1) what do we need static keyword in Java Explain with an example.

Ans:
In Java, the `static` keyword is used to define class-level members that are shared among all instances of the class. It can be applied to variables, methods, and nested classes. Here's an explanation with examples for each case:

1. Static variables:
   Static variables are shared by all instances of a class. They are associated with the class itself rather than with specific instances of the class. One common use of static variables is to maintain a count of objects created from a class. Here's an example:

```
public class MyClass {
    static int count;  // static variable

    public MyClass() {
        count++;  // increment count each time a new instance is created
    }
}
```

2. Static methods:
   Static methods are associated with the class itself and can be called without creating an instance of the class. They are commonly used for utility methods or operations that don't require access to instance-specific data. Here's an example:

```
public class MathUtils {
    public static int add(int a, int b) {
        return a + b;
    }
}
```

3. Static nested classes:
   Static nested classes are nested classes that are declared as static. They can be accessed without the need to create an instance of the outer class. Here's an example:

```
public class OuterClass {
    static class NestedClass {
        public void display() {
            System.out.println("NestedClass display method");
        }
    }
```

2) What is class loading and how does the Java program actually execute?

Ans:

1. Class loading is the process of loading the bytecode of a class into memory for execution by the JVM.

2. It involves locating the bytecode file (.class), creating a `Class` object, and performing steps like verification, preparation, and resolution.

3. Class loading is dynamic, follows a hierarchical delegation model, and allows for dynamic runtime behavior and flexibility in loading classes from various sources.

The execution of a Java program involves several steps:

1. Compilation: The Java source code is compiled into bytecode using the Java compiler (`javac`). The bytecode is a platform-independent representation of the program.

2. Class Loading: The Java Virtual Machine (JVM) loads the bytecode of the classes referenced by the program into memory. This process involves locating the bytecode files (.class) either from the local file system or from a network location, and creating `java.lang.Class` objects to represent the classes.

3. Linking: The JVM performs three substeps during the linking phase:
   - Verification: The bytecode is verified to ensure it is well-formed and doesn't violate any security constraints. This step checks for structural correctness and prevents potential security vulnerabilities.
   - Preparation: Memory is allocated for static variables and initialized with default values. This step ensures that the memory is set up properly for the class's static variables.
   - Resolution: Symbolic references to other classes or methods are resolved to their actual memory locations. This step resolves names to the corresponding bytecode, linking different classes together.

4. Initialization: The static variables of the classes are initialized with their explicit initial values or by executing static initializer blocks. This step prepares the static state of the classes for execution.

5. Execution: Once the necessary classes have been loaded, linked, and initialized, the JVM starts executing the program. It begins by invoking the `main` method of the class specified as the entry point. The program's execution follows the order of statements and method invocations as defined in the bytecode.

6. Runtime: During execution, the JVM manages memory, performs runtime checks, handles exceptions, and provides a runtime environment for the program. It manages the allocation and deallocation of memory, handles thread management, and performs garbage collection to reclaim memory occupied by objects that are no longer in use.

3) can we mark a local variable as static?

Ans:

No, it is not possible to mark a local variable as `static` in Java. The `static` keyword is used to define class-level members (variables, methods, nested classes) that are associated with the class itself rather than with specific instances of the class.

Local variables, on the other hand, are defined within a method, constructor, or block, and they are specific to the execution of that method, constructor, or block. They are created when the method/block is entered and destroyed when the method/block is exited. Local variables are not shared among different executions of the method or block.

Therefore, the `static` keyword is not applicable to local variables in Java. It can only be used with variables at the class level to create static variables, which are shared by all instances of the class.

4) Why is the static block executed before the main method in Java?

Ans:
The `static` block is executed before the `main` method in Java because it is part of the class initialization process, and it is designed to run before any other code within the class.

The reason behind this order of execution is to ensure that any necessary initialization tasks are performed before the program starts its actual execution in the `main` method. By executing the `static` block first, the class can set up its static variables, perform any required initialization logic, or load resources that are necessary for the subsequent execution of the program.

The `static` block provides a way to execute code that should be run once before the class is used or any objects are created. It allows you to perform static initialization tasks that need to be completed before the program starts running.

5) why is a static method also called a class method?

Ans:
A static method in Java is also referred to as a "class method" because it is associated with the

class itself rather than with individual instances or objects of the class. There are two main 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 operates on the class's static state and does not have access to instance-specific data. It can be invoked using the class name itself, without the need to create an instance of the class. Since it is associated with the class itself, it is commonly referred to as a "class method."

2. Shared among instances: A static method is shared among all instances of the class. It behaves the same way regardless of which instance invokes it. Static methods cannot access non-static (instance) variables or methods directly. They can only access other static members of the class or local variables within the static method itself. This shared behaviour among instances further emphasizes that it is a method related to the class as a whole.

6) What is the use of static block in Java?

Ans:

1. Initialization: The `static` block allows you to initialize static variables or perform initialization tasks that should be executed before the class is used or any objects are created.

2. Complex Initialization: It provides a way to perform complex initialization logic that cannot be achieved with a simple variable assignment. You can use the `static` block to perform calculations, load resources, establish connections, or set up the initial state of static variables.

3. One-time Operations: The `static` block is executed only once when the class is loaded into memory, ensuring that the initialization tasks are performed once and shared among all instances of the class. This is useful for tasks that should be done only once, regardless of the number of objects created.

4. Static Resources: If your class requires the loading of external resources, such as configuration files or database drivers, the `static` block provides a convenient place to load and set up these resources. It ensures that the resources are available before any methods or objects in the class utilize them.

5. Class-level Operations: The `static` block can contain any code that needs to be executed at the class level, independent of any specific instance. This includes running static methods, invoking static utility classes, or performing any other class-level operations that are not tied to instance-specific data.

7) Difference between static and instance variables.

Ans:
The key differences between Static variables and instance variables in Java:

1. Scope and Lifetime:

- Static Variables: Static variables, also known as class variables, have a scope that is tied to the class itself. They are shared among all instances of the class and can be accessed using the class name. Static variables exist for the entire duration of the program's execution.

- Instance Variables: Instance variables, also known as non-static variables, are associated with specific instances of a class. Each instance of the class has its own copy of instance variables. They are accessible within the instance methods of the class and exist as long as the instance exists.

2. Association:

- Static Variables: Static variables are associated with the class itself rather than with any specific instance. They are shared among all instances of the class, meaning that modifications to a static variable by one instance will be visible to all other instances of the class.

- Instance Variables: Instance variables are associated with individual instances of the class. Each instance has its own copy of the instance variables, allowing for separate values for each instance.

3. Initialization:

- Static Variables: Static variables are initialized once, during the class loading process, before any instances of the class are created. They can be explicitly initialized at the point of declaration or in a static initializer block.

- Instance Variables: Instance variables are initialized when a new instance of the class is created. They can be explicitly initialized at the point of declaration or within the class's constructors.

4. Access and Modifiability:

- Static Variables: Static variables can be accessed directly using the class name and the dot operator. They can also be modified and accessed by any instance of the class or even outside the class if the access modifier allows it.

- Instance Variables: Instance variables are accessed and modified through the instance of the class using the dot operator. Each instance has its own set of instance variables and changes to one instance's variables do not affect other instances.

5. Memory Allocation:

- Static Variables: Static variables are stored in a separate area of memory called the "Method Area" or "Static Memory." There is only one copy of each static variable, regardless of the number of instances of the class.

- Instance Variables: Instance variables are stored in each instance's memory allocation. Each instance of the class has its own memory space for instance variables.

8) Difference between static and non-static members.

Ans:
The key differences between static and non-static members:

1. Association:

- Static Members: Static members (variables and methods) are associated with the class itself rather than with any specific instance of the class. They are shared among all instances of the class and can be accessed using the class name.

- Non-Static Members: Non-static members (variables and methods) are associated with individual instances of the class. Each instance of the class has its own set of non-static members.

2. Access:

- Static Members: Static members can be accessed directly using the class name followed by the member name. They are accessible from both static and non-static contexts within the class.

- Non-Static Members: Non-static members are accessed through an instance of the class using the dot operator. They can be accessed from non-static contexts within the class, as well as from instance methods or constructors.

3. Memory Allocation:

- Static Members: Static members are stored in a separate area of memory called the "Method Area" or "Static Memory." There is only one copy of each static member, regardless of the number of instances of the class.

- Non-Static Members: Non-static members are stored in each instance's memory allocation. Each instance of the class has its own memory space for non-static members.

4. Initialization:

- Static Members: Static variables can be explicitly initialized at the point of declaration or in a static initializer block. Static methods can be used without creating an instance of the class.

- Non-Static Members: Non-static variables are typically initialized in the class's constructors or through assignment statements within the class. Non-static methods can access non-static variables and can only be invoked on instances of the class.

5. Modifiability:

- Static Members: Static variables can be modified and accessed by any instance of the class or even outside the class if the access modifier allows it. Static methods can modify static variables or call other static methods.

- Non-Static Members: Non-static variables are specific to individual instances and can be modified and accessed within the instance where they are defined. Non-static methods can access and modify both non-static variables and static variables.