Assignment

Q.1 In Java, creating an object involves these key steps:

1. Declaration: You declare a variable of the class type. This variable will hold a *reference* to the object.

2. Instantiation: You use the `new` keyword to create an instance of the class (the actual object in memory). This allocates memory for the object.

Example:
```
// Define a class (blueprint)
class Dog {
   String breed;
   String name;
   int age;

   //Method
   public void bark(){
      System.out.println("Woof!");
   }
}

public class Main {
   public static void main(String[] args) {
      // 1. Declaration: Declare a variable of type Dog
      Dog myDog;

      // 2. Instantiation: Create a Dog object using 'new'
      myDog = new Dog("Golden Retriever", "Buddy", 3);



      // 3. Accessing members (instance variables and methods)
      System.out.println("My dog's breed: " + myDog.breed); // Output: Golden Retriever
      System.out.println("My dog's name: " + myDog.name);   // Output: Buddy
      System.out.println("My dog's age: " + myDog.age);     // Output: 3
      myDog.bark();//Output: Woof!

      System.out.println("Your dog's breed: " + yourDog.breed); // Output: Labrador
      System.out.println("Your dog's name: " + yourDog.name);   // Output: Max
      System.out.println("Your dog's age: " + yourDog.age);     // Output: 5
      yourDog.bark();//Output: Woof!
   } }
```

Q.2 The `new` keyword in Java is fundamental to object-oriented programming. Its primary purpose is to **create objects (instances) of classes**. Here's a breakdown of what `new` does:

1. **Memory Allocation:** The most crucial function of `new` is to allocate memory on the heap (the dynamic memory area) to store the new object. The amount of memory allocated depends on the size of the class (the number and types of instance variables it has).

2. **Object Initialization:** After allocating memory, `new` initializes the object. This involves:

    - Setting instance variables to their default values (e.g., 0 for numeric types, `false` for booleans, `null` for object references).
    - **Calling the constructor:** The constructor is a special method of the class that is used to perform further initialization. It sets the initial state of the object, often based on values passed as arguments.

3. **Returning a Reference:** The `new` operator returns a *reference* (memory address) to the newly created object. This reference is then assigned to a variable of the class type

Q.3 1. Primitive Data Types:

- **byte:** Stores a single byte of data (8 bits). Range: -128 to 127.
-
- **short:** Stores a two-byte integer. Range: -32,768 to 32,767.
-
- **int:** Stores a four-byte integer. Range: -2,147,483,648 to 2,147,483,647.
- **long:** Stores an eight-byte integer. Range: -9,223,372,036,854,775,808 to 9,223,372,036,854,775,807.
- **float:** Stores a single-precision floating-point number.
-
- **double:** Stores a double-precision floating-point number.

- **char:** Stores a single character (Unicode character).
-
- **boolean:** Stores a boolean value, either `true` or `false`.

2. Reference Data Types:

- **Class Types:** These refer to objects created from classes. Examples include:

    - `String`
    - `Array`
    - `Date`
    - Custom-defined classes (e.g., `Employee`, `Car`)
- **Interface Types:** These refer to objects that implement a specific interface.

Q.4 Instance Variables

- Declaration: Declared inside a class but outside any method, constructor, or block.
-
- Scope: Accessible to all methods, constructors, and blocks within the same class.
- Lifetime: Their lifetime is tied to the object's lifetime. They are created when an object is created using the `new` keyword and destroyed when the object is garbage collected.
-
- Default Values: Have default values assigned to them (e.g., 0 for `int`, `null` for objects).
- Access Modifiers: Can have access modifiers like `public`, `private`, `protected`, or default (no modifier).

Local Variables

- Declaration: Declared inside methods, constructors, or blocks of code.
-
- Scope: Limited to the block of code where they are declared. They are not accessible outside that block.
-
- Lifetime: Their lifetime is limited to the execution of the block of code where they are declared. They are created when the block is entered and destroyed when the block is exited.
-
- No Default Values: Must be explicitly initialized before use. The compiler will generate an error if you try to use a local variable without initializing it.
-
- No Access Modifiers: Cannot have access modifiers.

Example

```
public class Car { // Class

   String model; // Instance variable

   String color; // Instance variable
```

```
    public Car(String model, String color) { // Constructor
        this.model = model;
        this.color = color;
    }

    public void startEngine() { // Method
        int engineTemperature = 20; // Local variable
        System.out.println("Engine started. Temperature: " + engineTemperature);
        System.out.println("Car model is: " + model); // Accessing instance variable
    }

    public void drive() {
        // engineTemperature = 50; // Error: Cannot access engineTemperature here
        System.out.println("Driving the " + color + " " + model); // Accessing instance variables
    }

    public static void main(String[] args) {
        Car myCar = new Car("Sedan", "Red");
        myCar.startEngine();
        myCar.drive();
    }
}
```

Q.5 n Java, memory is allocated differently for instance variables and local variables:

Instance Variables:

- Memory Area: Instance variables are stored in the heap memory.
-
- Heap Memory: The heap is a region of memory where objects are stored. When you create an object using the `new` keyword, the memory for that object, including its instance variables, is allocated on the heap.
-
- Lifetime: The memory occupied by instance variables exists as long as the object exists. When the object is no longer referenced (i.e., no variables are pointing to it), the garbage collector reclaims the memory.

Local Variables:

- Memory Area: Local variables are stored in the stack memory.
-
- Stack Memory: The stack is a region of memory that operates in a LIFO (Last-In, First-Out) manner. Each thread in a Java program has its own stack. When a method is called, a new frame is pushed onto the stack. This frame contains memory for the method's local variables, parameters, and return address. When the method completes, its frame is popped off the stack, and the memory for its local variables is automatically deallocated.
-
- Lifetime: The memory occupied by local variables exists only during the execution of the method or block in which they are declared.

Q.6 Method overloading in Java (and many other object-oriented languages) is a powerful feature that allows you to define multiple methods within the same class that have the *same name* but *different parameter lists*.

Key Characteristics of Method Overloading:

- Same Name: The methods must have the same name.
- Different Parameter Lists: The methods must have different:
    - Number of parameters: A method with two parameters is different from a method with three parameters.
    - Types of parameters: A method with an `int` parameter is different from a method with a `String` parameter.
    - Order of parameters: A method with parameters `(int x, String y)` is different from a method with parameters `(String y, int x)`.
-
-

- Return Type is Not a Factor: The return type of the method alone is *not* sufficient to distinguish overloaded methods. If two methods have the same name and parameter list but different return types, it will result in a compile-time error.

Example:

```
public class Calculator {


    // Method to add two integers

    public int add(int a, int b) {

        return a + b;

    }


    // Method to add three integers

    public int add(int a, int b, int c) {

        return a + b + c;

    }


    // Method to add two doubles

    public double add(double a, double b) {

        return a + b;

    }


    // Method to add a String and an int

    public String add(String str, int num) {

        return str + num;

    }
    public static void main(String[] args) {
        Calculator calc = new Calculator();


        System.out.println(calc.add(2, 3));        // Calls add(int, int)   Output: 5
        System.out.println(calc.add(2, 3, 4));     // Calls add(int, int, int) Output: 9
        System.out.println(calc.add(2.5, 3.7));   // Calls add(double, double) Output: 6.2
        System.out.println(calc.add("Result: ", 10)); // Calls add(String, int) Output: Result: 10
    }
}
```