What's the problem? Understanding the issues involved error handling

It's very hard to write programs that handle the unexpected, or even merely unusual, situations gracefully. Think about it: how often do we just say "assume the file exists" or "assume the user enters a positive integer like you asked for"? All the time! Real-world programs can't do that. They have to be built to respond reasonably to any and all wierd inputs, or unusual phenomena that can crop up while computing. These issues are made worse by the fact that well-designed code may end up being reused in different applications and even by different people than the original authors ever imagined. That means lots of things might be thrown at their code that the authors never imagined. None the less, they have to prepare for it.

So let's consider an example of a very simple program. What could go wrong?

Ex1.java

```
/**
 * Run this program like this: 'java Ex1 3,5,9,4'
 * and it will compute
 *                    ____________________________
 * floor[ 100 * /(1/3 + 1/5 + 1/9 + 1/4)/4) ]
 *
 * i.e. the floor of 100 x the square root of the
 * average of the reciprocals of the given numbers.
 */
public class Ex1
{
  public static void main(String[] args)
  {
    String[] A = args[0].split(",");
    int[] B = new int[A.length];
    for(int i = 0; i < A.length; i++)
      B[i] = Integer.parseInt(A[i]);
    double sum = 0;
    for(int i = 0; i < B.length; i++)
      sum += 1.0/B[i];
    double ssum = Math.sqrt(sum/B.length);
    int res = (int)(100.0*ssum);
    System.out.println(res);
  }
}
```

Obviously, this is a very simple program. As far as what could go wrong ... lots of stuff. Try some of these.

```
~/$ java Ex1 1,2,3
78
~/$ java Ex1 -1,-2,-3
0
~/$ java Ex1 1,0,3
2147483647
~/$ java Ex1 1,foo
Exception in thread "main" java.lang.NumberFormatException: For input string: "foo"
        at java.lang.NumberFormatException.forInputString(NumberFormatException.java:65)
        at java.lang.Integer.parseInt(Integer.java:492)
        at java.lang.Integer.parseInt(Integer.java:527)
        at Ex1.main(Ex1.java:15)
~/$ java Ex1 ,
0
~/$ java Ex1
Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException: 0
        at Ex1.main(Ex1.java:12)
```

The first call is OK. Everything else has an error ... some of which the program is telling you about, others not. Can you identify all the things that could go wrong here?

In fact, to be a bit more representative of real programs, let's assume that the we have things broken up into functions and maybe even split into different classes. We might get a situation like this.

Ex2.java

```
public class Ex2
{
  public static int getSF(String[] av)
  {
    String[] A = av[0].split(",");
    int[] B = new int[A.length];
    for(int i = 0; i < A.length; i++)
      B[i] = Integer.parseInt(A[i]);
    return SpecialFunc.compute(B);
  }
```

SpecialFunc.java

```
public class SpecialFunc
{
  public static int compute(int[] B)
  {
    double sum = 0;
    for(int i = 0; i < B.length; i++)
      sum += 1.0/B[i];
    double ssum = Math.sqrt(sum/B.length);
    int res = (int)(100.0*ssum);
    return res;
```

```
  }
  public static void main(String[] args) }
  {
    System.out.println(getSF(args));
  }
}
```

Of course all the same things could go wrong, but now they can go wrong in different places: some problems crop up in main, some in the getSF method, and some in the compute function that's in a totally separate class! So if we want to actually take care to account for these kind of things and act appropriately in every situation ... what can we do? Now we start to see some of the problems that handling errors (or "exceptions") present us with. Here are a few:

1. The place where the error occurrs might be far away from the code that is able to determine what best to do about the error. For example, SpecialFunc.compute() might encounter an error (like a zero value it is supposed to divide by), but it is only really the Ex2.main() method that would know whether an error message should be printed and the program terminated, or whether (for example) the user should be requested to enter a new string.
2. There can be many kinds of errors, and in some situations a portion of the program may need to know precisely which errors occurred while in other situations it may be more desireable to treat all errors or large groups of errors the same. For example, maybe we simply want main() to terminate with a usage message no matter what the error. On the other hand, maybe we want it to give detailed feedback about exactly what the problem is. Maybe we don't want to treat a zero in the input as an error (since we divide by it) but simply report the result as "infinity".
3. Different kinds of errors might require vastly different kinds and amounts of information pertaining to the circumstances that led to the error to properly remmediate. If the error is that a given filename turned out not to exist on the filesystem ... well, the name of the file that the system tried to find is useful information. If the error is that an attempt was made to access an array out of bounds ... well, where in the code this happened and what index was tried is useful information.

The moral of the story is that error handling is difficult. The article "Exception Handling in C++" by Bjarne Strouptrup (the author of the C++ language) is a really great read. It talks about a lot of the difficulties inherent in error handling, and it explains the design of G++'s exception handling mechanism, on which Java's is strongly based. He identifies four standard approaches to exception handling prior to the design of the C++ mechansim:

1. Terminate the program.
2. Return a value representing 'error'
3. Return a legal value and leave the program in an illegal state.
4. Call a function supplied to be called in case of 'error'.

He also details why each of these is insufficient. You ought to be able to come up with some good objections on your own.

Overview of the Java's Exception-handing mechanism

Java's exception handling mechanism basically provides an alternate way to "return" from a function (or, indeed, from an arbitrary block of code, as we'll see) which, instead of returning an object of the return-type specified in the function prototype, returns an object of type `Throwable`, which is a class in the Java API. This special kind of return is acheived with a "throw" statement. Like this:

```
throw new Throwable();
```

We refer to this as "throwing an exception" (or "error" in some cases).

In the code that called the function or evaluated the expression that caused the exception to be thrown, we need some way to capture that throwable object. This is accomplished with a try-catch block.

```
try {
  ... regular old code
}catch(Throwable e) {
  ... code to exectue if an exception was thrown in executing the "regular old code"
}
```

The semantics (meaning) of this is that the regular old code in the try block is executed as usual. If no exceptions are thrown while executing this code, the "catch" block is simply ignored. If, however, an exception is thrown at some point, the "regular old code" following that point is not executed. Instead, control jumps to the "catch" block, and the code in it is executed. This is where you deal with whatever the problem was that caused an exception to be thrown.

Catching an Exception

There are already several things we might do that cause exceptions to be thrown. You can see two by playing with the above program: indexing an array out of bounds, and calling Integer.parseInt() with a string that cannot be interpreted as an integer. So let's see if we can catch these exceptions. Where we catch them, and what we do as a result depends on what the goals of our program are. Here are two possiblities:

Ex2.java

```
public class Ex2
{
  public static int getSF(String[] av)
  {
    String[] A = av[0].split(",");
    int[] B = new int[A.length];
    for(int i = 0; i < A.length; i++)
      B[i] = Integer.parseInt(A[i]);
    return SpecialFunc.compute(B);
  }

  public static void main(String[] args)
  {
    try {
      System.out.println(getSF(args));
    } catch(Throwable e)
    {
      System.out.println("Bad stuff happened!");
    }
  }
}
```

Ex3.java

```
public class Ex3
{
  public static int getSF(String[] av)
  {
    try {
      String[] A = av[0].split(",");
      int[] B = new int[A.length];
      for(int i = 0; i < A.length; i++)
        B[i] = Integer.parseInt(A[i]);
      return SpecialFunc.compute(B);
    } catch(Throwable e)
    { // Shh! Act like nothing happened
      return 0;
    }
  }

  public static void main(String[] args)
  {
    System.out.println(getSF(args));
  }
}
```

Throwing an exception

Whether we want to provide the simple error message "Bad stuff happened!" or whether we want to put our heads in the sand and return zero in the face of any error, the solutions above are only half effective. Why? Because errors like dividing by zero or taking the square root of a negative number are not causing errors to be thrown, and therefore, our calling functions are blissfully unaware that they've even happened. To make our program complete (admittedly with a very simple and stupid kind of response to errors) SpecialFunc.compute() needs to throw exceptions in those cases. That's as simple as adding some checks with

```
throw new Throwable();
```

... for cases when the checks fail, with one little catch. Just as the compiler needs to know what the name, return type and parameters are a for a function, Java needs to know when a method might throw an exception; and it expects that fact to be made specific. How? By adding a "throws clause" to the end of the prototype.

SpecialFunc.java

```
public class SpecialFunc
{
  public static int compute(int[] B) throws Throwable
  {
    double sum = 0;
    for(int i = 0; i < B.length; i++)
    {
      if (B[i] == 0) throw new Throwable();
      sum += 1.0/B[i];
    }
    if (sum < 0) throw new Throwable();
    double ssum = Math.sqrt(sum/B.length);
    int res = (int)(100.0*ssum);
    return res;
  }
}
```

With this, our Ex3 version (which printed out zero whenever there was an error) works great. Our Ex2 version is not quite there, but for an interesting reason: the method getSF() calls SpecialFunc.compute(), which might end up throwing an exception, without making any provision for the possiblity. That, as it turns out, is a no-no! With a caveat that we will address shortly, any Java method that includes code that might result in an exception being thrown must either catch the exception, or pass it along to the next function down the call-stack — essentially itself throwing the exception. This means that such an "intermediate method", gertSF() in our case, must also be annotated with the "throws clause".

Ex2.java

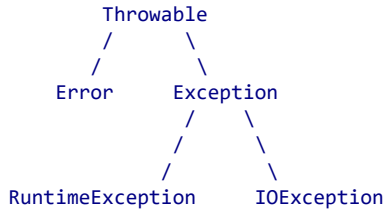
```
public class Ex2
{
  public static int getSF(String[] av) throws Throwable
  {
    String[] A = av[0].split(",");
    int[] B = new int[A.length];
    for(int i = 0; i < A.length; i++)
      B[i] = Integer.parseInt(A[i]);
    return SpecialFunc.compute(B);
  }
```

```
  public static void main(String[] args)
  {
    try {
      System.out.println(getSF(args));
    } catch(Throwable e)
    {
      System.out.println("Bad stuff happened!");
    }
  }
}
```

The Exception hierarchy — OOP in action

So that's the basic mechanism for error-handling in Java: throw and try-catch. (What could be more American that throwing and catching?) Here, however, where the object-oriented design case study comes in. We identified three problems we needed to address in an error handling mechanism, and so far we've only addressed point 1: we can deal with the fact that errors occur in one function, but need to be remediated in some other, possibly distant, function. But what about point 2, being able to distinguish between different kinds of errors when needed (and only when needed)? What about point 3, being able to deliver different kinds of data about the error to the point in the code that would actually remediate?

The answer to this is simple: derive different classes from Throwable that correspond to different kinds of errors. In fact, the API already has a whole class hierarchy rooted from Throwable and, of course, as we create our own exception classes we expand that hierarchy. To understand exception handling in Java, we really need to understand at least a part of that hierarchy.



What's important here is that the compiler does not require methods to handle (i.e. either catch or re-throw) Error's or RuntimeException's. Everything else *must* be handled!

How does a hierarchy like this solve our other two problems? Based on the type of exception we catch we can determine categories of errors that occurred. Using polymorphism, different Throwable sub-types, can present us with information specific to their sub-type, but through the uniform interface of methods available in type Throwable.

Catching different kinds of Exceptions

A given try block can actually have multiple catch's, each differing in the type they are trying to catch. Of course, all types are ultimately Throwable's, so what happens is the catches are tried top-to-bottom, and the first type that matches has its catch-block (and only its catch block) executed. This allows us to, for example, offer error messages that are actually informative!

Ex2.java

```
public class Ex2
{
  public static int getSF(String[] av) throws Throwable
  {
    String[] A = av[0].split(",");
    int[] B = new int[A.length];
    for(int i = 0; i < A.length; i++)
      B[i] = Integer.parseInt(A[i]);
    return SpecialFunc.compute(B);
  }

  public static void main(String[] args)
  {
    try {
      System.out.println(getSF(args));
    }
    catch (ArrayIndexOutOfBoundsException e)
    {
      System.out.println("Error! An argument is required.");
    }
    catch (NumberFormatException e)
    {
      System.out.println("Error! Argument included a non-integer value.");
    }
    catch(Throwable e)
    {
      System.out.println("Bad stuff happened!");
```

```
    }
  }
}
```

If we run this, we get output like the following:

```
~/$ java Ex2
Error! An argument is required.
~/$ java Ex2 1,df
Error! Argument included a non-integer value.
$ java Ex2 1,0
Bad stuff happened!
~/$ java Ex2 ,
0
~/$ java Ex2 -1
Bad stuff happened!
```

So we've got some errors covered with nice explanantions. Others no. Next class we'll use polymorphism and we'll define our own exceptions, all in the name of handling our errors nicely!
Note: can you tell me what would've happened if we'd put the "catch(Throwable e)" first instead of last?