1. **DIFFERENCE BETWEEN INTERFACE AND ABSTRACT CLASS.**

An interface only provide contracts and it is the responsibility of implementing classes to implement each and every single contract.OOP’s concept, Abstraction is a process of hiding the implementation details and showing only functionality to the user. Another way, it shows only essential things to the user and hides the internal details.

There are two ways to achieve abstraction in java

* Abstract class (0 to 100%)
* Interface (100%)

|  |  |
| --- | --- |
| **ABSTRACT CLASS** | **INTERFACE** |
| Abstract class can have abstract and non-abstract methods. Abstract class can have public, protected, private and default modifier methods. | Interface can have only abstract methods and by default all the methods are public. But since Java 8, it can have default and static methods also. |
| An abstract class may contain non-final and different flavours of variables. | Variables declared in a Java interface is by default public, static & final. |
| Abstract class can have constructor | Interfaces can’t have constructors |
| Abstract class can provide the implementation of interface. | Interface can't provide the implementation of abstract class. |
| Any class which want to use abstract class can extend abstract class using keyword extends. | for implementing interfaces keyword used is implements. |
| A class can extend only one class. | A class can implement any number of interfaces. This property is often referred as multiple inheritance in java. So multiple inheritance can be achieved from interface only. |

**public** **interface** testInterface {

**int** ***a***=9;

**static** **void** hello()

{

System.***out***.println("Hello, New Static Method Here");

}

**default** **void** hiInterface() {

System.***out***.println("Interface default method!");

}

**void** overrideMethod(String str);

}

**public** **abstract** **class** TestDemo **extends** demo **implements** testInterface{

@Override

**public** **void** overrideMethod(String str) {

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

}

**public** **abstract** **void** abstractMethod();

}

**class** TestDemo2 **extends** TestDemo{

@Override

**public** **void** abstractMethod() {

System.***out***.println("Override method:Abstract class");

}

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

TestDemo2 tc = **new** TestDemo2();

tc.overrideMethod("");

tc.hiInterface();

testInterface.*hello*(); tc.abstractMethod(); }}

1. **WHEN TO USE ABSTRACT CLASS AND INTERFACE IN JAVA WITH REAL TIME EXAMPLE?**

<https://howtodoinjava.com/oops/exploring-interfaces-and-abstract-classes-in-java/>

For Interface:

Interface is used when you don't know anything about implementation but know the contract that implementer should have to do in order to accomplish/complete the task. Interfaces form a contract between the class and the outside world. You want to take advantage of multiple inheritances.

An interface only provide contracts and it is the responsibility of implementing classes to implement each and every single contract provided to it.

An interface is the best fit for cases where you want to define only the characteristics of class, and you want to force all implementing entities to implement those characteristics.

For Abstract class:

Abstract class is used when you know partial implementation, where say out of 5 methods, you know implementation of 3 methods and don't know implementation of 2 methods in that case 2 methods will be abstract and you need to rely on implementer as a contract to must provide body of abstract methods to accomplish the task.

1. **WHAT IS SUPER KEYWORD?**

<https://www.geeksforgeeks.org/super-keyword/>

The**super** keyword in java is a reference variable that is used to refer parent class objects.  The keyword “super” came into the picture with the concept of Inheritance.

* **Use of super with variables:** This scenario occurs when we declare a variable within derived class with the same name of base class – this concept is known as *Variable Overriding*. We can access the base class variable within derived class using super keyword.
* **Use of super with methods:** If we declare a method within derived class with the same signature (method name same, return type same, argument/parameter list same) of base class – this concept is known as *Method Overriding*. We can access base class method using super keyword.
* **Use of super with constructors:** super keyword can also be used to access the parent class constructor. Only derived class constructor can access the base class constructor. Using super we can initialize the base class constructor. Super() must be the first statement in derived class constructor.

1. **WHAT IS THIS KEYWORD?**

* this can be used to refer current class instance variable.

**class** Student{

**int** rollno;

String name;

**float** fee;

Student(**int** rollno,String name,**float** fee){

**this**.rollno=rollno;

**this**.name=name;

**this**.fee=fee;

}  }

* this can be used to invoke current class method (implicitly)

**class** A{

**void** m(){System.out.println("hello m");}

**void** n(){

System.out.println("hello n");

//m();//same as this.m()

**this**.m();

}

* this() can be used to invoke current class constructor.

**class** A{

A(){

**this**(5);

System.out.println("hello a");

}

A(**int** x){

System.out.println(x);

}  }

* this can be passed as an argument in the method call.

**class** S2{

**void** m(S2 obj){

  System.out.println("method is invoked");

  }

**void** p(){

  m(**this**);

  }

* this can be passed as argument in the constructor call.
* this can be used to return the current class instance from the method.

**class** A{

A getA(){

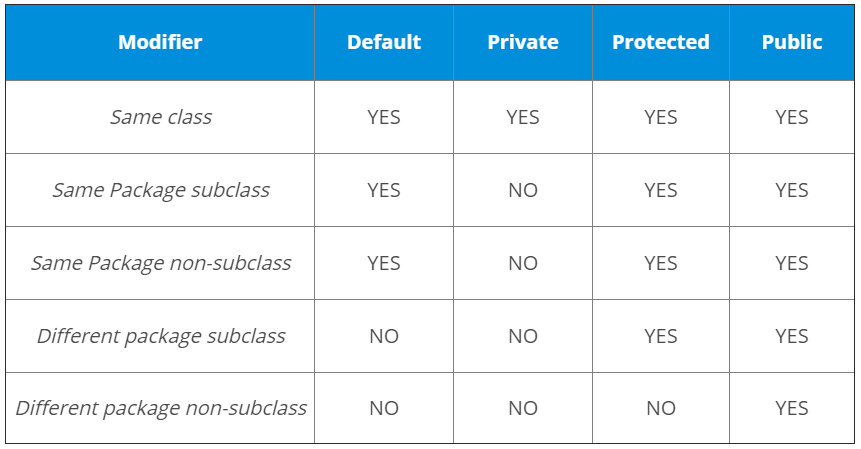
**return** **this**;

}

1. **WHAT IS THE USE OF FINAL KEYWORD?**

* If we declare a class as a final that can’t be inherited. So, using final we can prevent inheritance.
* If we declare a method as a final, that method available in derived class but we can’t override it. Here we can prevent method overriding.
* If we declare a variable as final then it is available in derived class but can’t modify or change it’s value.

1. **WHAT ARE DIFFERENT ACCESS MODIFIERS IN JAVA?**



1. **DIFFERENCE BETWEEN ARRAYLIST AND LINKEDLIST IN JAVA.**

ArrayList and LinkedList both implements List interface and maintains insertion order. Both are non-synchronized classes.

|  |  |
| --- | --- |
| **ARRAYLIST** | **LINKEDLIST** |
| ArrayList internally uses a dynamic array to store the elements. | LinkedList internally uses a doubly linked list to store the elements. |
| Manipulation with ArrayList is slow because it internally uses an array. If any element is removed/added from the array, all the bits are shifted in memory. | Manipulation with LinkedList is faster than ArrayList because it uses a doubly linked list, so no bit shifting is required in memory. |
| ArrayList is better for storing and accessing data. | LinkedList is better for manipulating data. |
| ArrayList maintains indexes and element data hence the memory consumption is low in ArrayList comparatively. | LinkedList maintains element data and two pointers for neighbour nodes hence the memory consumption is high in LinkedList comparatively. |

1. **WILL A FINALLY BLOCK EXECUTE AFTER A RETURN STATEMENT IN A METHOD IN JAVA?**

Yes, the finally block will be executed even after a return statement in a method.

The finally block will always execute even an exception occurred or not in Java. If we call the System.exit() method explicitly in the finally block then only it will not be executed. There are few situations where the finally will not be executed like JVM crash, power failure, software crash and etc. Other than these conditions, the finally block will be always executed.

1. **DIFFERENCE BETWEEN METHOD OVERLOADING AND METHOD OVERRIDING IN JAVA**

|  |  |
| --- | --- |
| **METHOD OVERLOADING** | **METHOD OVERRIDING** |
| Method overloading is a compile time polymorphism. | Method overriding is a run time polymorphism. |
| It is occurred within the class. | It is performed within two classes with inheritance relationship. |
| In this, methods must have same name and different signature. | In this, methods must have same name and same signature. |
| Return type can or can’t be same, but we must have to change the parameter. | Return type must be same. |
| 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. |
| class MethodOverloadingEx{  static int add(int a, int b){return a+b;}  static int add(int a, int b, int c){return a+b+c;}    public static void main(String args[]) {  System.out.println(add(4, 6)); ->10  System.out.println(add(4, 6, 7)); ->17  }  } | class Animal {  void eat(){System.out.println("Animal Class");}  }  class Dog extends Animal{  void eat(){System.out.println("Dog class");}  }  class MethodOverridingEx{  public static void main(String args[]) {  Dog d1=new Dog();  Animal a1=new Animal();  d1.eat(); -> Dog class  a1.eat();}} -> Animal class |

**What happens when method signature is the same and the return type is different?**

The compiler will give an error as the return value alone is not sufficient for the compiler to figure out which function it has to call.

Here, we can see that a method eat() has overridden in the derived class name Dog that is already provided by the base class name Animal.

When we create the instance of class Dog and call the eat() method, we see that only derived class eat() method run instead of base class method eat() and When we create the instance of class Animal and call the eat() method, we see that only base class eat() method run instead of derived class method eat().

So, it’s clear that in method overriding, method is bound to the instances on the run time which is decided by the JVM. That’s why it is called Run time polymorphism

1. **SLF4J VS LOG4J - WHICH ONE IS BETTER?**

SLF4J and Log4J focus on different areas and they are not similar components. SLF4J is a logging facade. The name says it all. SLF4J is Simple Logging Facade for Java. It is not a logging component and it does not do the actual logging. It is only an abstraction layer to an underlying logging component. In the case of Log4j, it is a logging component and it does the logging instructed to do.

Simple Logging Facade for Java (SLF4J) is an API designed to give generic access to many logging frameworks; log4j being one of them.

For example, below code you may write in your application class files:

*import org.slf4j.Logger;*

*import org.slf4j.LoggerFactory;*

*public class HelloWorld*

*{*

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

*{*

*Logger logger = LoggerFactory.getLogger(HelloWorld.class);*

*logger.info("Hello World");*

*}*

*}*

Now all you have to choose, which logging framework you need to use in runtime. For that, you will have to include two jar files:

* SLF4j binding jar file
* Desired logging framework jar files

e.g. to use log4j in your project, you will have to include below given jar files:

* slf4j-log4j12-1.7.12.jar
* log4j-1.2.17.jar

Once you have places both jar files in your application classpath, SLF4j will automatically detect it and start using log4j for processing the log statements based on configuration you provided in log4j configuration file.

In future, if you want to replace log4j with any other logging framework – All you have to do is replace the binding and logging jar files (along with configuration file). It’s easy. No need to change the actual source code files.

So essentially, SLF4J does not replace log4j, they work together. It removes the dependency on log4j from your application and make it easy to replace it in future with more capable library.

1. **JAVA SINGLETON PATTERN**

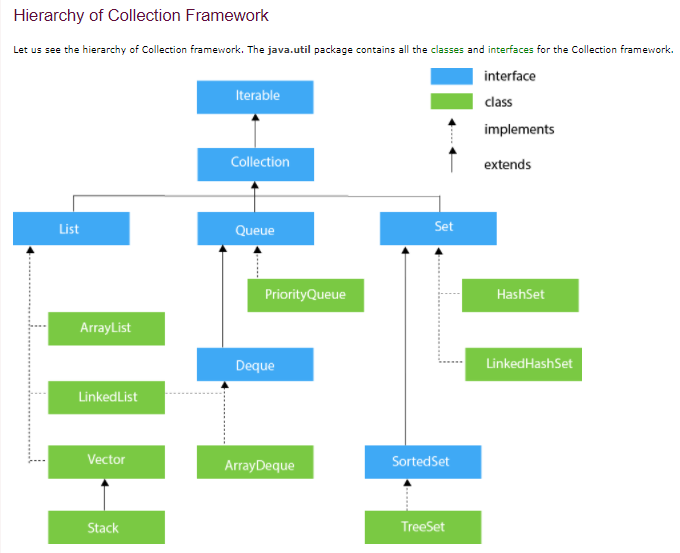
public class PropertiesUtil {  
 private static PropertiesUtil *propertiesUtilInstance* = null;  
 private Properties propApp = new Properties();  
 private PropertiesUtil() throws CustomException {  
 this.setProperties();  
 }  
  
 private void setProperties() throws CustomException {  
 try {  
 ClassLoader classLoader = PropertiesUtil.class.getClassLoader();  
 InputStream in = classLoader.getResourceAsStream("application.properties");  
 propApp.load(in);  
 in.close();  
 } catch (IOException e) {  
 throw new CustomException("");  
 }  
 }  
  
 public static synchronized PropertiesUtil getPropertiesUtil() throws CustomException {  
 if (*propertiesUtilInstance*==null){  
 synchronized (PropertiesUtil.class){  
 if (*propertiesUtilInstance*==null) {  
 *propertiesUtilInstance* = new PropertiesUtil();  
 }  
 }  
 }  
 return *propertiesUtilInstance*;  
 }  
 public String getApplication(String key){  
 return propApp.getProperty(key);  
 }  
}

1. **JAVA COLLECTION FRAMEWORK**

The Collection in Java is a framework that provides an architecture to store and manipulate the group of objects.

Java Collections can achieve all the operations that you perform on a data such as searching, sorting, insertion, manipulation, and deletion.

Java Collection framework provides many interfaces (Set, List, Queue, Deque) and classes (ArrayList, Vector, LinkedList, PriorityQueue, HashSet, LinkedHashSet, TreeSet).



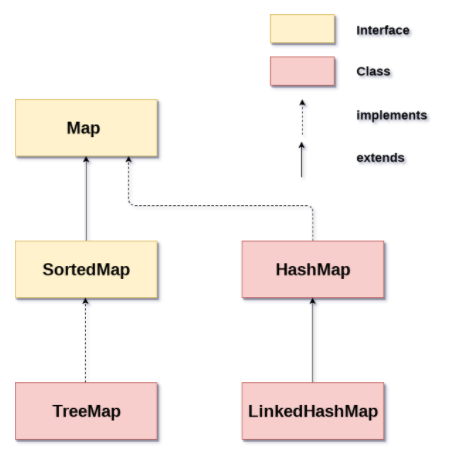
**Java Map Interface**

A map contains values on the basis of key, i.e. key and value pair. Each key and value pair is known as an entry. A Map contains unique keys.

A Map is useful if you have to search, update or delete elements on the basis of a key.

**Java Map Hierarchy**

There are two interfaces for implementing Map in java: Map and SortedMap, and three classes: HashMap, LinkedHashMap, and TreeMap. The hierarchy of Java Map is given below:



|  |  |
| --- | --- |
| [HashMap](https://www.javatpoint.com/java-hashmap) | HashMap is the implementation of Map, but it doesn't maintain any order. |
| [LinkedHashMap](https://www.javatpoint.com/java-linkedhashmap) | LinkedHashMap is the implementation of Map. It inherits HashMap class. It maintains insertion order. |
| [TreeMap](https://www.javatpoint.com/java-treemap) | TreeMap is the implementation of Map and SortedMap. It maintains ascending order. |

1. **DIFFERENCE BETWEEN SET AND LIST?**

The most noticeable differences are :

1. Set is unordered collection where List is ordered collection based on zero based index.
2. List allow duplicate elements but Set does not allow duplicates.
3. List does not prevent inserting null elements (as many you like), but Set will allow only one null element.
4. **DIFFERENCE BETWEEN LIST AND MAP?**

List is collection of elements where as map is collection of key-value pairs. They have separate top level interface.

public interface List<E> extends Collection<E> {

public interface Map<K,V> {

1. **STATIC KEYWORD IN JAVA:**

In Java, if a field is declared static, then exactly a single copy of that field is created and shared among all instances of that class. It doesn't matter how many times we initialize a class; there will always be only one copy of static field belonging to it. The value of this static field will be shared across all object of either same or any different class.

From the memory perspective, static variables go in a particular pool in JVM memory called Metaspace.

<https://www.baeldung.com/java-static#:~:text=In%20the%20Java%20programming%20language,all%20instances%20of%20the%20class>

1. **OOPS CONCEPT IN JAVA?**

* **ABSTRUCTION**: Abstraction is the concept of object-oriented programming that shows only essential information and hides unnecessary information from the users. This is also called as implementation hiding/ detail hiding.

There are two ways to achieve abstraction in java

* Abstract class (0 to 100%)
* Interface (Achieve 100% abstraction)
* **ENCAPSULATION:** Encapsulation is one of the fundamental concepts in object-oriented programming (OOP). It the concept of bundling data and methods together within one unit (that work on that data). This concept is also called information hiding/ data hiding. It is like a protective shield that prevents the data from being accessed by the code outside this shield.

Encapsulation can be achieved in java by declaring all the variables in a class as private and writing public methods in that class to set and get the values of private variables.

**ENCAPSULATION VS ABSTRACTION**

|  |  |
| --- | --- |
| **ABSTRACTION** | **ENCAPSULATION** |
| Abstraction is detail hiding(implementation hiding). | Encapsulation is data hiding(information hiding) |
| Abstraction shows only essential information and hides unnecessary information from the users | Encapsulation is bundling data and methods together within one unit. |
| There are two ways to achieve abstraction in java -  Abstract class (0 to 100%)  Interface (100%) | Encapsulation can be achieved in java by declaring all the variables in a class as private and writing public methods in that class to set and get the values of private variables. |

* **POLIMORPHISM**: Polymorphism is an important concept of object-oriented programming. It simply means more than one form. That is, the same method can perform different operations in different scenarios.

In Java polymorphism is mainly divided into two types:

* **Compile time Polymorphism:** Object-oriented programming languages, allows us to implement multiple methods within the same class that use the same name but a different set of parameters. That is called method overloading and represents a static form of polymorphism/compile time polymorphism. Due to the different sets of parameters, each method has a different signature. That allows the compiler to identify which method has to be called and to bind it to the method call.
* **Runtime Polymorphism:** Within an inheritance hierarchy, a subclass can override a method of its superclass. It also creates a form of polymorphism. Both methods, implemented by the super and subclass, share the same name and parameters but provide different functionality. If you instantiate the subclass, the JVM will always call the overridden method, even if you cast the subclass to its superclass. That is called dynamic polymorphism. This form of polymorphism doesn’t allow the compiler to determine the executed method. The JVM needs to do that at runtime.
* **INHERITANCE:** Inheritance is a mechanism in OOPs where you can derive a class from another class. In Java, each class can only be derived from one other class. That existing class is called a superclass or parent class. The derived class is called subclass or child class.

We use the keyword extends to identify the class that your subclass extends. If you don’t declare a superclass, your class implicitly extends the class Object. Object is the root of all inheritance hierarchies; it’s the only class in Java that doesn’t extend another class.

1. **HOW DO YOU HANDLE THE EXCEPTIONS IN YOUR PROJECT?**
2. **CAN AN INTERFACE EXTEND MULTIPLE INTERFACES IN JAVA?**

Yes, we can do it. An interface can extend multiple interfaces in Java.

interface A {

   public void test();

   public void test1();

}

interface B {

   public void test();

   public void test2();

}

interface C extends A,B {

   public void test3();

}

1. **MULTIPLE INHERITANCE BY INTERFACE IN JAVA**

Java does not support multiple inheritances but we can achieve the effect of multiple inheritances using interfaces. Multiple inheritance by interface occurs if a class implements multiple interfaces or also if an interface itself extends multiple interfaces.