Exceptions

Bad Things Happen

- Ideally, your code is perfect, but...
- Errors are inevitable, especially during runtime
- What are ways we can handle them?
  - Ignore them and hope they go away
    - Unacceptable
  - Error flags and states
    - The old "C style"
    - Not a very pleasant way to handle them at all
  - Exceptions
- Java has Exceptions built into the language itself

Exceptions

- When a program enters a state the program doesn't know how to handle, the coder specifies that an exception to `throw`
- The program exits the method it is in, and continues to do so until either the `main` method is exited or the exception is caught
- When the exception is caught, code can be specified to recover from the unexpected state.

Why Exceptions

- A method may not know what to do in an error state
  - Often, it shouldn't!
- "Fixing" an error in a method may be inappropriate
  - We will discuss some good coding design practices that may help you write "good" code
- "But I still want to fix it here! Why do I need to do this?"

Exceptions Motivation

- At the beginning of the school year, you receive a laptop computer (from the university, or you buy one)
- Two weeks into the semester, your computer does not turn on. Your computer is still under warranty (or has free technical support from the university)
- What do you do?

Exceptions Motivation

- Maybe you *could* diagnose and fix the problem. However, you are not sure what the problem is, and could potentially make the problem worse by trying. (Also, that could cost money)
- The solution: call technical support (for the people who gave you your laptop).
  - You recognize that your computer is not working correctly (is in an error state)
  - You contact the entity that provided you the laptop and expect that they fix the problem
- What happens if the first tier technical support cannot fix your problem?
  - The problem is escalated up a tier of technical support.

But Why Not Fix It There?

- Consider a method that takes an array and an index (or range of indexes) into the array
- What happens if one of the indices is less than zero, or larger than is allowed (`array.length`)?
  - You could probably fix it (assume zero, or max index).
  - Better question: Why did this error occur?
  - How did you get the indices? Was it user input? Calculated? Random?
  - There may be a problem with your code elsewhere that needs addressing!
- By fixing the problem in that method, you may be making your code less reusable (which as you write more code will become a larger concern).
- Exceptions can help you figure out *where* in your code there are problems!

Bug Hunting

Consider this source code:

```
import java.util.Random;

public class Main {

    private static Random rando;
    public static int bar(int a, int b)
    {
        return a / b;
    }
    public static int foo(int n)
    {
        rando = new Random( );
        int d = rando.nextInt(n) / n * 5;
        return bar(n, d);
    }
    public static void main(String[] args) {
        int result = foo(10);
    }
}
```

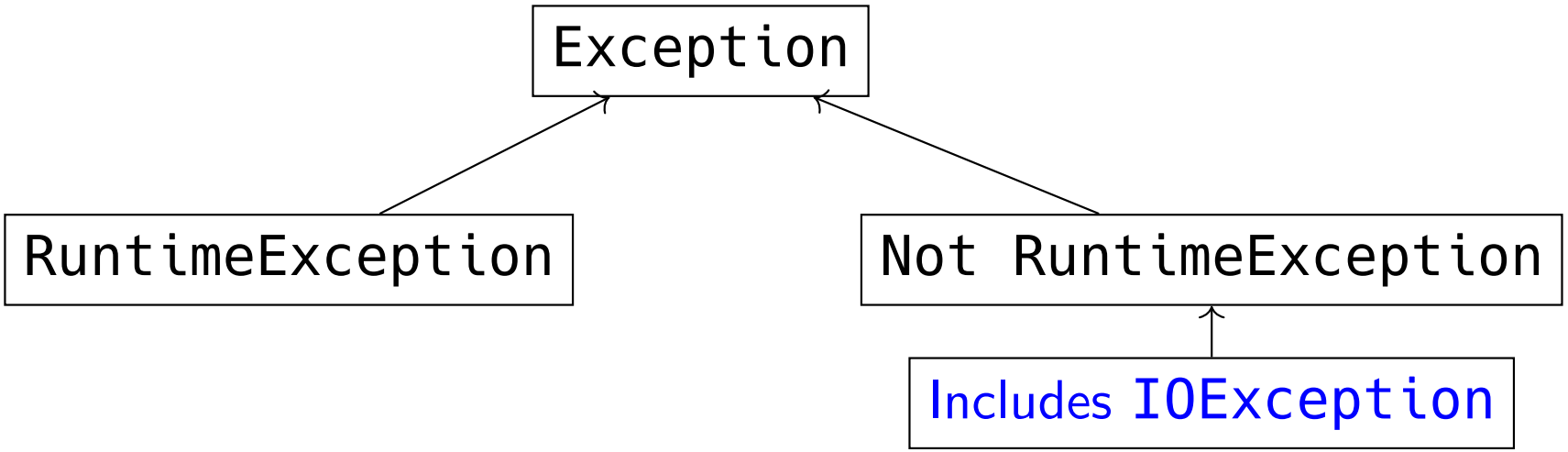
Output

```
Exception in thread "main" java.lang.ArithmeticException: / by zero
    at com.company.Main.bar(Main.java:10)
    at com.company.Main.foo(Main.java:16)
    at com.company.Main.main(Main.java:19)
```

Understanding the Output

- We know (from reading the code) that the program starts in `main`, which calls `foo`, which in turn calls `bar`.
- Looking at the output, we see that there was an *unhandled exception* (of the type Arithmetic Exception) in the method `bar`
  - Specifically, in line 10 of `Main.java`
- The next line tells us *where bar was called from*
  - In this case, it was in the method `foo` at line 16 of `Main.java`
- The next line tells us that `foo` was called from the `main` method at line 19
- This is called a **backtrace**
  - We can see where the exception originated from, and the methods that were called to reach that point
  - A back trace and a working knowledge of the debugger can be a very powerful tool in finding errors in our code

Classes of Exceptions



Specific Exception Types

- Subclasses of `IOException`
  - `EOFExcception`
  - `FileNotFoundException`
- Subclasses of `RuntimeException`
  - `InputMismatchException`
  - `NullPointerException`
  - `IndexOutOfBBoundsException`

Advertising Exceptions

- Certain exceptions must be *advertised* (or caught)
  - We call these Checked Exceptions
  - All exceptions except `RuntimeException`s are checked; `RuntimeException`s are Unchecked Exceptions
- Exceptions are advertised by putting `throws <exceptionType>` in the method signature
  - You should be as specific as possible when advertising exceptions!
- If you handle an exception, you do not need to advertise it

Advertising Exceptions

```
public static void myMethod() throws EOFException
{
//...
}
```

Catching Exceptions

```
try
{
    Scanner in = new Scanner (new File("C:/..."));
    num = in.nextInt( );
}
catch (FileNotFoundException e)
{
    System.out.println("File not found");
}
catch (InputMismatchException e)
{
    System.out.println("error number 2" + e);
}
```

Some Catching Caveats

- In the previous example, the `InputMismatchException` does not need to be handled
  - It is an unchecked exception; it does not need to be caught or advertise
- If `try` is followed by multiple catches:
  - each catch must handle a different exception
  - program tries in order until one matches
    - ergo, you should go from most specific to least specific
- There can be multiple `try`s per block/method, and you can nest them

Throwing Exceptions

```
throw new ExceptionType("mesg");
```

Throwing an Exception

```
// get input in int num;
if (num == -1)
    throw new IndexOutOfBoundsException("positive index");
```

Custom Exceptions

- Is incredibly easy!
- Have your class `extend Exception` (or `RuntimeException`)
  - Overload a no-arg constructor and a constructor that takes a string message
- Alternatively, have Netbeans/your IDE do it for you!

Example Exception

```
//note; could extend Runtime Exception to be unchecked
class BadJuJuException extends Exception
{
    public BadJuJuException( ) {
        super( );
    }
    public BadJuJuException(String mesg) {
        super (mesg);
    }
}
```