**OBJECT:**

* An object is an instance of a class and has a unique identity.
* The identity of an object distinguishes it from other objects.
* Classes and objects are closely linked to each other.
* While an object has a unique identity, a class is an abstraction of the common properties of various objects.

**Declaring an object**

**Syntax:**

class\_name object\_name;

**Note:**

* When we declare an object, the memory is not allocated to it.

**Declare and instantiate an object**

**Syntax:**

class\_name object\_name = new class\_name();

**Note:**

* To allocate memory to the object, we need to instantiate the object by using the ***new*** keyword.
* The ***new*** keyword allocates memory to an object.
* It also returns a reference to that memory location in the object variable.

**Example:**

public class Demo{

public static void main(String[] args){

Demo obj = new Demo(); //object

}

}

**Scanner class:**

* After importing the ***Scanner class***, we need to create its object.
* It uses ***java.util*** package.

**Syntax:**

Scanner <objectname> = new Scanner(System.in);

**Note:**

* The predefined methods of the ***Scanner class***, such as **nextInt()** and **nextLine()** can be used to read an integer or a string value.

**Example:**

String option = <objectname >.nextLine();

**Difference between next(), nextInt() and nextLine() in java?**

**Solution:**

* In Java, the Scanner class provides methods for reading input from various sources, such as the keyboard.
* Among these methods, **next()**, **nextInt()**, and **nextLine()** are commonly used to read different types of input.

**next():**

* **Purpose:** Reads the next token of input as a String.
* **Behavior:** It reads input until it encounters a delimiter (default is whitespace).
* **Usage:** Useful for reading a single word or token at a time.

**Example:**

Scanner scanner = new Scanner(System.in);

String word = scanner.next();

**Note:**

* If the input is "Hello World", scanner.next() will return "Hello".

**nextInt():**

* **Purpose:** Reads the next token of input as an int.
* **Behavior:** It parses the token as an integer. If the input is not a valid integer, it throws an InputMismatchException.
* **Usage:** Useful for reading integer values from the input.

**Example:**

Scanner scanner = new Scanner(System.in);

int number = scanner.nextInt();

**Note:**

* If the input is "42", scanner.nextInt() will return 42.

**nextLine():**

* **Purpose:** Reads the entire line of input as a String.
* **Behavior:** It reads the input until it encounters a newline character (the end of the line). It consumes the newline character at the end of the line.
* **Usage:** Useful for reading a whole line of text or for input that may include spaces.

**Example:**

Scanner scanner = new Scanner(System.in);

String line = scanner.nextLine();

**Note:**

* If the input is "Hello World", scanner.nextLine() will return "Hello World".

**Key Differences:**

**Input Handling:**

* **next()** reads the next token, stopping at whitespace.
* **nextInt()** reads the next token and parses it as an integer.
* **nextLine()** reads the entire line, including spaces.

**Data Type:**

* **next()** returns a String.
* **nextInt()** returns an int.
* **nextLine()** returns a String.

**Handling Newlines:**

* **next()** and **nextInt()** do not consume the newline character after the input. This can cause issues if you try to read a line immediately after using these methods.
* **nextLine()** consumes the newline character, making it suitable for reading whole lines.

**Example of Handling Mixed Input:**

* When using next(), nextInt(), and nextLine() together, you might need to handle leftover newline characters to avoid issues.

**Example:**

Scanner scanner = new Scanner(System.in);

System.out.print("Enter an integer: ");

int number = scanner.nextInt(); // Reads integer, but leaves newline character

scanner.nextLine(); // Consumes the leftover newline

System.out.print("Enter a line of text: ");

String line = scanner.nextLine(); // Now reads the full line properly

**Note:**

* In this example, **scanner.nextLine()** is used after **scanner.nextInt()** to consume the newline character that remains after reading the integer. This prevents issues with subsequent **nextLine()** calls.

**PACKAGE:**

* A package is a collection of classes.
* A package provides the space essentially used to organize classes that are related to each other.
* We can create multiple packages to organize multiple categories of classes.

**Syntax:**

package <package\_name>

**Note:**

* Packages are represented and stored by Java as directories in the file system.

**ACCESSING CLASS MEMBERS**

* The class members describe the characteristics and behavior of an object.
* We may need to hide or protect certain class members from other classes in a java application.
* In other words, we may want to restrict access to sensitive data that belongs to a class so that it is not modified.
* The access specifiers and modifiers provided in java programming language are used to identify the part of the class.
* Such a data members and methods that needs to be accessed by other class objects.

**Using Objects:**

* Objects are used to access the members of a class.
* We can access the data member of a class by specifying syntax below.

**Syntax:**

object\_name.data\_member\_name

**Example:**

public class Demo{

int rollno = 28;

public static void main(String[] args){

Demo obj = new Demo(); //object

System.out.println(obj.rollno); //Accessing class member

}

}

**Note:**

* Whenever we execute a java application, the ***main()*** method is the first to be executed by the **JVM**.

**Method of a class accessed by using object**

**Example:**

public class Demo{

int rollno;

void show()

{

System.out.println(rollno);

}

public static void main(String[] args){

Demo obj = new Demo(); //object

obj.show();

}

}

**Static:**

* In Java, the static keyword is used to declare class-level members—fields, methods, and blocks—that belong to the class itself rather than to any specific instance of the class.
* This means that static members are shared among all instances of the class.

**Use of static in Java**

* **Static Variables:** Also known as class variables, these are shared among all instances of the class. They are used to store data that is common to all objects.
* **Static Methods:** These methods can be called on the class itself rather than on instances of the class. They can only access static variables and call other static methods.
* **Static Blocks:** Used for initializing static variables. They run once when the class is loaded.
* **Static Classes:** Nested classes can be declared as static. A static nested class can be instantiated without an instance of the outer class.

**Example:**

public class Counter {

// Static variable

private static int count = 0;

// Constructor

public Counter() {

count++;

}

// Static method

public static int getCount() {

return count;

}

// Static block

static {

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

}

// Non-static method

public void display() {

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

}

// Static nested class

public static class StaticNestedClass {

public void show() {

System.out.println("Static nested class method.");

}

}

public static void main(String[] args) {

Counter c1 = new Counter();

Counter c2 = new Counter();

System.out.println("Total count: " + Counter.getCount()); // Access static method

c1.display(); // Access static variable from non-static method

Counter.StaticNestedClass nested = new Counter.StaticNestedClass();

nested.show(); // Access static nested class

}

}

**Using Static Variables in Java**

**Example:**

public class Counter {

// Static variable to keep track of the number of instances

private static int count = 0;

// Constructor

public Counter() {

count++; // Increment static variable each time a new object is created

}

// Static method to get the value of the count

public static int getCount() {

return count;

}

// Instance method to display the count

public void display() {

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

}

public static void main(String[] args) {

// Create instances of Counter

Counter c1 = new Counter();

Counter c2 = new Counter();

Counter c3 = new Counter();

// Display the count using a static method

System.out.println("Total number of Counter instances: " + Counter.getCount());

// Display the count using instance method (same result as static method)

c1.display();

c2.display();

c3.display();

}

}

**Interview Questions**

**What is the difference between static and instance methods?**

**Answer:**

Static methods belong to the class, not instances. They can be called without creating an object of the class and can only access static variables and methods. Instance methods belong to a specific object and can access both instance and static variables and methods.

**Can a static method access instance variables?**

**Answer:**

No, a static method cannot directly access instance variables because instance variables are associated with specific instances, while static methods belong to the class itself.

**What is the purpose of a static block?**

**Answer:**

A static block is used to initialize static variables. It runs only once when the class is first loaded into memory. It’s useful for performing setup tasks that need to be done only once.

**How does the static keyword affect memory management?**

**Answer:**

Static variables and methods are stored in the method area of the JVM and are shared across all instances of the class. This can save memory when you have data or methods that need to be common to all instances.

**What are static classes and how do they differ from non-static classes?**

**Answer:**

Static classes in Java are nested classes that do not require an instance of the outer class to be instantiated. They can access only the static members of the outer class. Non-static classes (or inner classes) can access both static and instance members of the outer class and require an instance of the outer class to be instantiated.

**Can a static method be overridden?**

**Answer:**

No, static methods cannot be overridden in the same way instance methods can. They can be hidden by defining another static method with the same name in the subclass. However, this is not true polymorphism, as static methods are bound at compile-time.