Exception Handling

the throwable hierarchy

Among the more recognizable indications of Java's exception handling mechanism is the stack trace printed when an `Exception` is thrown out of `main()`. In this scenario, some abnormal condition gives rise to an exceptional situation where the flow of the program must change course. For instance, consider that each argument from the command line is given in the form of a `String`. If that `String` object represents an integer value, we need to parse the `String` object so that we can use it as an integer.

```
class Divide {
    public static void main(String [] args) {
        int dividend = Integer.parseInt(args[0]);
        int divisor = Integer.parseInt(args[1]);
        int quotient = dividend / divisor;
        System.out.println(quotient);
    }
}
```

Parsing an integer value from the `String` object opens the possibility for abnormal cases, because the `String` object may not actually represent a number. If this program were executed from the command line like so:

```
[user@notnotbc]$ java Divide one all
Exception in thread "main" NumberFormatException: For input string: "one"
    at java.lang.NumberFormatException.forInputString(Unknown Source)
    at java.lang.Integer.parseInt(Unknown Source)
    at java.lang.Integer.parseInt(Unknown Source)
    at Divide.main(Divide.java:3)
```

then the values of the arguments would be `one` and `all`, from which we cannot parse any integer value. Therefore, we run up against an exceptional case, where we have tried to parse an integer from a `String` object which does not represent one. A `NumberFormatException` is eventually thrown out of `main()` as can be seen above. The stack trace provides a means of determining exactly where a faulty operation was executed. In the case above, we see the `NumberFormatException` was originally thrown in a method called `forInputString()`.

The way one knows whether a method may or may not `throw` an `Exception` is by looking at the method signature in the API. From the Java API:

```
public static int parseInt(String s) throws NumberFormatException
```

However, some operations do not have a method signature – i.e. array indexing, arithmetic operations, etc. The code above could `throw` an `Exception` for many reasons other than bad input to the `parseInt()` method. For instance, here are a few invalid inputs and their corresponding outputs:

```
[user@notnotbc]$ java Divide
Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException: 0
    at Divide.main(Divide.java:4)

[user@notnotbc]$ java Divide 1
Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException: 1
    at Divide.main(Divide.java:5)

[user@notnotbc]$ java Divide 1 0
Exception in thread "main" java.lang.ArithmeticException: / by zero
    at Divide.main(Divide.java:6)
```

In the first case, no arguments were given and the code does not check for this possibility. The result is an attempt to access an array of length 0 which throws an `ArrayIndexOutOfBoundsException`:

```
int divisor = Integer.parseInt(args[0]);
```

The second case is very similar except the code tries to access the second index of an array of length 1:

```
int dividend = Integer.parseInt(args[1]);
```

In the third case, a division by zero is attempted which the code does not guard against. In this case, we see an `ArithmeticException`.

In all three of these cases, notice that the `Exception`s did not need to be handled or declared. We are used to seeing `Exception`s handled in `try-catch` blocks or declared with the `throws` statement. With `IOException`s, for instance, the compiler will terminate compilation if the `Exception` is not handled or declared. For instance, the code below triggers the proceeding compilation errors:

```
class Demo {
    public static void main(String [] args) {
        BufferedReader br = new BufferedReader(
                                new InputStreamReader(System.in));

        String input = br.readLine();
        br.close();
    }
}
```

```
Demo.java:8: error: unreported exception IOException; must be caught or
declared to be thrown
        String input = br.readLine();
                                  ^
Demo.java:9: error: unreported exception IOException; must be caught or
declared to be thrown
        br.close();
                ^
2 errors
```

We may either append a `throws` clause to the method signature:

```
public static void main(String [] args) throws IOException { ... }
```

or wrap the appropriate methods in `try-catch` blocks.

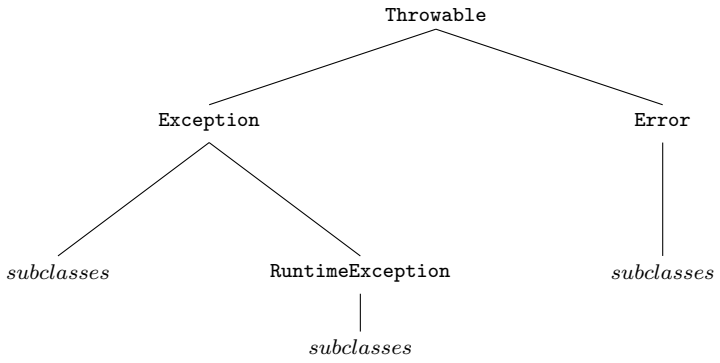
On the other hand, as seen previously, the method `parseInt()` in the `Divide` class explicitly decalares a `NumberFormatException` as can be seen in the API. Though, it neither has to be handled nor declared by the calling method – in this case `main()`. In fact, even if a `main()` method were written as follows, there would be no complaints from the compiler.

```
public static void main(String [] args) {
    throw new NumberFormatException();
}
```

The code above simply throws a `NumberFormatException` and does nothing else. There isn't any ambiguity at all about whether an `Exception` will be thrown, but still no reporting is necessary. The reason why will be explained in the next section after discussing the `Throwable` hierarchy.

`Throwable`

There are many different types of `Exception`s and, in fact, there are even classes that can be thrown that are not, strictly speaking, `Exception`s. At the root of this hierarchy of classes is the `Throwable` class. Any class that is a subclass of `Throwable` may be used in conjunction with the `throw` operator.



The hierarchy above shows two subclasses of `Throwable`: `Error` and `Exception`. The `Error` class is reserved for a particular type of exceptional case from which a program generally can't recover. This includes things like: `OutOfMemoryError`, `ThreadDeath`, `BootStrapMethodError`, etc. These are things that an average Java program would not handle as they are errors that indicate some sort of systemic problem. For instance, the `Divide` class above should not know how to handle an `OutOfMemoryError`. It is almost certain that it isn't a problem with the logic of the `Divide` class, but something systemic that should be handled elsewhere.

The other type of `Throwable` is the `Exception`. `Exception`s generally indicate errors that are, more than likely, relevant to the execution of the particular program in which they were thrown and, therefore, should be handled by that program. All subclasses of `Exception` may be separated into two categories: `RuntimeException`s and everything else.

`RuntimeException`s generally indicate logical errors in the flow of a program. For instance, the `NumberFormatException`, `ArrayIndexOutOfBoundsException`, and `ArithmeticException` which were thrown in in the `Divide` class above are subclasses of `RuntimeException`. It can be seen that they are indications that the logic of the `Divide` class is faulty in some regard. Namely, checks should have been written into the class so that the code would not allow: invalid input strings to `parseInt()`, indexing out of the bounds of the `args` array, or a division by zero.

```
class Divide {
    static boolean isNumeric(String s) {
        if(s == null) return false;
        if(s.equals("")) return false;

        for(char c : s.toCharArray()) {
            if(!Character.isDigit(c))
                return false;
        }

        return true;
    }

    public static void main(String [] args) {
        if(args.length != 2) {
            System.err.println("Usage: java Divide <dividend> <divisor>");
            return;
        }

        int dividend;
        if(isNumeric(args[0])) {
            dividend = Integer.parseInt(args[0]);
        } else {
            System.err.println("Invalid arg: " + args[0]);
            return;
        }

        int divisor;
        if(isNumeric(args[1])) {
            divisor = Integer.parseInt(args[1]);
        } else {
            System.err.println("Invalid arg: " + args[1]);
            return;
        }

        if(divisor == 0) {
            System.err.println("Divisor cannot be zero.");
            return;
        }

        int quotient = dividend / divisor;
        System.out.println(quotient);
    }
}
```

The code above will never suffer from the `RuntimeException`s from which it did previously. This is because the logic of the code now takes the steps to prevent a runtime error from happening. It validates the input strings given as arguments to make sure they can be interpreted as numbers. It ensures that the correct number of arguments was given. And, finally, it checks that the second number given is non-zero.

Another way to rewrite the code in the `Divide` class is to put the entire `main()` method in a `try-catch` block.

```
class Divide {
    public static void main(String [] args) {
        try {
            int divisor = Integer.parseInt(args[0]);
            int dividend = Integer.parseInt(args[1]);
            int quotient = divisor / dividend;
            System.out.println(quotient);
        } catch(RuntimeException r) {
            System.err.println(r.getMessage());
        }
    }
}
```

On the face of it, this would seem to be a much more concise way to write the `Divide` class. However, catching `RuntimeException`s is not an ideal approach to handling problems during the execution of a program. One reason is that it can quickly lead to muddled code and difficulty providing errors to the caller of the method. Consider the output when the program receives invalid inputs:

```
[user@notnotbc]$ java Divide one all
For input string: "one"

[user@notnotbc]$ java Divide
0

[user@notnotbc]$ java Divide 1
1

[user@notnotbc]$ java Divide 1 0
/ by zero
```

These are not particularly informative error messages and, in fact, may be very misleading and confusing. One might separate out the `catch` clauses to provide more informative messages:

```
class Divide {
    public static void main(String [] args) {
        try {
            int divisor = Integer.parseInt(args[0]);
            int dividend = Integer.parseInt(args[1]);
            int quotient = divisor / dividend;
            System.out.println(quotient);
        } catch(ArrayIndexOutOfBoundsException a) {
            System.err.println("Usage: java Divide <dividend> <divisor>");
        } catch(NumberFormatException n) {
            System.err.println("Bad Input " + n.getMessage());
        } catch(ArithmeticException r) {
            System.err.println("Divisor cannot be zero.");
        }
    }
}
```

The output then changes to:

```
[user@notnotbc]$ java Divide
Usage: java Divide <dividend> <divisor>

[user@notnotbc]$ java Divide one all
Bad Input For input string: "one"
```

```
[user@notnotbc]$ java Divide 1
Usage: java Divide <dividend> <divisor>

[user@notnotbc]$ java Divide 1 0
Divisor cannot be zero.
```

This is certainly an improvement, but imagine that there were multiple arrays being indexed in the `try` clause. In that scenario, we would not know which array was being indexed improperly if an `ArrayIndexOutOfBoundsException` were caught. The same is the case with the `ArithmeticException`. Arithmetic operations happen all the time in the execution of a program and the generality of the `catch` takes away from the ability to pinpoint exactly where an error has occured and, for instance, provide useful feedback to a user.

Of course, each statement could be wrapped in a `try-catch` block and then the error messages would be extraordinarily specific. However, this is an unnatractive alternative and makes things very messy. In general, the solution is to *not handle* `RuntimeException`s *and implement robust logic to prevent possible errors*. For this reason, `RuntimeException`s – and all subclasses of `RuntimeException` – are considered *unchecked exceptions*.

Unchecked exceptions do not have to be handled or declared in your code. In other words, the compiler does not concern itself with the possibility of `RuntimeException`s in any way (as can be seen from the previous section). This is because, as we have seen, unchecked exceptions generally should not be handled through Java's provided exception handling mechanism. If the compiler did require that we handle or declare them, Java code would be terribly cluttered with `try-catch` blocks and many, many methods would have to declare `throws` clauses.

There are no strict rules about which `RuntimeException` should or should not be handled or declared. Though, as a general rule: unchecked exceptions should be guarded against through checks in the logic of a program (via input validiation, etc.). Handling some unchecked exceptions may sometimes be an easier and cleaner choice. However, doing so should be considered a deviation from the norm. *A program should* ***not*** *rely on unchecked exceptions to control flow*. This means, for instance, that `ArrayIndexOutOfBoundsException`s should not be caught in order to terminate a loop:

```
try {
    int i = 0;
    while(true)
        System.out.println(arr[i++]);
} catch(ArrayIndexOutOfBoundsException a) { }
```

The piece of code above is a wildly unnecessary use of Java's exception handling mechanisms. It is certainly a contrived case, but it sheds light on what not to do with unchecked exceptions.

Note that `Error`s – the other half of the `Throwable` hierarchy – are also unchecked. That is, `Error`s should generally not be handled by your program. However, this is for a different reason than why `RuntimeException`s should not be handled. They are unchecked because they are usually not indicating an issue truly relevant to the program and, therefore, should not be handled by the program. In other words, `RuntimeException`s are unchecked because the program logic should prevent them from happening, while `Error`s are unchecked because it usually isn't the responsibility of the program to handle those scenarios.

All other subclasses of `Exception` and `Throwable` are checked – i.e. have to be handled or declared. These `Exception`s are used in situations where the flow of execution is interrupted in some way, though this interruption is out of the control of the program. A very common checked exception is the `IOException`. For instance, consider a program that tries to create a new file but encounters an error in the process related to the file system. In such a scenario, the normal flow of execution does not necessarily need to come to halt the way it would with

an `Error`. Though, it's also a scenario in which the programmer couldn't have prevented the problem with logical checks, for instance. So, the programmer specifies a routine for *handling the specific scenario*. This is why checked exceptions are the ones which should be encountered in `try-catch` blocks.