UNIT-5

EXCEPTION HANDLING AND MULTITHREADED PROGRAMMING

5.1TYPES OF ERROR

There Are Three Types Of Errors:

1.Compile-time Errors:

- Compile-time errors, also known as compilation errors or syntax errors, occur during the compilation phase of the program.
- These errors are caused by violations of the Java syntax rules or incorrect usage of language constructs.
- Examples of compile-time errors include missing semicolons, undefined variables, incompatible types, or using a reserved keyword as an identifier.
- When a compile-time error occurs, the compiler generates an error message, and the program cannot be compiled until the errors are fixed.

2.Runtime Errors:

- Runtime errors, also called exceptions, occur during the execution of the program.
- These errors are caused by exceptional conditions or unexpected situations that arise while the program is running.
- Common examples of runtime errors include division by zero (**ArithmeticException**), accessing an array element with an invalid index (**ArrayIndexOutOfBoundsException**), or attempting to open a non-existent file (**FileNotFoundException**).
- When a runtime error occurs, an exception object is thrown, which can be caught and handled using exception handling mechanisms like try-catch blocks. If unhandled, the program terminates abruptly.

3.Logical Errors:

- Logical errors, also known as semantic errors or bugs, occur when the program's logic or algorithm is incorrect.
- These errors do not cause the program to crash or generate error messages but result in undesired or incorrect behavior.
- Logical errors can be challenging to identify as they do not produce any compile-time or runtime errors. The program runs successfully, but the output or behavior is not as expected.
- Detecting and fixing logical errors typically involves careful code review, debugging, and testing**.**

It is important to note that compile-time errors are caught by the compiler during the compilation process, runtime errors occur during program execution and can be caught and handled using exception handling, while logical errors require careful analysis and debugging to identify and fix.

Understanding the different types of errors in Java is crucial for developing robust and error-free software. Proper testing, code review, and debugging techniques are essential to identify and resolve these errors effectively.

5.2 BASIC CONCEPTS OF EXCEPTION HANDLING

Exception handling in Java is a mechanism that allows you to handle and recover from exceptional situations or errors that occur during the execution of a program. It helps in making the program more robust by providing a structured approach to deal with unexpected or exceptional scenarios.

Exception

- An exception is an event that occurs during the execution of a program and disrupts the normal flow of the program.
- Exceptions represent various types of errors, such as runtime errors, input/output errors, arithmetic errors, and others.
- In Java, exceptions are represented by classes that are derived from the Throwable class or one of its subclasses.

5.3 TRY AND CATCH BLOCK

The try-catch block is used to handle exceptions and provide a mechanism to gracefully handle exceptional situations during the execution of a program. The try-catch block consists of a try block and one or more catch blocks.

```
try {
  // Code that may throw an exception
} catch (ExceptionType1 ex1) {
 // Exception handling code for ExceptionType1
} catch (ExceptionType2 ex2) {
// Exception handling code for ExceptionType2
} catch (Exception ex) {
  // Generic exception handling code for any other exception
}
```

The code that may throw an exception is placed within the try block.

If an exception occurs within the try block, the control immediately transfers to the corresponding catch block based on the type of exception.

Each catch block specifies the type of exception it can handle. If the caught exception matches the type specified in a catch block, that catch block is executed.

If no catch block is found that matches the type of the thrown exception, the exception propagates to the next level of the method call stack or terminates the program if it reaches the main method without being caught.

After executing the catch block, the program continues with the code that follows the try-catch block.

EXAMPLE:

```
try {
  int result = divide(10, 0); // This method may throw an ArithmeticException
```

```
    System.out.println("Result: " + result);
} catch (ArithmeticException ex) {
    System.out.println("Exception caught: " + ex.getMessage());
}

// ...

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

In this example, the divide method is called within the try block, and it may throw an ArithmeticException if the divisor is zero. The catch block following the try block catches the ArithmeticException and executes the exception handling code.

You can have multiple catch blocks to handle different types of exceptions. The catch blocks are evaluated sequentially, and the first catch block that matches the thrown exception type is executed.

It's important to handle exceptions appropriately in the catch block, which may include logging the error, displaying an error message to the user, or taking corrective actions.

By using try-catch blocks, you can handle exceptions and prevent them from causing the program to crash. It provides a mechanism for error handling, allowing your program to gracefully recover from exceptional situations and continue its execution.

5.4 THROW AND THROWS

throw and **throws** are two keywords used in exception handling to deal with exceptions and specify exception propagation.

1.throw Keyword:

- The throw keyword is used to explicitly throw an exception from within a method or a block of code.
- It is followed by an instance of an exception class or a subclass of Throwable.
- When a throw statement is encountered, the normal flow of the program is disrupted, and the specified exception is thrown.
- It allows you to generate and throw exceptions explicitly to handle exceptional scenarios.

```
public void validateAge(int age) {
    if (age < 0) {
    throw new IllegalArgumentException("Age cannot be negative.");
```

```
    }
}
```

In this example, the **validateAge** method throws an **IllegalArgumentException** if the given age is negative. The throw keyword is used to throw the exception, and a custom error message is provided.

2.throws Keyword:

- The throws keyword is used in a method declaration to indicate that the method may throw one or more types of exceptions.
- It specifies that the method is not handling the exception itself but is instead passing the responsibility to the calling method or the JVM.
- Multiple exceptions can be declared using a comma-separated list.

```
public void readFile() throws IOException, FileNotFoundException {
    // Code that may throw IOException or FileNotFoundException
}
```

In this example, the **readFile** method declares that it may throw **IOException** or **FileNotFoundException**. The responsibility of handling these exceptions is passed to the caller of the **readFile** method.

When a method throws an exception using the throws keyword, it must be either caught and handled using a try-catch block by the calling method, or the calling method should also declare the exception using the throws keyword in its own declaration.

It's important to note that throw is used to raise an exception explicitly within a method, whereas **throws** is used to indicate that a method may potentially **throw** an exception, allowing the caller to handle it or propagate it further.

Both **throw** and throws keywords play a crucial role in handling and propagating exceptions in Java programs, ensuring that exceptional scenarios are appropriately handled and managed.

5.5 USER DEFINE EXCEPTION

In Java, you can define your own exceptions by creating a subclass of the **Exception** class or one of its existing subclasses. This allows you to create custom exceptions that represent specific exceptional conditions in your application.

1.Create A Subclass Of The Exception Class Or One Of Its Existing Subclasses:

You can extend the Exception class directly or choose a more specific subclass such as **RuntimeException** or **IOException**, depending on the nature of the exception you want to create.

To create a custom checked exception, extend the Exception class.

To create a custom unchecked exception, extend the **RuntimeException** class.

Example of a custom exception by extending Exception class:

```
public class MyException extends Exception {
```

```
 // Constructor(s) and additional methods can be defined
}
```

Example of a custom exception by extending Exception class subclass:

```
public class MyException extends RuntimeException {
 // Constructor(s) and additional methods can be defined
}
```

2.Customize the exception class:

You can provide constructors to initialize the exception object with a specific error message or any additional information.

You can add additional methods or properties to the exception class to provide more functionality or context.

Example:

```
public class InvalidInputException extends RuntimeException {
private String input;


public InvalidInputException(String message, String input) {
  super(message);
this.input = input;
  }


   public String getInput() {
   return input;
  }
}
```

In this example, a custom unchecked exception called **InvalidInputException** is defined. It extends the **RuntimeException** class and includes an additional property input. It also provides a constructor to initialize the exception object with an error message and the invalid input value.

Once you have defined a custom exception, you can use it in your code by throwing instances of your exception class when necessary. For example:

```
public void processInput(String input) {
   if (input == null) {
   throw new InvalidInputException("Input cannot be null", input);
}
// Process the input
}
```

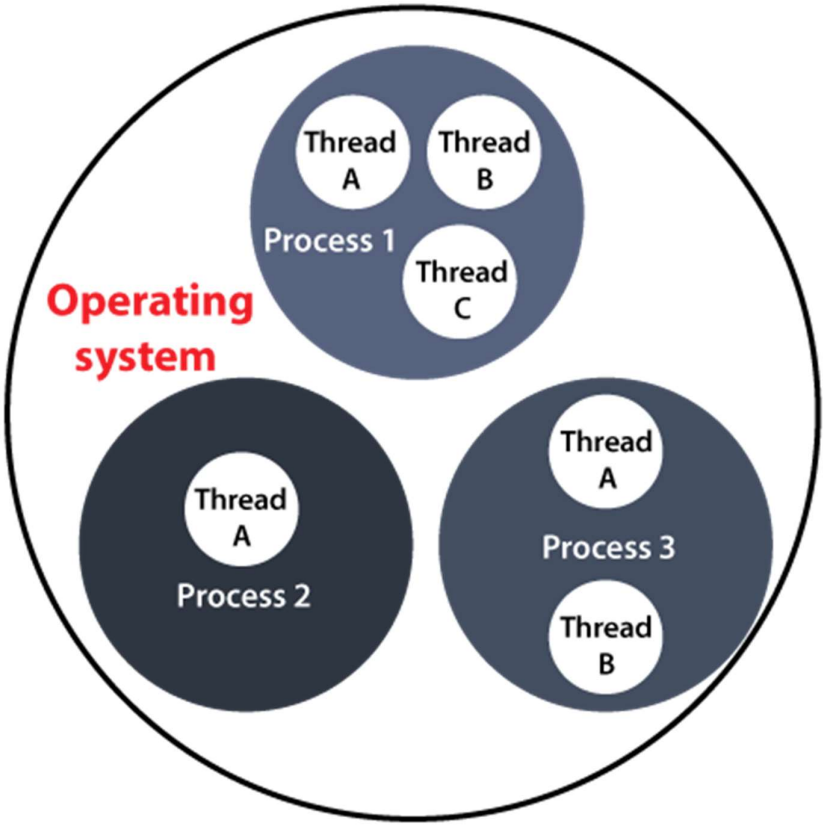
In this example, the **processInput** method throws an instance of the **InvalidInputException** when the input is **null**.

By creating user-defined exceptions, you can represent specific exceptional conditions in your application and provide meaningful error messages or additional information to aid in debugging and error handling. It allows you to design a more robust and expressive exception hierarchy tailored to your application's needs.

5.6 INTRODUCTION OF THREAD

A Thread is a very light-weighted process, or we can say the smallest part of the process that allows a program to operate more efficiently by running multiple tasks simultaneously.

In order to perform complicated tasks in the background, we used the Thread concept in Java. All the tasks are executed without affecting the main program. In a program or process, all the threads have their own separate path for execution, so each thread of a process is independent.



5.7 IMPLEMENTATION OF THREAD

In Java, there are two primary ways to implement threads: by extending the **Thread** class or by implementing the **Runnable** interface. Let's look at both approaches:

1.Extending the Thread class:

In this approach, you create a new class that extends the **Thread** class and override its **run**() method, which represents the code that will be executed in the new thread.

You can then create an instance of your custom thread class and start the thread by calling the **start**() method.

EXAMPLE:

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

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

2.Implementing the Runnable interface:

In this approach, you create a class that implements the **Runnable** interface and implement the **run**() method defined by the interface.

You then create an instance of your custom class and pass it to a **Thread** object's constructor.

Finally, you call the **start**() method on the **Thread** object to start the new thread.

EXAMPLE:

```
class MyRunnable implements Runnable {
   @Override
   public void run() {
    // Code to be executed in the new thread
  System.out.println("Thread is running");
 }
}

public class Main {
  public static void main(String[] args) {
   MyRunnable runnable = new MyRunnable();
   Thread thread = new Thread(runnable);
```

```
thread.start(); // Starts the execution of the new thread
} }
```

Both approaches create a new thread of execution, and the run() method is called when the thread starts. You can perform any desired operations within the run() method. The start() method initiates the execution of the thread, which will run concurrently with the main thread.

It's important to note that creating and managing threads requires careful consideration of thread synchronization and coordination to avoid potential issues such as race conditions or data inconsistencies. You should use synchronization mechanisms like synchronized blocks or locks when accessing shared resources to ensure thread safety.

Additionally, Java provides other features and utilities for thread management, such as thread synchronization, interruption, thread pools, and concurrent data structures, which can be explored based on specific requirements.

Remember to handle any exceptions thrown within the **run**() method appropriately, as uncaught exceptions in a thread can cause the program to terminate unexpectedly.

5.8 THREAD LIFE CYCLE AND METHOD

The life cycle of a thread in Java refers to its various states and transitions that occur during its execution. A thread goes through several stages from its creation to termination.

The Java thread life cycle consists of the following states:

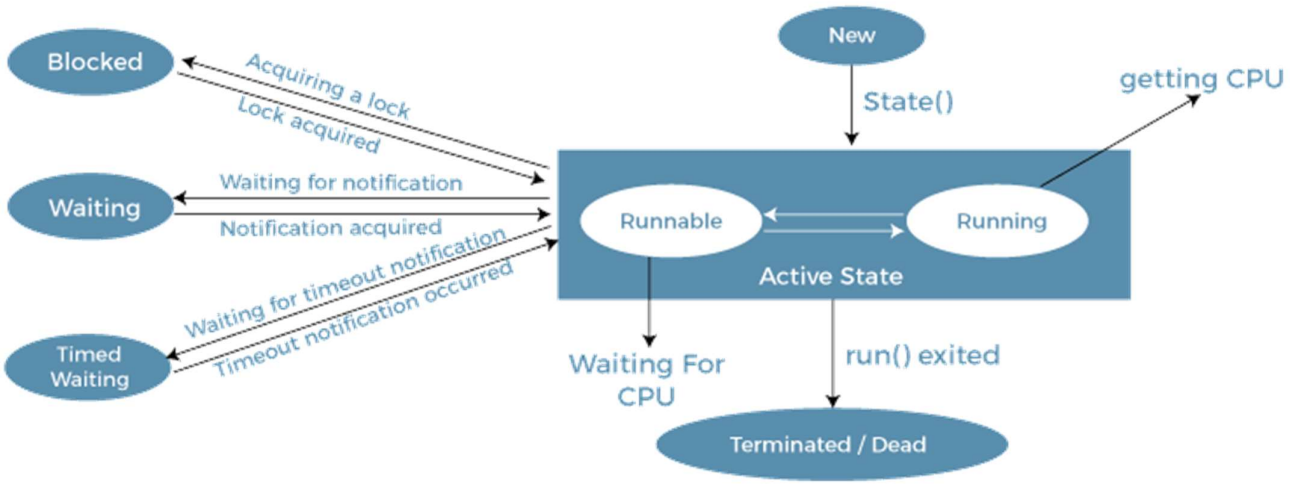
New

Active

Blocked / Waiting

Timed Waiting

Terminated



Life Cycle of a Thread

1.New:

The thread is in the new state when it has been created but has not yet started.

In this state, the thread has been instantiated but has not yet invoked the **start**() method.

2.Runnable:

When the **start**() method is called, the thread enters the runnable state.

In this state, the thread is eligible for execution, but it may or may not be currently executing.

The thread scheduler selects a thread from the runnable pool and executes it.

3.Running:

A thread enters the running state when it starts executing its **run**() method.

In this state, the actual execution of the thread's code is taking place.

The thread remains in this state until it is paused, interrupted, or its execution completes.

4.Blocked/Waiting:

A thread can transition to the blocked or waiting state for various reasons:

Blocked: If the thread is waiting for a monitor lock to enter a synchronized block or method and another thread holds the lock.

Waiting: If the thread is waiting indefinitely for another thread to perform a particular action, such as waiting for a condition to be satisfied or waiting for a **notify**/**notifyAll** signal.

5.Timed Waiting:

Similar to the blocked/waiting state, a thread can also enter a timed waiting state, where it waits for a specific period of time.

This state is reached when the thread calls methods such as **Thread.sleep(), Object.wait()**, or **Thread.join()** with a specified timeout.

6.Terminated:

A thread enters the terminated state when its **run**() method completes execution or when it is terminated prematurely.

Once in the terminated state, the thread cannot be resumed or restarted.

It's important to note that the transitions between these states are managed by the Java Virtual Machine (JVM) and the thread scheduler. The scheduler determines which thread to execute based on factors like thread priorities, fairness, and the scheduling algorithm used by the JVM.

Understanding the life cycle of a thread is essential for proper thread management and synchronization. It helps in designing concurrent applications and handling synchronization issues effectively.

Additionally, Java provides methods and utilities to manage and control threads throughout their life cycle, such as **Thread.sleep()**, **Thread.yield(),Thread.interrupt(),** and synchronization mechanisms like locks, conditions, and barriers.

5.9 MULTITHREADING

Multithreading in Java refers to the concurrent execution of multiple threads within a single program. It allows different threads to run independently, performing tasks simultaneously and improving overall application performance and responsiveness. Java provides built-in features and APIs to support multithreading, making it easier to develop concurrent applications.

1.Thread Creation:

Java provides two primary ways to create threads: by extending the Thread class or by implementing the Runnable interface (as discussed earlier).

The Thread class and Runnable interface provide methods and hooks for thread initialization, execution, and termination.

2.Thread Synchronization:

When multiple threads access shared resources simultaneously, synchronization is required to avoid data inconsistencies and race conditions.

Java provides synchronization mechanisms like synchronized blocks, synchronized methods, and explicit locks (e.g., Lock interface and **ReentrantLock** class) to ensure thread safety and proper synchronization between threads.

3.Thread Scheduling and Priorities:

Java's thread scheduler determines the order and timing of thread execution.

Thread priorities (ranging from 1 to 10) can be assigned to threads to influence their relative execution order.

However, thread priorities do not guarantee precise ordering, and the actual behavior may vary across JVM implementations and platforms.

4.Thread Coordination:

Threads can be coordinated to work together or communicate using synchronization mechanisms like **wait**(), **notify**(), and **notifyAll**().

These methods are used to pause, resume, or wake up threads based on specific conditions, allowing threads to communicate and synchronize their actions.

5.Thread Safety:

Thread safety ensures that shared resources are accessed in a safe and consistent manner by multiple threads.

Proper synchronization, using synchronization blocks or locks, is essential to achieve thread safety and prevent race conditions and data corruption.

6.Executors and Thread Pools:

The **Executor** framework in Java provides higher-level abstractions for managing and controlling threads.

Thread pools can be created using the **ExecutorService** interface, allowing for efficient management and reuse of threads.

7.Concurrent Data Structures:

Java provides concurrent data structures, such as **ConcurrentHashMap** and **ConcurrentLinkedQueue**, which are designed for safe access and manipulation by multiple threads concurrently.

These data structures handle synchronization internally, making them suitable for multithreaded environments.

Multithreading is particularly useful in scenarios where tasks can be divided into smaller units of work that can run independently or when tasks involve blocking operations like I/O or network requests. By utilizing multiple threads, you can maximize resource utilization and enhance the responsiveness and performance of your Java applications.

However, it's crucial to handle synchronization and coordination between threads carefully to avoid synchronization issues, deadlocks, and other concurrency-related problems. Proper design, synchronization mechanisms, and thread safety practices are necessary to ensure the correct and efficient execution of multithreaded programs.