**Class and Object in Java**

**1. Class:**

A **class** is like a blueprint or template for creating objects. It defines the properties (also known as **attributes** or **variables**) and actions (known as **methods** or **functions**) that objects created from the class will have.

Think of a **class** as a **recipe** for making objects. Just as a recipe defines the ingredients and steps to make a dish, a class defines what properties and actions an object will have.

* A **class** does not have any actual data. It only describes how an object will look and behave.
* A **class** contains variables (attributes) and methods (functions) that are common to all objects created from it.

**Example of a Class:**

class Car {

// Instance variables (properties of the car)

String color;

String model;

int year;

// Method (action) of the car

void drive() {

System.out.println("The car is driving.");

}

}

Here:

* The **Car class** has three properties: color, model, and year.
* It also has one **method**: drive(), which describes an action the car can perform.

**2. Object:**

An **object** is an instance of a **class**. It is a memory block from where we can store and retrieve data and execute methods etc. Once you have a class, you can create many **objects** based on that class. Each object will have its own unique data (for example, different color, model, or year for a Car object).

Think of an **object** as an **actual car** built from the blueprint (class) of the Car class. Each car object can have specific values for its properties (e.g., one car may be red, another may be blue).

* An **object** is created from a class using the new keyword.
* Each object can hold different values for the same properties, but they share the same structure and behaviors defined in the class.

**Example of Creating Objects:**

public class Main {

public static void main(String[] args) {

// Creating objects of the Car class

Car car1 = new Car(); // Object 1 of Car

Car car2 = new Car(); // Object 2 of Car

// Assigning values to the properties of car1

car1.color = "Red";

car1.model = "Tesla";

car1.year = 2022;

// Assigning values to the properties of car2

car2.color = "Blue";

car2.model = "Ford";

car2.year = 2020;

// Calling the drive method for both objects

car1.drive(); // Output: The car is driving.

car2.drive(); // Output: The car is driving.

// Displaying the details of the cars

System.out.println("Car 1: " + car1.color + " " + car1.model + " " + car1.year);

System.out.println("Car 2: " + car2.color + " " + car2.model + " " + car2.year);

}

}

**Output:**

The car is driving.

The car is driving.

Car 1: Red Tesla 2022

Car 2: Blue Ford 2020

**Key Differences Between Class and Object:**

| **Aspect** | **Class** | **Object** |
| --- | --- | --- |
| **Definition** | A class is a blueprint or template. | An object is an instance (or actual realization) of a class. |
| **Purpose** | Defines the properties and behaviors of objects. | Holds the actual data and interacts with other objects. |
| **Memory** | No memory is allocated for a class until objects are created. | Memory is allocated when an object is created. |
| **Example** | A class is like a "Car" blueprint. | An object is like a specific "Red Tesla 2022". |
| **Containment** | A class contains variables and methods. | An object contains actual values for variables and can call methods. |

**Real-World Analogy:**

* **Class**: Think of a **class** as a **blueprint** for a house. It contains details like the number of rooms, doors, windows, etc.
* **Object**: An **object** is like an **actual house** built from that blueprint. The house will have specific values for the rooms, such as 3 bedrooms, 2 bathrooms, etc.

**Important components of a Class**

**Instance variables** are the attributes that belong to an instance (or object) of the class. These variables are defined within a class but outside any method or constructor.

**A method** in Java defines a behavior or functionality that objects of a class can perform. Methods can accept parameters and return values.

**A constructor** is a special method used to initialize objects. It has the same name as the class and does not return a value. Constructors can be overloaded (having multiple constructors with different parameters).

**The this keyword** in Java refers to the current object. It is primarily used to differentiate between instance variables and parameters when they have the same name and also to call other constructors in the same class.

**Example Program 1**

// Define the 'Car' class

class Car {

// Instance variables

String make;

String model;

int year;

// Constructor to initialize the instance variables

Car(String make, String model, int year) {

// 'this' refers to the current object's instance variable

this.make = make;

this.model = model;

this.year = year;

}

// Method to display information about the car

void displayInfo() {

System.out.println("Car Make: " + make);

System.out.println("Car Model: " + model);

System.out.println("Car Year: " + year);

}

}

public class Main {

public static void main(String[] args) {

// Creating an object of the 'Car' class

Car myCar = new Car("Toyota", "Corolla", 2022);

// Calling the method to display information

myCar.displayInfo();

}

}

**Example Program 2**

class Book {

String title;

String author;

int publicationYear;

// Constructor with no arguments

Book() {

title = "Unknown Title";

author = "Unknown Author";

publicationYear = 0;

}

// Constructor with parameters

Book(String title, String author, int publicationYear) {

this.title = title;

this.author = author;

this.publicationYear = publicationYear;

}

void displayBookInfo() {

System.out.println("Title: " + title);

System.out.println("Author: " + author);

System.out.println("Publication Year: " + publicationYear);

}

public static void main(String[] args) {

// Using the constructor with no arguments

Book defaultBook = new Book();

defaultBook.displayBookInfo();

// Using the constructor with parameters

Book specificBook = new Book("1984", "George Orwell", 1949);

specificBook.displayBookInfo();

}

}

**Example Program 3**

class Calculator {

// Method without arguments and without return

void printGreeting() {

System.out.println("Welcome to the Calculator!");

}

// Method with arguments and without return

void printSum(int a, int b) {

System.out.println("Sum of " + a + " and " + b + " is: " + (a + b));

}

// Method with arguments and with return

int add(int a, int b) {

return a + b;

}

// Method without arguments and with return

int getRandomNumber() {

// Returning a random number between 1 and 100

return (int) (Math.random() \* 100) + 1;

}

public static void main(String[] args) {

// Creating an object of 'Calculator' class

Calculator calc = new Calculator();

// Calling the method without arguments and without return

calc.printGreeting();

// Calling the method with arguments and without return

calc.printSum(10, 20);

// Calling the method with arguments and with return

int sum = calc.add(5, 15);

System.out.println("Returned Sum: " + sum);

// Calling the method without arguments and with return

int randomNumber = calc.getRandomNumber();

System.out.println("Random Number: " + randomNumber);

}

}

**Static member**

In Java, static is a keyword used for memory management and can be applied to variables, methods, blocks, and nested classes. When a variable or method is declared as static, it means that the variable or method belongs to the class itself rather than to instances (objects) of the class. This can be useful for data or functionality that should be shared across all instances of a class.

**Static Variable**

A static variable is a variable that is declared with the static keyword. It is shared among all instances (objects) of the class. Instead of each object having its own copy of the variable, only one copy exists, and it is shared by all objects.

**Characteristics of Static Variables:**

* A static variable is initialized only once, at the start of the execution.
* It is common to all instances of the class, and its value is the same for every object.
* Static variables can be accessed directly by the class name or via an object reference.

**Static Method**

A static method is a method that is declared with the static keyword. Static methods belong to the class, not to any particular instance of the class. This means that a static method can be called without creating an object of the class.

**Characteristics of Static Methods:**

* Static methods can access only static variables and static methods of the class.
* Static methods cannot access instance variables or instance methods directly (they need an object to do so).
* They can be invoked using the class name or via an object reference.

class Counter {

// Static variable to count the number of times the method is called

static int count = 0;

// Static method to increment the count

static void incrementCount() {

count++;

System.out.println("Count: " + count);

}

public static void main(String[] args) {

// Calling the static method without creating an object

Counter.incrementCount(); // Output: Count: 1

Counter.incrementCount(); // Output: Count: 2

Counter.incrementCount(); // Output: Count: 3

}

}

### ****Key Differences Between Static Variables and Instance Variables****

| **Feature** | **Static Variable** | **Instance Variable** |
| --- | --- | --- |
| **Definition** | Belongs to the class. | Belongs to an object (instance). |
| **Memory Allocation** | One copy shared by all objects. | Each object has its own copy. |
| **Access** | Can be accessed using the class name or object. | Can only be accessed through an object. |
| **Lifetime** | Exists as long as the class is loaded in memory. | Exists as long as the object exists. |

### ****Key Differences Between Static Methods and Instance Methods****

| **Feature** | **Static Method** | **Instance Method** |
| --- | --- | --- |
| **Definition** | Belongs to the class. | Belongs to an object (instance). |
| **Access** | Can be accessed using the class name or object. | Can only be accessed through an object. |
| **Instance Variables** | Cannot directly access instance variables or methods. | Can access both instance and static variables. |
| **Use Case** | Useful for utility methods that don't depend on object state. | Used to operate on instance-specific data. |

**Example: Demonstrating Both Static and Instance**

Let’s consider a **Library Management System** that tracks the number of books in a library and details about each book (like title, author, and ISBN). The system will have the following features:

1. **Total number of books** in the library: This will be a **static variable** because it is common across all books and should be updated every time a new book is added to the library.
2. **Book-specific details** such as the title, author, and ISBN: These will be **instance variables** because they are unique to each book.
3. **Methods for adding books and displaying book details**: The method for adding books will update the static variable totalBooks, while the method for displaying details will operate on the instance variables of each book.

**Program Code**

class Book {

// Instance variables (unique to each book)

String title;

String author;

String isbn;

// Static variable (shared across all books)

static int totalBooks = 0; // Tracks the total number of books in the library

// Constructor to initialize book details

Book(String title, String author, String isbn) {

this.title = title;

this.author = author;

this.isbn = isbn;

totalBooks++; // Increment totalBooks whenever a new book is added

}

// Instance method to display book details

void displayBookDetails() {

System.out.println("Title: " + title);

System.out.println("Author: " + author);

System.out.println("ISBN: " + isbn);

System.out.println("Total Books in Library: " + totalBooks); // Access static variable

}

// Static method to display the total number of books

static void displayTotalBooks() {

System.out.println("Total Books in Library: " + totalBooks); // Access static variable

}

}

public class LibraryManagement {

public static void main(String[] args) {

// Creating book objects

Book book1 = new Book("The Great Gatsby", "F. Scott Fitzgerald", "9780743273565");

Book book2 = new Book("To Kill a Mockingbird", "Harper Lee", "9780061120084");

Book book3 = new Book("1984", "George Orwell", "9780451524935");

// Displaying details of individual books

book1.displayBookDetails();

System.out.println();

book2.displayBookDetails();

System.out.println();

book3.displayBookDetails();

System.out.println();

// Displaying total number of books in the library using static method

Book.displayTotalBooks();

}

}

**Static Block vs Instance Block in Java**

In Java, blocks are used to initialize values and set up states for variables. Both **static blocks** and **instance blocks** serve similar purposes but are used in different contexts.

**Static Block**

A **static block** is used for initializing static variables or performing tasks that need to be executed only once when the class is first loaded into memory. It is executed when the class is loaded by the JVM, and it is executed only once in the entire lifetime of the class.

**Characteristics of Static Block:**

* It is associated with the **class**, not with individual objects.
* It is executed **only once**, the first time the class is loaded into memory.
* It is used to initialize **static variables** or perform class-level initialization.
* The static block is executed before the **main method**.

**Syntax:**

static {

// Static initialization block

}

**Example:**

class Example {

// Static variable

static int count;

// Static block to initialize static variable

static {

count = 10;

System.out.println("Static block executed. Count initialized to " + count);

}

public static void main(String[] args) {

// This will call the static block if it has not been executed yet

System.out.println("Main method executed.");

}

}

**Output:**

Static block executed. Count initialized to 10

Main method executed.

In this example:

* The **static block** is executed first, initializing the static variable count to 10.
* The **main method** is executed afterward, but only once per class loading.

**Instance Block**

An **instance block** (also known as an **initializer block**) is used to initialize instance variables when an object is created. It runs every time a new instance of the class is created, before the constructor is called. It allows for common initialization code across all constructors.

**Characteristics of Instance Block:**

* It is associated with **objects** (instances) of the class.
* It is executed every time a **new object** of the class is created.
* It runs **before the constructor** when an object is initialized.
* Multiple instance blocks can exist in a class.

**Syntax:**

{

// Instance initialization block

}

**Example:**

class Example {

// Instance variable

int number;

// Instance block to initialize instance variable

{

number = 5;

System.out.println("Instance block executed. Number initialized to " + number);

}

// Constructor

Example() {

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

}

public static void main(String[] args) {

// Creating objects to trigger instance block

Example obj1 = new Example();

Example obj2 = new Example();

}

}

**Output:**

Instance block executed. Number initialized to 5

Constructor executed.

Instance block executed. Number initialized to 5

Constructor executed.

In this example:

* The **instance block** is executed every time an object of the class is created, before the constructor.
* When two objects obj1 and obj2 are created, the **instance block** is executed twice (once for each object).

**Key Differences Between Static and Instance Blocks**

| **Feature** | **Static Block** | **Instance Block** |
| --- | --- | --- |
| **Execution Timing** | Executed when the class is loaded into memory, **once** per class. | Executed every time a new instance is created, **before the constructor**. |
| **Usage** | Used for **static initialization**, initializing static variables, or performing class-level initialization. | Used for **instance initialization**, initializing instance variables across constructors. |
| **Access to Variables** | Can only access **static** variables and methods. | Can access both **instance** and **static** variables and methods. |
| **When to Use** | Use when you need to perform operations that should only happen once, such as setting up static fields. | Use when you need to initialize instance variables or execute common logic for all constructors. |
| **Execution Order** | Executed **before the main method**. | Executed **before the constructor** for each object created. |

**Example: Demonstrating Both Static and Instance Blocks**

Let’s consider a simple **Online Store System** that keeps track of products and their information. We’ll use **static blocks** and **instance blocks** to simulate different aspects of the system.

**Scenario:**

1. **Static Block**:
   * The online store has a **static discount code** that applies to all products when a promotion is running. This discount is applied to all products in the store and needs to be initialized once when the store starts.
   * We will use a **static block** to initialize this **global discount** code and set it at the start.
2. **Instance Block**:
   * Each product has individual properties, such as **name**, **price**, and **quantity**. These properties are unique to each product.
   * The store’s products have a **common initialization** logic (like setting default quantity), so we’ll use an **instance block** to handle this setup for every new product.

class Product {

// Static variable to store the common discount code

static String discountCode;

// Instance variables to store product details

String name;

double price;

int quantity;

// Static block to initialize the discount code

static {

discountCode = "SUMMER2025"; // Discount code that applies to all products

System.out.println("Static Block: Discount code initialized: " + discountCode);

}

// Instance block to initialize product-specific details

{

quantity = 100; // Default quantity for each product

System.out.println("Instance Block: Quantity initialized to " + quantity);

}

// Constructor to initialize product name and price

Product(String name, double price) {

this.name = name;

this.price = price;

System.out.println("Constructor: Product " + name + " with price " + price + " initialized.");

}

// Instance method to display product details

void displayProductDetails() {

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

System.out.println("Price: " + price);

System.out.println("Quantity in Stock: " + quantity);

System.out.println("Discount Code: " + discountCode); // Accessing static variable

}

// Static method to display the discount code for the store

static void displayDiscountCode() {

System.out.println("Current Discount Code: " + discountCode); // Accessing static variable

}

}

public class OnlineStore {

public static void main(String[] args) {

// Creating the first product

Product product1 = new Product("Laptop", 800);

product1.displayProductDetails();

System.out.println();

// Creating the second product

Product product2 = new Product("Smartphone", 600);

product2.displayProductDetails();

System.out.println();

// Displaying the static discount code using the static method

Product.displayDiscountCode();

}

}