**OBJECT ORIENTED PROGRAMMING**

* WHAT IS THE MAIN DIFFERENCE BETWEEN A CLASS AND AN OBJECT?
* A class is a template for creating objects in program whereas the object is an instance of a class
* A class is a logical entity while object is a physical entity
* A class does not allocate memory space on the other hand object allocates memory space
* You can declare class only once but you can create more than one object using a class
* Classes can't be manipulated while objects can be manipulated
* Classes doesn't have any values, whereas objects have its own value
* You can create class using "class" keyword while hand you can create object using "new" keyword in Java
* WHAT IS ENCAPSULATION? EXPLAIN WITH A USE CASE.
* Encapsulation in Java is a mechanism of wrapping the data (variables) and code acting on the data (methods) together as a single unit. In encapsulation, the variables of a class will be hidden from other classes, and can be accessed only through the methods of their current class. Therefore, it is also known as **data hiding**
* To achieve encapsulation in Java:
  + Declare the variables of a class as private
  + Provide public setter and getter methods to modify and view the variables values
  + Example:
    - class Account {
    - private int account\_number;
    - private int account\_balance;
    - public void show Data() {
    - //code to show data
    - }
    - public void deposit(int a) {
    - if (a < 0) {
    - //show error
    - } else
    - account\_balance = account\_balance + a;
    - }
    - }
    - Suppose a hacker managed to gain access to the code of your bank account. Now, he tries to deposit amount -100 into your account by two ways. Let see his first method or approach.
    - **Approach 1:**He tries to deposit an invalid amount (say -100) into your bank account by manipulating the code.
    - Now, the question is – *Is that possible?* Let investigate.
    - Usually, a variable in a class are set as "private" as shown below. It can only be accessed with the methods defined in the class. No other class or object can access them.
    - If a data member is private, it means it can only be accessed within the same class. No outside class can access private data member or variable of other class.
    - So in our case hacker cannot deposit amount -100 to your account.
    - **Approach 2**: Hacker's first approach failed to deposit the amount. Next, he tries to do deposit a amount -100 by using "deposit" method.
    - But method implementation has a check for negative values. So the second approach also fails.
    - But method implementation has a check for negative values. So the second approach also fails.
    - The entire code can be thought of a capsule, and you can only communicate through the messages. Hence the name encapsulation.
* Explain polymorphism with a used case
* Polymorphism is the ability of an object to take on many forms. The most common use of polymorphism in OOP occurs when a parent class reference is used to refer to a child class object.
* Any Java object that can pass more than one IS-A test is considered to be polymorphic. In Java, all Java objects are polymorphic since any object will pass the IS-A test for their own type and for the class Object.
* It is important to know that the only possible way to access an object is through a reference variable. A reference variable can be of only one type. Once declared, the type of a reference variable cannot be changed.
* The reference variable can be reassigned to other objects provided that it is not declared final. The type of the reference variable would determine the methods that it can invoke on the object.
* A reference variable can refer to any object of its declared type or any subtype of its declared type. A reference variable can be declared as a class or interface type.
* Use case:
  + public interface Vegetarian{}
  + public class Animal{}
  + public class Deer extends Animal implements Vegetarian{}
* Now, the Deer class is considered to be polymorphic since this has multiple inheritance. Following are true for the above examples:
  + A Deer IS-A Animal
  + A Deer IS-A Vegetarian
  + A Deer IS-A Deer
  + A Deer IS-A Object
  + When we apply the reference variable facts to a Deer object reference, the following declarations are legal –
    - Deer d = new Deer();
    - Animal a = d;
    - Vegetarian v = d;
    - Object o = d;
  + All the reference variables d, a, v, o refer to the same Deer object in the heap
* EXPLAIN OVERRIDING AND OVERLOADING WITH EXAMPLES
* Overloading occurs when two or more methods in one class have the same method name but different parameters.
* Overriding means having two methods with the same method name and parameters (i.e., method signature). One of the methods is in the parent class and the other is in the child class. Overriding allows a child class to provide a specific implementation of a method that is already provided its parent class.
* Example for overriding:
  + - **class** Dog{
    - **public** **void** bark(){
    - System.out.println("woof ");
    - }
    - }
    - **class** Hound **extends** Dog{
    - **public** **void** sniff(){
    - System.out.println("sniff ");
    - }
    - **public** **void** bark(){
    - System.out.println("bowl");
    - }
    - }
    - **public** **class** OverridingTest{
    - **public** **static** **void** main(String [] args){
    - Dog dog = **new** Hound();
    - dog.bark();
    - }
    - }
  + Output: bowl
* Example for overloading:
  + - **class** Dog{
    - **public** **void** bark(){
    - System.out.println("woof ");
    - }
    - *//overloading method*
    - **public** **void** bark(**int** num){
    - **for**(**int** i=0; i<num; i++)
    - System.out.println("woof ");
    - }
    - }
  + In this overloading example, the two bark method can be invoked by using different parameters. Compiler know they are different because they have different method signature (method name and method parameter list).
* WHAT IS INHERITANCE AND DIFFERENT TYPE OF INHERITANCE. EXPLAIN WITH A USE CASE
* Inheritance is the process of creating a new Class, called the **Derived Class** , from the existing class, called the **Base Class** . The Inheritance has many advantages, the most important of them being the reusability of code. Rather than developing **new Objects** from scratch, new code can be based on the work of other developers, adding only the new features that are needed. The reuse of **existing classes** saves time and effort.
* However, inheritance may be implemented in different combinations in **Object-Oriented Programming languages** as illustrated in figure and they include:
  + Single Inheritance
  + Multi Level Inheritance
  + Hierarchical Inheritance
  + Hybrid Inheritance
  + Multipath inheritance
  + Multiple Inheritance
* Single inheritance: When a class extends another one class only then we  call it a single inheritance.

Eg: Class A

{

public void methodA()

{

System.out.println("Base class method");

}

}

Class B extends A

{

public void methodB()

{

System.out.println("Child class method");

}

public static void main(String args[])

{

B obj = new B();

obj.methodA(); //calling super class method

obj.methodB(); //calling local method

}

}

* Multiple inheritance: **Multiple Inheritance** refers to the concept of one class extending (Or inherits) more than one base class.

Eg: interface Backend {

// abstract class

public void connectServer();

}

class Frontend {

public void responsive(String str) {

System.out.println(str + " can also be used as frontend.");

}

}

// Language extends Frontend class

// Language implements Backend interface

class Language extends Frontend implements Backend {

String language = "Java";

// implement method of interface

public void connectServer() {

System.out.println(language + " can be used as backend language.");

}

public static void main(String[] args) {

// create object of Language class

Language java = new Language();

java.connectServer();

// call the inherited method of Frontend class

java.responsive(java.language);

}

}

* Multilevel inheritance: **Multilevel inheritance** refers to a mechanism in OO technology where one can inherit from a derived class, thereby making this derived class the base class for the new class.

Eg: Class X

{

public void methodX()

{

System.out.println("Class X method");

}

}

Class Y extends X

{

public void methodY()

{

System.out.println("class Y method");

}

}

Class Z extends Y

{

public void methodZ()

{

System.out.println("class Z method");

}

public static void main(String args[])

{

Z obj = new Z();

obj.methodX(); //calling grand parent class method

obj.methodY(); //calling parent class method

obj.methodZ(); //calling local method

}

}

* Hierarchial inheritance: In such kind of inheritance one class is inherited by many**sub classes**.

Eg: class A

{

public void methodA()

{

System.out.println("method of Class A");

}

}

class B extends A

{

public void methodB()

{

System.out.println("method of Class B");

}

}

class C extends A

{

public void methodC()

{

System.out.println("method of Class C");

}

}

class D extends A

{

public void methodD()

{

System.out.println("method of Class D");

}

}

class JavaExample

{

public static void main(String args[])

{

B obj1 = new B();

C obj2 = new C();

D obj3 = new D();

//All classes can access the method of class A

obj1.methodA();

obj2.methodA();

obj3.methodA();

}

}

Output: method of Class A

Output:method of Class A

Output: method of Class A

* Hybrid inheritance:  Hybrid inheritance is a combination of**Single** and **Multiple inheritance.**

**Eg:** class C

{

public void disp()

{

System.out.println("C");

}

}

class A extends C

{

public void disp()

{

System.out.println("A");

}

}

class B extends C

{

public void disp()

{

System.out.println("B");

}

}

class D extends A

{

public void disp()

{

System.out.println("D");

}

public static void main(String args[]){

D obj = new D();

obj.disp();

}

}

Output: D

* WHAT IS AN ABSTRACT CLASS?
  + A class which is declared with the abstract keyword is known as an abstract class in [Java](https://www.javatpoint.com/java-tutorial). It can have abstract and non-abstract methods (method with the body).
  + **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, for example, sending SMS where you type the text and send the message. You don't know the internal processing about the message delivery.
* WHAT IS AN INTERFACE AND HOW MULTIPLE INHERITANCE IS ACHIEVED WITH THIS?
  + An **interface in Java** is a blueprint of a class. It has static constants and abstract methods.
  + The interface in Java is a mechanism to achieve [*abstraction*](https://www.javatpoint.com/abstract-class-in-java). There can be only abstract methods in the Java interface, not method body. It is used to achieve abstraction and multiple [inheritance in Java](https://www.javatpoint.com/inheritance-in-java).
  + In other words, you can say that interfaces can have abstract methods and variables. It cannot have a method body
  + Java Interface also **represents the IS-A relationship**
  + It cannot be instantiated just like the abstract class.
  + An interface is declared by using the interface keyword. It provides total abstraction; means all the methods in an interface are declared with the empty body, and all the fields are public, static and final by default. A class that implements an interface must implement all the methods declared in the interface.
  + multiple inheritance is not supported in the case of [class](https://www.javatpoint.com/object-and-class-in-java) because of ambiguity. However, it is supported in case of an interface because there is no ambiguity. It is because its implementation is provided by the implementation class. For example:
    - * **interface** Printable{
      * **void** print();
      * }
      * **interface** Showable{
      * **void** print();
      * }
      * **class** TestInterface3 **implements** Printable, Showable{
      * **public** **void** print(){System.out.println("Hello");}
      * **public** **static** **void** main(String args[]){
      * TestInterface3 obj = **new** TestInterface3();
      * obj.print();
      * }
      * }
      * Output: hello
* WHAT ARE THE ACCESS MODIFIERS?
  + The access modifiers in Java specifies the accessibility or scope of a field, method, constructor, or class. We can change the access level of fields, constructors, methods, and class by applying the access modifier on it.
  + There are four types of Java access modifiers:
    - Private: the access level of a private modifier is only within the class. It cannot be accessed from outside the class.
    - Default: The access level of a default modifier is only within the package. It cannot be accessed from outside the package. If you do not specify any access level, it will be the default.
    - Protected: The access level of a protected modifier is within the package and outside the package through child class. If you do not make the child class, it cannot be accessed from outside the package
    - Public: The access level of a public modifier is everywhere. It can be accessed from within the class, outside the class, within the package and outside the package.
* WHAT ARE THE VARIOUS TYPE OF CONSTRUCTORS?
* Constructor is a block of code that initializes the newly created object. A constructor resembles an instance method in java but it’s not a method as it doesn’t have a return type. In short constructor and method are different(More on this at the end of this guide). People often refer constructor as special type of method in Java.
* Constructor has same name as the class and looks like this in a java code.
  + public class MyClass{
  + //This is the constructor
  + MyClass(){
  + }
  + ..
  + }
* WHAT IS “THIS” POINTER ?
* There can be a lot of usage of **java this keyword**. In java, this is a **reference variable** that refers to the current object.
* 6 usages of java ‘THIS’ keyword:
  + this can be used to refer current class instance variable
  + this can be used to invoke current class method (implicitly
  + this() can be used to invoke current class constructor
  + this can be passed as an argument in the method call
  + this can be passed as argument in the constructor call
  + this can be used to return the current class instance from the method
* WHAT IS STATIC AND DYNAMIC BINDING?
* Connecting a method call to the method body is known as binding
* There are two types of binding
  + Static Binding (also known as Early Binding)
  + Dynamic Binding (also known as Late Binding)
* When type of the object is determined at compiled time(by the compiler), it is known as **static** **binding**. If there is any private, final or static method in a class, there is static binding
* When type of the object is determined at run-time, it is known as **dynamic binding**
* HOW MANY INSTANCES CAN BE CREATED FOR AN ABSTRACT CLASS AND WHY?
* No you can't, instead you can create instance of all other classes extending that abstract class. Because it's abstract and an object is concrete. An abstract class is sort of like a template, or an empty/partially empty structure, you have to extend it and build on it before you can use it.
* WHICH OOPS CONCEPT IS USED AS A REUSE MECHANISM AND EXPLAIN WITH A USE CASE.
  + **Inheritance** is the OOPS concept that can be used as reuse mechanism.
  + Use case:
    - In object-oriented programming, we can use inheritance when we know there is an "is a" relationship between a child and its parent class. Some examples would be
      * A person *is a* human.
      * A cat *is an* animal.
      * A car *is a*  vehicle.
    - In each case, the child or subclass is a *specialized* version of the parent or superclass. Inheriting from the superclass is an example of code reuse. To better understand this relationship, take a moment to study the Car class, which inherits from Vehicle:
      * **class** **Vehicle** {
      * **String** brand;
      * **String** color;
      * **double** weight;
      * **double** speed;
      * **void** move() {
      * **System**.**out**.println("The vehicle is moving");
      * }
      * }
      * **public** **class** **Car** **extends** **Vehicle** {
      * **String** licensePlateNumber;
      * **String** owner;
      * **String** bodyStyle;
      * **public** **static** **void** main(**String**... inheritanceExample) {
      * **System**.**out**.println(**new** **Vehicle**().brand);
      * **System**.**out**.println(**new** **Car**().brand);
      * **new** **Car**().move();
      * }
      * }
* IDENTIFY ONE PRACTICAL SCENARIO FOR EACH PILLAR OF OOPS CONCEPT.
  + The four pillars for OOP are Abstraction, Encapsulation, Inheritance, Polymorphism
  + Abstraction : Abstraction is the process of showing only essential/necessary features of an entity/object to the outside world and hide the other irrelevant information. For example to open your TV we only have a power button, It is not required to understand how infra-red waves are getting generated in TV remote control
  + Encapsulation : Encapsulation means wrapping up data and member function (Method) together into a single unit i.e. class. Encapsulation automatically achieve the concept of data hiding providing security to data by making the variable as private and expose the property to access the private data which would be public
  + Inheritance : The ability of creating a new class from an existing class. Inheritance is when an object acquires the property of another object. Inheritance allows a class (subclass) to acquire the properties and behavior of another class (super-class). It helps to reuse, customize and enhance the existing code. So it helps to write a code accurately and reduce the development tim
  + Polymorphism: Polymorphism is derived from 2 Greek words: poly and morphs. The word "poly" means many and "morphs" means forms. So polymorphism means "many forms". A subclass can define its own unique behavior and still share the same functionalities or behavior of its parent/base class. A subclass can have their own behavior and share some of its behavior from its parent class not the other way around. A parent class cannot have the behavior of its subclass

[**Next →**](https://www.javatpoint.com/interface-in-java)[**← Prev**](https://www.javatpoint.com/final-keyword)

UNIT TESTING AND JUNIT

* WHAT IS UNIT TESTING?
  + Unit testing means testing the smaller units of your application, like classes and methods
  + A unit test is a way of testing a unit - the smallest piece of code that can be logically isolated in a system. In most programming languages, that is a function, a subroutine, a method or property.
* WHAT IS THE DIFFERENCE BETWEEN MANUAL TESTING AND AUTOMATED TESTING
  + The biggest **difference between manual and automation testing** is who executes the test case. In manual testing, the human tester does it. In automation testing, the tool does it
  + Manual testing is the process in which QA analysts execute tests one-by-one in an individual manner. The purpose of manual testing is to catch bugs and feature issues before a software application goes live
  + Both have their strengths and weaknesses. Manual testing is slow and tedious. But its strength is that it better handles complex scenarios. Automated testing requires coding and test maintenance. But on the plus side, it is much faster and covers many more permutations
  + Automation testing is the process in which testers utilize tools and scripts to automate testing efforts.
  + Automation testing helps testers execute more test cases and improve test coverage. When comparing manual vs. automation testing, manual takes longer. Automated testing is more efficient.
* IS IT NECESSARY TO WRITE THE TEST CASE FOR EVERY LOGIC? IF YES, WHY?
  + No, we should write the test case only for that logic that can be reasonably broken.
* WHAT ARE THE FEATURES OF JUNIT?
  + Opensource
  + Annotation support for test cases
  + Assertion support for checking the expected result
  + Assertion support for checking the expected result
* WHAT ARE THE IMPORTANT JUNIT ANNOTATIONS? AND ITS USAGE IN CODING?
  + The test runner is used to execute the test cases.
* @Test : JUnit provides an annotation called **@Test,** which tells the JUnit that the public void method in which it is used can run as a test case
  + @BeforeClass : **@Before** annotation is used on a method containing[Java](https://www.guru99.com/java-tutorial.html)code to run before each test case. i.e it runs before each test execution
  + @Before : **@Before** annotation is used on a method containing[Java](https://www.guru99.com/java-tutorial.html)code to run before each test case. i.e it runs before each test execution
  + @After: **@After** annotation is used on a method containing java code to run after each test case. These methods will run even if any exceptions are thrown in the test case or in the case of assertion failures.
  + @AfterClass : This annotation can be used if you want to execute some statements after all test cases for e.g. Releasing resources after executing all test cases.
* WHAT DOES ASSERT CLASS?
  + Assert is a method useful in determining Pass or Fail status of a test case, The assert methods are provided by the class org.junit.Assert which extends java.lang.Object class
  + There are various types of assertions like Boolean, Null, Identical etc
  + Junit provides a class named Assert, which provides a bunch of assertion methods useful in writing test cases and to detect test failure
* WHAT IS CODE COVERAGE?
  + Code coverage means measuring how much of your code is executed during your unit tests. Basically, that means that after running your unit tests, you get a report showing you how many percent of the code that was executed during the tests, and also what lines precisely that were executed
  + With the code coverage report in hand, you can analyse what parts of your code that was **not** run, and modify your unit tests so these parts of the code **are** executed.
  + The larger the coverage, the closer to 100%, that is, the better chances you have of having bug free code
* WHAT ARE THE BEST PRACTICES TO PERFORM UNIT TESTING?
  + **Unit tests should be trustworthy**: The test must fail if the code is broken and only if the code is broken. If it doesn't, we cannot trust what the test results are telling us
  + **Unit testing should be maintainable and readable**: When production code changes, tests often need to be updated, and possibly debugged as well. So it must be easy to read and understand the test, not only for whoever wrote it, but for other developers as well. Always organize and name your tests for clarity and readability.
  + **Unit tests should verify a single-use case**: Good tests validate one thing and one thing only, which means that typically, they validate a single use-case. Tests that follow this best practice are simpler and more understandable, and that is good for maintainability and debugging. Tests that validate more than one thing can easily become complex and time-consuming to maintain. Don't let this happen.
  + **Unit tests should be isolated**: Tests should be runnable on any machine, in any order, without affecting each other. If possible, tests should have no dependencies on environmental factors or global/external state. Tests that have these dependencies are harder to run and usually unstable, making them harder to debug and fix, and end up costing more time than they save
  + **Unit tests should be automated**: Make sure tests are being run in an automated process. This can be daily, or every hour, or in a Continuous Integration or Delivery process. The reports need to be accessible to and reviewed by everyone on the team. As a team, talk about which metrics you care about: code coverage, modified code coverage, number of tests being run, performance, etc
  + **Use a good mixture of unit and integration tests**: testing pyramid model is a commonly-used model to describe the ideal distribution of testing resources. The idea is that as you go up in the pyramid, tests are usually more complex to build, more fragile, slower to run, and slower to debug. Lower levels are more isolated and more integrated, faster, and simpler to build and debug. Therefore, automated unit tests should make up the bulk of your tests.
  + **Unit tests should be executed within an organized test practise**: In order to drive the success of your testing at all levels, and make the unit testing process scalable and sustainable, you will need some additional practices in place. First of all, this means **writing unit tests as you write your application code**. Some organizations write the tests before the application code ([test-driven](https://dzone.com/articles/why-developers-dont-use-tdd) or [behavior-driven](https://dzone.com/articles/tddbdd-an-introduction-amp-usage-guide" \t "_blank) programming). The important thing is that tests go hand-in-hand with the application code. The tests and application code should even be reviewed together in the code review process. Reviews help you understand the code being written (because they can see the expected behavior) and improve tests too!
* WHAT IS MOCKING?
  + Mocking is a technique of unit testing a class, where we mock an external dependency in order to test our classes and methods. When unit tests are written well with mocks, they would not have any external dependencies and will not fail when external stuff changes.

.