Java Errors and Exceptions

When executing Java code, different errors can occur, such as coding errors made by the programmer, errors due to incorrect input, or any number of other unforeseeable things. When an error occurs, Java will normally stop the program and generate an error message. The technical description for this is that Java will "throw an exception" or "throw an error". These errors stop our code in its tracks, meaning that no matter what other code we had in our program, everything will halt where the exception was thrown.

But let's clarify before we move on, because exceptions are not the same as errors.

#Errors

Errors are usually caused by serious problems that are outside the control of the program, such as running out of memory or a system crash. Errors are represented by the `Error` class and its subclasses. Some common examples of errors in Java include:

- **OutOfMemoryError:** Thrown when the Java Virtual Machine (JVM) runs out of memory.
- **StackOverflowError:** Thrown when the call stack overflows due to too many method invocations.

Since errors are generally caused by problems that cannot be recovered from, it's usually not appropriate for a program to catch errors. Instead, the best course of action is usually to just log the error and exit the program.

#Exceptions

Exceptions, on the other hand, are used to handle issues that *can be* recovered from within our programs. Exceptions are represented by the `Exception` class and its subclasses. Some common examples of exceptions in Java include:

- **NullPointerException**: Thrown when a null reference is accessed.
- **IllegalArgumentException:** Thrown when an illegal argument is passed to a method.
- **IOException:** Thrown when an I/O operation fails.

Since exceptions can be caught and handled within a program, it’s common to include code to catch and handle exceptions in Java programs. By handling exceptions, you can provide more informative error messages to users and prevent the program from crashing.

Errors and exceptions represent different types of problems that can occur during program execution. Errors are usually caused by serious problems that cannot be recovered from, while exceptions are used to handle recoverable errors within a program. Given that we can't write our code to recover from errors, we'll be focusing on exceptions.

When an exception occurs in the wild, it'd be better to have our program continue running, even if it faces some adversity. In order to achieve this functionality, we can provide our programs with `try` and `catch` blocks–sets of instructions to try, and what to do if the program catches an error.

#Java try and catch

The try statement allows you to define a block of code to watch for exceptions in while it is being executed.
The catch statement allows you to define a block of code to be executed if an exception occurs in the try block.
The `try` and `catch` keywords come in pairs:

CopyCC#C++ClojureCSSDartGoHaskellHTMLJavaJavaScriptJSONJSXKotlinMarkdown PascalPerlPHPPlain TextPythonRRubyRustSchemeShellSQLSwiftTypescriptVB.NETVBScriptXMLYAML

```
try {
// The block of code we'd like to accomplish, if it's possible
} catch (Exception e) {
```

```
// What to do if the try fails-handling any returned
exceptions (e)
}
```

Consider the following example:

Run

Main.java

```
1 public class Main {
2   public static void main(String[ ] args) {
3     int[] myNumbers = {1, 2, 3, 4, 5, 6, 7, 8};
4     System.out.println(myNumbers[8]);
5     System.out.println("That was easy.");
6   }
7 }
8
```

CONSOLESHELL

At what line will the above code fail? Why? Hint: think about how arrays are indexed.

8

4

1

3

If you click run on the IDE above, the code will fail and present you with an **error message** like the one below:

```
Exception in thread "main"
java.lang.ArrayIndexOutOfBoundsException: 8
at Main.main(Main.java:4)
```

Without try/catch, the exception will simply output to the console and the program is terminated, and the second `println` never runs. The exception helpfully tells us where the trouble has occurred (`Main.java:4` means line 4 of our `Main.java` file), and we now have a chance to get in there and debug the issue.

Using try/catch, we can achieve two goals: we can personalize how we see the error ***and*** ensure that the code to follow continues running:

Run

Main.java

```
  public static void main(String[ ] args) {
    try {
      int[] myNumbers = {1, 2, 3, 4, 5, 6, 7, 8};
      System.out.println(myNumbers[8]);
    } catch (Exception e) {
      System.out.println("Something went wrong in numbers printing function. E
rror: " + e);
    }
    System.out.println("That was easy.");
  }
}
```

CONSOLESHELL

Our new output:

CopyCC#C++ClojureCSSDartGoHaskellHTMLJavaJavaScriptJSONJSXKotlinMarkdown PascalPerlPHPPlain TextPythonRRubyRustSchemeShellSQLSwiftTypescriptVB.NETVBScriptXMLYAML

```
Something went wrong in numbers printing function. Error:
java.lang.ArrayIndexOutOfBoundsException: 8
That was easy.
```

As your code grows in complexity, you're going to need to learn to embrace errors, and enjoy the process of hunting them down and fixing them. While the above message has lost its line number reference, we've ensured that the program didn't terminate before we wanted it to.

Although this is a bit of a tangent, it bears repeating: we've also written our code to tell us about the function in which the exception occurred, rather than simply returning the error stack message. Sometimes, the line numbers in our exceptions will call out an issue with something that isn't exactly the code we need to fix, as is the case when previously written lines of code haven't been closed properly, or an outside resource hasn't been properly imported into the file. In a project that spans many files, appropriate error messaging with language that makes sense in a more direct way than the original error message saves us time.

Think back to some of the errors you've seen so far in your time here at Academy. In some of our more complex mini-projects, was there a time when this kind of error handling would have helped you out? If so, how? Hint: think about comments you've left yourself, and how they could be reflected in your error messaging.

Here's how error handling messages could have helped:
File Loading Errors: There was an issue with loading the file, file not being found or having incorrect formatting, proper error handling catch these issues. Informative error messages provide guidance on checking the file's path, ensuring the correct format, or even suggesting alternative solutions.
Data Processing Errors: During data processing, there instances where certain columns have missing or unexpected values, causing calculations to fail. Instead of just crashing with a generic error, error handling identify these problematic data points and provide context about what's causing the issue.

#Try, Catch, and Finally

Sometimes, you need code to run whether or not the desired outcome was achieved, but you want to tie it to the original `try`. In those cases, you can follow your try and catch with the `finally` keyword. When your `try` and/or `catch` block has finished running, the `finally` block will run no matter what.

```
try {
// The block of code we'd like to accomplish
} catch (Exception e) {
// What to do if the try fails and to handle any returned
errors
} finally {
// Code to run no matter what the outcome of the try/catch
block above was
}
```

See the example below:

Run

Main.java

```
public class Main {
  public static void main(String[] args) {
    try {
      int[] myNumbers = {1, 2, 3, 4, 5, 6, 7, 8};
      System.out.println(myNumbers[8]);
    } catch (Exception e) {
      System.out.println("Something went wrong in numbers printing function. E
rror: " + e);
    } finally {
      System.out.println("The 'try catch' block has finished.");
    }
    System.out.println("That was easy.");
  }
}
```

CONSOLESHELL

#Knowledge Check

Fill In The Blank

If the code in a try block fails to run properly, the block will be run.

Fill In The Blank

Whether the try fails or succeeds doesn't matter to the block: it will run either way.

Of the following, select all reasons for using try and catch in your code:

to trigger code blocks that run when an error is encountered.

to make code that can literally never fail, under any circumstances.

to ensure that our program doesn't crash when an error/exception is encountered.
to utilize error messages as we see fit, rather than printing them directly to the console.

#Extra Resources

In upcoming lessons, you're going to go through examples similar to the ones that John walks through in the video below. There's even one sneaky situation at the end of the video regarding another keyword, for those of you who really like to know all the details:

Open in new tab

https://youtu.be/1XAfapkBQjk

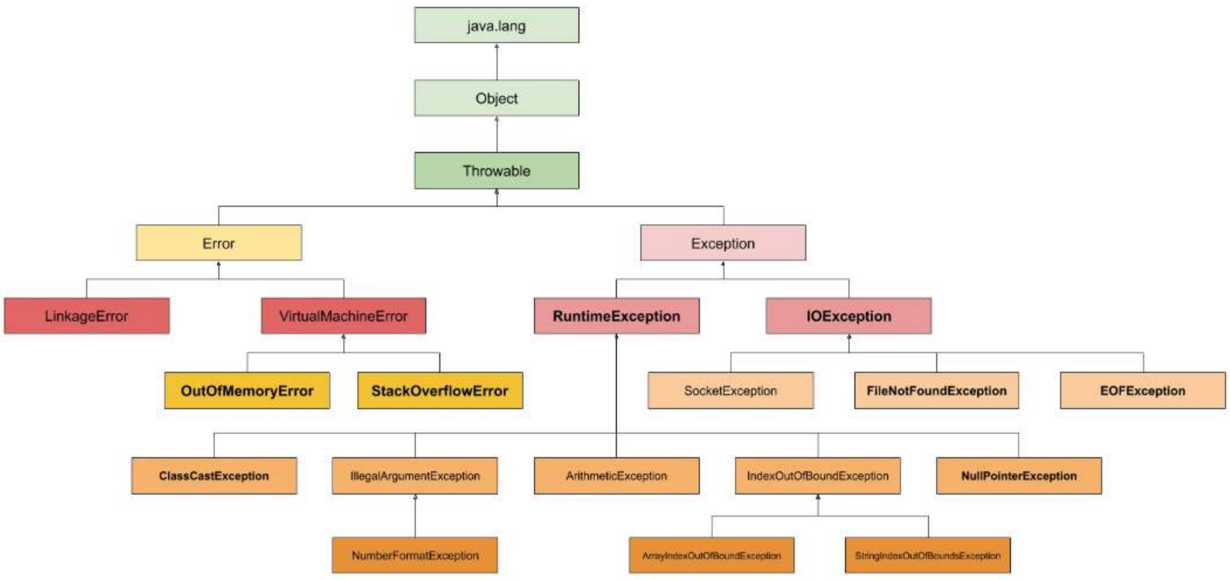
Exception Handling Program

Exception handling is one of the core components of all programming languages. It takes a look at what might go wrong and sets up conditions that gracefully handle those exceptions, rather than allowing the code to simply fail.

Commonly, errors/thrown exceptions result from one of the following issues:

1. **Faulty input**: when a user or other program provides the wrong inputs to the system.
2. **Files don't exist**: when the program attempts to access a file and finds that it isn't there.
3. **I/O operations**: when the input and output aren't what's expected.
4. **Parsing Collections, Array, etc**: as in when you attempt to access an index of an array that isn't there.
5. **Coding errors**: when we, as devs, make mistakes.

For a visual, here is the class hierarchy for handling exceptions in Java:



A chart showing the hierarchy of exception handlers in Java

We'll go over exception handling in more detail throughout the week, but the chart above is a nice one to have handy. After all, you can tell Java to catch any of the specific exceptions that you see above, and handle that exception however you want it to be handled.

For this exercise, however, we'll keep it simple. Let's start by programming an example of an arithmetic exception, catching it, and returning a custom message.

#Problem Statement

Using the starter code provided in the IDE, write a program to catch an error according to the following steps:

Steps to Follow:

1. Define a try-catch block
2. In the try block, declare integers `a` = 11 and `b` = 0;
3. Print the value of `a / b` (`a` divided by `b`).
4. Define the catch block to watch for an `ArithmeticException` error, and print "The number cannot be divided by zero" when it encounters such an error.

If the above isn't clear enough, you can access the hint below. Please try your best before looking, and make sure you get your code into the IDE for this lesson before moving on.

```java
public class ExceptionLearning {

    public static void main(String[] args) {

        try {
            // Block of code to try
            int a = 11;
            int b = 0;
            System.out.println(a / b); // This will cause an ArithmeticExcepti
on
        } catch (ArithmeticException e) {
            // Block of code to handle errors
            System.out.println("The number cannot be divided by zero");
        }
    }
}
```