**Class and object**

**Class:**

**A class in Java is like a blueprint or template for creating objects. It defines properties (attributes) and behaviors (methods) that the objects created from the class will have.**

**Example: Think of a class like the blueprint for a table. It specifies the dimensions, material, and design. However, the blueprint itself is not a table; it's just a plan for creating tables.**

**In programming terms, you are the class that gives the blueprint (instructions) to a carpenter (the program). The carpenter follows these instructions to create a real table (object).**

**Object:**

**An object is an instance of a class. It is a real-world entity created based on the blueprint (class). In Java, everything revolves around objects. Once the JVM creates an object using the class as a blueprint, the object will have the properties and behaviors defined by the class.**

**Example: If the blueprint is the class, then the actual table made by the carpenter is the object. Each table (object) may have its own state (specific size, color, etc.) but follows the general structure defined by the blueprint (class).**

**Stack And Heap Memory**

**1. Stack Memory:**

* **Stack memory is used for static memory allocation and storing local variables.**
* **It holds primitive data types (int, float, char, etc.) and references to objects (but not the objects themselves).**
* **Each time a method is called, a new block (or frame) is created on the stack for that method. This block contains all the method's local variables.**
* **When the method execution completes, its block is removed from the stack (last in, first out - LIFO structure).**

**Example of Stack Memory:**

**public class StackExample {**

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

**int a = 10; // stored in Stack (primitive)**

**int b = 20; // stored in Stack (primitive)**

**add(a, b); // method call creates a new stack frame for the method 'add'**

**}**

**public static void add(int x, int y) {**

**int sum = x + y; // x, y, and sum are stored in the stack frame for 'add'**

**System.out.println(sum);**

**}**

**}**

**Explanation:**

* **Variables a and b are stored in the main() method's stack frame.**
* **When the add() method is called, x, y, and sum are stored in the add() method's stack frame.**
* **Once the add() method finishes execution, its stack frame is destroyed, and x, y, and sum are removed.**

**2. Heap Memory:**

* **Heap memory is used for dynamic memory allocation, and it stores all objects created in Java using the new keyword.**
* **It holds the actual objects, and the references to these objects are stored in the stack.**
* **Heap memory is managed by the Java Garbage Collector, which reclaims memory used by objects that are no longer referenced.**

**Example of Heap Memory:**

**public class HeapExample {**

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

**// Person object is created in the heap, reference stored in the stack**

**Person person1 = new Person("John", 25);**

**Person person2 = new Person("Alice", 30);**

**}**

**}**

**class Person {**

**String name; // stored in heap**

**int age; // stored in heap**

**Person(String name, int age) {**

**this.name = name;**

**this.age = age;**

**}**

**}**

**Explanation:**

* **The two Person objects (person1 and person2) are created in the heap memory.**
* **However, their references (or pointers) are stored in the stack memory in the main() method's stack frame.**
* **The name and age fields of each object are also stored in the heap memory.**

**Static Variable In Java**

**A static variable in Java (also known as a class variable) is a variable that is shared across all instances of a class. Unlike instance variables that are unique to each object, a static variable is created once and shared by all objects of that class. Let's explore static variables in detail:**

**1. Characteristics of a Static Variable:**

* **Belongs to the class, not objects: A static variable is associated with the class itself rather than any specific instance (object) of the class. This means it is shared among all instances of the class.**
* **Single copy across all instances: Only one copy of the static variable exists, no matter how many objects of the class are created. All objects refer to the same copy of the static variable.**
* **Memory allocation: Static variables are stored in the method area (or the class area in the JVM) and are allocated memory only once, when the class is loaded.**
* **Access via class name: Static variables can be accessed directly using the class name, without needing to create an object of the class. However, they can also be accessed via objects.**

**2. Declaration of Static Variable:**

**Static variables are declared using the static keyword inside a class but outside any method, constructor, or block.**

**class MyClass {**

**// Static variable**

**static int count = 0;**

**}**

**3. Use Cases of Static Variables:**

* **To share common data: Since static variables are shared among all instances, they are often used for storing common data that needs to be accessed or modified by all instances. For example, a variable that tracks the number of objects created can be declared as static.**
* **Memory efficiency: Static variables are used when a variable needs to be shared but creating multiple copies of it would be inefficient. For example, a static variable can be used to hold configuration settings for an application.**

**4. How Static Variables Work:**

* **Without Objects: A static variable can be accessed directly using the class name, even without creating an object of the class.**

**MyClass.count = 5;**

* **With Objects: Static variables can also be accessed through objects, but it is not common practice.**

**MyClass obj1 = new MyClass();**

**MyClass obj2 = new MyClass();**

**// Accessing static variable using objects**

**obj1.count = 10;**

**obj2.count = 20;**

**// The value of 'count' will be 20 for both objects**

**System.out.println(obj1.count); // Output: 20**

**System.out.println(obj2.count); // Output: 20**

**5. Example: Counting the Number of Objects Created**

**Let’s look at an example where we use a static variable to keep track of the number of objects created for a class.**

**class MyClass {**

**// Static variable to count number of objects**

**static int objectCount = 0;**

**// Constructor**

**MyClass() {**

**// Incrementing the count every time an object is created**

**objectCount++;**

**}**

**// Method to display the object count**

**static void displayObjectCount() {**

**System.out.println("Number of objects created: " + objectCount);**

**}**

**}**

**public class Main {**

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

**// Creating objects**

**MyClass obj1 = new MyClass();**

**MyClass obj2 = new MyClass();**

**MyClass obj3 = new MyClass();**

**// Displaying the count of objects created**

**MyClass.displayObjectCount(); // Output: "Number of objects created: 3"**

**}**

**}**

**In this example:**

* **The static variable objectCount keeps track of the number of objects created.**
* **Since objectCount is static, all instances of MyClass share the same variable.**
* **Every time a new object is created, the constructor increments objectCount, and the total number of objects created can be retrieved using the displayObjectCount method.**

**Static Method in Java**

**A static method in Java is a method that belongs to the class rather than instances (objects) of the class. Like static variables, static methods can be called without creating an object of the class. They are commonly used to perform operations that are related to the class itself or to operate on static variables.**

**Key Characteristics of Static Methods:**

* **Belongs to the class: A static method is associated with the class, not any particular instance. It is defined using the static keyword.**
* **Access via class name: Static methods can be called directly using the class name, without the need to instantiate an object.**
* **No access to instance variables: Static methods cannot access instance variables or instance methods directly, because they are not tied to a specific object.**
* **Can access static variables and static methods: Static methods can only access static variables and other static methods directly.**
* **Cannot use this or super keywords: Since static methods are not associated with any particular instance, they cannot use this (which refers to the current object) or super (which refers to the parent class).**

**Declaration of a Static Method:**

**Static methods are declared using the static keyword.**

**class MyClass {**

**static void myStaticMethod() {**

**// Method code**

**}**

**}**

**Calling a Static Method:**

**Static methods can be called:**

1. **Using the class name:**

**MyClass.myStaticMethod();**

1. **Using an object (although not recommended):**

**MyClass obj = new MyClass();**

**obj.myStaticMethod(); // This works, but it's not the best practice.**

**Example: Static Method in Action**

**Let’s see an example where a static method is used to perform an operation related to the class, such as counting the number of objects created.**

**class MyClass {**

**// Static variable to keep track of object count**

**static int count = 0;**

**// Constructor**

**MyClass() {**

**count++; // Increment count when an object is created**

**}**

**// Static method to display the object count**

**static void displayCount() {**

**System.out.println("Total objects created: " + count);**

**}**

**}**

**public class Main {**

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

**MyClass obj1 = new MyClass();**

**MyClass obj2 = new MyClass();**

**// Calling the static method using the class name**

**MyClass.displayCount(); // Output: "Total objects created: 2"**

**}**

**}**

**Static Block in Java (Static Initialization Block)**

**A static block (also called a static initialization block) in Java is a block of code that is executed only once when the class is loaded into memory by the Java ClassLoader. It is mainly used to initialize static variables or execute logic that needs to run before the object creation or any static method is called.**

**Key Points about Static Blocks:**

* **Executed when the class is loaded: The static block is executed as soon as the class is loaded into memory, even before the main method or any constructor is executed.**
* **Runs only once: It runs only one time per class, no matter how many objects are created.**
* **Multiple static blocks: You can have multiple static blocks in a class, and they are executed in the order they appear.**
* **Used for static variable initialization: Static blocks are often used to initialize static variables or perform any setup that's required before the class is used.**

**Syntax of a Static Block:**

**class MyClass {**

**static {**

**// Static block code**

**}**

**}**

**Example: Static Block in Action**

**class MyClass {**

**static int a;**

**static int b;**

**// Static block to initialize static variables**

**static {**

**System.out.println("Static block executed");**

**a = 10;**

**b = 20;**

**}**

**// Constructor**

**MyClass() {**

**System.out.println("Constructor executed");**

**}**

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

**System.out.println("Main method started");**

**// Accessing static variables**

**System.out.println("a = " + a); // a = 10**

**System.out.println("b = " + b); // b = 20**

**// Creating an object of MyClass**

**MyClass obj = new MyClass();**

**}**

**}**

**Encapsulation in Java**

**Imagine a capsule (like a pill). The medicine inside the capsule is protected, and you can’t see or directly access it. You just take the capsule, and it works. Similarly, in Java, encapsulation hides the internal data of an object and only allows access through defined methods.**

**How Encapsulation Works:**

1. **Private Variables: You make the variables (data) of a class private so that no one outside the class can directly access or change them.**
2. **Public Methods: You provide public methods (getters and setters) to access and modify the private variables. These methods control how the data is accessed and changed.**

**Example:**

**class Person {**

**// Private variable, can't be accessed directly outside this class**

**private String name;**

**// Getter method to access the private variable**

**public String getName() {**

**return name;**

**}**

**// Setter method to modify the private variable**

**public void setName(String newName) {**

**name = newName;**

**}**

**}**

**public class Main {**

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

**Person person = new Person();**

**// Using setter to set the name**

**person.setName("Onkar");**

**// Using getter to get the name**

**System.out.println(person.getName()); // Output: Onkar**

**}**

**}**