1. **Why do we need static keyword in java Explain with an example.**

Ans. In Java, the **static** keyword is used to define class-level variables and methods that can be accessed without creating an instance of the class. When a variable or method is declared as **static**, it belongs to the class itself rather than to any specific instance of the class.

```
public class Example {
   private static int count = 0; // static variable

   public static void incrementCount() { // static method
      count++;
   }

   public static void main(String[] args) {
      Example.incrementCount(); // calling static method without creating an
instance
      System.out.println("Count: " + Example.count); // accessing static variable
directly
   }
}
```

In this example, we have a class called **Example**. It has a static variable **count** and a static method **incrementCount()**.
The **count** variable is declared as **static**, which means it is shared among all instances of the **Example** class. We can access this variable using the class name (**Example.count**) without creating an instance of the class. In the **main()** method, we print the value of **count**.
The **incrementCount()** method is also declared as **static**. This means we can invoke the method using the class name (**Example.incrementCount()**) without creating an instance of the class. It increments the value of the **count** variable.

2. **What is class loading and how does the java program actually executes?**

Ans. Class loading is the process in Java where the Java Virtual Machine (JVM) loads classes into memory so that they can be executed. When a Java program is executed, the JVM follows a series of steps to load and execute the program.
Here's a simplified overview of how a Java program is executed:

1. Compilation: The Java source code is compiled into bytecode by the Java compiler (**javac**). The bytecode is platform-independent and is stored in **.class** files.
2. Class Loading: The JVM's class loader is responsible for loading the bytecode of classes into memory. It searches for the required **.class** files and reads them into

	memory. During the class loading process, the JVM performs several tasks, including verification, preparation, and resolution. • Verification: The JVM verifies the bytecode to ensure it is structurally and semantically correct. It checks for security constraints and potential bytecode violations. • Preparation: The JVM allocates memory for static variables and initializes them with default values. It also prepares data structures to store method and field references. • Resolution: The JVM resolves symbolic references to direct references. It looks up and links the references to the appropriate memory locations. 3. Initialization: After the class loading process, the JVM initializes the static variables and executes the static initializer blocks in the class. The static initializer blocks contain code that runs only once when the class is loaded. 4. Execution: Once the necessary classes are loaded and initialized, the JVM starts executing the program. It locates the **main()** method, which serves as the entry point of the program, and starts executing the statements in the **main()** method sequentially. During the execution phase, the JVM interprets the bytecode or just-in-time (JIT) compiles it into machine code for improved performance. The JVM manages memory allocation, garbage collection, exception handling, and other runtime operations.
3. **Ans.**	**Can we mark a local variable as static.** No, it is not possible to mark a local variable as **static** in Java. The **static** keyword is used to define class-level variables and methods that are shared among all instances of the class. Local variables, on the other hand, are declared within methods, constructors, or blocks and have a limited scope and lifetime. The **static** keyword is not applicable to local variables because it is not meaningful in the context of local variables. Local variables are specific to a particular invocation of a method or block and are created and destroyed each time the method or block is executed.
4. **Ans.**	**Why is the static block executed before the main method in java?** In Java, a **static** block is a special block of code that is executed when the class is loaded into memory by the Java Virtual Machine (JVM). It is executed before the **main()** method, or any other method or constructor, is called. The primary purpose of a **static** block is to perform initialization tasks or set up static resources required by the class. It allows you to execute code that should run once before any instances of the class are created or any other methods are invoked.

	the **static** block is executed first, even before the **main()** method is called. This ensures that any necessary initialization tasks are performed before the execution of the program starts. The order of execution is as follows: 1. The JVM loads the class into memory. 2. The **static** block is executed. 3. The **main()** method is called and executed. It's important to note that the **static** block is executed only once when the class is loaded into memory, regardless of the number of instances created or method invocations.
5. **Ans.**	**Why is a static method also called a class method?** A static method in Java is also commonly referred to as a class method because it is associated with the class itself rather than with any specific instance of the class. Here are a few reasons why a static method is called a class method: 1. Associated with the Class: A static method is defined at the class level rather than at the instance level. It belongs to the class itself, not to any particular object or instance of the class. This means that you can invoke a static method using the class name, without the need to create an instance of the class. 2. Accessible without Object Instantiation: Since a static method is associated with the class and not with any instance, you can call the static method directly using the class name, without creating an object of the class. This makes it easily accessible without the need to instantiate the class. 3. Class-Level Operations: Static methods often perform operations or calculations that are not specific to any particular instance of the class. They may work with static variables or perform common utility tasks that do not require access to instance-specific state or behavior. 4. Shared among Instances: Static methods are shared among all instances of a class. They are not tied to any specific instance's state and can be accessed and invoked by any instance of the class. This shared nature makes them suitable for defining utility methods or common behaviors that apply to all instances of the class. By calling a static method a class method, it emphasizes the fact that the method is associated with the class as a whole and not with individual objects or instances of the class. It helps distinguish static methods from instance methods that are tied to specific instances and require object instantiation to be invoked.
6. **Ans.**	**What is the use of static block in java?**

The static block in Java, also known as a static initializer block, is a special block of code that is executed when the class is loaded into memory by the Java Virtual Machine (JVM). It is used to perform initialization tasks or set up static resources required by the class.

Here are some common uses of the static block:

1. Initialization of Static Variables: The static block is often used to initialize static variables. Since static variables are shared among all instances of a class, they are initialized only once, and the static block provides a convenient place to perform this initialization. The static block runs before the variables are accessed, ensuring that they have the desired initial values.
2. Loading and Initialization of Resources: The static block can be used to load and initialize external resources required by the class, such as configuration files, database connections, or other dependencies. The static block is executed before any other methods are called, making it a suitable place to perform such resource initialization tasks.
3. Exception Handling: The static block can also be used for exception handling. It allows you to catch and handle exceptions that occur during the class loading process. This can be useful for logging or displaying error messages related to the class initialization.
4. Complex Initialization Logic: If the initialization of a class requires more complex logic or computations, the static block provides a convenient way to encapsulate that logic. It allows you to perform computations, conditional checks, or any other necessary setup before the class is ready for use.

7.

Ans.

Difference between Static and Instance variables.

	Static Variables	**Instance Variables**
Declaration	Declared using the `static` keyword within a class, outside of any method or block.	Declared within a class, but outside of any method or block, without the `static` keyword.
Associated with	Associated with the class itself, not with any specific instance.	Associated with individual instances (objects) of a class.
Memory Allocation	Allocated memory once and shared among all instances of the class.	Each instance has its own memory allocation.

Initialization	Initialized with default values (0, false, or null) if not explicitly initialized.	Initialized with default values (0, false, or null) if not explicitly initialized.
Value Modification	Can be modified directly using the class name or through an instance. Changes are visible to all instances.	Each instance has its own copy of the variable and can be modified independently without affecting other instances.
Access	Can be accessed using the class name or through an instance.	Accessed through an instance of the class.
Access from Static Context	Can access other static variables or methods. Cannot directly access instance variables or methods.	Can access both static and instance variables or methods.
Lifetime	Exists as long as the class is loaded in memory.	Exists as long as the instance of the class exists.
Usage	Suitable for defining constants, shared resources, or variables that are common to all instances.	Used for storing object-specific state or data.
Relationship	Not related to any specific instance.	Associated with a specific instance.

8.

Ans.

Difference between Static and non static variables.

	STATIC	**NON-STATIC**
Declaration	Declared with the `static` keyword within a class, outside of any method or block.	Declared within a class without the `static` keyword.

Associated with	Associated with the class itself, not with any specific instance.	Associated with individual instances (objects) of a class.
Memory Allocation	Allocated memory once and shared among all instances of the class.	Each instance has its own memory allocation.
Initialization	Initialized with default values (0, false, or null) if not explicitly initialized.	Initialized with default values (0, false, or null) if not explicitly initialized.
Value Modification	Can be modified directly using the class name or through an instance. Changes are visible to all instances.	Each instance has its own copy of the variable and can be modified independently without affecting other instances.
Access	Can be accessed using the class name or through an instance.	Accessed through an instance of the class.
Access from Static Context	Can access other static variables or methods. Cannot directly access instance variables or methods.	Can access both static and instance variables or methods.
Lifetime	Exists as long as the class is loaded in memory.	Exists as long as the instance of the class exists.
Usage	Suitable for defining constants, shared resources, or variables that are common to all instances.	Used for storing object-specific state or data.

Relationship	Not related to any specific instance.	Associated with a specific instance.