ASSIGNMENT

Q1 ans-

In Java, to create an object, you need to follow these steps:

1. **Define a Class**: First, you need to define a class that serves as a blueprint for the type of object you want to create. A class defines the data (attributes) and behavior (methods) of the objects of that type.

2. **Instantiate the Object**: Once you have defined the class, you can create an instance (object) of that class using the `new` keyword followed by the class constructor.

Here's a step-by-step guide to create an object in Java:

Step 1: Define a Class

```java
public class MyClass {
    // Attributes (variables)
    int myVariable;
    String myString;

    // Methods (behaviors)
    void displayInfo() {
        System.out.println("myVariable: " + myVariable);
        System.out.println("myString: " + myString);
    }
}
```

```

Step 2: Instantiate the Object

```java
public class Main {
    public static void main(String[] args) {
        // Creating an object of MyClass
        MyClass myObject = new MyClass();

        // Accessing attributes and methods of the object
        myObject.myVariable = 42;
        myObject.myString = "Hello, World!";
        myObject.displayInfo();
    }
}
```

In this example, we have created a simple class `MyClass` with two attributes (`myVariable` and `myString`) and a method (`displayInfo`). In the `Main` class, we created an object `myObject` of type `MyClass` using the `new` keyword, and then we accessed and assigned values to the attributes of the object and called its method `displayInfo()`.

When you run the `Main` class, it will output:

```
myVariable: 42
myString: Hello, World!
```
```

This demonstrates the process of creating and using an object in Java. Objects allow you to encapsulate data and behavior, making code more organized and easier to maintain. You can create multiple objects from the same class, each with its own unique data and state.

Q2 ans-

In Java, the `new` keyword is used to dynamically allocate memory and create objects of classes at runtime. It is an essential part of the process of object instantiation and memory allocation. When you use the `new` keyword, it allocates memory for the object and returns a reference (memory address) to that object.

The main uses of the `new` keyword are as follows:

1. **Object Instantiation**: The primary purpose of the `new` keyword is to create instances (objects) of classes. When you create an object using `new`, Java allocates memory for the object and initializes its attributes with their default values.

Example:

```java
MyClass myObject = new MyClass();
```

2. **Dynamic Memory Allocation**: Unlike primitive data types, which are stored in stack memory, objects are stored in the heap memory. The `new` keyword allocates memory in the heap for the object, making it available for use even after the method that created it is completed.

Example:

```java
int[] numbers = new int[5]; // Creates an array of integers with 5 elements in the heap memory.
```

3. **Creating Arrays**: The `new` keyword is used to create arrays of any data type, including primitive types and objects. It dynamically allocates memory for the array and initializes its elements with default values (0 for numeric types, `false` for booleans, and `null` for objects).

Example:

```java
int[] numbers = new int[5]; // Creates an array of integers with 5 elements.
String[] names = new String[3]; // Creates an array of strings with 3 elements.
```

4. **Creating Objects of Custom Classes**: You can use the `new` keyword to create objects of your custom-defined classes. It is a fundamental step in object-oriented programming, where classes serve as blueprints, and objects represent instances of those classes.

Example:

```java
MyClass myObject = new MyClass(); // Creates an object of the custom class "MyClass."
```

It's important to note that objects created with `new` must be explicitly deallocated (garbage collected) by Java's memory management system when they are no longer referenced by any part of the program. Java's automatic garbage collector frees memory used by objects that are no longer needed, helping to manage memory efficiently and prevent memory leaks.

Q3 ans-

In Java, variables can be classified into different types based on their scope and where they are defined. The main types of variables in Java are:

1. **Local Variables**:

- Local variables are declared within a method, constructor, or block of code.

- They have limited scope and are only accessible within the method, constructor, or block in which they are declared.

- Local variables must be initialized before they are used.

Example:

```java
public void exampleMethod() {
    int x = 10; // Local variable
    System.out.println(x);
}
```

2. **Instance Variables (Non-Static Variables)**:

- Instance variables are declared within a class, but outside any method or block.

- They belong to the instance (object) of the class and have separate copies for each instance of the class.

- Instance variables are initialized with default values (e.g., 0 for numeric types, `false` for booleans, and `null` for object references) if not explicitly initialized.

Example:

```java
public class MyClass {
    int instanceVariable; // Instance variable
}
```

3. **Class Variables (Static Variables)**:

- Class variables are declared with the `static` keyword within a class, but outside any method or block.

- They are shared among all instances (objects) of the class and have only one copy in memory.

- Class variables are initialized with default values if not explicitly initialized.

Example:

```java
public class MyClass {
    static int classVariable; // Class variable
}
```

4. **Parameter Variables**:

- Parameter variables are used in method or constructor signatures to receive values from the caller.

- They act as local variables within the method or constructor and have scope limited to that method or constructor.

- Parameter variables are initialized with the values passed as arguments when the method or constructor is called.

Example:

```java
public void exampleMethod(int parameter1, String parameter2) {
    // parameter1 and parameter2 are parameter variables
    System.out.println(parameter1 + " " + parameter2);
}
```

The choice of variable type depends on the intended usage and scope requirements. Local variables are used for temporary storage within a method, instance variables for object-specific data, and class variables for data shared among all instances of a class. Parameter variables enable methods to accept external inputs and operate on them.

Q4 ans-

The main differences between instance variables and local variables in Java are related to their scope, lifetime, and where they are declared within a class. Here's a breakdown of the differences:

Instance Variables:

1. **Scope**:
   - Instance variables are declared within a class but outside any method, constructor, or block.
   - They have class-level scope and are accessible to all methods, constructors, and blocks within the class.

2. **Lifetime**:
   - Instance variables exist as long as the instance (object) of the class to which they belong exists.
   - They are created when an object is created (memory is allocated for the object) and are destroyed when the object is garbage-collected.

3. **Initialization**:
   - Instance variables are initialized with default values if not explicitly initialized. Numeric types are initialized to 0, booleans to `false`, and object references to `null`.

4. **Usage**:
   - Instance variables are used to represent attributes or properties of an object.
   - Each instance (object) of the class has its own separate copy of instance variables.

Local Variables:

1. **Scope**:
   - Local variables are declared within a method, constructor, or block of code (such as an if-statement or loop).
   - They have a limited scope and are accessible only within the method, constructor, or block in which they are declared.

2. **Lifetime**:
   - Local variables exist only as long as the method, constructor, or block in which they are declared is being executed.

- They are created when the method, constructor, or block is entered and are destroyed when it is exited.

3. **Initialization**:
   - Local variables must be explicitly initialized before they are used. Unlike instance variables, they do not have default values.

4. **Usage**:
   - Local variables are used for temporary storage and calculations within a method, constructor, or block.
   - They are typically used to store intermediate values or perform specific operations within a limited context.

Example demonstrating instance and local variables:

```java
public class MyClass {
    int instanceVariable; // Instance variable

    public void exampleMethod() {
        int localVar = 10; // Local variable

        System.out.println(instanceVariable); // Accessing instance variable
        System.out.println(localVar); // Accessing local variable
    }
}
```

In this example, `instanceVariable` is an instance variable, which belongs to the instance (object) of the class. `localVar` is a local variable, which is declared within the `exampleMethod()` and has limited scope within that method.

Q5 ans-

The main differences between instance variables and local variables in Java are related to their scope, lifetime, and where they are declared within a class. Here's a breakdown of the differences:

Instance Variables:

1. **Scope**:

   - Instance variables are declared within a class but outside any method, constructor, or block.

   - They have class-level scope and are accessible to all methods, constructors, and blocks within the class.

2. **Lifetime**:

   - Instance variables exist as long as the instance (object) of the class to which they belong exists.

   - They are created when an object is created (memory is allocated for the object) and are destroyed when the object is garbage-collected.

3. **Initialization**:

   - Instance variables are initialized with default values if not explicitly initialized. Numeric types are initialized to 0, booleans to `false`, and object references to `null`.

4. **Usage**:

   - Instance variables are used to represent attributes or properties of an object.

   - Each instance (object) of the class has its own separate copy of instance variables.

Local Variables:

1. **Scope**:

   - Local variables are declared within a method, constructor, or block of code (such as an if-statement or loop).

   - They have a limited scope and are accessible only within the method, constructor, or block in which they are declared.

2. **Lifetime**:

   - Local variables exist only as long as the method, constructor, or block in which they are declared is being executed.

- They are created when the method, constructor, or block is entered and are destroyed when it is exited.

3. **Initialization**:

- Local variables must be explicitly initialized before they are used. Unlike instance variables, they do not have default values.

4. **Usage**:

- Local variables are used for temporary storage and calculations within a method, constructor, or block.

- They are typically used to store intermediate values or perform specific operations within a limited context.

Example demonstrating instance and local variables:

```java
public class MyClass {
    int instanceVariable; // Instance variable

    public void exampleMethod() {
        int localVar = 10; // Local variable

        System.out.println(instanceVariable); // Accessing instance variable
        System.out.println(localVar); // Accessing local variable
    }
}
```

In this example, `instanceVariable` is an instance variable, which belongs to the instance (object) of the class. `localVar` is a local variable, which is declared within the `exampleMethod()` and has limited scope within that method.

Q6 ans-

Method overloading is a feature in Java that allows a class to have multiple methods with the same name but different parameters. When you define two or more methods with the same name but different parameter lists, you are overloading the method.

In Java, method overloading is achieved by changing the number of parameters, the data types of parameters, or the order of parameters in the method's signature. The return type of the method is not considered during method overloading; the parameter list is the key factor.

Advantages of Method Overloading:

1. **Code Reusability**: Method overloading allows you to reuse method names and make the code more concise and readable.

2. **Flexibility**: It provides flexibility in calling methods with different argument types without the need for different method names.

3. **Easier to Remember**: Method overloading enables you to use the same method name for logically related operations, making it easier to remember and understand.

Example of Method Overloading:

```java
public class MathOperations {
    // Method to add two integers
    public int add(int a, int b) {
        return a + b;
    }

    // Method to add three integers (overloaded version)
    public int add(int a, int b, int c) {
        return a + b + c;
```

```
    }

    // Method to add two double values (overloaded version)
    public double add(double a, double b) {
        return a + b;
    }
}
```

In this example, the `MathOperations` class has three overloaded `add` methods. The first method takes two integers and returns their sum. The second method takes three integers and returns their sum. The third method takes two double values and returns their sum. The method names are the same (`add`), but the number and types of parameters are different, which allows us to use the same method name for different scenarios.

When you call the `add` method with different argument types or numbers of arguments, Java determines the appropriate version of the method to be executed based on the provided arguments.