PHAS3459: Scientific Programming Using Object Oriented Languages

Module 3: Exceptions

Dr. B. Waugh and Dr. S. Jolly
Department of Physics & Astronomy, UCL

Contents

1. Aims of Module 3

- Understand how to structure programs to handle errors.
- Be able to:
    - implement `try`, `catch` and `finally` structures;
    - throw Exceptions in your code;
    - declare Exceptions in your method/class definitions.

2. Introduction to Exceptions

One difference between average code and good code is how robust the code is against unexpected conditions. For example the program may be presented with nonsensical numerical values, or an expected input file may be missing, or a call to an external resource such as a database may fail. No matter how well you design your system to minimise such occurrences, they *will* occur. Java has an inbuilt mechanism for dealing with the unexpected. Indeed, the Java programmer is forced to use this mechanism. It is called *exception handling*.

The basic idea behind exception handling is that the unexpected (*exceptional*) conditions can be flagged, and thereby brought to the attention of the *calling code*, the higher level routine that needs to deal with that condition. For example, suppose a physics data analysis program accesses calibration constants from a database. The database access fails because the database server is offline. Then you might want the program to do one of several things, including:

- try connecting to an alternative database server that should be able to deliver the same calibration constants;
- assume a certain reasonable set of default calibration constants and continue with the data analysis;
- terminate, alerting the user to the problem.

This is a very realistic example encountered often in scientific programming. There is no general right answer for what to do: it depends on the details of the analysis being performed. Exception handling is the mechanism that allows the programmer to deal with such situations in a controlled way, by fixing the cause of the exception if possible, ensuring the exception does no damage further along in the program execution, or simply logging the presence of the error.

You have already met several cases where error handling would have been useful, for example:

- in the `ThreeVector` class, when the angle with respect to input vector (0,0,0) is requested to be calculated;
- in the `MyCounter` class, when we try and set the counter beyond the maximum allowed value;
- in the `Complex` class, when division by a complex number of zero magnitude is requested.

When a method detects such an unexpected condition, and the programmer wishes this to be signalled to the calling code, the action taken is to *"throw an exception"* which is then *"caught"* by the higher-level method.

2.1 A Simple Example

The following code fragment illustrates how a method can throw an exception:

```
public static int process(int j) throws Exception {
  if (j==13) {
    throw new Exception("Unlucky number: "+j);
  }
  return 2*j;
}
```

The first thing to note is that the `Exception` is created with the `new` operator in exactly the same way as any other object. Indeed in Java exceptions are just objects. There are different kinds of exception objects, related to each other through *inheritance*. One can extend the set by defining one's own exception classes. We will revisit this topic in the module on inheritance.

In order to call this method we must enclose the calling code in a `try` block as follows:

```
for (int k=10; k<20; k++) {
  try {
    int n = process(k);
    System.out.println("in: "+k+"  out: "+n);

  }
  catch (Exception e) {
    System.out.println(e);
  }
}
```

We *try* to perform this action, but if something goes wrong the method we call may *throw* an exception, and we have to be prepared to *catch* it. Note that if a method throws an exception, it does not need to return a value.

2.2 "`try, catch, finally`" Structures

Sometimes there may be some action that must be taken regardless of whether an exception is thrown. For example, when reading data from a file (which we will do in a later module) we may encounter errors along the way but we still have to ensure the file is closed afterwards. The general format of the structure is as follows:

```
try {
  // Code that can throw an exception
  // ...
}
catch (Exception e) {
  // Code that handles the exception
  // ...
}
finally {
  // Code that executes regardless
  // of whether an exception is
  // thrown or not
}
```

For example, modifying the simple example given above:

```
for (int k=10; k<20; k++) {
  try {
    int n = process(k);
    System.out.println("in: "+k+"  out: "+n);
  }
  catch (Exception e) {
    System.out.println(e);
  }
  finally {
    System.out.println("done "+k);
  }
}
```

2.3 Checked and Runtime Exceptions

The exceptions we have seen so far are called *checked* exceptions. The Java compiler forces us to declare when a method may throw an exception using the `throws` keyword:

```
public void doSomething() throws Exception { }
```

We are also forced to deal with the possible exception in some way when calling the method: the compiler will generate an error if we do not put the call within a `try` block or if there is no matching `catch` or `finally` block.

Checked exceptions are used extensively in the classes of the Java Runtime Environment, especially in those dealing with input and output of data, which is why we are learning about them now. They should be used in any software that has to deal with “exceptional” situations that arise when interacting with the world outside the program.

There is another type of exception, the *runtime exception*, represented by the `RuntimeException` class, which can be used in a much simpler way. It need not be declared with `throws` so it is often hard to know whether a given method might throw a runtime exception without detailed examination of the source code. A method that can throw a runtime exception may be called without enclosing the calling code in a `try` block.

This makes runtime exceptions simpler to use, but it does mean that they bring a risk that

the program may crash when it is run. This is in fact the very reason they exist: they are intended for cases where a problem arises because of a programming error rather than some external input, and where there is no reasonable action the program can take to recover the situation. In this case letting the program terminate with some explanatory message may be the best we can do. Try running the following example and study the resulting error message:

```
public static void main(String[] args) {
  for (int k=10; k<20; k++) {
    int n = process(k);
    System.out.println("in: "+k+"  out: "+n);
  }
}

public static int process(int j) throws RuntimeException {
  if (j==13) {
    throw new RuntimeException("Unlucky number: "+j);
  }
  return 2*j;
}
```

3. Using Exceptions

3.1 Where to Catch Exceptions

It is good practice to separately handle all the places in the code where an exception could be thrown, rather than implement a “catch-all” case. For example the following code cannot discern the actual function call producing the unexpected result:

```
double va = 1.0;
double vb = 0.0;
double vd = 0.0;
try {
  Aclass aa = new Aclass();
  vb = aa.somefunction1();
  double vc = Arithmetic.divide(va,vb);
  vd = aa.somefunction2();
  double e = Arithmetic.divide(va,vd);
}
catch (Exception e){
  System.out.println("Not sure whether somefunction1() "+
                "or somefunction2() produced the zero !");
}
double vf = vb*vd;
```

This code would be better written with an explicit check after each method call that could result in an exception being thrown:

```
double va = 1.0;
double vb = 0.0;
double vd = 0.0;
Aclass aa = new Aclass();
try {
  vb = aa.somefunction1();
  double vc = Arithmetic.divide(va,vb);
}
catch (Exception e){
  System.out.println("somefunction1() is producing zeros");
}
try {
  vd = aa.somefunction2()
  double ve = Arithmetic.divide(va,vd);
}
catch (Exception e){
  System.out.println("somefunction2() is producing zeros");
}
double vf = vb*vd;
```

Of course, one should not separate out code into different `try-catch` blocks that is interdependent. Consider the following example:

```
double va = 1.0;
double vb = 0.0;
double vc = 0.0;
double vd = 0.0;
Aclass aa = new Aclass();
try {
  vb = aa.somefunction1();
  vc = Arithmetic.divide(va,vb);
}
catch (Exception e){
  System.out.println("somefunction1() is producing zeros");
}
try {
  vd = aa.somefunction2()
  double ve = Arithmetic.divide(vc,vd);
}
catch (Exception e){
  System.out.println("somefunction2() is producing zeros");
}
double vf = vb*vd;
```

What happens in the first `try-catch` block of the method `aa.somefunction1()` returns zero or `NaN`? This will cause the `Arithmetic.divide` method to throw an exception. But the code within the second `try-catch` block, which depends on the result of the calculation of `vc`, will still execute as the exception from the first `try-catch` block has already been caught! At this point, we don't want the calculation of `double ve = Arithmetic.divide(vc,vd)` to be carried out, since it's expecting the value of `vc` to be non-zero (or at least to have been calculated correctly in the first `try-catch` block).

As such, code that is interdependent should be placed within the same `try-catch` block. If some part of the code depends on some previous code executing successfully, it shouldn't execute if the previous code throws an exception.

Something else that should be avoided is the use of *nested* exceptions:

```
double va = 1.0;
double vb = 0.0;
double vd = 0.0;
Aclass aa = new Aclass();
try {
  vb = aa.somefunction1();
  double vc = Arithmetic.divide(va,vb);
  try {
    vd = aa.somefunction2()
    double ve = Arithmetic.divide(va,vd);
  }
  catch (Exception e){
    System.out.println("somefunction2() is producing zeros");
  }
}
catch (Exception e){
  System.out.println("somefunction1() is producing zeros");
}
double vf = vb*vd;
```

This makes the code more difficult to follow, whilst decreasing the utility of the exception handling, particularly if several layers of nesting are used. Proper programming practice is, firstly, to make sure that the error messages displayed when the exception is thrown are suitably clear to allow the error to be identified. Then, if possible, split the code into separate `try-catch` blocks and catch each exception separately.

3.2 Variable Scope

What happens if, in the previous code fragments, the variables "`vb`" and "`vd`" were not declared as doubles right at the top, but rather inside the `try` block were they are used? It turns out that the Java compiler would complain about the final line "`double vf = vb*vd;`". This is an example of "variable scope". Essentially, the "scope", or range of code for which a variable is defined, is defined by the nearest enclosing curly brackets "`{}`". In the examples above, the variables "`vc`" and "`ve`" are *only* defined in their respective `try` blocks. By virtue of being declared up-front, the variables "`vb`" and "`vd`" are available to the whole code fragment including the last line, and they merely have their values redefined in the `try` blocks.

This trick of declaring variables up-front to widen their scope applies also to user-defined types. However there is not necessarily a well defined "initial value" for objects of these types. We can get around this problem by declaring the variables to be "`null`" as in the following pseudo-code example:

```
// set object's value to a
// "dummy" value
MyCounter xx = null;

try {
  xx = new MyCounter(150.0);
  xx.incrementCounter();
}
catch (Exception e){
  System.out.println("ERROR:"+e.getMessage());
}
try {

  // If the constructor failed above,
  // xx will still be null, and the
  // next line will cause an exception
  // ...

  xx.print();

  // Note that if the constructor
  // did not fail, yet the call to
  // "incrementCounter" failed, the
  // object xx will still be non-null
  // and further methods can be
  // called without causing a
  // NullPointerException.
}
catch (NullPointerException e){
  // ... which will be caught here.
  System.out.println("ERROR: xx is not defined; it is still null");
}
```

This example also shows the use of the specific exception class "`NullPointerException`" which is thrown when an attempt is made to call a method of an object that is still "`null`", either through not having been set or through constructor failure as above.

3.3 Throwing More Than One Exception

Here is some example code showing two different exception types that could be thrown by some code. Don't worry about the details of the file input/output code which will be covered in future modules:

```
public static int getIntegerFromFile(String filename)
  throws java.io.FileNotFoundException, java.io.IOException
{
  // If filename does not exist
  // then this will throw a
  // java.io.FileNotFoundException

  FileReader f = new FileReader(filename);

  // If file exists we continue

  // We could have a problem reading
  // the file in which case read()
  // will throw a java.io.IOException

  int x = f.read();

  // if all is well we will return
  // to the calling code

  return x;
}
```

Calling code might catch these exceptions as follows:

```
try {
  int x = getIntegerFromFile("run1_data");
}
catch (FileNotFoundException e1){
  System.out.println("The file DID NOT exist ");
}
catch (IOException e2) {
  System.out.println("File opened OK but we "+
      "could not read it - the error is: "+e2);
}
finally {
  // Code to close the file
}
```

3.4 Delegation

It can be a complicated issue deciding what action to take in the event of an exception being thrown. In general it's best to deal with errors as close as possible to where they occur. Nonetheless, it's sometimes only in higher level calling code that the full contextual information is available to take appropriate action upon receipt of an exception. In order to facilitate this, it's possible to “re-throw” or “delegate upwards”, as in the following example:

```
  public static void mycode() throws MyException {
  try {
    int x = getIntegerFromFile("run1_data");
    int y = getIntegerFromDatabase("oracle");
  }
  catch (DataBaseException e1){
    throw new MyException("DB ERROR");
  }
  catch (FileNotFoundException e2){
    throw new MyException("FILE ERROR");
  }
  catch (IOException e3) {
    throw new MyException("FILE ERROR");
  }
}
```

You can see that the `catch` blocks do nothing other than throw an exception themselves. Note that in this example they happen to throw a different type of (user-defined) exception. The type of exception thrown forms part of the declaration of the "`mycode()`" method, as we have already seen above. The higher level handling code might look something like this:

```
  try {
    boolean using_database = false;
    boolean using_file = true;
    MyClass.mycode();
  }
  catch (MyException e) {
    if (e.getMessage().equals("FILE ERROR") && using_file) {
      System.out.println("WE ARE IN BIG TROUBLE");
    }
    else {
      System.out.println("A PROBLEM BUT WE DO NOT CARE...")
    }
    // etc .. for the database case
```

Note that the method "`getMessage()`" is defined for all exception types to return a string, and this string is compared to literal strings referring to specific error messages in the above example using the "`equals()`" operator. This will be revisited in future Modules.

A crucial point is that all exceptions other than those derived from `RuntimeException` ***must*** be handled somewhere in the program as a whole. Failure to do so will generate a compile-time error.

4. Summary

Dealing effectively with errors is critical in designing robust computer programs. Java provides an integral framework — **Exception Handling** — for dealing with errors in a controlled way. In this Module we have met most of the key concepts necessary to begin introducing exception handling into your programs.