**Java**

**Unit 2**

* **Introduction to classes: Class fundamental**

A class in Java is a set of objects which shares common characteristics/ behavior and common properties/ attributes. It is a user-defined blueprint or prototype from which objects are created. For example, Student is a class while a particular student named Ravi is an object.

**Properties of Java Classes**

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

**Class Declaration in Java**

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

**Example of Java Class**

class Student {

// data member (also instance variable)

int id;

// data member (also instance variable)

String name;

public static void main(String args[])

{

// creating an object of

// Student

Student s1 = new Student();

System.out.println(s1.id);

System.out.println(s1.name);

}

}

**Components of Java Classes**

In general, class declarations can include these components, in order:

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, { }.*

**Java Objects**

An object in Java is a basic unit of Object-Oriented Programming and represents real-life entities. Objects are the instances of a class that are created to use the attributes and methods of a class.  A typical Java program creates many objects, which as you know, interact by invoking methods. An object consists of:

1. **State**: 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 of an object: dog

***Java Objects***

Objects correspond to things found in the real world. For example, a graphics program may have objects such as “circle”, “square”, and “menu”. An online shopping system might have objects such as “shopping cart”, “customer”, and “product”.

**Declaring Objects (Also called instantiating a class)**

When an object of a class is created, the class is said to be instantiated. All the instances share the attributes and the behavior of the class. But the values of those attributes, i.e. the state are unique for each object. A single class may have any number of instances.

Example:

*Java Object Declaration*

As we declare variables like (type name;). This notifies the compiler that we will use the name to refer to data whose type is type. With a primitive variable, this declaration also reserves the proper amount of memory for the variable. So for reference variables , the type must be strictly a concrete class name. In general, we **can’t**create objects of an abstract class or an interface.

Dog tuffy;

If we declare a reference variable(tuffy) like this, its value will be undetermined(null) until an object is actually created and assigned to it. Simply declaring a reference variable does not create an object.

**Initializing a Java object**

The new operator instantiates a class by allocating memory for a new object and returning a reference to that memory. The new operator also invokes the class constructor.

**Ways to Create an Object of a Class**

There are four ways to create objects in Java. Strictly speaking, there is only one way(by using a *new*keyword), and the rest internally use a *new*keyword.

**1. Using new keyword**

It is the most common and general way to create an object in Java.

**Example:**

// creating object of class Test  
Test t = new Test();

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

There is a pre-defined class in java.lang package with name Class. The forName(String className) method returns the Class object associated with the class with the given string name. We have to give a fully qualified name for a class. On calling the new Instance() method on this Class object returns a new instance of the class with the given string name.

// creating object of public class Test  
// consider class Test present in *com.p1* package  
Test obj = (Test)Class.forName("com.p1.Test").newInstance();

**3. Using clone() method**

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

// creating object of class Test  
Test t1 = new Test();  
// creating clone of above object  
Test t2 = (Test)t1.clone();

**4. Deserialization**

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

FileInputStream file = new FileInputStream(filename);  
ObjectInputStream in = new ObjectInputStream(file);  
Object obj = in.readObject();

**Creating multiple objects by one type only**

In real-time, we need different objects of a class in different methods. Creating a number of references for storing them is not a good practice and therefore we declare a static reference variable and use it whenever required. In this case, the wastage of memory is less. The objects that are not referenced anymore will be destroyed by the [Garbage Collector](https://www.geeksforgeeks.org/garbage-collection-java/)of Java.

Example:

Test test = new Test();  
test = new Test();

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.

**Example:**

class Animal {}  
class Dog extends Animal {}  
class Cat extends Animal {}  
public class Test  
{  
 // using Dog object  
 Animal obj = new Dog();  
 // using Cat object  
 obj = new Cat();  
}

**Anonymous Objects in Java**

Anonymous objects are objects that are instantiated but are not stored in a reference variable.

* They are used for immediate method calls.
* They will be destroyed after method calling.
* They are widely used in different libraries. For example, in AWT libraries, they are used to perform some action on capturing an event(eg a key press).
* In the example below, when a key button(referred to by the btn) is pressed, we are simply creating an anonymous object of EventHandler class for just calling the handle method.

btn.setOnAction(new EventHandler()  
{  
 public void handle(ActionEvent event)  
 {  
 System.out.println("Hello World!");  
 }  
});

**Difference between Java Class and Objects**

The differences between class and object in Java are as follows:

| **Class** | **Object** |
| --- | --- |
| Class is the blueprint of an object. It is used to create objects. | An object is an instance of the class. |
| No memory is allocated when a class is declared. | Memory is allocated as soon as an object is created. |
| A class is a group of similar objects. | An object is a real-world entity such as a book, car, etc. |
| Class is a logical entity. | An object is a physical entity. |
| A class can only be declared once. | Objects can be created many times as per requirement. |
| An example of class can be a car. | Objects of the class car can be BMW, Mercedes, Ferrari, etc. |

* **method declaration**

A method in Java has various attributes like access modifier, return type, name, parameters, etc.

Methods can be declared using the following syntax:

1. accessModifier returnType methodName(parameters..)
2. {
3. *//logic of the function*
4. }

Let us understand what each attribute means -

* **accessModifier**: It defines the method's access type, i.e., from where it can be accessed in your application.
* **returnType**: It represents the data type of the value returned by the function. For example, a method in Java declared with int return type should return an integer value.
* **methodName**: Represents the identifier that can be used to call the method when required.
* **parameters**: These are the arguments passed into a method necessary for the function's logic. We can pass data to the methods by specifying them within the parentheses if the methods have data.

**Naming a Method**

The naming convention for a method in Java is generally a verb, in mixed case, with the first letter in lowercase and the first letter of each internal word capitalized. This naming convention is called the Camel case.

Example:

1. void addNumbersInArray(ArrayList<Integer> list) {...}

**Method Calling**

What we saw above is declaring a method. To use the method's functionality, you need to "call" a method. This is as simple as writing the method's name by passing its parameters. We should also take note of the function's return type.

**Example**

1. class Factorial {
2. *// Declaring method to find factorial of given number n*
3. static int findFactorial(int number) {
4. *// Standard logic to calculate factorial*
5. int answer = 1, i;
6. for (i = 2; i <= number; i++) answer \*= i;
7. return answer;
8. }
9. *// Driver method*
10. public static void main(String[] args) {
11. *//Calling the method findFactorial*
12. int result = findFactorial(7);
13. System.out.println("Factorial is : " + result);
14. }
15. }

**Output:**

Factorial is : 5040

**Types of Methods in Java**

**Predefined Method**

These are methods that Java class libraries define. They are also called standard library methods or built-in methods. They can be used by directly calling them. Some examples include print() in the package java.io.PrintStream, min() and max() as defined in Math class, etc.

1. public class Demo {
2. public static void main(String[] args) {
3. *//sqrt() is an in-built fucntion defined in the Math class*
4. System.out.println("Square root of 289 is : " + Math.sqrt(289));
5. }
6. }

Square root of 289 is : 17.0

**User-defined Method**

As you might have guessed, a method with custom logic is a user-defined method. The above example of converting temperature from Celsius to Fahrenheit is an example of a user-defined method in Java.

**How to Call or Invoke a User-defined Method?**

To invoke a user-defined method in Java, specify its name and provide any required arguments. Then, In the main method, call the method using its name followed by parentheses containing the arguments. Handle the return value if applicable by storing it in a variable or using it directly.

1. public class Calculator {
2. *// User-defined method to add two numbers*
3. public static int add(int num1, int num2) {
4. return num1 + num2;
5. }
6. *// User-defined method to multiply two numbers*
7. public static int multiply(int num1, int num2) {
8. return num1 \* num2;
9. }
10. public static void main(String[] args) {
11. int a = 5;
12. int b = 3;
13. *// Calling the add method*
14. int sum = add(a, b);
15. System.out.println("Sum: " + sum);
16. *// Calling the multiply method*
17. int product = multiply(a, b);
18. System.out.println("Product: " + product);
19. }
20. }

**Static Method**

*A method in a class declared as static does not need an object of the class to invoke it.* All the above built-in methods are static, so you could invoke the sqrt() method without creating an object of the Math class. In the example of a user-defined method to convert Celsius to Fahrenheit, the method is also static.

1. class StaticMethodDemo {
2. *//method declaration*
3. static void staticMethod() {
4. System.out.println("This is a static method.");
5. }
6. }
7. class DriverClass {
8. *// Driver method*
9. public static void main(String[] args) {
10. *// No need to create an object of the class StaticMethodDemo*
11. StaticMethodDemo.staticMethod();
12. }
13. }

This is a static method.

**Instance Method**

Instance methods in Java are attached to the objects of a class rather than the class itself. Here, the method belongs to a class whose object must be created to call the function. This is seen in the code snippet below, where an object **obj** of the Demo class is created to call the addNumbers() method.

1. class InstanceMethodDemo {
2. *//method declaration*
3. public void instanceMethod() {
4. System.out.println("This is an instance method.");
5. }
6. }
7. class DriverClass {
8. *// Driver method*
9. public static void main(String[] args) {
10. *// Create object of the class InstanceMethodDemo*
11. InstanceMethodDemo inst = new InstanceMethodDemo();
12. inst.instanceMethod();
13. }
14. }

This is an instance method.

**Abstract Method**

*A method without any implementation but only the method signature is called an abstract method.* An abstract method can be declared in an *abstract class or an interface*. A regular class can implement an abstract method by extending the abstract class containing the abstract method.

Abstract methods are used where you need to share the code among closely related classes or have many standard parameters.

* Abstract method in an abstract class

1. abstract class Factorial {
2. *//method declaration*
3. public abstract int findFactorial(int n);
4. }
5. class DriverClass extends Factorial {
6. *//Implementing abstract method in abstract class Temperature*
7. *//If not implemented, it throws a Compilation error*
8. public int findFactorial(int n) {
9. int answer = 1, i;
10. for (i = 2; i <= n; i++) answer \*= i;
11. return answer;
12. }
13. *// Driver method*
14. public static void main(String[] args) {
15. Factorial obj = new DriverClass();
16. int result = obj.findFactorial(7);
17. System.out.println("7! is equal to : " + result);
18. }
19. }

7! is equal to : 5040

**Factory method**

The factory method creates and returns the objects to the client. A factory method may accept an input that denotes the type of object that needs to be created. Factory methods belong to a specific design pattern called "Factory pattern" which is a way to dynamically return an object of a class it belongs to, at run-time based on the user's choice.

For example, if we have a program to customize the choice of condiments you can add to your coffee, you can get the object for the chosen condiment on the go when it is created.

**Why Do We Need Methods in Java?**

* Methods are essentially used to divide the logic of the whole program into manageable chunks with logic separation.
* Code can be reused throughout the program by calling the same method in multiple places. This is the single best use of methods in Java.
* Often, methods are used to hide or encapsulate implementation details.
* Methods also improve code readability, making it easier to understand the logic of the program.
* **overloading**

In Java, two or more [methods](https://www.programiz.com/java-programming/methods) may have the same name if they differ in parameters (different number of parameters, different types of parameters, or both). These methods are called overloaded methods and this feature is called method overloading. For example:

void func() { ... }

void func(int a) { ... }

float func(double a) { ... }

float func(int a, float b) { ... }

Here, the func() method is overloaded. These methods have the same name but accept different arguments.

In Java, a constructor is just like a method but without return type. It can also be overloaded like Java methods.

Constructor [overloading in Java](https://www.javatpoint.com/method-overloading-in-java) is a technique of having more than one constructor with different parameter lists. They are arranged in a way that each constructor performs a different task. They are differentiated by the compiler by the number of parameters in the list and their types.

**Method Overloading** in Java is a concept that allows multiple methods within a class to have the same name but different parameters. It enables a class to perform similar operations with different types or numbers of inputs, making the code more readable and adaptable.

**Key Aspects of Method Overloading:**

1. **Same Method Name:**
   * All overloaded methods must have the same name. This name must be identical across the methods being overloaded.
2. **Different Parameters:**
   * Overloaded methods must differ in the number or type of parameters. They can differ in:
     + **Number of Parameters:** Methods can have different numbers of parameters.
     + **Type of Parameters:** Methods can have different types of parameters.
     + **Order of Parameters:** Methods can have parameters in a different order.
3. **Return Type:**
   * Overloading is not determined by the return type alone. Overloaded methods must differ by their parameter lists. Methods with the same name and different return types are not considered overloaded.
4. **Compile-Time Polymorphism:**
   * Method overloading is an example of compile-time (or static) polymorphism. The method to be executed is determined at compile-time based on the method signature (name and parameter list).

**Example of Method Overloading:**

1. class MathOperations {
2. // Method with one integer parameter
3. int add(int a, int b) {
4. return a + b;
5. }
6. // Method with two double parameters
7. double add(double a, double b) {
8. return a + b;
9. }
10. // Method with three integer parameters
11. int add(int a, int b, int c) {
12. return a + b + c;
13. }
14. }
15. public class OverloadingExample {
16. public static void main(String[] args) {
17. MathOperations mathOps = new MathOperations();
18. // Calls the method with two integer parameters
19. System.out.println("Sum of 10 and 20: " + mathOps.add(10, 20));
20. // Calls the method with two double parameters
21. System.out.println("Sum of 10.5 and 20.5: " + mathOps.add(10.5, 20.5));
22. // Calls the method with three integer parameters
23. System.out.println("Sum of 10, 20, and 30: " + mathOps.add(10, 20, 30));
24. }
25. }

**Explanation:**

* The MathOperations class has three overloaded add() methods, each with a different parameter list.
* The correct add() method is called based on the arguments provided in the main() method.
  + mathOps.add(10, 20) calls the method with two int parameters.
  + mathOps.add(10.5, 20.5) calls the method with two double parameters.
  + mathOps.add(10, 20, 30) calls the method with three int parameters.

**Benefits of Method Overloading:**

1. **Improved Code Readability:**
   * Using the same method name for similar operations improves code readability and maintainability.
2. **Flexibility:**
   * Method overloading provides flexibility in performing operations with different types or numbers of arguments using a single method name.
3. **Code Reusability:**
   * Overloaded methods enable reusing the method name for different operations, reducing the need for creating multiple methods with different names for similar functionality.

**Why method overloading?**

Suppose, you have to perform the addition of given numbers but there can be any number of arguments (let’s say either 2 or 3 arguments for simplicity).

In order to accomplish the task, you can create two methods sum2num(int, int) and sum3num(int, int, int) for two and three parameters respectively. However, other programmers, as well as you in the future may get confused as the behavior of both methods are the same but they differ by name.

The better way to accomplish this task is by overloading methods. And, depending upon the argument passed, one of the overloaded methods is called. This helps to increase the readability of the program.

**How to perform method overloading in Java?**

Here are different ways to perform method overloading:

**1. Overloading by changing the number of parameters**

1. class MethodOverloading {
2. private static void display(int a){
3. System.out.println("Arguments: " + a);
4. }
5. private static void display(int a, int b){
6. System.out.println("Arguments: " + a + " and " + b);
7. }
8. public static void main(String[] args) {
9. display(1);
10. display(1, 4);
11. }
12. }

**Output**:

Arguments: 1

Arguments: 1 and 4

**2. Method Overloading by changing the data type of parameters**

1. class MethodOverloading {
2. // this method accepts int
3. private static void display(int a){
4. System.out.println("Got Integer data.");
5. }
6. // this method accepts String object
7. private static void display(String a){
8. System.out.println("Got String object.");
9. }
10. public static void main(String[] args) {
11. display(1);
12. display("Hello");
13. }
14. }

**Output**:

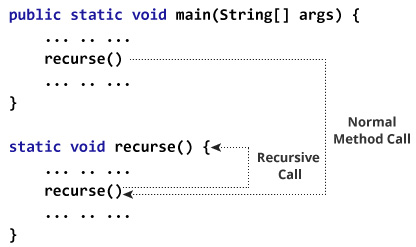
Got Integer data.

Got String object.

Here, both overloaded methods accept one argument. However, one accepts the argument of type int whereas other accepts String object.

* **recursion**

In Java, a [method](https://www.programiz.com/java-programming/methods) that calls itself is known as a recursive method. And, this process is known as recursion.



In the above example, we have called the recurse() method from inside the main method (normal method call). And, inside the recurse() method, we are again calling the same recurse method. This is a recursive call.

In order to stop the recursive call, we need to provide some conditions inside the method. Otherwise, the method will be called infinitely.

Hence, we use the [if...else statement](https://www.programiz.com/java-programming/if-else-statement) (or similar approach) to terminate the recursive call inside the method.

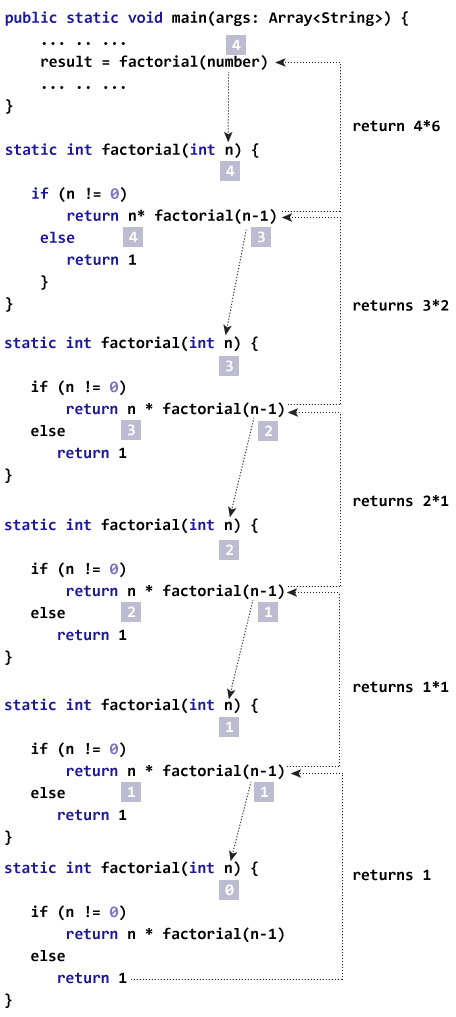
**Example: Factorial of a Number Using Recursion**

1. class Factorial {
2. static int factorial( int n ) {
3. if (n != 0) // termination condition
   1. return n \* factorial(n-1); // recursive call
4. else
   1. return 1;
5. }
6. public static void main(String[] args) {
7. int number = 4, result;
8. result = factorial(number);
9. System.out.println(number + " factorial = " + result);
10. }
11. }

**Output**:

4 factorial = 24

**Working of Factorial Program**



* **constructors**

In [Java](https://www.javatpoint.com/java-tutorial), a constructor is a block of codes similar to the method. It is called when an instance of the [class](https://www.javatpoint.com/object-and-class-in-java) is created. At the time of calling constructor, memory for the object is allocated in the memory.

It is a special type of method which is used to initialize the object.

Every time an object is created using the new() keyword, at least one constructor is called.

It calls a default constructor if there is no constructor available in the class. In such case, Java compiler provides a default constructor by default.

There are two types of constructors in Java: no-arg constructor, and parameterized constructor.

It is called constructor because it constructs the values at the time of object creation. It is not necessary to write a constructor for a class. It is because java compiler creates a default constructor if your class doesn't have any.

**Rules for creating Java constructor**

There are rules defined for the constructor-

1. Constructor name must be the same as its class name
2. A Constructor must have no explicit return type
3. A Java constructor cannot be abstract, static, final, and synchronized

**Types of Java constructors**

There are two types of constructors in Java:

1. Default constructor (no-arg constructor)
2. Parameterized constructor



Java Default Constructor

A constructor is called "Default Constructor" when it doesn't have any parameter.

Syntax of default constructor:

* + 1. <class\_name>(){}

Example of default constructor

//Java Program to create and call a default constructor

1. **class** Bike1{
2. //creating a default constructor
3. Bike1(){System.out.println("Bike is created");}
4. //main method
5. **public** **static** **void** main(String args[]){
6. //calling a default constructor
7. Bike1 b=**new** Bike1();
8. }
9. }

Output:

Bike is created

**Java Parameterized Constructor**

A constructor which has a specific number of parameters is called a parameterized constructor.

Why use the parameterized constructor?

The parameterized constructor is used to provide different values to distinct objects. However, you can provide the same values also.

Example of parameterized constructor

In this example, we have created the constructor of Student class that have two parameters. We can have any number of parameters in the constructor.

//Java Program to demonstrate the use of the parameterized constructor.

1. **class** Student4{
2. **int** id;
3. String name;
4. //creating a parameterized constructor
5. Student4(**int** i,String n){
6. id = i;
7. name = n;
8. }
9. //method to display the values
10. **void** display(){System.out.println(id+" "+name);}
11. **public** **static** **void** main(String args[]){
12. //creating objects and passing values
13. Student4 s1 = **new** Student4(111,"Karan");
14. Student4 s2 = **new** Student4(222,"Aryan");
15. //calling method to display the values of object
16. s1.display();
17. s2.display();
18. }
19. }

Output:

111 Karan

222 Aryan

|  |  |
| --- | --- |
| **Java Constructor** | **Java Method** |
| A constructor is used to initialize the state of an object. | A method is used to expose the behavior of an object. |
| A constructor must not have a return type. | A method must have a return type. |
| The constructor is invoked implicitly. | The method is invoked explicitly. |
| The Java compiler provides a default constructor if you don't have any constructor in a class. | The method is not provided by the compiler in any case. |
| The constructor name must be same as the class name. | The method name may or may not be same as the class name. |



**Java Copy Constructor**

There is no copy constructor in Java. However, we can copy the values from one object to another like copy constructor in C++.

There are many ways to copy the values of one object into another in Java. They are:

* By constructor
* By assigning the values of one object into another
* By clone() method of Object class

In this example, we are going to copy the values of one object into another using Java constructor.

//Java program to initialize the values from one object to another object.

**class** Student6{

**int** id;

    String name;

    //constructor to initialize integer and string

    Student6(**int** i,String n){

    id = i;

    name = n;

    }

    //constructor to initialize another object

    Student6(Student6 s){

    id = s.id;

    name =s.name;

    }

**void** display(){System.out.println(id+" "+name);}

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

    Student6 s1 = **new** Student6(111,"Karan");

    Student6 s2 = **new** Student6(s1);

    s1.display();

    s2.display();

   }

}

Output:

111 Karan

111 Karan

* **this keyword**

There can be a lot of usage of **Java this keyword**. In Java, this is a **reference variable** that refers to the current object.



Usage of Java this keyword

1. [this can be used to refer current class instance variable.](https://www.javatpoint.com/this1)
2. [this can be used to invoke current class method (implicitly)](https://www.javatpoint.com/this2)
3. [this() can be used to invoke current class constructor.](https://www.javatpoint.com/this3)
4. [this can be passed as an argument in the method call.](https://www.javatpoint.com/this4)
5. [this can be passed as argument in the constructor call.](https://www.javatpoint.com/this5)
6. [this can be used to return the current class instance from the method.](https://www.javatpoint.com/this6)



**1) this: to refer current class instance variable**

The this keyword can be used to refer current class instance variable. If there is ambiguity between the instance variables and parameters, this keyword resolves the problem of ambiguity.

Understanding the problem without this keyword

Let's understand the problem if we don't use this keyword by the example given below:

1. **class** Student{
2. **int** rollno;
3. String name;
4. **float** fee;
5. Student(**int** rollno,String name,**float** fee){
6. rollno=rollno;
7. name=name;
8. fee=fee;
9. }
10. **void** display(){System.out.println(rollno+" "+name+" "+fee);}
11. }
12. **class** TestThis1{
13. **public** **static** **void** main(String args[]){
14. Student s1=**new** Student(111,"ankit",5000f);
15. Student s2=**new** Student(112,"sumit",6000f);
16. s1.display();
17. s2.display();
18. }}

**Output:**

0 null 0.0

0 null 0.0

In the above example, parameters (formal arguments) and instance variables are same. So, we are using this keyword to distinguish local variable and instance variable.

Solution of the above problem by this keyword

1. **class** Student{
2. **int** rollno;
3. String name;
4. **float** fee;
5. Student(**int** rollno,String name,**float** fee){
6. **this**.rollno=rollno;
7. **this**.name=name;
8. **this**.fee=fee;
9. }
10. **void** display(){System.out.println(rollno+" "+name+" "+fee);}
11. }
13. **class** TestThis2{
14. **public** **static** **void** main(String args[]){
15. Student s1=**new** Student(111,"ankit",5000f);
16. Student s2=**new** Student(112,"sumit",6000f);
17. s1.display();
18. s2.display();
19. }}

**Output:**

111 ankit 5000.0

112 sumit 6000.0

**2) this: to invoke current class method**

You may invoke the method of the current class by using the this keyword. If you don't use the this keyword, compiler automatically adds this keyword while invoking the method.

Let's see the example



1. **class** A{
2. **void** m(){System.out.println("hello m");}
3. **void** n(){
4. System.out.println("hello n");
5. //m();//same as this.m()
6. **this**.m();
7. }
8. }
9. **class** TestThis4{
10. **public** **static** **void** main(String args[]){
11. A a=**new** A();
12. a.n();
13. }}

**Output:**

hello n

hello m

**3) this() : to invoke current class constructor**

The this() constructor call can be used to invoke the current class constructor. It is used to reuse the constructor. In other words, it is used for constructor chaining.

**Calling default constructor from parameterized constructor:**

1. **class** A{
2. A(){System.out.println("hello a");}
3. A(**int** x){
4. **this**();
5. System.out.println(x);
6. }
7. }
8. **class** TestThis5{
9. **public** **static** **void** main(String args[]){
10. A a=**new** A(10);
11. }}

**Output:**

hello a

10

**Calling parameterized constructor from default constructor:**

1. **class** A{
2. A(){
3. **this**(5);
4. System.out.println("hello a");
5. }
6. A(**int** x){
7. System.out.println(x);
8. }
9. }
10. **class** TestThis6{
11. **public** **static** **void** main(String args[]){
12. A a=**new** A();
13. }}

**Output:**

5

hello a

**Real usage of this() constructor call**

The this() constructor call should be used to reuse the constructor from the constructor. It maintains the chain between the constructors i.e. it is used for constructor chaining.

Let's see the example given below that displays the actual use of this keyword.

1. **class** Student{
2. **int** rollno;
3. String name,course;
4. **float** fee;
5. Student(**int** rollno,String name,String course){
6. **this**.rollno=rollno;
7. **this**.name=name;
8. **this**.course=course;
9. }
10. Student(**int** rollno,String name,String course,**float** fee){
11. **this**(rollno,name,course);//reusing constructor
12. **this**.fee=fee;
13. }
14. **void** display(){System.out.println(rollno+" "+name+" "+course+" "+fee);}
15. }
16. **class** TestThis7{
17. **public** **static** **void** main(String args[]){
18. Student s1=**new** Student(111,"ankit","java");
19. Student s2=**new** Student(112,"sumit","java",6000f);
20. s1.display();
21. s2.display();
22. }}

**Output:**

111 ankit java 0.0

112 sumit java 6000.0

Rule: Call to this() must be the first statement in constructor.

1. **class** Student{
2. **int** rollno;
3. String name,course;
4. **float** fee;
5. Student(**int** rollno,String name,String course){
6. **this**.rollno=rollno;
7. **this**.name=name;
8. **this**.course=course;
9. }
10. Student(**int** rollno,String name,String course,**float** fee){
11. **this**.fee=fee;
12. **this**(rollno,name,course);//C.T.Error
13. }
14. **void** display(){System.out.println(rollno+" "+name+" "+course+" "+fee);}
15. }
16. **class** TestThis8{
17. **public** **static** **void** main(String args[]){
18. Student s1=**new** Student(111,"ankit","java");
19. Student s2=**new** Student(112,"sumit","java",6000f);
20. s1.display();
21. s2.display();
22. }}

**Output:**

Compile Time Error: Call to this must be first statement in constructor

**4) this: to pass as an argument in the method**

The this keyword can also be passed as an argument in the method. It is mainly used in the event handling. Let's see the example:

1. **class** S2{
2. **void** m(S2 obj){
3. System.out.println("method is invoked");
4. }
5. **void** p(){
6. m(**this**);
7. }
8. **public** **static** **void** main(String args[]){
9. S2 s1 = **new** S2();
10. s1.p();
11. }
12. }

**Output:**

method is invoked

Application of this that can be passed as an argument:

In event handling (or) in a situation where we have to provide reference of a class to another one. It is used to reuse one object in many methods.

**5) this: to pass as argument in the constructor call**

We can pass the this keyword in the constructor also. It is useful if we have to use one object in multiple classes. Let's see the example:

1. **class** B{
2. A4 obj;
3. B(A4 obj){
4. **this**.obj=obj;
5. }
6. **void** display(){
7. System.out.println(obj.data);
8. //using data member of A4 class
9. }
10. }
12. **class** A4{
13. **int** data=10;
14. A4(){
15. B b=**new** B(**this**);
16. b.display();
17. }
18. **public** **static** **void** main(String args[]){
19. A4 a=**new** A4();
20. }
21. }

Output:10

**6) this keyword can be used to return current class instance**

We can return this keyword as an statement from the method. In such case, return type of the method must be the class type (non-primitive). Let's see the example:

Syntax of this that can be returned as a statement

1. return\_type method\_name(){
2. **return** **this**;
3. }

Example of this keyword that you return as a statement from the method

1. **class** A{
2. A getA(){
3. **return** **this**;
4. }
5. **void** msg(){System.out.println("Hello java");}
6. }
7. **class** Test1{
8. **public** **static** **void** main(String args[]){
9. **new** A().getA().msg();
10. }
11. }

**Output:**

Hello java

Proving this keyword

Let's prove that this keyword refers to the current class instance variable. In this program, we are printing the reference variable and this, output of both variables are same.

1. **class** A5{
2. **void** m(){
3. System.out.println(**this**);//prints same reference ID
4. }
5. **public** **static** **void** main(String args[]){
6. A5 obj=**new** A5();
7. System.out.println(obj);//prints the reference ID
8. obj.m();
9. }
10. }

**Output:**

A5@22b3ea59

A5@22b3ea59

What is Java Keywords?

Java keywords are also known as reserved words. Keywords are particular words that act as a key to a code. These are predefined words by Java so they cannot be used as a variable or object name or class name.

* **garbage collection**

Garbage Collection is a key feature of the Java programming language that automatically manages memory allocation and deallocation for objects that are created in an eden space.

Garbage Collection in Java allows developers to focus on writing code without worrying about memory management, making Java a popular choice for building complex and large-scale applications. However, understanding how Garbage Collection works is essential for Java developers to optimize their code's performance and avoid common memory-related errors.

What is OutofMemoryErrors?

OutofMemoryError is a type of error that occurs when a program or application attempts to allocate more memory than the available amount. This error occurs when the Java Virtual Machine (JVM) or another platform runs out of memory while trying to run an application.

An OutofMemoryError typically occurs when an application or program tries to create new objects, but the JVM is unable to allocate memory to accommodate them. This error can also occur when an application is using too much memory and is not releasing it properly.

When an OutofMemoryError occurs, the application will usually crash and terminate. This error is common in programs that deal with large amounts of metadata, such as an image or video processing applications, or programs that handle large databases.

To resolve this error, you may need to increase the amount of memory available to the application or optimize the application's memory usage. This can be done by modifying the JVM parameters or by using a memory profiler tool to identify memory leaks or inefficient memory usage.

How does garbage collection work in Java?

In Java, all objects are stored on the heap, which is a portion of memory that is reserved for dynamic allocation of objects. When an object is no longer being referenced by any part of the program, it becomes eligible for garbage collection.

The garbage collector in Java periodically scans the heap memory to find objects that aren’t being used. The process of garbage collection involves several steps, including marking, sweeping, and compacting.

* Marking - The first step of garbage collection involves marking all objects that are still being referenced by the program. This is done by starting with a set of root objects, such as global variables, local variables, and method parameters, and then tracing all the objects that are reachable from those roots. Objects that cannot be reached from the roots are considered eligible for garbage collection.
* Sweeping - After the marking phase, the garbage collector sweeps through the Java heap to identify and reclaim the memory that is used by the objects that are no longer being referenced. This involves deallocating the memory that is used by the unused objects, and adding it back to the free memory pool.
* Compacting - In some garbage collection algorithms, the sweeping phase is followed by a compaction phase, where the memory that is used by the remaining objects is rearranged to minimize fragmentation. This involves moving objects closer together and creating larger contiguous blocks of free memory.

The Java virtual machine (JVM) automatically performs garbage collection, so the programmer does not have to manually manage memory. The garbage collector runs on a separate thread and typically operates in the background, so it doesn't affect the normal execution of the program.

Types of garbage collectors in Java

There are two main types of Garbage Collection algorithms in Java: Full Garbage Collection and Incremental Garbage Collection.

Full or Major Garbage Collection

Full garbage collection is a process in which a garbage collector (a part of a programming language's runtime system) searches through all of the memory that is used by a program and compiles any objects that are no longer being used by the program. These objects are then marked as garbage and are eligible for removal from memory.

Full garbage collection is typically performed by the runtime system of a programming language that uses automatic memory management, such as Java or Python. During the process, the garbage collector pauses the program's execution to perform the search for garbage objects, which can result in a temporary slowdown in the program's performance.

Full garbage collection is usually triggered when the amount of memory that is used by a program reaches a certain threshold or when the program requests a new block of memory, and there is not enough free memory available. The goal of full garbage collection is to reclaim memory that isn’t needed by the program, making it available for use by other parts of the program or by other programs running on the same machine.

Incremental or Minor Garbage Collection

Incremental garbage collection is a type of memory management technique that is used by programming languages and runtime environments to automatically reclaim memory that is no longer needed by a program. It does so by identifying objects in memory that aren’t being used, and freeing up the memory they occupy so that it can be reused by other parts of the program.

In incremental garbage collection, the garbage collector periodically scans the program's memory for unreachable objects in the young generation's heap memory. Instead of stopping the program's execution during this scanning process, the garbage collector divides the scanning process into small, manageable pieces called "increments". During each increment, the garbage collector scans a portion of the program's memory, identifying any objects that are not needed and marking them as available for reuse.

By using increments, the garbage collector can reclaim memory in small chunks, without interrupting the program's execution for an extended period. This helps to ensure that the program remains responsive and doesn't experience significant pauses or delays as a result of the garbage collection process.

However, incremental garbage collection can be less efficient than other types of garbage collection techniques, such as mark-and-sweep or generational garbage collection, because it requires more frequent scans of the program's memory. Additionally, the use of increments can introduce some overhead into the program's execution, as the garbage collector needs to maintain state information between each increment.

Java garbage collection benefits

Overall, Java's garbage collection provides many benefits that make it a valuable tool for developers. Here are some benefits of using Java's garbage collection:

1. No manual memory management
2. Prevents memory leaks
3. Dynamic memory allocation
4. Better performance
5. Memory optimization

* **No manual memory management:** With garbage collection, developers do not have to manually manage memory allocation and deallocation. This means that programmers can focus more on writing code and less on managing memory, which can help reduce errors and improve productivity.
* **Prevents memory leaks:**Garbage collection helps prevent memory leaks, which can occur when a program does not release memory that is no longer needed. This can cause the program to consume more memory than necessary, leading to slow performance, and eventually crashing.
* **Dynamic memory allocation:** Java's garbage collection enables dynamic memory allocation, which means that memory is allocated as needed at runtime. This helps prevent memory allocation errors and can make the program more efficient.
* **Better performance:** Garbage collection can help improve the performance of a program by reducing the amount of time spent managing memory. This can lead to faster execution times and a more responsive program.
* **Memory optimization:** Garbage collection can optimize the use of memory by reusing memory that is not used by one part of the program for other parts of the program. This can help reduce memory usage and improve overall program efficiency.

Developers benefit from Java garbage collection’s ability to automatically manage memory, prevent memory leaks, enable dynamic memory allocation, improve performance, and optimize memory usage, garbage collection can help developers write better, more efficient programs.

What events trigger Java garbage collection?

In Java, garbage collection is triggered automatically by the JVM (Java Virtual Machine) when it determines that the heap is getting full or when a certain amount of time has passed.

There are several events that can trigger garbage collection in Java:

* **Heap space allocation:** When the JVM needs to allocate memory for a new object, and there is not enough space in the heap, it triggers garbage collection to reclaim unused memory or store them in the survivor space.
* **System.gc() method call:** You can explicitly request garbage collection by calling the System.gc() method, although there is no guarantee that it will run.
* **Old generation threshold:** Garbage collection can also be triggered when the heap size of the old generation heap space (which stores long-lived objects) reaches a certain threshold.
* **PermGen/Metaspace threshold**: In Java versions before Java 8, garbage collection can also be triggered when the size of the PermGen (Permanent Generation) or Metaspace (in Java 8 and later) memory areas reaches a certain threshold.
* **Time-based:** Sometimes, garbage collection can be triggered based on a time interval. For example, the JVM might trigger garbage collection every hour or every day, regardless of memory usage.

It's worth noting that the exact behavior of garbage collection in Java can vary depending on the JVM implementation and configuration.

How to request JVM to run Garbage Collector

To request the Java Virtual Machine (JVM) to run garbage collector, you can follow these steps:

1. **Call the System.gc() method:**This method is used to request the JVM to run the garbage collector. It is not guaranteed that the garbage collector will run immediately after this method is called.
2. **Use the Runtime.getRuntime().gc() method:** This method is similar to the System.gc() method, but it is less likely to be overridden by the JVM implementation.
3. **Use the -XX:+DisableExplicitGC JVM flag:** This flag disables explicit garbage collection requests. This means that even if you call System.gc() or Runtime.getRuntime().gc(), the garbage collector will not be triggered.

It's important to note that explicitly requesting the garbage collector to run is generally not recommended, as the JVM is designed to automatically manage memory allocation and garbage collection. Explicit garbage collection requests can sometimes have a negative impact on performance.

When is an Object eligible for Garbage Collection?

An object in a programming language is eligible for garbage collection when it is no longer referenced by any part of the program. Automatic garbage collection is a process that is performed by the programming language runtime environment to reclaim memory.

In most modern programming languages, garbage collection is performed automatically by the runtime environment. The specific algorithms that are used for garbage collection can vary depending on the programming language and implementation, but the general principle is the same: the runtime environment periodically scans the heap (the portion of memory that is used for dynamically allocated objects) to identify objects that are no longer reachable from any live object in the program. Once an object is identified as not being reachable, it is marked as garbage, and its memory can be reclaimed.

The exact timing of when an object is eligible for garbage collection depends on the specific garbage collection algorithm that is used by the runtime environment. Some algorithms are more aggressive than others and may reclaim memory more quickly, while others may delay garbage collection to optimize performance. However, in general, the programmer does not need to worry about managing memory manually, as the runtime environment takes care of this automatically.

What garbage collectors are available for Java?

There are several java garbage collectors, including:

1. Serial Garbage Collector
2. Parallel Garbage Collector
3. Concurrent Mark Sweep (CMS) Collector
4. G1 Garbage Collector

* **Serial Garbage Collector:** The Serial Collector is the default garbage collector in Java and is typically used in small to medium-sized applications that do not require high throughput. This type of collector helps prevent the common “stop the world” events from occurring.
* **Parallel Garbage Collector:** The Parallel Collector is designed for high-throughput applications and is particularly useful in applications that require large heaps because it uses multiple CPUs to speed up the process. It’s important to note that this type of collector freezes application threads when you run a garbage collector.
* **Concurrent Mark Sweep (CMS) Collector:** The CMS Collector is designed for applications that require low pause times and is useful in applications that have many live objects.
* **G1 Garbage Collector:**The G1 Collector is designed for large heaps and can handle a mix of short and long-lived objects. It uses multiple threads to concurrently scan and compact the heap.

Each garbage collector has its own strengths and weaknesses, and the choice of which collector to use depends on the specific needs of the application. It is also possible to configure and tune the garbage collector settings to optimize performance for a particular application.

What is the difference between garbage collection and memory leak?

Garbage collection and memory leaks are both related to memory management in computer programs, but they have different meanings and implications.

As stated previously, garbage collection is typically performed by the programming language or runtime environment, and it helps ensure that programs do not consume more memory than they need to. Garbage collection identifies memory that is free to use by other parts of the program or by other programs running on the computer.

On the other hand, a memory leak occurs when a program fails to release the memory that it has allocated, even when that memory is no longer needed. As a result, the program continues to consume memory over time, leading to the eventual depletion of available memory, which can cause the program or the entire operating system to crash. Memory leaks are typically caused by bugs in the program, and they can be difficult to identify and fix.

In summary, garbage collection is a process for automatically freeing up memory that is no longer needed. Memory leaks occur when memory is allocated but not released by a program, causing a gradual accumulation of memory usage.

* **the finalization**

In Java, finalization is a process related to garbage collection. It provides a mechanism for performing cleanup operations before an object is permanently discarded by the garbage collector. The finalize() method can be overridden in a class, allowing developers to specify what should happen before an object is collected.

However, finalization has limitations:

1. **Uncertainty:** There's no guarantee when or if the finalize() method will be called.
2. **Performance Impact:** Relying on finalization can lead to performance issues due to the unpredictable nature of garbage collection.
3. **Obsolescence:** Java has deprecated the finalize() method in favor of more robust resource management techniques like try-with-resources and the AutoCloseable interface.

In summary, finalization is an older method for resource cleanup but is generally discouraged in modern Java programming.

**Finalization** is a mechanism in Java where an object can clean up resources before being collected by the garbage collector. This is done by overriding the finalize() method in a class. Here’s a deeper dive into the theory, benefits, and drawbacks.

**Benefits:**

1. **Resource Cleanup:** Finalization allows objects to release system resources (e.g., file handles, sockets) that are not automatically managed by the garbage collector. This can be useful in certain situations where you want to ensure that resources are freed when an object is no longer needed.
2. **Legacy Support:** For older Java versions or libraries where modern resource management (like try-with-resources) isn't available or implemented, finalization offers a fallback method for resource cleanup.
3. **Automatic Trigger:** Once implemented, the finalize() method is automatically invoked by the garbage collector, potentially reducing the need for explicit cleanup code by the programmer.

**Drawbacks:**

1. **Unpredictable Timing:** The timing of when (or if) the finalize() method is called is entirely controlled by the garbage collector. This unpredictability can lead to delays in resource release, especially if the system is low on memory.
2. **Performance Overhead:** Implementing finalization adds a layer of complexity for the garbage collector, which can slow down the system. Objects that override finalize() often take longer to be collected.
3. **No Guarantee of Execution:** There’s no guarantee that the finalize() method will ever be called. For example, if the JVM terminates abruptly, the method may not run at all, leaving resources unfreed.
4. **Error-Prone:** Writing reliable finalization code can be tricky, especially in multithreaded environments. It’s easy to introduce subtle bugs, such as holding onto references that prevent objects from being collected.
5. **Deprecated:** Due to the above issues, Java has deprecated the finalize() method as of Java 9. Modern practices favor the use of try-with-resources and implementing the AutoCloseable interface, which provide more reliable and efficient resource management.