1. General Form of a Class

A class in Java is a blueprint for objects that defines a set of attributes (fields) and methods (behaviors). The general structure of a class is as follows:

Syntax:

```
class ClassName {
    // Fields (variables)
    dataType variableName;

    // Constructor
    ClassName() {
        // Constructor body
    }

    // Methods
    returnType methodName() {
        // Method body
    }
}
```

Example code:

```
class Car {
    String model;
    int year;

    Car(String model, int year) {
        this.model = model;
        this.year = year;
    }

    void displayInfo() {
        System.out.println("Model: " + model + ", Year: " + year);
    }
}

public class Main {
    public static void main(String[] args) {
        Car car = new Car("Honda Civic", 2020);
        car.displayInfo();
    }
}
```

Output:

Model: Honda Civic, Year: 2020

Q 2. Methods in Java

i) Method with Parameters

A method that accepts values as inputs. Parameters allow data to be passed to the method.

- Helps in reusing the same method with different inputs.
- Improves modularity.

Example code:

```
class Calculator {
    void sum(int a, int b) {
        System.out.println("Sum: " + (a + b));
    }
}
public class Main {
    public static void main(String[] args) {
        Calculator calc = new Calculator();
        calc.sum(5, 10);
    }
}
```

Output:
Sum: 15

ii) Method without Parameters

A method that doesn't accept any input values.

- Useful when the task is fixed and doesn't need any inputs.

Example:
```
class Greeter {
    void greet() {
        System.out.println("Hello!");
    }
}
public class Main {
    public static void main(String[] args) {
        Greeter greetObj = new Greeter();
        greetObj.greet();
    }
}
```

Output: Hello!

iii) Method that Returns a Value

A method that returns a value using the return keyword.

- Helps in processing and sending back a result.
- The return type of the method must match the type of data being returned.

Example code:

```
class Calculator {
    int multiply(int a, int b) {
        return a * b;
    }
}
public class Main {
    public static void main(String[] args) {
        Calculator calc = new Calculator();
        int result = calc.multiply(5, 10);
        System.out.println("Multiplication: " + result);
    }
}
```

Output:

```
Multiplication: 50
```

Q3. Constructors in Java

i) Automatic Constructor

- If no constructor is provided, Java automatically creates a default constructor.
- This constructor has no parameters and does nothing special other than creating an object.

ii) Default Constructor

- A constructor with no parameters.
- Initializes the object with default or initial values.

Example code:

```
class Car {
    String model;

    Car() {
        model = "Unknown Model";
    }

    void displayModel() {
        System.out.println("Model: " + model);
    }
}
public class Main {
    public static void main(String[] args) {
        Car car = new Car();
        car.displayModel();
    }
}
```

Output:

Model: Unknown Model

iii) Parameterized Constructor

- Constructor that accepts parameters to initialize object properties.

Example code:

```
class Car {
    String model;
    int year;

    Car(String model, int year) {
        this.model = model;
        this.year = year;
    }

    void displayInfo() {
        System.out.println("Model: " + model + ", Year: " + year);
    }
}
public class Main {
    public static void main(String[] args) {
        Car car = new Car("Toyota Corolla", 2019);
        car.displayInfo();
    }
}
```

Output: Model: Toyota Corolla, Year: 2019

Q4. this Keyword in Java

this refers to the current instance of the class.

- Used to refer to instance variables when they are shadowed by method parameters.
- Can be used to call another constructor in the same class.
- Helps avoid ambiguity between instance variables and parameters.

Example code:

```
class Car {
    String model;

    Car(String model) {
        this.model = model;  // this.model refers to the instance variable
    }

    void displayModel() {
        System.out.println("Model: " + this.model);
    }
}
public class Main {
    public static void main(String[] args) {
        Car car = new Car("Tesla Model 3");
        car.displayModel();
    }
}
```

Output: Model: Tesla Model 3

Q5. Garbage Collection in Java

Garbage collection is the process by which Java automatically reclaims memory for objects that are no longer in use.

- Purpose: Free up memory by removing unreferenced objects.
- How: Java uses the gc() method to suggest garbage collection.

Example code:

```
class Car {
    protected void finalize() {
        System.out.println("Object is garbage collected");
    }
}
public class Main {
    public static void main(String[] args) {
        Car car = new Car();
        car = null;
        System.gc();
    }
}
```

Output:

```
Object is garbage collected
```

Q6. Method Overloading

Method overloading allows methods with the same name but different parameters to exist in the same class.

i) Overloading by Number of Parameters

- Methods differ by the number of parameters.

Example code:

```
class Calculator {
    void add(int a, int b) {
        System.out.println("Sum: " + (a + b));
    }

    void add(int a, int b, int c) {
        System.out.println("Sum: " + (a + b + c));
    }
}
public class Main {
    public static void main(String[] args) {
        Calculator calc = new Calculator();
        calc.add(5, 10);
        calc.add(5, 10, 15);
    }
}
```

Output:

```
Sum: 15
Sum: 30
```

ii) Overloading by Data Type

- Methods differ by the data type of parameters.

Example:

```
class Calculator {
    void add(int a, int b) {
        System.out.println("Sum (int): " + (a + b));
    }

    void add(double a, double b) {
        System.out.println("Sum (double): " + (a + b));
    }
}
public class Main {
    public static void main(String[] args) {
        Calculator calc = new Calculator();
        calc.add(5, 10);
        calc.add(5.5, 10.5);
    }
}
```

Op:

```
Sum (int): 15
Sum (double): 16.0
```

iii) Overloading by Sequence of Parameters

- Methods differ by the order of parameters.

Example:

```
class Calculator {
    void display(int a, double b) {
        System.out.println("Int and Double");
    }

    void display(double a, int b) {
        System.out.println("Double and Int");
    }
}
public class Main {
    public static void main(String[] args) {
        Calculator calc = new Calculator();
        calc.display(5, 10.5);
        calc.display(5.5, 10);
    }
}
```

Output:

```
Int and Double
Double and Int
```

Q7. Constructor Overloading

Constructor overloading allows multiple constructors with different parameter lists in the same class

Example code:

```
class Car {
    String model;
    int year;

    // Default constructor
    Car() {
        this.model = "Unknown";
        this.year = 2020;
    }

    // Parameterized constructor
    Car(String model, int year) {
        this.model = model;
        this.year = year;
    }

    void displayInfo() {
        System.out.println("Model: " + model + ", Year: " + year);
    }
}
public class Main {
    public static void main(String[] args) {
        Car car1 = new Car();
        Car car2 = new Car("Ford Mustang", 2021);
        car1.displayInfo();
        car2.displayInfo();
    }
}
```

Output:
Model: Unknown, Year: 2020
Model: Ford Mustang, Year: 2021

Q8. Static Variable & Static Method

Static Variable

A static variable is shared among all instances of a class.

- Belongs to the class, not to individual objects.
- Memory allocation happens only once when the class is loaded.
- All objects of the class share the same static variable.

Example code:

```
class Car {
    static int carCount = 0;  // static variable
    String model;

    Car(String model) {
        this.model = model;
        carCount++;
    }

    void displayCar() {
        System.out.println("Model: " + model);
    }
}
public class Main {
    public static void main(String[] args) {
        Car car1 = new Car("Honda");
        Car car2 = new Car("Toyota");
        System.out.println("Total Cars: " + Car.carCount);  // Accessing static variable
    }
}
```

Output:
Total Cars: 2

Static Method

A static method belongs to the class rather than to instances of the class.

- Can be called without creating an object of the class.
- Can only access static variables directly.
- Used for utility or helper methods that don't need to access instance variables.

Example code:

```
class Calculator {
    static int add(int a, int b) {  // static method
        return a + b;
    }
}
public class Main {
    public static void main(String[] args) {
        int sum = Calculator.add(5, 10);  // calling static method without object
        System.out.println("Sum: " + sum);
    }
}
```

Output: Sum: 15

Q9. Nested and Inner Class in Java

Nested Class
A nested class is a class defined inside another class.

- It logically groups classes that are only used in one place.
- Nested classes can be static or non-static.

Types of Nested Classes:

1. **Static Nested Class:**
   - Declared with the static keyword.
   - Can access static members of the outer class.
2. **Inner Class (Non-static Nested Class):**
   - Not declared as static.
   - Has access to all members (both static and non-static) of the outer class.

Example of Static Nested Class:

```
class OuterClass {
    static String message = "Hello from Outer Class";

    static class NestedClass {
        void displayMessage() {
            System.out.println(message);  // Can access static variables of outer class
        }
    }
}
public class Main {
    public static void main(String[] args) {
        OuterClass.NestedClass nested = new OuterClass.NestedClass();
        nested.displayMessage();
    }
}
```

Output:
Hello from Outer Class

Example of Inner Class:

```
class OuterClass {
    String message = "Hello from Outer Class";

    class InnerClass {
        void displayMessage() {
            System.out.println(message);  // Can access non-static variables of outer class
        }
    }
}
public class Main {
    public static void main(String[] args) {
        OuterClass outer = new OuterClass();
        OuterClass.InnerClass inner = outer.new InnerClass();
        inner.displayMessage();
    }
}
```

Output:
Hello from Outer Class