Chapter 12: Exception Handling

Introduction

In the world of programming, errors are inevitable. Programs often deal with unpredictable inputs, system-level interruptions, or logical faults. Without proper handling, such errors can cause the program to crash or behave abnormally. **Exception Handling** provides a structured mechanism to detect, handle, and recover from runtime errors in a clean and manageable way.

Exception handling is crucial for building **robust, reliable, and maintainable software**. Modern programming languages like **Java, C++, Python, and C#** offer built-in support for exception handling. In this chapter, we will focus primarily on Java-style exception handling, as it is the standard for many object-oriented and enterprise-level applications.

12.1 What is an Exception?

An **exception** is an event that occurs during the execution of a program that disrupts the normal flow of instructions.

Types of Errors

- **Compile-time Errors:** Syntax or semantic errors (e.g., missing semicolon, undeclared variables).
- **Runtime Errors:** Errors that occur during execution (e.g., divide by zero, file not found).
- **Logical Errors:** Flaws in the algorithm or logic (e.g., incorrect output due to wrong formula).

Exception vs. Error

- **Exception**: Recoverable condition (e.g., FileNotFoundException).
- **Error**: Serious issues (e.g., StackOverflowError, OutOfMemoryError) which are usually not handled in the code.

12.2 Exception Hierarchy in Java

```
java.lang.Throwable
├── Error
│   └── e.g., OutOfMemoryError, StackOverflowError
└── Exception
    ├── Checked Exceptions
    │   └── e.g., IOException, SQLException
    └── Unchecked Exceptions (RuntimeException)
        └── e.g., NullPointerException, ArithmeticException
```

12.3 Need for Exception Handling

- Prevents abrupt termination.
- Increases code readability and maintainability.
- Allows centralized error management.
- Separates normal logic from error-handling logic.

12.4 Exception Handling Keywords

try

- Used to define a block of code to be tested for errors.

catch

- Handles the exception thrown by the `try` block.

finally

- A block that is always executed, regardless of exception occurrence.

throw

- Used to explicitly throw an exception.

throws

- Declares exceptions that a method may throw.

12.5 Basic Syntax

```
try {
    // Code that may throw an exception
} catch (ExceptionType name) {
    // Code to handle the exception
} finally {
    // Optional block that always executes
}
```

12.6 Checked vs Unchecked Exceptions

Checked Exceptions

- Checked at compile time.
- Must be either caught or declared using `throws`.
- Examples: IOException, SQLException.

Unchecked Exceptions

- Checked at runtime.
- Programmer's responsibility to avoid these.
- Examples: NullPointerException, ArithmeticException.

12.7 Multiple Catch Blocks

```
try {
    // Risky code
} catch (IOException e) {
    // Handle IO
} catch (ArithmeticException e) {
    // Handle Arithmetic
} catch (Exception e) {
    // Handle all other exceptions
}
```

- Catch from most specific to most general.
- Only one catch block executes per exception.

12.8 Nested try Blocks

You can have `try` blocks inside `try` blocks for localized exception handling.

```
try {
    try {
        // Nested risky code
    } catch (Exception e) {
        // Inner exception handling
    }
} catch (Exception e) {
    // Outer exception handling
}
```

12.9 finally Block

- Executes regardless of exception being thrown or not.
- Commonly used for cleanup activities (e.g., closing files, releasing resources).

```
try {
    // code
} catch (Exception e) {
    // handler
} finally {
```

```
    // always executes
}
```

12.10 throw Keyword

Used to manually throw an exception.

```
throw new ArithmeticException("Division by zero");
```

12.11 throws Keyword

Used in method declaration to propagate exceptions.

```
public void readFile() throws IOException {
    // code that may throw IOException
}
```

12.12 Custom Exceptions

You can define your own exception classes by extending the `Exception` class.

```
class AgeTooLowException extends Exception {
    public AgeTooLowException(String message) {
        super(message);
    }
}
```

Usage:

```
if (age < 18) {
    throw new AgeTooLowException("Age must be 18+");
}
```

12.13 Best Practices in Exception Handling

- Catch specific exceptions, not generic ones.
- Don’t suppress exceptions silently.
- Use `finally` for resource cleanup.
- Avoid using exceptions for normal control flow.
- Log exception details (stack trace).
- Create meaningful custom exceptions.

12.14 Exception Propagation

If an exception is not caught in the current method, it propagates to the calling method.

```
void methodA() {
    methodB();
}
void methodB() throws IOException {
    // throws exception
}
```

12.15 Common Exceptions in Java

Exception Name	Type	Description
ArithmeticException	Unchecked	Dividing by zero
NullPointerException	Unchecked	Object reference is null
ArrayIndexOutOfBounds	Unchecked	Invalid index access in an array
NumberFormatException	Unchecked	Invalid number conversion
FileNotFoundException	Checked	File doesn't exist
IOException	Checked	General IO failure

12.16 Exception Handling in Other Languages (Brief Comparison)

Language	Exception Handling
Java	try-catch-finally, checked & unchecked
C++	try-catch-throw, no checked exceptions
Python	try-except-finally, all are runtime exceptions
C#	Similar to Java, no checked exceptions

Summary

Exception handling is an essential concept in advanced programming that allows developers to gracefully manage runtime errors. By using constructs like `try`, `catch`, `finally`, `throw`, and `throws`, developers can write robust and fault-tolerant code. Proper use of exception handling improves code readability, maintainability, and user experience. Remember, exceptions should be treated as **exceptions**, not as a substitute for logic control.
