CSE 1325 - Object-Oriented Programming

Exception Handling

Alex Dillhoff

University of Texas at Arlington

Exception Handling

What should a program do in the event of a crash?

Exception Handling

What should a program do in the event of a crash?

- Save the user state
- Gracefully exit (if possible)
- Warn the user (logging)

Errors in a Program

An error that occurs in one component of a program should not cause the entire program to crash.

At the very least, there should be an opportunity to save the current state.

Errors in a Program

In languages like C, error codes can be returned from functions.

This is not fool-proof! Bad input can always cause runtime errors.

Errors in a Program

What errors appear most often?

Errors in a Program

What errors appear most often?

- **Input Errors** - It is not always easy to predict which input your program will receive.
- **Device Errors** - Interaction with physical devices such as physical drives or other machines could fail.
- **Code Errors** - A method could return a bad value which results in a future error.

Errors in a Program

Java resolves these types of errors with **exception handling**.

That is, a special object that is *thrown* due to some error.

Exceptions in Java

Everything is a class in Java and **exceptions** are no... exception.

An **exception** object contains information about the error and can be *caught* by an **exception handler**.

Exceptions in Java

Exceptions derive from the `Throwable` class.

This class has two subclasses `Error` and `Exception`.

The `Error` subclass is typically reserved for internal Java errors.
We will focus on the `Exception` class.

Exceptions in Java

The `Exception` class has two useful subclasses: `IOException` and `RuntimeException`.

A `RuntimeException` is typically thrown due to a programming errors.

Other exceptions are typically thrown due to errors outside of your program's control.

Exceptions in Java

Examples of `RuntimeException`:

- `ArrayIndexOutOfBoundsException` - An attempt to access an invalid entry in an array.
- `IllegalArgumentException` - Invalid input to a method was given.
- `NumberFormatException` - Occurs when an input cannot be converted to the appropriate number.
- `NullPointerException` - Attempting to call a method on an object that is `null`.

Exceptions in Java

Example: `NumberFormatExceptionTest.java`

Exceptions in Java

Examples of `IOException`

- `FileNotFoundException` - An attempt to open a file that does not exist.
- `EOFException` - The end of a file was reached unexpectedly.
- `FileSystemException` - Thrown when a file system operation fails on one or two files.
- `NullPointerException` - Attempting to call a method on an object that is `null`.

Exceptions in Java

Exceptions that derive from `Exception` or `RuntimeException` are called **unchecked exceptions**.

All other exceptions are **checked exceptions** and the compiler will make sure you provide handlers for them.

Exceptions in Java

Generally, **unchecked exceptions** can be resolved by writing more robust code.

Create checks for boundaries, verify user input, etc.

Our approach to exception handling will skew more towards **checked exceptions** for now.

Checked Exceptions

Let's revisit a file I/O example. When we attempt to open a file using a `Scanner` object, we are required to handle an `IOException`.

```
public static void main(String[] args) throws IOException {
    Scanner in = new Scanner(new File("file.txt"));
}
```

Checked Exceptions

In that example, we absolve ourselves of the responsibility of handling the file exception by throwing it.

If the file `"file.txt"` cannot be found, the program crashes.

Checked Exceptions

You should throw exceptions from your method if

1. You call a method that also throws an exception.
2. Your method detects an error and throws a checked exception.

If you don't throw exceptions in these cases, then any calls to your method could lead to program termination.

Checked Exceptions

Methods can also throw multiple exceptions.

```
void readFile() throws IOException, NumberFormatException {
    ...
}
```

Throwing Exceptions

Any exception can be used in your code by **throwing** it.

That is, if you foresee a potential error then you can throw a relevant exception.

Throwing Exceptions

Example: `ThrowExceptionExample.java`

Throwing Exceptions

The previous example shows how we can throw exceptions at any time we please.

We should try to keep our exceptions descriptive and necessary.

Once an exception is thrown in a method, its return value is irrelevant.

Catching Exceptions

We may choose to catch and handle exceptions ourselves, allowing our code to gracefully deal with otherwise catastrophic errors.

This is done via the `try-catch` block.

Catching Exceptions

Example: `CatchFileExceptionExample.java`

Catching Exceptions

The keywords here are `try` and `catch`.

If your code would execute a statement that may throw a checked exception, you should wrap the logic in a `try-catch` block.

Catching Exceptions

If a statement in the `try` block throws an exception, no further statements in that block are executed.

Any exceptions not explicitly caught by the block will cause the method to exit.

If that exception is not caught elsewhere on the stack, the program will terminate.

Catching Exceptions

In general, you should refer to the official Java documentation to learn which exceptions your program should handle given a particular class or method.

Catching Multiple Exceptions

If you need to handle multiple exceptions in a `try-catch` block, just add them to the chain.

Example: `MultipleExceptionExample.java`

Examining Exceptions

You can retrieve error details of exceptions with the following class methods.

- `getMessage()` - Retrieve the error message associated with the exception.
- `printStackTrace()` - Prints the entire method stack trace leading to the exception.

Examining Exceptions

Additional control over the stack trace is accessible via the `StackWalker` class.

`https://docs.oracle.com/en/java/javase/16/docs/api/java.base/java/lang/StackWalker.html`

Rethrowing Exceptions

There may be times when we want to handle an exception and rethrow a new one.

This is perfectly valid in a number of circumstances, but there is a right way to do it.

Rethrowing Exceptions

Example: `RethrowExceptionExample.java`

Rethrowing Exceptions

Another reason you may wish to rethrow an exception is if your method only logs the error.

In this case, the exception should be rethrown and handled elsewhere.

The `finally` Block

Once an exception is caught and handled, no further statements in a method are executed.

This could lead to memory leaks or other related issues if any resource handles are not properly closed.

The `finally` Block

To deal with this, Java provides the `finally` block that appears as the last block in a `try-catch-finally` block.

The `finally` Block

Example: `FinallyExample.java`

try-with-Resources

Similar to other languages such as Python, Java (starting with SE 7) provides a useful pattern called **try-with-Resources**.

If a resource implements the `Closeable` interface, you can use a convenient pattern.

try-with-Resources

Example: `TryWithResourcesExample.java`

try-with-Resources

Classes such as `Scanner` implement the `Closeable` interface.

They will automatically close without an explicit call to the `close()` method as long as they are in a `try-with-Resources` block.

Creating Custom Exceptions

Need a custom exception that does not exist in the Java API?

You can create a new class that extends `Exception`.

Creating Custom Exceptions

Consider a program that parses a CSV file.

We may want to create a custom exception to show an error in parsing a line of CSV.

Creating Custom Exceptions

Example: `CsvParseTest.java`