Unit-04

Exception Handling and Multithreading

4.1 The Exception Hierarchy

The exception hierarchy in Java is a well-defined structure that categorizes different types of exceptions. All exception classes in Java are subclasses of the java.lang.Throwable class. The Throwable class has two main subclasses: Error and Exception.

Throwable:

The superclass for all errors and exceptions in Java. Only objects of this class (or its subclasses) can be thrown or caught.

1. **Error:**
   Represents serious problems that a reasonable application should not try to catch. Errors are typically related to the environment in which the application is running. For example:
   - OutOfMemoryError
   - StackOverflowError
   - VirtualMachineError

2. **Exception:**
   Represents conditions that a reasonable application might want to catch. Exceptions are further divided into two main categories:

   a) **Checked Exceptions**:
   Exceptions that are checked at compile-time. These must be either caught or declared in the method signature using the throws keyword. For example:
      - IOException
      - SQLException
      - ClassNotFoundException

   b) **Unchecked Exceptions (RuntimeException)**:
   Exceptions that are not checked at compile-time. These are subclasses of RuntimeException and do not need to be explicitly caught or declared in the method signature. For example:
      - NullPointerException
      - ArrayIndexOutOfBoundsException

- ArithmeticException

4.2 Exception handling fundamentals

Exception handling in Java provides a structured mechanism to handle runtime errors, ensuring smooth program execution.

1) **Try Block**: Encloses code that might throw an exception.
   → Used to test a block of code for errors.
   → Contains statements that might throw an exception.
   → If an exception occurs, it is thrown to the catch block.
2) **Catch Block**: Handles the exception thrown by the try block.
   → Contains code that is executed if an exception occurs.
   → Can catch and handle specific exception types.
   → Multiple catch blocks can be used to handle different types of exceptions.
3) **Finally Block**: Executes code that needs to run regardless of an exception being thrown or not.
   → Typically used for cleanup operations like closing files or releasing resources.
   → Executes after try and catch blocks, regardless of whether an exception was thrown or caught.
4) **Throw Keyword**: Used to explicitly throw an exception.
   → Used for throwing custom exceptions or specific conditions.
   → Helps in managing errors in a controlled manner.
5) **Throws Keyword**: Declares exceptions that a method might throw.
   → Used in method signatures to specify which exceptions can be thrown.
   → Forces the caller of the method to handle or declare the exception.

4.3 Throwing, Re-throwing and Catching Exceptions

1. Throwing Exceptions: Use the `throw` keyword to explicitly throw an exception.

→ Can throw both checked and unchecked exceptions.
→ Often used to enforce a particular business rule or constraint.

```
public class ThrowExample {
    public static void validateAge(int age) {
        if (age < 18) {
            throw new IllegalArgumentException("Age must be 18 or older."); } } }
```

2. Re-throwing Exceptions: Catch an exception and then throw it again.

→ Allows for logging or processing before re-throwing.
→ Can change the type of exception if necessary.

```
public class ReThrowExample {
    public static void method () throws Exception {
        try {
            // Some code that throws an exception
        } catch (Exception e) {
            // Perform some processing if needed
            throw e; // Re-throwing the exception
         }  } }
```

3. **Catching Exceptions**: Use the `catch` block to handle exceptions.

→ Catch blocks can specify different exception types.
→ Allows for custom error handling and recovery.

```
public class CatchExample {
    public static void main(String[] args) {
        try {
            int result = 10 / 0;
        } catch (ArithmeticException e) {
            System.out.println("Caught an ArithmeticException: " + e.getMessage());
        } } }
```

4.4 try, catch, throw, throws, and finally keywords

try: Defines a block of code to be tested for exceptions.

→ Contains code that might throw an exception.
→ Followed by one or more catch blocks or a finally block.

```
try {
   // Code that may throw an exception
}
```

catch: Defines a block of code to handle exceptions.

→ Can catch specific exception types.
→ Allows for custom error handling.

```
catch (ExceptionType e) {
   // Code to handle the exception
}
```

throw: Used to explicitly throw an exception.

→ Throws a new exception.
→ Typically used to enforce specific constraints or rules.

```
throw new ExceptionType("Error message");
```

throws: Declares that a method might throw exceptions.

→ Used in method signatures.
→ Forces the caller to handle or declare the exception.

```
public void method() throws ExceptionType {
   // Method code
}
```

finally: Defines a block of code that will always execute, regardless of an exception.

→ Used for cleanup operations.
→ Ensures that code executes after try and catch blocks.

```
finally {
    // Code that will always execute
}
```

4.5 Multithreading fundamentals

Multithreading in Java allows concurrent execution of two or more threads. Threads are lightweight processes that share the same address space.

Thread: The smallest unit of a process.

→ Each thread runs in parallel and has its own call stack.
→ Java provides built-in support for multithreaded programming.

Multithreading Benefits:

→ Better CPU utilization.
→ Improved application performance.
→ Simplified modeling of real-world problems.

Multithreading Challenges:

→ Synchronization issues.
→ Deadlocks.
→ Race conditions.

4.6 Thread class and Runnable Interface

Thread Class

Directly creates a thread by extending the `Thread` class. This approach is straightforward but comes with certain limitations and specific characteristics: Extending the Thread Class, Creating an Instance, Starting the Thread.

→ Provides methods to start, run, and control the thread.

→ Requires overriding the `run` method.

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

   public static void main(String[] args) {
      MyThread thread = new MyThread();
      thread.start();
   } }
```

Methods to Control the Thread:

- start(): Initiates the execution of the thread.
- sleep(long millis): Pauses the thread for a specified duration.
- join(): Waits for the thread to terminate.
- interrupt(): Interrupts the thread's execution.
- setPriority(int newPriority): Sets the thread's priority.
- setName(String name): Assigns a name to the thread for identification.

Runnable Interface

Creates a thread by implementing the `Runnable` interface and passing an instance to a `Thread` object. This approach is more flexible and preferred in many scenarios like Implementing the Runnable Interface, Creating an Instance, Passing the Runnable to a Thread, Starting the Thread

→ More flexible as it allows extending another class.
→ Requires implementing the `run` method.

```
public class MyRunnable implements Runnable {
   public void run() {
        System.out.println("Thread is running.");   }
    public static void main(String[] args) {
        MyRunnable myRunnable = new MyRunnable();
        Thread thread = new Thread(myRunnable);
        thread.start();
}  }
```

Thread Control Methods:

- start(): Initiates the execution of the thread.
- sleep(long millis): Pauses the thread for a specified duration.
- join(): Waits for the thread to terminate.
- interrupt(): Interrupts the thread's execution.
- setPriority(int newPriority): Sets the thread's priority.
- setName(String name): Assigns a name to the thread for identification

Differences Between `Thread` Class and `Runnable` Interface

Thread Class:

→ Extending the `Thread` class means your class cannot extend any other class.
→ More suitable when you want to override other methods of the `Thread` class.
→ Direct access to thread control methods.

Runnable Interface:

→ Implementing `Runnable` allows you to extend another class.
→ More flexible and suitable for task sharing.
→ Promotes separation of task and thread management.