Chapter 10
Exceptions

Introduction

Sometimes something goes wrong in attempting to perform the calculations that our code represents. Here are some examples:

- we try to open a file that doesn't exist for reading
- we try to access an element of an array beyond the last element (out-of-bounds index)
- we try to treat null like an actual object, and access members (fields or methods) in it.
- we try to divide by zero
- we try to parse a String to an int via `Integer.parseInt(strExpr)`, but there's no int represented in the String.

In each of these situations, there's not really a best action for the program to automatically take. Even if there were, there are legitimate reasons to prefer an alternative response. In these situations, rather than try to handle this situation, the program quits its normal control flow, and instead raises an exception and immediately leaves each block of code that was running until the exception is handled.

Rather than just have a single String represent the error, Java (and many other languages) creates a hierarchy of classes that represent different classes of errors via inheritance. Here are a few of those exception classes:

Exception Class Name //Indication or Notes

```
java.lang.Throwable                   //(implicitly inherits from Object).
  • Exception
      • RuntimeException              //for recoverable events.
          • NullPointerException //using null like an object
          • ClassCastException   //cast to class-type that wasn't possible
          • IndexOutOfBoundsException
              • ArrayIndexOutOfBoundsException
              • StringIndexOutOfBoundsException
          • ArithmeticException  // bad arithmetic, like "divide by zero"
      • IOException
          • FileNotFoundException //attempted to open non-existing file
          • EOFException          //end of file reached (no more content)
  • Error                             //unrecoverable events: e.g., out of memory
```

For example, Exception extends Throwable, ArithmeticException extends RuntimeException, EOFException extends IOException, and so on. Because these are classes, each of them is a different type, and we will use these types in our code to be as general or specific as we'd like: we can deal with any IndexOutOfBoundsException, or just NullPointerExceptions, or all exceptions at once with the Exception type.

When do exceptions occur?

Exceptions can occur in almost any place:

1. in expressions that fail:
   - `5/0` (divide by zero: ArithmeticExpression)
   - `xs[i]` (when i is invalid index: ArrayOutOfBoundsException)
   - `sq.side` (when sq==null: NullPointerException)
   - `(Square) "2x2"` (casting, String is not a Square: ClassCastException)
2. within methods that we call:
   - any method that contains expressions like the above
   - `Integer.parseInt("no numbers here!")` (NumberFormatException)
   - `Scanner sc = new Scanner(new FileReader("foo.txt"));` (FileNotFoundException)
3. in expressions that purposefully raise an exception:
   - throw new Exception("something went wrong!");
   - throw new ArithmeticException("needed an even integer, found " + x);

In all of these cases, the normal notion of what instruction should run next is abandoned, and instead the block of code being called is immediately exited with the exception value instead of whatever return value was expected, and the method that called that method is also exited with the exception value, and so on, until the main method (the whole program) crashes.

Your Turn!

- Try using expressions that divide by zero, use an invalid index, use a null pointer as if it pointed to an object, and see the exceptions crash the program.
- Try calling the Integer.parseInt method and creating a Scanner on a non-existent file to see the exceptions crash the program.
- Notice how we get different information, and different types of exceptions, for each case.
- ***Create your own exception values***, and throw them: just like the third group of example sources of exceptions, try creating objects of the ArithmeticException class and the NullPointerException class. You might need to look them up to find out what parameters are needed for the constructors (google "`java ArithmeticException`"), but often there's a single-String-argument constructor.

Catching Exceptions

We can stop this program crash from happening, using a try-catch block. Whenever a block of code is capable of causing a certain kind of exception, we can wrap that block up in a try block, accompanied by a catch block that addresses that specific **type** of exception.

```
try {
      int[] xs = {5,3,2,4,6};
      int x = xs[23];
}
catch (ArrayIndexOutOfBoundsException e) {
      System.out.println("whoops! bad index.");
}
```

Of course, we're just using printing to make it convenient to see where program flow actually goes, but it would be normal to have no printing inside the catch block, and instead just have code that addresses the situation.

As you read the documentation for various classes, you might find notes that a method may throw certain exceptions. This is your cue to decide if you want your code to handle that exception or not. (We'll learn what happens if you don't in a moment).

Your Turn!

- Given the examples of exceptions above, and the type of exception listed next to it, try to **catch** the exception in each case:
    - division by zero
    - accessing a null pointer as if it referred to an actual object value
    - opening a file that doesn't exist

Multiple catch blocks

A single try-block may contain code that could cause a variety of different exceptions. We don't have to zoom in and wrap each single statement in a try-catch block of a different variety.

```java
public static int foo (int[] xs, int ix, int factor, Square sq) {

      try {
          int temp = xs[ix];  // ix might be an invalid index. xs might be null.
          sq.side = temp / factor;  // sq might be null.  factor might be zero.
          System.out.println( (Square) (Object) sq.side); // class cast exception.
      }
      catch (ArrayIndexOutOfBoundsException e) {
          System.out.println("uh-oh index: " + e);
          return -1;
      }
      catch (NullPointerException e) {
          System.out.println("booo, null! " + e);
          return -2;
      }
      catch (ArithmeticException e) {
          System.out.println("uh-oh, math. " + e);
          return -3;
      }
      catch (Exception e) {
          System.out.println("I forgot to specially handle this: " + e);
          return -4;
      }
      return sq.side;
}
```

The single try-block of this method can cause ArrayIndexOutOfBoundsException, NullPointerException, ArithmeticException, and ClassCastException exceptions. We can separately try to catch any or all of these. We do so for the first three (array index, null pointer, divide by zero).

However, we also write a general catch block to catch any Exception value. Due to inheritance, this Exception catch-block could actually catch all the other exceptions. If we wanted to perform the same action regardless of which exception occurred, we could have used just this one catch-block.

Being able to include catch-blocks for exception types that are parent classes and child classes of each other allows us to create filters of increasing generality that will catch any exceptions as necessary. It is important to note:

Only one exception can occur in a try-block, and then only the first catch-block that matches the exception's type will be executed.

So, even though a NullPointerException occurs, because we had the NullPointerException catch-block before the Exception block, only the NullPointerException catch-block will run. If we instead had placed the Exception catch-block first, it happens to match all the other more specific exceptions, and when listed first, it would be the only one that would ever run.

Your Turn!

- Write a method that accepts an Object parameter, casts it to a Square, and returns its side. Use a try-catch block to catch both the NullPointerException and the ClassCastException.
    - Feed testing inputs to trigger both catch-blocks.
- The following code will open a file, read a line, and close it. It can cause an exception, though, when the file doesn't exist. Run the code to find the exception type (be sure to use a filename that doesn't exist!), then add a try-catch block to keep the program from crashing.

```
Scanner sc = new Scanner (new FileReader ("foo.txt"));
System.out.println(sc.nextLine());
sc.close();
```

- Modify the previous task's code to ask the user for the filename in the first place; as often as the user gives a non-existent file name, keep looping over and over to ask them again until a valid filename is given without letting the program crash.
- Use a Scanner to read an int from the keyboard via standard input (System.in). As often as the user types a non-integer in the input, keep gracefully asking again until you successfully read the int.

`finally` Blocks

In addition to try and catch blocks, we can also add finally blocks. A finally block is placed after the last catch block. It always runs:

- If the try block is successful with no exceptions, it runs.
- If the try block fails with an exception and the exception is caught, it runs.
- If the try block fails with an exception and it is not caught, the finally block runs.

Consider the following code:

```
public static void finallyMethod (Square s) {
        try {
            System.out.println(5/s.side);
        }
        catch (ArithmeticException e) {
            System.out.println("caught infinity.");
        }
        finally {
            System.out.println("finally.");
        }
}
```

Your Turn!

- Test the above code with a Square that:
    - has a non-zero side length
    - has a side length of zero
    - is null

  → in all cases, does "finally" print?

  → the finally block runs before a propagated exception proceeds to propagate.

- throw another exception in the finally block. Re-test the three examples. What happens now? Which exception propagates, the original one or the new one?
    - Lesson: make your catch and finally blocks simple enough that they don't also throw exceptions.

Propagating Exceptions

If we don't want to handle an exception (or don't know what to do), we can also allow the exception to crash its way out of its current location. Even if the suspicious code is in a try-catch block, the particular exception might not match the types afforded by any of the catch-blocks. Or, we might not even wrap the offending code in a try-catch block at all.

There are two groups of Exceptions: those that can silently be propagated, called "unchecked exceptions", and those that must explicitly be announced as propagated, called "checked exceptions".

The unchecked exceptions are the Error class, the RuntimeException class, and any children of these two classes. The Error class represents things that can't reasonably be recovered from within a program – if the hard disk is failing, or the program has run out of memory, or some other physical issue happens to have arisen, the program's crashing is likely the least of the user's concerns. From the other direction, RuntimeException exceptions represent common events that are likely recoverable – if we find a null pointer, perhaps we know what to return instead; if we find that a class cast conversion didn't work, perhaps we know what to do instead. For these alternate reasons, we don't have to explicitly list when we aren't handling these types of exceptions; indeed, we've been silently propagating these exceptions throughout our code all along.

In essence, unchecked exceptions usually represent bugs in the code, and if the programmer sufficiently tests the code, these should no longer happen. Whenever they do, it indicates that the program wasn't written to handle all the corner cases, but that it could be edited to handle this situation. It would be an immense nuisance to have to explicitly handle all possible exceptions – it would destroy the readability of almost everything.

The checked exceptions are all others: things like FileNotFoundException and other IOExceptions may be the most common checked exceptions you will experience. If a method attempts code that can throw one of these exceptions, we must explicitly confess that the particular method may result in one of these exceptions rather than completing with regular control flow:

```
public static int readNumFile  (String filename) throws FileNotFoundException {
      Scanner sc = new Scanner (new File (filename));
      return sc.nextInt();
}
```

Notice that this example is trying to read a file, which might not exist. So we add "throws FileNotFoundException" to the method signature, **after** the parameters. This method also could throw a java.util.InputMismatchException, which is a child class of java.util.NoSuchElementException, which is a child class of java.lang.RuntimeException. Thus it is an implicit exception and we didn't have to list it.

In practice, you will perhaps find the Java compiler generate an error/complaint when you use code that is capable of generating a checked exception, but you forgot to explicitly list it in the method signature.

You can explicitly list unchecked exceptions too; the compiler allows it, and it can serve as extra documentation about the likely behavior of a method.

Your Turn!

- We don't have many checked exception types that we can run across yet (but we'll see one more IOException when we look at file I/O). So there's not much to do here yet.
- Write your own method that opens a file and assumes it exists, and keeps reading numbers from it until the end of the file is reached. What exceptions will you explicitly propagate? What other exceptions will you find on the way, and then handle with a try-catch block? Writing an optimistic version of your program and letting it crash is a simple way to discover the correct exception type.

Creating Your Own Exception Classes

Exception classes are just that: classes. If you want a more specific exception, you can just extend the specific class you want, perhaps add your own fields, add a constructor or two, and then create them.

```
public class MyEx extends Exception {

    public int x;

    public MyEx (int x) {
        this.x = x;
    }

    public String toString () {
        return "MyEx: "+x;
    }
}
```

- Although only extending some exception class is necessary to create your own exception class, it is good to extend the most specific exception class you can.
- Also, it is good to always provide the toString method.
- If you extend a RuntimeException class or an Error class, your class will also be an unchecked exception.

Throwing Exceptions Manually.

You can now create objects of your classes that happen to be exception classes, and then throw them:

```
MyEx me = new MyEx(5);
throw me;
```

This code could then show up inside a try-catch block (catching a `MyEx`), or since we elected to directly extend the `Exception` class, we could also explicitly propagate this checked exception in the method signature.

Given this functionality, you can now approach a programming task by thinking, what hard-to-solve situations might arise here? Can I just create my own exception class, throw one as necessary, and regroup/fix the situation from somewhere else in the code? This is particularly useful if the tricky/confusing/bad situation occurs in one place in code, and the needed information to solve the situation occurs somewhere else in the code.

Your Turn!

- Create your own exception class, named `NegativeIntegerException`, which is a child class of the `ArithmeticException` class. Give it meaningful fields, a constructor, and an overriden toString method.
- Write a small program that gets a number from the user via `System.in` and a `Scanner`. If they enter a negative number, create one of your exception objects and throw it.

Continued Propagation

Some of the exceptions we've caught were actually generated inside other methods. It is possible for an exception to be uncaught from method to method, but if the exception were a checked exception, each method would have to explicitly list that it throws the exception that it is choosing to allow to propagate beyond its scope.

Your Turn!

- Write three methods that call each other named methodA, methodB, and methodC. Have the innermost one throw one of your NegativeIntegerExceptions, and let the other two explicitly throw it beyond their own scope.
- Now, have the outermost method wrap the call to the middle method in a try-catch block that catches NegativeIntegerExceptions.
  → At each stage, your methods can choose to catch some exceptions, and propagate others. But the checked exceptions must be listed, always.