HelloWorld program in Java

```
public class HelloWorld {
    public static void main(String[] args) {
        System.out.println("Hello, World!");
    }
}
```

Class Declaration:

```
public class HelloWorld {
```

- `public`: An access modifier that allows the class to be accessible from anywhere.
- `class`: A keyword used to declare a class.
- `HelloWorld`: The name of the class. In Java, the class name should match the filename.

Main Method:

```
public static void main(String[] args) {
```

- `public`: An access modifier that allows the method to be accessible from anywhere.
- `static`: A keyword indicating that the method belongs to the class, not instances of the class.
- `void`: The return type of the method, indicating that it does not return any value.
- `main`: The name of the method. It is the entry point of any Java program.
- `String[] args`: The parameter for the main method. It is an array of strings that stores command-line arguments.

Print Statement:

```
System.out.println("Hello, World!");
```

- `System`: A predefined class that provides access to the system.
- `out`: A static member of the `System` class, representing the standard output stream (usually the console).
- `println`: A method of the `PrintStream` class that prints a line of text to the console.
- `"Hello, World!"`: The string to be printed.

Data Types

In programming, a **data type** specifies the kind of data that a variable can hold. Understanding data types is fundamental because it determines how data is stored, manipulated, and used in a program. In Java, data types are categorized into two main groups: primitive and non-primitive (reference) data types.

1. Primitive Data Types

Primitive data types are the basic building blocks in Java. They are predefined by the language and represent single values.

Characteristics of Primitive Data Types:

- **Fixed Size:** Each primitive type has a predefined size and range.
- **Direct Storage:** They hold actual values, not references to objects.
- **No Methods:** Primitive types do not have methods or properties.

Detailed Breakdown:

1. **byte**
   - **Size:** 8-bit
   - **Range:** -128 to 127
   - **Usage:** Suitable for saving space in large arrays, especially when the data is limited to this range.
2. **short**
   - **Size:** 16-bit
   - **Range:** -32,768 to 32,767
   - **Usage:** Used to save memory in large arrays, especially when the data is within this range.
3. **int**
   - **Size:** 32-bit
   - **Range:** -2^31 to 2^31 - 1 (approximately -2.14 billion to 2.14 billion)
   - **Usage:** Default data type for integer values. Commonly used for arithmetic operations.
4. **long**
   - **Size:** 64-bit
   - **Range:** -2^63 to 2^63 - 1 (approximately -9.22 quintillion to 9.22 quintillion)
   - **Usage:** Used for larger integer values when `int` is insufficient.
5. **float**
   - **Size:** 32-bit IEEE 754 floating point
   - **Range:** Approximately ±3.40282347E+38F (6-7 significant decimal digits)
   - **Usage:** Used for floating-point calculations with less precision.
6. **double**
   - **Size:** 64-bit IEEE 754 floating point

    - **Range:** Approximately ±1.79769313486231570E+308 (15 significant decimal digits)
    - **Usage:** Used for more precise floating-point calculations.
7. **char**
    - **Size:** 16-bit
    - **Range:** 0 to 65,535 (Unicode characters)
    - **Usage:** Used to store single characters.
8. **boolean**
    - **Size:** 1-bit
    - **Values:** `true` and `false`
    - **Usage:** Represents logical values, typically used in conditional statements.

2. Non-Primitive (Reference) Data Types

Non-primitive data types are more complex and can be created using classes, interfaces, and arrays. They refer to objects and can store references to data.

Characteristics of Non-Primitive Data Types:

- **Variable Size:** The size can vary depending on the object or array.
- **Indirect Storage:** They store references to objects rather than the actual data.
- **Methods and Properties:** They can have methods and properties.

Detailed Breakdown:

1. **Classes:**
    - **Description:** A blueprint for creating objects. Classes define the data and behavior of objects.

data-attributes/variables

behavior - Method/function

Example:

```
public class Car {

    String model;

    int year;

    void startEngine() {
```

```
        System.out.println("Engine started.");
    }
}
```

2. **Objects:**
    - **Description:** Instances of classes. Objects encapsulate data and functionality defined by their class.

      Example:

      ```
      Car myCar = new Car();
      myCar.model = "Toyota";
      myCar.year = 2020;
      myCar.startEngine();
      ```

3. **Arrays:**
    - **Description:** A container that holds a fixed number of values of a single type. Arrays can be of primitive or reference types.

      Example:

      ```
      int[] numbers = {1, 2, 3, 4, 5};
      String[] names = {"Alice", "Bob", "Charlie"};
      ```

4. **Interfaces:**
    - **Description:** Abstract types that define a set of methods without implementing them. Classes implement interfaces to provide specific behaviors.

      Example:

      ```
      public interface Printable {
          void print();
      }
      public class Document implements Printable {
          public void print() {
      ```

```
            System.out.println("Printing document.");
        }
    }
```

5. **Enumerations (Enums):**
    - **Description:** Special classes that represent a group of constants (unchangeable variables).

      Example:

      ```
      public enum Day {
          SUNDAY, MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY,
      SATURDAY
      }

      Day today = Day.MONDAY;
      ```

6. **Strings:**
    - **Description:** A sequence of characters. In Java, `String` is a class, not a primitive type, and represents a sequence of characters.

      Example:

      ```
      String greeting = "Hello, World!";
      ```

- **Primitive Data Types:** Include `byte`, `short`, `int`, `long`, `float`, `double`, `char`, and `boolean`. They represent simple values, have fixed sizes, and are efficient in terms of performance and memory usage.
- **Non-Primitive (Reference) Data Types:** Include classes, objects, arrays, interfaces, enums, and strings. They refer to complex data structures, can hold multiple values, and support methods and properties.

Java Access Modifiers

Access modifiers in Java are keywords used to specify the visibility or accessibility of classes, methods, variables, and constructors. They control how these elements can be accessed from other classes. Understanding access modifiers is crucial for encapsulation, one of the core principles of object-oriented programming.

Java provides four main access modifiers:

1. `public`
2. `protected`
3. `default` (no modifier)
4. `private`

1. `public`

- **Definition:** The `public` access modifier indicates that the member (class, method, variable) is accessible from any other class or package.
- **Usage:**
    - **Classes:** A `public` class can be accessed from any other class.
    - **Methods:** A `public` method can be called from any other class.
    - **Variables:** A `public` variable can be accessed and modified from any other class.
    - **Constructors:** A `public` constructor can be used to create instances of the class from anywhere.

      Example:

```
public class PublicExample {

    public int publicValue;

    public void publicMethod() {

        System.out.println("Public method.");

    }

}
```

2. `protected`

- **Definition:** The `protected` access modifier allows access to the member from within the same package and by subclasses (even if they are in different packages).
- **Usage:**
  - **Classes:** The `protected` access modifier cannot be applied to top-level classes.
  - **Methods:** A `protected` method can be accessed within the same package and by subclasses.
  - **Variables:** A `protected` variable can be accessed within the same package and by subclasses.
  - **Constructors:** A `protected` constructor allows instantiation within the same package and by subclasses.

    Example:

```
public class BaseClass {

    protected int protectedValue;

    protected void protectedMethod() {

        System.out.println("Protected method.");

    }

}

public class SubClass extends BaseClass {

    void accessProtected() {

        protectedValue = 10; // Accessible

        protectedMethod();  // Accessible

    }

}
```

3. `default` (Package-Private)

- **Definition:** When no access modifier is specified, the member has package-private (or default) access. It is accessible only within the same package.
- **Usage:**
  - **Classes:** A package-private class is only accessible within its own package.
  - **Methods:** A package-private method can be accessed only within the same package.
  - **Variables:** A package-private variable can be accessed only within the same package.
  - **Constructors:** A package-private constructor can be used only within the same package.

    Example:

```
class DefaultExample {

    int defaultValue;

    void defaultMethod() {

        System.out.println("Default method.");

    }

}
```

4. `private`

- **Definition:** The `private` access modifier restricts access to the member only within the same class. It is not accessible from outside the class, not even by subclasses.
- **Usage:**
  - **Classes:** The `private` access modifier cannot be applied to top-level classes.
  - **Methods:** A `private` method can be accessed only within its own class.
  - **Variables:** A `private` variable can be accessed only within its own class.
  - **Constructors:** A `private` constructor is used for singleton patterns and to prevent instantiation from outside the class.

    Example:

```
public class PrivateExample {

    private int privateValue;
```

```
        private void privateMethod() {
            System.out.println("Private method.");
        }
        public void accessPrivate() {
            privateValue = 10; // Accessible
            privateMethod();  // Accessible
        }
    }
```

Class: A blueprint for creating objects. Classes define the data and behavior of objects.

Object: Instances of classes. Objects encapsulate data and functionality defined by their class.

```
    /* Java Naming convention
     Class Name - start with capital
     Variable should be start with alphabets, it should not
have special chars, small letter, if you have two words in
variable name, it should separate out by underscore(_)
    Method - camelCase
    package name should be in small letter*/
public class Car{
        //attribute/properties/variables/fields
        String model;
        int year;
        String color;

        //behavior/method/function
        public void startEngine(){
            System.out.println("Engine is starting.. ");
        }
        public void stopEngine(){
```

```
            System.out.println("Engine is stopping.. ");
      }
}

public class CarObj{
      public static void main(String args[]){
                  //creating the object of class
                  Car myCar = new Car();

                  myCar.model = "Audi";
                  myCar.year = 2023;
                  myCar.color = "Black";

                  myCar.startEngine();
                  myCar.stopEngine();

                  System.out.println(myCar.model);
                  System.out.println(myCar.year);
                  System.out.println(myCar.color);
      }
}
```