edureka!

JAVA OOP CHEAT SHEET

Learn JAVA from experts at https://www.edureka.co

Object Oriented Programming in Java

Java is an Object Oriented Programming language that produces software for multiple platforms. An object-based application in Java is concerned with declaring classes, creating objects from them and interacting between these objects.

Java Class

```
class Test {
       // class body
       member variables
       methods
 }
```

Java Object

```
//Declaring and Initializing an object
Test t = new Test();
```

Constructors

Default Constructor

```
class Test{
 /* Added by the Java Compiler at the Run Time
 public Test(){
 }
 */
 public static void main(String args[]) {
   Test testObj = new Test();
  }
}
```

Parameterized Constructor

```
public class Test {
   int appId;
   String appName;
//parameterized constructor with two parameters
Test(int id, String name){
   this.appId = id;
   this.appName = name;
}
void info(){
   System.out.println("Id: "+appId+" Name: "+appName);
}

public static void main(String args[]){
   Test obj1 = new Test(11001,"Facebook");
   Test obj2 = new Test(23003,"Instagram");
   obj1.info();
   obj2.info();
 }
}
```

JAVA CERTIFICATION TRAINING

Inheritance

Single Inheritance

```
Class A {
  //your parent class code
}
Class B extends A {
   //your child class code
}
```

Multi Level Inheritance

```
Class A {
  //your parent class code
}
Class B extends A {
   //your code
}
Class C extends B {
    //your code
}
```

Hybrid Inheritance

```
      A                          A
     / \                         |
    B   C          (OR)         B
   /     \                     / \
  D       E                   C   D
```

Hierarchical Inheritance

```
Class A {
  //your parent class code
}

Class B extends A {
   //your child class code
}

Class C extends A {
    //your child class code
}
```

Multiple Inheritance

```
Class A {
  //your parent class code
}
Class B {
   //your parent class code
}
Class C extends A,B {
    //your child class code
}
```

Polymorphism

Compile Time Polymorphism

```
class Calculator {
  static int add(int a, int b){
  return a+b;
 }
static double add( double a, double b){
  return a+b;
 }
public static void main(String args[]){
  System.out.println(Calculator.add(123,17));
  System.out.println(Calculator.add(18.3,1.9));
 }
}
```

Run Time Polymorphism

```
public class Mobile{
void sms(){System.out.println("Mobile class");}
}

//Extending the Mobile class
public class OnePlus extends Mobile{
//Overriding sms() of Mobile class
 void sms(){
  System.out.println(" OnePlus class");
 }

public static void main(String[] args) {
  OnePlus smsObj= new OnePlus();
  smsObj.sms();
 }
}
```

Abstraction

Abstract Class

```
public abstract class MyAbstractClass
 {
    public abstract void abstractMethod();
    public void display(){
     System.out.println("Concrete method");
  }
 }
```

Interface

```
//Creating an Interface
public interface Bike { public void start(); }
//Creating classes to implement Bike interface
class Honda implements Bike{
  public void start() {
   System.out.println("Honda Bike");
} }
class Apache implements Bike{
  public void start() {
    System.out.println("Apache Bike");
} }
class Rider{
  public static void main(String args[]){
   Bike b1=new Honda();
   b1.start();
   Bike b2=new Apache();
   b2.start();
} }
```

Encapsulation

```
public class Artist {
   private String name;
   //getter method
   public String getName() { return name; }
   //setter method
   public void setName(String name) { this.name = name; }
  }
public class Show{
  public static void main(String[] args){
   //creating instance of the encapsulated class
   Artist s=new Artist();
   //setting value in the name member
   s.setName("BTS");
   //getting value of the name member
   System.out.println(s.getName());
 }
}
```

Modifiers in Java

Access Modifiers

Scope	Private	Default	Protected	Public
Same class	Yes	Yes	Yes	Yes
Same package subclass	No	Yes	Yes	Yes
Same package non-subclass	No	Yes	Yes	Yes
Different package subclass	No	No	Yes	Yes
Different package non-subclass	No	No	No	Yes

Non - Access Modifiers

Type	Scope
Static	Makes the attribute dependent on a class
Final	Once defined, doesn't allow any changes
Abstract	Makes the classes and methods abstract
Synchronized	Used to synchronize the threads

Classes & Objects

Java Classes

A **class** in Java is a blueprint which includes all your data. A class contains fields (**variables**) and methods to describe the behavior of an object.

```
class Test {
       member variables // class body
       methods
}
```

Java Objects

An **object** is a major element in a class which has a state and behavior. It is an instance of a class which can access your data. The 'new' keyword is used to create the *object*.

```
//Declaring and Initializing an object
 Test t = new Test();
```

Java Constructors

Constructors

A constructor is a block of code that initializes a newly created object. It is similar to a method in Java but doesn't have any return type and its name is same as the class name. There are 3 types of constructors:

1. Default Constructor (No-Argument Constructor)
2. Parameterized Constructor

Parameterized Constructor

This constructor is called parameterized as it contains one or more parameters. It is used to provide different values to the distinct objects at the time of their creation.

Default Constructor

This constructor is created by default by the java compiler at the time of class creation if no other constructor is declared in the class. Sometimes its also called no-argument constructor as it doesn't contain any parameters.

```
 class Test{
 // Added by the Java Compiler at the Run
Time
 public Test(){
 }
 public static void main(String args[]) {
   Test testObj = new Test();
  }
}
```

```
public class Test {
 int appId;
 String appName;
 //parameterized constructor with two
parameters
 Test(int id, String name) {
    this.appId = id;
    this.appName = name;
}
void info() {
 System.out.println("Id: "+appId+" Name:
"+appName);
}
public static void main(String args[]){
 Test obj1 = new Test(11001,"Facebook");
 Test obj2 = new Test(23003,"Instagram");
 obj1.info();
 obj2.info();
 }
}
```

Modifiers in Java

Access Modifiers

Java access modifiers specify the scope of accessibility of a data member, method, constructor or class.

SCOPE	Private	Default	Protected	Public
Same class	Yes	Yes	Yes	Yes
Same package subclass	No	Yes	Yes	Yes
Same package non-subclass	No	Yes	Yes	Yes
Different package subclass	No	No	Yes	Yes
Different package non-subclass	No	No	No	Yes

Non Access Modifiers

The non-access modifiers in Java, do not change the accessibility of **variables** and methods rather they provide special properties. These modifiers can alter the behavior of elements as well.

Type	Scope
Static	Makes the attribute dependent on a class
Final	Once defined, doesn't allow any changes
Abstract	Makes the classes and methods abstract
Synchronized	Used to synchronize the threads

Inheritance

Types Of Inheritance in Java

Inheritance is the property of a child/derived/subclass which allows it to inherit the properties(data members) and functionalities(methods) from its parent/base/superclass.

- All objects have the Object class as their top parent.
- Methods can be overridden but attributes can not.
- To call a parent class constructor, super() is used.

Java supports 5 types of inheritance:

1. Single Inheritance
2. Multi-level Inheritance
3. Hierarchical Inheritance
4. Hybrid Inheritance
5. Multiple Inheritance

Single Inheritance

In this, a class inherits the properties of a single parent class.

Hierarchical Inheritance

In hierarchical inheritance, one parent can have one or more child/sub/derived classes.

```
Class A{
  //your parent class code
}
Class B extends A {
   //your child class code
}
Class C extends A {
    //your child class code
}
```

Hybrid Inheritance

Hybrid Inheritance is the combination of more than one type of inheritance in a single program, for example, you can combine a multilevel inheritance with a hierarchical inheritance.

```
Class A{
  //your parent class code
}
Class B extends A {
   //your child class code
}
```

Multi Level Inheritance

In multi-level inheritance, one class has more than one parent class but at different levels of inheritance.

```
Class A{
  //your parent class code
}
Class B extends A {
   //your code
}
Class C extends B {
    //your code
}
```

```
      A
      /
   B     C
   /
 D         E
```

Multiple Inheritance

Multiple inheritance is not supported in Java as it leads to the **diamond problem**. The diamond problem is an ambiguity where the compiler doesn't know which superclass method to execute in case the superclasses has a method with the same name.

```
                    A
                   /
{abc()} B         C {abc()}
                      /
                    D {?}
```

*But multiple inheritance in Java can be achieved using interfaces.

Polymorphism

Polymorphism is the ability of a variable, function or an object to take multiple forms. It allows you to define one interface or method and have multiple implementations. There are two types of polymorphism in Java.

1. Compile Time Polymorphism
2. Runtime Polymorphism

Compile Time Polymorphism

Also called static binding, as the type of the object is determined at the compile time by the compiler itself.
Example: Method Overloading

```
class Calculator {
static int add(int a, int b){
return a+b;
}
static double add( double a, double b){
return a+b;
}
public static void main(String args[]){
System.out.println(Calculator.add(123,17));
System.out.println(Calculator.add(18.3,1.9));
}
}
```

Runtime Polymorphism

Also called dynamic binding as the overridden method is resolved at runtime rather than compile-time. In this, a reference variable is used to call an overridden method of a superclass at run time. Example: Method Overriding.

```
public class Mobile{
void sms(){
System.out.println("Mobile class");
}
}
//Extending the Mobile class
public class OnePlus extends Mobile{
//Overriding sms() of Mobile class
void sms(){
System.out.println(" OnePlus class");
}
public static void main(String[] args)
{
OnePlus smsObj= new OnePlus();
smsObj.sms();
}
}
```

Abstraction

Ways To Achieve Abstraction

Abstraction is the process of hiding the details and showing only the necessary things to the user. You can achieve abstraction in two ways in Java:

1. Using Abstract Class (0-100%)
2. Using Interface (100%)

Abstract Class

Abstract Class is a class which is declared with an abstract keyword and cannot be instantiated. Few pointers to create an abstract class:

- It can contain abstract and non-abstract methods.

Interface

An interface in java is a blueprint of a class that contains static constants and abstract methods. It represents the IS-A relation. You need to implement an interface to use its methods or constants.

- It can contain constructors and static methods as well.
- It can contain final methods which force the subclass not to change the body of the method.

```
public abstract class MyAbstractClass
 {
    public abstract void abstractMethod();
    public void display(){
System.out.println("Concrete method");  }
 }
```

Encapsulation

Encapsulation is a process of binding your data and code together as a single unit using getter and setter methods.

You need to perform two steps to achieve encapsulation:

- Declare the variables of a class as private.
- Provide public setter and getter methods to modify and view the variables values.

```
public class Artist {
private String name;
//getter method
public String getName() { return name; }
//setter method
public void setName(String name) { this.name
= name; }
}
public class Show{
public static void main(String[] args){
//creating instance of the encapsulated class
Artist s=new Artist();
//setting value in the name member
s.setName("V");
//getting value of the name member
System.out.println(s.getName());
} }
```

```
//Creating an Interface
public interface Bike { public void start();
}
//Creating classes to implement Bike
interface
class Honda implements Bike{
public void start() {
System.out.println("Honda Bike"); }
}
class Apache implements Bike{
public void start() {
System.out.println("Apache Bike"); }
}
class Rider{
public static void main(String args[]){
Bike b1=new Honda();
b1.start();
Bike b2=new Apache();
b2.start();
}
}
```

Association

Association is the relation between two different classes that is established via their objects. Association can be in many forms:

- One-to-One
- One-to-Many
- Many-to-One
- Many-to-Many.

Aggregation

Aggregation is a special form of Association which represents **the Has-A** relationship. It is an uni-directional Association where both the entries can survive individually.

Composition

Composition is a more restrictive form of aggregation that makes two entities highly dependent on each other. It represents the **part-of** relationship where the composed object **cannot exist** without the other entity.

Download Core Java Cheat Sheet for Beginners Edureka

*With this, we come to an end of **Java OOP Cheat Sheet**. You can check out the **Java Training** by Edureka, a trusted online learning company with a network of more than 250,000 satisfied learners spread across the globe. Edureka's **Java J2EE and SOA Training & Certification** course is designed for students and professionals who want to be a Java Developer. The course is designed to give you a head start into Java programming and train you for both core and advanced Java concepts along with various Java frameworks like Hibernate & Spring.*

Recommended videos for you