Core Java:

- Declarations and Access Control
- Object Orientation
- Operators
- Flow Control
- Exceptions
- Gradle Fundamentals
- TDD with Junit 5
- Strings, /O
- Formatting, and Parsing
- Generics and Collections
- Threads.

Declarations and Access Control

Object Orientation

Classes and Objects

Class: A blueprint for creating objects. It defines a type of object by encapsulating data (attributes) and methods (functions) that operate on the data.

Object: An instance of a class. It represents a specific realization of the class with actual values.

Encapsulation

Encapsulation involves bundling the data (attributes) and methods (behaviors) that operate on the data into a single unit, the class. It also involves controlling access to the data using access modifiers (private, protected, public).

Purpose: Encapsulation helps protect the internal state of an object and only exposes necessary functionalities.

Inheritance

Inheritance is a mechanism by which a new class (subclass or derived class) inherits the attributes and methods of an existing class (superclass or base class).

Purpose: It promotes code reuse and establishes a hierarchical relationship between classes. A subclass can extend or override the behavior of its superclass.

Polymorphism

Polymorphism allows objects of different classes to be treated as objects of a common superclass. It enables one interface to be used for a general class of actions, with specific action implementations determined at runtime.

Types:

Compile-Time Polymorphism (Method Overloading): Multiple methods with the same name but different parameters.

Runtime Polymorphism (Method Overriding): A subclass provides a specific implementation of a method that is already defined in its superclass.

Abstraction

Abstraction involves hiding the complex implementation details and showing only the essential features of an object. It focuses on what an object does rather than how it does it.

Purpose: It helps simplify complex systems by modeling high-level functionalities without delving into the internal workings.

Formatting, and Parsing

Formatting

Formatting refers to the process of converting data into a specific string representation, often for display purposes. In Java, this typically involves converting numbers, dates, and other objects into a readable or standardized format. Formatting ensures that data is presented in a user-friendly way, matching locale-specific conventions or custom patterns.

Examples:

1. **Numbers:** You might format a number to include commas for thousands or to display a fixed number of decimal places.
2. **Dates:** Dates can be formatted to display in various styles, such as "MM/dd/yyyy" or "yyyy-MM-dd HH:mm".
3. **Custom Objects:** You can format custom objects to produce a readable string that represents the object's state.

Parsing

Parsing is the reverse process of formatting. It involves converting a string representation of data back into its original or usable form. Parsing is essential for reading user input, processing data files, or interpreting formatted strings.

Examples:

1. **Numbers:** Parsing converts a string like "1,234.56" back into a numeric type, such as double or int.
2. **Dates:** Parsing a string like "2024-07-30" converts it into a Date or LocalDateTime object.
3. **Custom Objects:** Parsing can convert formatted strings representing custom objects back into actual instances of those objects.

```
import java.time.LocalDateTime;
import java.time.format.DateTimeFormatter;
```

```
public class DateTimeFormatterExample {
    public static void main(String[] args) {
        LocalDateTime now = LocalDateTime.now();
        DateTimeFormatter formatter = DateTimeFormatter.ofPattern("dd-MM-yyyy HH:mm:ss");

        // Formatting
        String formattedDateTime = now.format(formatter);
        System.out.println("Formatted DateTime: " + formattedDateTime);

        // Parsing
        String dateTimeStr = "29-07-2024 10:15:30";
        LocalDateTime dateTime = LocalDateTime.parse(dateTimeStr, formatter);
        System.out.println("Parsed DateTime: " + dateTime);
    }
}
```

1.Explain the evolution of JAVA. List out java buzzwords:

Evolution of Java

Java, developed by James Gosling at Sun Microsystems and released in 1995, has undergone significant evolution:

1. **Java 1.0 (1996)**: Initial release targeting web applets.
2. **Java 1.1 (1997)**: Introduced inner classes, JavaBeans, and reflection.
3. **Java 2 (1998)**: Introduced Swing, collections framework.
4. **Java 1.3 (2000)**: Introduced Hotspot JVM, JNDI.
5. **Java 1.4 (2002)**: Added assert keyword, NIO, logging API.
6. **Java 5.0 (2004)**: Introduced generics, annotations, enhanced for loop.
7. **Java 6 (2006)**: Performance improvements, scripting language support.
8. **Java 7 (2011)**: Diamond operator, try-with-resources.
9. **Java 8 (2014)**: Lambda expressions, Stream API, new Date and Time API.
10. **Java 9 (2017)**: Module system, JShell.
11. **Java 10 (2018)**: Local-variable type inference.
12. **Java 11 (2018)**: LTS release, HTTP Client API.
13. **Java 12-20 (2019-2023)**: Continuous improvements, including new features like text blocks, pattern matching, records, and virtual threads.

Java Buzzwords

1. **Simple**
2. **Object-Oriented**
3. **Platform-Independent**

4. **Secured**
5. **Robust**
6. **Multithreaded**
7. **Architecture-Neutral**
8. **Interpreted**
9. **High Performance**
10. **Distributed**
11. **Dynamic**

Java Overview

Java is a class-based, object-oriented programming language designed to have minimal implementation dependencies. It enables developers to write code once and run it anywhere (WORA), as compiled Java code can run on all platforms supporting Java without recompilation. Java was developed by James Gosling and released by Sun Microsystems in 1995. It is widely used for desktop, web, and mobile applications due to its simplicity, robustness, and security features. Java was later acquired by Oracle Corporation.

Key Java Terminology

1. **Java Virtual Machine (JVM)**: Executes Java bytecode, making Java platform-independent.
2. **Bytecode**: Intermediate code generated by the Java compiler, executed by the JVM.
3. **Java Development Kit (JDK)**: Includes tools for developing, compiling, and debugging Java applications.
4. **Java Runtime Environment (JRE)**: Allows running Java applications; includes JVM and libraries.
5. **Garbage Collector**: Manages memory by automatically reclaiming unused objects.
6. **Class Path**: The file path where the Java runtime and compiler look for class files.

Primary Features of Java

1. **Platform Independent**: Java bytecode can run on any platform with a JVM.
2. **Object-Oriented**: Follows OOP principles like abstraction, encapsulation, inheritance, and polymorphism.
3. **Simple**: Lacks complex features like pointers and operator overloading.
4. **Robust**: Strong memory management, exception handling, and type-checking.
5. **Secure**: No pointers, runs in a controlled environment (sandbox).
6. **Distributed**: Supports creating distributed applications.
7. **Multithreaded**: Supports concurrent execution of threads.
8. **Portable**: Write once, run anywhere.
9. **High Performance**: Uses Just-In-Time (JIT) compiler.
10. **Dynamic**: Supports dynamic loading of classes and methods.

11. **Sandbox Execution**: Ensures safe execution of code in a restricted environment.
12. **Write Once Run Anywhere**: Java bytecode runs on any platform with a JVM.
13. **Compilation and Interpretation**: Combines the advantages of both compilation and interpretation.

Basic Java Syntax

- **class**: Declares a class.
- **public**: Access specifier.
- **static**: Allows method to be called without instantiating the class.
- **void**: Specifies method does not return a value.
- **main**: Entry point of the program.
- **String[] args**: Command-line arguments.
- **System.out.println**: Prints to the console.
-

Exceptions

Q: What is a Java Exception? List out the difference between checked and Unchecked exceptions with a program examples.

The **Exception Handling in Java** is one of the powerful *mechanisms to handle the runtime errors* so that the normal flow of the application can be maintained.

Exception is an abnormal condition. In Java, an exception is an event that disrupts the normal flow of the program. It is an object which is thrown at runtime.

Exception Handling is a mechanism to handle runtime errors such as **ClassNotFoundException**, **IOException, SQLException, RemoteException, etc.**

Advantage of Exception Handling:

The core advantage of exception handling is to maintain the normal flow of the application. An exception normally disrupts the normal flow of the application; that is why we need to handle exceptions. Let's consider a scenario.

Hierarchy of Java Exception classes:

Java Exception Handling Example:

```
import java.io.BufferedReader;
import java.io.FileReader;
import java.io.IOException;

public class ExceptionExample {
    public static void main(String[] args) {
        // Unchecked exception example (ArithmeticException)
        try {
            int result = divide(10, 0); // This will cause ArithmeticException
            System.out.println("Result: " + result);
        } catch (ArithmeticException e) {
            System.out.println("Caught an ArithmeticException: " + e.getMessage());
        }

        // Unchecked exception example (NullPointerException)
        try {
            String s = null;
            printLength(s); // This will cause NullPointerException
        } catch (NullPointerException e) {
            System.out.println("Caught a NullPointerException: " + e.getMessage());
        }

        // Checked exception example (IOException)
        try {
            readFile("nonexistentfile.txt"); // This will cause IOException
        } catch (IOException e) {
            System.out.println("Caught an IOException: " + e.getMessage());
        }

        // Unchecked exception example (ArrayIndexOutOfBoundsException)
        try {
            int[] array = new int[5];
            array[10] = 50; // This will cause ArrayIndexOutOfBoundsException
        } catch (ArrayIndexOutOfBoundsException e) {
            System.out.println("Caught an ArrayIndexOutOfBoundsException: " + e.getMessage());
        }

        // Unchecked exception example (NumberFormatException)
        try {
            String number = "abc";
            int parsedNumber = Integer.parseInt(number); // This will cause NumberFormatException
        } catch (NumberFormatException e) {
            System.out.println("Caught a NumberFormatException: " + e.getMessage());
        }
    }

    // Method that may throw an ArithmeticException (unchecked exception)
    public static int divide(int a, int b) {
```

```
        return a / b;
    }

    // Method that may throw a NullPointerException (unchecked exception)
    public static void printLength(String s) {
        System.out.println(s.length());
    }

    // Method that may throw an IOException (checked exception)
    public static void readFile(String fileName) throws IOException {
        BufferedReader reader = new BufferedReader(new FileReader(fileName));
        String line;
        while ((line = reader.readLine()) != null) {
            System.out.println(line);
        }
        reader.close();
    }
}
```

Common Scenarios of Java Exceptions

There are given some scenarios where unchecked exceptions may occur. They are as follows:

1) A scenario where ArithmeticException occurs

If we divide any number by zero, there occurs an ArithmeticException.

```
1. int a=50/0;//ArithmeticException
```

2) A scenario where NullPointerException occurs

If we have a null value in any variable, performing any operation on the variable throws a NullPointerException.

```
1. String s=null;
2. System.out.println(s.length());//NullPointerException
```

3) A scenario where NumberFormatException occurs

If the formatting of any variable or number is mismatched, it may result into NumberFormatException. Suppose we have a string variable that has characters; converting this variable into digit will cause NumberFormatException.

```
1. String s="abc";
2. int i=Integer.parseInt(s);//NumberFormatException
```

4) A scenario where ArrayIndexOutOfBoundsException occurs

When an array exceeds to it's size, the ArrayIndexOutOfBoundsException occurs. there may be other reasons to occur ArrayIndexOutOfBoundsException. Consider the following statements.

1. int a[]=new int[5];
2. a[10]=50; //ArrayIndexOutOfBoundsException

Types of Exceptions

1. **Checked Exceptions**: These are exceptions that are checked at compile-time. They must be either caught or declared in the method signature using the throws keyword.
2. **Unchecked Exceptions**: These are exceptions that are not checked at compile-time, but rather at runtime. They are subclasses of RuntimeException and do not need to be declared or caught.

Checked vs. Unchecked Exceptions

Checked Exceptions

- **Compile-time**: Checked exceptions are checked at compile-time.
- **Handling**: They must be handled using try-catch blocks or declared in the method signature using the throws keyword.
- **Examples**: IOException, SQLException, FileNotFoundException.

Unchecked Exceptions

- **Runtime**: Unchecked exceptions occur at runtime.
- **Handling**: They do not need to be explicitly handled or declared.
- **Examples**: NullPointerException, ArrayIndexOutOfBoundsException, ArithmeticException

Checked Exception Example:

```
import java.io.*;
public class CheckedExceptionExample {
    public static void main(String[] args) {
        try {
            readFile("nonexistentfile.txt");
        } catch (IOException e) {
            System.out.println("Caught IOException: " + e.getMessage());  }  }
    // Method that throws a checked exception
    public static void readFile(String fileName) throws IOException {
        FileReader file = new FileReader(fileName);
        BufferedReader fileInput = new BufferedReader(file);
        for (int counter = 0; counter < 3; counter++) {
            System.out.println(fileInput.readLine()); }
        fileInput.close();}}
```

Unchecked Exception Example:

```
public class UncheckedExceptionExample {
    public static void main(String[] args) {
        try {
            divide(10, 10);
        } catch (ArithmeticException e) {
            System.out.println("Caught ArithmeticException: " + e.getMessage());    }}
    // Method that causes an unchecked exception
    public static void divide(int a, int b) {
        int result = a / b; // This will throw ArithmeticException
        System.out.println("Result: " + result);}}
```

Highlighting the Differences between Checked and Unchecked Exceptions in Java

Serial No.	Checked Exception	Unchecked Exception
1.	Checked exceptions occur during compile time when the source code is being converted into an executable code.	Unchecked exceptions occur during runtime when the program is in execution.
2.	The compiler checks the checked exception.	The compiler does not check these types of exceptions.
3.	Checked exceptions can be manually created.	Unchecked exceptions can also be manually created.
4.	This type of exception is considered as a subclass of the exception class.	Unchecked exceptions occur during runtime, therefore, they are not included in the exception class.
5.	The Java Virtual Machine requires the checked exception to be caught or handled.	The Java Virtual Machine does not require the unchecked exception to be caught or handled.

Threads

Q:Design and implement JAVA cod snippet to create five threads with different priorities. Send two threads of the highest priority to sleep state. Check the aliveness of the threads and mark which is long lasting.

```
public class ThreadPriorityExample {
    public static void main(String[] args) {
        // Create five threads with different priorities
        Thread thread1 = new Thread(new Task(), "Thread-1");
        Thread thread2 = new Thread(new Task(), "Thread-2");
        Thread thread3 = new Thread(new Task(), "Thread-3");
        Thread thread4 = new Thread(new Task(), "Thread-4");
        Thread thread5 = new Thread(new Task(), "Thread-5");

        // Set different priorities
        thread1.setPriority(Thread.MIN_PRIORITY); // 1
        thread2.setPriority(Thread.NORM_PRIORITY); // 5
        thread3.setPriority(Thread.NORM_PRIORITY); // 5
        thread4.setPriority(Thread.MAX_PRIORITY); // 10
        thread5.setPriority(Thread.MAX_PRIORITY); // 10

        // Start the threads
        thread1.start();
        thread2.start();
        thread3.start();
        thread4.start();
        thread5.start();

        // Put the highest priority threads to sleep
        try {
            Thread.sleep(1000); // Ensure all threads have started
            System.out.println(thread4.getName() + " is going to sleep.");
            thread4.sleep(2000);
            System.out.println(thread5.getName() + " is going to sleep.");
            thread5.sleep(2000);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }

        // Check the aliveness of the threads
        System.out.println(thread1.getName() + " is alive: " + thread1.isAlive());
        System.out.println(thread2.getName() + " is alive: " + thread2.isAlive());
        System.out.println(thread3.getName() + " is alive: " + thread3.isAlive());
        System.out.println(thread4.getName() + " is alive: " + thread4.isAlive());
        System.out.println(thread5.getName() + " is alive: " + thread5.isAlive());

        // Join the threads to ensure main thread waits for them to finish
        try {
```

```
            thread1.join();
            thread2.join();
            thread3.join();
            thread4.join();
            thread5.join();
        } catch (InterruptedException e) {
            e.printStackTrace();
        }

        // Determine which thread lasted the longest
        System.out.println("All threads have finished execution.");
    }
}

class Task implements Runnable {
    @Override
    public void run() {
        for (int i = 0; i < 5; i++) {
            System.out.println(Thread.currentThread().getName() + " - Priority: " +
Thread.currentThread().getPriority() + " - Count: " + i);
            try {
                Thread.sleep(500); // Simulate work with sleep
            } catch (InterruptedException e) {
                e.printStackTrace();
            }} }}
```

Explanation:

1. **Thread Creation and Priority Setting:**
    - Five threads are created, each running a Task which prints its name, priority, and a count.
    - Priorities are set using setPriority with Thread.MIN_PRIORITY, Thread.NORM_PRIORITY, and Thread.MAX_PRIORITY.
2. **Starting Threads:**
    - Threads are started using the start method.
3. **Sleeping Threads:**
    - The main thread sleeps for a second to ensure all threads have started.
    - The two highest priority threads (thread4 and thread5) are then put to sleep for two seconds each.
4. **Checking Aliveness:**
    - The isAlive method is used to check if the threads are still running.
5. **Joining Threads:**
    - The join method is used to make sure the main thread waits for all threads to finish execution.
6. **Task Execution:**
    - The run method of the Task class simulates some work by printing a message and sleeping for 500 milliseconds in a loop.

Q: Design JAVA code snippet to demonstrate the creation of Threads in two different ways.

Method 1: Extending the Thread Class

```
// Thread1 class extends the Thread class

class Thread1 extends Thread {

   public void run() {

      for (int i = 0; i < 5; i++) {

         System.out.println("Thread: " + i);

         try {

            Thread.sleep(1000); // pause for 1 second

         } catch (InterruptedException e) {

            e.printStackTrace();} } } }

public class ExampleThread1 {

   public static void main(String[] args) {

      Thread1 thread1 = new Thread1();

      thread1.start(); // start the thread }}
```

Method 2: Implementing the Runnable Interface:

```
// MyRunnable class implements the Runnable interface
class MyRunnable implements Runnable {
    public void run() {
        for (int i = 0; i < 5; i++) {
            System.out.println("Thread2: " + i);
            try {
                Thread.sleep(1000); // pause for 1 second
            } catch (InterruptedException e) {
                e.printStackTrace();}}}}

public class ExamThread {
    public static void main(String[] args) {
        Thread thread2 = new Thread(new MyRunnable()); // create a thread with the
runnable
        thread2.start(); // start the thread
    }
}
```

Operators

. Arithmetic Operators

These operators are used to perform basic arithmetic operations.

Operator	Description	Example
+	Addition	a + b
-	Subtraction	a - b
*	Multiplication	a * b
/	Division	a / b
%	Modulus (remainder)	a % b

2. Unary Operators

These operators operate on a single operand.

Operator	Description	Example
+	Unary plus (promotes the value)	+a
-	Unary minus (negates the value)	-a
++	Increment (increases by 1)	a++ or ++a
--	Decrement (decreases by 1)	a-- or --a
!	Logical complement (inverts value)	!a

3. Assignment Operators

These operators are used to assign values to variables.

Operator	Description	Example
=	Simple assignment	a = b
+=	Add and assign	a += b
-=	Subtract and assign	a -= b
*=	Multiply and assign	a *= b
/=	Divide and assign	a /= b
%=	Modulus and assign	a %= b
<<=	Left shift and assign	a <<= b
>>=	Right shift and assign	a >>= b
>>>=	Unsigned right shift and assign	a >>>= b

Operator	Description	Example
&=	Bitwise AND and assign	a &= b
^=	Bitwise XOR and assign	a ^= b
`	=`	Bitwise OR and assign

4. Relational Operators

These operators are used to compare two values.

Operator	Description	Example
==	Equal to	a == b
!=	Not equal to	a != b
>	Greater than	a > b
<	Less than	a < b
>=	Greater than or equal	a >= b
<=	Less than or equal	a <= b

5. Logical Operators

These operators are used to combine conditional statements.

Operator	Description	Example
&&	Logical AND	a && b
!	Logical NOT	!a

6. Bitwise Operators

These operators are used to perform bit-level operations.

Operator	Description	Example
&	Bitwise AND	a & b
`	`	Bitwise OR
^	Bitwise XOR	a ^ b
~	Bitwise complement	~a
<<	Left shift	a << 2
>>	Right shift	a >> 2
>>>	Unsigned right shift	a >>> 2

7. Ternary Operator

This is a shorthand for the if-else statement.

Operator	Description	Example
?:	Ternary (conditional) operator	a = (b > c) ? b : c;

8. Instanceof Operator

This operator is used to test whether an object is an instance of a specific class or subclass.

Operator	Description	Example
instanceof	Type comparison operator	obj instanceof String

Code:

```
public class OperatorExamples22 {
    public static void main(String[] args) {
        // ------------------------------
        // Arithmetic Operators
        // ------------------------------
        int a = 10, b = 20;
        System.out.println("Arithmetic Operators:");
        System.out.println("Addition (a + b): " + (a + b)); // Addition
        System.out.println("Subtraction (a - b): " + (a - b)); // Subtraction
        System.out.println("Multiplication (a * b): " + (a * b)); // Multiplication
        System.out.println("Division (b / a): " + (b / a)); // Division
        System.out.println("Modulus (b % a): " + (b % a)); // Modulus

        // ------------------------------
        // Unary Operators
        // ------------------------------
        int x = 10;
        System.out.println("\nUnary Operators:");
        System.out.println("Initial value of x: " + x); // 10
        System.out.println("Unary plus (++x): " + ++x); // Unary plus and increment
        System.out.println("Post-increment (x++): " + x++); // Post-increment
        System.out.println("Value of x after x++: " + x); // Value after post-increment
        System.out.println("Unary minus (--x): " + --x); // Unary minus and decrement
        System.out.println("Post-decrement (x--): " + x--); // Post-decrement
        System.out.println("Value of x after x--: " + x); // Value after post-decrement

        // ------------------------------
        // Assignment Operators
        // ------------------------------
        int y = 10;
        System.out.println("\nAssignment Operators:");
        y += 5; // y = y + 5
        System.out.println("Add and assign (y += 5): " + y); // 15
        y -= 3; // y = y - 3
        System.out.println("Subtract and assign (y -= 3): " + y); // 12
        y *= 2; // y = y * 2
        System.out.println("Multiply and assign (y *= 2): " + y); // 24
        y /= 4; // y = y / 4
        System.out.println("Divide and assign (y /= 4): " + y); // 6
        y %= 3; // y = y % 3
        System.out.println("Modulus and assign (y %= 3): " + y); // 0

        // ------------------------------
        // Relational Operators
```

```
        // ------------------------------
        System.out.println("\nRelational Operators:");
        System.out.println("Equal to (a == b): " + (a == b)); // false
        System.out.println("Not equal to (a != b): " + (a != b)); // true
        System.out.println("Greater than (a > b): " + (a > b)); // false
        System.out.println("Less than (a < b): " + (a < b)); // true
        System.out.println("Greater than or equal to (a >= b): " + (a >= b)); // false
        System.out.println("Less than or equal to (a <= b): " + (a <= b)); // true

        // ------------------------------
        // Logical Operators
        // ------------------------------
        boolean c = true, d = false;
        System.out.println("\nLogical Operators:");
        System.out.println("Logical AND (c && d): " + (c && d)); // false
        System.out.println("Logical OR (c || d): " + (c || d)); // true
        System.out.println("Logical NOT (!c): " + !c); // false

        // ------------------------------
        // Bitwise Operators
        // ------------------------------
        int e = 5, f = 3;
        System.out.println("\nBitwise Operators:");
        System.out.println("Bitwise AND (e & f): " + (e & f)); // 1 (0101 & 0011 = 0001)
        System.out.println("Bitwise OR (e | f): " + (e | f)); // 7 (0101 | 0011 = 0111)
        System.out.println("Bitwise XOR (e ^ f): " + (e ^ f)); // 6 (0101 ^ 0011 = 0110)
        System.out.println("Bitwise complement (~e): " + ~e); // -6 (bitwise complement of 5)
        System.out.println("Left shift (e << 1): " + (e << 1)); // 10 (0101 << 1 = 1010)
        System.out.println("Right shift (e >> 1): " + (e >> 1)); // 2 (0101 >> 1 = 0010)

        // ------------------------------
        // Ternary Operator
        // ------------------------------
        int g = 10, h = 20;
        System.out.println("\nTernary Operator:");
        int max = (g > h) ? g : h;
        System.out.println("Ternary (max = (g > h) ? g : h): " + max); // 20

        // ------------------------------
        // instanceof Operator
        // ------------------------------
        String str = "Hello";
        System.out.println("\ninstanceof Operator:");
        boolean result = str instanceof String;
        System.out.println("instanceof (str instanceof String): " + result); // true
    }

}
```

Q: Explain the following with program examples:

i)< = ii) > = iii) >> iv) << v) >>>

```
public class OperatorExamples {
    public static void main(String[] args) {
        int a = 5;
        int b = 10;
        int num = 8;
        int negativeNum = -8;
        // Less than or equal to (<=)
        System.out.println("Example of <= operator:");
        if (a <= b) {
            System.out.println(a + " is less than or equal to " + b);
        } else {
            System.out.println(a + " is greater than " + b);
        }
        // Greater than or equal to (>=)
        System.out.println("\nExample of >= operator:");
        if (b >= a) {
            System.out.println(b + " is greater than or equal to " + a);
        } else {
            System.out.println(b + " is less than " + a);
        }
        // Right shift (>>)
        System.out.println("\nExample of >> operator:");
        int rightShiftResult = num >> 2; // binary: 1000 >> 2 = 0010
        System.out.println(num + " >> 2 = " + rightShiftResult);
        // Left shift (<<)
        System.out.println("\nExample of << operator:");
        int leftShiftResult = num << 2; // binary: 1000 << 2 = 100000
        System.out.println(num + " << 2 = " + leftShiftResult);
        // Unsigned right shift (>>>)
        System.out.println("\nExample of >>> operator:");
        int unsignedRightShiftResult = negativeNum >>> 2;
        System.out.println(negativeNum + " >>> 2 = " + unsignedRightShiftResult);}}
```

Explanation:

1. **Less than or equal to (<=):**
    - Checks if a (5) is less than or equal to b (10).
    - Output: 5 is less than or equal to 10.
2. **Greater than or equal to (>=):**
    - Checks if b (10) is greater than or equal to a (5).
    - Output: 10 is greater than or equal to 5.
3. **Right shift (>>):**
    - Shifts the bits of num (8, binary 1000) to the right by 2 positions.
    - 8 >> 2 results in 2 (binary 0010).
    - Output: 8 >> 2 = 2.
4. **Left shift (<<):**

- Shifts the bits of num (8, binary 1000) to the left by 2 positions.
- 8 << 2 results in 32 (binary 100000).
- Output: 8 << 2 = 32.

5. **Unsigned right shift (>>>):**
    - Shifts the bits of negativeNum (-8, binary 11111111111111111111111111111000) to the right by 2 positions, filling with 0s.
    - -8 >>> 2 results in 1073741822 (binary 00111111111111111111111111111110).
    - Output: -8 >>> 2 = 1073741822.

Generics and Collections

List out the best practices for Java Collections Framework?

The Java Collections Framework (JCF) is a unified architecture for representing and manipulating collections, enabling collections to be easily manipulated, traversed, and managed. It consists of interfaces, implementations (classes), and algorithms to manage collections of objects.

Key Interfaces in the Java Collections Framework

1. **Collection Interface**: The root of the collection hierarchy.
    - **List**: An ordered collection (sequence) that allows duplicate elements.
    - **Set**: A collection that does not allow duplicate elements.
    - **Queue**: A collection used to hold multiple elements prior to processing.
    - **Deque**: A double-ended queue that allows elements to be added or removed from both ends.
2. **Map Interface**: An object that maps keys to values, with no duplicate keys.
    - **SortedMap**: A Map that maintains its entries in ascending order of keys.
    - **NavigableMap**: A SortedMap extended with navigation methods.

Common Implementations

1. **List Implementations**
    - **ArrayList**: Resizable array implementation of the List interface.
    - **LinkedList**: Doubly-linked list implementation of the List and Deque interfaces.
    - **Vector**: Synchronized resizable array implementation (legacy).
    - **Stack**: A subclass of Vector that implements a last-in-first-out (LIFO) stack (legacy).
2. **Set Implementations**
    - **HashSet**: Hash table implementation of the Set interface.
    - **LinkedHashSet**: Hash table and linked list implementation of the Set interface, with predictable iteration order.
    - **TreeSet**: Red-black tree implementation of the NavigableSet interface.
3. **Queue Implementations**

- **PriorityQueue**: A priority heap-based implementation of the Queue interface.
- **LinkedList**: Can also be used as a Queue implementation.

4. **Map Implementations**

    - **HashMap**: Hash table implementation of the Map interface.
    - **LinkedHashMap**: Hash table and linked list implementation of the Map interface, with predictable iteration order.
    - **TreeMap**: Red-black tree implementation of the NavigableMap interface.
    - **Hashtable**: Synchronized hash table implementation (legacy).

5. **Deque Implementations**

    - **ArrayDeque**: Resizable array implementation of the Deque interface.
    - **LinkedList**: Can also be used as a Deque implementation.

Key Methods in Collection Interface

- **add(E e)**: Adds the specified element to the collection.
- **remove(Object o)**: Removes the specified element from the collection, if present.
- **size()**: Returns the number of elements in the collection.
- **isEmpty()**: Checks if the collection is empty.
- **iterator()**: Returns an iterator over the elements in the collection.
- **contains(Object o)**: Checks if the collection contains the specified element.
- **clear()**: Removes all elements from the collection.

Choose the Right Collection Type:

Use the most appropriate collection type for your needs (e.g., ArrayList for fast random access, LinkedList for frequent inserts/deletes, HashSet for unique elements).

Prefer Interfaces Over Implementations:

Declare variables and parameters using interfaces (e.g., List, Set, Map) rather than specific implementations (e.g., ArrayList, HashSet, HashMap).

```
List<String> list = new ArrayList<>();
Set<String> set = new HashSet<>();
Map<String, String> map = new HashMap<>();
```

Initial Capacity:

Set an initial capacity if you know the expected size of the collection to avoid unnecessary resizing.

```
ArrayList<String> list = new ArrayList<>(100);
HashMap<String, String> map = new HashMap<>(200);
```

Use Generics:

Always use generics to ensure type safety and avoid ClassCastException.

```
List<String> list = new ArrayList<>();
```

Avoid Raw Types:

Avoid using raw types as they lead to unsafe operations and warnings.

```
Bad
List list = new ArrayList();
 Good
List<String> list = new ArrayList<>();
```

Use Collections Utility Class:

Utilize methods from the Collections utility class for common operations like sorting, searching, and reversing.

```
Collections.sort(list);
Collections.reverse(list);
```

Immutability:

Use immutable collections where possible to ensure thread safety and to avoid accidental modification.

```
List<String> immutableList = Collections.unmodifiableList(list);
```

Fail-Fast Iterators:

Be aware that most collections provide fail-fast iterators which throw ConcurrentModificationException if the collection is modified while iterating.

```
List<String> list = new ArrayList<>();
 Iterator<String> iterator = list.iterator();
 while (iterator.hasNext()) {
       // Do not modify the list here
       String item = iterator.next();
    }
```

Avoid Unnecessary Synchronization:

Prefer using concurrent collections from java.util.concurrent over manually synchronizing collections.

```
ConcurrentHashMap<String, String> map = new ConcurrentHashMap<>();
```

Use forEach and Streams:

Leverage the enhanced for loop, forEach method, and Stream API for cleaner and more readable code.

```
list.forEach(System.out::println);
list.stream().filter(s -> s.startsWith("A")).forEach(System.out::println);
```

Specific Collection Types Best Practices:

1. **Lists:**
    - Prefer ArrayList for most cases unless you need constant time for inserts and removes, in which case use LinkedList.
    - Use subList method for partitioning lists but be cautious as changes to sublist reflect in the original list.
2. **Sets:**
    - Use HashSet for most cases due to its performance. Use TreeSet when you need sorted sets or LinkedHashSet when you need a predictable iteration order.
    - Avoid using mutable objects as elements in sets.
3. **Maps:**
    - Prefer HashMap for most cases due to its performance. Use TreeMap for sorted maps or LinkedHashMap for predictable iteration order.
    - Use getOrDefault to handle default values easily.
        - map.getOrDefault(key, defaultValue);
4. **Queues:**
    - Use PriorityQueue for priority-based ordering. Use ArrayDeque as a general-purpose queue/deque implementation.
    - Avoid LinkedList for queue operations due to potential performance issues and use ArrayDeque instead.

Using List Interface:

```
import java.util.*;

public class ListExample {

    public static void main(String[] args) {

        List<String> arrayList = new ArrayList<>();

        arrayList.add("Apple");

        arrayList.add("Banana");

        arrayList.add("Orange");

        System.out.println("ArrayList: " + arrayList);

        List<String> linkedList = new LinkedList<>();

        linkedList.add("Dog");

        linkedList.add("Cat");

        linkedList.add("Cow");

        System.out.println("LinkedList: " + linkedList);

        // Iterating over a list

        for (String fruit : arrayList) {

            System.out.println(fruit);}}}
```

Using Map Interface:

```
package collections;

import java.util.HashMap;
import java.util.LinkedHashMap;
import java.util.Map;
import java.util.TreeMap;

public class MapExample {
    public static void main(String[] args) {
        // HashMap
        Map<String, String> hashMap = new HashMap<>();
        hashMap.put("1", "Apple");
        hashMap.put("2", "Banana");

        // TreeMap
        Map<String, String> treeMap = new TreeMap<>();
        treeMap.put("1", "Apple");
        treeMap.put("2", "Banana");
```

```
        // LinkedHashMap
        Map<String, String> linkedHashMap = new LinkedHashMap<>();
        linkedHashMap.put("1", "Apple");
        linkedHashMap.put("2", "Banana");

        System.out.println("HashMap: " + hashMap); // No guaranteed order
        System.out.println("TreeMap: " + treeMap); // Sorted order by key
        System.out.println("LinkedHashMap: " + linkedHashMap); // Insertion order

        // Using getOrDefault
        System.out.println("Value for key 3: " + hashMap.getOrDefault("3", "Not Found")); // Not Found
    }
}
```

```
package collections;

import java.util.ArrayDeque;

import java.util.PriorityQueue;

import java.util.Queue;

public class QueueExample {

    public static void main(String[] args) {

        // PriorityQueue

        Queue<String> priorityQueue = new PriorityQueue<>();

        priorityQueue.add("Apple");

        priorityQueue.add("Banana");

        priorityQueue.add("Cherry");

        // ArrayDeque

        Queue<String> arrayDeque = new ArrayDeque<>();

        arrayDeque.add("Apple");

        arrayDeque.add("Banana");

        arrayDeque.add("Cherry");

        // LinkedList as Queue

        Queue<String> linkedListQueue = new ArrayDeque<>(); // Avoid LinkedList, use ArrayDeque

        linkedListQueue.add("Apple");
```

```
        linkedListQueue.add("Banana");
        linkedListQueue.add("Cherry");

        System.out.println("PriorityQueue: " + priorityQueue); // Priority order
        System.out.println("ArrayDeque: " + arrayDeque); // FIFO order
        System.out.println("LinkedList Queue: " + linkedListQueue); // FIFO order
    }
}
```

```
package collections;

import java.util.HashSet;
import java.util.LinkedHashSet;
import java.util.TreeSet;

public class SetExample {
    public static void main(String[] args) {
        // HashSet
        HashSet<String> hashSet = new HashSet<>();
        hashSet.add("Apple");
        hashSet.add("Banana");
        hashSet.add("Cherry");

        // LinkedHashSet
        LinkedHashSet<String> linkedHashSet = new LinkedHashSet<>();
        linkedHashSet.add("Apple");
        linkedHashSet.add("Banana");
```

```
        linkedHashSet.add("Cherry");


        // TreeSet
        TreeSet<String> treeSet = new TreeSet<>();
        treeSet.add("Banana");
        treeSet.add("Apple");
        treeSet.add("Cherry");


        System.out.println("HashSet: " + hashSet); // Random order
        System.out.println("LinkedHashSet: " + linkedHashSet); // Insertion order
        System.out.println("TreeSet: " + treeSet); // Sorted order
    }
}
```

Strings, I/O

What is String Parsing in Java? Explain parsing of Integers to String with a program example.

String parsing in Java refers to the process of converting a String into another data type, such as an integer, double, boolean, etc. This is often necessary when dealing with data input from users, files, or other external sources that are initially read as strings but need to be converted to another type for further processing.

Parsing Integers from Strings

Parsing an integer from a string in Java can be done using the Integer.parseInt() method or the Integer.valueOf() method. Both methods take a String as an argument and convert it into an int.

- Integer.parseInt(String s): Converts the string s into a primitive int.
- Integer.valueOf(String s): Converts the string s into an Integer object.

```
public class StringParsingExample {
    public static void main(String[] args) {
        // Parsing String to int using parseInt
        String numberStr1 = "123";
        int number1 = Integer.parseInt(numberStr1);
        System.out.println("Parsed int using parseInt: " + number1);

        // Parsing String to Integer using valueOf
        String numberStr2 = "456";
        Integer number2 = Integer.valueOf(numberStr2);
        System.out.println("Parsed Integer using valueOf: " + number2);

        // Converting int to String using toString
        int number3 = 789;
        String numberStr3 = Integer.toString(number3);
        System.out.println("Converted int to String using toString: " + numberStr3);

        // Converting Integer to String using toString
        Integer number4 = 101112;
        String numberStr4 = number4.toString();
        System.out.println("Converted Integer to String using toString: " + numberStr4);

        // Another way to convert int to String using String.valueOf
        String numberStr5 = String.valueOf(number3);
        System.out.println("Converted int to String using String.valueOf: " + numberStr5);
    }}
```

Explanation:

1. **Parsing String to int using parseInt:**
   - Integer.parseInt(numberStr1): Converts the string "123" to the primitive integer 123.
2. **Parsing String to Integer using valueOf:**
   - Integer.valueOf(numberStr2): Converts the string "456" to the Integer object 456.
3. **Converting int to String using toString:**
   - Integer.toString(number3): Converts the primitive integer 789 to the string "789".
4. **Converting Integer to String using toString:**
   - number4.toString(): Converts the Integer object 101112 to the string "101112".
5. **Another way to convert int to String using String.valueOf:**
   - String.valueOf(number3): Converts the primitive integer 789 to the string "789".

```
public class StringExample {

    public static void main(String[] args) {

        String str = "Hello, World!";

        System.out.println("Length: " + str.length());

        System.out.println("Substring: " + str.substring(7, 12));

        System.out.println("Replace 'World' with 'Java': " + str.replace("World", "Java"));

        System.out.println("To Upper Case: " + str.toUpperCase());

        System.out.println("Trimmed: " + "   Hello   ".trim());

    }

}
```

I/O (Input/Output) in Java

Java provides a rich set of classes for I/O operations, both for reading from and writing to files, as well as handling other I/O operations.

1. Reading from Files

Using FileReader and BufferedReader:

```
import java.io.BufferedReader;
import java.io.FileReader;
import java.io.IOException;
public class FileReadExample {
    public static void main(String[] args) {
        try (BufferedReader br = new BufferedReader(new FileReader("example.txt"))) {
            String line;
            while ((line = br.readLine()) != null) {
                System.out.println(line);
            }
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

2. Writing to Files

Using FileWriter and BufferedWriter:

java

Copy code

```
import java.io.BufferedWriter;
import java.io.FileWriter;
import java.io.IOException;
```

```
public class FileWriteExample {
    public static void main(String[] args) {
        try (BufferedWriter bw = new BufferedWriter(new FileWriter("example.txt"))) {
            bw.write("Hello, World!");
            bw.newLine();
            bw.write("Welcome to Java I/O.");
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

3. Reading from Console

Using Scanner:

java

Copy code

```
import java.util.Scanner;

public class ConsoleReadExample {
    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);

        System.out.println("Enter your name: ");
        String name = scanner.nextLine();

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

```
        scanner.close();
    }
}
```

4. Writing to Console

Using System.out:

java

Copy code

```
public class ConsoleWriteExample {
    public static void main(String[] args) {
        System.out.println("Hello, World!");
        System.out.print("Welcome to Java I/O.");
    }
}
```

Explain how type safety can be achieved in Collections with an example.

Type safety in collections can be achieved through the use of generics in Java. Generics allow you to specify the type of elements that a collection can hold, which helps catch type-related errors at compile-time rather than at runtime. This ensures that only objects of the specified type can be added to the collection, preventing ClassCastException and improving code readability and maintainability.

Example without Generics (Not Type-Safe)

Without generics, collections are not type-safe, meaning they can hold any type of objects. This can lead to runtime errors when retrieving and casting elements.

```
import java.util.*;

public class NonGenericExample {
    public static void main(String[] args) {
        List list = new ArrayList(); // Raw type, not type-safe
        list.add("Hello");
        list.add(123); // No compile-time error, but not safe

        for (Object obj : list) {
            String str = (String) obj; // Causes ClassCastException at runtime
            System.out.println(str);
        }
    }
}
```

Example with Generics (Type-Safe)

With generics, you can specify the type of elements that a collection can hold, making the code type-safe and preventing runtime errors.

```
import java.util.ArrayList;
import java.util.List;

public class GenericExample {
    public static void main(String[] args) {
        List<String> list = new ArrayList<>(); // Type-safe, holds only Strings
        list.add("Hello");
        // list.add(123); // Compile-time error, prevents adding non-String

        for (String str : list) {
            System.out.println(str); // No need for casting, safe
        }
    }
}
```

Explanation:

1. **Without Generics (Not Type-Safe):**
   - The `list` is declared without a type, so it can hold any type of objects.
   - Adding different types of objects (`String` and `Integer`) is allowed.
   - When retrieving elements, a `ClassCastException` can occur if the types do not match.
2. **With Generics (Type-Safe):**
   - The `list` is declared with a type parameter `<String>`, making it type-safe.
   - Only `String` objects can be added to the list.
   - Attempting to add an `Integer` to the list will result in a compile-time error, preventing runtime issues.
   - No casting is needed when retrieving elements, reducing the risk of `ClassCastException`.

Benefits of Using Generics:

- **Compile-Time Type Checking:** Errors are caught during compilation rather than at runtime, making the code more robust.
- **Elimination of Casting:** No need to cast elements when retrieving them from the collection, improving code readability and safety.
- **Code Reusability:** Generic methods and classes can be written to work with any object type, making the code more reusable and flexible.

Design Java code snippet to demonstrate the following

i.foreach loop ii.Switch iii.dowhile

```
package collections.flowcontrol;

public class FlowControlExample {



   public static void main(String[] args) {

      // 1. Conditional Statements

      int number = 10;

      // if statement

      if (number > 0) {

         System.out.println("The number is positive.");

      }

      // if-else statement

      if (number > 0) {

         System.out.println("The number is positive.");
```

```
} else {
    System.out.println("The number is not positive.");
}
// if-else if-else statement
if (number > 0) {
    System.out.println("The number is positive.");
} else if (number < 0) {
    System.out.println("The number is negative.");
} else {
    System.out.println("The number is zero.");
}
// switch statement
int day = 3;
switch (day) {
    case 1:
        System.out.println("Monday");
        break;
    case 2:
        System.out.println("Tuesday");
        break;
    case 3:
        System.out.println("Wednesday");
        break;
    default:
        System.out.println("Invalid day");
}
// 2. Looping Statements
```

```
// for loop
System.out.println("For loop:");
for (int i = 0; i < 5; i++) {
    System.out.println(i);
}
// Enhanced for loop (for-each loop)
int[] numbers = {1, 2, 3, 4, 5};
System.out.println("Enhanced for loop:");
for (int num : numbers) {
    System.out.println(num);
}
// while loop
System.out.println("While loop:");
int j = 0;
while (j < 5) {
    System.out.println(j);
    j++;
}
// do-while loop
System.out.println("Do-while loop:");
int k = 0;
do {
    System.out.println(k);
    k++;
} while (k < 5);

// 3. Branching Statements
```

```
        // break statement
        System.out.println("Break statement:");
        for (int i = 0; i < 10; i++) {
            if (i == 5) {
                break;
            }
            System.out.println(i);
        }

        // continue statement
        System.out.println("Continue statement:");
        for (int i = 0; i < 10; i++) {
            if (i % 2 == 0) {
                continue;
            }
            System.out.println(i);
        }
        // return statement
        int result = square(5);
        System.out.println("Square of 5 is: " + result);
    }
    // Method to demonstrate return statement
    public static int square(int number) {
        return number * number;
    }
}
```

1. Conditional Statements

Purpose: Determine which block of code should be executed based on conditions.

- **if Statement:** Executes a block of code if a specified condition is true.
- **if-else Statement:** Executes one block of code if the condition is true, and another block if the condition is false.
- **if-else if-else Statement:** Checks multiple conditions in sequence and executes the corresponding block of code for the first condition that is true.
- **switch Statement:** Selects one of many code blocks to execute based on the value of an expression. It's useful for handling multiple possible values of a single variable.

2. Looping Statements

Purpose: Repeat a block of code multiple times.

- **for Loop:** Used when the number of iterations is known beforehand. It includes initialization, condition, and iteration expressions.
- **Enhanced for Loop (for-each Loop):** Iterates over elements in an array or collection, simplifying the syntax for iterating through each element.
- **while Loop:** Repeats a block of code as long as a specified condition is true. The condition is evaluated before each iteration.
- **do-while Loop:** Executes a block of code once, then repeats as long as a specified condition is true. The condition is evaluated after each iteration.

3. Branching Statements

Purpose: Alter the flow of execution by jumping to different parts of the code.

- **break Statement:** Exits the current loop or switch statement and transfers control to the statement following the loop or switch.
- **continue Statement:** Skips the remaining code in the current iteration of the loop and proceeds with the next iteration.
- **return Statement:** Exits from the current method and optionally returns a value

Write about

i)Gradle Fundamentals ii) TDD with Junit 5.

i)Gradle Fundamentals:

Gradle is an open-source build automation tool that is designed to be flexible enough to build almost any type of software. It is widely used in Java projects but is also suitable for building projects in other languages. Gradle combines the best features of Ant and Maven and introduces a Groovy-based DSL (Domain-Specific Language) for writing build scripts.

Gradle Tutorial

Gradle tutorial provides basic and advanced concepts of the Gradle tool. Our Gradle tutorial is developed for beginners and professionals.

Our Gradle tutorial includes **project task**, **installation and configuration**, **Gradle build**, **Gradle Build Scans**, **Gradle dependencies**, **Gradle Projects**, **Gradle eclipse plug-in**, **Gradle with Java**, **Gradle with spring**, **Gradle with Android** and more concepts related to Gradle.

Gradle is an advanced general-purpose build management tool that is based on **Groovy and Kotlin**. It is a **build automation** tool that is an open-source and based on the concepts of Apache Maven and Apache Ant. It is developed for the multi-projects, which can be quite large. It has been developed for building automation on many languages and platforms, including **Java**, **Scala**, **Android**, **C / C ++**, and **Groovy.**

What is Gradle?

Gradle is an open source **build automation** tool that is based on the concept of **Apache Maven** and **Apache Ant**. It is capable of building almost any type of software. It is designed for the multi-project build, which can be quite large. It introduces a **Java and Groovy-based DSL(Domain Specific Language)** instead of XML (Extensible Markup Language) for declaring the project configuration. It uses a DAG (Directed Acyclic Graph) to define the order of executing the task.

Gradle offers an elastic model that can help the development lifecycle from compiling and packaging code for web and mobile applications. It provides support for the **building**, **testing**, and **deploying software** on different platforms. It has been developed for building automation on many languages and platforms, including Java, Scala, Android, C / C ++, and Groovy. Gradle provides integration with several development tools and servers, including Eclipse, IntelliJ, Jenkins, and Android Studio.

Gradle is used by large projects such as **Spring Projects**, **Hibernate Projects**, and **Grails Projects**. Some Leading Enterprise companies like **LinkedIn** and **Netflix** use Gradle.

Gradle was initially released in 2007, and it is stably released on November 18, 2019 (latest version 6.0.1). Gradle has taken the advantages of both Ant and Maven and remove the drawbacks of both.

What is a Build Tool?

Build tools are **programs that are used to automate the creation of executable** applications from source code. The building process involves compiling, linking, and **packaging the code into a useful or executable form**. Developers often implement the build process manually for small projects. But this cannot be done for large projects where it is complicated to keep track of what is needed for construction, in what order, and what dependencies are in the building process. Using the automation tools makes the build process more consistent.

Projects and Tasks in Gradle

Gradle describes everything on the basis of **projects** and **tasks**. Every **Gradle build** contains one or more projects, and these projects contain some tasks.

Gradle Projects

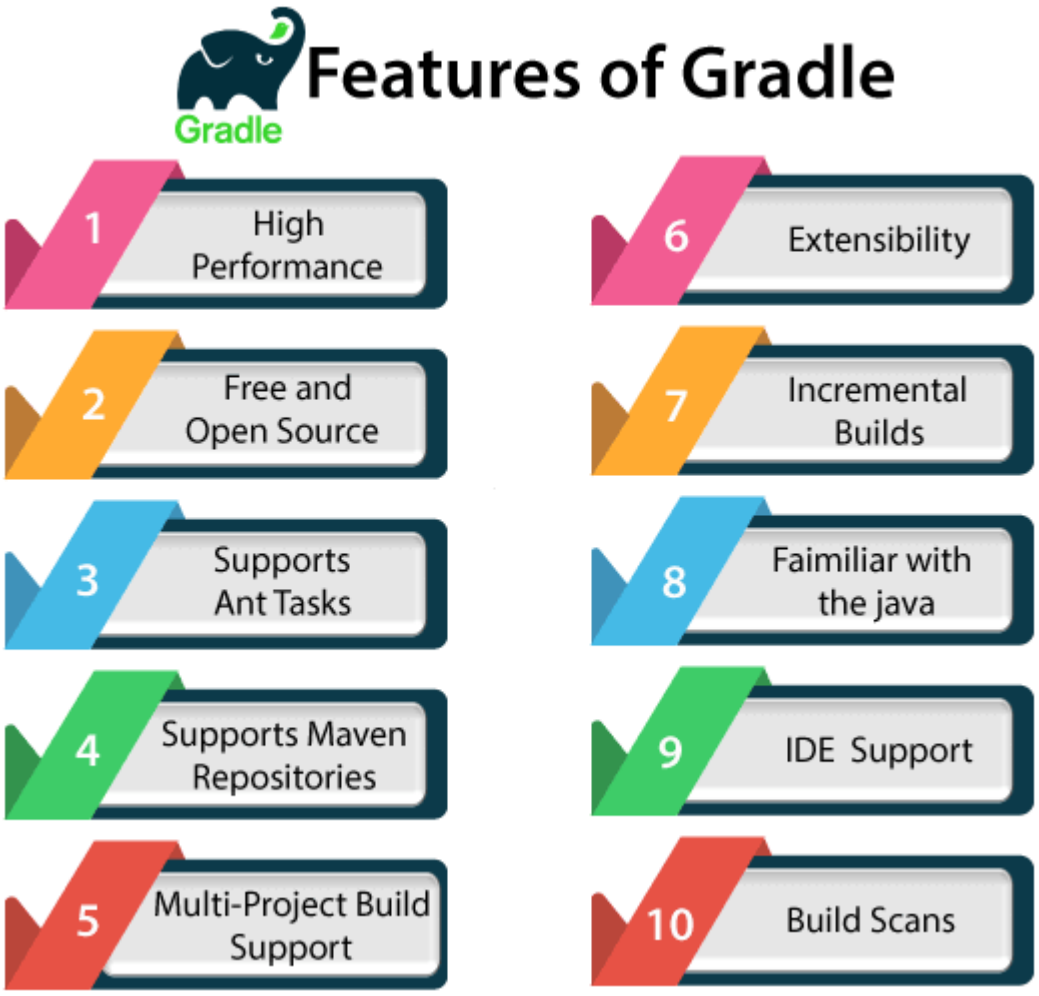
In Gradle, A project represents a library JAR or a web application. It may also represent a distribution ZIP, which is assembled from the JARs produced by other projects. A project could be deploying your application to staging or production environments. Each project in Gradle is made up of one or more tasks.

Gradle Tasks

In Gradle, Task is a single piece of work that a build performs. For example, it could **compile classes**, **create a JAR**, **Generate Javadoc**, and **publish some archives** to a repository and more.

Features of Gradle

Some remarkable features of Gradle are as follows:



High Performance

Gradle quickly completes the task by reusing the output from the previous execution. It processes tasks whose only input is changed and executes the task in parallel. Thus it avoids unnecessary tasks and provides faster performance.

Free and open-source

Gradle is an **open-source** tool and is licensed under the **Apache License (ASL)**.

Provide support for Ant Tasks and Maven repositories

Gradle provides support for the Ant build projects; we can import an Ant build project and reuse all of its tasks. However, we can also make Ant-based Gradle tasks. The integration includes the properties, paths, and more.

Gradle supports the Maven repository. Maven Repositories are designed to publish and fetch dependencies of the project. Therefore we can continue to use any available repository infrastructure.

Multi-project build support

Gradle provides **powerful support for the multi-project builds**. A multi-project build may contain a root project and one or more subprojects that may also have subprojects. We can flexibly define our layout with the Gradle.

A project can simply be dependent on other projects or dependencies. We can describe a graph of dependencies among projects. Gradle also supports partial builds. It means that Gradle will find out whether a project, upon which our project depends, needs to be rebuilt. If the project needs to be rebuilt, Gradle will do so before building our own project.

Extensibility

Extensibility is one of the decent features of Gradle. We can easily extend the Gradle to provide our task types or build models. For an example of this, see Android Build Support: It adds several new build concepts such as flavor and builds types.

Incremental Builds

Gradle facilitates us with an incremental build, which means it **executes only the necessary tasks**. If we compile source code, it will check if the sources have changed since the previous execution. If the code is changed, then it will be executed; but, if the code is not changed, then it will skip the execution, and the task is marked as updated. There are many algorithms in Gradle to do so.

Familiar with the Java

We need a JVM to run the Gradle, so our machine should have a Java Development Kit (JDK). Gradle is familiar with most of the Java features. It is a bonus for the java users as we can use the standard Java APIs in our build logic, such as **plug-ins** and **custom tasks**. Therefore it makes it easy to run Gradle on different platforms.

Gradle isn't limited to building just JVM projects; it also provides support for building native projects.

IDE Support

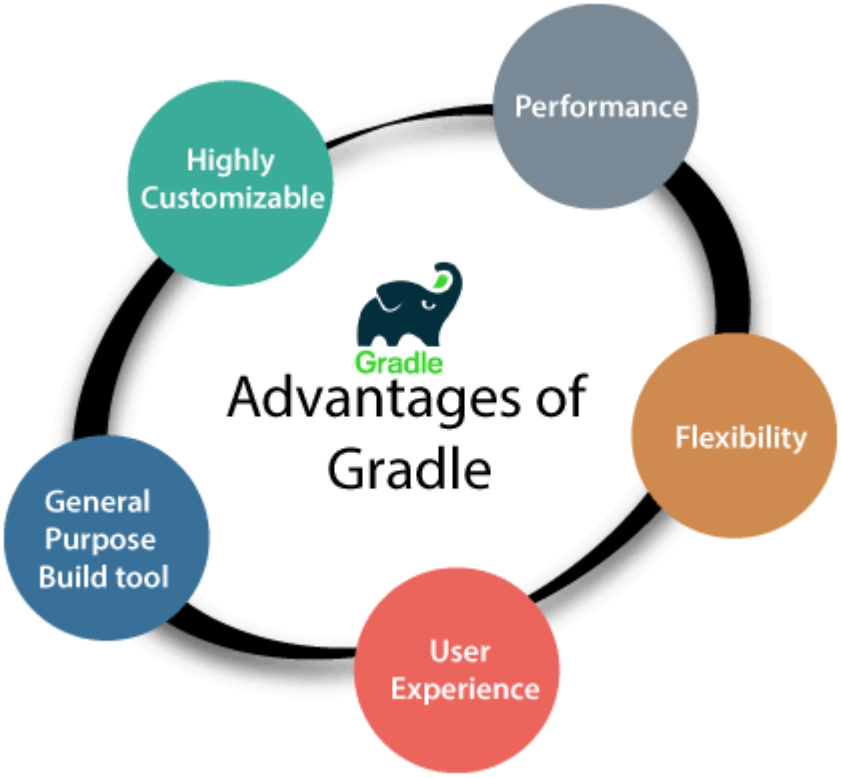
Gradle has **support for several IDE's**. They are allowed to import the Gradle builds and interact with them. Gradle also generates the required solution files to load a project into Visual Studio.

Build Scans

The **Build Scans** provides comprehensive information about build run that can be used to identify build issues. They also help us to diagnose the problems with a build's performance. The build scans can be shared with others; this can be useful if we need the advice to fix an issue with the build.

Advantages of Gradle

Some significant benefits of Gradle are as following:



Highly Customizable

Gradle is highly customizable and extensible. It can be customized to different projects under different technologies. It can be customized in many ways like it can be used in Java projects, Android projects, Groovy projects, and more.

Performance

Gradle is very fast in performance. It is about two times faster than Maven in all scenarios and a hundred times faster in large builds using build-cache.

Flexibility

Gradle is a flexible tool. Gradle is a plug-in based build tool. We can create our plug-in in different programming language like Java, Groovy, Kotlin, Scala, and more. If we want to add any functionality after deployment to the project, to do this, we can create a plug-in and give control to the codebase.

User Experience

Gradle supports a wide range of IDE's to provide an improved user experience. Most people prefer to work on the IDE, but many users prefer to work on the terminal, Gradle provides a command-line interface for them. Gradle command-line interface offers many powerful features like Gradle Tasks, Command line completion, and more.

It also provides interactive web-based UI for debugging and optimizing builds.

Gradle is a general-purpose build tool

Gradle is a general-purpose build tool; it allows us to build any type of software.

TDD with JUnit 5

Test-Driven Development (TDD) is a software development process that relies on the repetition of a very short development cycle: first, the developer writes a failing automated test case that defines a desired improvement or new function, then produces the minimum amount of code to pass that test, and finally refactors the new code to acceptable standards.

JUnit 5 is the latest version of JUnit, which is a popular testing framework for Java. It introduces several new features and improvements over its predecessors, making it more flexible and extensible.

TDD Cycle with JUnit 5

The TDD cycle consists of three main steps:

1. **Write a Failing Test**:
    - Start by writing a test for a new feature or a bug fix that you want to implement. This test should initially fail since the feature is not yet implemented.
2. **Write the Minimum Code to Pass the Test**:
    - Implement the simplest possible code that will make the test pass. The focus here is on getting the test to pass, not on writing perfect code.
3. **Refactor the Code**:
    - Clean up the code, ensuring it is well-structured and adheres to coding standards. Run the tests again to ensure that they still pass after refactoring.

Setting Up JUnit 5 with Gradle

To get started with JUnit 5, you need to include the necessary dependencies in your `build.gradle` file:

```
groovy
Copy code
plugins {
    id 'java'
}

repositories {
    mavenCentral()
}

dependencies {
    testImplementation 'org.junit.jupiter:junit-jupiter:5.7.0'
}

test {
    useJUnitPlatform()
}
```

Example of TDD with JUnit 5

Let's walk through an example of using TDD to implement a simple calculator class with an `add` method.

Step 1: Write a Failing Test

Create a test class and write a test for the `add` method that does not yet exist.

```
import org.junit.jupiter.api.Test;
import static org.junit.jupiter.api.Assertions.*;

public class CalculatorTest {
    @Test
    public void testAddition() {
        Calculator calculator = new Calculator();
        assertEquals(5, calculator.add(2, 3), "2 + 3 should equal 5");
    }
}
```

Step 2: Write the Minimum Code to Pass the Test

Create the `Calculator` class and implement the `add` method.

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

Run the test using `./gradlew test`. The test should pass now.

Step 3: Refactor the Code

Since the implementation is straightforward, there might not be much to refactor. However, as you add more features, refactoring will become more important to maintain clean and manageable code.

Additional Features with TDD

Let's add a new feature to our calculator: a `subtract` method.

Step 1: Write a Failing Test

Add a new test case for the `subtract` method.

```
public class CalculatorTest {
    // Existing test
    @Test
    public void testAddition() {
        Calculator calculator = new Calculator();
        assertEquals(5, calculator.add(2, 3), "2 + 3 should equal 5");
    }

    // New test
    @Test
    public void testSubtraction() {
        Calculator calculator = new Calculator();
        assertEquals(1, calculator.subtract(3, 2), "3 - 2 should equal 1");
    }
}
```

Step 2: Write the Minimum Code to Pass the Test

Implement the `subtract` method in the `Calculator` class.

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

```
    public int subtract(int a, int b) {
        return a - b;
    }
}
```

Run the tests again using `./gradlew test`. Both tests should pass now.

Step 3: Refactor the Code

Ensure that the code is clean and adheres to best practices. In this simple example, the code is already straightforward, but as more features are added, refactoring will be necessary.

Advantages of TDD

1. **Better Code Quality**:
   - TDD encourages developers to write only the necessary code to pass tests, which often results in simpler and cleaner code.
2. **Reduced Bugs**:
   - Since tests are written before the code, many bugs are caught early in the development process.
3. **Documentation**:
   - Tests serve as a form of documentation, explaining what the code is supposed to do.
4. **Refactoring Confidence**:
   - With a comprehensive suite of tests, developers can refactor code with confidence, knowing that any breaking changes will be caught by the tests.

Object Orientation

1. Classes and Objects

- **Class:** A blueprint that defines the structure and behavior (attributes and methods) of objects.
- **Object:** An instance of a class, representing a specific entity with actual values for its attributes.

2. Encapsulation

- **Definition:** Bundles data (attributes) and methods (functions) into a single unit (class) and controls access to the data.
- **Access Modifiers:** Use private, protected, and public to control visibility and access.

3. Inheritance

- **Definition:** A mechanism that allows one class (subclass) to inherit attributes and methods from another class (superclass), promoting code reuse and establishing a hierarchy.

4. Polymorphism

- **Definition:** The ability to use a single interface to represent different underlying forms. It allows methods to perform different tasks based on the object they are acting upon.
- **Types:** Includes method overloading (same method name with different parameters) and method overriding (redefining a method in a subclass).

5. Abstraction

- **Definition:** Hides complex implementation details and shows only the necessary features of an object. It simplifies interactions with objects and reduces complexity.
- **Abstract Classes and Interfaces:** Used to define abstract methods that must be implemented by subclasses or implementing classes.

6. Composition

- **Definition:** A design principle where one class contains another class, representing a "has-a" relationship. It is used to build complex types from simpler ones.

```
// Class definition
public class Animal {
    // Encapsulation
    private String name;

    // Constructor
    public Animal(String name) {
        this.name = name;
    }

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

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

    // Method
    public void makeSound() {
        System.out.println("Some sound");
    }
}

// Inheritance
public class Dog extends Animal {
    // Constructor
    public Dog(String name) {
        super(name);
    }

    // Method Overriding
    @Override
    public void makeSound() {
        System.out.println("Bark");
    }
}

// Interface
public interface Pet {
    void play();
}
```

```
// Polymorphism
public class Main {
    public static void main(String[] args) {
        Animal myDog = new Dog("Buddy");
        myDog.makeSound(); // Output: Bark

        // Using interface
        Pet pet = new Dog("Charlie");
        pet.play(); // You can define the play method in Dog class
    }
}
```