Lec 29 Exception Handling:

Exception handling in Java is a way to manage unexpected events or errors that happen while a program is running, preventing the program from crashing and allowing it to respond gracefully. Java provides several tools, like try-catch blocks, to detect and handle these errors, making programs more reliable and user-friendly.

What is an Exception?

An exception is an unexpected event or error that occurs during the execution of a program, such as entering the wrong type of input or dividing by zero. Exceptions can cause a program to stop running abnormally if not handled.

When Do Exceptions Occur?

Exceptions can occur at compile time (when the code is being checked before running) or at run time (while the program is running). Both types can interrupt the normal flow of a program.

Example: Exception from User Input

If a program expects an integer input but receives a different type (like a string), it throws an input mismatch exception, causing the program to stop unless the error is handled.

Purpose of Exception Handling

Exception handling is a mechanism to deal with run time errors so the program can continue running or end gracefully, instead of crashing. It helps maintain control flow even when errors occur.

Types of Errors: Error vs. Exception

Not all errors can be handled by the program. Some errors, like memory overflow, are unmanageable, while exceptions (such as input mismatch or arithmetic errors) can be caught and managed.

Types of Exceptions: Checked vs. Unchecked

Checked exceptions are detected at compile time and must be handled by the programmer, while unchecked exceptions occur at run time and may or may not be handled. Examples of unchecked exceptions include arithmetic and input mismatch exceptions.

Ways to Handle Exceptions

Java provides three main ways to handle exceptions: using try-catch blocks, try-finally blocks, and the throws keyword. The try-catch block is the most common and effective method.

How Try-Catch Blocks Work

Code that may throw an exception is placed inside a try block. If an exception occurs, control jumps to the catch block, which handles the specific error. This prevents the program from crashing and allows custom error messages.

Handling Multiple Exceptions

Multiple exceptions can be handled by using several catch blocks or by combining exception types in a single catch block using the pipe (|) operator. Each catch block can handle different types of exceptions with specific messages.

Exception Hierarchy in Java

All exceptions in Java are organized in a hierarchy. The root is the Throwable class, which has two main subclasses: Exception (for exceptions that can be handled) and Error (for serious problems that usually cannot be handled). Under Exception, there are further subclasses like RuntimeException, which includes common unchecked exceptions.

Catching Parent and Child Exceptions

If you don’t know the exact type of exception, you can catch a parent class (like Exception or RuntimeException), which will handle all its child exceptions. However, specific exceptions should be caught first, followed by more general ones to avoid unreachable code.

Other Exception Handling Techniques

Besides basic try-catch, Java allows the use of try-finally blocks, nested try-catch blocks, and more advanced patterns for complex scenarios. These provide flexibility in managing different types of errors and cleanup operations.

Exception handling in Java helps manage unexpected errors during program execution, preventing crashes and allowing the program to respond gracefully. Key tools include try-catch blocks, finally blocks for cleanup, and the throws keyword to delegate exception handling. Choosing the right approach depends on where errors might occur and how much control the user needs over error responses.

What is an Exception?

An exception is an unexpected event or error that occurs during program execution, often due to invalid input or faulty operations, causing the normal flow of the program to stop.

Types of Exceptions: Checked and Unchecked

Checked exceptions are detected at compile time and must be handled, while unchecked exceptions occur at runtime and may not require handling. Examples include arithmetic exceptions and input mismatch exceptions.

Handling Exceptions: try, catch, finally, throws

The try-catch block is used to catch and handle exceptions, while the finally block ensures certain code

runs regardless of an exception. The throws keyword passes responsibility for handling exceptions to the method caller.

Rules for try, catch, and finally Blocks

A try block cannot stand alone and must be followed by either a catch or finally block. Catch blocks handle specific exceptions, while finally always executes for cleanup, even if a return statement occurs.

Practical Example: Handling User Input

Using try-catch, user input is collected and summed. If an input error occurs (like entering text instead of a number), the catch block handles it and returns a default value (e.g., zero), ensuring the program doesn't crash.

The Purpose and Use of the finally Block

The finally block is used to execute code that must run after try and catch, such as closing resources or printing messages, regardless of whether an exception occurred. This ensures cleanup always happens.

finally for Resource Management

The finally block is especially important for releasing resources like database connections or files, preventing resource leaks that could slow down or crash applications.

Valid Combinations: try-catch-finally and try-finally

Java allows try-catch-finally and try-finally structures, but not catch or finally alone. try-finally is useful when you want cleanup code to always run, even if you don't need to catch exceptions.

Introduction to the throws Keyword

The throws keyword is used in method signatures to indicate that the method might throw a checked exception, passing responsibility for handling it to the caller. This is often used for compile-time (checked) exceptions.

Example: Checked Exception with Class.forName

Using Class.forName can throw a ClassNotFoundException at compile time. Java forces

developers to handle such exceptions with try-catch or declare them with throws to prevent runtime errors.

Compile-Time vs. Runtime Exception Handling

Compile-time exceptions (checked) require handling before the code runs, as they are likely to occur in sensitive operations like database access or file handling. This acts as a precaution, reducing the risk of runtime failures.

Delegating Exception Handling with throws

By using throws, a method can pass exception handling responsibility to its caller, allowing the user of the method to decide how to handle errors. If not handled, the exception is passed up to the JVM, which may terminate the program.

Combining try-catch and throws

If a method uses both try-catch and throws, exceptions handled inside try-catch won't be propagated, but any unhandled exceptions will still be passed on using throws.

API Design: Handling vs. Delegating Exceptions

When designing APIs, developers can choose to handle exceptions internally (simplifying usage for the client) or delegate handling to the client using throws (giving more control over error responses). Each approach has trade-offs in flexibility and code length.

Keywords: throw, throws, and Throwable

The throw keyword is used to create and throw custom exceptions, throws is used in method signatures to declare possible exceptions, and Throwable is the root class for all exceptions and errors in Java.

Conclusion and Next Steps

Exception handling in Java is essential for robust programs. Understanding when and how to use try-catch, finally, throws, and custom exceptions helps in writing safer, more maintainable code.

Exception handling in Java is a way to manage errors in programs, allowing you to handle both standard and custom error situations so your program doesn’t crash

unexpectedly. You can create your own exceptions for specific needs in your application and use special keywords like throw and throws to manage these errors.

What Is Exception Handling?

Exception handling is the process of dealing with errors or unexpected events in a program, preventing it from crashing and allowing for graceful error management.

Ways to Handle Exceptions

Java provides multiple ways to handle exceptions, including try-catch blocks (for catching errors), multiple catch blocks (for different error types), and the finally block (for code that must always run).

Checked vs Unchecked Exceptions

Checked exceptions are errors checked at compile time and must be handled or declared, while unchecked exceptions occur at runtime and are often due to programming errors.

The `throw` Keyword and Custom Exceptions

The throw keyword is used to create and trigger your own exceptions, known as custom exceptions, which are helpful when standard exceptions do not fit your specific application needs.

Application-Specific Exception Examples

Sometimes, what counts as an error depends on your application's rules, like enforcing a minimum age in a gym app, even if Java itself doesn't consider it an error.

Creating and Throwing Custom Exceptions

To create a custom exception, define a new class that extends either Exception (for checked exceptions) or RuntimeException (for unchecked exceptions), then use throw to trigger it when needed.

Throwing Built-in and Custom Exceptions

The throw keyword can also trigger built-in exceptions, like ArithmeticException, or your own custom ones, depending on the situation in your code.

Compile-Time vs Runtime Exceptions

Compile-time (checked) exceptions must be handled or declared, while runtime (unchecked) exceptions may not need explicit handling, but both can be thrown using the throw keyword.

How to Create Your Own Exception Class

Decide if your exception should be checked or unchecked by extending Exception (checked) or RuntimeException (unchecked), and ensure your class is part of the exception hierarchy.

Adding Messages to Custom Exceptions

Custom exception classes can accept error messages by defining a constructor that takes a string and passing this message to the superclass using super(message).

Difference Between `throw`, `throws`, and `Throwable`

throw is for explicitly causing an exception, throws is for declaring that a method might cause an exception, and Throwable is the root class for all exceptions and errors in Java.

The `finally` Keyword and Cleanup

The finally block runs code after try-catch, used for cleanup actions like releasing resources, and is related to garbage collection.

Summary and Next Steps

Reviewing exception handling concepts prepares you for more advanced topics like multi-threading, where understanding interfaces and classes is important.