

ATME COLLEGE OF ENGINEERING

13th KM Stone, Bannur Road, Mysore - 560 028



A T M E
College of Engineering

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING-CYBER SECURITY



LECTURE NOTES

COURSE: Object Oriented Programming with Java

COURSE CODE: 1BCS302

SEMESTER: III

COURSE COORDINATOR: Dr.Shwetha G K

(ACADEMIC YEAR 2026-27)

INSTITUTIONAL VISION AND MISSION

Vision

Development of academically excellent, culturally vibrant, socially responsible and globally competent human resources.

Mission

- To keep pace with advancements in knowledge and make the students competitive and capable at the global level.
- To create an environment for the students to acquire the right physical, intellectual, emotional and moral foundations and shine as torch bearers of tomorrow's society.
- To strive to attain ever-higher benchmarks of educational excellence.

DEPARTMENT VISION AND MISSION

Vision

To nurture competent and ethical cyber security professionals capable of contributing to secure and resilient digital ecosystems.

Mission:

M1: To impart strong foundational knowledge in cyber security through effective teaching-learning practices and a continuously evolving curriculum that addresses emerging cyber threats and technologies.

M2: To promote research, innovation, and industry collaboration in emerging areas of cyber security to address real-world digital security challenges.

M3: To instill professional ethics, teamwork, leadership qualities, and a sense of social responsibility in addressing real-world cyber challenges.

M4: To foster a culture of lifelong learning among students by encouraging higher studies, professional development, and active engagement with the global cyber security community.

Program Educational Objectives:

PEO1 –Excel as cyber security professionals or entrepreneurs by applying strong fundamentals of cyber security engineering.

PEO2 – Pursue higher studies, research, and professional certifications while engaging in lifelong learning to keep pace with evolving cyber technologies and threats.

PEO3 – Exhibit leadership, teamwork, professional ethics, and social responsibility while addressing cyber security challenges in multidisciplinary environments.

Program Specific Outcomes:

PSO1 – Apply principles of computer science, programming, algorithms, operating systems, and computer networks to design and develop efficient and reliable computing solutions.

PSO2 – Identify, analyze, and mitigate cyber threats and vulnerabilities using techniques related to network security, ethical hacking, penetration testing, and digital forensics.

PSO3 – Apply cyber security policies, cyber laws, risk management practices, and incident response strategies to ensure the protection and resilience of digital systems.

Exceptions: In Java, an exception is an unwanted or unexpected event that occurs during the execution of a program, disrupting the normal flow of the program's instructions.

- It is an **object** that describes an error or an unusual condition that has occurred.
- Java uses exceptions to handle runtime errors, ensuring the program can recover gracefully without abruptly terminating.

Types of Exceptions: In Java, exceptions are classified into two main categories: **Checked Exceptions** and **Unchecked Exceptions**. There is also a special category for **Errors**, which are severe issues outside the programmer's control.

Checked Exceptions (Compile-Time Exceptions)

Checked exceptions are exceptions that are checked by the compiler at compile-time. If not handled properly (using try-catch) or declared (using throws), the program will not compile.

Characteristics:

- Represent recoverable conditions.
- Examples: Issues with I/O, database operations, etc.
- The programmer must explicitly handle or declare them.

Examples:

1. `IOException`: Occurs during input/output operations.
2. `SQLException`: Related to database operations.
3. `FileNotFoundException`: Thrown when trying to access a file that does not exist.

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

public class CheckedExceptionExample {
    public static void main(String[] args) {
        try {
            File file = new File("nonexistent.txt");
            FileReader reader = new FileReader(file);
        } catch (Exception e) {
            System.out.println("File not found: " + e.getMessage());
        }
    }
}
```

Unchecked Exceptions (Runtime Exceptions)

Unchecked exceptions are exceptions that occur at runtime and are not checked by the compiler. They usually result from programming errors, such as improper logic or invalid usage of data structures.

Characteristics:

- Represent issues that occur due to bugs in the code (e.g., logical errors).
- Examples: Division by zero, null references, invalid array access, etc.

Examples:

1. `ArithmeticException`: Thrown for illegal arithmetic operations, e.g., division by zero.
2. `NullPointerException`: Thrown when trying to access an object with a null reference.
3. `ArrayIndexOutOfBoundsException`: Thrown when accessing an invalid array index.

```
public class UncheckedExceptionExample {
    public static void main(String[] args) {
        int[] numbers = {1, 2, 3};
        try {
            System.out.println(numbers[5]); // Throws ArrayIndexOutOfBoundsException
        } catch (ArrayIndexOutOfBoundsException e) {
            System.out.println("Exception: " + e.getMessage());
        }
    }
}
```

Errors

Errors represent serious issues beyond the programmer's control, often related to the environment in which the program is running. They are not meant to be handled by the program.

Characteristics:

- Represent fatal conditions.
- Examples: System failures, resource exhaustion, etc.
- Generally, the program cannot recover from errors.

Examples:

1. **OutOfMemoryError**: Thrown when the JVM runs out of memory.
2. **StackOverflowError**: Thrown when the stack size is exceeded due to deep recursion.
3. **VirtualMachineError**: Thrown when the JVM fails to operate.

```
public class ErrorExample {
    public static void main(String[] args) {
        try {
            recursiveCall(); // Causes StackOverflowError
        } catch (StackOverflowError e) {
            System.out.println("Error: Stack overflow occurred.");
        }
    }

    public static void recursiveCall() {
        recursiveCall();
    }
}
```

Uncaught Exceptions in Java

An **uncaught exception** occurs when a program throws an exception but does not handle it using a try-catch block. This results in the program terminating abruptly and displaying an error message along with a stack trace.

Characteristics of Uncaught Exceptions

1. **Runtime Termination**: The program stops execution immediately when the exception occurs.
2. **Stack Trace**: Java prints the stack trace to show where the exception originated and its propagation through the call stack.

3. **Common Cause:** Usually results from a lack of proper exception handling or missing checks for risky operations.
4. **Examples:** Uncaught exceptions often involve runtime exceptions like `ArithmeticException`, `NullPointerException`, or `ArrayIndexOutOfBoundsException`.

```
public class UncaughtExceptionExample {  
    public static void main(String[] args) {  
        int a = 10;  
        int b = 0;  
        // Division by zero without handling  
        int result = a / b; // Throws ArithmeticException  
        System.out.println("Result: " + result); // This line is not executed  
    }  
}
```

```
C:\TestJava>javac UncaughtExceptionExample.java
```

```
C:\TestJava>java UncaughtExceptionExample  
Exception in thread "main" java.lang.ArithmeticException: / by zero  
    at UncaughtExceptionExample.main(UncaughtExceptionExample.java:6)
```

Propagation of Uncaught Exceptions

When an exception is not caught:

1. The JVM checks the call stack for a try-catch block.
2. If no try-catch block is found in the current method, the exception propagates to the calling method.
3. If no method in the stack handles the exception, the JVM terminates the program and prints the stack trace.

How to Prevent Uncaught Exceptions

1. **Use try-catch blocks:**
Handle exceptions at appropriate points to avoid abrupt termination.
2. **Validate Inputs:**
Ensure proper checks are in place to prevent exceptions.
3. **Log and Monitor Exceptions:**
Use logging frameworks to capture and analyze exceptions instead of terminating the program.

Using try and catch in Java

The try-catch mechanism in Java is used for exception handling, allowing developers to manage runtime errors gracefully. The try block contains the code that may throw an exception, and the catch block handles the exception if it occurs.

Structure of try and catch

1. **try block:**
 - Contains code that may generate exceptions.

- Only exceptions occurring within this block will be handled by the associated catch blocks.

2. catch block:

- Handles specific exceptions that occur in the try block.
- You can have multiple catch blocks for different exception types.

```
try {  
    // Code that may throw an exception  
} catch (ExceptionType1 e1) {  
    // Handle ExceptionType1  
} catch (ExceptionType2 e2) {  
    // Handle ExceptionType2  
}
```

Key Points About Using try and catch

- **Only One Exception is Caught:** When an exception occurs, only the first matching catch block is executed, and the rest are ignored.
- **Order of catch Blocks:** The catch blocks should be ordered from most specific to most general exceptions. Otherwise, the compiler will throw an error.
- **The Exception Object:** Each catch block receives an exception object (e.g., e) that contains information about the exception.
- **Optional finally Block:** The finally block, if present, executes after the try or catch block, whether an exception occurs or not.

Advantages of Using try and catch

1. **Prevents Program Crashes:** Ensures the program does not terminate abruptly.
2. **Error Handling and Recovery:** Allows the program to recover and continue execution.
3. **Separates Error Logic:** Cleanly separates error-handling code from the main logic.

Simple try-catch block

```
public class TryCatchExample {  
    public static void main(String[] args) {  
        try {  
            int result = 10 / 0; // This will throw ArithmeticException  
        } catch (ArithmeticException e) {  
            System.out.println("Exception caught: " + e.getMessage());  
        }  
        System.out.println("Program continues...");  
    }  
}
```

Output:

```
C:\TestJava>java TryCatchExample  
Exception caught: / by zero  
Program continues...
```

Multiple catch Blocks in Java

In Java, a try block can be followed by multiple catch blocks to handle different types of exceptions separately. This allows developers to provide specific exception-handling logic for various types of errors that may arise.

Key Points about Multiple catch Blocks

- **Handling Specific Exceptions:** Each catch block is designed to handle a particular type of exception.
- **Order Matters:**
 - The order of the catch blocks is important.
 - Specific exceptions must be caught before more general exceptions (e.g., `ArrayIndexOutOfBoundsException` before `Exception`).
 - If a general exception is placed first, it will handle all exceptions, and subsequent catch blocks will become unreachable, causing a compilation error.
- **One Exception Per Catch:** Only the first catch block that matches the thrown exception type will execute, even if multiple exceptions occur in the try block.

Example:

```
public class MultipleCatchExample {  
    public static void main(String[] args) {  
        try {  
            int[] numbers = {1, 2, 3};  
            System.out.println(numbers[5]); // Throws ArrayIndexOutOfBoundsException  
        } catch (ArithmeticException e) {  
            System.out.println("Arithmetic Exception caught: " + e.getMessage());  
        } catch (ArrayIndexOutOfBoundsException e) {  
            System.out.println("Array Index Out Of Bounds Exception caught: " + e.getMessage());  
        } catch (Exception e) {  
            System.out.println("General Exception caught: " + e.getMessage());  
        }  
    }  
}
```

Output:

```
C:\TestJava>javac MultipleCatchExample.java  
  
C:\TestJava>java MultipleCatchExample  
Array Index Out Of Bounds Exception caught: Index 5 out of bounds for length 3
```

Nested try Statements in Java

A **nested try statement** refers to placing one try block inside another try block. This can be useful when different blocks of code within the try block need different exception handling logic.

Key Points about Nested try Statements

1. **Multiple Layers of Exception Handling:**
You can have multiple levels of exception handling, where exceptions from different parts of the try block are handled at different levels.
2. **Inner catch Block:**
Each inner try block can have its own catch block to handle exceptions specific to that block. If an exception occurs in the inner try block, it will be caught by the corresponding catch block. If an exception occurs in the outer try block, it can be handled by the outer catch.

3. Control Flow:

The flow of control depends on where the exception is thrown. If an exception occurs in an inner try, it is handled by the inner catch. If not, the outer catch will handle exceptions from the outer try.

4. Finally Block:

The finally block, if used, will execute after the try-catch blocks, regardless of whether an exception occurred.

Example:

```
public class NestedTry {  
    public static void main(String[] args) {  
        try {  
            // Outer try block  
            int[] numbers = {1, 2, 3};  
            System.out.println(numbers[2]); // This will print 3  
  
            try {  
                // Inner try block  
                int result = 10 / 0; // This will throw ArithmeticException  
            } catch (ArithmeticException e) {  
                System.out.println("Inner catch: Division by zero occurred");  
            }  
  
            // Outer try continues after inner try  
            System.out.println("Outer try continues...");  
        } catch (ArrayIndexOutOfBoundsException e) {  
            System.out.println("Outer catch: Array index out of bounds");  
        }  
  
        System.out.println("Program continues...");  
    }  
}
```

Output:

```
C:\TestJava>javac NestedTry.java  
  
C:\TestJava>java NestedTry  
3  
Inner catch: Division by zero occurred  
Outer try continues...  
Program continues...
```

throw, throws, and finally in Java

In Java, **throw**, **throws**, and **finally** are important keywords used in exception handling. They help developers manage errors more effectively and ensure that resources are released properly. Let's explore each of them in detail:

throw Keyword

The throw keyword is used to explicitly throw an exception from a method or a block of code. When an exception is thrown, the normal flow of the program is disrupted, and the control is transferred to the nearest catch block that can handle the exception.

Key Points about throw:

- It is followed by an instance of Throwable (either an Exception or an Error).

- Can throw both checked and unchecked exceptions.
- Once the exception is thrown, the execution stops at that point, and control is transferred to the appropriate catch block.

throw new ExceptionType("Error message");

Example:

```
public class ExceptionEx {  
    public static void main(String[] args) {  
        try {  
            checkAge(15); // Will throw an exception  
        } catch (IllegalArgumentException e) {  
            System.out.println("Exception caught: " + e.getMessage());  
        }  
    }  
  
    public static void checkAge(int age) {  
        if (age < 18) {  
            throw new IllegalArgumentException("Age must be 18 or older.");  
        }  
        System.out.println("Age is valid");  
    }  
}
```

Output:

```
C:\TestJava>javac ExceptionEx.java  
  
C:\TestJava>java ExceptionEx  
Exception caught: Age must be 18 or older.
```

throws Keyword

The throws keyword is used in a method declaration to specify that a method may throw one or more exceptions. This is used for **checked exceptions** (exceptions that are subclasses of Exception but not RuntimeException). It tells the compiler and the developer that a method might throw certain exceptions, and the caller of the method needs to handle them.

Key Points about throws:

- It is placed after the method signature.
- It is used to declare exceptions that a method might throw.
- It can declare multiple exceptions, separated by commas.

public void methodName() throws ExceptionType1, ExceptionType2 { // method body }

Example:

```
import java.io.IOException;

class ExceptionEx {
    public static void main(String[] args) {
        try {
            readFile();
        } catch (IOException e) {
            System.out.println("IOException caught: " + e.getMessage());
        }
    }

    public static void readFile() throws IOException {
        throw new IOException("File not found");
    }
}
```

Output:

```
IOException caught: File not found
```

finally Keyword

The finally block is used to execute code that must run after the try-catch blocks, regardless of whether an exception was thrown or not. It is typically used for clean-up activities such as closing files, releasing resources, or resetting states.

Key Points about finally:

- The finally block always executes, whether or not an exception occurs.
- It is optional, but if present, it will execute after the try and catch blocks.
- It can be used for code that must always execute, such as closing resources or network connections.
- If a return statement is encountered in a try or catch block, the finally block will still execute before the method returns the value.

```
try {
    // Code that may throw an exception
} catch (ExceptionType e) {
    // Exception handling code
} finally {
    // Code to clean up, close resources, etc.
}
```

Example:

```

public class FinallyExample {
    public static void main(String[] args) {
        try {
            System.out.println("Inside try block");
            int result = 10 / 0; // This will throw ArithmeticException
        } catch (ArithmeticException e) {
            System.out.println("Exception caught: " + e.getMessage());
        } finally {
            System.out.println("This will always be executed, even if exception occurs");
        }
        System.out.println("Program continues...");
    }
}

```

Output:

```

C:\TestJava>javac FinallyExample.java

C:\TestJava>java FinallyExample
Inside try block
Exception caught: / by zero
This will always be executed, even if exception occurs
Program continues...

```

Key Differences

Aspect	throw	throws
Definition	Used to explicitly throw an exception.	Used to declare that a method might throw an exception.
Usage	Inside the body of a method or code block.	In the method signature (declaration).
Functionality	Actually throws an exception.	Declares the exceptions that a method may throw.
Syntax	throw new ExceptionType("Message");	public void methodName() throws ExceptionType;
When to Use	When you want to trigger an exception manually.	When a method might throw a checked exception and needs to be handled by the caller.
Handled by	Handled in a catch block or propagated further.	Handled by the caller (using try-catch or further throws).
Example	throw new IllegalArgumentException("Invalid input");	public void readFile() throws IOException { ... }

Java's Built-in Exceptions:

In Java, exceptions are used to handle situations that disrupt the normal flow of execution. Java provides a variety of **built-in exceptions** (also known as **predefined exceptions**), which are part of the Java class library and are subclasses of the Throwable class. These exceptions are categorized into two main types:

1. **Checked Exceptions** – Exceptions that must be explicitly handled by the programmer (either through a try-catch block or by declaring the exception using throws).
2. **Unchecked Exceptions** – Exceptions that are not required to be handled by the programmer. These are usually runtime exceptions, which typically occur due to programming errors.

1. ArithmeticException

- **Type:** Unchecked Exception (RuntimeException)
- **Description:** Thrown when an exceptional arithmetic condition occurs, such as division by zero.

2. NullPointerException

- **Type:** Unchecked Exception (RuntimeException)
- **Description:** Thrown when an application attempts to use null where an object is required, such as calling a method on a null object reference.

3. ArrayIndexOutOfBoundsException

- **Type:** Unchecked Exception (RuntimeException)
- **Description:** Thrown when trying to access an array index that is out of bounds (i.e., index is less than 0 or greater than or equal to the length of the array).

4. ClassNotFoundException

- **Type:** Checked Exception
- **Description:** Thrown when an application tries to load a class using Class.forName() or other similar methods, but the class is not found.

5. IOException

- **Type:** Checked Exception
- **Description:** A general class for exceptions produced by failed or interrupted I/O operations, such as reading from a file or writing to a network socket.

6. FileNotFoundException

- **Type:** Checked Exception (Subclass of IOException)
- **Description:** Thrown when an attempt to open the file denoted by a specified pathname has failed.

7. NumberFormatException

- **Type:** Unchecked Exception (RuntimeException)
- **Description:** Thrown when an attempt to convert a string to a number fails, such as when a string is not a valid representation of a number.

8. StackOverflowError

- **Type:** Error (Not Exception)
- **Description:** A subclass of Error, this is thrown when the stack overflows, typically due to deep recursion or an infinite recursive method call.

Creating Your Own Exception Subclasses in Java

In Java, you can define your own custom exceptions by subclassing the existing exception classes, typically `Exception` or `RuntimeException`. Custom exceptions allow you to provide more meaningful error messages or handle specific error conditions that are relevant to your application.

Steps to Create a Custom Exception:

1. **Subclass the `Exception` or `RuntimeException` class.**
 - If you want to create a **checked exception**, extend `Exception`.
 - If you want to create an **unchecked exception**, extend `RuntimeException`.
2. **Provide constructors for your custom exception.**
 - You typically define at least two constructors: one for the default message and one for a more detailed message or for passing a `Throwable` cause.
3. Optionally, override methods such as **`getMessage()`** or **`toString()`** for more customized behavior.

Refer Lab Program 9

Chained Exceptions in Java

Chained exceptions, also known as **exception chaining**, refer to the practice of linking one exception to another. This technique is useful when an exception occurs, and you want to provide additional context by attaching the original exception (the cause) to a new one. It allows you to trace the root cause of an error, providing more useful information for debugging and logging.

Java provides built-in support for exception chaining, enabling developers to preserve the cause of an exception even when it is re-thrown in a different form.

Why Use Chained Exceptions?

Chained exceptions are especially useful in scenarios where:

1. **Wrapping Low-Level Exceptions:** When a low-level exception (such as `IOException`) is thrown, you might want to throw a higher-level, domain-specific exception (such as `DataProcessingException`) while preserving the original cause of the error.
2. **Error Context Preservation:** When catching one exception and throwing another, you can maintain the original exception so that the root cause is not lost.
3. **Improved Debugging:** Chained exceptions provide a full stack trace of the exception chain, helping you understand how the error propagated.

Creating Chained Exceptions

Java provides two primary mechanisms for chaining exceptions:

1. **Using `Throwable` constructors:** You can pass the original exception (cause) to the new exception when creating it.

Syntax: `new Exception("New exception message", originalException);`

2. **Using `initCause()` method:** If you've already created an exception, you can later associate a cause with it using this method. *Syntax: `exception.initCause(cause);`*

Example:

```

public class ChainedExceptionExample {
    public static void main(String[] args) {
        try {
            methodA();
        } catch (CustomException e) {
            System.out.println("Caught custom exception: " + e.getMessage());
            System.out.println("Cause: " + e.getCause());
        }
    }

    public static void methodA() throws CustomException {
        try {
            methodB();
        } catch (ArithmeticException e) {
            throw new CustomException("Custom exception occurred while calling methodB", e);
        }
    }

    public static void methodB() {
        int result = 10 / 0; // This will throw ArithmeticException
    }
}

// Custom Exception Class
class CustomException extends Exception {
    public CustomException(String message, Throwable cause) {
        super(message, cause);
    }
}

```

Output:

```

C:\TestJava>javac ChainedExceptionDemo.java

C:\TestJava>java ChainedExceptionDemo
Caught custom exception: Custom exception occurred while calling methodB
Cause: java.lang.ArithmeticException: / by zero

```

Using initCause method Example:

```

public class ChainedExceptionDemo {
    public static void main(String[] args) {
        try {
            methodA();
        } catch (CustomException e) {
            System.out.println("Caught custom exception: " + e.getMessage());
            System.out.println("Cause: " + e.getCause());
        }
    }

    public static void methodA() throws CustomException {
        try {
            methodB();
        } catch (ArithmeticException e) {
            CustomException customException =
                new CustomException("Custom exception occurred while calling methodB");
            customException.initCause(e); // Assigning the cause
            throw customException;
        }
    }

    public static void methodB() {
        int result = 10 / 0; // This will throw ArithmeticException
    }
}

// Custom Exception Class
class CustomException extends Exception {
    public CustomException(String message) {
        super(message);
    }
}

```

The Java Thread Model

The Java Thread Model provides a robust framework for building multithreaded applications that allow multiple tasks to run concurrently. It is based on **shared memory** and **preemptive multitasking**, where the operating system or JVM thread scheduler manages thread execution.

Key Features of the Java Thread Model

1. **Multithreading:** Allows concurrent execution of multiple threads within the same program to maximize CPU utilization and enhance application performance.
2. **Lightweight Threads:** Threads in Java are lightweight compared to processes as they share the same memory space but have independent execution paths.
3. **Thread Priorities:** Each thread has a priority that helps the scheduler determine the execution order.
4. **Synchronization:** Ensures that multiple threads can work together without interfering with shared resources, avoiding race conditions and deadlocks.
5. **Portability:** The thread model is implemented by the JVM, making it platform-independent.

Components of the Java Thread Model

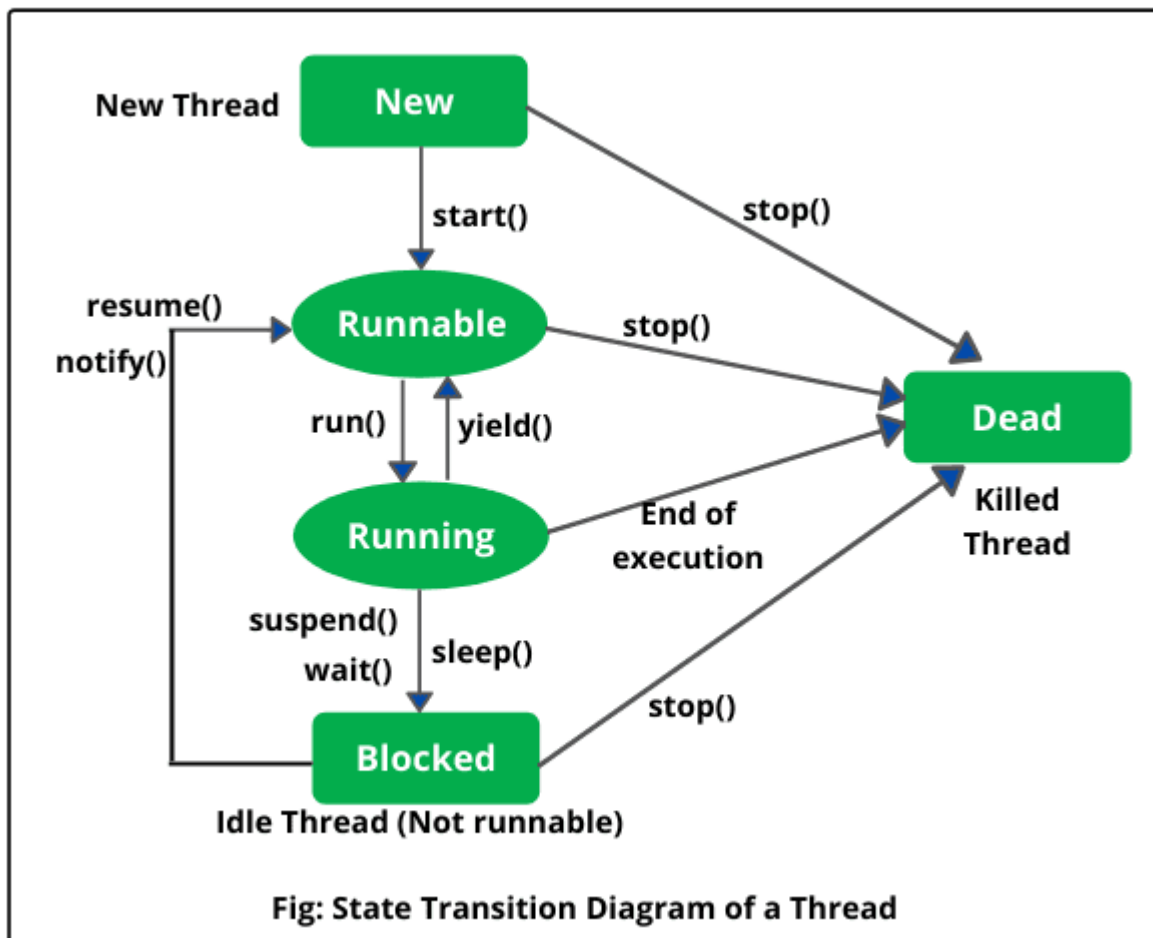
1. **Thread:** A thread represents an independent path of execution in a program.
2. **Thread Class and Runnable Interface:**
 - Java provides the Thread class and Runnable interface to create and manage threads.
3. **Thread States:**
 - Threads move through different states: NEW, RUNNABLE, BLOCKED, WAITING, TIMED_WAITING, and TERMINATED.
4. **Thread Scheduler:**
 - The JVM's thread scheduler determines which thread gets CPU time. The scheduling algorithm may vary between JVM implementations.
5. **Thread Synchronization:**
 - Java provides synchronized blocks and methods to coordinate thread access to shared resources.

Thread Lifecycle in Java

The lifecycle of a thread in Java consists of several states:

1. **New:** A thread is created but not yet started.
2. **Runnable:** The thread is ready to run but waiting for the CPU.
3. **Running:** The thread is executing its task.

4. **Blocked/Waiting:** The thread is paused, waiting for a resource or signal.
5. **Terminated:** The thread has completed its execution or was stopped.



Advantages of the Java Thread Model

1. **Concurrency:** Multiple tasks can execute concurrently, improving application responsiveness.
2. **Efficient Resource Utilization:** Threads share the same memory space, reducing memory overhead compared to processes.
3. **Scalability:** Multithreading allows applications to scale efficiently with multi-core processors.
4. **Simplified Design:** Java's thread model simplifies the development of concurrent applications through built-in classes and synchronization features.

Creating a Thread

Java provides two primary ways to create and manage threads:

1. **Extending the Thread Class**
2. **Implementing the Runnable Interface**

Extending the Thread Class:

Creating a Thread by Extending the Thread Class

- The Thread class provides built-in functionality to create and manage threads.
- To create a thread:
 - Extend the Thread class.
 - Override its run() method.
 - Call the start() method to begin execution.

Example:

```
class MyThread extends Thread {  
    public void run() {  
        System.out.println("Thread is running: " + Thread.currentThread().getName());  
    }  
}  
  
public class MyThread1 {  
    public static void main(String[] args) {  
        MyThread t1 = new MyThread(); // Create a thread  
        t1.start(); // Start the thread  
    }  
}
```

Output:

```
C:\TestJava>javac MyThread1.java  
  
C:\TestJava>java MyThread1  
Thread is running: Thread-0
```

Creating a Thread by Implementing the Runnable Interface

- Implement the Runnable interface when your class needs to perform multiple tasks and cannot extend Thread due to inheritance constraints.
- Define the run() method in your class and pass an instance of it to a Thread object.

Example:

```
class MyRunnable implements Runnable {  
    public void run() {  
        System.out.println("Thread is running: " + Thread.currentThread().getName());  
    }  
}  
  
public class MyThread1 {  
    public static void main(String[] args) {  
        MyRunnable myRunnable = new MyRunnable();  
        Thread t1 = new Thread(myRunnable); // Create a thread  
        t1.start(); // Start the thread  
    }  
}
```

Output:

```
C:\TestJava>javac MyThread1.java  
  
C:\TestJava>java MyThread1  
Thread is running: Thread-0
```

Differences Between Thread and Runnable

Feature	Extending Thread	Implementing Runnable
Inheritance	Cannot extend another class.	Can extend another class.
Code Reusability	Less reusable.	More reusable.
Memory Efficiency	Slightly less efficient.	More memory-efficient.

Creating Multiple Threads

In Java, multiple threads can run concurrently, performing different tasks simultaneously. Each thread executes independently and has its own execution path.

Steps to Create Multiple Threads

1. Define a thread class by extending Thread or implementing Runnable.
2. Create multiple instances of the thread class.
3. Start each thread using the start() method.

Example: Creating Multiple Threads Using Thread Class

```
class MyThread extends Thread {
    private String name;

    MyThread(String name) {
        this.name = name;
    }

    public void run() {
        for (int i = 1; i <= 3; i++) {
            System.out.println(name + " is running: " + i);
            try {
                Thread.sleep(500); // Pause for 500ms
            } catch (InterruptedException e) {
                System.out.println("Thread interrupted");
            }
        }
    }
}

public class MultiThread {
    public static void main(String[] args) {
        MyThread t1 = new MyThread("Thread 1");
        MyThread t2 = new MyThread("Thread 2");

        t1.start(); // Start the first thread
        t2.start(); // Start the second thread
    }
}
```

Output: (Order may vary due to concurrent execution)

```
C:\TestJava>javac MultiThread.java

C:\TestJava>java MultiThread
Thread 1 is running: 1
Thread 2 is running: 1
Thread 1 is running: 2
Thread 2 is running: 2
Thread 1 is running: 3
Thread 2 is running: 3
```

Example: Creating Multiple Threads Using Runnable Interface

```
class MyRunnable implements Runnable {
    private String name;

    MyRunnable(String name) {
        this.name = name;
    }

    public void run() {
        for (int i = 1; i <= 3; i++) {
            System.out.println(name + " is running: " + i);
            try {
                Thread.sleep(500); // Pause for 500ms
            } catch (InterruptedException e) {
                System.out.println("Thread interrupted");
            }
        }
    }
}

public class MultiThread {
    public static void main(String[] args) {
        Thread t1 = new Thread(new MyRunnable("Thread 1"));
        Thread t2 = new Thread(new MyRunnable("Thread 2"));

        t1.start(); // Start the first thread
        t2.start(); // Start the second thread
    }
}
```

Output:

```
C:\TestJava>javac MultiThread.java

C:\TestJava>java MultiThread
Thread 2 is running: 1
Thread 1 is running: 1
Thread 1 is running: 2
Thread 2 is running: 2
Thread 2 is running: 3
Thread 1 is running: 3
```

Key Points About Multiple Threads

1. **Concurrency:** Threads execute independently and concurrently.
2. **Synchronization:** Proper synchronization must be ensured when threads share resources to avoid issues like race conditions.
3. **Thread Safety:** Using synchronized blocks or methods ensures safe access to shared resources.

Using `isAlive()` and `join()` in Java

Java provides the `isAlive()` and `join()` methods in the `Thread` class to manage thread execution and monitor thread states. These methods help coordinate multiple threads effectively.

1. `isAlive()`

Method Purpose

The `isAlive()` method is used to check whether a thread is still running or has finished its execution.

Key Points

- Returns **true** if the thread has started and has not yet terminated.
- Returns **false** if the thread is in the `NEW` state or has completed execution (`TERMINATED` state).

Syntax: `boolean isAlive();`

2. `join()`

Method Purpose

The `join()` method ensures that the calling thread waits for the specified thread to complete its execution before continuing.

Key Points

- Allows one thread to wait for another thread to finish.
- Helps in coordinating thread execution.
- There are three overloaded versions:
 - `void join()`: Waits indefinitely until the thread finishes.
 - `void join(long millis)`: Waits for the thread to finish or until the specified time has elapsed.
 - `void join(long millis, int nanos)`: Waits for the thread to finish or until the specified time (in milliseconds and nanoseconds) has elapsed.

Syntax: `void join()` throws `InterruptedException`

Key Differences Between `isAlive()` and `join()`

Aspect	<code>isAlive()</code>	<code>join()</code>
Purpose	Checks if a thread is still running.	Waits for a thread to complete.
Return Value	Returns a boolean (true or false).	Returns nothing (void).

Aspect	isAlive()	join()
Usage	Monitoring thread state.	Synchronizing thread execution.
Blocking	Non-blocking.	Blocking until thread finishes or timeout.
Example Use Case	Periodically check a thread's status.	Ensure thread completion before proceeding.

Example:

```

class MyThread extends Thread {
    private String threadName;

    MyThread(String threadName) {
        this.threadName = threadName;
    }

    public void run() {
        for (int i = 1; i <= 3; i++) {
            System.out.println(threadName + " is running: Step " + i);
            try {
                Thread.sleep(1000); // Simulate work by sleeping for 1 second
            } catch (InterruptedException e) {
                System.out.println(threadName + " interrupted.");
            }
        }
        System.out.println(threadName + " has finished.");
    }
}

public class AliveJoin {
    public static void main(String[] args) {
        MyThread t1 = new MyThread("Thread 1");
        MyThread t2 = new MyThread("Thread 2");

        t1.start(); // Start the first thread
        t2.start(); // Start the second thread

        // Use isAlive() to check the status of the threads
        while (t1.isAlive() || t2.isAlive()) {
            System.out.println("Checking threads...");
            System.out.println("Thread 1 alive: " + t1.isAlive());
            System.out.println("Thread 2 alive: " + t2.isAlive());
            try {
                Thread.sleep(500); // Pause to avoid flooding the output
            } catch (InterruptedException e) {
                System.out.println("Main thread interrupted.");
            }
        }
    }
}

```

```
}

System.out.println("Waiting for threads to complete using join()...");
try {
    t1.join(); // Ensure Thread 1 has completed
    t2.join(); // Ensure Thread 2 has completed
} catch (InterruptedException e) {
    System.out.println("Main thread interrupted while waiting for join.");
}

System.out.println("All threads have finished. Exiting main thread.");
}
```

Output:

```
C:\TestJava>javac AliveJoin.java

C:\TestJava>java AliveJoin
Checking threads...
Thread 1 alive: true
Thread 2 alive: true
Thread 1 is running: Step 1
Thread 2 is running: Step 1
Checking threads...
Thread 1 alive: true
Thread 2 alive: true
Thread 1 is running: Step 2
Thread 2 is running: Step 2
Checking threads...
Thread 1 alive: true
Thread 2 alive: true
Checking threads...
Thread 1 alive: true
Thread 2 alive: true
Thread 1 is running: Step 3
Thread 2 is running: Step 3
Checking threads...
Thread 1 alive: true
Thread 2 alive: true
Checking threads...
Thread 1 alive: true
Thread 2 alive: true
Checking threads...
Thread 1 alive: true
Thread 2 alive: true
Thread 1 has finished.
Thread 2 has finished.
Waiting for threads to complete using join()...
All threads have finished. Exiting main thread.
```

Thread Priorities in Java

Thread priorities in Java determine the relative importance of threads when they compete for CPU time. Every thread has a priority value, which helps the thread scheduler decide the order in which threads should be executed. However, thread priorities do not guarantee execution order; they are just hints to the thread scheduler.

Key Concepts

1. Priority Range

- Thread priorities are integers ranging from **MIN_PRIORITY** (1) to **MAX_PRIORITY** (10).
- The default priority is **NORM_PRIORITY** (5).

2. Setting and Getting Priorities

- You can set a thread's priority using the `setPriority()` method.
- Use the `getPriority()` method to retrieve a thread's priority.

3. Priority Constants

- `Thread.MIN_PRIORITY = 1`
- `Thread.NORM_PRIORITY = 5`
- `Thread.MAX_PRIORITY = 10`

4. Purpose of Priorities

- Threads with higher priorities are more likely to be selected for execution by the thread scheduler, but this depends on the JVM implementation and operating system.
- Lower-priority threads may still execute if no higher-priority threads are runnable.

Thread Priority Methods

- **Set a Thread's Priority:** `public final void setPriority(int newPriority)`
- **Get a Thread's Priority:** `public final int getPriority()`

Example:

```

class MyThread extends Thread {
    MyThread(String name) {
        super(name); // Set thread name
    }

    public void run() {
        System.out.println(getName() + " with priority " + getPriority() + " is running.");
    }
}

public class ThreadPriority {
    public static void main(String[] args) {
        // Create three threads with different priorities
        MyThread t1 = new MyThread("Thread-1");
        MyThread t2 = new MyThread("Thread-2");
        MyThread t3 = new MyThread("Thread-3");

        // Set priorities
        t1.setPriority(Thread.MIN_PRIORITY);
        t2.setPriority(Thread.NORM_PRIORITY);
        t3.setPriority(Thread.MAX_PRIORITY);

        // Start threads
        t1.start();
        t2.start();
        t3.start();
    }
}

```

Output:

```
C:\TestJava>javac ThreadPriority.java

C:\TestJava>java ThreadPriority
Thread-2 with priority 5 is running.
Thread-1 with priority 1 is running.
Thread-3 with priority 10 is running.
```

Important Notes**1. Platform-Dependent Behavior:**

- Thread priority handling depends on the JVM and OS. On some systems, priorities might have no effect.

2. Starvation Risk:

- Threads with low priority might not get enough CPU time if high-priority threads dominate.

3. Use Priorities Carefully:

- Setting thread priorities should be done judiciously to avoid unpredictable behavior.

Synchronization in Java

Definition: Synchronization in Java is a mechanism to control access to shared resources in a multithreaded environment. It ensures that only one thread can access a critical section of code at a time, preventing data inconsistency and thread interference.

Why Synchronization is Needed

- 1. Shared Resources:** In multithreaded applications, threads often share resources like variables, objects, or files.
- 2. Thread Interference:** Without synchronization, multiple threads might access and modify shared data simultaneously, leading to incorrect results.
- 3. Consistency:** Synchronization helps maintain consistency and prevents race conditions.

Types of Synchronization**1. Synchronized Methods**

- Synchronizes the entire method.
- The thread must acquire the lock on the object (or class for static methods) before executing the method.

```
public synchronized void methodName() {  
    // critical section  
}
```

Example:

```
class Counter {
    private int count = 0;

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

    public int getCount() {
        return count;
    }
}

public class SynchronizedMethodExample {
    public static void main(String[] args) throws InterruptedException {
        Counter counter = new Counter();

        Thread t1 = new Thread(() -> {
            for (int i = 0; i < 1000; i++) counter.increment();
        });

        Thread t2 = new Thread(() -> {
            for (int i = 0; i < 1000; i++) counter.increment();
        });

        t1.start();
        t2.start();

        t1.join();
        t2.join();

        System.out.println("Final Count: " + counter.getCount());
    }
}
```

Output:

```
C:\TestJava>javac SynchronizedMethodExample.java
C:\TestJava>java SynchronizedMethodExample
Final Count: 2000
```

2. Synchronized Blocks

- Synchronizes only a specific block of code, allowing finer-grained control.
- Requires a lock on a specific object.

Syntax:

```
synchronized(object)
{
    //
    critical section
}
```

Example:

```

class Counter {
    private int count = 0;

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

    public int getCount() {
        return count;
    }
}

public class SynchronizedBlockExample {
    public static void main(String[] args) throws InterruptedException {
        Counter counter = new Counter();

        Thread t1 = new Thread(() -> {
            for (int i = 0; i < 1000; i++) counter.increment();
        });

        Thread t2 = new Thread(() -> {
            for (int i = 0; i < 1000; i++) counter.increment();
        });

        t1.start();
        t2.start();

        t1.join();
        t2.join();

        System.out.println("Final Count: " + counter.getCount());
    }
}

```

Output:

```

C:\TestJava>javac SynchronizedBlockExample.java
C:\TestJava>java SynchronizedBlockExample
Final Count: 2000

```

3. Static Synchronization

- Synchronizes static methods.
- Locks on the class object, not the instance.

Syntax:

public static synchronized void methodName() { // critical section }

Example:

```
class SharedResource {  
    public static synchronized void printMessage(String message) {  
        System.out.print("[");  
        System.out.print(message);  
        System.out.println("]");  
    }  
}  
  
public class StaticSynchronizationExample {  
    public static void main(String[] args) {  
        Thread t1 = new Thread(() -> SharedResource.printMessage("Hello"));  
        Thread t2 = new Thread(() -> SharedResource.printMessage("World"));  
  
        t1.start();  
        t2.start();  
    }  
}
```

Output:

```
C:\TestJava>javac StaticSynchronizationExample.java  
  
C:\TestJava>java StaticSynchronizationExample  
[Hello]  
[World]
```

Inter-Thread Communication

Inter-thread communication is a mechanism to allow threads to communicate with each other efficiently in a synchronized environment. Java provides methods like `wait()`, `notify()`, and `notifyAll()` to achieve this.

Methods for Communication

- **wait()**: Makes the current thread wait until another thread invokes `notify()` or `notifyAll()` on the same object.
- **notify()**: Wakes up a single thread waiting on the object's monitor.
- **notifyAll()**: Wakes up all threads waiting on the object's monitor.

```
class SharedResource {  
    private int data;  
    private boolean available = false;  
  
    public synchronized void produce(int value) {  
        while (!available) {  
            try {  
                wait(); // Wait until the consumer consumes the data  
            } catch (InterruptedException e) {  
                e.printStackTrace();  
            }  
        }  
        data = value;  
        available = true;  
    }  
}
```

```

        System.out.println("Produced: " +
            value); notify(); // Notify the
            consumer
        }

        public synchronized int
        consume() { while
        (!available) {
            try {
                wait(); // Wait until the producer produces data
            } catch (InterruptedException
                e) { e.printStackTrace();
            }
        }
        available = false;
        System.out.println("Consumed: "
            + data); notify(); // Notify the
            producer
        return data;
    }
}

public class
    InterThreadCommunicati
on { public static void
    main(String[] args) {
        SharedResource resource = new SharedResource();

        Thread producer = new
        Thread(() -> { for (int
            i = 1; i <= 5; i++) {
                resource.produce(i);
            }
        });

        Thread consumer = new
        Thread(() -> { for (int
            i = 1; i <= 5; i++) {
                resource.consume();
            }
        });
    }
}

```

Output:

```

C:\TestJava>javac InterThreadCommunication.java
C:\TestJava>java InterThreadCommunication
Produced: 1
Consumed: 1
Produced: 2
Consumed: 2
Produced: 3
Consumed: 3
Produced: 4
Consumed: 4
Produced: 5
Consumed: 5

```

Suspending, Resuming, and Stopping Threads

These actions allow for manual control of thread execution, but their use is deprecated in favor of modern thread management techniques (like using flags or wait/notify).

a. Suspending a Thread

- **Method:** `suspend()` (deprecated)
- Pauses a thread's execution until it is resumed.

b. Resuming a Thread

- **Method:** `resume()` (deprecated)
- Resumes a thread that has been suspended.

c. Stopping a Thread

- **Method:** `stop()` (deprecated)
- Forcefully terminates a thread's execution.

Modern Alternative: Use flags and thread-safe mechanisms.

Example:

```
class ControlledThread extends Thread {  
    private volatile boolean suspended = false;  
  
    public void suspendThread() {  
        suspended = true;  
    }  
  
    public void resumeThread() {  
        suspended = false;  
        synchronized (this) {  
            notify();  
        }  
    }  
  
    public void run() {  
        for (int i = 1; i <= 5; i++) {  
            synchronized (this) {  
                while (suspended) {  
                    try {  
                        wait(); // Pause execution  
                    } catch (InterruptedException e) {  
                        e.printStackTrace();  
                    }  
                }  
            }  
            System.out.println("Thread running: Step " + i);  
            try {  
                Thread.sleep(1000); // Simulate work  
            } catch (InterruptedException e) {
```

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

public class SuspendResumeExample {
    public static void main(String[] args) throws InterruptedException {
        ControlledThread thread = new ControlledThread();
        thread.start();

        Thread.sleep(2000);
        thread.suspendThread(); // Suspend the thread
        System.out.println("Thread suspended.");

        Thread.sleep(2000);
        thread.resumeThread(); // Resume the thread
        System.out.println("Thread resumed.");
    }
}
```

Output:

```
C:\TestJava>javac SuspendResumeExample.java

C:\TestJava>java SuspendResumeExample
Thread running: Step 1
Thread running: Step 2
Thread suspended.
Thread running: Step 3
Thread resumed.
Thread running: Step 4
Thread running: Step 5
```

Obtaining a Thread's State

Java provides methods to monitor the state of threads during their lifecycle.

Thread Lifecycle States

1. **NEW:** The thread is created but not yet started.
2. **RUNNABLE:** The thread is ready to run and waiting for CPU time.
3. **BLOCKED:** The thread is waiting for a monitor lock.
4. **WAITING:** The thread is waiting indefinitely for another thread to perform an action.
5. **TIMED_WAITING:** The thread is waiting for a specific period.
6. **TERMINATED:** The thread has finished execution.

Methods to Get Thread State

- **getState()**: Returns the current state of the thread.

Example:

```
public class ThreadStateExample {  
    public static void main(String[] args) throws InterruptedException {  
        Thread thread = new Thread(() -> {  
            System.out.println("Thread is running...");  
            try {  
                Thread.sleep(1000); // Simulate work  
            } catch (InterruptedException e) {  
                e.printStackTrace();  
            }  
            System.out.println("Thread finished.");  
        });  
  
        System.out.println("State: " + thread.getState()); // NEW  
        thread.start();  
  
        while (thread.isAlive()) {  
            System.out.println("State: " + thread.getState()); // RUNNABLE or TIMED_WAITING  
            Thread.sleep(100); // Monitor periodically  
        }  
  
        System.out.println("State: " + thread.getState()); // TERMINATED  
    }  
}
```

Output:

```
C:\TestJava>javac ThreadStateExample.java  
  
C:\TestJava>java ThreadStateExample  
State: NEW  
State: RUNNABLE  
Thread is running...  
State: TIMED_WAITING  
State: TIMED_WAITING  
State: TIMED_WAITING  
State: TIMED_WAITING  
State: TIMED_WAITING  
State: TIMED_WAITING  
State: TIMED_WAITING  
State: TIMED_WAITING  
State: TIMED_WAITING  
Thread finished.  
State: TERMINATED
```