# Static Keyboard

1. Why do we need static keyword in Java Explain with an example?

Ans1.

In Java, the `static` keyword is used for memory management and to create class-level methods and variables. When a member (variable or method) is declared as `static`, it belongs to the class rather than any specific instance of the class. This means:

1. **Static Variables**: These are shared among all instances of the class.
2. **Static Methods**: These can be called without creating an instance of the class.
3. **Static Blocks**: These are used to initialize static variables.

### Example 1: Static Variable

```
class Student {
    static String schoolName = "ABC School"; // Shared among all instances
    String name;

    Student(String name) {
        this.name = name;
    }

    void display() {
        System.out.println(name + " studies in " + schoolName);
    }
}

public class StaticExample {
    public static void main(String[] args) {
        Student s1 = new Student("Alice");
        Student s2 = new Student("Bob");

        s1.display();
```

```
        s2.display();

        // Changing static variable
        Student.schoolName = "XYZ School";

        s1.display();
        s2.display();
    }
}
```

Output:

```
Alice studies in ABC School
Bob studies in ABC School
Alice studies in XYZ School
Bob studies in XYZ School
```

Example 2: Static Method

```
class MathUtil {
    static int square(int x) {
        return x * x;
    }
}

public class StaticMethodExample {
    public static void main(String[] args) {
        int result = MathUtil.square(5); // No need to create an instance
        System.out.println("Square: " + result);
    }
}
```

Output:

```
Square: 25
```

Example 3: Static Block

```
class StaticBlockExample {
    static int num;

    // Static block executes before main()
```

```
    static {
        num = 100;
        System.out.println("Static block executed!");
    }

    public static void main(String[] args) {
        System.out.println("Value of num: " + num);
    }
}
```

Output:

```
Static block executed!
Value of num: 100
```

2. What is class loading and how does the Java program actually executes .

Ans2. **Class Loading & Java Program Execution Flow**

Java follows a **lazy class loading** mechanism, meaning that classes are loaded into memory only when they are required at runtime.

## 1. Class Loading Process in Java

The Java ClassLoader is responsible for loading classes into memory when required. The process consists of three main steps:

### 1.1. Loading

- The **ClassLoader** loads the `.class` file (bytecode) into memory when it is first referenced.
- Java has three built-in class loaders:
    - **Bootstrap ClassLoader** → Loads core Java classes (e.g., `java.lang.*`).
    - **Extension ClassLoader** → Loads classes from `lib/ext` directory.
    - **Application ClassLoader** → Loads user-defined classes from the `classpath`.

### 1.2. Linking

After a class is loaded, it undergoes three steps:

- **Verification**: Ensures bytecode follows Java rules and is not corrupted.
- **Preparation**: Allocates memory for static variables and initializes them with default values.
- **Resolution**: Converts symbolic references (e.g., method calls) to actual memory references.

#### 1.3. Initialization

- Static variables are assigned actual values (if provided).
- Static blocks are executed (if present).

3. Can we mark a local variable as static

Ans3. No, **we cannot mark a local variable as `static` in Java**.

**Local variables belong to a method and exist only during method execution.**

**Static variables belong to the class and exist independently of objects.**

**If a local variable were `static`, it would be shared across multiple method calls, which contradicts the nature of local variables.**

4. Why is the static block executed before the main method in java

Ans4. The **static block is executed before the `main()` method** in Java because it is part of the **class initialization process**. When a class is loaded into memory, the JVM executes all **static blocks** in the order they appear **before calling `main()`**.

### How It Works?

1. **Class is loaded** by the ClassLoader.
2. **Static variables are allocated memory** and initialized with default values.
3. **Static blocks are executed** to initialize static variables.
4. **`main()` method is executed**.

### Example:

```
class Demo {
    static {
        System.out.println("Static block executed!");
    }

    public static void main(String[] args) {
        System.out.println("Main method executed!");
    }
}
```

Output:

```
Static block executed!
Main method executed!
```

5. Why is a static method also called a class method
Ans5. A **static method is also called a class method** because it belongs to the **class itself**, not to any specific instance of the class.

### Reasons:

1. **No Instance Required** – Static methods can be called using the class name, without creating an object.
2. **Shared Across All Objects** – They operate at the class level and have no access to instance variables (`non-static` members).
3. **Memory Efficiency** – Since they don't require an instance, they help save memory by avoiding unnecessary object creation.

### Example:

```
class MathUtil {
    static int square(int x) {
        return x * x;
    }
}

public class Main {
    public static void main(String[] args) {
        System.out.println(MathUtil.square(5)); // Calling without an object
```

```
    }
}
```

Output:

25

6. What is the use of static blocks in java

Ans6. **Use of Static Blocks in Java**

A **static block** in Java is used to initialize **static variables** and execute **one-time setup code** before the `main()` method or any object creation.

### Key Uses:

1. **Initializing Static Variables**
   - Static blocks help initialize complex static data before the class is used.
2. **Executing Code Before `main()`**
   - Since static blocks run when the class is loaded, they execute **before `main()`**.
3. **Loading External Resources (Files, Database Connections, Libraries, etc.)**
   - Useful for setting up configurations or loading native libraries.

7. Difference between Static and Instance variable

Ans7. **Difference Between Static and Instance Variables in Java**

| Feature | Static Variable 🔹 | Instance Variable 🔸 |
|---|---|---|
| **Belongs To** | Class (shared by all objects) | Individual object (each object gets its own copy) |
| **Memory Location** | Stored in **Method Area** (shared memory) | Stored in **Heap Memory** (separate for each object) |
| **Access Method** | Can be accessed using **ClassName.variable** or object | Accessed only through an object |
| **Initialization** | Default values are assigned when the class is loaded | Initialized when an object is created |
| **Lifetime** | Exists as long as the class is loaded | Exists as long as the object exists |
| **Example** | `static int count;` | `int age;` |

Example Code

```
class Demo {
    static int count = 0; // Static variable (shared)
    int age; // Instance variable (unique for each object)

    Demo(int age) {
        this.age = age;
        count++; // Increases for every object created
    }

    void display() {
        System.out.println("Age: " + age + ", Count: " + count);
    }
}

public class Main {
    public static void main(String[] args) {
        Demo obj1 = new Demo(25);
        Demo obj2 = new Demo(30);

        obj1.display();
        obj2.display();
    }
}
```

Output:

```
Age: 25, Count: 2
Age: 30, Count: 2
```

8. Difference between static and non static members

Ans8. **Difference Between Static and Non-Static Members in Java**

| Feature | Static Members 🔹 | Non-Static Members 🔸 |
|---|---|---|
| **Belongs To** | Class (shared by all instances) | Individual object (each instance has its own copy) |

| | | |
|---|---|---|
| **Access Method** | Accessed using **ClassName.member** or object | Accessed only through an object |
| **Memory Location** | Stored in **Method Area** (shared memory) | Stored in **Heap Memory** (separate for each object) |
| **When Created?** | When the class is loaded | When an object is created |
| **When Destroyed?** | When the class is unloaded | When the object is garbage collected |
| **Can Access** | **Only static members** (cannot access instance variables/methods) | Both static and non-static members |
| **Usage** | Shared data, utility methods | Object-specific behavior |
| **Example** | `static int count;` | `int age;` |

Example Code

```
class Demo {
    static int count = 0;  // Static member (shared)
    int age;  // Non-static member (unique per object)

    Demo(int age) {
        this.age = age;
        count++;  // Shared among all objects
    }

    static void showCount() {  // Static method
        System.out.println("Count: " + count);
    }

    void display() {  // Non-static method
        System.out.println("Age: " + age + ", Count: " + count);
    }
}
```

```
public class Main {
   public static void main(String[] args) {
      Demo obj1 = new Demo(25);
      Demo obj2 = new Demo(30);

      obj1.display();
      obj2.display();

      Demo.showCount(); // Calling static method
   }
}
```

**Output:**

```
Age: 25, Count: 2
Age: 30, Count: 2
Count: 2
```