Assignment-15

Static Keyword

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

Ans:

In Java, the **static** keyword is used to define a class-level variable, method or nested class that belongs to the class rather than an instance of the class.

Here are some reasons why we need the **static** keyword in Java:

1. To define class-level variables: When a variable is declared as static, only one copy of the variable is created for the entire class, rather than a separate copy for each object of the class. This can save memory and can also help to ensure that all instances of the class share the same value for the variable.

Example:

```
public class Circle {
    public static final double PI = 3.14;
    private double radius;

    public Circle(double radius) {
        this.radius = radius;
    }
    public double getArea() {
        return PI * radius * radius;
    }
}
```

In the above example, the **PI** variable is declared as static and final, which means that it belongs to the class rather

than an instance of the class. This ensures that all instances of the **Circle** class use the same value of **PI**.

2. To define class-level methods: When a method is declared as static, it can be called without creating an instance of the class. This can be useful for utility methods that don't require any state information from the class.

Example

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

In the above example, the **add** method is declared as static, which means it can be called without creating an instance of the **MathUtils** class.

3. To define a nested class: When a nested class is declared as static, it can be accessed without creating an instance of the outer class. This can be useful for grouping related functionality together in a single class.

Example:

```
public class OuterClass {
    public static class InnerClass {
        public void doSomething() {
            System.out.println("Doing something...");
        }
    }
}
```

In the above example, the **InnerClass** is declared as static, which means it can be accessed without creating an instance of the **OuterClass**.

In summary, the **static** keyword in Java is used to define class-level variables, methods, and nested classes that belong to the class rather than an instance of the class. This can help to save memory, improve performance, and provide a way to group related functionality together in a single class

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

Ans:

In Java, class loading is the process of loading Java classes and interfaces into the Java Virtual Machine (JVM) from compiled Java bytecode files. Class loading is a key part of the Java runtime environment, as it is responsible for finding and loading the required class files when a Java program is executed.

When a Java program is executed, the JVM first loads the classes and interfaces that are required for the program to run. The JVM uses a class loader system to load classes and interfaces on demand. The class loader system is responsible for locating the compiled class files that correspond to the Java classes and interfaces that are needed by the program.

The class loader system uses a hierarchical structure, where each class loader is responsible for loading classes and interfaces from a specific location in the classpath. When a class or interface is needed by the program, the class loader system checks each class loader in the hierarchy, starting with the parent class loader and moving down to the child class loader, until the required class or interface is found.

Once the required classes and interfaces have been loaded, the JVM creates an instance of the main class of the program and starts executing it. The main class of the program typically contains the **main()** method, which is the entry point for the program. The JVM executes the **main()** method, which in turn calls other methods and uses other classes and interfaces as needed.

During program execution, the JVM dynamically loads additional classes and interfaces as they are needed by the program. This allows Java programs to be very flexible and adaptable, as they can dynamically load new classes and interfaces at runtime.

In summary, class loading is the process of loading Java classes and interfaces into the JVM from compiled bytecode files, and is a key part of the Java runtime environment. The JVM uses a hierarchical class loader system to load classes and interfaces on demand, and dynamically loads additional classes and interfaces as needed during program execution. The main class of the program contains the **main()** method, which is the entry point for the program and triggers the start of program execution.

3.Can we mark a local local variable as static variable ?

Ans:

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

The **static** keyword is used to declare class-level variables, methods, and nested classes, and is not valid for local variables. Local variables are created and destroyed every time their enclosing method or block is executed, and are not shared among instances of the class or accessible outside the scope of their method or block.

In Java, the **static** keyword can only be used with class-level variables, methods, and nested classes, as they belong to the class rather than an instance of the class.

Here's an example to demonstrate that attempting to mark a local variable as **static** will result in a compilation error:

```
public class ExampleClass {

    public void exampleMethod() {

        static int exampleVar = 0; // This will result in a compilation error

        int localVar = 0;

        // Method code goes here
```

```
    }
}
```

In the above example, attempting to mark the **exampleVar** variable as **static** will result in a compilation error, as local variables cannot be marked as **static**.

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

Ans:

In Java, the **static** block is executed before the **main** method because it is a part of the class loading process.

When a Java program is executed, the JVM loads the required classes and interfaces into memory. During the class loading process, the JVM looks for any **static** blocks in the class and executes them in the order they appear in the code.

Once all **static** blocks have been executed, the **main** method is executed as the entry point of the program. This is because the **main** method is a special method that is required in order to run a Java program. The **main** method is defined with the **static** keyword, which means it belongs to the class rather than an instance of the class.

The order of execution for a Java program is as follows:

1. The JVM loads the required classes and interfaces into memory.
2. The JVM executes any **static** blocks in the class in the order they appear in the code.
3. The **main** method is executed as the entry point of the program.

Here's an example to demonstrate the order of execution:

csharp

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

In the above example, the **static** block is executed before the **main** method, and the output will be:

Output:

```
Static block executed
```

Main method executed

In summary, the **static** block is executed before the **main** method in Java because it is a part of the class loading process. The **main** method is executed after all **static** blocks have been executed, and serves as the entry point of the program.

5.Whyis a Static method also called a class method ?

Ans:

A static method is also called a class method because it belongs to the class rather than an instance of the class.

In Java, a static method is declared with the **static** keyword, which means that it is associated with the class rather than an instance of the class. This means that the static method can be called on the class itself, rather than on an object or instance of the class.

Since a static method is associated with the class rather than an instance of the class, it is also sometimes referred to as a class method. This is because the static method can be accessed and called on the class itself, without the need for an instance of the class.

Here's an example to demonstrate the use of a static method:

```
public class ExampleClass {
    public static void staticMethod() {
        System.out.println("This is a static method");
    }
    public void instanceMethod() {
        System.out.println("This is an instance method");
    }
}
public class Main {
    public static void main(String[] args) {
        ExampleClass.staticMethod(); // This calls the static method on the class
        ExampleClass instance = new ExampleClass();
        instance.instanceMethod(); // This calls the instance method on an instance of the class
    }
}
```

In the above example, the **staticMethod()** is declared as a static method and can be called on the class itself, while the **instanceMethod()** is an instance method that can only be called on an instance of the class.

In summary, a static method is called a class method because it is associated with the class rather than an instance of the class, and can be called on the class itself.

6. What is the use of Static and Instance variables ?

Ans:

In Java, both static and instance variables serve different purposes and are used in different contexts.

A static variable is a variable that belongs to the class rather than an instance of the class. It is declared with the **static** keyword and is shared among all instances of the class. Static variables are used to store data that is common to all instances of the class. They can be accessed and modified by both static and instance methods.

An instance variable, on the other hand, is a variable that belongs to an instance of the class. It is not declared with the **static** keyword and is unique to each instance of the class. Instance variables are used to store data that is specific to each instance of the class. They can be accessed and modified only by instance methods.

Here's an example to demonstrate the use of static and instance variables:

```
public class ExampleClass {
    private static int staticVar;
    private int instanceVar;

    public static void setStaticVar(int value) {
        staticVar = value;
    }

    public void setInstanceVar(int value) {
        instanceVar = value;
    }

    public void printVariables() {
        System.out.println("Static variable: " + staticVar);
```

```
        System.out.println("Instance variable: " + instanceVar);
    }
}

public class Main {
    public static void main(String[] args) {
        ExampleClass.setStaticVar(10);       // This sets the static variable on the class
        ExampleClass instance1 = new ExampleClass();
        instance1.setInstanceVar(20);          // This sets the instance variable on the first instance of the class
        ExampleClass instance2 = new ExampleClass();
        instance2.setInstanceVar(30);          // This sets the instance variable on the second instance of the class

        instance1.printVariables();            // This prints the variables for the first instance of the class
        instance2.printVariables();            // This prints the variables for the second instance of the class
    }
}
```

In the above example, the **staticVar** is a static variable that is shared among all instances of the class, while the **instanceVar** is an instance variable that is unique to each instance of the class. The **setStaticVar()** method is a static method that sets the value of the static variable, while the **setInstanceVar()** method is an instance method that sets the value of the instance variable. The **printVariables()** method is an instance method that prints the values of both variables.

In summary, static variables are used to store data that is common to all instances of the class, while instance variables are used to store data that is specific to each instance of the class. Both static and instance variables are useful in different contexts and serve different purposes.

7.Difference B/w(between) Static and Instance variable ?

Ans:

The main difference between static and instance variables in Java is that a static variable is associated with the class itself, while an instance variable is associated with an instance of the class.

Here are some key differences between static and instance variables:

1. Declaration: A static variable is declared with the **static** keyword, while an instance variable is not.
2. Memory Allocation: A static variable is allocated memory in the static memory area, while an instance variable is allocated memory in the heap memory area.
3. Scope: A static variable is accessible to all instances of the class, while an instance variable is accessible only to the instance of the class on which it is declared.
4. Lifetime: A static variable exists for the entire lifetime of the program, while an instance variable exists for the lifetime of the instance of the class on which it is declared.
5. Initialization: A static variable is initialized only once, when the class is loaded into memory, while an instance variable is initialized each time a new instance of the class is created.
6. Access: A static variable can be accessed through the class name, while an instance variable can be accessed through the instance of the class.

Here's an example to illustrate the differences between static and instance variables:

```
public class ExampleClass {
    private static int staticVar;
    private int instanceVar;

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

        obj1.staticVar = 10;
        obj1.instanceVar = 20;

        obj2.staticVar = 30;
        obj2.instanceVar = 40;

        System.out.println("obj1.staticVar: " + obj1.staticVar);
        System.out.println("obj1.instanceVar: " + obj1.instanceVar);

        System.out.println("obj2.staticVar: " + obj2.staticVar);
        System.out.println("obj2.instanceVar: " + obj2.instanceVar);
    }
```

}

In the above example, **staticVar** is a static variable and **instanceVar** is an instance variable. Two instances of the **ExampleClass** class are created, and their static and instance variables are assigned different values. The **staticVar** variable is accessed through the class name (**ExampleClass.staticVar**) and the **instanceVar** variable is accessed through the instance of the class (**obj1.instanceVar**).

In summary, static variables are associated with the class itself and are shared among all instances of the class, while instance variables are associated with an instance of the class and are unique to each instance. Static variables are declared with the `static` keyword, while instance variables are not.

8.Difference Between(B/w) static and Non-Static Members ?

Ans:

In Java, there are two types of class members: static and non-static (also called instance members). Here are the key differences between static and non-static members:

1. Declaration: A static member is declared with the **static** keyword, while a non-static member is not.
2. Memory Allocation: A static member is allocated memory in the static memory area, while a non-static member is allocated memory in the heap memory area.
3. Association: A static member is associated with the class itself, while a non-static member is associated with an instance of the class.
4. Scope: A static member is accessible to all instances of the class, while a non-static member is accessible only to the instance of the class on which it is called.
5. Lifetime: A static member exists for the entire lifetime of the program, while a non-static member exists only for the lifetime of the instance of the class on which it is called.
6. Initialization: A static member is initialized only once, when the class is loaded into memory, while a non-static member is initialized each time a new instance of the class is created.
7. Access: A static member can be accessed through the class name, while a non-static member can be accessed only through an instance of the class.

Here's an example to illustrate the differences between static and non-static members:

```
public class ExampleClass {
    private static int staticVar;
    private int instanceVar;

    public static void staticMethod() {
        staticVar = 10;
```

```
        // instanceVar = 20; // This would cause a compile-time error
    }

    public void instanceMethod() {
        staticVar = 30;
        instanceVar = 40;
    }
}
```

In the above example, **staticVar** is a static member and **instanceVar** is a non-static member. **staticMethod()** is a static method and **instanceMethod()** is a non-static method. Note that **staticMethod()** cannot access **instanceVar**, because it is associated with an instance of the class, while **staticVar** is associated with the class itself. Similarly, **instanceMethod()** can access both **staticVar** and **instanceVar**, because it is called on an instance of the class.

In summary, static members are associated with the class itself and are shared among all instances of the class, while non-static members are associated with an instance of the class and are unique to each instance. Static members are declared with the **static** keyword, while non-static members are not. Static members can be accessed through the class name, while non-static members can be accessed only through an instance of the class.