**OOPS with JAVA**

***Chapter 1 Definition, Principles, Examples, Criticism of OOPS***

Object-Oriented Programming or Java OOPs concept refers to --using objects in programming and treating objects as a primary source to implement what is to happen in the code.

**What are the main principles of OOP?**

Object-oriented programming is based on the following principles:

* **Encapsulation.**The [encapsulation](https://www.techtarget.com/searchnetworking/definition/encapsulation) principle states that all important information is contained inside an object and only select information is exposed. The implementation and state of each object are privately held inside a defined class. Other objects do not have access to this class or the authority to make changes. They are only able to call a list of public functions or methods. This characteristic of data hiding provides greater program security and avoids unintended data corruption.
* **Abstraction.** Objects only reveal internal mechanisms that are relevant for the use of other objects, hiding any unnecessary implementation code. The derived class can have its functionality extended. This concept can help developers more easily make additional changes or additions over time.
* **Inheritance.**Classes can reuse code and properties from other classes. Relationships and subclasses between objects can be assigned, enabling developers to reuse common logic, while still maintaining a unique hierarchy. Inheritance forces more thorough data analysis, reduces development time and ensures a higher level of accuracy
* **Polymorphism.**Objects are designed to share behaviors, and they can take on more than one form. The program determines which meaning or usage is necessary for each execution of that object from a parent class, reducing the need to duplicate code. A child class is then created, which extends the functionality of the parent class. [Polymorphism](https://www.techtarget.com/whatis/definition/polymorphism) enables different types of objects to pass through the same interface.
* **Coupling.** This is the degree to which software elements are connected to one another. For example, if a class has its attributes change, then any other coupled class also changes.
* **Association.** This is the connection between one or more classes. Associations can be one to one, many to many, one to many or many to one.

*While Simula is credited as being the first object-oriented programming language, many other programming languages are used with OOP today. But some programming languages pair with OOP better than others. For example, programming languages that are considered pure OOP languages treat everything as objects.*

Other programming languages are designed primarily for OOP but with some procedural processes included. Some of the [most popular programming languages](https://www.tiobe.com/tiobe-index/) are designed for, or with, OOP in mind.

For example, popular pure OOP languages include the following:

* Ruby.
* Scala.
* JADE.
* Emerald.

Programming languages designed primarily for OOP include the following:

* [Java](https://www.theserverside.com/definition/Java).
* [Python](https://www.techtarget.com/whatis/definition/Python).
* [C++](https://www.techtarget.com/searchdatamanagement/definition/C).

Other programming languages that pair with OOP include the following:

* Visual Basic .NET.
* [PHP](https://www.techtarget.com/whatis/definition/PHP-Hypertext-Preprocessor).
* [JavaScript](https://www.theserverside.com/definition/JavaScript).

***Benefits of OOP include the following:***

* **Modularity.** Encapsulation enables objects to be self-contained, making troubleshooting and collaborative development easier.
* **Reusability.** Code can be reused through inheritance, meaning a team does not have to write the same code multiple times.
* **Productivity.** Programmers can construct new programs quickly through the use of multiple libraries and reusable code.
* **Easily upgradable and scalable.**Programmers can implement system functionalities independently.
* **Interface descriptions.** Descriptions of external systems are simple, due to message-passing techniques that are used for object communication.
* **Security.** Using encapsulation and abstraction, complex code is hidden, software maintenance is easier and internet protocols are protected.
* **Flexibility.** Polymorphism enables a single function to adapt to the class it is placed in. Different objects can also pass through the same interface.
* **Code maintenance.** Parts of a system can be updated and maintained without needing to make significant adjustments.
* **Lower cost.** Other benefits, such as its maintenance and reusability, reduce development costs.

## Criticism of OOPS

Developers have criticized the object-oriented programming model for multiple reasons. The largest concern is that OOP overemphasizes the data component of software development and does not focus enough on computation or [algorithms](https://www.techtarget.com/whatis/definition/algorithm). Additionally, OOP code may be more complicated to write and take longer to compile.

Other common criticisms include the fact that inheritance comes with drawbacks, such as fragile base classes. Additionally, objects are sometimes more clear while isolated but are harder to understand when operating in the actual program.

Alternative methods to OOP include the following:

* **Functional programming.**This includes languages such as Erlang and Scala, which are used for telecommunications and fault-tolerant systems.
* **Structured or modular programming.** This includes languages such as PHP and C#.
* **Imperative programming.** This alternative to OOP focuses on function rather than models. [Imperative programming](https://www.techtarget.com/whatis/definition/imperative-programming) languages include C++ and Java.
* **Declarative programming.**This programming method involves statements on what the task or desired outcome is but not how to achieve it. [Declarative programming](https://www.techtarget.com/searchitoperations/definition/declarative-programming) languages include Prolog and Lisp.
* **Logical programming.** This method, which is based mostly on formal logic and uses languages such as Prolog, contains a set of sentences that express facts or rules about a problem domain. It focuses on tasks that can benefit from rule-based logical queries.

Most advanced programming languages enable developers to combine models because they can be used for different programming methods. For example, JavaScript and Scala can be used for OOP and functional programming.

**Why java not purely OOP language?**

Pure Object Oriented Language or Complete Object Oriented Language are Fully Object Oriented Language that supports or have features that treats everything inside the program as objects. It doesn’t support primitive datatype(like int, char, float, bool, etc.). There are seven qualities to be satisfied for a programming language to be pure object-oriented. They are:

1. Encapsulation/Data Hiding
2. Inheritance
3. Polymorphism
4. Abstraction
5. All predefined types are objects
6. All user defined types are objects
7. All operations performed on objects must be only through methods exposed at the objects.

***Chapter 2 Class(es) and Object(s)***

A class in Java is a **set of objects** which shares common characteristics/ behavior and common properties/ attributes. It is a user-defined prototype from which objects are created.

### Properties of Java Classes

1. Class is not a real-world entity. It is just a template / blueprint /prototype from which objects are created.
2. Class does not occupy memory.
3. Class is a set of variables (of different data types) and methods.
4. A Class in Java can contain:
   * Data member(s)
   * Method(s)
   * Constructor(s)
   * Nested Classes
   * Interface(s)

***Syntax :***

*access\_modifier* **class** <*class\_name*>  
{   
 data member;   
 method;   
 constructor;  
 nested class;  
 interface;  
}

By default a class has public or default access (within or outside of package) when no access modifier is supplied.

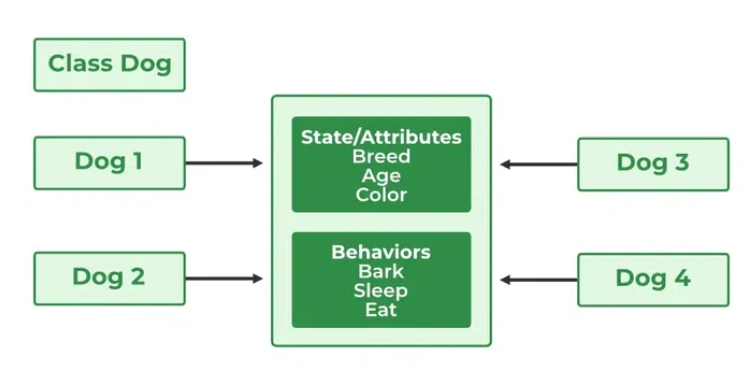
1. ***Modifiers****: A class can be public or has default access*
2. ***Class keyword:****class keyword is used to create a class.*
3. ***Class name:****The name should begin with an initial letter (capitalized by convention).*
4. ***Superclass(if any):****The name of the class’s parent (superclass), if any, preceded by the keyword extends. A class can only extend (subclass) one parent.*
5. ***Interfaces(if any):****A comma-separated list of interfaces implemented by the class, if any, preceded by the keyword implements. A class can implement more than one interface.*
6. ***Body:****The class body is surrounded by braces, { }.*

**Object(s)**

An object in Java is a building block of Object-Oriented Programming and represents real-life entities. Objects are the instances of a class that are created to use the attributes(data members/instance variables) and methods (member functions )of a class.  A typical Java program creates many objects, which as you know, interact by invoking methods. An object consists of :

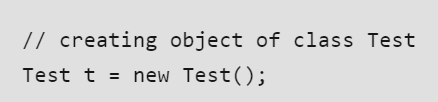
1. **State**(data) It is represented by attributes of an object. It also reflects the properties of an object.
2. **Behavior**: It is represented by the methods of an object. It also reflects the response of an object with other objects.
3. **Identity**: It gives a unique name to an object and enables one object to interact with other objects.

Example:

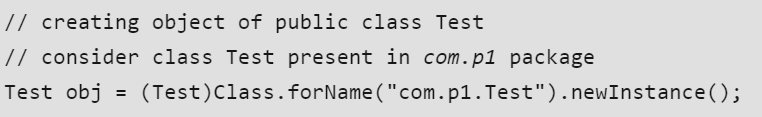


## Different Ways to Create an Object of a Class

* 1. New Keyword

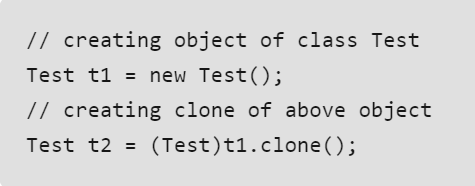


### ****Using Class.forName(String className) method****



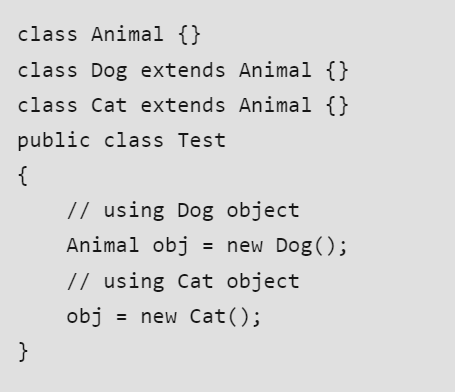
* 1. Using clone()

**clone()** method is present in the Object class. It creates and returns a copy of the object.



4. De-serialization is a technique of reading an object from the saved state in a file.

### Creating multiple objects by one type only



In the inheritance system, we use a parent class reference variable to store a sub-class object. In this case, we can switch into different subclass objects using the same referenced variable.

***Chapter 3 Access Modifiers of Java***

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

Access modifiers help to restrict the scope of a class, constructor, variable, method, or data member. It provides security, accessibility, etc to the user depending upon the access modifier used with the element .

1. **Private**: The access level of a private modifier is only within the class. It cannot be accessed from outside the class.
2. **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.
3. **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.
4. **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.

*12 Modifiers in Java are public, private, protected, default, final, synchronized, abstract, native, strictfp, transient, and volatile.*

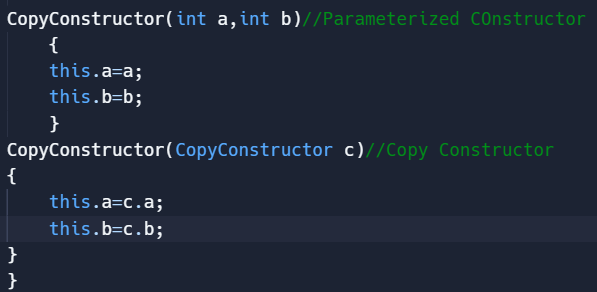
There are many non-access modifiers, such as static, abstract, synchronized, native, volatile, transient,etc .

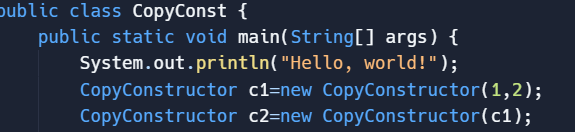
***Chapter 4 Constructors ,Type and Usage***

* A constructor is a special method used to initialize objects. The constructor is called when an instance of a class is created. It can be used to set initial values for object attributes. Constructors have the same name as the class and do not have a return type, not even void.
* At the time of calling the constructor, memory for the object is allocated in the memory.
* Every time an object is created using the new() keyword, at least one constructor is called.
* Types Default Constructor

Parameterized Constructor

Copy Constructor: Unlike other constructors copy constructor is passed with another object which copies the data available from the passed object to the newly created object.





**Access Modifiers with Constructor**

**Public Constructor:**

* **Visibility:** Accessible from any other class.
* **Usage:** When you want to allow any part of your program to create instances of the class.

**Private Constructor:**

* **Visibility:** Accessible only within the class itself.
* **Usage:** Commonly used in Singleton design patterns where only one instance of the class should exist, or to prevent instantiation from outside the class.

**Protected Constructor:**

* **Visibility:** Accessible within the same package and subclasses.
* **Usage:** Useful when you want to allow subclassing but prevent direct instantiation from outside the package.

**Default (Package-Private) Constructor:**

* **Visibility:** Accessible only within the same package.
* **Usage:** When you want to restrict instantiation to classes within the same package.

### *Chapter 5 Abstraction*

Abstraction states that Objects only reveal internal mechanisms that are relevant for the use of other objects, hiding any unnecessary implementation code. **Abstraction in Java** is the process in which we only show essential details/functionality to the user. The non-essential implementation details are not displayed to the user.

There are two ways to achieve abstraction in java

1. Abstract class (0 to 100%)
2. Interface (100%)

### Abstract class in Java

A class which is declared as abstract is known as an **abstract class**. It can have :-

* Abstract Method
* Concrete Method
* Class Variable(static fields) is a class variable
* Instance Variable is an instance variable
* Static Method
* Final Method

Points to remember-:

* An abstract class must be declared with an abstract keyword.
* It can have abstract and non-abstract methods.
* It cannot be instantiated. That is, an abstract class can not be directly instantiated with the [*new operator*](https://www.geeksforgeeks.org/new-operator-java/)
* It can have [constructors](https://www.javatpoint.com/java-constructor) and static methods also.
* It can have final methods which will force the subclass not to change the body of the method.

### *Abstract Method in Java*

A method which is declared as abstract and does not have implementation is known as an abstract method.

### Interface(is A relationship)



* In Java, an interface is a reference type, similar to a class, that can contain only constants, method signatures, default methods, static methods, and nested types. Interfaces cannot contain instance fields or constructors.
* An interface in Java is a blueprint of a behavior. A Java interface contains static constants and abstract methods. Interfaces can only have methods that are public abstract and variables that are public, static and final by default.
* They are used to specify a set of methods that a class must implement. **An interface is a way to achieve abstraction and multiple inheritance in Java.** Java Interface also **represents the IS-A relationship**.
* *Any class can extend only 1 class, but any class can implement an infinite number of interfaces.*

**So, the question arises why use interfaces when we have abstract classes?**

The reason is , abstract classes may contain non-final variables, whereas variables in the interface are final, public, and static.

***Chapter 6 Encapsulation***

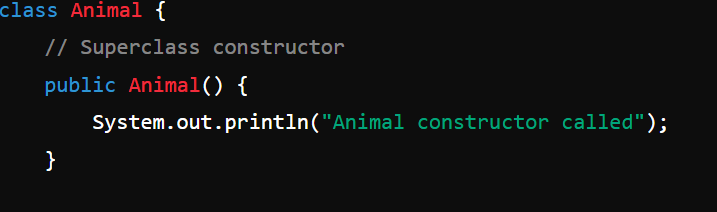
* The [encapsulation](https://www.techtarget.com/searchnetworking/definition/encapsulation) principle states that all important information is contained inside an object and only select information is exposed. The implementation and state of each object are privately held inside a defined class. Other objects do not have access to this class or the authority to make changes. They are only able to call a list of public functions or methods. This characteristic of data hiding provides greater program security and avoids unintended data corruption.
* Encapsulation is defined as the wrapping up of data under a single unit. It is the mechanism that binds together code and the data it manipulates.
* Technically in encapsulation, the variables or data of a class are hidden from any other class and can be accessed only through any member function of its class in which they are declared. As in encapsulation, the data in a class is hidden from other classes, so it is also known as data hiding.
* Encapsulation can be achieved by declaring all the variables in the class as private and writing public methods in the class to set and get the values of variables.

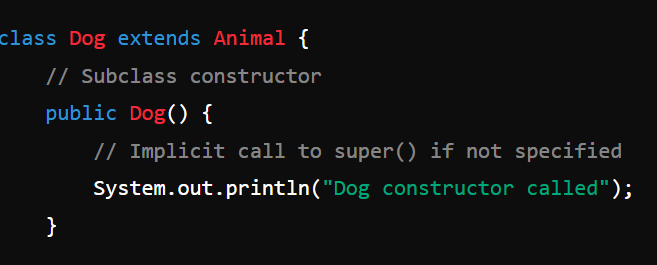
1. ***public class Person {***
2. ***// Private fields***
3. ***private String name;***
4. ***private int age;***
5. ***// Public constructor***
6. ***public Person(String name, int age) {***
7. ***this.name = name;***
8. ***this.age = age;***
9. ***}***
10. ***// Public getter for name***
11. ***public String getName() {***
12. ***return name;***
13. ***}***
14. ***// Public setter for name***
15. ***public void setName(String name) {***
16. ***this.name = name;***
17. ***}***
18. ***// Public getter for age***
19. ***public int getAge() {***
20. ***return age;***
21. ***}***
22. ***// Public setter for age***
23. ***public void setAge(int age) {***
24. ***if (age > 0) { // Simple validation***
    * ***this.age = age;***
25. ***}***
26. ***}***
27. ***// Method to display person's details***
28. ***public void display() {***
29. ***System.out.println("Name: " + name + ", Age: " + age);***
30. ***}***
31. ***}***
32. ***public class Main {***
33. ***public static void main(String[] args) {***
34. ***// Create a new Person object***
35. ***Person person = new Person("Alice", 30);***
36. ***// Access and modify the person's data through getters and setters***
37. ***person.display(); // Output: Name: Alice, Age: 30***
38. ***person.setName("Bob");***
39. ***person.setAge(35);***
40. ***person.display(); // Output: Name: Bob, Age: 35***
41. ***// Attempt to set an invalid age***
42. ***person.setAge(-5); // No effect due to validation in setter***
43. ***person.display(); // Output: Name: Bob, Age: 35***
44. ***}***
45. ***}***
46. ***// Method to display person's details***
47. ***public void display() {***
48. ***System.out.println("Name: " + name + ", Age: " + age);***
49. ***}***
50. ***}***
51. ***public class Main {***
52. ***public static void main(String[] args) {***
53. ***// Create a new Person object***
54. ***Person person = new Person("Alice", 30);***
55. ***// Access and modify the person's data through getters and setters***
56. ***person.display(); // Output: Name: Alice, Age: 30***
57. ***person.setName("Bob");***
58. ***person.setAge(35);***
59. ***person.display(); // Output: Name: Bob, Age: 35***
60. ***// Attempt to set an invalid age***
61. ***person.setAge(-5); // No effect due to validation in setter***
62. ***person.display(); // Output: Name: Bob, Age: 35***
63. ***}***
64. ***}***

***Chapter 7 Inheritance***

***Definition and Constructor chaining***

Constructor chaining in inheritance is a process where a constructor calls another constructor in the same class (using this()) or in the parent class (using super()). This is useful to ensure that the parent class is properly initialized before the subclass's constructor runs, allowing for the construction of the object to proceed from the top of the inheritance hierarchy down to the bottom.

******

******

# 

***Use of Super Keyword***

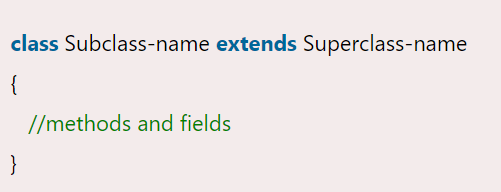
The super keyword in Java is used within a subclass to refer to its immediate superclass. It serves several purposes, including calling superclass constructors and accessing superclass methods and fields that have been overridden or hidden in the subclass.

* **Calling Superclass Constructors:** Use super(arguments) to call a specific constructor in the superclass.
* **Accessing Superclass Methods:** Use super.methodName(arguments) to call a method from the superclass that has been overridden in the subclass.
* **Accessing Superclass Fields:** Use super.fieldName to access a field from the superclass that has been hidden by a field with the same name in the subclass.

Classes can reuse code and properties from other classes. **Inheritance in Java** is a mechanism in which one object acquires all the properties and behaviors of a parent object. Relationships and subclasses between objects can be assigned, enabling developers to reuse common logic, while still maintaining a unique hierarchy. Inheritance forces more thorough data analysis, reduces development time and ensures a higher level of accuracy.

Inheritance represents the **IS-A relationship** which is also known as a parent-child relationship. It is used for:-

* For [Method Overriding](https://www.javatpoint.com/method-overriding-in-java) (so [runtime polymorphism](https://www.javatpoint.com/runtime-polymorphism-in-java) can be achieved).
* For Code Reusability.



Note

* Things that are generally **Inherited:** Non-private instance fields, non-private methods, protected members, default (package-private) members.
* **Not Inherited:** Constructors, private fields and methods, static members, final methods (cannot be overridden).

**Why multiple Inheritance not permissible in java?**

Multiple inheritance, where a class can inherit from more than one superclass, is not supported in Java. This decision was made to simplify the language and avoid certain complexities and issues associated with multiple inheritance, especially the "Diamond Problem." Here are the main reasons why multiple inheritance is not allowed in Java:

### 1. ****The Diamond Problem:****

The Diamond Problem occurs when a class inherits from two classes that both inherit from a common superclass. This creates ambiguity when the subclass inherits methods or fields from the superclass.

* Class D inherits from both B and C.
* Both B and C inherit from A.

If class A has a method doSomething(), and both B and C override this method, class D would inherit two versions of doSomething(), leading to ambiguity about which version to use.

* Class D inherits from both B and C.
* Both B and C inherit from A.

If class A has a method doSomething(), and both B and C override this method, class D would inherit two versions of doSomething(), leading to ambiguity about which version to use.

### How Java Provides Alternatives to Multiple Inheritance

To achieve the benefits of multiple inheritance without its drawbacks, Java provides interfaces and default methods.

#### Interfaces:

Interfaces allow a class to inherit method signatures from multiple sources without implementing the methods. A class can implement multiple interfaces.

# Association, Composition and Aggregation in Java

In Java and other object-oriented programming languages, association, composition, and aggregation are ways to define relationships between classes. These relationships help in modeling real-world scenarios by establishing how objects of different classes interact with each other.

 **Association:** Represents a general relationship where classes are aware of each other. There is no ownership.

 **Composition:** Represents a strong relationship where the contained object (component) cannot exist independently of the container object (composite). There is strong ownership.

 **Aggregation:** Represents a weak relationship where the contained object (component) can exist independently of the container object (aggregate). There is weak ownership.

**Aggregation**

**It is a special form of Association where:**

* It represents Has-A’s relationship.
* It is a **unidirectional association** i.e. a one-way relationship. For example, a department can have students but vice versa is not possible and thus unidirectional in nature.
* In Aggregation,**both entries can survive individually** which means ending one entity will not affect the other entity.

**Composition**

Composition is a restricted form of Aggregation in which two entities are highly dependent on each other.

* It represents **part-of** relationship.
* In composition, both entities are dependent on each other.
* When there is a composition between two entities, the composed object **cannot exist** without the other entity.

***Chapter 8 Polymorphism***

Polymorphism allows you to define one interface and have multiple implementations. The word “poly” means many and “morphs” means forms, So it means many forms.

**Types of Java Polymorphism**

In Java Polymorphism is mainly divided into two types:

* Compile-time Polymorphism

It is also known as static polymorphism. This type of polymorphism is achieved by function overloading or operator overloading.

*But Java doesn’t support the Operator Overloading.*

* Runtime Polymorphism

It is also known as Dynamic Method Dispatch. It is a process in which a function call to the overridden method is resolved at Runtime. This type of polymorphism is achieved by Method Overriding. [**Method overriding**](https://www.geeksforgeeks.org/overriding-in-java/), on the other hand, occurs when a derived class has a definition for one of the member functions of the base class. That base function is said to be **overridden**.

### Different ways to overload the method

There are two ways to overload the method in java

1. By changing number of arguments
2. By changing the data type

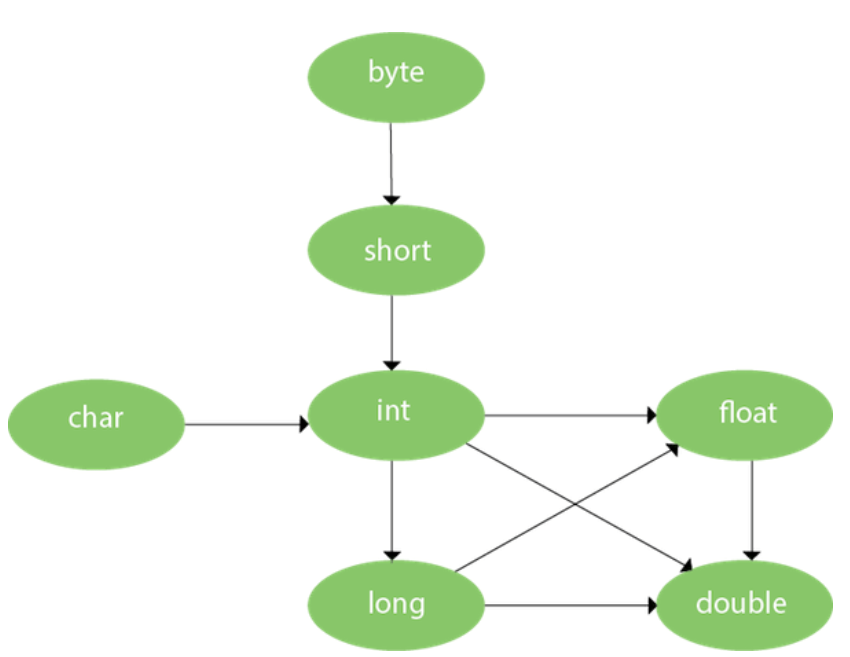
#### In Java, Method Overloading is not possible by changing the return type of the method only

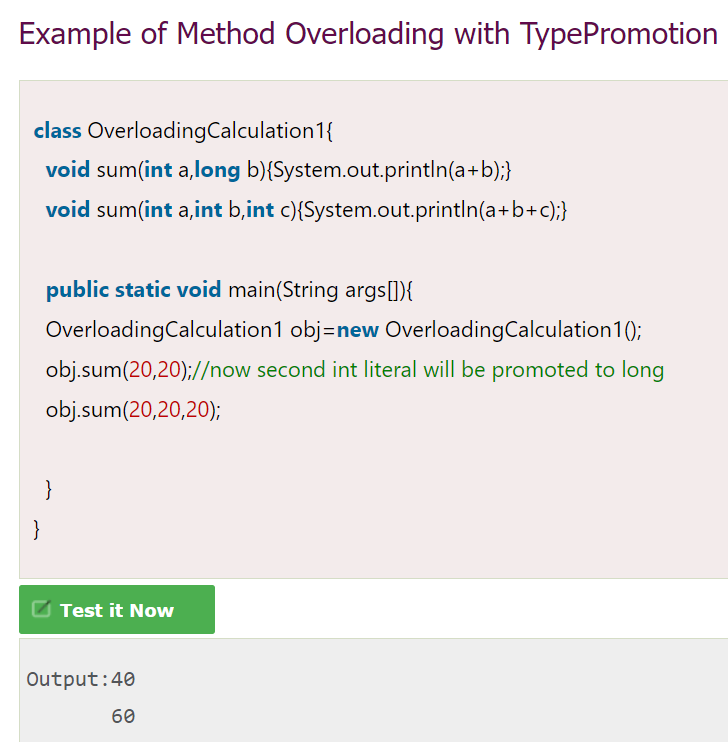
### Can we overload java main() method?

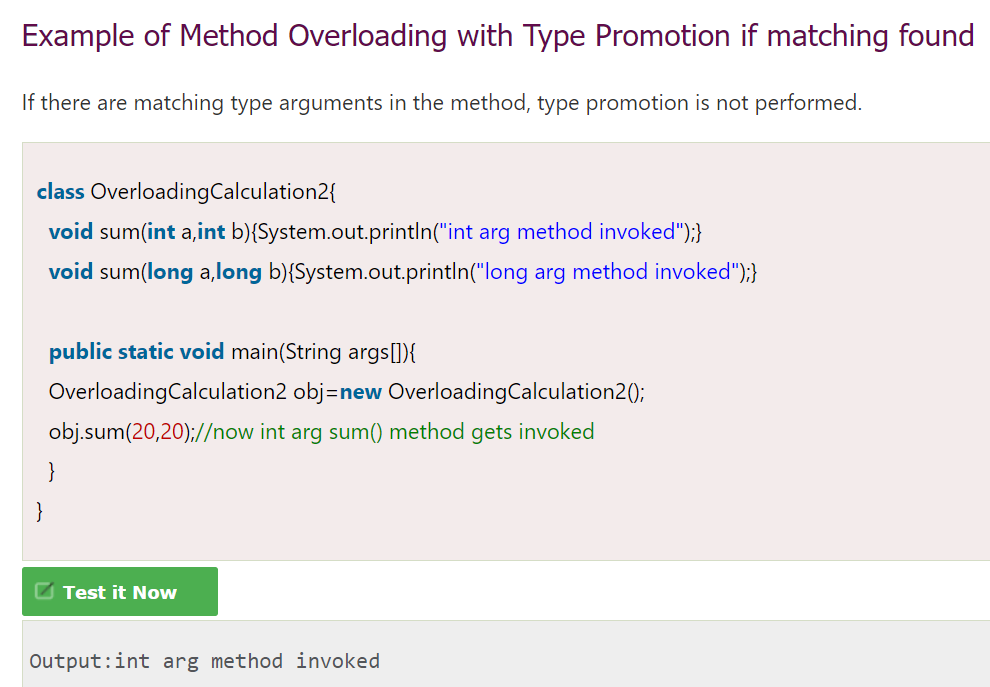
Yes, by method overloading. You can have any number of main methods in a class by method overloading. But [JVM](https://www.javatpoint.com/jvm-java-virtual-machine) calls main() method which receives string array as arguments only

## Method Overloading and Type Promotion

One type is promoted to another implicitly if no matching datatype is found. Let's understand the concept by the figure given below:

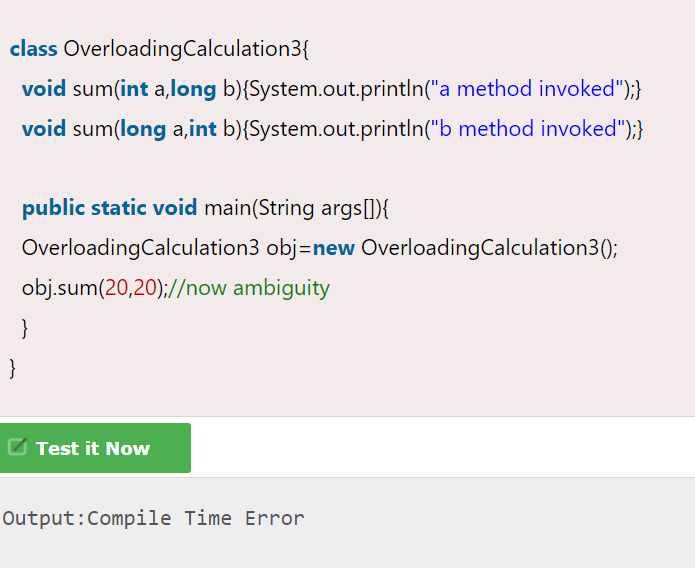
******

******

******

### Example of Method Overloading with Type Promotion in case of ambiguity

If there are no matching type arguments in the method, and each method promotes similar number of arguments, there will be ambiguity.



***Chapter 9 Miscellaneous***

The **java instanceof operator** is used to test whether the object is an instance of the specified type (class or subclass or interface).

The instanceof in java is also known as type *comparison operator* because it compares the instance with type. It returns either true or false. If we apply the instanceof operator with any variable that has null value, it returns false

1. **class** Simple1{
2. **public** **static** **void** main(String args[]){
3. Simple1 s=**new** Simple1();
4. System.out.println(s **instanceof** Simple1);//true
5. }
6. }
7. **class** Animal{}
8. **class** Dog1 **extends** Animal{//Dog inherits Animal
10. **public** **static** **void** main(String args[]){
11. Dog1 d=**new** Dog1();
12. System.out.println(d **instanceof** Animal);//true
13. }
14. }

An object of subclass type is also a type of parent class. For example, if Dog extends Animal then object of Dog can be referred by either Dog or Animal class.

1. **class** Dog2{
2. **public** **static** **void** main(String args[]){
3. Dog2 d=**null**;
4. System.out.println(d **instanceof** Dog2);//false
5. }
6. }

If we apply instanceof operator with a variable that have null value, it returns false.

***Object Cloning clone() method***:

* To use the Object.clone() method, we have to change a lot of syntaxes to our code, like implementing a Cloneable interface, defining the clone() method and handling CloneNotSupportedException, and finally, calling Object.clone() etc.
* We have to implement cloneable interface while it doesn't have any methods in it. We just have to use it to tell the JVM that we can perform clone() on our object.
* Object.clone() is protected, so we have to provide our own clone() and indirectly call Object.clone() from it.
* Object.clone() doesn't invoke any constructor so we don't have any control over object construction.
* If you want to write a clone method in a child class then all of its superclasses should define the clone() method in them or inherit it from another parent class. Otherwise, the super.clone() chain will fail.
* Object.clone() supports only shallow copying but we will need to override it if we need deep cloning.

1. **class** Student **implements** Cloneable{
2. **int** rollno;
3. String name;
5. Student(**int** rollno,String name){
6. **this**.rollno=rollno;
7. **this**.name=name;
8. }
10. **public** Object clone()**throws** CloneNotSupportedException{
11. **return** **super**.clone();
12. }
14. **public** **static** **void** main(String args[]){
15. **try**{
16. Student s1=**new** Student(101,"amit");
18. Student s2=(Student)s1.clone();
19. System.out.println(s1.rollno+" "+s1.name);
20. System.out.println(s2.rollno+" "+s2.name);
22. }**catch**(CloneNotSupportedException c){}
24. }
25. }