Static-NON ACCESS MODIFIERS

Static in **Java** is a non-access modifier used **for** methods, variables, blocks, and nested classes. It is primarily associated with the **class rather** than instances of the class, meaning that the static keyword creates a class-level member. This allows **static** members to be accessed directly by the **class name** without needing to instantiate the class.

1. Static Variables:

Class Variables: Static variables are also known as class variables because they are associated with the class and common to all its instances. They are declared by using the keyword static.
Initialization: They are initialized only once, at the start of the execution. The static variable will get memory only once, in the **class area** at the time of class **loading**.
Example Usage: They can be used to refer to the common property of all objects (which is not unique for each object), **e.g., the company name of employees, college name of students, etc.**
Memory Efficiency: Using static variables makes your program more memory efficient (it saves memory). Static variables are stored in the static memory, and they are part of the class and are created when the class is loaded by the JVM (Java Virtual Machine).

Key Characteristics of Static Variables:

- **Shared Among Instances**: All instances of the class share the same static variable. If changes are made to the static variable by one instance, all other instances will see the effect of the change.
- **Memory Efficient**: Static variables are allocated **memory only on**ce in the class area at the time of class loading, which makes the program more memory efficient.
- **Class Level Access**: Static variables can be accessed directly by the class name, without needing to instantiate an object of the class.

Example 1 Static Variable:

Lets Consider a simple example where we have a class **Employee** with a static variable **companyName**. Since the company name is common to all employees, it makes sense to make it a static variable. So let's start.

```
public class Employee {
    // Static variable
    public static String companyName = "LinkedinCommunity";

    // Instance variables
    private String name;
    private int id;

    // Constructor
    public Employee(String name, int id) {
        this.name = name;
        this.id = id;
    }

    // Method to display employee details
    public void displayDetails() {
        System.out.println("ID: " + id + ", Name: " + name + ", Company: " + companyName);
    }

    // Main method
    public static void main(String[] args) {
        Employee emp1 = new Employee("Alice", 1);
        Employee emp2 = new Employee("Bob", 2);

        // Displaying details of employees
        emp1.displayDetails();
        emp2.displayDetails();

        // Changing the company name
        Employee.companyName = "LinkedinCommunity Learning";

        // Displaying details again to see the effect of change
        emp1.displayDetails();
        emp2.displayDetails();
    }
}
```

In this example:

- **companyName** is a static variable associated with the class **Employee** and shared by all its instances.
- Each **Employee** object has its **own name** and id (instance variables) but shares a **single copy** of the static variable **companyName**.
- When we change the companyName to " **LinkedinCommunity Learning** ", this change is reflected for both emp1 and emp2, as they share the same companyName.
- The displayDetails method shows how to access the static variable. It can be accessed directly inside the class **without** using the class **name**, but from outside the class, you should use **ClassName.variableName**, e.g., **Employee.companyName**.

Usage Considerations:

- Use static variables for properties that are common to all instances of a class, such as company names, counter variables to keep track of the number of objects created, or constants.
- Remember that since static variables are shared among all instances of a class, changes made by one instance will affect all other instances.
- Access static variables using the class name from outside the class to highlight that they belong to the class level, not to any particular instance.

Another Example 2 for Static Variable:

```
public class MyClass {

    // Static variable
    static int staticCounter = 0;

    // Instance variable
    int instanceCounter = 0;

    public MyClass() {
        staticCounter++;  // Increment the static counter
        instanceCounter++;  // Increment the instance counter
    }

public static void main(String[] args) {
    MyClass obj1 = new MyClass();
    MyClass obj2 = new MyClass();
    MyClass obj3 = new MyClass();

    // Accessing the static variable through the class name
    System.out.println("Static Counter: " + MyClass.staticCounter);  // Output: 3

    // Instance variables are accessed through the instances, as before
    System.out.println("Instance Counter for obj1: " + obj1.instanceCounter);  // Output: 1
    System.out.println("Instance Counter for obj2: " + obj2.instanceCounter);  // Output: 1
    System.out.println("Instance Counter for obj3: " + obj3.instanceCounter);  // Output: 1
}

}
```

This approach clarifies the nature of **staticCounter** as a **class variable**, not an **instance variable**, and aligns with Java best practices for accessing static members. In contrast, the **instanceCounter** variable is unique to each instance and is independent of the static variable's state.

POINTS TO REMEMBER

1. Single Copy:

Static variables are stored in the static memory, and there is only a single copy of each static variable, regardless of how many instances of the class exist. This means all instances of the class **share the same static variable.**

2. Class Level Scope:

Since static variables are associated with a class rather than any individual instance, they can be accessed directly by the class name and don't require an object to be accessed (though they can be accessed via objects, this is not recommended due to readability concerns).

3. Initialization:

Static variables are initialized when the class is loaded by the Java Virtual Machine (JVM). They can be initialized during declaration or within a static block in the class. Static blocks are executed when the class is first loaded, making them ideal for complex initialization sequences.

4. Default Values:

If a static variable is not explicitly initialized, it will be given a default value based on its data type (null for objects, 0 for numeric types, false for boolean, etc.), similar to instance variables.

5. Memory Efficiency:

Using static variables can make your program more memory-efficient since they do not belong to any instance; each class only has one copy, regardless of the number of instances.

6. Lifetime:

The lifetime of static variables spans the entire execution of the program. They are created when the class is loaded and destroyed when the program terminates or the class is unloaded.

2. Static Methods:

Static methods in Java, also known as class methods, are defined using the static modifier. These methods belong to the class rather than any particular instance of the class. This means they can be called without creating an object of the class. Here's a detailed look at static methods:

Key Characteristics of Static Methods:

1. **Class Level**: Static methods are associated with the class itself, not with any specific object created from the class. As a result, they can be invoked directly using the class name.
2. **Access to Static Members Only**: Within static methods, you can only directly access other static methods and static variables. They cannot access instance methods and instance variables directly because they do not operate on instances of the class.
3. **No this or super**: Since static methods do not belong to a specific instance, they cannot use **this** or **super** keywords, as those are related to instance-level context.
4. **Usage**: Static methods are commonly used for utility or helper functions that don't require any object state. Since they don't require an instance to be invoked, they're also used in scenarios where methods need to be accessed without creating an object, like in the **main** method which serves as the entry point for Java applications.

When to Use Static Methods:

- **Utility Functions**: When a function performs a general-purpose task that doesn't depend on instance variables, it should be declared static. For example, a method to calculate the square of a number could be static since it doesn't rely on any object state.
- **Factory Methods**: Static methods are often used for factory method patterns, where a method is responsible for creating instances of a class.
- **Main Method**: The **main** method in Java is static, allowing the JVM to invoke it without creating an instance of the class.

Example Code Static Method:

Consider a class **Calculator** with a static method that calculates the sum of two numbers:

```
public class Calculator {

    // Static method to add two numbers
    public static int add(int num1, int num2) {
        return num1 + num2;
    }

    // Instance method
    public int multiply(int num1, int num2) {
        return num1 * num2;
    }

    public static void main(String[] args) {
        // Calling the static method without creating an instance
        int sum = Calculator.add(5, 3);
        System.out.println("Sum: " + sum);

        // Static methods can also be called using an object, but it's not recommended
        Calculator calc = new Calculator();
        sum = calc.add(10, 20);  // Not recommended
        System.out.println("Sum: " + sum);

        // Calling an instance method requires an instance of the class
        int product = calc.multiply(5, 4);
        System.out.println("Product: " + product);
    }
}
```

In this example, the **add** method is **static** and can be called **without** creating an instance of the **Calculator** class. The **multiply** method, however, is an **instance** method and **requires an object** to be called.

Best Practices:

1. Use Static Methods When the Method Does Not Depend on Instance Variables.

```
public class Utility {

    // Static method that calculates the square of a number
    public static int square(int number) {
        return number * number;
    }

    public static void main(String[] args) {
        // Call the static method without an object
        int result = Utility.square(5);
        System.out.println("Square: " + result);
    }
}
```

2. Avoid Using Static Methods to Access or Modify the State of an Object

Avoid using static methods to access or modify the state of an object; they should be used for operations that are independent of object state.

```
public class Counter {

    private int count = 0; // Instance variable

    // Non-static method to increment the counter
    public void increment() {
        count++;
    }

    public int getCount() {
        return count;
    }

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

```
        Counter counter = new Counter();
        counter.increment(); // Correct way to modify instance state
        System.out.println("Count: " + counter.getCount());
    }
}
```

In this example, the increment method is not static because it modifies the count instance variable. Making increment a static method would be incorrect as static methods should not access or modify the instance state.

Instance Members:

- **Instance Variables**: These are specific to each instance of a class. Each object has its own copy of an instance variable.
- **Instance Methods**: These operate on the instance variables of the object that invokes the method. They require an instance of the class to be called because they can access and modify the instance variables that belong to that specific object.

Static Members:

- **Static Variables**: These are class-level variables. There is only one copy of a static variable, and it is shared among all instances of the class.
- **Static Methods**: These belong to the class rather than any particular instance. They can be called without creating an instance of the class. Because they don't operate on an instance, they cannot directly access or modify instance variables.

Why Making increment method Static Would Be Incorrect:

1. **Access to Instance Variables**: The **increment** method modifies **count**, which is an instance variable. A static method cannot directly access instance variables because it does not have access to an instance of the class (i.e., there's no **this** context). Instance variables are tied to specific objects, but a static method is not tied to any object.

2. **Object State Modification**: The purpose of the **increment** method is to modify the state (**count**) of a specific **Counter** object. Making this method static would remove its ability to operate on an object's state, contradicting the method's intended functionality.

3. **Conceptual Mismatch**: Static methods are meant for operations that are independent of the state of specific instances of a class. The operation performed by **increment** is inherently tied to the state of a particular **Counter** object, making it conceptually inappropriate to be static.

Example of Incorrect Static `increment` Method:

```
public class Counter {
    private int count = 0; // Instance variable

    // Incorrect use of static with an instance variable
    public static void increment() {
        count++;  // Compilation error: Cannot make a static reference to the non-static field count
    }
}
```

If we tried to make increment static and access the count variable, we'd encounter a compilation error because static methods can't access instance variables like count

3. Be Cautious with Static Methods in Subclasses

- Be cautious with static methods in subclasses: they don't override methods from the superclass since static methods are not part of the instance of a class. This can lead to unexpected behavior if you're not clear about which method is being called.

Example 1:

```
class Parent {
    // Static method in the base class
    static void display() {
        System.out.println("Display in Parent");
    }
}

class Child extends Parent {
    // This does not override the display in Parent, it's a separate method
    static void display() {
        System.out.println("Display in Child");
    }
}
```

```
public class TestStaticMethods {

    public static void main(String[] args) {
        Parent parent = new Parent();
        Parent childAsParent = new Child();

        parent.display(); // Calls Parent's display
        childAsParent.display(); // Also calls Parent's display because the method
is static

        // To call Child's display, we need a reference of type Child
        Child child = new Child();
        child.display(); // Calls Child's display
    }
}
```

In this example, both the Parent and Child classes have a static method named display. Even though Child extends Parent, Child's display method does not override Parent's display method because static methods are associated with the class, not instances. When you call display on a Parent reference pointing to a Child object, it calls the Parent's display method, illustrating the need for caution with static methods in subclasses.

Why the Above Example 1 Might Be Confusing:

- The example shows accessing static methods through instances (**parent.display()** and **child.display()**) to illustrate how static methods are associated with the class type of the reference, not the class type of the object. This can be confusing because it suggests that creating instances is necessary to access static methods, which is not the case.

- The example intends to demonstrate that even when you have a reference of type **Parent** pointing to an object of type **Child** (**Parent childAsParent = new Child();**), calling a static method (**childAsParent.display();**) invokes the method defined in the reference's compile-time type (**Parent**), not the actual object's runtime type (**Child**). This is a unique behavior of static methods and differs from how instance methods behave with polymorphism.

Clarified Example 1:

To avoid confusion, it's better to access static methods using the class name, as shown below:

```
public class TestStaticMethods {

    public static void main(String[] args) {
        // Correct way to call static methods
        Parent.display(); // Calls Parent's display
        Child.display(); // Calls Child's display

        // Although possible, it's discouraged to call static methods through
instances
        // Parent parent = new Parent();
        // parent.display(); // Not recommended
        // Child child = new Child();
        // child.display(); // Not recommended
    }
}
```

In this revised example, static methods are accessed in the recommended way, using the class names directly. This approach clearly communicates that the methods are static and avoids the potential confusion associated with accessing static methods through instances.

Example 2:

```
class Parent {
    // Static method in the parent class
    static void display() {
        System.out.println("Display in Parent");
    }
}

class Child extends Parent {
    // Child class can access the parent's static method

    static void callParentDisplay() {
        // Calling the static method of the parent class
        Parent.display();
    }
}

public class TestStaticMethodCall {
    public static void main(String[] args) {
        // Calling the parent's static method from the child class without creating an instance
        Child.callParentDisplay(); // This will output "Display in Parent"
    }
}
```

In this example, the Child class has a static method **callParentDisplay** that calls the Parent class's static method display using Parent.display(). When **Child.callParentDisplay()** is called in the main method, it internally calls Parent.display(), demonstrating that the child class can access static methods of its parent class directly through the class name, without the need for creating an instance.

MORE CODE EXAMPLE FOR STATIC METHODs

Example 3.

```
class Device {
    static void control() {
        System.out.println("Controlling Device");
    }
}

class TV extends Device {
    static void control() {
        System.out.println("Controlling TV");
    }
}

class DVDPlayer extends Device {
    static void control() {
        System.out.println("Controlling DVD Player");
    }
}

public class UniversalRemote {
    public static void main(String[] args) {
        Device myRemote = new TV();  // Compile-time: Remote is set to control
TV
        myRemote.control();  // Output: "Controlling Device"

        myRemote = new DVDPlayer();  // Compile-time: Remote is set to control
DVD Player
        myRemote.control();  // Output: "Controlling Device"
    }
}
```

Above Code Explanation:

This code demonstrates how static methods behave in an inheritance hierarchy in Java, particularly **focusing on method hiding and static method binding at compile time**. Let's break down the key concepts and components of the code to understand it better:

Classes and Static Methods

1. **Device Class**: This is the base class with a static method **control()** that prints "Controlling Device".
2. **TV Class**: This is a subclass of **Device** that hides the **control()** method of **Device** with its own static method that prints "Controlling TV". It doesn't override it because static methods cannot be overridden; they can only be hidden in subclasses.
3. **DVDPlayer Class**: Another subclass of **Device**, similar to **TV**, it hides the **control()** method with its version, printing "Controlling DVD Player".

Main Method and Static Method Behavior

In the **main** method of the **UniversalRemote** class, the code demonstrates how static methods are invoked based on the reference type, not the object type.

First Part:

```
Device myRemote = new TV();
myRemote.control();  // Output: "Controlling Device"
```

Here, **myRemote** is a reference variable of type **Device**, but it points to an object of type **TV**. When **myRemote.control()** is called, it invokes the **control** method of the **Device** class, not **TV**, despite **myRemote** pointing to a **TV** object. This is because static method calls are resolved at compile time based on the reference type (**Device** in this case), not the runtime type of the object (**TV**).

Second Part:

```
myRemote = new DVDPlayer();
myRemote.control();  // Output: "Controlling Device"
```

Now, the myRemote reference is reassigned to point to a DVDPlayer object. However, when myRemote.control() is called, it still invokes the control method of the Device class. Again, this is because myRemote is a Device type reference, and static methods are bound based on the reference type at compile time, not the runtime type.

Key Takeaways For Static Methods:

- **Static Method Binding**: Static methods are bound at compile time based on the reference type. Even if the reference points to a subclass object at runtime, the static method of the reference type (declared type) is called.
- **Method Hiding**: When a subclass defines a static method with the same signature as a static method in the superclass, the subclass's method hides the superclass's method. This is not the same as overriding, where a subclass's instance method can override a superclass's instance method.
- **Best Practices**: **Accessing static methods using a reference variable, as shown in this example, can be confusing and is generally discouraged**. It's clearer and more conventional to call static methods using the class they belong to, like **Device.control()** or **TV.control()**, to avoid ambiguity.

This example is illustrative of some of the subtleties of Java's handling of static methods in an inheritance context, highlighting the importance of understanding how static methods are resolved and the difference between method hiding and overriding.

CONCEPT OF REFERENCE

Imagine you have a remote control for a drone. The remote control itself doesn't fly, but it controls the drone that does. In this analogy:

- The **drone** is like an **object** in Java, which is an instance of a class.
- The **remote control** is like a **reference** in Java, which points to or refers to an object.

When you create an object in Java using the **new** keyword, you are essentially creating a new "drone" (object). To control or access this drone, you need a remote control (reference). Here's how it looks in code:

```
Drone myDrone = new Drone();
```

- **Drone** is the class, like the model or type of drone you have.
- **myDrone** is the reference, like your remote control. It's the name you use to access the object.
- **new Drone()** is the actual object, like the physical drone being created and ready to fly.

So, the reference (**myDrone**) points to an actual object in memory (an instance of the **Drone** class). When you want the drone to fly, you use the remote control (**myDrone.fly()**). Similarly, in Java, when you want to call a method on an object, you use the reference.

Key Points:

- A **reference** is a variable that holds the memory address of an object. It points to the actual object in the computer's memory.
- You use this reference to access the object's methods and attributes.
- A reference can be reassigned to point to another object, or it can be set to **null**, meaning it points to no object.

A "reference" is similar to a pointer in other programming languages, but with a higher level of abstraction. It's a variable that holds the memory address of an

object, not the object itself. When you create an object in Java, you're creating it in the heap memory, and the variable you assign it to is a reference to that object.

```
ClassName objectReference = new ClassName();
```

- **ClassName** is the type of the object you're creating, and it determines what kind of data (object) the reference will point to.
- **objectReference** is the reference variable. It stores the memory address where the actual **ClassName** object is located in the heap. You use this reference to interact with the object.
- **new ClassName()** is the part of the code that actually creates the object in memory (heap), and the memory address of this new object is assigned to **objectReference**.

The reference variable (**objectReference**) acts like a remote control for the object it points to, allowing you to access and manipulate the object's fields and methods.

3. Static Classes:

- **Nested Classes Only**: In Java, only nested classes can be static. A static nested class is behaviorally a top-level class that has been nested in another class for packaging convenience.
- **Access**: Static nested classes can access all static members of its enclosing class, including private ones, but it cannot access non-static members.

Key Characteristics of Static Nested Classes:

1. **Nested Classes Only**: Static classes can only be nested within other classes. They cannot be top-level classes.
2. **Behavior**: They behave like top-level classes but are packaged within another class for logical grouping or to encapsulate them within the outer class for better organization.
3. **Access to Enclosing Class Members**: **Static nested classes can access the static members of the enclosing class, including private static members**. However, they cannot access the instance variables or instance methods of the enclosing class directly, since they do not belong to an instance of the enclosing class.

Example of Static Nested Class:

Consider a scenario where we have an **OuterClass** that represents a type of computer, and within it, we have a static nested class **CPU** that represents a component of the computer.

```
public class OuterClass {
    private static String computerType = "Gaming Computer";
    private int id;

    // Static nested class
    public static class CPU {
        public void displayType() {
            // Can access the static variable of the outer class
            System.out.println("The computer type is: " + computerType);

            // Cannot access the instance variable 'id' of the outer class directly
            // System.out.println("Outer class ID: " + id); // This would be an error
        }
    }

    public void setId(int id) {
        this.id = id;
    }

    public static void main(String[] args) {
        OuterClass outer = new OuterClass();
        outer.setId(101);

        // Creating an instance of the static nested class
        OuterClass.CPU cpu = new OuterClass.CPU();
        cpu.displayType();
    }
}
```

In the Above example:

- **OuterClass** represents the outer class that has a static variable **computerType** and an instance variable **id**.
- **CPU** is a static nested class defined within **OuterClass**. It has a method **displayType** that can access the static variable **computerType** of **OuterClass** but cannot access the instance variable **id** directly because **CPU** does not have a reference to an instance of **OuterClass**.

- In the **main** method, an instance of **CPU** is created without needing an instance of **OuterClass**. The **displayType** method is then called to demonstrate access to the outer class's static variable.

Example 2:

```
public class OuterClass {

    // Static variable of the outer class
    private static String staticVariable = "Outer Static Variable";

    // Static nested class
    public static class NestedClass {
        public void display() {
            // Can access the static variable of the outer class
            System.out.println("Message from the nested class: " + staticVariable);
        }
    }

    public static void main(String[] args) {
        // Creating an instance of the static nested class
        OuterClass.NestedClass nestedObject = new OuterClass.NestedClass();

        // Calling a method of the nested class
        nestedObject.display();  // Outputs: Message from the nested class: Outer
Static Variable
    }
}
```

Key Points of the above Example 2:

- **Nested Class Declaration**: **NestedClass** is declared static and is nested within **OuterClass**. It behaves like any other top-level class but is packaged within **OuterClass** for better organization.

- **Access to Static Members**: **NestedClass** has access to the static members of **OuterClass** (like **staticVariable**), even if they are private. This is shown in the **display** method, where **NestedClass** accesses and prints **OuterClass**'s **staticVariable**.

- **Instantiation**: An instance of a static nested class is created without needing an instance of the outer class. In the **main** method, **new OuterClass.NestedClass()** creates an instance of **NestedClass**.

- **Method Call**: The **display** method of **NestedClass** is called on its instance, showing how a static nested class can be used similarly to a top-level class.

NOTE: **This documentation has focused exclusively on the nuances of the "static" non-access modifier in Java.** There are other significant non-access modifiers, such as "final," "abstract," "synchronized," and "volatile," which also play crucial roles in Java programming. Each of these keywords encompasses a breadth of functionality and intricacies, similar to "static." We plan to explore each of these modifiers in detail in our upcoming discussions, delving into their unique attributes and applications within Java.

final," "abstract," "synchronized," and "volatile," in upcoming days. Happy Learning.