

**for**(**int** i=len-1;i>-1;i--)

{

reverse=reverse+chr.charAt(i);

}

System.***out***.println("Reversed String is: "+reverse);

**if**(chr.equals(reverse))

{

System.***out***.println("The Given String is Palindrome");

}**else**{

System.***out***.println("The Given String is not a Palindrome");

}

}

}

You are taking Locator Type and Locator value in a single string and then you use split function based on your separator. i.e. “:”. Suppose your locator has a “:” in it. So Split(“:”)[1] will not work. How to overcome?

**public** **class** ClassExample{

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

**int** i;

String strValue = "Souni:selenium:value";

**for**(i=0;i<strValue.length();i++){

**if**(strValue.charAt(i)==':')

**break**;

}

String strSubvalue2 = strValue.substring(i+1);

System.***out***.println(strSubvalue2);

}

# I/O Scanner class

**import** java.util.Scanner;

**public** **class** ClassScanner {

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

Scanner scan = **new** Scanner(System.***in***);//system.in is system input stream

System.***out***.println("enter your name:");

String Name = scan.nextLine();

System.***out***.println("your input = "+Name);

}

}

# File I/O – working with text file

**public** **class** ClassFile {

**public** **void** folderCreator(String path){

//Step one - create object of file class ad provide the path of file.

File file = **new** File(path);

//mkdirs() method use to create folder

file.mkdirs();

**if**(file.exists())

System.***out***.println("File Created");

**else**

System.***out***.println("No File Found");

file.delete();

}

**public** **void** fileCreator(String path) **throws** IOException{

File file = **new** File(path);

file.createNewFile();

}

**public** **void** fileWriter(String path) **throws** IOException{

//Step1 create object of FileWriter and pass the path of the text file.

FileWriter fw = **new** FileWriter(path);

//create BufferedWriter object and pass the FileWriter obj to it

BufferedWriter br = **new** BufferedWriter(fw);

br.write("Souni");

br.newLine();

br.write("Shamik");

br.close();

fw.close();

}

**public** **void** fileReader(String path) **throws** IOException{

FileReader fr = **new** FileReader(path);

BufferedReader br = **new** BufferedReader(fr);

String line = **null**;

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

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

}

br.close();

fr.close();

}

}

**public** **class** ClassMain {

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

String path = "C:\\Users\\Souni\\Desktop\\Selenium";

String pathTXT = "C:\\Users\\Souni\\Desktop\\Selenium.txt";

ClassFile obj = **new** ClassFile();

//obj.folderCreator(path);

//obj.fileCreator(pathTXT);

//obj.fileWriter(pathTXT);

obj.fileReader(pathTXT);

}

}

# Working with property file

**public** **class** ClassProperty {

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

File file = **new** File("C:\\ JavaForPractice\\PropertyFile.properties");

FileInputStream fs = **new** FileInputStream(file);

Properties prop = **new** Properties();

prop.load(fs);

String s1 = prop.getProperty("java");

System.***out***.println(s1);

}

}

# **Immutable String in Java**

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

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

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

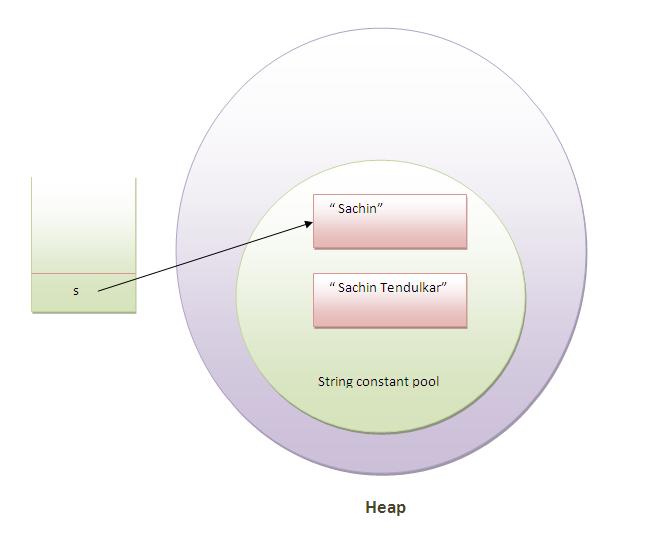
1. class Testimmutablestring{
2. public static void main(String args[]){
3. String s="Sachin";
4. s.concat(" Tendulkar");//concat() method appends the string at the end
5. System.out.println(s);//will print Sachin because strings are immutable objects
6. }
7. }



[**Test it Now**](http://www.javatpoint.com/opr/test.jsp?filename=Testimmutablestring)

Output:Sachin

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



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

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

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



[**Test it Now**](http://www.javatpoint.com/opr/test.jsp?filename=Testimmutablestring1)

Output:Sachin Tendulkar

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

### **Why string objects are immutable in java?**

|  |
| --- |
| Because java uses the concept of string literal.Suppose there are 5 reference variables,all referes to one object "sachin".If one reference variable changes the value of the object, it will be affected to all the reference variables. That is why string objects are immutable in java. |

# Working with Excel File