Step 1: Create a Java File

1. Open **Notepad** (or any code editor like **VS Code** or **IntelliJ IDEA**).
2. Type this code:

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

3. Save the file as `HelloWorld.java` in a folder, e.g., `C:\JavaPrograms`.

Compile the Program

1. Open **Command Prompt**.
2. Navigate to your folder:

```
cd C:\JavaPrograms
```

3. Compile the Java program:

```
javac HelloWorld.java
```

- If everything is correct, this will create a file called `HelloWorld.class`.
- This `.class` file contains the **bytecode** that Java can run.

Run the Program

```
java HelloWorld
```

Output:

```
Hello, World!
```

Before writing code, you need:

1. **Java Development Kit (JDK)** – Download and install the latest JDK from Oracle or OpenJDK.
2. **IDE (Optional but Recommended)** – Use **IntelliJ IDEA**, **Eclipse IDE**, or **VS Code**.

After installing JDK, verify installation:

```
java -version
javac -version
```

You should see versions printed.

Step 2: Your First Java Program

Java programs start with a **class** and a `main` method.

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

Explanation

- `public class HelloWorld` → Defines a class named `HelloWorld`.
- `public static void main(String[] args)` → The entry point of the program. Every Java program starts here.
- `System.out.println("Hello, World!");` → Prints text to the console.

How to run:

```
javac HelloWorld.java   # Compiles the code
java HelloWorld         # Runs the program
```

Output:

```
Hello, World!
```

Step 3: Variables and Data Types

Java is **statically typed**, so you must declare variable types.

```
public class VariablesExample {
    public static void main(String[] args) {
        int age = 25;          // Integer
        double height = 5.9;   // Decimal number
        char grade = 'A';      // Single character
        String name = "Alice"; // String of text
        boolean isStudent = true; // true/false

        System.out.println("Name: " + name);
        System.out.println("Age: " + age);
        System.out.println("Height: " + height);
        System.out.println("Grade: " + grade);
        System.out.println("Student? " + isStudent);
    }
}
```

Explanation:

- `int`, `double`, `char`, `String`, `boolean` → Basic data types.
- `+` operator concatenates strings.

Step 4: Conditional Statements

Java lets you control the flow using `if`, `else if`, and `else`.

```
public class ConditionExample {
    public static void main(String[] args) {
        int number = 10;

        if (number > 0) {
            System.out.println("Positive number");
        } else if (number < 0) {
            System.out.println("Negative number");
        } else {
            System.out.println("Zero");
        }
    }
}
```

Explanation:

- `if` checks the condition.
- `else if` checks another condition if the first is false.
- `else` runs if all conditions fail.

Step 5: Loops

Loops allow repeating actions.

For Loop

```
for (int i = 1; i <= 5; i++) {
    System.out.println("Count: " + i);
}
```

While Loop

```
int i = 1;
while (i <= 5) {
    System.out.println("Count: " + i);
    i++;
}
```

Explanation:

- `for` loop: Initializes `i`, checks condition, increments.
- `while` loop: Checks condition first, repeats until false.

Step 6: Arrays

Arrays store multiple values in a single variable.

```
public class ArrayExample {
    public static void main(String[] args) {
        int[] numbers = {1, 2, 3, 4, 5};

        for (int i = 0; i < numbers.length; i++) {
            System.out.println("Number: " + numbers[i]);
        }
    }
}
```

Explanation:

- `numbers.length` → Size of the array.
- Indexing starts at 0.

Step 7: Methods (Functions)

Methods help you organize reusable code.

```
public class MethodExample {
    public static void main(String[] args) {
        int sum = addNumbers(5, 10);
        System.out.println("Sum: " + sum);
    }

    // Method to add two numbers
    public static int addNumbers(int a, int b) {
        return a + b;
    }
}
```

Explanation:

- `public static int addNumbers(int a, int b)` → Method that returns an integer.
- `return a + b;` → Sends result back to the caller.

Step 8: Object-Oriented Programming (OOP) Basics

Java is **object-oriented**, so everything revolves around **classes** and **objects**.

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

    // Constructor
    Car(String c, int y) {
        color = c;
        year = y;
    }

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

public class OOPExample {
    public static void main(String[] args) {
        Car myCar = new Car("Red", 2020);
        myCar.displayInfo();
    }
}
```

Explanation:

- `class Car` → Blueprint for a car object.
- `Constructor` → Initializes object values.
- `displayInfo()` → Method belonging to the object.
- `myCar` → Instance of `Car`.

Advanced Java concept you listed. I'll make sure it's beginner-friendly but thorough.

---

1. Inheritance & Polymorphism

Code Example

```
// Parent class
class Animal {
    void sound() {
        System.out.println("Animal makes a sound");
    }
}

// Child class
class Dog extends Animal {
    // Method overriding
    @Override
    void sound() {
        System.out.println("Dog barks");
    }
}

public class InheritancePolymorphismExample {
    public static void main(String[] args) {
        Animal myAnimal = new Animal();
        Animal myDog = new Dog(); // Polymorphism: parent reference, child
object

        myAnimal.sound(); // Calls parent method
        myDog.sound();    // Calls overridden method in Dog class
    }
}
```

Explanation

- `class Dog extends Animal` → `Dog` inherits from `Animal`.
- **Overriding**: `Dog` provides its own implementation of `sound()`.
- **Polymorphism**: A parent reference (`Animal`) can point to a child object (`Dog`), and Java decides at runtime which `sound()` to call.

Output

```
Animal makes a sound
Dog barks
```

---

2. Interfaces and Abstract Classes

Code Example

```
// Interface
interface Vehicle {
    void start(); // Abstract method
}

// Abstract class
abstract class Car implements Vehicle {
    String brand;

    Car(String brand) {
        this.brand = brand;
    }

    // Concrete method
    void displayBrand() {
        System.out.println("Brand: " + brand);
    }
}

// Concrete class
class Tesla extends Car {
    Tesla(String brand) {
        super(brand);
    }

    // Implement interface method
    @Override
    public void start() {
        System.out.println("Tesla starts silently");
    }
}

public class InterfaceAbstractExample {
    public static void main(String[] args) {
        Tesla myTesla = new Tesla("Tesla Model S");
        myTesla.displayBrand();
        myTesla.start();
    }
}
```

Explanation

- `interface Vehicle` → defines a contract.
- `abstract class Car` → can have both abstract (not implemented) and concrete methods.
- `class Tesla` → must implement all abstract/interface methods.

Output

```
Brand: Tesla Model S
Tesla starts silently
```

---

3. Collections (ArrayList & HashMap)

Code Example

```
import java.util.ArrayList;
import java.util.HashMap;

public class CollectionsExample {
    public static void main(String[] args) {
        // ArrayList
        ArrayList<String> fruits = new ArrayList<>();
        fruits.add("Apple");
        fruits.add("Banana");
        fruits.add("Orange");

        System.out.println("Fruits list:");
        for (String fruit : fruits) {
            System.out.println(fruit);
        }

        // HashMap
        HashMap<Integer, String> students = new HashMap<>();
        students.put(101, "Alice");
        students.put(102, "Bob");
        students.put(103, "Charlie");

        System.out.println("\nStudents map:");
        for (Integer id : students.keySet()) {
            System.out.println("ID: " + id + ", Name: " + students.get(id));
        }
    }
}
```

Explanation

- **ArrayList** → Resizable array, stores elements in order.
- **HashMap** → Stores key-value pairs. Keys must be unique.

Output

```
Fruits list:
Apple
Banana
Orange

Students map:
ID: 101, Name: Alice
ID: 102, Name: Bob
ID: 103, Name: Charlie
```

---

4. Exception Handling (try-catch)

Code Example

```
public class ExceptionExample {
    public static void main(String[] args) {
        int[] numbers = {1, 2, 3};

        try {
            System.out.println(numbers[5]); // This will throw ArrayIndexOutOfBoundsException
        } catch (ArrayIndexOutOfBoundsException e) {
            System.out.println("Error: Index out of bounds!");
        } finally {
            System.out.println("This block always executes");
        }

        System.out.println("Program continues after exception handling");
    }
}
```

Explanation

- `try` → code that might throw an exception.
- `catch` → handles the exception gracefully.
- `finally` → always runs, used for cleanup (optional).

Output

```
Error: Index out of bounds!
This block always executes
Program continues after exception handling
```

---

5. File I/O and Working with Data

Code Example

```
import java.io.File;
import java.io.FileWriter;
import java.io.FileReader;
import java.io.IOException;

public class FileIOExample {
    public static void main(String[] args) {
        try {
            // Write to file
            FileWriter writer = new FileWriter("example.txt");
            writer.write("Hello, Java File I/O!\n");
            writer.write("This is a second line.");
            writer.close();

            // Read from file
            File file = new File("example.txt");
            FileReader reader = new FileReader(file);
            int ch;
```

```
            System.out.println("File content:");
            while ((ch = reader.read()) != -1) {
                System.out.print((char) ch);
            }
            reader.close();
        } catch (IOException e) {
            System.out.println("An error occurred: " + e.getMessage());
        }
    }
}
```

Explanation

- **FileWriter** → Writes text to a file.
- **FileReader** → Reads text from a file character by character.
- `try-catch` → Handles potential IO exceptions.

Output

```
File content:
Hello, Java File I/O!
This is a second line.
```