Java Final Notes

Table of Content

- Custom Exception Classes

5. Java Input/Output (I/O)
    - Streams and Readers/Writers
    - File I/O
    - Serialization
6. Generics
    - Introduction to Generics
    - Generic Classes
    - Generic Methods
    - Wildcards
7. Collections Framework
    - Lists, Sets, and Maps
    - ArrayList, LinkedList, HashSet, TreeSet, HashMap, TreeMap, etc.
    - Iterators
    - Sorting and Searching
8. Multithreading
    - Introduction to Threads
    - Creating and Managing Threads
    - Synchronization
    - Thread Safety
    - Thread Communication
    - Thread Pools
9. Java I/O and Networking
    - File Handling
    - Streams (Byte Streams and Character Streams)
    - Network Programming (TCP/IP, UDP, Sockets)
10. JDBC (Java Database Connectivity)

- Introduction to Databases
- Connecting to Databases
- Executing SQL Queries
- Transaction Management

11. Java GUI (Graphical User Interface) Programming
    - Introduction to Swing
    - Components (Buttons, Labels, Text Fields, etc.)
    - Event Handling
    - Layout Managers
12. Java Reflection
    - Introduction to Reflection
    - Obtaining Class Information
    - Dynamic Class Loading
    - Accessing and Modifying Objects at Runtime
13. Java 8 Features
    - Lambda Expressions
    - Functional Interfaces
    - Stream API
    - Default and Static Methods in Interfaces
    - Date and Time API
14. Design Patterns
    - Creational Patterns
    - Structural Patterns
    - Behavioral Patterns
    - Singleton, Factory, Observer, Strategy, etc.
15. Java Best Practices and Coding Standards
    - Naming Conventions

- Code Formatting
- Exception Handling Best Practices
- Memory Management
- Performance Optimization

16. Java Testing and Debugging
    - Unit Testing (JUnit)
    - Debugging Techniques and Tools
17. Introduction to JavaFX (optional)
    - JavaFX Basics
    - GUI Components
    - Event Handling in JavaFX

Introduction to Java:

History of Java:

- Java was developed by James Gosling and his team at Sun Microsystems (now owned by Oracle Corporation) in the mid-1990s.
- It was originally designed for programming consumer electronic devices, but its focus shifted towards internet programming.
- Java's development was influenced by the need for a platform-independent language that could run on various devices and operating systems.
- The first version of Java, Java 1.0, was released in 1996.

Java Features and Benefits:

- Simple and easy to learn: Java has a straightforward syntax and a rich set of libraries, making it accessible for beginners.
- Object-oriented: Java follows an object-oriented programming (OOP) paradigm, allowing for modular and reusable code.
- Platform independence: Java programs can run on any platform with a Java Virtual Machine (JVM), making them highly portable.
- Robust and secure: Java includes features like garbage collection and built-in exception handling, ensuring reliable and secure code.

- Rich API: Java provides a vast standard library (API) for various tasks, from input/output operations to networking and database access.
- Multithreading: Java supports concurrent programming with built-in thread management, allowing for efficient utilization of system resources.
- High performance: Java's Just-In-Time (JIT) compilation and optimized runtime make it a high-performance language.

Java Virtual Machine (JVM):

- The Java Virtual Machine (JVM) is an integral part of the Java platform and acts as an execution environment for Java programs.
- It provides a layer of abstraction between the Java code and the underlying operating system.
- JVM interprets compiled Java bytecode and translates it into machine code specific to the host system.
- It handles memory management, garbage collection, and runtime optimizations for efficient execution of Java programs.
- JVM implementations are available for various platforms, allowing Java programs to run consistently across different systems.

Java Development Kit (JDK):

- The Java Development Kit (JDK) is a software development environment that provides tools, libraries, and documentation for Java development.
- It includes the Java compiler (javac) to compile Java source code into bytecode.
- JDK also contains the Java Runtime Environment (JRE), which includes the JVM and necessary libraries to run Java applications.
- Developers use the JDK to write, compile, and debug Java programs.
- It is available in different versions, with each version introducing new features and improvements.

Getting Started with Java:

- To start programming in Java, you need to install the Java Development Kit (JDK) on your system.
- Once installed, you can use a text editor or Integrated Development Environment (IDE) to write Java code.

- Java programs are structured into classes, where each class represents a blueprint for objects.
- A simple "Hello, World!" program in Java looks like this:

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

- In this example, we define a class named "HelloWorld" with a "main" method.
- The "main" method is the entry point of the program, and it prints the string "Hello, World!" to the console using the `System.out.println` statement.
- To run the Java program, you need to compile it using the `javac` command, and then execute it using the `java` command.

Java Fundamentals:

Variables and Data Types:

- Variables are used to store data in memory during program execution. In Java, you need to declare a variable before using it.
- Java supports various data types, including primitive types (int, double, boolean) and reference types (String, arrays, objects).
- Primitive data types in Java are categorized into four groups: integer types, floating-point types, character type, and boolean type.
- Examples of variable declaration and initialization:

```
int age; // variable declaration
age = 25; // variable initialization

double salary = 50000.0; // variable declaration and initialization

boolean isStudent = true; // variable declaration and initialization

String name = "John"; // variable declaration and initialization
```

Operators and Expressions:

- Operators are used to perform operations on variables and values. Java supports a wide range of operators, including arithmetic, assignment, comparison, logical, and more.
- Arithmetic operators: +, -, *, /, % (modulus)
- Assignment operators: =, +=, -=, *=, /=
- Comparison operators: ==, !=, >, <, >=, <=
- Logical operators: && (AND), || (OR), ! (NOT)
- Example expressions:

```
int x = 10;
int y = 5;
int sum = x + y; // arithmetic operation

boolean isTrue = (x > y) && (x != 0); // logical operation
```

Control Flow Statements (if-else, switch, loops):

- Control flow statements are used to control the execution flow of a program based on certain conditions or repetitive tasks.
- If-else statements allow you to execute different blocks of code based on a condition.
- Switch statements are used for multi-way branching based on different cases.
- Loops (for, while, do-while) enable you to repeatedly execute a block of code.

```
int age = 18;

if (age >= 18) {
    System.out.println("You are eligible to vote."); // executed if the condition is t
rue
} else {
    System.out.println("You are not eligible to vote."); // executed if the condition
 is false
}

int day = 2;
String dayName;

switch (day) {
    case 1:
        dayName = "Monday";
        break;
    case 2:
```

```
        dayName = "Tuesday";
        break;
    default:
        dayName = "Unknown";
}

for (int i = 0; i < 5; i++) {
    System.out.println(i); // executes the loop body 5 times
}

int[] numbers = {1, 2, 3, 4, 5};
for (int num : numbers) {
    System.out.println(num); // iterates over the array elements
}
```

Arrays:

- Arrays are used to store multiple values of the same data type in a single variable.
- Arrays have a fixed size, determined at the time of declaration.
- Example array declaration and initialization:

```
int[] numbers = new int[5]; // declaration with size
numbers[0] = 1; // assigning values to array elements
numbers[1] = 2;
numbers[2] = 3;
numbers[3] = 4;
numbers[4] = 5;

String[] names = {"John", "Alice", "Bob"}; // declaration with initialization

int length = numbers.length; // accessing the length of the array
```

Strings:

- Strings in Java are objects that represent a sequence of characters.
- Strings are immutable, meaning they cannot be modified once created.
- String manipulation and operations are performed using various methods provided by the

String class.

- Example string operations:

```
String name = "John";
int length = name.length(); // gets the length of the string
```

```
char firstChar = name.charAt(0); // gets the first character of the string

String upperCaseName = name.toUpperCase(); // converts the string to uppercase

boolean containsA = name.contains("a"); // checks if the string contains a specific su
bstring

String concatenated = name + " Doe"; // concatenates two strings

String formatted = String.format("Hello, %s!", name); // formats a string using placeh
olders
```

Object-Oriented Programming (OOP) Concepts:

Classes and Objects:

- Classes are the blueprint or template for creating objects in Java.
- Objects are instances of a class that represent real-world entities or concepts.
- Classes define the properties (attributes) and behaviors (methods) that objects of that class can have.
- Example of class definition and object instantiation:

```
// Class definition
public class Car {
    // Instance variables
    private String make;
    private String model;
    private int year;

    // Constructor
    public Car(String make, String model, int year) {
        this.make = make;
        this.model = model;
        this.year = year;
    }

    // Instance method
    public void startEngine() {
        System.out.println("Engine started.");
    }
}

// Object instantiation
Car myCar = new Car("Toyota", "Camry", 2022);
```

Encapsulation:

- Encapsulation is the process of bundling data (instance variables) and methods that operate on that data within a class.
- It provides data hiding and protects the internal state of objects from direct manipulation.
- Access to the data is typically controlled through getter and setter methods.
- Example of encapsulation:

```
public class BankAccount {
    private double balance;

    public double getBalance() {
        return balance;
    }

    public void deposit(double amount) {
        // Perform necessary checks and update balance
    }

    public void withdraw(double amount) {
        // Perform necessary checks and update balance
    }
}

BankAccount account = new BankAccount();
double balance = account.getBalance(); // Accessing the balance through a getter metho
d
account.deposit(1000); // Updating the balance through a method
```

Inheritance:

- Inheritance is a mechanism that allows a class to inherit properties and behaviors from another class.
- The class that inherits is called the subclass or derived class, and the class being inherited from is called the superclass or base class.
- Subclasses can extend or override the inherited members and also add new members.
- Example of inheritance:

```
public class Shape {
    protected int x;
    protected int y;
```

```
    public Shape(int x, int y) {
        this.x = x;
        this.y = y;
    }

    public void draw() {
        System.out.println("Drawing shape at (" + x + ", " + y + ")");
    }
}

public class Circle extends Shape {
    private int radius;

    public Circle(int x, int y, int radius) {
        super(x, y);
        this.radius = radius;
    }

    @Override
    public void draw() {
        System.out.println("Drawing circle at (" + x + ", " + y + ") with radius " + r
adius);
    }
}

Circle circle = new Circle(5, 5, 10);
circle.draw(); // Call to overridden method in the Circle class
```

Polymorphism:

- Polymorphism allows objects of different classes to be treated as objects of a common superclass.
- It enables the use of a single interface to represent different implementations.
- Polymorphism is achieved through method overriding and method overloading.
- Example of polymorphism:

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

public class Cat extends Animal {
    @Override
    public void makeSound() {
        System.out.println("Meow!");
    }
}

public class Dog extends Animal {
    @Override
```

```
    public void makeSound() {
        System.out.println("Woof!");
    }
}

Animal cat = new Cat();
Animal dog = new Dog();

cat.makeSound(); // Polymorphic call to makeSound method of Cat

 class
dog.makeSound(); // Polymorphic call to makeSound method of Dog class
```

Abstraction:

- Abstraction is the process of simplifying complex systems by breaking them down into manageable and understandable components.
- It focuses on the essential properties and behaviors of an object while hiding the unnecessary details.
- Abstract classes and interfaces are used to achieve abstraction in Java.
- Example of abstraction using an abstract class:

```
public abstract class Shape {
    protected int x;
    protected int y;

    public Shape(int x, int y) {
        this.x = x;
        this.y = y;
    }

    public abstract void draw();
}

public class Circle extends Shape {
    private int radius;

    public Circle(int x, int y, int radius) {
        super(x, y);
        this.radius = radius;
    }

    @Override
    public void draw() {
        System.out.println("Drawing circle at (" + x + ", " + y + ") with radius " + r
adius);
    }
}
```

```
Shape shape = new Circle(5, 5, 10);
shape.draw(); // Polymorphic call to the draw method of Circle class
```

Interfaces:

- Interfaces define a contract or a set of methods that a class must implement.
- They provide a way to achieve multiple inheritance and enable loose coupling between classes.
- Classes implement interfaces using the `implements` keyword.
- Example of interfaces:

```
public interface Drawable {
    void draw();
}

public class Circle implements Drawable {
    private int radius;

    public Circle(int radius) {
        this.radius = radius;
    }

    @Override
    public void draw() {
        System.out.println("Drawing circle with radius " + radius);
    }
}

Drawable drawable = new Circle(10);
drawable.draw(); // Polymorphic call to the draw method of Circle class
```

Packages:

- Packages are used to organize classes into a hierarchical structure and avoid naming conflicts.
- They provide a way to group related classes and provide access control.
- Packages are declared at the beginning of Java source files using the `package` keyword.
- Example of packages:

```
package com.company.project;

public class MyClass {
```

```
    // Class code goes here
}
```

Exception Handling:

Introduction to Exceptions:

- Exceptions are events that occur during the execution of a program that disrupt the normal flow of code.
- They represent errors, exceptional conditions, or unexpected situations that need to be handled.
- Exceptions can occur due to various reasons, such as invalid input, resource unavailability, or programming errors.

Handling Exceptions (try-catch, finally):

- Exception handling allows us to gracefully handle exceptions and provide alternative flows in case of errors.
- The try-catch block is used to catch and handle exceptions.
- The code that may throw an exception is placed inside the try block.
- If an exception occurs within the try block, it is caught and handled in the catch block.
- The finally block is optional and is executed regardless of whether an exception occurs or not.
- Example of exception handling:

```
try {
    // Code that may throw an exception
    int result = divide(10, 0);
    System.out.println("Result: " + result);
} catch (ArithmeticException ex) {
    // Exception handling for ArithmeticException
    System.out.println("Error: " + ex.getMessage());
} finally {
    // Code that will always execute
    System.out.println("Finally block executed.");
}

public int divide(int dividend, int divisor) {
    return dividend / divisor;
}
```

Checked and Unchecked Exceptions:

- Checked exceptions are exceptions that need to be declared in the method signature or handled explicitly using try-catch blocks.
- Examples of checked exceptions include IOException, SQLException, and ClassNotFoundException.
- Unchecked exceptions, also known as runtime exceptions, do not require explicit handling or declaration.
- Examples of unchecked exceptions include NullPointerException, ArrayIndexOutOfBoundsException, and IllegalArgumentException.

Custom Exception Classes:

- In addition to built-in exceptions, you can create your own exception classes by extending the Exception class or its subclasses.
- Custom exceptions allow you to represent specific application-specific errors or exceptional conditions.
- Example of a custom exception class:

```
public class CustomException extends Exception {
    public CustomException(String message) {
        super(message);
    }
}

public class MyClass {
    public static void main(String[] args) {
        try {
            validateInput(10);
        } catch (CustomException ex) {
            System.out.println("Error: " + ex.getMessage());
        }
    }

    public static void validateInput(int value) throws CustomException {
        if (value < 0) {
            throw new CustomException("Invalid input: Value cannot be negative.");
        }
    }
}
```

In the example above, the `CustomException` class extends the `Exception` class to create a custom exception. The `validateInput` method throws the `CustomException` if the input value is negative, and it is caught and handled in the `main` method.

Java Input/Output (I/O):

Streams and Readers/Writers:

- In Java, I/O operations are performed using streams, which provide a convenient way to read from or write to a data source.
- Streams can be classified into two types: byte streams and character streams.
- Byte streams (`InputStream` and `OutputStream`) are used for binary data, while character streams (`Reader` and `Writer`) are used for text data.
- Readers and writers are designed to handle character-based data and provide methods for reading and writing characters or strings.
- Example of reading from a file using `BufferedReader`:

```
import java.io.*;

public class ReadFileExample {
    public static void main(String[] args) {
        try (BufferedReader reader = new BufferedReader(new FileReader("input.txt")))
 {
            String line;
            while ((line = reader.readLine()) != null) {
                System.out.println(line);
            }
        } catch (IOException ex) {
            System.out.println("Error reading file: " + ex.getMessage());
        }
    }
}
```

File I/O:

- File I/O operations involve reading from or writing to files on the file system.
- The `File` class is used to represent files and directories in Java.
- File I/O operations are typically performed using byte streams or character streams along with file-related classes.
- Example of writing to a file using `BufferedWriter`:

```
import java.io.*;

public class WriteFileExample {
    public static void main(String[] args) {
        try (BufferedWriter writer = new BufferedWriter(new FileWriter("output.txt")))
{
```

```
            writer.write("Hello, World!");
            writer.newLine();
            writer.write("This is a sample file.");
        } catch (IOException ex) {
            System.out.println("Error writing to file: " + ex.getMessage());
        }
    }
}
```

Serialization:

- Serialization is the process of converting an object into a byte stream, which can be saved to a file or sent over the network.
- In Java, serialization is achieved by implementing the `Serializable` interface.
- The `ObjectOutputStream` and `ObjectInputStream` classes are used to write and read serialized objects, respectively.
- Example of serializing an object to a file:

```
import java.io.*;

public class SerializeExample {
    public static void main(String[] args) {
        try (ObjectOutputStream outputStream = new ObjectOutputStream(new FileOutputSt
ream("data.ser"))) {
            Student student = new Student("John Doe", 25, "Computer Science");
            outputStream.writeObject(student);
            System.out.println("Object serialized.");
        } catch (IOException ex) {
            System.out.println("Error serializing object: " + ex.getMessage());
        }
    }
}

class Student implements Serializable {
    private String name;
    private int age;
    private String major;

    public Student(String name, int age, String major) {
        this.name = name;
        this.age = age;
        this.major = major;
    }

    // Getters and setters
}
```

In the example above, the `Student` class implements the `Serializable` interface, allowing its objects to be serialized. The `ObjectOutputStream` is used to write the serialized object to a file.

Generics:

Introduction to Generics:

- Generics in Java provide a way to create reusable code that can work with different types.
- They allow the definition of classes, interfaces, and methods that can operate on parameters of specified types.
- Generics enhance type safety and eliminate the need for explicit type casting.
- Generics are widely used in collections and algorithms to provide type-safe data structures and operations.

Generic Classes:

- Generic classes are classes that can work with different types.
- They are declared using type parameters, which are specified within angle brackets (<>) after the class name.
- Type parameters can be used as placeholders for actual types that will be provided when creating objects of the generic class.
- Example of a generic class:

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

    public Box(T value) {
        this.value = value;
    }

    public T getValue() {
        return value;
    }

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

Box<Integer> integerBox = new Box<>(10); // Create a Box object with Integer type
int value = integerBox.getValue(); // Get the value as an Integer
```

Generic Methods:

- Generic methods are methods that can work with different types.
- They are declared using type parameters, which are specified within angle brackets (<>) before the return type.
- Type parameters can be used as placeholders for actual types that will be determined at the time of method invocation.
- Example of a generic method:

```
public class ArrayUtils {
    public static <T> T getFirstElement(T[] array) {
        if (array != null && array.length > 0) {
            return array[0];
        }
        return null;
    }
}

String[] names = {"John", "Alice", "Bob"};
String firstElement = ArrayUtils.getFirstElement(names); // Invoke generic method with
String type
```

Wildcards:

- Wildcards are used in generics to provide flexibility in accepting different types.
- There are two types of wildcards: the upper bounded wildcard (`<?>`) and the lower bounded wildcard (`<? extends T>` or `<? super T>`).
- The upper bounded wildcard restricts the type to be a specific type or any of its subtypes.
- The lower bounded wildcard restricts the type to be a specific type or any of its supertypes.
- Example of using wildcards:

```
public class NumberUtils {
    public static double sum(List<? extends Number> numbers) {
        double total = 0.0;
        for (Number number : numbers) {
            total += number.doubleValue();
        }
        return total;
    }
}
```

```
List<Integer> integers = List.of(1, 2, 3);
double sum = NumberUtils.sum(integers); // Invoke method with a List of Integer
```

In the example above, the `sum` method accepts a list of any type that extends `Number`, allowing it to handle `Integer`, `Double`, and other number types.

Collections Framework:

Lists, Sets, and Maps:

- The Collections Framework in Java provides a set of interfaces and classes to work with collections of objects.
- Lists are ordered collections that allow duplicate elements. They maintain the order of insertion.
- Sets are collections that do not allow duplicate elements. They typically do not maintain any specific order.
- Maps are key-value pairs, where each element is associated with a unique key.
- Example of using lists, sets, and maps:

```
import java.util.*;

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

Set<Integer> numbers = new HashSet<>();
numbers.add(1);
numbers.add(2);
numbers.add(1);

Map<String, Integer> ages = new HashMap<>();
ages.put("Alice", 25);
ages.put("Bob", 30);
```

ArrayList, LinkedList, HashSet, TreeSet, HashMap, TreeMap, etc.:

- The Collections Framework provides several concrete implementations of lists, sets, and maps.
- ArrayList is an implementation of the List interface that uses a dynamic array to store elements.

- LinkedList is another implementation of the List interface that uses a doubly-linked list to store elements.
- HashSet is an implementation of the Set interface that uses a hash table to store elements. It does not maintain any specific order.
- TreeSet is another implementation of the Set interface that stores elements in a sorted order.
- HashMap is an implementation of the Map interface that uses a hash table to store key-value pairs. It does not maintain any specific order.
- TreeMap is another implementation of the Map interface that stores key-value pairs in a sorted order based on the keys.
- Example of using different collections:

```
import java.util.*;

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

Set<Integer> set = new HashSet<>();
set.add(1);
set.add(2);

Map<String, Integer> map = new HashMap<>();
map.put("Alice", 25);
map.put("Bob", 30);
```

Iterators:

- Iterators are used to traverse or iterate over the elements of a collection sequentially.
- The `Iterator` interface provides methods like `hasNext()` to check if there are more elements, and `next()` to retrieve the next element.
- Example of using an iterator:

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

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

Sorting and Searching:

- The Collections Framework provides utility methods for sorting and searching elements in lists.
- The `Collections` class contains static methods like `sort()` to sort lists, and `binarySearch()` to perform binary search on sorted lists.
- Example of sorting and searching:

```
List<Integer> numbers = new ArrayList<>(List.of(5, 3, 1, 4, 2));

Collections.sort(numbers); // Sort the list

int index = Collections.binarySearch(numbers, 3); // Perform binary search
if (index >= 0) {
    System.out.println("Element found at index " + index);
} else {
    System.out.println("Element not found");
}
```

In the example above, the `sort()` method is used to sort the list in ascending order, and the `binarySearch()` method is used to search for the element `3` in the sorted list.

Multithreading:

Introduction to Threads:

- Threads are lightweight units of execution within a program that can run concurrently.
- Multithreading allows multiple threads to execute in parallel, providing better utilization of system resources.
- Threads can be used to perform tasks concurrently, handle input/output operations, or improve responsiveness in user interfaces.

Creating and Managing Threads:

- In Java, threads can be created by extending the `Thread` class or implementing the `Runnable` interface.
- Extending the `Thread` class requires overriding the `run()` method, which contains the code to be executed by the thread.
- Implementing the `Runnable` interface requires implementing the `run()` method as well, and the `Runnable` object can be passed to a `Thread` constructor.

- Example of creating and starting a thread:

```
public class MyThread extends Thread {
    @Override
    public void run() {
        // Code to be executed by the thread
        System.out.println("Thread is running.");
    }
}

public class Main {
    public static void main(String[] args) {
        Thread thread = new MyThread();
        thread.start(); // Start the thread
    }
}
```

Synchronization:

- Synchronization is used to control access to shared resources or critical sections of code in a multithreaded environment.
- The `synchronized` keyword can be used to mark methods or blocks of code to ensure that only one thread can execute them at a time.
- Synchronization prevents data races and ensures data consistency.
- Example of using synchronization:

```
public class Counter {
    private int count;

    public synchronized void increment() {
        count++;
    }
}
```

Thread Safety:

- Thread safety refers to the property of code or data structures that can be safely accessed and manipulated by multiple threads without causing data corruption or inconsistencies.
- Thread-safe code ensures that shared data is accessed in a synchronized manner or by using thread-safe data structures.
- Common techniques for achieving thread safety include synchronization, the use of atomic operations, and immutability.

- Example of thread-safe code:

```
public class SafeCounter {
    private AtomicInteger count = new AtomicInteger(0);

    public void increment() {
        count.incrementAndGet();
    }
}
```

Thread Communication:

- Thread communication allows threads to interact with each other by sharing information or coordinating their actions.
- Common techniques for thread communication include using shared variables, signaling mechanisms like `wait()` and `notify()`, and higher-level constructs like `Locks` and `Conditions`.
- Example of thread communication using `wait()` and `notify()`:

```
public class Message {
    private String content;
    private boolean available = false;

    public synchronized String receive() {
        while (!available) {
            try {
                wait(); // Wait until a message is available
            } catch (InterruptedException ex) {
                Thread.currentThread().interrupt();
            }
        }
        available = false;
        notifyAll(); // Notify waiting threads
        return content;
    }

    public synchronized void send(String message) {
        while (available) {
            try {
                wait(); // Wait until the previous message is consumed
            } catch (InterruptedException ex) {
                Thread.currentThread().interrupt();
            }
        }
        content = message;
        available = true;
        notifyAll(); // Notify waiting threads
    }
}
```

Thread Pools:

- Thread pools are a mechanism for managing and reusing threads in an application.
- Instead of creating a new thread for each task, a thread pool maintains a pool of worker threads that can be used to execute tasks.
- Java provides the `Executor` framework for managing thread pools, which includes classes like `ThreadPoolExecutor` and `Executors`.
- Thread pools improve performance by reducing the overhead of

thread creation and destruction.

- Example of using a thread pool:

```
ExecutorService executor = Executors.newFixedThreadPool(5);

for (int i = 0; i < 10; i++) {
    Runnable task = new MyTask();
    executor.execute(task);
}

executor.shutdown();
```

In the example above, a fixed-size thread pool is created using `Executors.newFixedThreadPool()`. Tasks are submitted to the thread pool using the `execute()` method of the `ExecutorService`.

Java I/O and Networking:

File Handling:

- File handling in Java involves reading from or writing to files on the file system.
- The `java.io` package provides classes and interfaces for file I/O operations.
- Examples of file handling operations include creating, reading, writing, copying, and deleting files.
- File handling operations can be performed using byte streams (`InputStream`, `OutputStream`) or character streams (`Reader`, `Writer`).
- Example of reading from a file using character streams:

```
import java.io.*;

public class FileReaderExample {
    public static void main(String[] args) {
        try (BufferedReader reader = new BufferedReader(new FileReader("input.txt")))
 {
            String line;
            while ((line = reader.readLine()) != null) {
                System.out.println(line);
            }
        } catch (IOException ex) {
            System.out.println("Error reading file: " + ex.getMessage());
        }
    }
}
```

Streams (Byte Streams and Character Streams):

- Streams in Java are used for reading from or writing to a source, such as files, network connections, or in-memory buffers.
- Byte streams (`InputStream`, `OutputStream`) are used for binary data, while character streams (`Reader`, `Writer`) are used for text data.
- Byte streams provide methods for reading or writing individual bytes or arrays of bytes.
- Character streams provide methods for reading or writing characters or strings.
- Example of writing to a file using byte streams:

```
import java.io.*;

public class FileOutputStreamExample {
    public static void main(String[] args) {
        try (FileOutputStream fos = new FileOutputStream("output.txt")) {
            String message = "Hello, World!";
            byte[] data = message.getBytes();
            fos.write(data);
        } catch (IOException ex) {
            System.out.println("Error writing to file: " + ex.getMessage());
        }
    }
}
```

Network Programming (TCP/IP, UDP, Sockets):

- Network programming in Java involves communication between client and server applications over a network.

- Java provides classes and interfaces in the `java.net` package for network programming.
- TCP/IP (Transmission Control Protocol/Internet Protocol) and UDP (User Datagram Protocol) are the two main protocols used for network communication.
- Sockets are the endpoints for network communication in Java.
- Examples of network programming include creating client-server applications, sending and receiving data over a network, and establishing socket connections.
- Example of a TCP/IP client-server application:

```
import java.io.*;
import java.net.*;

// Server
public class Server {
    public static void main(String[] args) {
        try (ServerSocket serverSocket = new ServerSocket(1234)) {
            System.out.println("Server listening on port 1234...");
            Socket socket = serverSocket.accept();
            System.out.println("Client connected: " + socket.getInetAddress());

            // Handle client communication
            // ...

            socket.close();
        } catch (IOException ex) {
            System.out.println("Server error: " + ex.getMessage());
        }
    }
}

// Client
public class Client {
    public static void main(String[] args) {
        try (Socket socket = new Socket("localhost", 1234)) {
            System.out.println("Connected to server: " + socket.getInetAddress());

            // Handle server communication
            // ...

            socket.close();
        } catch (IOException ex) {
            System.out.println("Client error: " + ex.getMessage());
        }
    }
}
```

In the example above, the server listens on port 1234 using a `ServerSocket`. The client connects to the server using a `Socket` with the server's IP address and port

number.

JDBC (Java Database Connectivity):

Introduction to Databases:

- Databases are used to store and manage structured data efficiently.
- Databases provide features such as data persistence, data retrieval, data manipulation, and data integrity.
- Relational databases are the most common type of databases, which organize data into tables with rows and columns.
- Popular relational database management systems (RDBMS) include MySQL, Oracle, PostgreSQL, and SQL Server.

Connecting to Databases:

- JDBC (Java Database Connectivity) is a Java API for connecting to databases and executing SQL queries.
- The JDBC API provides a set of interfaces and classes for database connectivity.
- To connect to a database, you need to load the appropriate JDBC driver and establish a connection using a connection string that specifies the database URL, username, and password.
- Example of connecting to a MySQL database using JDBC:

```
import java.sql.*;

public class DatabaseConnection {
    public static void main(String[] args) {
        String url = "jdbc:mysql://localhost:3306/mydatabase";
        String username = "root";
        String password = "password";

        try (Connection connection = DriverManager.getConnection(url, username, passwo
rd)) {
            System.out.println("Connected to the database!");
        } catch (SQLException ex) {
            System.out.println("Database connection error: " + ex.getMessage());
        }
    }
}
```

Executing SQL Queries:

- Once connected to a database, you can execute SQL queries to retrieve or modify data.
- The JDBC API provides interfaces such as `Statement` and `PreparedStatement` for executing SQL statements.
- `Statement` is used for simple SQL queries, while `PreparedStatement` is used for parameterized queries to prevent SQL injection.
- Example of executing a SELECT query using JDBC:

```
try (Statement statement = connection.createStatement()) {
    String query = "SELECT * FROM employees";
    ResultSet resultSet = statement.executeQuery(query);

    while (resultSet.next()) {
        int id = resultSet.getInt("id");
        String name = resultSet.getString("name");
        int age = resultSet.getInt("age");
        System.out.println("ID: " + id + ", Name: " + name + ", Age: " + age);
    }
} catch (SQLException ex) {
    System.out.println("Query execution error: " + ex.getMessage());
}
```

Transaction Management:

- Transactions ensure the atomicity, consistency, isolation, and durability (ACID) properties of database operations.
- In JDBC, you can manage transactions using the `Connection` object.
- By default, JDBC operates in auto-commit mode, where each SQL statement is treated as a separate transaction.
- To perform multiple SQL statements as a single transaction, you can disable auto-commit and explicitly commit or rollback the transaction.
- Example of transaction management using JDBC:

```
try {
    connection.setAutoCommit(false);

    // Perform multiple SQL statements

    connection.commit(); // Commit the transaction
} catch (SQLException ex) {
```

```
    connection.rollback(); // Rollback the transaction
    System.out.println("Transaction error: " + ex.getMessage());
}
```

In the example above, the `setAutoCommit(false)` method disables auto-commit, allowing multiple SQL statements to be executed as a single transaction. If an exception occurs, the transaction is rolled back using the `rollback()` method.

That covers the brief notes and examples for JDBC (Java Database Connectivity). If you have any more topics, please let me know!

Java GUI (Graphical User Interface) Programming:

Introduction to Swing:

- Swing is a Java GUI toolkit provided by Oracle for building desktop applications.
- It offers a rich set of components, such as buttons, labels, text fields, and dialogs, for creating interactive user interfaces.
- Swing components are lightweight and platform-independent, allowing applications to run on different operating systems.

Components (Buttons, Labels, Text Fields, etc.):

- Swing provides a wide range of components that can be used to build GUIs.
- Common Swing components include buttons, labels, text fields, checkboxes, radio buttons, lists, tables, and panels.
- Components are added to containers, such as frames or panels, to form the GUI layout.
- Example of creating and adding components to a JFrame:

```
import javax.swing.*;

public class MyGUIApp {
    public static void main(String[] args) {
        JFrame frame = new JFrame("My GUI App");
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

        JLabel label = new JLabel("Welcome to Swing!");
        JButton button = new JButton("Click Me");
        JTextField textField = new JTextField();
```

```
        frame.add(label);
        frame.add(button);
        frame.add(textField);

        frame.pack();
        frame.setVisible(true);
    }
}
```

Event Handling:

- Event handling in Swing allows components to respond to user actions, such as button clicks or mouse movements.
- Events are generated by user interactions or system actions, and event listeners are used to handle these events.
- Swing components use the observer design pattern, where listeners are registered with components to receive and handle events.
- Example of adding an event listener to a button:

```
button.addActionListener(e -> {
    // Handle button click event
    System.out.println("Button clicked!");
});
```

Layout Managers:

- Layout managers in Swing are used to arrange and position components within containers.
- They automatically handle the sizing and positioning of components based on specified rules.
- Common layout managers include `FlowLayout`, `BorderLayout`, `GridLayout`, and `GridBagLayout`.
- Layout managers provide flexibility in designing GUIs that can adapt to different screen sizes and resolutions.
- Example of using a layout manager to arrange components:

```
import javax.swing.*;
import java.awt.*;

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

```
        JFrame frame = new JFrame("My GUI App");
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

        JPanel panel = new JPanel();
        panel.setLayout(new FlowLayout());

        JLabel label = new JLabel("Welcome to Swing!");
        JButton button = new JButton("Click Me");
        JTextField textField = new JTextField(20);

        panel.add(label);
        panel.add(button);
        panel.add(textField);

        frame.add(panel);
        frame.pack();
        frame.setVisible(true);
    }
}
```

In the example above, a `JPanel` is used as a container with the `FlowLayout` layout manager to arrange the components in a horizontal flow.

Java Reflection:

Introduction to Reflection:

- Reflection is a powerful feature in Java that allows the inspection and manipulation of classes, interfaces, methods, and fields at runtime.
- It enables dynamic access to class information and the ability to create, modify, or invoke objects and their members dynamically.
- Reflection is often used in frameworks, libraries, and tools that require runtime analysis or dynamic behavior.

Obtaining Class Information:

- The `java.lang.Class` class provides methods to obtain information about a class at runtime.
- Reflection allows you to retrieve information such as class name, superclass, implemented interfaces, constructors, methods, and fields.
- Example of obtaining class information using reflection:

```
import java.lang.reflect.*;

public class ReflectionExample {
```

```
    public static void main(String[] args) {
        Class<Person> personClass = Person.class;

        // Get class name
        String className = personClass.getName();
        System.out.println("Class Name: " + className);

        // Get superclass
        Class<? super Person> superClass = personClass.getSuperclass();
        System.out.println("Superclass: " + superClass.getName());

        // Get implemented interfaces
        Class<?>[] interfaces = personClass.getInterfaces();
        for (Class<?> intf : interfaces) {
            System.out.println("Interface: " + intf.getName());
        }

        // Get constructors
        Constructor<?>[] constructors = personClass.getConstructors();
        for (Constructor<?> constructor : constructors) {
            System.out.println("Constructor: " + constructor.getName());
        }

        // Get methods
        Method[] methods = personClass.getMethods();
        for (Method method : methods) {
            System.out.println("Method: " + method.getName());
        }

        // Get fields
        Field[] fields = personClass.getFields();
        for (Field field : fields) {
            System.out.println("Field: " + field.getName());
        }
    }
}

class Person {
    private String name;
    public int age;

    public Person(String name, int age) {
        this.name = name;
        this.age = age;
    }

    public void sayHello() {
        System.out.println("Hello, I'm " + name);
    }
}
```

Dynamic Class Loading:

- Reflection allows dynamic loading of classes at runtime using the `ClassLoader` class.

- Dynamic class loading enables the loading and instantiation of classes based on runtime conditions or configurations.
- Example of dynamically loading a class using reflection:

```
public class DynamicClassLoading {
    public static void main(String[] args) {
        try {
            Class<?> calculatorClass = Class.forName("com.example.Calculator");
            Object calculator = calculatorClass.newInstance();

            Method addMethod = calculatorClass.getMethod("add", int.class, int.class);
            int result = (int) addMethod.invoke(calculator, 5, 3);
            System.out.println("Result: " + result);
        } catch (ClassNotFoundException | InstantiationException |
                 IllegalAccessException | NoSuchMethodException |
                 InvocationTargetException ex) {
            System.out.println("Error: " + ex.getMessage());
        }
    }
}

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

Accessing and Modifying Objects at Runtime:

- Reflection allows you to access and modify the fields and methods of objects dynamically at runtime.
- You can retrieve field values, set field values, invoke methods, and perform other operations on objects using reflection.
- Example of accessing and modifying objects at runtime using reflection:

```
public class ObjectReflection {
    public static void main(String[] args) {
        Person person = new Person("John Doe", 25);

        try {
            Class<?> personClass = person.getClass();

            // Accessing field values
            Field

 nameField = personClass.getDeclaredField("name");
            nameField.setAccessible(true);
            String name = (String) nameField.get(person);
```

```
            System.out.println("Name: " + name);

            // Modifying field values
            Field ageField = personClass.getDeclaredField("age");
            ageField.setAccessible(true);
            ageField.setInt(person, 30);
            System.out.println("Modified Age: " + person.age);

            // Invoking methods
            Method sayHelloMethod = personClass.getDeclaredMethod("sayHello");
            sayHelloMethod.setAccessible(true);
            sayHelloMethod.invoke(person);
        } catch (NoSuchFieldException | IllegalAccessException |
                 NoSuchMethodException | InvocationTargetException ex) {
            System.out.println("Error: " + ex.getMessage());
        }
    }
}
```

In the example above, reflection is used to access and modify the private fields of a `Person` object. The `getDeclaredField()` method is used to retrieve the field, and the `setAccessible()` method is used to allow access to private fields. Similarly, the `getDeclaredMethod()` method is used to retrieve the method, and the `invoke()` method is used to invoke the method on the object.

Java 8 Features:

Lambda Expressions:

- Lambda expressions introduce a concise syntax for writing anonymous functions in Java.
- They enable functional programming by treating functions as first-class citizens.
- Lambda expressions are commonly used in functional interfaces to represent behavior that can be passed as arguments or assigned to variables.
- Example of using lambda expressions:

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

// Using lambda expression to sort the names
Collections.sort(names, (a, b) -> a.compareTo(b));

// Using lambda expression in forEach loop
names.forEach(name -> System.out.println(name));
```

Functional Interfaces:

- Functional interfaces are interfaces that have a single abstract method.
- They are used as the basis for lambda expressions and method references.
- Java 8 introduced the `java.util.function` package, which provides a set of functional interfaces, such as `Predicate`, `Function`, and `Consumer`.
- Example of using functional interfaces:

```
Predicate<Integer> isEven = num -> num % 2 == 0;
System.out.println(isEven.test(4)); // true

Function<String, Integer> lengthFunc = str -> str.length();
int length = lengthFunc.apply("Hello"); // 5

Consumer<String> printUpperCase = str -> System.out.println(str.toUpperCase());
printUpperCase.accept("java"); // JAVA
```

Stream API:

- The Stream API provides a declarative and functional way of processing collections of objects.
- Streams allow for operations like filtering, mapping, reducing, and collecting on collections in a concise manner.
- Stream operations can be performed sequentially or in parallel to leverage multi-core processors.
- Example of using the Stream API:

```
List<Integer> numbers = Arrays.asList(1, 2, 3, 4, 5);

int sum = numbers.stream()
                .filter(n -> n % 2 == 0)
                .mapToInt(n -> n * 2)
                .sum();

System.out.println(sum); // 12
```

Default and Static Methods in Interfaces:

- Java 8 introduced default and static methods in interfaces.
- Default methods provide a default implementation for interface methods, allowing backward compatibility for existing implementations.

- Static methods in interfaces allow utility methods to be defined directly in the interface.
- Example of default and static methods in interfaces:

```
interface Vehicle {
    default void start() {
        System.out.println("Starting the vehicle...");
    }

    static void honk() {
        System.out.println("Honking the horn!");
    }
}

class Car implements Vehicle {
    // No need to implement the default start() method

    public static void main(String[] args) {
        Car car = new Car();
        car.start(); // Default method from Vehicle interface
        Vehicle.honk(); // Static method from Vehicle interface
    }
}
```

Date and Time API:

- Prior to Java 8, date and time manipulation was done using the `java.util.Date` and `java.util.Calendar` classes, which were error-prone and cumbersome.
- Java 8 introduced a new Date and Time API (`java.time`) that provides a more comprehensive and intuitive way of working with dates, times, and intervals.
- The new API includes classes such as `LocalDate`, `LocalTime`, `LocalDateTime`, `ZonedDateTime`, and `Duration`, among others.
- Example of using the Date and Time API:

```
LocalDate currentDate = LocalDate.now();
System.out.println("Current Date: " + currentDate);

LocalTime currentTime = LocalTime.now();
System.out.println("Current Time: " + currentTime);

LocalDateTime currentDateTime = LocalDateTime.now();
System.out.println("Current Date and Time: " + currentDateTime);
```

Design Patterns:

Design patterns are proven solutions to common problems that occur in software design. They provide reusable and well-tested solutions that can help in creating flexible, maintainable, and scalable software systems. Here are some commonly known design patterns:

Creational Patterns:

- Singleton: Ensures that only one instance of a class is created and provides a global point of access to it.
- Factory Method: Defines an interface for creating objects, but lets subclasses decide which class to instantiate.
- Abstract Factory: Provides an interface for creating families of related or dependent objects without specifying their concrete classes.
- Builder: Separates the construction of complex objects from their representation, allowing the same construction process to create different representations.
- Prototype: Creates new objects by copying existing objects and modifying them as needed.

Structural Patterns:

- Adapter: Converts the interface of a class into another interface that clients expect, allowing classes with incompatible interfaces to work together.
- Decorator: Dynamically adds responsibilities to an object by wrapping it in an object of a decorator class.
- Composite: Composes objects into tree structures to represent part-whole hierarchies, allowing clients to treat individual objects and compositions uniformly.
- Proxy: Provides a surrogate or placeholder for another object to control access to it.
- Facade: Provides a simplified interface to a complex subsystem, making it easier to use.

Behavioral Patterns:

- Observer: Defines a one-to-many dependency between objects, so that when one object changes state, all its dependents are notified and updated automatically.

- Strategy: Defines a family of algorithms, encapsulates each one, and makes them interchangeable, allowing algorithms to be selected at runtime.
- Template Method: Defines the skeleton of an algorithm in a base class, allowing subclasses to redefine certain steps of the algorithm without changing its structure.
- Command: Encapsulates a request as an object, allowing the parameterization of clients with different requests, queues, or log requests, and supports undoable operations.
- Iterator: Provides a way to access the elements of an aggregate object sequentially without exposing its underlying representation.

These are just a few examples of design patterns. There are many more design patterns available, each addressing different design problems and providing effective solutions. It's important to choose the appropriate design pattern based on the problem at hand and the specific requirements of the software system.

Remember that design patterns are not strict rules but guidelines that can be adapted and modified to suit the specific needs of a project. They help in achieving software that is modular, maintainable, and extensible.

Java Best Practices and Coding Standards:

Naming Conventions:

- Use meaningful and descriptive names for variables, methods, classes, and packages.
- Follow camelCase naming for variables and methods (e.g., `firstName`, `calculateTotal`).
- Use PascalCase naming for class and interface names (e.g., `Customer`, `OrderService`).
- Use lowercase for package names (e.g., `com.example.myproject`).
- Constants should be in uppercase with underscores separating words (e.g., `MAX_VALUE`).

Code Formatting:

- Use consistent indentation (typically 4 spaces) to improve code readability.

- Use braces `{}` for control structures and loop bodies, even if they contain a single statement.
- Limit line length to around 80-120 characters to improve code readability.
- Add appropriate whitespace between operators, keywords, and operands to improve code readability.
- Follow a consistent and logical order for class members (fields, constructors, methods).

Exception Handling Best Practices:

- Catch specific exceptions rather than using general catch blocks.
- Handle exceptions at the appropriate level in the code hierarchy.
- Log exceptions or provide meaningful error messages for debugging and troubleshooting.
- Avoid catching exceptions unless you can handle them appropriately.
- Favor checked exceptions for conditions that can be recovered from, and unchecked exceptions for fatal or unexpected conditions.

Memory Management:

- Properly manage object creation and destruction to prevent memory leaks.
- Release resources explicitly when they are no longer needed, such as closing files or database connections.
- Use try-with-resources or finally blocks to ensure resources are always released, even in the presence of exceptions.
- Avoid excessive object creation or unnecessary object cloning to conserve memory.
- Use appropriate data structures and algorithms to optimize memory usage.

Performance Optimization:

- Profile and benchmark your code to identify performance bottlenecks.
- Use efficient algorithms and data structures to optimize time and space complexity.
- Minimize object creation and unnecessary memory allocation.

- Use appropriate collection types based on the requirements (e.g., ArrayList vs LinkedList).
- Cache frequently used data or expensive computations, where appropriate.
- Use concurrency and parallelism techniques to utilize multi-core processors.

In addition to the above, it's important to follow general best practices such as writing modular and reusable code, documenting your code, writing meaningful comments, and writing unit tests to ensure code correctness.

Adhering to coding standards and best practices improves code maintainability, readability, and collaboration among developers. It also helps in producing high-quality code that is easier to debug, maintain, and extend.

Remember to also consider any specific coding standards or guidelines provided by your organization or development team.

Java Testing and Debugging:

Unit Testing (JUnit):

- Unit testing is a fundamental part of software development that ensures individual units of code are functioning correctly.
- JUnit is a popular testing framework for Java that provides annotations, assertions, and test runners to write and execute unit tests.
- Write tests that cover different scenarios and edge cases to ensure robust code.
- Test methods should be independent, isolated, and repeatable.
- Use assertions to validate expected results and behavior.
- Use test fixtures and setup methods to prepare the environment for testing.
- Organize tests into test suites and test classes for better maintainability.
- Example of a simple JUnit test:

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

public class CalculatorTest {

    @Test
    public void testAddition() {
```

```
        Calculator calculator = new Calculator();
        int result = calculator.add(2, 3);
        Assertions.assertEquals(5, result);
    }
}
```

Debugging Techniques and Tools:

- Debugging is the process of identifying and resolving issues or bugs in your code.
- Use print statements or logging to display variable values and trace program flow.
- Utilize breakpoints to pause code execution at specific lines and inspect variables.
- Step through the code line by line to understand the execution flow.
- Use the debugging tools provided by Integrated Development Environments (IDEs) such as IntelliJ IDEA, Eclipse, or NetBeans.
- Explore features like variable inspection, call stack, watches, and expression evaluation in your debugger.
- Use conditional breakpoints to break execution only when specific conditions are met.
- Analyze error messages, exceptions, and stack traces to identify the root cause of issues.
- Write isolated test cases to reproduce and debug specific problems.
- Collaborate with colleagues or use code review tools to get a fresh perspective on your code.

Remember to use debugging techniques and tools effectively to diagnose and resolve issues efficiently. Understanding the program flow, identifying variables' values, and isolating the problem area are crucial in successful debugging.

Introduction to JavaFX:

JavaFX is a Java framework for building desktop applications with rich graphical user interfaces (GUIs). It provides a set of libraries and APIs to create interactive and visually appealing applications. Here are the key concepts of JavaFX:

JavaFX Basics:

- JavaFX applications are built using a scene graph, which represents the visual hierarchy of UI elements.
- The entry point of a JavaFX application is the `Application` class, which contains the `start()` method.
- The `start()` method sets up the primary stage (window) and creates the scene graph.
- JavaFX applications require the JavaFX runtime environment, which is included in the Java Development Kit (JDK) since Java 8.

GUI Components:

- JavaFX provides a wide range of GUI components, including buttons, labels, text fields, checkboxes, radio buttons, lists, tables, and more.
- Components are organized in a hierarchical structure, with the `Scene` as the top-level container and `Parent` classes as intermediate containers.
- Layout managers, such as `VBox`, `HBox`, `BorderPane`, and `GridPane`, are used to control the positioning and sizing of components within containers.
- JavaFX also supports CSS styling for customizing the appearance of components.

Event Handling in JavaFX:

- Event handling in JavaFX follows the event-driven programming paradigm.
- JavaFX provides an event model based on the observer pattern, where event sources generate events and event handlers respond to them.
- Event sources include GUI components like buttons, mouse clicks, key presses, etc.
- Event handlers are implemented as event listener interfaces, such as `EventHandler` and `ChangeListener`, or using lambda expressions.
- Event handling in JavaFX can be done by registering event handlers using the `setOn<EventName>()` methods or through FXML file bindings.

Example of JavaFX Application:
Here's a simple JavaFX application that displays a window with a button and handles its click event:

```
import javafx.application.Application;
import javafx.scene.Scene;
```

```
import javafx.scene.control.Button;
import javafx.scene.layout.StackPane;
import javafx.stage.Stage;

public class HelloWorld extends Application {

    public static void main(String[] args) {
        launch(args);
    }

    @Override
    public void start(Stage primaryStage) {
        Button button = new Button("Click Me");
        button.setOnAction(event -> System.out.println("Button clicked!"));

        StackPane root = new StackPane();
        root.getChildren().add(button);

        Scene scene = new Scene(root, 300, 200);

        primaryStage.setTitle("Hello World");
        primaryStage.setScene(scene);
        primaryStage.show();
    }
}
```

In this example, we create a `Button` and add an event handler using a lambda expression. The button is placed in a `StackPane`, which is then added to the `Scene` of the primary stage. Finally, the stage is displayed.

JavaFX offers many more features, including animations, multimedia, charts, and CSS styling. It provides a modern and flexible platform for developing desktop applications with rich user interfaces.

Note: JavaFX has been decoupled from the core JDK since Java 11 and is now available as a separate library. It is recommended to use the latest version of JavaFX with the corresponding JDK.