Class 13: Java Exception Handling

Exception handling is an important concept in Java that allows you to gracefully handle runtime errors, also known as exceptions, in your java code. It ensures that your program doesn't crash when an unexpected error occurs and provides a mechanism to recover from these errors. Let's dive into the details of exception handling in Java with examples.

1. **What are Exceptions?**

   In Java, exceptions are objects that represent unexpected events or errors that can occur during program execution. These events can be caused by various reasons,

   such as invalid user input, file not found, or division by zero.

   Exceptions are divided into two main categories:

   - checked exceptions (compile-time exceptions) and
   - unchecked exceptions (runtime exceptions).

The try-catch Block:

To handle exceptions, you use a **try-catch** block. The code that might throw an exception goes inside the **try** block, and you catch and handle the exception in the **catch** block.

Java code

```
try { //

Code that may throw an exception

} catch (ExceptionType e) {

// Handle the exception

}
```

The finally Block:

You can also use a **finally** block to execute code that must be run whether an exception occurs or not. For example, you can use it to close resources like files or database connections.

javaCopy code

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

catch (ExceptionType e) {

// Handle the exception
```

```
 }
 finally { // Code that always runs }
```

2. Checked Exceptions: Checked exceptions are exceptions that are checked at compile time. This means the compiler ensures that you handle these exceptions either by catching them or declaring that your method throws them. Common examples include **IOException**, **FileNotFoundException**, and **SQLException**.

Java code

```
import java.io.BufferedReader;
 import java.io.FileReader;
import java.io.IOException;
public class CheckedExceptionExample {
 public static void main(String[] args) {
try {
BufferedReader reader = new BufferedReader(new FileReader("file.txt"));
String line = reader.readLine();
System.out.println(line);
reader.close();
}


 catch (IOException e)
 {
 System.out.println("An error occurred: " + e.getMessage());
 } } }
```

3. Unchecked Exceptions (Runtime Exceptions): Unchecked exceptions are exceptions that are not checked at compile time, and you are not required to catch or declare them. Common examples include **NullPointerException**, **ArithmeticException**, and **ArrayIndexOutOfBoundsException**.

java code

```
public class UncheckedExceptionExample {
 public static void main(String[] args) {
int[] numbers = { 1, 2, 3 };
int result = numbers[5]; // ArrayIndexOutOfBoundsException
System.out.println(result);
 }
}
```

4. Throwing Exceptions: You can throw exceptions manually using the **throw** keyword. This is useful for creating custom exceptions or rethrowing exceptions in your code.

Java code

```
public class CustomExceptionExample {
public static void main(String[] args) {
try {
throw new CustomException("This is a custom exception.");
}
catch (CustomException e) {
System.out.println("Caught custom exception: " + e.getMessage());
}
 }
 }
class CustomException extends Exception {
public CustomException(String message) {
 super(message);
}
}
```

5. Handling Multiple Exceptions: You can catch multiple exceptions by using multiple **catch** blocks or a multi-catch block.

java code

```
try { //
Code that may throw an exception
}
catch (ExceptionType1 e1) {
// Handle the first type of exception
```

```
 }
catch (ExceptionType2 e2){
 // Handle the second type of exception
 }
catch (Exception e) {
// Handle any other exceptions
}
```

6. Exception Propagation: When an exception is thrown, it can propagate up the call stack until it's caught and handled. You can use the **throws** clause in a method signature to declare that a method may throw a specific exception.

java code

```
public void myMethod() throws CustomException {
// Code that may throw CustomException
}
```

7. Common Exception Handling Strategies:

- Graceful Degradation: Provide an alternative way for your program to continue running even if an exception occurs.
- Fallback Mechanisms: Use default values or alternative methods when an operation fails.
- Retry Mechanisms: Retry an operation if it fails, up to a certain number of attempts.

18. Best Practices:

- Always handle exceptions; don't leave empty **catch** blocks.
- Log exceptions to aid in debugging and monitoring.
- Be specific about the exceptions you catch; don't catch generic **Exception** types unless necessary.
- Close resources like files, streams, and database connections in **finally** blocks.

Exception handling is crucial for writing robust and reliable Java applications. It helps you identify and recover from errors, ensuring your programs work smoothly even in the face of unexpected issues.