Object Orientation:

Object-oriented programming (OOP) is a programming paradigm that revolves around the concept of "objects". An object is a self-contained entity that consists of both data (attributes or properties) and behavior (methods or functions). Here's a simple example in Java:

```
// Define a class representing a Car
class Car {
    // Attributes
    String make;
    String model;
    int year;

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

    // Method to display information about the car
    public void displayInfo() {
        System.out.println("Make: " + make);
        System.out.println("Model: " + model);
        System.out.println("Year: " + year);
    }
}

public class Main {
    public static void main(String[] args) {
        // Create an instance of Car
        Car myCar = new Car("Toyota", "Camry", 2020);

        // Call the displayInfo method to display car information
        myCar.displayInfo();
    }
}
```

Generics and Collections:

Generics allow us to create classes, interfaces, and methods that operate with types specified later (during instantiation). They provide compile-time type safety. Collections, on the other hand, are objects that group multiple elements into a single unit. Java provides a rich set of collection classes in the **java.util** package. Here's an example:

```
import java.util.ArrayList;

public class Main {
    public static void main(String[] args) {
        // Create a list of integers using ArrayList (a type of
Collection)
        ArrayList<Integer> numbers = new ArrayList<>();

        // Add elements to the list
        numbers.add(10);
        numbers.add(20);
        numbers.add(30);
```

```
        // Iterate through the list and print each element
        for (Integer num : numbers) {
            System.out.println(num);
        }
    }
}
```

OOP Feature

In this example:

- We define an abstract class **Shape** with an abstract method **calculateArea()**.
- Two concrete classes **Rectangle** and **Circle** inherit from **Shape** and implement **calculateArea()** method.
- We define a class **Dog** that inherits from another class **Animal** and overrides the **sound()** method.
- We define a class **Student** to demonstrate simple class and object creation.
- Finally, we define a class **Calculator** to demonstrate method overloading.

```
// Define an abstract class representing a Shape
abstract class Shape {
    // Abstract method to calculate area
    public abstract double calculateArea();
}
 class Animal {
    // Abstract method to calculate area
    public  void sound(){
     System.out.print("The animal sounds");
    }
}

// Define a class representing a Rectangle that inherits from Shape
class Rectangle extends Shape {
    double length;
    double width;

    // Constructor
    public Rectangle(double length, double width) {
        this.length = length;
        this.width = width;
    }

    // Override abstract method to calculate area
    @Override
    public double calculateArea() {
        return length * width;
```

```
    }
}

// Define a class representing a Circle that inherits from Shape
class Circle extends Shape {
    double radius;

    // Constructor
    public Circle(double radius) {
        this.radius = radius;
    }

    // Override abstract method to calculate area
    @Override
    public double calculateArea() {
        return Math.PI * radius * radius;
    }
}

// Define a class representing a Dog that inherits from Animal
class Dog extends Animal {
    @Override
    public void sound() {
        System.out.println("Dog barks");
    }
}

// Define a class representing a Student
class Student {
    // Attributes
    String name;
    int age;

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

    // Method to display information about the student
    public void displayInfo() {
        System.out.println("Name: " + name);
        System.out.println("Age: " + age);
    }
}

// Define a class representing a Calculator
class Calculator {
```

```java
    // Method to add two integers
    public int add(int a, int b) {
        return a + b;
    }

    // Method to add two doubles
    public double add(double a, double b) {
        return a + b;
    }
}

public class Main {
    public static void main(String[] args) {
        // Create an instance of Rectangle
        Rectangle rectangle = new Rectangle(5, 3);

        // Calculate and display area of the rectangle
        System.out.println("Area of the rectangle: " +
rectangle.calculateArea());

        // Create an instance of Circle
        Circle circle = new Circle(5);

        // Calculate and display area of the circle
        System.out.println("Area of the circle: " +
circle.calculateArea());

        // Create an instance of Dog
        Dog dog = new Dog();

        // Call the sound method of Dog
        dog.sound();

        // Create an instance of Student
        Student student = new Student("John", 20);

        // Call the displayInfo method to display student information
        student.displayInfo();

        // Create an instance of Calculator
        Calculator calc = new Calculator();

        // Call the overloaded add methods
        System.out.println("Sum of integers: " + calc.add(5, 3));
        System.out.println("Sum of doubles: " + calc.add(2.5, 3.5));
    }
}
```

Access Control:

Access control in Java refers to the mechanism of controlling the visibility and accessibility of classes, methods, and variables.

1. **Access Modifiers**: Java provides four access modifiers to control access to classes, methods, and variables:
    - **public**: Accessible from anywhere.
    - **protected**: Accessible within the same package or by subclasses.
    - **default** (no modifier): Accessible within the same package.
    - **private**: Accessible only within the same class.

```
// Example of access control in Java

// Class with different access modifiers
class MyClass {
    public int publicVar; // Public variable
    protected int protectedVar; // Protected variable
    int defaultVar; // Default variable
    private int privateVar; // Private variable

    // Method with different access modifiers
    public void publicMethod() {
        System.out.println("Public method");
    }

    protected void protectedMethod() {
        System.out.println("Protected method");
    }

    void defaultMethod() {
        System.out.println("Default method");
    }

    private void privateMethod() {
        System.out.println("Private method");
    }
}

// Another class to demonstrate access
public class AnotherClass {
    public static void main(String[] args) {
        MyClass obj = new MyClass();

        // Accessing variables and methods from MyClass
        obj.publicVar = 10; // Public variable
```

```
        obj.publicMethod(); // Public method

        // Other variables and methods are not accessible here
    }
}
```

Exceptions

Exceptions in Java are a way to handle errors and unexpected situations that may occur during the execution of a program. Exceptions provide a mechanism to gracefully handle errors and recover from them without terminating the program abruptly.

```
public class ExceptionExample {
    public static void main(String[] args) {
        try {
            // Code that may throw an exception
            int result = divide(10, 0); // This will throw an
ArithmeticException
            System.out.println("Result: " + result); // This line will not be
executed
        } catch (ArithmeticException e) {
            // Catch block to handle the exception
            System.out.println("Error: Division by zero");
        } finally {
            // Finally block, always executes regardless of whether an
exception occurs
            System.out.println("Finally block executed");
        }
    }

    public static int divide(int a, int b) {
        // Method that may throw an exception
        return a / b; // This may throw an ArithmeticException if b is zero
    }
}
```

In this example:

- We have a **divide** method that attempts to perform division. If the divisor (**b**) is zero, it will throw an **ArithmeticException**.
- In the **main** method, we call the **divide** method within a **try** block. If an exception occurs during the execution of the **try** block, it is caught by the corresponding **catch** block.
- The **catch** block handles the **ArithmeticException** by printing an error message.
- The **finally** block always executes, regardless of whether an exception occurs. It is commonly used for cleanup tasks, such as closing resources.

Java provides a hierarchy of exception classes, with **Throwable** as the root class. Exceptions can be categorized into two types: checked exceptions and unchecked exceptions.

- **Checked Exceptions**: These are exceptions that the compiler forces you to handle or declare. Examples include **IOException**, **SQLException**, etc.
- **Unchecked Exceptions**: These are exceptions that do not need to be explicitly handled or declared. Examples include **NullPointerException**, **ArrayIndexOutOfBoundsException**, etc.

You can create your own custom exceptions by extending the **Exception** class or one of its subclasses.

Here are some scenarios where you might use a **finally** block:

1. **Resource Cleanup**: You can use the **finally** block to release resources, such as closing files, database connections, or network sockets, that were acquired in the **try** block. This ensures that resources are properly cleaned up even if an exception occurs.
2. **Guaranteed Execution**: Code inside the **finally** block is guaranteed to execute, regardless of whether an exception occurs in the **try** block or not. This can be useful for tasks that should always be performed, such as logging or finalizing operations.

Gradle Fundamentals:

What is Gradle? Gradle is a powerful build automation tool used primarily for Java projects, but it's also widely adopted in other languages and platforms. It uses a Groovy or Kotlin-based DSL (domain-specific language) for describing build configurations. .It's often used as an alternative to Apache Maven and Ant.

Why is it used? Gradle is used to automate the build process of software projects. It helps in compiling source code, managing dependencies, running tests, packaging applications, and more. By automating these tasks, Gradle improves productivity, reduces errors, and ensures consistency in the build process.

Advantages over others:

- **Flexibility:** Gradle offers flexibility in defining build scripts and supports multiple programming languages and platforms.
- **Performance:** Gradle's incremental build feature ensures that only modified components are recompiled, leading to faster build times.
- **Dependency Management:** Gradle integrates seamlessly with dependency management systems like Maven and Ivy, making it easy to manage project dependencies.
- **Plugin Ecosystem:** Gradle has a rich ecosystem of plugins that extend its functionality and provide support for various tasks and integrations.
- **Customization:** Gradle allows for fine-grained customization of build processes, enabling developers to tailor the build to their specific requirements.

Features:

- Declarative Build Scripts: Build configurations are defined using a Groovy or Kotlin-based DSL, which provides a declarative approach to describing build tasks.
- Dependency Management: Gradle manages project dependencies and allows for easy resolution and retrieval of libraries from remote repositories.
- Incremental Builds: Gradle's incremental build feature ensures that only modified components are rebuilt, improving build performance.
- Multi-project Builds: Gradle supports multi-project builds, allowing developers to manage dependencies and build configurations across multiple projects.
- Plugin System: Gradle has a robust plugin system that extends its functionality and provides support for various tasks and integrations.

Installation (brief): To install Gradle, you can download the distribution from the official Gradle website (https://gradle.org/install/) and follow the installation instructions provided. Alternatively, you can use a package manager like SDKMAN! (https://sdkman.io/) to install Gradle on Unix-based systems. Once installed, you can verify the installation by running the **gradle --version** command in your terminal.

Test-Driven Development (TDD) with JUnit 5:

What is Test-Driven Development (TDD)? Test-Driven Development (TDD) is a software development approach where tests are written before the actual implementation code. The process involves writing a failing test case, writing the minimum amount of code to make the test pass, and then refactoring the code to improve its design while ensuring that all tests still pass.

Why is it used? TDD helps improve code quality, maintainability, and reliability by ensuring that the code meets the specified requirements and behaves as expected. It encourages better design practices, such as writing modular and testable code, and helps catch bugs early in the development cycle, reducing the cost of fixing them later.

Advantages of TDD:

- **Improved Code Quality:** TDD encourages writing clean, modular, and testable code, leading to higher code quality.
- **Faster Feedback:** TDD provides immediate feedback on the correctness of the code through automated tests, enabling developers to catch and fix issues early.
- **Better Design:** TDD promotes better design practices by focusing on writing small, cohesive units of code that are easier to understand, maintain, and refactor.
- **Reduced Debugging Time:** Since bugs are caught early in the development process, TDD helps reduce the time spent on debugging and fixing issues later.
- **Regression Testing:** TDD ensures that existing functionality remains intact when new features are added or existing code is modified, thereby preventing regression bugs.

Features of JUnit 5:

- **Annotations:** JUnit 5 provides a set of annotations (e.g., **@Test**, **@BeforeEach**, **@AfterEach**, etc.) for defining test cases, setup, and teardown methods.
- **Assertions:** JUnit 5 provides a rich set of assertion methods (e.g., **assertEquals**, **assertTrue**, **assertNotNull**, etc.) for verifying the expected behavior of the code under test.
- **Parameterized Tests:** JUnit 5 supports parameterized tests, allowing developers to run the same test with different input values.
- **Extensions:** JUnit 5 introduces the concept of extensions, which allow developers to add custom behavior to test classes (e.g., mocking frameworks, test lifecycle callbacks, etc.).
- **Dynamic Tests:** JUnit 5 supports dynamic tests, enabling developers to generate tests dynamically at runtime based on certain conditions.

Installation (brief): JUnit 5 can be added to a Java project using a build automation tool like Gradle or Maven. For Gradle, you can add the following dependency to your **build.gradle** file:

testImplementation 'org.junit.jupiter:junit-jupiter-api:5.7.2'

testRuntimeOnly 'org.junit.jupiter:junit-jupiter-engine:5.7.2'

Strings in Java:

What are Strings? In Java, strings are sequences of characters. They are objects of the **String** class, which provides various methods for manipulating strings. Strings in Java are immutable, meaning that once created, their values cannot be changed.

Why are Strings used? Strings are used to represent text data in Java programs. They are widely used for storing and manipulating textual information, such as names, messages, file contents, etc. Strings play a crucial role in many aspects of Java programming, including input/output operations, text processing, and user interfaces.

Advantages of Strings:

- **Text Manipulation:** Strings provide methods for various text manipulation operations, such as concatenation, substring extraction, searching, replacing, and formatting.
- **Interoperability:** Strings can be easily converted to and from other data types, such as numeric types, characters, and byte arrays, making them versatile for data conversion tasks.
- **Internationalization:** Java's **String** class provides support for Unicode characters, allowing developers to work with strings in different languages and character sets.
- **Memory Efficiency:** Due to their immutability, strings can be efficiently shared among multiple variables and threads, reducing memory consumption and avoiding data corruption issues.

Common String Operations:

- **Concatenation:** Combining two or more strings together using the **+** operator or the **concat()** method.
- **Substring Extraction:** Extracting a portion of a string using the **substring()** method.

- **Length:** Getting the length of a string using the **length()** method.
- **Searching:** Searching for a specific substring within a string using methods like **indexOf()** or **contains()**.
- **Replacing:** Replacing occurrences of a substring with another substring using the **replace()** method.
- **Formatting:** Formatting strings using methods like **format()** or **printf()** for constructing formatted output.
- **Splitting:** Splitting a string into an array of substrings based on a delimiter using the **split()** method.

```
import java.util.Arrays;
public class StringExample {
    public static void main(String[] args) {
        String str1 = "Hello";
        String str2 = "World";

        // Concatenation
        String greeting = str1 + ", " + str2 + "!";
        System.out.println("Concatenated String: " + greeting);
        //using concat -> String result = str1.concat("
").concat(str2);

        // Substring extraction
        String subStr = greeting.substring(0, 5);
        System.out.println("Substring: " + subStr);

        // Length
        int len = greeting.length();
        System.out.println("Length of String: " + len);

        // Searching
        boolean containsWorld = greeting.contains("World");
        System.out.println("Contains 'World': " + containsWorld);

        // Replacing
        String replacedStr = greeting.replace("World", "Java");
        System.out.println("Replaced String: " + replacedStr);

        // Formatting
        double value = 123.456;
        String formattedStr = String.format("%.2f", value);
        System.out.println("Formatted String: " + formattedStr);

        // Splitting
        String[] parts = greeting.split(",");
        System.out.println("Split String: " + Arrays.toString(parts));
    }
```

```
}

```

The format specifier **"%.2f"** means:

- **%**: Indicates a format specifier.
- **.2**: Specifies that we want two digits after the decimal point.
- **f**: Indicates that the value is a floating-point number.

Input/Output (I/O) Operations in Java:

What is Input/Output (I/O)? Input/Output (I/O) refers to the process of transferring data between a program and external devices such as files, network sockets, or the console. In Java, I/O operations are primarily performed using streams, which represent sequences of data.

Types of Streams:

1. **Byte Streams:** Used for reading and writing raw binary data, such as images, audio, and video files. Example classes include **InputStream** and **OutputStream**.
2. **Character Streams:** Used for reading and writing text data encoded in characters. Example classes include **Reader** and **Writer**.
3. **Buffered Streams:** Wrapper classes that improve I/O performance by buffering data in memory before writing it to or reading it from an underlying stream. Example classes include **BufferedReader** and **BufferedWriter**.
4. **Object Streams:** Used for serializing and deserializing Java objects. Example classes include **ObjectInputStream** and **ObjectOutputStream**.
5. **File Streams:** Specialized streams for reading from and writing to files on the filesystem. Example classes include **FileInputStream** and **FileOutputStream**.
6. **Network Streams:** Streams for reading from and writing to network sockets. Example classes include **SocketInputStream** and **SocketOutputStream**.

Common I/O Operations:

1. **Reading from Input Streams:** Use input streams to read data from sources such as files, network sockets, or standard input (e.g., keyboard).
2. **Writing to Output Streams:** Use output streams to write data to destinations such as files, network sockets, or standard output (e.g., console).
3. **File Operations:** Perform operations such as creating, deleting, renaming, or querying properties of files and directories using classes like **File**.
4. **Exception Handling:** Handle exceptions that may occur during I/O operations, such as file not found, permission denied, or disk full errors.

Example: Reading from a File and Writing to Another File:

```
import java.io.*;

```

```java
public class FileIOExample {
    public static void main(String[] args) {
        try {
            // Reading from input file
            BufferedReader reader = new BufferedReader(new
FileReader("./FSD/input.txt"));
            String line;
            StringBuilder content = new StringBuilder();
            while ((line = reader.readLine()) != null) {
                content.append(line).append("\n");
            }
            reader.close();

            // Writing to output file
            BufferedWriter writer = new BufferedWriter(new
FileWriter("./FSD/output.txt"));
            writer.write(content.toString());
            writer.close();

            System.out.println("File copied successfully!");
        } catch (IOException e) {
            System.err.println("Error reading/writing file: " +
e.getMessage());
        }
    }
}
```

- We use **BufferedReader** to read lines from the input file (**input.txt**) and **FileReader** to create the input stream.
- We use **BufferedWriter** to write data to the output file (**output.txt**) and **FileWriter** to create the output stream.
- We handle **IOException** to catch any exceptions that may occur during file I/O operations.

Important Note: It's essential to close the streams after use to release system resources properly. The **try-with-resources** statement can be used to automatically close the streams.

Formatting and Parsing in Java:

What is Formatting? Formatting refers to the process of converting data from its internal representation to a human-readable format or vice versa. In Java, formatting involves converting data types such as numbers, dates, and times to strings or parsing strings to extract data.

Why Formatting and Parsing? Formatting and parsing are essential for presenting data in a meaningful way to users and for converting user input into a usable format within a Java program. They are used extensively in applications involving user interfaces, data processing, and communication with external systems.

Common Formatting and Parsing Tasks:

1. **Number Formatting:** Formatting numbers to display them with specific precision, locale-specific symbols, and grouping separators.
2. **Date and Time Formatting:** Formatting dates and times according to different patterns, locales, and time zones.
3. **Parsing:** Converting strings into primitive data types (such as integers, floating-point numbers) or objects (such as dates, times) using parsing methods.

Example: Number Formatting and Parsing:

```
import java.text.*;
import java.util.*;

public class NumberFormattingExample {
    public static void main(String[] args) {
        // Number formatting
        double num = 12345.6789;
        NumberFormat formatter = NumberFormat.getInstance(Locale.US);
        String formattedNum = formatter.format(num);
        System.out.println("Formatted Number: " + formattedNum);

        // Number parsing
        String str = "12,345.6789";
        try {
            Number parsedNum = formatter.parse(str);
            double parsedValue = parsedNum.doubleValue();
            System.out.println("Parsed Number: " + parsedValue);
        } catch (ParseException e) {
            System.err.println("Error parsing number: " + e.getMessage());
        }
    }
}
```

- We use **NumberFormat.getInstance(Locale.US)** to create a number formatter for the US locale.
- We format the number **12345.6789** using the formatter, resulting in the formatted string **"12,345.6789"**.
- We parse the string **"12,345.6789"** back into a number using **formatter.parse(str)**, obtaining a **Number** object that we convert to a **double** value.

Common Format Specifiers:

- **Number Format Specifiers: %d** for integers, **%f** for floating-point numbers, **%e** for scientific notation.
- **Date and Time Format Specifiers:** Patterns like **"yyyy-MM-dd"** for dates, **"HH:mm:ss"** for times, and **"yyyy-MM-dd HH:mm:ss"** for date-time combinations.

Formatting and Parsing Classes in Java:

- **NumberFormat:** Used for number formatting and parsing.
- **DateFormat and SimpleDateFormat:** Used for date and time formatting and parsing.
- **DecimalFormat:** A subclass of **NumberFormat** for formatting decimal numbers.
- **MessageFormat:** Used for formatting text messages with variable placeholders.

Formatting and parsing are crucial for handling different types of data in Java applications. They provide flexibility in presenting data to users and interpreting user input accurately.

```
import java.text.*;
import java.util.*;

public class DateFormatExample {
    public static void main(String[] args) {
        // Current date and time
        Date currentDate = new Date();

        // Using DateFormat to format dates
        DateFormat dateFormat = DateFormat.getDateInstance(DateFormat.SHORT,
Locale.US);
        String formattedDate = dateFormat.format(currentDate);
        System.out.println("Formatted Date (Short): " + formattedDate);

        dateFormat = DateFormat.getDateInstance(DateFormat.LONG, Locale.US);
        formattedDate = dateFormat.format(currentDate);
        System.out.println("Formatted Date (Long): " + formattedDate);

        // Using SimpleDateFormat for custom date formatting
        SimpleDateFormat sdf = new SimpleDateFormat("yyyy-MM-dd HH:mm:ss");
        String customFormattedDate = sdf.format(currentDate);
        System.out.println("Custom Formatted Date: " + customFormattedDate);

        // Parsing a date string using SimpleDateFormat
        String dateString = "2024-05-11 15:30:00";
        try {
            Date parsedDate = sdf.parse(dateString);
            System.out.println("Parsed Date: " + parsedDate);
        } catch (ParseException e) {
            System.err.println("Error parsing date: " + e.getMessage());
        }
    }
}
```

Explanation:

1. We create a **Date** object representing the current date and time.

2. We use **DateFormat.getDateInstance(style, locale)** to obtain a date formatter with a specific style (short or long) and locale.
3. We format the current date using the obtained **DateFormat** instance.
4. Similarly, we use **SimpleDateFormat** to create a custom date format pattern and format the current date accordingly.
5. We parse a date string (**"2024-05-11 15:30:00"**) using a **SimpleDateFormat** instance and handle any parsing errors.

```
Formatted Date (Short): 5/20/22
Formatted Date (Long): May 20, 2022
Custom Formatted Date: 2022-05-20 12:28:37
9. Parsed Date: Sat May 11 15:30:00 IST 2024
```

Threads in Java allow concurrent execution of tasks within a single process. They provide a way to execute multiple tasks simultaneously, making use of the available CPU cores efficiently. Here's an overview of threads in Java:

1. **Thread Basics:** In Java, threads are represented by instances of the **Thread** class or by implementing the **Runnable** interface. Threads have their own call stack and can run concurrently with other threads.
2. **Creating Threads:**
    - Extending the **Thread** class: You can create a new thread by extending the **Thread** class and overriding its **run()** method.
    - Implementing the **Runnable** interface: You can create a new thread by implementing the **Runnable** interface and providing the task to be executed in the **run()** method.
3. **Starting Threads:** Once a thread is created, it can be started using the **start()** method. When started, the **run()** method of the thread is invoked, and the thread begins its execution.
4. **Thread Lifecycle:** Threads in Java have several states, including:
    - **New:** The thread is created but not yet started.
    - **Runnable:** The thread is ready to run and waiting for CPU time.
    - **Blocked:** The thread is waiting for a resource (e.g., I/O operation) or waiting to acquire a lock.
    - **Waiting:** The thread is waiting indefinitely for another thread to perform a particular action.
    - **Terminated:** The thread has completed its execution or terminated unexpectedly.
5. **Thread Synchronization:** In a multi-threaded environment, synchronization is essential to prevent data corruption and ensure thread safety. Java provides mechanisms like **synchronized** blocks/methods and locks to synchronize access to shared resources.

6. **Thread Priority:** Java allows you to set the priority of a thread using the **setPriority()** method. Threads with higher priority may get more CPU time, but it's not guaranteed and depends on the underlying platform.
7. **Thread Groups:** Threads can be organized into thread groups for easier management and monitoring. Thread groups provide methods to enumerate threads, set thread priorities, and handle uncaught exceptions.
8. **Daemon Threads:** Daemon threads are background threads that run intermittently in the background to perform tasks such as garbage collection or automatic data updates. They are terminated automatically when all non-daemon threads have terminated.

Threads are a powerful feature of Java that enable concurrent programming and parallel execution of tasks. However, they also introduce challenges such as race conditions and deadlock, which need to be carefully handled to ensure the correctness and reliability of multi-threaded applications.

Example 1: Using Inheritance

```
class ThreadByInheritance extends Thread {
    public void run() {
        System.out.println("Thread running...");
    }
}

public class ThreadByInheritanceExample {
    public static void main(String[] args) {
        // Create and start a thread using inheritance
        ThreadByInheritance thread = new ThreadByInheritance();
        thread.start();

        // Display main thread state
        Thread mainThread = Thread.currentThread();
        System.out.println("Main thread state: " + mainThread.getState());
    }
}
```

- We create a class **ThreadByInheritance** that extends the **Thread** class and overrides the **run()** method.
- In the **main()** method, we create an instance of **ThreadByInheritance**, start it using **start()**, and display the state of the main thread.

Example 2: Using Runnable Interface

```
class ThreadByRunnable implements Runnable {
    public void run() {
        System.out.println("Thread running...");
    }
}

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

```
        // Create a Runnable instance
        ThreadByRunnable runnable = new ThreadByRunnable();
        // Create a Thread instance passing the Runnable
        Thread thread = new Thread(runnable);
        // Start the thread
        thread.start();

        // Display main thread state
        Thread mainThread = Thread.currentThread();
        System.out.println("Main thread state: " + mainThread.getState());
    }
}
```

- We create a class **ThreadByRunnable** that implements the **Runnable** interface and overrides the **run()** method.
- In the **main()** method, we create an instance of **ThreadByRunnable** and pass it to the constructor of **Thread**.
- We start the thread using **start()**.

Generics: Generics in Java allow you to create classes, interfaces, and methods that operate on types specified at compile time. They provide type safety by allowing you to define classes and methods with parameters that work with any data type. Generics are widely used in collections to create type-safe data structures.

Collections: Collections in Java provide a way to store and manipulate groups of objects. They are implemented as classes in the Java Collections Framework and offer various data structures like lists, sets, queues, and maps. Collections provide methods for adding, removing, searching, and iterating over elements.

[Refer classnotes for more examples]

1. **Executor, ExecutorService, ScheduledExecutorService:**
    - **Executor**: Represents an object that executes submitted tasks asynchronously.
    - **ExecutorService**: Extends **Executor** and provides methods to manage the lifecycle of threads.
    - **ScheduledExecutorService**: Extends **ExecutorService** and adds support for scheduling tasks to run after a delay or periodically.

```
2. import java.util.concurrent.*;
3.
4. public class ExecutorExample {
5.     public static void main(String[] args) {
6.         ExecutorService executor = Executors.newFixedThreadPool(2);
7.
8.         executor.submit(() -> {
9.             System.out.println("Task 1 executed");
10.        });
11.
```

```
        executor.submit(() -> {
            System.out.println("Task 2 executed");
        });

        executor.shutdown();
    }
}

```

2. **Future:**

    - Represents the result of an asynchronous computation.
    - Allows you to retrieve the result or handle exceptions once the computation is complete.

```
import java.util.concurrent.*;

public class FutureExample {
    public static void main(String[] args) throws InterruptedException, ExecutionException {
        ExecutorService executor = Executors.newSingleThreadExecutor();

        Future<String> future = executor.submit(() -> {
            Thread.sleep(2000);
            return "Hello from Future";
        });

        // Do other tasks while waiting for the result
        System.out.println("Waiting for future result...");

        String result = future.get(); // Blocking call, waits until the result is available
        System.out.println("Future result: " + result);

        executor.shutdown();
    }
}

```

3. **CountDownLatch, CyclicBarrier, Semaphore:**

    - **CountDownLatch**: Allows one or more threads to wait until a set of operations being performed in other threads completes.
    - **CyclicBarrier**: Synchronizes multiple threads at a barrier point and waits until all threads reach that point.
    - **Semaphore**: Controls access to a shared resource using permits.

```
4. import java.util.concurrent.CountDownLatch;
5.
6. public class CountDownLatchExample {
7.     public static void main(String[] args) throws InterruptedException
   {
8.         CountDownLatch latch = new CountDownLatch(3); // Initialize
   with the number of operations to wait for
9.
10.        Runnable task = () -> {
11.            try {
12.                // Simulate some task
13.                Thread.sleep(1000);
14.                System.out.println("Task completed");
15.                latch.countDown(); // Signal that the task is completed
16.            } catch (InterruptedException e) {
17.                e.printStackTrace();
18.            }
19.        };
20.
21.        // Start three tasks
22.        new Thread(task).start();
23.        new Thread(task).start();
24.        new Thread(task).start();
25.
26.        latch.await(); // Wait until all tasks are completed
27.        System.out.println("All tasks completed");
28.    }
29.}
30.
```

```
import java.util.concurrent.CyclicBarrier;

public class CyclicBarrierExample {
    public static void main(String[] args) {
        CyclicBarrier barrier = new CyclicBarrier(3, () -> {
            System.out.println("Barrier reached, all threads arrived");
        });

        Runnable task = () -> {
            try {
                System.out.println("Thread started");
                Thread.sleep(1000);
                System.out.println("Thread waiting at barrier");
                barrier.await(); // Wait at the barrier point
                System.out.println("Thread resumed");
            } catch (Exception e) {
```

```
                e.printStackTrace();
            }
        };

        new Thread(task).start();
        new Thread(task).start();
        new Thread(task).start();
    }
}
import java.util.concurrent.Semaphore;

public class SemaphoreExample {
    public static void main(String[] args) {
        Semaphore semaphore = new Semaphore(2); // Initialize with the number
of permits

        Runnable task = () -> {
            try {
                semaphore.acquire(); // Acquire a permit
                System.out.println(Thread.currentThread().getName() + "
acquired permit");
                Thread.sleep(1000); // Simulate some task
                semaphore.release(); // Release the permit
                System.out.println(Thread.currentThread().getName() + "
released permit");
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        };

        new Thread(task).start();
        new Thread(task).start();
        new Thread(task).start();
    }
}
```

4. **ThreadFactory:**
    1. Factory for creating new threads.
    2. Allows you to customize thread creation, such as naming, priority, and daemon status.

```
5. import java.util.concurrent.*;
6.
7. public class ThreadFactoryExample {
8.     public static void main(String[] args) {
9.         ThreadFactory threadFactory = Executors.defaultThreadFactory();
10.        Thread thread = threadFactory.newThread(() -> {
```

```
            System.out.println("Thread created by ThreadFactory");
        });
        thread.start();
    }
}
```

5. **BlockingQueue, DelayQueue:**
    - **BlockingQueue**: A queue that supports operations that wait for the queue to become non-empty when retrieving an element and wait for space to become available in the queue when storing an element.
    - **DelayQueue**: A specialized priority queue for delaying the execution of elements based on their expiration time.
6. **Lock:**
    - Provides a higher-level alternative to **synchronized** blocks for controlling access to shared resources.
    - Allows for more fine-grained control over locking and unlocking.

```
import java.util.concurrent.locks.*;

public class LockExample {
    private static int counter = 0;
    private static Lock lock = new ReentrantLock();

    public static void increment() {
        lock.lock();
        try {
            counter++;
        } finally {
            lock.unlock();
        }
    }

    public static void main(String[] args) throws InterruptedException {
        Thread thread1 = new Thread(() -> {
            for (int i = 0; i < 1000; i++) {
                increment();
            }
        });

        Thread thread2 = new Thread(() -> {
            for (int i = 0; i < 1000; i++) {
                increment();
            }
        });
```

```
34.
35.         thread1.start();
36.         thread2.start();
37.         thread1.join();
38.         thread2.join();
39.
40.         System.out.println("Counter: " + counter);
41.     }
42.}
43.
```