1. What is class?

In object-oriented programming (OOP), a class is a blueprint or a template for creating objects. It defines a collection of related data and behavior (methods) that the objects will possess when they are instantiated.

A class encapsulates the properties (attributes) and behaviors (methods) of an object, allowing you to define its structure and functionality. The properties represent the state or characteristics of an object, while the methods define the actions or operations that the object can perform.

For example, let's say we want to model a class called "Car." The Car class would define the attributes of a car object such as its color, model, and speed. It would also define methods like "accelerate," "brake," and "changeGear" to represent the actions a car can perform.

Once a class is defined, you can create instances or objects of that class. Each object created from the class will have its own set of attributes and can perform the methods defined by the class. For instance, you can create multiple car objects from the Car class, each with its own color, model, and speed, and each object can independently accelerate, brake, or change gears.

Classes provide a way to organize and structure code in a modular and reusable manner. They promote the concept of encapsulation, where the internal workings of an object are hidden from the outside world, and interactions with the object are done through a well-defined interface provided by the class.

In summary, a class in object-oriented programming acts as a blueprint that defines the properties and behaviors of objects, allowing you to create multiple instances of the class and work with them in a consistent and organized manner.

1. What is object?

In object-oriented programming (OOP), an object is a specific instance of a class. It represents a runtime entity that has state (attributes) and behavior (methods) defined by its class.

An object is created by instantiating a class. When you create an object, it has its own unique set of attributes and can perform the methods defined in its class. Each object has its own state, which means that the attribute values of one object can be different from another object of the same class.

To continue with the previous example, if "Car" is a class, an object of the Car class could be a specific car instance, such as a red Ford Mustang. The object would have its own specific attributes like color, model, and speed, which may differ from other car objects.

Objects allow you to work with data and perform actions in a modular and organized way. They encapsulate related data and behavior, promoting code reusability and separation of concerns. You can create multiple objects from the same class, each representing a distinct instance with its own state and behavior.

Objects interact with each other by invoking methods or accessing attributes of other objects. This interaction enables the objects to collaborate and perform tasks as part of a larger program.

In summary, an object in object-oriented programming is a specific instance of a class. It has its own unique state and can perform actions defined by its class. Objects provide a way to represent and manipulate data in a structured and modular manner.

1. What is constructer in oops?

In Java, a constructor is a special method within a class that is used to initialize objects. It is called automatically when an object of a class is created. The constructor has the same name as the class and does not have a return type, not even void.

The constructor is used to set the initial state of an object by assigning values to its instance variables or performing other necessary setup operations. It ensures that the object is properly initialized before it is used.

Key points about constructors in Java:

1. Constructor name: The constructor has the same name as the class. For example, if the class name is "Car," the constructor would be defined as "public Car() {}".

2. Default constructor: If you don't explicitly define a constructor for a class, Java provides a default constructor with no arguments. It initializes the object with default values (null for object references, 0 for numeric types, and false for boolean type). However, if you define a constructor explicitly, the default constructor is not generated automatically.

3. Parameterized constructor: You can define constructors with parameters to accept values and initialize the object's state based on those values. For example, you can have a constructor like "public Car(String color, String model) {}" that takes color and model as arguments.

4. Overloading constructors: Like methods, constructors can be overloaded, which means you can define multiple constructors with different parameter lists. This allows you to create objects using different initialization options.

5. Constructor chaining: Constructors can call other constructors within the same class using the "this" keyword. This is known as constructor chaining and allows you to reuse code and provide different initialization paths.

Constructors play a crucial role in object creation and initialization in Java. They ensure that objects are properly set up and ready to be used. By defining constructors, you have control over the initialization process and can enforce certain rules or constraints during object creation.

1. What is Methods in java?

In Java, a method is a block of code that performs a specific task. It is a collection of statements that are grouped together and given a name. Methods are used to organize code into logical units, promote code reusability, and make the code more readable and maintainable.

Key points about methods in Java:

1. Method declaration: A method is declared within a class and consists of a method signature and a method body. The method signature includes the access modifier (public, private, protected, or none), the return type (void if the method does not return a value), the method name, and any parameters the method accepts. For example: `public int calculateSum(int a, int b) { }`

2. Method parameters: Methods can accept zero or more parameters, which are values passed into the method for it to work with. Parameters are defined within parentheses after the method name in the method declaration. For example, in the method `calculateSum(int a, int b)`, `a` and `b` are the parameters.

3. Method return type: The return type specifies the type of value that the method returns after performing its task. If a method does not return a value, its return type is `void`. If a method returns a value, the return type indicates the type of that value. For example, a method with `int` return type would return an integer value.

4. Method body: The method body contains the statements or code that defines the behavior of the method. It is enclosed within curly braces `{}`. This is where the actual computation, operations, or actions take place.

5. Method invocation: To use a method, you invoke or call it by its name followed by parentheses. If the method accepts parameters, you pass the values for those parameters inside the parentheses. For example, `int result = calculateSum(5, 3);`.

6. Method overloading: In Java, you can have multiple methods with the same name but different parameter lists. This is called method overloading. The compiler differentiates between these methods based on the number, types, and order of the parameters. It allows you to have methods with similar functionality but different ways of accepting input.

7. Method modifiers: Java provides access modifiers like `public`, `private`, and `protected`, which determine the accessibility of methods. These modifiers control from where and by whom a method can be accessed.

Methods are essential building blocks in Java programs. They encapsulate reusable code, promote code organization and modularization, and enable the implementation of various functionalities and behaviors within a class.

1. Types of methods in java?

A:User defined method:

In Java, a user-defined method is a method that is created by the programmer to perform a specific task or set of operations. These methods are not built-in or provided by the Java language itself but are defined by the programmer to fulfill their specific requirements.

User-defined methods are also known as custom methods or programmer-defined methods. They allow you to encapsulate a block of code into a reusable unit, which can be called from different parts of the program whenever needed. User-defined methods help in promoting code reusability, readability, and maintainability.

To create a user-defined method in Java, you need to follow these steps:

1. Method declaration: Define the method by specifying the access modifier (public, private, protected, or none), the return type (void if the method does not return a value or the type of the value it returns), the method name, and any parameters the method accepts.

2. Method body: Write the code statements inside the method body to define the behavior of the method. This is where you implement the logic or operations that the method needs to perform.

3. Method invocation: Call or invoke the user-defined method from another part of the program whenever you want to execute the code statements inside that method. The method can be called by its name, followed by parentheses and any required arguments.

Here's an example of a simple user-defined method in Java:

```java

public class Example {

public static void main(String[] args) {

// Call the user-defined method

printMessage();

}

// User-defined method to print a message

public static void printMessage() {

System.out.println("Hello, world!");

}

}

```

In the above example, the `printMessage()` method is a user-defined method that prints the message "Hello, world!". The method is defined with a return type of `void` since it doesn't return any value. It is called from the `main()` method using its name, `printMessage()`, which executes the code inside the method and displays the message on the console.

User-defined methods can have parameters, return values, and perform complex operations depending on the specific requirements of the program. They allow you to modularize your code, improve code organization, and make your program more readable and maintainable.

B.Build in method in java:

In Java, there are numerous built-in methods that are provided by the Java standard libraries. These methods are part of various classes and can be directly accessed without the need for additional implementation. Here are some examples of commonly used built-in methods in Java:

1.`String.length()`: This method is used to determine the length of a string by returning the number of characters in the string. For example:

String str = "Hello, World!";

int length = str.length();

System.out.println("Length of the string: " + length);

Output: Length of the string: 13

2. `Math.max()`: This method returns the larger of two numbers. For example:

int a = 5;

int b = 8;

int maxNumber = Math.max(a, b);

System.out.println("Maximum number: " + maxNumber);

Output: Maximum number: 8

3. `Arrays.sort()`: This method is used to sort an array in ascending order. For example:

int[] numbers = {5, 2, 8, 1, 4};

Arrays.sort(numbers);

System.out.println("Sorted array: " + Arrays.toString(numbers));

Output: Sorted array: [1, 2, 4, 5, 8]

4. `ArrayList.size()`: This method returns the number of elements in an ArrayList. For example:

ArrayList<String> fruits = new ArrayList<>();

fruits.add("Apple");

fruits.add("Banana");

fruits.add("Orange");

int size = fruits.size();

System.out.println("Size of ArrayList: " + size);

Output: Size of ArrayList: 3

5. `Scanner.nextLine()`: This method reads a line of input from the user through the console. For example:

Scanner scanner = new Scanner(System.in);

System.out.print("Enter your name: ");

String name = scanner.nextLine();

System.out.println("Hello, " + name + "!");

Output: Enter your name: John

Hello, John!

These are just a few examples of built-in methods in Java. The Java standard libraries provide a wide range of methods for various purposes, including string manipulation, mathematical operations, array handling, input/output operations, and more. These methods save developers time and effort by providing pre-implemented functionality for common tasks.

1. Why used this keyword explain with an example?

In Java, the `this` keyword is a reference to the current instance of the class. It can be used within an instance method or constructor to refer to the object on which the method is being invoked. The `this` keyword is mainly used to differentiate between instance variables and method parameters or to invoke other constructors within the same class.

Here's an example to illustrate the usage of the `this` keyword:

public class Person {

private String name;

private int age;

public Person(String name, int age) {

this.name = name;

this.age = age;

}

public void printDetails() {

System.out.println("Name: " + this.name);

System.out.println("Age: " + this.age);

}

public void updateAge(int age) {

this.age = age;

}

}

In the above example, the `Person` class has two instance variables, `name` and `age`, which are also the names of the method parameters in the constructor. To distinguish between the instance variables and the method parameters, the `this` keyword is used.

In the constructor, `this.name` refers to the instance variable `name`, while `name` refers to the constructor parameter `name`. Similarly, `this.age` refers to the instance variable `age`, while `age` refers to the constructor parameter `age`.

The `printDetails()` method uses the `this` keyword to refer to the instance variables `name` and `age` explicitly, though it is optional in this case.

The `updateAge()` method also uses the `this` keyword to assign the value of the method parameter `age` to the instance variable `age`. Without the `this` keyword, the assignment would refer to the method parameter itself, and the instance variable would remain unchanged.

The `this` keyword is particularly useful in scenarios where the names of instance variables clash with method parameters or when you need to access the current instance within the class. It helps in resolving the ambiguity and accessing the correct member or invoking the appropriate constructor.

1. What is inheritance in java ?

In Java, inheritance is a fundamental object-oriented programming concept that allows you to create new classes based on existing classes. It enables you to define a new class (called the "derived class" or "subclass") that inherits the properties and behaviors (methods) of an existing class (called the "base class" or "superclass"). The derived class can then add or modify the inherited properties and behaviors.

Inheritance in Java provides several benefits:

1. Code Reusability: Inheritance allows you to reuse existing code by inheriting the members (variables and methods) from a base class. This helps avoid duplicating code and promotes a more modular and maintainable code structure.

2. Extensibility: Derived classes can extend the functionality of the base class by adding new properties and behaviors. This enables you to model more specific or specialized objects based on a more general class.

3. Method Overriding: Inherited methods can be overridden in the derived class to provide specific implementations. This allows you to customize the behavior of inherited methods to suit the requirements of the derived class.

4. Polymorphism: Inheritance is closely tied to polymorphism, which allows objects of a derived class to be treated as objects of the base class. This facilitates code flexibility, as the same code can operate on objects of different classes that share a common base class.

To establish an inheritance relationship in Java, you use the `extends` keyword when defining a class. Here's an example:

```java

class Vehicle {

// Base class

// Properties and methods

}

class Car extends Vehicle {

// Derived class

// Additional properties and methods

}

```

In this example, the `Car` class extends the `Vehicle` class, indicating that `Car` is a specialized type of `Vehicle`. The `Car` class inherits all the properties and methods defined in the `Vehicle` class, allowing you to access and use them in the `Car` class.

Note that in Java, single inheritance is supported, meaning a class can extend only one superclass. However, a superclass can have multiple subclasses, forming an inheritance hierarchy. To address the need for multiple inheritance-like behavior, Java introduced interfaces, which allow a class to implement multiple interfaces.

1. Types of inheritance in java?

In Java, there are several types of inheritance that you can utilize:

1. Single Inheritance: Single inheritance is the simplest form of inheritance, where a class extends only one superclass. It allows a subclass to inherit the properties and behaviors of a single superclass.

2. Multilevel Inheritance: Multilevel inheritance involves a chain of inheritance, where a derived class extends another derived class. This creates a hierarchical structure of classes. For example, class A is the superclass of class B, and class B is the superclass of class C. Thus, class C inherits properties and behaviors from both class A and class B.

3. Hierarchical Inheritance: Hierarchical inheritance occurs when multiple classes inherit from a single superclass. In this type of inheritance, a single superclass is extended by multiple subclasses, each having its additional properties and behaviors. For example, class A is the superclass, and classes B, C, and D are its subclasses.

4. Multiple Inheritance (through Interfaces): Java does not support multiple inheritance directly (i.e., a class cannot extend multiple classes). However, Java supports a form of multiple inheritance through interfaces. An interface defines a contract of methods that a class can implement. A class can implement multiple interfaces, effectively inheriting the methods declared in those interfaces.

5. Hybrid Inheritance: Hybrid inheritance combines multiple types of inheritance, such as single inheritance, multilevel inheritance, and hierarchical inheritance. It is a more complex form of inheritance that involves the combination of these inheritance types.

It's important to note that while Java supports single inheritance and multiple inheritance through interfaces, it does not directly support multiple inheritance of classes to avoid some of the complications and ambiguities that can arise with multiple inheritance.

1. What is polymorphism in java?

**Polymorphism in Java** is a concept by which we can perform a single action in different ways. Polymorphism is derived from 2 Greek words: poly and morphs. The word "poly" means many and "morphs" means forms. So polymorphism means many forms.

There are two types of polymorphism in Java: compile-time polymorphism and runtime polymorphism. We can perform polymorphism in java by method overloading and method overriding.

If you overload a static method in Java, it is the example of compile time polymorphism. Here, we will focus on runtime polymorphism in java.

If you overriding methods in java that’s are the Example of runtime polymorphism.

1. What is Method overloading ?
2. If a [class](https://www.javatpoint.com/object-and-class-in-java) has multiple methods having same name but Working principle are different , it is known as **Method Overloading.**
3. **Method are overloaded two ways: By changing numbers of parameater and By changing types of parameater. You always know that return type of the methods are always same.**
4. **Advantage:** If we have to perform only one operation, having same name of the methods increases the readability of the [program](https://www.javatpoint.com/java-programs).
5. **What is method overriding in java ?**
6. if subclass (child class) has the same method as declared in the parent class, it is known as **method overriding in Java**. In other words, If a subclass provides the specific implementation of the method that has been declared by one of its parent class, it is known as method overriding.
7. Usage of Java Method Overriding: Method overriding is used to provide the specific implementation of a method which is already provided by its superclass. Method overriding is used for runtime polymorphism.

#### Rules for Java Method Overriding

1. The method must have the same name as in the parent class
2. The method must have the same parameter as in the parent class.
3. There must be an IS-A relationship (inheritance).
4. Encapsulation in java.
5. **Encapsulation in Java** is a *process of wrapping code and data together into a single unit*, for example, a capsule which is 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 a fully encapsulated class.

1. **Advantage of Encapsulation in Java:** It provides you the **control over the data**. Suppose you want to set the value of id which should be greater than 100 only, you can write the logic inside the setter method. You can write the logic not to store the negative numbers in the setter methods.It is a way to achieve **data hiding** in Java because other class will not be able to access the data through the private data members.The encapsulate class is **easy to test**. So, it is better for unit testing.
2. Access modifier in java?
3. In java programming four types of access modifier:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Access modifier | Within same class | Within sub-class | Within package | Out side of package |
| public | yes | yes | yes | yes |
| private | yes | no | No | No |
| protected | yes | Yes but only one time inherited | yes | No |
| default | yes | yes | no | no |
|  |  |  |  |  |

1. Access modifier is used for visibility control all the members /methods .
2. What is data hiding in java ?

Data hiding is a technique of hiding internal object details, i.e., data members. It is an object-oriented programming technique. Data hiding ensures, or we can say guarantees to restrict the data access to class members. It maintains data integrity.

Data hiding means hiding the internal data within the class to prevent its direct access from outside the class.

If we talk about data encapsulation so, **Data encapsulation** hides the private methods and class data parts, whereas **Data hiding** only hides class data components. Both data hiding and data encapsulation are essential concepts of object-oriented programming. **Encapsulation** wraps up the complex data to present a simpler view to the user, whereas **Data hiding** restricts the data use to assure data security.

Data hiding also helps to reduce the system complexity to increase the robustness by limiting the interdependencies between software components. Data hiding is achieved by using the private access specifier.

### What is Association in java.

### 1.Association in Java defines the connection between two classes that are set up through their objects. Association manages ****one-to-one, one-to-many****, and ****many-to-many**** relationships. In [Java](https://www.javatpoint.com/java-tutorial), the multiplicity between objects is defined by the ****Association****. It shows how objects communicate with each other and how they use the functionality and services provided by that communicated object. Association manages ****one-to-one,one-to-many,many-to-one**** and ****many-tomany**** relationships.

### Examples:

### A person can have only one passport. It defines the one-to-one

If we talk about the Association between a College and Student, a College can have many students. It defines the **one-to-many**

A state can have several cities, and those cities are related to that single state. It defines the **many-to-one**

A single student can associate with multiple teachers, and multiple students can also be associated with a single teacher. Both are created or deleted independently, so it defines the **many-to-many**

### Types of association in java.

### Association in Java

### 1) IS-A Association

The IS-A Association is also referred to as [Inheritance](https://www.javatpoint.com/inheritance-in-java). We all know about Inheritance in Java and if you don't know about it.

### 2) HAS-A Association

The **HAS-A Association** is further classified into two parts, i.e., Aggregation and Composition. Let's understand the difference between both of them one by one.

### What is aggregation in java.

1. in Java, the [**Aggregation**](https://www.javatpoint.com/aggregation-in-java) association defines the **HAS-A** relationship. Aggregation follows the one-to-one or one-way relationship. If two entities are in the aggregation composition, and one entity fails due to some error, it will not affect the other entity.
2. Let's take the example of a toy and its battery. The battery belongs to a toy, and if the toy breaks and deletes from our database, the battery will still remaining in our database, and it may still be working. So in Aggregation, objects always have their own lifecycles when the ownership exists there.

### What is Composition in java.

1. A restricted form of the **Aggregation** where the entities are strongly dependent on each other. Unlike Aggregation, [Composition](https://www.javatpoint.com/composition-in-java) represents the **part-of** relationship. When there is an aggregation between two entities, the aggregate object can exist without the other entity, but in the case of Composition, the composed object can't exist. To learn more about Composition, [click here](https://www.javatpoint.com/composition-in-java).
2. Let's take an example to understand the concept of **Composition**.
3. We create a class **Mobile** that contains variables, i.e., **name, ram** and **rom**. We also create a class **MobileStore** that has a reference to refer to the list of mobiles. A mobile store can have more than one mobile. So, if a mobile store is destroyed, then all mobiles within that particular mobile store will also be destroyed because mobiles cannot exist without a mobile store. The relationship between the mobile store and mobiles is Composition.