COMP 401

Prasun Dewan[1]

19. Exceptions

So far, we have been fairly lax about error handling in that we have either ignored errors, or used a simple-minded approach of terminating the program on encountering the first error. Moreover, error handling code was mixed with regular programming code. Furthermore, error messages to the user, which belong to the user-interface, were mixed with computation code. We will see how these problems can be overcome by using exceptions.

User Arguments & Exception Handling

Consider the following program, which prints the first argument provided by the user.

```
package main;
public class AnArgPrinter{
   public static void main(String args[])    {
      System.out.println(args[0]);
   }
}
```

Figure 1: Printing a User Argument

A problem with this program is that the user may have forgotten to supply an argument when executing the program. Thus, the value, 0, provided as the index to *args* may be out of bounds. Of course this is not what the program expects, so it does perform correctly if the expected argument is supplied.

So what will happen if the unexpected case happens? Some compilers, including Java, will automatically put code in the program to check if array subscripts are out of bounds, and will terminate the program, mentioning that the problem was a subscripting error. Other, less friendly, but more efficient compilers do not bother to do so. The program will compute an appropriate memory address where the element is expected, and print whatever value is at that memory location. If the memory address is outside the range of memory allocated to the program, then it will dump core, giving no hint of the problem.

None of these solutions are friendly to the users if the program, who either get garbage printed out; or are told about an subscripting error, whose cause they probably cannot correlate with the actual error they committed; or are given a core dump message. Thus, it is important for the programmer to detect the error and react appropriately to it.

We can, of course, use a conditional statement to explicitly check for the error:

```
if (args.length == 0 ) {
        System.out.println("Did not specify the argument to be printed. Terminating program.");
System.exit(-1);
} else {
   System.out.println(args[0]);
 }
```

However, this solution mixes code handling of both the expected and exceptional cases, making the program harder to understand and write. The alternative code given in Figure 12 reduces this problem:

```
    try   {
       System.out.println(args[0]);
    }
    catch (ArrayIndexOutOfBoundsException e) {
        System.out.println("Did not specify the argument to be printed. Terminating program.");
System.exit(-1);
    }
```

Figure 2: Handling Exceptions

It relies on the fact that the Java compiler does insert subscript -checking code in the program. It uses three related concepts: *exceptions*, *throw bocks*, and *catch blocks*, which we saw before when looking at user input. When the code detects a subscripting error, it "throws" an exception "out" of the *try* block, which is then "caught" by the catch block written by the programmer. More precisely, it causes the program to jump out of the try block enclosing the statement that threw the exception, and execute the catch block, passing it an instance of the class ArrayOutOfBoundsException. The catch block declares the class of the exception it expects to handle, and receives as an argument any exception of this class that is thrown in the try block preceding it. It can react appropriately to the exception. In this example it prints for the user an error message indicating the cause of the problem.

In comparison to the previous solution, this solution has the advantage that the expected and exceptional cases are clearly delineated, making it possible to implement and understand the code in two passes. In the first pass we can worry about the excepted cases and later, in the second pass, we can address the exceptional cases. There are several other advantages of exceptions, which we will see when we study them in more detail.

Separating error detection and handling

Consider the following code

```
package main;

import java.io.BufferedReader;
import java.io.InputStreamReader;
import java.io.IOException;

public  class AnArgsPrinter {

        static BufferedReader inputStream = new BufferedReader(new InputStreamReader(System.in));

public static void main (String args[])  {
   echoLines(numberOfInputLines(args));
}

static int numberOfInputLines(String[] args) {
   try {
      return Integer.parseInt(args[0]);
   } catch (ArrayIndexOutOfBoundsException e) {
      System.out.println("Did not enter an argument.")
      return  0;
    }
}

static void echoLines (int numberOfInputLines) {
   try {
      for (int inputNum = 0; inputNum < numberOfInputLines; inputNum++)
         System.out.println(inputStream.readLine());
   } catch (IOException e) {
      System.out.println("Did not input " + numberOfInputLines +  " input strings
before input was closed. ");
      System.exit(-1);
   }
}
```

}:

It is an extension of the previous code that echoes multiple lines. The number of lines to be input is specified as the first argument. As before, if this argument is not specified, a subscript exception is thrown, and the user is given an error message. Recall that a user can close input by entering an EOF (End of File) indicator. If a program tries to read a closed input stream, Java throws an IOException. In this example, such an exception would be thrown if the user closes input before entering the expected lines of input. Therefore the catch block of this exception gives an appropriate error message.

Though this example uses exception handlers, it still is not perfectly satisfactory. The function, getNumberOfInputLines() must provide a return value and so it chooses an arbitrary value, which cannot be distinguished from a value actually entered by the user. Moreover, the function is responsible for both computing (the number of lines) and user-interface issues. Error messages really belong to the user interface as there are multiple ways to report errors. In our example, they could be reported in the system transcript or a dialogue box. Also echoLines() and getNumberOfLines() make different decisions regarding what to do with an error – echoLines() halts the program while getNumberOfLines() decides to return to the caller. If they have been written independently, such non uniformity was to be expected. More important, neither of them has enough context to make the right decision.

The problem seems to be that error detection and handling are coupled here, that is, done by one method. The solution is to handle errors, not in these two methods, but in the main method. The two methods are the ones that detect the errors, so they need a way to communicate the detected errors to the main method. One way to do so is pass back error codes to the main method. This would work fo a procedure - we can pass back an error code instead of no value. However, it will not always work for a function as it may not be possible to distinguish a legal value from an error value. For example, an integer function that can compute any integer cannot pass back an error value. Another alternative may be to write to common variables shared by the caller and callee, but this also has problems. Other methods that have access to the scope in which the variables are declared have access to them, violating the least privilege principle. More problematic, if there are multiple recursive calls to the same function, one call may override the value written by another call.

Fortunately, Java provides a more elegant solution in which exceptions values can be “returned” to the caller, and its called, and so on, until some method in the call chain decided to process them. In general, a method does not need to catch all exceptions thrown by statements executed by it. Java lets the caller of the method catch all of these exceptions that are not caught by the method, and lets its caller catch the exceptions it ignores, and so on. Thus, exceptions are *propagated* through the call chain until they are caught. There are two reasons for this:

- *Calling Context*: It is often the callers who know what to do with an exception, since they often have more context, so propagating the exception to the callers is a good idea. In fact, it is in such cases that

exceptions are really useful, since a method does not have to explicitly pass back error codes to its caller. (Of course, in the case of a main method, it is a bad idea to ignore the exception, since if main does not catch the exception, its caller, the interpreter, has to catch it, and it cannot give a meaningful error message.)

- *Return Values*: If an exception occurs in a function, and there is no legal value to return, then a good way to inform the caller that there is no return value is to propagate the exception to it.

If a method does not catch an exception, the can header can have a *throws clause* acknowledging the exception, that is, indicating that it is not catching this exception and, thus, implicitly throwing the exception to its caller:

```
static void echoLines (int numberOfInputLines)  throws IOException {
      for (int inputNum = 0; inputNum < numberOfInputLines; inputNum++)
         System.out.println(inputStream.readLine());
}

         static int numberOfInputLines(String[] args) throws ArrayIndexOutOfBoundsException {
      return Integer.parseInt(args[0]);
}
```

In our example, the main method now catches these exceptions:

```
  public static void main (String args[])  {
            try {
              echoLines(numberOfInputLines(args));
            } catch (ArrayIndexOutOfBoundsException e) {
                  System.out.println("Did not enter an argument. Assuming a single input line.”);
                   echoLines(1);
            } catch (IOException e) {
                  System.out.println("Did not input the correct number of input strings before input was closed.
           ");
            }
    }
```

When an exception is thrown by either the echoLines() or numberOfInputLines() methods, the try block in which the call to the method was made is terminated and a catch block associated with the try block is executed. As this code shows, it is possible to associate a try block with multiple catch blocks for different kinds of exceptions. When an exception is thrown, the catch block matching the exception is executed.

This code shows the benefits of propagating exceptions. The main method does a more intelligent handling of the missing argument error, passing a default value of 1 to echoLines(). The function did not know that is return value was to be passed to echoLines() and thus did not have the context to provide this flexible error handling. Moreover, the function did not have to worry about what value to return in case of an error. Finally, previously, the function was responsible for both computing (the number of lines) and user-interface issues. In the new version, it is responsible only for computation and reporting an exception. How the exception is translated into an error message is the responsibility of another piece of code – the main method.Thus, we have written code that meets all of the objections raised earlier.

What if the main method above did not catch these exceptions? In this case, the method would be terminated and the exceptions thrown to the caller of the method. We can document this in the declaration of the main method:

```
public static void main (String args[])  throws IOException, ArrayIndexOutOfBoundsException {
   echoLines(numberOfInputLines(args));

 }
```

The caller of main is the interpreter. If an exception is thrown to the interpreter, it simply prints the names of the exceptions and the current stack. Passing the buck to the interpreter is a bad idea as the interpreter's output may be meaningless to the user.

Nested catch blocks

As it turns out, the code for main given above is wrong, as echolines(1), invoked in the catch block for ArrayIndexOutOfBoundsException can throw an IOException. The solution is to have a try-catch block around it, giving rise to nested catch blocks as shown below:

```
  public static void main (String args[])  {
          try {
                echoLines(numberOfInputLines(args));
          } catch (ArrayIndexOutOfBoundsException e) {
               System.out.println("Did not enter an argument. Assuming a single input line.");
                try {
                     echoLines(1);
                } catch (IOException ioe) {
                      System.out.println("Did not input one input string, which is the default in case of missing
          argument, before input was closed. ");
                }
          } catch (IOException e) {
               System.out.println("Did not input the correct number of input strings before input was closed.
          ");
          }
```

```
        }
```

We must make sure that exceptions names in the nested catch blocks are different, otherwise Java will complain. This is why we have given them separate names, e and ioe.

Checked vs. Unchecked Exceptions

What if we omit the throws clause in echoLines():

```
static void echoLines (int numberOfInputLines)  {
      for (int inputNum = 0; inputNum < numberOfInputLines; inputNum++)
         System.out.println(inputStream.readLine());
}
```

Interestingly, Java will not let us get away if we do not do acknowledge the exception propagated to the caller. Why this insistence? It is better documentation since it tells the caller of the method that it must be prepared to handle this exception.

On the other hand, it will let us get away with not acknowledging the subscript error exception:

```
static int numberOfInputLines(String[] args) {
      return Integer.parseInt(args[0]);
}
```

Thus, the method above neither catches nor acknowledges throwing of the exception, and yet is legal

Why this non-uniformity? Imagine having to list ArrayIndexOutOfBoundsException in the throws clause of every method that indexes an array! The reason why this is unacceptable is that just because a method performs an operation that can throw an exception does not mean that an exception may actually be thrown. For instance, many if not most methods that index arrays probably will never access an out of bound subscript. It is misleading to require these methods to either catch the exception or acknowledge them in the throws class, as in the case below.

```
 static void safeArrayIndexer throws ArrayIndexOutOfBoundsException () {
   String args[] = {“hello”, “goodbye”};
   System.out.println(args[1]);
}
```

However, Java cannot determine if an operation that can throw an exception will ever do so in a particular method – it can only determine which operations are called by a method. It is for this reason that Java does not require ArrayIndexOutOfBoundsException to be caught or listed in a *throws* clause.

And yet, it does do so in the case of IOException! What is the fundamental difference between these two kinds of exceptions?

To understand the difference[2], it might be useful to understand the origin of exceptions. Why are exceptions thrown, that is, what are the different kinds of unexpected events that a program might face? We can divide these events into two categories:

- *Unexpected User Input*: Users did not follow the rules we expected them to follow. This is the reason for both kinds of exceptions in the example above.
- *Unexpected Internal Inconsistency*: The program has an internal inconsistency, which was detected during program execution. For instance, a method might have created an array of size 1 rather than 2, and an exception would be raised if a caller of the procedure tries to access the first element of the array (without checking its length, which is expected to be 2), as shown below.

```
public static void unsafeArrayIndexer () {

   String args[] = {“hello”};

   System.out.println(args[1]);

}
```

However, acknowledging an uncaught exception in the header is misleading if such an internal inconsistency does not occur.

Based on this division, we can formulate the following exception rule:

Exception Documentation Rule: Exceptions thrown because of unexpected user input that are not caught by a method should be acknowledged in the header so that the caller of the method can easily determine what exceptions it must handle. This rule does not apply to exceptions thrown because of unexpected internal inconsistency.

Relatively speaking, we should expect internal inconsistencies to be rare but unexpected user input to be common. Moreover, the former can be avoided by careful programming, whereas the latter cannot be controlled by the programmer. Therefore, not requiring uncaught potential internal inconsistencies to be listed in the throws clause is fine, since it is probably a false alarm for most programs and hurts the careful programmer who does not write inconsistent code.

Java does not know which exceptions are thrown because of unexpected user input and which because of internal inconsistency. It does, however, define two classes of exceptions: those that have the class *RunTimeException* as a superclass and those that do not. ArrayIndexOutOfBounds exception is an example of the former and IOException is an example of the latter. It allows uncaught “runtime” exceptions to be not declared in the throws clause while requiring uncaught non-runtime exceptions to be declared in the

[2] What follows is what I believe is the reason for the difference. It is not necessarily the reason used by the designers of Java.

throws clause. The term “runtime” is in quotes and misleading since all exceptions are thrown at runtime (i.e. while the program is running).

These Java rules are consistent with the exception documentation rule as long as *runtime exceptions= internal Inconsistency*, that is, an exception thrown because of an internal inconsistency is a runtime exception and an unexpected user input a non-runtime exception. However, this is not always the case, as our example shows, where unexpected use input (a missing argument) causes a runtime exception. In such a case, we can voluntarily catch or list in the throws clause those runtime exceptions that are actually thrown because of unexpected user input, as we did when we put the array indexing exception in the throw class. However, there is no way to force every caller of the method in which the exception is first thrown that also does not handle the exception to also voluntarily list it. Another alternative, which fixes this problem, is to convert an uncaught runtime exception thrown by unexpected user input into an appropriate non-runtime exception, as shown below:

```
static int numberOfInputLines(String[] args) throws IOException {
try {
      return Integer.parseInt(args[0]);
} catch (ArrayIndexOutOfBoundsException e) {
     throw new IOException();
}
```

The throw statement used here must be followed by an instance of an exception class. It has the effect of throwing the exception at the point in the program. Now Java will force every caller that does not handle it to acknowledge it. Moreover, the conversion alternative provides a better name for the missing argument exception, which can considered a special form of an I/O exception that has to do with user input entered when the program is started rather than when it is executing.

Let us sum up the discussion on “runtime” vs. “non runtime” exceptions, which we will call unchecked vs. checked exception (the checking is for acknowledgement in a header of uncaught exceptions).

In the case of unchecked exceptions there are no rules relating method bodies to headers. On the other hand, in the case of checked exceptions, the following rule holds true:

uncaught in method body => acknowledged in method header

We can also think of the rule as:

unacknowledged in method header => caught in method body

Normally, the first form of the rule is useful, since we first decide in the method whether we want to catch it or not and then decide what the header should be. But the second form is useful when the header is constrained by an interface.

What are the rules relating throws clauses of corresponding method headers in interfaces and classes?

For checked exceptions, the following matching rule holds true:

unacknowledged in interface method-header => unacknowledged in class method-header

Together, these two rules can be used to force a method body to catch a checked exception:

unacknowledged in interface method-header => unacknowledged in class method-header

unacknowledged in class method-header => caught in class method-body

To illustrate, consider the following interface method:

```
public interface StringIterator {
        ...
        public String next();
}
```

The following implementation of it is legal:

```
public class AnInputStreamScanner implements StringIterator {
        ...
        public String next() {
                try {
                        return inputStream.readLine();
                } catch (IOException e) {
                        return “”;
                }
        }
}
```

However, the following implementation is not:

```
public class AnInputStreamScanner implements StringIterator {
        ...
        public String next()  throws IOException{
                return inputStream.readLine();
        }
}
```

as the header in the class acknowledges the exception while the corresponding header in the interface does not.

On the other hand the following implementation is legal:

```
public class AnInputStreamScanner implements StringIterator {
        ...
        public String next()  throws java.util.NoSuchElementException {
                return inputStream.readLine();

        }
}
```

The reason is that NoSuchElementException is an unchecked exception. For such an exception, just as there is no rule to match method corresponding method bodies and headers , there is also no rule to match corresponding interface and class method headers. Without the first rule, the second rule is useless as it cannot force the programmer to catch an exception in the body of the method.

It is a good idea for an enumerator to throw some exception if next() is called when there are no more elements. As we see above, Java provides the default class, java.util.NoSuchElementException, for such exceptions. Since most programs can be expected to check if there are more elements before requesting the next element, this is correctly an unchecked exception. In the special case when the next element could not be determined because of a user error, throwing a checked exception makes sense. It is possible for the method, thus, to throw both exceptions to handle user errors and empty Iterator cases:

```
public String next()  throws java.util.NoSuchElementException, java.io.IOException {
                return inputStream.readLine();

}
```

Printing exceptions and stack traces

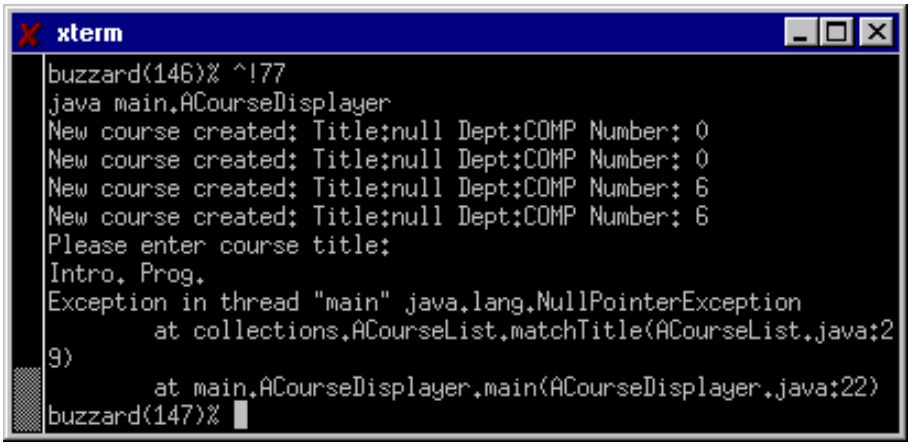
When an exception corresponding to an internal error occurs, the user debugging it may wish to find out details about the exception. It is possible to print an exception:

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

which simply prints a message associated with the exception, which by default in the class of the exception. It is also possible to print the exception class, message, and a trace of the stack using the *printStackTrace()* instance method defined in class Exception:

```
catch (ArrayIndexOutOfBoundsException e) {
    e.printStackTrace();
}
```

There are two possible stacks that could be of interest – the stack that existed when the exception was thrown or the current stack. As this stack is retrieved from the exception object, which is not modified as it is propagated through various method calls, the former is printed, as shown below:



Here, the exception is caught by the interpreter, which is able to print the stack that existed when it was thrown.

It is not very useful to print the stack trace, exception class, or the exception message for a user exception, as the user is not expected to know how user errors are mapped to internal exceptions.

Exceptions in Functions

Consider the following implementation of numberOfInputLines:

```
static int numberOfInputLines(String[] args) {
  try {
    return Integer.parseInt(args[0]);
  } catch (ArrayIndexOutOfBoundsException e) {
```

```
      System.out.println("Did not enter an argument.")
    }
}
```

Java will complain that no value is returned by the catch block. This is the reason we used the following implementation, which returns a value in both the try and catch block.

```
static int numberOfInputLines(String[] args) {
   try {
      return Integer.parseInt(args[0]);
   } catch (ArrayIndexOutOfBoundsException e) {
      System.out.println("Did not enter an argument.")
      return  0;
    }
}
```

However, the following is legal:

```
static int numberOfInputLines(String[] args)  {
   try {
      return Integer.parseInt(args[0]);
   } catch (ArrayIndexOutOfBoundsException e) {
      System.out.println("Did not enter an argument.") ;
      throw e;
    }
}
```

Here the catch block simply throws the exception it caught after doing a partial processing of it. To understand these examples, let us look at the general rules governing an exception thrown in a function. There are two cases:

- *Exception completely handled*: The function catches the exception without throwing any other exception. In this case, the caller of the function believes there was no exception, and expects a return value. For instance, in the first variation of the function, the call to numberOfInputLines() expects a return value. Thus, in this case, it is the responsibility of the exception handler in the function to return some value.
- *Exception thrown*: The method either does not catch the exception or partially handles it and throws another one in response to the original exception. In this case, the exception gets propagated to the caller, which is an indication to it that there is no legal return value. For instance, in the third version of the function above, the statement containing the call to the function is terminated and control transfers to the catch block in the caller, if the call was made in a try block. If the call was not made in a try block, the statement containing the function call will still be terminated, and control transferred to the calling procedure, and so on. Thus, in this case, the implementation of the function does not, and in fact, cannot, return an object to the caller.

Thus, when an exception occurs in a method, if you cannot think of a valid value to return to a function, simply throw an exception to the caller, even if you have handled the exception by, say, printing an error message. As we saw above, a function cannot throw an arbitrary exception.

Defining New Exception Classes

Let us go again at the following variation of numberOfInputLines:

```
static int numberOfInputLines(String[] args) throws IOException {
try {
      return Integer.parseInt(args[0]);
} catch (ArrayIndexOutOfBoundsException e) {
     throw new IOException();
}
```

Recall that we converted an ArrayIndexBoundsException to an IOException to make it more meaningful for the caller. Unfortunately, now the caller cannot distinguish between the two exceptions thrown by the methods it calls in its try block since the echoLines() method also throws this exception:

```
static void echoLines (int numberOfInputLines)  throws IOException {
      for (int inputNum = 0; inputNum < numberOfInputLines; inputNum++)
         System.out.println(inputStream.readLine());
}
```

One way to overcome this problem is to give a string argument to the constructor used to instantiate the converted exception:

```
static int numberOfInputLines(String[] args) throws IOException {
try {
      return Integer.parseInt(args[0]);
} catch (ArrayIndexOutOfBoundsException e) {
     throw new IOException("Missing First Argument");
}
```

This solution would work well if all the caller of this procedure was simply printing the stack trace when it caught the exception. One of the functions of *printStackTrace*() is to print the message passed to the constructor of the exception. This string is retrieved by calling the *getMessage*() instance method of the exception. Thus, *printStackTrace*() will produce different output for the two IOExceptions it catches.

However, this solution does not work so well if the caller wants to perform different actions for the two exceptions, such as exiting the program in one case and continuing execution in the other. The program can call the *getMessage()* method of the two exceptions to differentiate between them, but using a string to "type" an exception is not the best solution. It requires a character-by-character comparison, and more importantly, it is error prone, since we can erroneously compare the wrong string, or more likely, change

the string in the called method without changing it in the calling method. Java can provide us no help in catching these errors.

Therefore, a more direct and safe method is to define a new exception class that accurately reflects the cause of the error:

```
public AMissingArgumentException extends IOException {
    public AMissingArgumentException (String message) {
            super(message);
    }
}
```

In this declaration, we create a subclass of IOException that inherits the full behavior of its superclass. Its constructor simply calls the corresponding constructor of its superclass, which can be expected to initialize an appropriate instance variable. This variable can later be accessed by the inherited *getMessage()* method, which in turn is invoked by *printStackTrace()* and *println().* When declaring a new exception class, we are free to override any of these methods. We could have used a different exception class as the superclass of this new exception. In particular, we could have used the class *Exception,* which is the superclass of all exceptions. It is important to create a new exception class as a subclass of an existing exception class – otherwise Java does not know it is an exception class and will not let us throw instances of it.

The new exception is thrown in the code below:

```
static int numberOfInputLines(String[] args) throws IOException {

try {

      return Integer.parseInt(args[0]);

} catch (ArrayIndexOutOfBoundsException e) {

     throw new AMissingArgumentException ("Missing First Argument");

}
```

and caught in the code below:

```
public static void main (String args[])  {
 try {
   echoLines(numberOfInputLines(args));
 } catch (AMissingArgumentException e) {
       System.out.println(e); // prints e.getMessage()
       echoLines(1);
 } catch (IOException e) {
```

```
      System.out.println("Did not input the correct number of input strings before input was closed. ");
}
   }
```

As we see above, the caller can easily differentiate between the two exceptions:

Catching Multiple Exception Classes

Defining a new exception class for a special kind of exception of an existing class may seem more cumbersome if we do not want to provide special treatment for the exception. Suppose in both cases, we wanted to simply print the exception message and exit abnormally:

```
try {
   echoLines(numberOfInputLines(args));
} catch (AMissingArgumentException e) {
       System.out.println(e);
       System.exit(-1);
} catch (IOException e) {
      System.out.println(e);
      System.exit(-1);
}
```

Here, we have duplicated the exception handling code. Thus this solution requires more programmer effort and is error-prone since we may change the code in one handler and not the other.

As it turns out, even though exceptions of two different classes are now thrown by the called method, we can create a single handler for both exceptions:

```
try {
   echoLines(numberOfInputLines(args));
} catch (IOException e) {
      System.out.println(e);
      System.exit(-1);
}
```

The reason is that AMissingArgunmentException is a subclass of IOException. Java allows a handler (that is, a catch block) for an exception type, T1, to handle exceptions of every type T2 that IS-A T1, for the same reasons it allows assignment of values of T2 to variables/method parameters declared as T1.

These rules allow the same exception to be processed by more than one exception handler. For instance, in the code with duplicated handlers, both handlers can process an instance of AMissingArgumentException. When an exception occurs in a try block, Java chooses the first handler in the catch list attached to the try block. Thus, in our example, it chooses the first catch block for AMissingArgumentException.

However, if we had reversed the positions of the two handlers, and these two handlers did different things, we would not get the desired behavior:

```
try {
   echoLines(numberOfInputLines(args));
 } catch (IOException e) {
       System.out.println(e);
       System.exit(-1);
} catch (AMissingArgumentException e) {
        System.out.println(e);
}
 }
```

Figure 3: Unreachable Catch Block

Both exceptions would be processed by the first catch block. Thus, handlers of more specific exception types should be listed before those of more general types. Fortunately, Java will give us an error message if we do not do so. Ideally, it should automatically find the most specific handler, but it does not do that.

Truth in Advertisement of Checked Exceptions

Consider the following interface and its implementation:

```
public interface StringIterator {
        public boolean hasNext();
        public String next() throws Exception;
}

public class AnInputStreamScanner implements StringIterator {
        ...
        public String next()  throws IOException{
                return inputStream.readLine();
        }
}
```

The exceptions acknowledged by the next() headers in the class and interface do not match. Thus, apparently, there is no truth in advertisement in this case. Should Java complain?

To answer this question, let us consider the consequences of not being truthful here on a user of the interface method that (directly or indirectly) handles the exception thrown by it , such as the one shown below:

```
void print (StringIterator stringIterator) {
try {
        while (stringIterator.hasNext())
                System.out.println(stringIterator.next());
} catch (Exception e) {…}
}
```

Because the interface advertises an instance of Exception, the user must either define a handler for this class (or its superclass, which does not exist in this case as Exception is the root class of all exception classes), as shown above. If the actual exception thrown is IOException, this handler will still be able to process it because of the rules of assignment. Thus, there is no harm in letting the interface header acknowledge an exception class that is more general than the one acknowledged by the class header. Acknowleding a more general class than the one thrown is a stronger advertisement as it claims that the method throws a larger set of exceptions. In the example above, it says that the method can throw any exception. A stronger advertisement is allowed as it ensures that the actual exception will be handled.

Similarly, consider the following interface and class:

```
public interface StringIterator {
        ...
        public String next() throws IOException;
}

public class AnInputStreamScanner implements StringIterator {
        ...
        public String next() {
                try {
                        return inputStream.readLine();
                } catch (IOException e) {
                        return “”;
        }
}
```

Again, the two headers do not match, but again, this is fine as the interface advertises a larger set of exceptions than the one thrown, which is the null set in this example. The handler of a caller of the interface method must catch the IOException:

```
void print (StringIterator stringIterator) {
try {
```

```
        while (stringIterator.hasNext())
                System.out.println(stringIterator.next());
} catch (IOException e) {…}
}
```

In this case, the catch block will be never executed, which is Ok. The danger we want to avoid is the absence of a catch block to handle an exception.

This can happen in the following interface/class pair:

```
public interface StringIterator {
        public boolean hasNext();
        public String next() throws IOException;
}

public class AnInputStreamScanner implements StringIterator {
        ...
        public String next()  throws Exception {
                return  inputStream.readLine() + … ;
        }
}
```

Here the set of exceptions acknowledged by the interface header is smaller than the one thrown. A legal user of the interface such as:

```
void print (StringIterator stringIterator) {
        try {
                while (stringIterator.hasNext())
                        System.out.println(stringIterator.next());
        } catch (IOException e) {…}
}
```

would not be able to process the larger set, which in this case includes all exceptions.

A similar problem can occur if the interface does not acknowledge exceptions when it should be:

```
public interface StringIterator {
        public boolean hasNext();
        public String next();
}

public class AnInputStreamScanner implements StringIterator {
        ...
        public String next()  throws IOException{
```

```
                return inputStream.readLine();
        }
}
```

A legal user of the interface handler is allowed to not have a try catch block:

```
void print (StringIterator stringIterator) {
                while (stringIterator.hasNext())
                        System.out.println(stringIterator.next());
}
```

and thus would not be able to catch the IOException thrown by the method implementation.

So far, we have assumed that the implementatiion header advertise the (checked) exceptions that are actually thrown. What if this is not the case:

```
public interface StringIterator {
        public boolean hasNext();
        public String next() throws Exception;
}

public class AnInputStreamScanner implements StringIterator {
        ...
        public String next()  throws Exception {
                return inputStream.readLine();
        }
}
```

Here the implementation (and interface) header acknowledges the set of all exceptions, which is a super set of the IOExceptions actually thrown. Again, this is safe, as the caller can process the actual exceptions:

```
void print (StringIterator stringIterator) {
try {
        while (stringIterator.hasNext())
                System.out.println(stringIterator.next());
} catch (Exception e) {…}
}
```

However, acknowledging a smaller set is not allowed

```
public interface StringIterator {
        public boolean hasNext();
```

```
        public String next() throws AMissingArgumentException;
}

public class AnInputStreamScanner implements StringIterator {
        ...
        public String next()  throws AMissingArgumentException {
                return inputStream.readLine();
        }
}

void print (StringIterator stringIterator) {
try {
        while (stringIterator.hasNext())
                System.out.println(stringIterator.next());
} catch (AMissingArgumentException e) {…}
        }
```

In the above example, a legal user of the implementation is not able to handle the actual exception thrown.

We can now more precisely state the rules for checked exceptions:

exception of type T1 uncaught in method body => exception of type T2, where T2 IS-A T1, acknowledged in method header

exception of type T1 unacknowledged in interface method-header => exception of type T2, where T2 IS-A T1, unacknowledged in class method-header

The reason for the rules is that a caller that is capable of catching an exception of type T1 can also handle an exception of a type that IS-A T1. This is analogous to allowing a drug company to overstate but not understate the bad side effects of a drug. If the stated side effects are drowsiness and nausea, and you don't get drowsiness, you will not complain. But if the stated side effects are only nausea and you get drowsiness (while driving, for instance) then you will probably sue!

An intuitive explanation is that if you are good, you are allowed to say you are bad (in not handling exceptions in this case), as you will not disappoint people. On the other hand, if you are bad, you are not allowed to say you are good, as you may let people down.

Catching Expected Events

Consider the following code:

```
try {
  for (;;) {
     String s =  inputStream.readLine();
     process(s);
   }
} catch (IOException e) { };
```

Notice that this code does not explicitly check for EOF, using the Java exception mechanism to exit the loop. While this loop will work in that it will process user input until EOF is detected, it is misleading. It seems to indicate that encountering an EOF is an exception, whereas, EOF is actually an expected event and required to terminate the loop. In fact, not getting an EOF is an exceptional (internal inconsistency) event, since the loop will never terminate!

Thus, avoid using exception handlers to process expected events unless you are forced to do so.

Intra-Method Propagation

Consider the following two code fragments:

```
while (Iterator.hasNext())
   try {
      System.out.println((String) Iterator.next());
   } catch (ClassCastException e) { e.printStackTrace());}

try {
   while (Iterator.hasNext())
         System.out.println((String) Iterator.next());
} catch (ClassCastException e) {e.printStackTrace());}
```

Figure 4: Intra-Method Exception Propagation

We assume in both fragments that Iterator is of type Iterator. We cast each element as a value of type String, and print it in case the class cast is successful, and the stack trace in case of a ClassCastException is thrown.

What is the difference? In the first case, the try is around the println() statement, while in the second case, it is around the while. When an exception occurs in a statement, Java tries to find a handler attached to that statement. If it finds one, it terminates the statement, and executes the handler. Otherwise, it repeats the process for a statement that nests this statement. If it reaches the top-level block in the method, then, as we saw before, it propagates the exception to the caller of the method. Thus, exceptions are propagated through the nesting chain within a method, before being propagated through the calling chain. We call the former, intra-method propagation, and the latter as inter-method propagation.

So what happens above? In the first, when a ClassCastException occurs in a println, the println is terminated, its catch block executed, and the loop resumed. In the second case, the whole while loop is terminated, and the attached catch block executed.

Thus, in the first case, all elements in the Iterator are processed, while in the second one, only those elements that occur before the first non-String. The first solution is probably better in most situations since we get to see all the errors all the strings, while in the second case, we see only the first error and some of the strings.

In general, when an exception is thrown, your program should try to make as much further progress, as possible.

The amount of progress we can make depends very much on the situation. The more related the errors, the less progress you can make.

Consider your latest assignment to report errors. In this assignment, you must scan and parse the input expression. Scanning involves identifying the individual tokens, detecting which ones are illegal, while parsing involves processing sequences of tokens. Consider the following input:

5 % 2 ^ 1

If we were just scanning the input for illegal tokens, then, after finding an illegal token, we can continue the scan, looking for more illegal tokens. This is because the processing of a new token does not depend on the tokens we have seen before that.

Now consider the following input, which is free of any scanning errors:

5 + 2 4 / - 2

Suppose after scanning, we are interested in parsing it. Unfortunately, after encountering the first parsing error, we cannot really make reliable progress, since we do not know which subsequent errors depend on this error. For instance, are there are two errors (4 and /) or one error (transposition of 4 and /). Some parsers do try and be smart, but they are really guessing the user's intention, as you have seen while using compilers. So you may want to terminate parsing after the first error, but should try not do so in the case of scanning.

Summary

Exceptions can be used to force programmers to address errors, separate error handling from regular processing, and distribute error handling among various methods in the program. They are first-class objects united by a common super class. It is possible to create our own Exception classes.