1. History and Features of Java

History:

- **Developed by:** Sun Microsystems (initiated by James Gosling)
- **Released in:** 1995
- **Acquired by:** Oracle Corporation in 2010

Features:

- **Object-Oriented:** Everything is treated as an object.
- **Platform Independent:** Write once, run anywhere (WORA) using the Java Virtual Machine (JVM).
- **Simple and Familiar:** Easy to learn if you're familiar with C/C++.
- **Secure:** Provides a secure runtime environment.
- **Robust:** Strong memory management and exception handling.
- **Multithreaded:** Supports concurrent execution of two or more threads.
- **High Performance:** Just-In-Time compiler helps in improving performance.

- **Distributed:** Facilitates distributed computing with EJB and RMI.
- **Dynamic:** Capable of adapting to an evolving environment.

2. Installation and Setup

Installing JDK

1. **Download JDK:**
   - Go to the official Oracle website or OpenJDK website.
   - Download the appropriate JDK version for your operating system.
2. **Install JDK:**
   - Run the installer and follow the on-screen instructions.
   - Set the installation path (usually `C:\Program Files\Java\jdk-xx.x.x`).
3. **Set Environment Variables (Windows):**
   - **JAVA_HOME:** Set to the JDK installation path.
   - **Path:** Add `JAVA_HOME\bin` to the system `Path`.

On macOS/Linux:

- Add the following lines to your shell profile (`.bash_profile`, `.zshrc`, etc.):

```
export JAVA_HOME=/path/to/jdk
export PATH=$JAVA_HOME/bin:$PATH
```

4. **Verify Installation:**
   - Open a terminal/command prompt.
   - Run `java -version` and `javac -version` to check the installation.

Setting up IDE (Eclipse, IntelliJ IDEA, or VS Code)

1. **Download VS Code:**
   - Go to the official VS Code website and download the installer.
2. **Install VS Code:**
   - Run the installer and follow the instructions.
3. **Install Java Extension Pack:**
   - Open VS Code and go to the Extensions view.
   - Search for “Java Extension Pack” and install it.
4. **Setup Project:**
   - Create a new folder for your project.
   - Open the folder in VS Code and create a new Java file.

3. First Java Program

Writing and Executing a Simple Program

1. **Open your IDE.**
2. **Create a new Java Class:**
   - In Eclipse/IntelliJ, right-click on the `src` folder and select `New > Class`.
   - In VS Code, create a new file with a `.java` extension.
3. **Write the following code:**

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

4. **Save the file as `HelloWorld.java`.**

Understanding the Main Method

- **public:** Accessible from anywhere.
- **static:** Belongs to the class, not instances of it.
- **void:** Does not return any value.
- **main:** The entry point of the application.
- **String[] args:** Command-line arguments passed to the program.

Running the Program

- **In Eclipse/IntelliJ:**
  - Right-click on the file and select Run `As > Java Application`.
- **In VS Code:**
  - Open the terminal in VS Code.
  - Compile the program: `javac HelloWorld.java`
  - Run the program: `java HelloWorld`

You should see the output:

```
Hello, World!
```

Understanding the Main Method

The main method is the entry point of any Java application. Let's break down the components of the main method:

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

- **public:** This is an access modifier, which means that the method is accessible from outside the class. Since the JVM needs to access the `main` method, it must be public.
- **static:** The `main` method is static, which means it belongs to the class rather than an instance of the class. This allows the JVM to invoke the `main` method without creating an instance of the class.
- **void:** This indicates that the `main` method does not return any value.
- **main:** This is the name of the method. The JVM looks for the `main` method as the starting point of the application.

- **String[] args:** This is a parameter passed to the `main` method. It is an array of `String` objects, which can hold command-line arguments passed to the program when it is executed. For example, if you run `java HelloWorld arg1 arg2`, `args` will contain `["arg1", "arg2"]`.
- **System.out.println("Hello, World!");** This statement prints the string "Hello, World!" to the console. `System.out` is an instance of `PrintStream`, and `println` is a method of `PrintStream` that prints the argument passed to it followed by a new line.

Java Basics

Syntax and Data Types

Syntax:

- **Class Definition:** A Java program typically starts with a class definition.

```
public class MyClass {
    // code
}
```

- **Main Method:** The entry point of any Java application.

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

- **Statements:** Each statement in Java ends with a semicolon (`;`).

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

Data Types:
Java has two categories of data types: Primitive Data Types and Reference/Object Data Types.

- **Primitive Data Types:** There are 8 primitive data types in Java.
  - **byte:** 8-bit integer, ranges from -128 to 127.
  - **short:** 16-bit integer, ranges from -32,768 to 32,767.
  - **int:** 32-bit integer, ranges from -2^31 to 2^31-1.
  - **long:** 64-bit integer, ranges from -2^63 to 2^63-1.
  - **float:** Single-precision 32-bit floating point.
  - **double:** Double-precision 64-bit floating point.
  - **char:** 16-bit Unicode character.
  - **boolean:** Has only two possible values: `true` and `false`.
- **Reference Data Types:** These are objects and arrays.
  - Examples include `String`, `Arrays`, `Classes`, etc.

Variables and Data Types

Variables in Java must be declared before they are used. A variable declaration includes the data type and the variable name.

```
int number;         // Declares an
integer variable named number
float decimal;      // Declares a float
variable named decimal
char letter;        // Declares a
character variable named letter
boolean isTrue;     // Declares a boolean
variable named isTrue
```

You can also initialize variables at the time of declaration:

```
int number = 10;        // Declares and
initializes number with 10
float decimal = 5.5f;  // Declares and
initializes decimal with 5.5
char letter = 'A';      // Declares and
initializes letter with 'A'
boolean isTrue = true; // Declares and
initializes isTrue with true
```

Type Casting

Type casting is converting a variable from one data type to another. There are two types of casting:

- **Implicit Casting (Widening):** Automatic conversion from a smaller to a larger type.

```
int num = 10;
double doubleNum = num;  // Implicit casting: int to double
```

- **Explicit Casting (Narrowing):** Manual conversion from a larger to a smaller type.

```
double doubleNum = 10.5;
int num = (int) doubleNum;  // Explicit casting: double to int
```

Constants

Constants are variables whose values cannot be changed once assigned. In Java, constants are defined using the `final` keyword.

```
final int MAX_VALUE = 100;
final float PI = 3.14159f;
final String GREETING = "Hello, World!";
```

The `final` keyword ensures that the value of the variable cannot be modified. Constants are typically named in uppercase letters to distinguish them from regular variables.

Example Program Demonstrating the Above Concepts

Example Program Demonstrating the Above Concepts

```
public class JavaBasics {
    public static void main(String[]
args) {
        // Variables and Data Types
        int number = 10;
        float decimal = 5.5f;
        char letter = 'A';
        boolean isTrue = true;

        // Implicit Casting
        double doubleNum = number;
        System.out.println("Implicit
Casting (int to double): " + doubleNum);

        // Explicit Casting
        int intNum = (int) decimal;
        System.out.println("Explicit
Casting (float to int): " + intNum);

        // Constants
        final int MAX_VALUE = 100;
        final float PI = 3.14159f;
        final String GREETING = "Hello,
World!";

        System.out.println("Constants: "
+ MAX_VALUE + ", " + PI + ", " +
GREETING);
```

```
        // Using variables
        System.out.println("Number: " +
number);
        System.out.println("Decimal: " +
decimal);
        System.out.println("Letter: " +
letter);
        System.out.println("Boolean: " +
isTrue);
    }
}
```

Output:

```
Implicit Casting (int to double): 10.0
Explicit Casting (float to int): 5
Constants: 100, 3.14159, Hello, World!
Number: 10
Decimal: 5.5
Letter: A
Boolean: true
```

Operators in Java

Arithmetic Operators

Arithmetic operators are used to perform mathematical operations like addition, subtraction, multiplication, division, and modulus.

- **Addition (+):** Adds two operands.
- **Subtraction (–):** Subtracts the right-hand operand from the left-hand operand.
- **Multiplication (∗):** Multiplies two operands.
- **Division (/):** Divides the left-hand operand by the right-hand operand.
- **Modulus (%):** Returns the remainder of the division of the left-hand operand by the right-hand operand.

Example:

```
int a = 10;
int b = 5;
int sum = a + b;     // 15
int difference = a - b;  // 5
int product = a * b; // 50
int quotient = a / b;   // 2
int remainder = a % b;  // 0
```

Relational Operators

Relational operators are used to establish relations between operands.

- **Equal to (==):** Checks if two operands are equal.
- **Not equal to (!=):** Checks if two operands are not equal.
- **Greater than (>):** Checks if the left operand is greater than the right operand.
- **Less than (<):** Checks if the left operand is less than the right operand.
- **Greater than or equal to (>=):** Checks if the left operand is greater than or equal to the right operand.
- **Less than or equal to (<=):** Checks if the left operand is less than or equal to the right operand.

Example:

```
int x = 10;
int y = 5;

boolean isEqual = x == y;        // false
boolean isNotEqual = x != y;     // true
boolean isGreater = x > y;       // true
boolean isLess = x < y;          // false
boolean isGreaterOrEqual = x >= y;  //
true
boolean isLessOrEqual = x <= y;     //
false
```

Logical Operators

Logical operators are used to combine multiple conditions.

- **Logical AND (&&):** Returns true if both operands are true.
- **Logical OR (||):** Returns true if at least one of the operands is true.
- **Logical NOT (!):** Reverses the logical state of its operand.

Example:

```
boolean condition1 = true;
boolean condition2 = false;

boolean result1 = condition1 &&
condition2;  // false (true && false)
boolean result2 = condition1 ||
condition2;  // true (true || false)
boolean result3 = !condition1;   //
false (!true)
```

Bitwise Operators

Bitwise operators perform operations at the bit level.

- **Bitwise AND (&):** Performs bitwise AND operation.
- **Bitwise OR (|):** Performs bitwise OR operation.
- **Bitwise XOR (^):** Performs bitwise XOR (exclusive OR) operation.
- **Bitwise NOT (~):** Inverts all the bits of the operand.
- **Left Shift (<<):** Shifts bits to the left by a specified number of positions.
- **Right Shift (>>):** Shifts bits to the right by a specified number of positions.
- **Unsigned Right Shift (>>>):** Shifts bits to the right, filling leftmost bits with zero.

Example:

```
int num1 = 5;   // binary: 101
int num2 = 3;   // binary: 011

int bitwiseAnd = num1 & num2;   // 1
(101 & 011)
int bitwiseOr = num1 | num2;    // 7
(101 | 011)
int bitwiseXor = num1 ^ num2;   // 6
(101 ^ 011)
int bitwiseNot = ~num1;         // -6
(inverts all bits of num1)
int leftShift = num1 << 1;      // 10
(shifts bits of num1 left by 1)
int rightShift = num1 >> 1;     // 2
(shifts bits of num1 right by 1)
```

Assignment Operators

Assignment operators are used to assign values to variables.

- **Assignment (=):** Assigns the value on the right-hand side to the variable on the left-hand side.
- **Addition Assignment (+=):** Adds the right operand to the left operand and assigns the result to the left operand.
- **Subtraction Assignment (–=):** Subtracts the right operand from the left operand and assigns the result to the left operand.
- **Multiplication Assignment (*=):** Multiplies the right operand with the left operand and assigns the result to the left operand.
- **Division Assignment (/=):** Divides the left operand by the right operand and assigns the result to the left operand.
- **Modulus Assignment (%=):** Takes modulus using two operands and assigns the result to the left operand.

Example:

```
int num = 10;
num += 5;   // equivalent to: num = num
+ 5; (num becomes 15)
num -= 3;   // equivalent to: num = num
- 3; (num becomes 12)
num *= 2;   // equivalent to: num = num
* 2; (num becomes 24)
num /= 4;   // equivalent to: num =
num / 4; (num becomes 6)
num %= 5;   // equivalent to: num = num
% 5; (num becomes 1)
```

Unary Operators

Unary operators operate on a single operand.

- **Unary Plus (+):** Indicates a positive value (usually redundant).
- **Unary Minus (–):** Negates the value of its operand.
- **Increment (++):** Increases the value of its operand by 1.
- **Decrement (––):** Decreases the value of its operand by 1.
- **Logical NOT (!):** Reverses the logical state of its operand.

Example:

```
int x = 10;
int y = -x;      // y becomes -10 (unary
minus)
x++;             // x becomes 11
(increment)
y--;             // y becomes -11
(decrement)
boolean isTrue = !true;   // isTrue
becomes false (logical NOT)
```

Control Statements

Conditional Statements

Conditional statements allow you to execute a block of code based on the evaluation of a condition.

- **if Statement:**

```
int x = 10;
if (x > 5) {
    System.out.println("x is greater
than 5");
}
```

- **if-else Statement:**

```
int y = 3;
if (y % 2 == 0) {
    System.out.println("y is even");
} else {
    System.out.println("y is odd");
}
```

- **if-else-if ladder:**

```
int grade = 75;
if (grade >= 90) {
    System.out.println("A");
} else if (grade >= 80) {
    System.out.println("B");
} else if (grade >= 70) {
    System.out.println("C");
} else {
    System.out.println("F");
}
```

- **switch Statement:**

```
int day = 2;
switch (day) {
    case 1:
        System.out.println("Monday");
        break;
    case 2:
        System.out.println("Tuesday");
        break;
    // cases for other days
    default:
        System.out.println("Invalid
day");
}
```

Looping Statements

Looping statements allow you to execute a block of code repeatedly.

- **for Loop:**

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

- **while Loop:**

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

- **do-while Loop:**

```
int j = 1;
do {
    System.out.println(j);
    j++;
} while (j <= 5);
```

Object-Oriented Programming (OOP) Concepts in Java

Classes and Objects

Classes in Java are blueprints or templates that define the structure and behavior of objects.

- **Defining a Class:**

```
// Class definition
public class Car {
    // Fields (attributes)
    String brand;
    int year;

    // Methods (behaviors)
    void displayInfo() {
        System.out.println("Brand: " +
brand + ", Year: " + year);
    }
}
```

- **Creating Objects:**

Objects are instances of classes that are created using the new keyword.

```
public class Main {
    public static void main(String[]
args) {
        // Creating objects
        Car car1 = new Car();
        Car car2 = new Car();

        // Accessing fields and methods
        car1.brand = "Toyota";
        car1.year = 2020;
        car1.displayInfo();  // Output:
Brand: Toyota, Year: 2020

        car2.brand = "Honda";
        car2.year = 2021;
        car2.displayInfo();  // Output:
Brand: Honda, Year: 2021
    }
}
```

Methods

Methods are functions defined within a class that perform specific tasks.

- **Method Definition and Calling:**

```
public class Calculator {
    // Method definition
    public int add(int a, int b) {
        return a + b;
    }

    public static void main(String[]
args) {
        // Method calling
        Calculator calc = new
Calculator();
        int result = calc.add(5, 3);  //
result will be 8
        System.out.println("Addition
Result: " + result);
    }
}
```

- **Method Overloading:**

Method overloading allows a class to have more than one method having the same name but different parameters.

```
public class Calculator {
    public int add(int a, int b) {
        return a + b;
    }

    // Overloaded method
    public double add(double a, double
b) {
        return a + b;
    }

    public static void main(String[]
args) {
        Calculator calc = new
Calculator();
        int result1 = calc.add(5,
3);          // calls int add(int, int)
        double result2 = calc.add(2.5,
3.5);   // calls double add(double,
double)

        System.out.println("Result 1: "
+ result1);
        System.out.println("Result 2: "
+ result2);
    }
}
```

Constructors

Constructors are special methods used to initialize objects. They have the same name as the class and are called when an object of the class is created.

- **Default Constructor:**

```
public class Car {
    String brand;
    int year;

    // Default constructor
    public Car() {
        brand = "Unknown";
        year = 0;
    }
}
```

- **Parameterized Constructor:**

```
public class Car {
    String brand;
    int year;

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

Inheritance

Inheritance allows one class (subclass/child class) to inherit the properties and behaviors of another class (superclass/parent class).

- **Superclass (Parent Class):**

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

- **Subclass (Child Class):**

```
public class Dog extends Animal {
    void sound() {
        System.out.println("Dog barks");
    }
}
```

- **Method Overriding:**

Subclasses can override methods defined in their superclass to provide specific implementations.

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

public class Dog extends Animal {
    @Override
    void sound() {
        System.out.println("Dog barks");
    }
}
```

- **super Keyword:**

The super keyword is used to access methods or constructors of the superclass from the subclass.

```
public class Animal {
    String color;

    Animal(String color) {
        this.color = color;
    }
}

public class Dog extends Animal {
    String breed;

    Dog(String color, String breed) {
        super(color);  // calling
superclass constructor
        this.breed = breed;
    }
}
```

Polymorphism

Polymorphism means the ability of a single method to perform different operations based on the object it is acting upon.

- **Compile-time Polymorphism (Method Overloading):**

```
public class Calculator {
    public int add(int a, int b) {
        return a + b;
    }

    public double add(double a, double
b) {
        return a + b;
    }
}
```

- **Runtime Polymorphism (Method Overriding):**

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

public class Dog extends Animal {
    @Override
    void sound() {
        System.out.println("Dog barks");
    }
}
```

Encapsulation

Encapsulation refers to the bundling of data (fields) and methods that operate on the data into a single unit (class).

- **Access Modifiers:**

Access modifiers control the visibility of classes, fields, constructors, and methods.

  - **private:** Accessible only within the same class.
  - **default (no modifier):** Accessible only within the same package.
  - **protected:** Accessible within the same package and by subclasses.
  - **public:** Accessible from anywhere.

- **Getters and Setters:**

Getters and setters are methods used to access and update the values of private fields.

```java
public class Person {
    private String name;
    private int age;

    // Getter for name
    public String getName() {
        return name;
    }

    // Setter for name
    public void setName(String name) {
        this.name = name;
    }

    // Getter for age
    public int getAge() {
        return age;
    }

    // Setter for age
    public void setAge(int age) {
        this.age = age;
    }
}
```

Abstraction

Abstraction refers to hiding the implementation details and showing only the essential features of an object.

- **Abstract Classes:**
Abstract classes cannot be instantiated on their own and can contain abstract methods (methods without a body).

```
abstract class Shape {
    abstract void draw();  // abstract
method
}

class Circle extends Shape {
    void draw() {
        System.out.println("Drawing a
circle");
    }
}
```

- **Interfaces:**

Interfaces are similar to abstract classes but can only contain constants and abstract methods.

```
interface Animal {
    void sound();  // abstract method
}

class Dog implements Animal {
    public void sound() {
        System.out.println("Dog barks");
    }
}
```

Advanced Java Concepts

Let's delve into each of these advanced Java concepts:

Exception Handling

Exception handling in Java is used to handle runtime errors (exceptions) gracefully, preventing abrupt termination of programs.

- **Try, Catch, and Finally Blocks:**

```
try {
    // code that may throw an exception
} catch (ExceptionType1 ex1) {
    // handle ExceptionType1
} catch (ExceptionType2 ex2) {
    // handle ExceptionType2
} finally {
    // optional block, executes
regardless of exception occurrence
}
```

- **Throw and Throws:**
  - **`throw`** is used to explicitly throw an exception.
  - **`throws`** is used in method signature to declare exceptions that a method can throw.

```
public void withdraw(int amount) throws
InsufficientFundsException {
    if (balance < amount) {
        throw new
InsufficientFundsException("Not enough
balance");
    }
    // withdraw logic
}
```

- **Custom Exceptions:**

```
// Custom exception class
public class InsufficientFundsException
extends Exception {
    public
InsufficientFundsException(String
message) {
        super(message);
    }
}
```

Collections Framework

The **Collections Framework** in Java provides a unified architecture for manipulating and storing collections of objects.

- **Interfaces:**
  - **List:** Ordered collection (e.g., ArrayList, LinkedList).
  - **Set:** Collection with no duplicate elements (e.g., HashSet, TreeSet).
  - **Map:** Key-value pair collection (e.g., HashMap, TreeMap).
- **Classes:**
  - **ArrayList:** Resizable array implementation of List.
  - **LinkedList:** Doubly-linked list implementation of List.
  - **HashSet:** Hash table-based implementation of Set.
  - **TreeSet:** Red-black tree implementation of Set.
  - **HashMap:** Hash table-based implementation of Map.
  - **TreeMap:** Red-black tree implementation of Map.

- **Iterators and Enhanced for Loop:**

```
List<String> names = new ArrayList<>();
names.add("Alice");
names.add("Bob");

// Using Iterator
Iterator<String> iterator =
names.iterator();
while (iterator.hasNext()) {
    String name = iterator.next();
    System.out.println(name);
}

// Using Enhanced for Loop
for (String name : names) {
    System.out.println(name);
}
```

Java I/O

Java I/O (Input/Output) deals with reading and writing data from/to different sources.

- **File Handling:**

```
File file = new File("example.txt");
```

- **Byte and Character Streams:**
  - **Byte Streams:** Deal with raw binary data (e.g., FileInputStream, FileOutputStream).
  - **Character Streams:** Deal with Unicode characters (e.g., FileReader, FileWriter).
- **BufferedReader and BufferedWriter:**

```
try (BufferedReader br = new
BufferedReader(new
FileReader("file.txt"))) {
    String line;
    while ((line = br.readLine()) !=
null) {
        System.out.println(line);
    }
} catch (IOException e) {
    e.printStackTrace();
}
```

Generics

Generics provide a way to parameterize types in Java, enabling you to create classes, interfaces, and methods that operate on objects of various types.

- **Generic Classes and Methods:**

```
public class Box<T> {
    private T value;

    public void setValue(T value) {
        this.value = value;
    }

    public T getValue() {
        return value;
    }
}

Box<Integer> intBox = new Box<>();
intBox.setValue(10);
int value = intBox.getValue();  // value
will be 10
```

- **Bounded Types:**

```
public class Box<T extends Number> {
    // T must be a subclass of Number
}
```

Multithreading

Multithreading in Java allows concurrent execution of two or more parts of a program for maximum utilization of CPU.

- **Creating Threads:**
  - **Extending Thread class:**

```
class MyThread extends Thread {
    public void run() {
        // thread logic
    }
}
```

-
  - **Implementing Runnable interface:**

```
class MyRunnable implements Runnable {
    public void run() {
        // thread logic
    }
}
```

- **Thread Lifecycle:**
  - **New:** Thread is created.
  - **Runnable:** Thread is ready to run.
  - **Running:** Thread is executing.
  - **Blocked/Waiting:** Thread is waiting for a resource.
  - **Dead/Terminated:** Thread has completed execution.
- **Synchronization:**

```
synchronized void synchronizedMethod() {
    // synchronized block
}
```

Lambda Expressions

Lambda expressions provide a concise way to represent anonymous functions (functions without a name).

- **Functional Interfaces:**

Interfaces with a single abstract method are called functional interfaces.

```
@FunctionalInterface
public interface MyInterface {
    void myMethod();
}
```

- **Lambda Expression Syntax:**

```
MyInterface obj = () -> {
    // lambda body
};
```

Stream API

Stream API introduced in Java 8 provides a mechanism for processing collections of objects in a functional manner.

- **Example Operations:**
  - **Filtering:** `stream.filter(condition)`
  - **Mapping:** `stream.map(transform)`
  - **Sorting:** `stream.sorted()`
  - **Collecting:** `stream.collect(Collectors.toList())`

```
List<String> names =
Arrays.asList("Alice", "Bob",
"Charlie");

// Print names starting with 'A'
names.stream()
     .filter(name ->
name.startsWith("A"))
     .forEach(System.out::println);
```

Java Development Tools

Java development involves various tools and frameworks to streamline coding, testing, building, and managing projects effectively. Here's an overview of some essential tools used in Java development:

JUnit Testing

JUnit is a popular Java framework for writing and running unit tests.

- **Writing Test Cases:**

```
import org.junit.Test;
import static
org.junit.Assert.assertEquals;

public class MyTest {

    @Test
    public void testAddition() {
        assertEquals(5, 2 + 3);
    }
}
```

- **Assertions:** Assertions in JUnit are used to verify the expected outcome of a test.
- **Test Suites:** Test suites in JUnit allow you to aggregate multiple test cases into a single unit.

Build Tools

Build tools automate the process of building (compiling, packaging, and deploying) Java applications.

- **Maven:**

Maven is a widely used build automation tool primarily for Java projects. It manages dependencies and builds processes using a declarative XML-based configuration.

  - **pom.xml Example:**

```
<project xmlns="http://maven.apache.org/
POM/4.0.0"
         xmlns:xsi="http://www.w3.org/
2001/XMLSchema-instance"
         xsi:schemaLocation="http://
maven.apache.org/POM/4.0.0 http://
maven.apache.org/xsd/maven-4.0.0.xsd">
    <modelVersion>4.0.0</modelVersion>
    <groupId>com.example</groupId>
    <artifactId>my-project</artifactId>
    <version>1.0-SNAPSHOT</version>
    <dependencies>
        <!-- Dependencies -->
    </dependencies>
</project>
```

- **Gradle:**

Gradle is another popular build automation tool that uses a Groovy-based domain-specific language (DSL) or Kotlin for build configuration.

  - **build.gradle Example:**

```
plugins {
    id 'java'
}

repositories {
    mavenCentral()
}

dependencies {
    implementation
'com.google.guava:guava:30.1-jre'
    testImplementation
'junit:junit:4.13.2'
}
```