CS1Bh Lecture Note 21

Exceptions

We have already met the notion of *exception* in CS1Ah, Lecture Note 29 and in several example programs and program practicals. Exceptions are used for handling error conditions in programs in a clean way. In this note we look at exceptions in more detail.[1]

21.1 What's an Exception and Why Do I Care?

The Java language uses exceptions to provide error-handling capabilities for its programs.

> **Definition:** *An exception is an event that occurs during the execution of a program, which disrupts the normal flow of instructions.*

Many kinds of error can cause exceptions—problems ranging from serious hardware errors, such as a hard disk crash, to simple programming errors, such as trying to access an out-of-bounds array element. When such an error occurs within a Java method, the method creates an exception object and hands it off to the runtime system. The exception object contains information about the exception.

- In Java terminology, creating an exception object and handing it to the runtime system is called *throwing an exception*.
- After a method throws an exception, the runtime system leaps into action to find some code to *handle* the exception. The exception handler which provides this code is said to *catch the exception*.

If the runtime system exhaustively searches all of the methods on the call stack without finding an appropriate exception handler, the runtime system (and consequently the Java program) terminates.

By using exceptions to manage errors, Java programs have many advantages over the error management techniques traditional in languages such as C. We discuss three: Separating Error Handling Code from "Regular" Code; Propagating Errors Up the Call Stack; Grouping Error Types and Error Differentiation.

[1] This note is mainly derived from the exceptions thread of Sun's Java tutorial.

Separating Error Handling Code from "Regular" Code

Without exceptions, error detection, reporting, and handling often lead to confusing spaghetti code. For example, suppose that you have a function that reads an entire file into memory. In pseudo-code, your function might look something like this:

```
readFile {
   open the file;
   determine its size;
   allocate that much memory;
   read the file into memory;
   close the file;
}
```

At first glance this function seems simple enough, but it ignores many potential errors. What happens if the file can't be opened? What happens if the length of the file can't be determined? What happens if enough memory can't be allocated? What happens if the read fails? What happens if the file can't be closed? To answer these questions within your `readFile` function, without exceptions, you'd have to add a lot of code to do error detection, reporting and handling. Your function would end up looking something like this:

```
errorCodeType readFile(file) {
    initialize errorCode = 0;
    open the file;
    if (theFileIsOpen) {
        determine the length of the file;
        if (gotTheFileLength) {
            allocate that much memory;
            if (gotEnoughMemory) {
                read the file into memory;
                if (readFailed) {
                    errorCode = -1;
                }
            } else {
                errorCode = -2;
            }
        } else {
            errorCode = -3;
        }
        close the file;
        if (theFileDidntClose && errorCode == 0) {
            errorCode = -4;
        } else {
            errorCode = errorCode and -4;
        }
    } else {
        errorCode = -5;
    }
    return errorCode;
}
```

With error detection built in, your original 7 lines have been inflated to 29 lines of code—a bloat factor of almost 400 percent. Worse, there's so much error detection, reporting, and returning that the original 7 lines of code are lost in the clutter. And worse yet, the logical flow of the code has also been lost in the clutter, making it difficult to tell if the code is doing the right thing: Is the file really being closed if the function fails to allocate enough memory? It's even more difficult to ensure that the code continues to do the right thing after you modify the function three months after writing it. Many programmers "solve" this problem by simply ignoring it—errors are "reported" when their programs crash.

Java provides an elegant solution to the problem of error management: exceptions. Exceptions enable you to write the main flow of your code and deal with the, well, exceptional cases elsewhere. With exceptions, your `readFile` function could look something like this:

```
readFile {
    try {
        open the file;
        determine its size;
        allocate that much memory;
        read the file into memory;
        close the file;
    } catch (fileOpenFailed) {
        doSomething;
    } catch (sizeDeterminationFailed) {
        doSomething;
    } catch (memoryAllocationFailed) {
        doSomething;
    } catch (readFailed) {
        doSomething;
    } catch (fileCloseFailed) {
        doSomething;
    }
}
```

Note that exceptions don't spare you the effort of doing the work of detecting, reporting, and handling errors. What exceptions do provide for you is the means to separate all the grungy details of what to do when something out-of-the-ordinary happens from the main logic of your program. In addition, the bloat factor for error management code in this program is about 250 percent–compared to 400 percent in the previous example.

Propagating Errors Up the Call Stack

Exception handlers are set up by the code that calls the code which may throw the exception. The run-time systems looks for code to handle an exception by examining the call stack which records a stack of callers waiting for the results. A second advantage of exceptions is the ability to propagate error reporting up the call stack of methods. Suppose that the `readFile` method is the fourth method in a series of nested method calls made by your main program: method1 calls method2, which calls method3, which finally calls readFile.

```
method1                          call method2;
```

Suppose also that `method1` is the only method interested in the errors that occur within `readFile`. Old-fashioned error notification techniques force `method2` and `method3` to propagate the error codes returned by `readFile` up the call stack until the error codes finally reach `method1`—the only method that is interested in them.

As you learned earlier, the Java runtime system searches backwards through the call stack to find any methods that are interested in handling a particular exception. A Java method can "duck" any exceptions thrown within it, thereby allowing a method further up the call stack to catch it. Thus only the methods that care about errors have to worry about detecting errors.

```
method1 {
    try {
            call method2;
    } catch (exception) {
            doErrorProcessing;
    }
 }

 method2 throws exception {
    call method3;
 }

 method3 throws exception {
    call readFile;
 }
```

However, as you can see from the pseudo-code, ducking an exception does require some effort on the part of the "middleman" methods. Any checked exceptions that can be thrown within a method are part of that method's public programming interface and must be specified in the throws clause of the method. Thus a method informs its callers about the exceptions that it can throw, so that the callers can intelligently and consciously decide what to do about those exceptions.

Grouping Error Types and Error Differentiation

Often exceptions fall into categories or groups. For example, you could imagine a group of exceptions, each of which represents a specific type of error that can occur when manipulating an array: the index is out of range for the size of the array, the element being inserted into the array is of the wrong type, or the element being searched for is not in the array. Furthermore, you can imagine that some methods would like to handle all exceptions that fall within a category (all array exceptions), and other methods would like to handle specific exceptions (just the invalid index exceptions, please).

Because all exceptions that are thrown within a Java program are first-class objects, grouping or categorization of exceptions is a natural outcome of the class hierarchy.

Java exceptions must be instances of `Throwable` or any `Throwable` descendant. As for other Java classes, you can create subclasses of the `Throwable` class and subclasses of your subclasses. Each "leaf" class (a class with no subclasses) represents a specific type of exception and each "node" class (a class with one or more subclasses) represents a group of related exceptions.

For example, `ArrayException` is a subclass of `Exception` (a subclass of `Throwable`) and has three subclasses: `InvalidIndexException`, `ElementTypeException`, and `NoSuchElementException`, which are all leaf classes. Each one represents a specific type of error that can occur when manipulating an array. One way a method can catch exceptions is to catch only those that are instances of a leaf class. For example, an exception handler that handles only invalid index exceptions has a catch statement like this:

```
catch (InvalidIndexException e) { . . . }
```

ArrayException is a node class and represents any error that can occur when manipulating an array object, including those errors specifically represented by one of its subclasses. A method can catch an exception based on its group or general type by specifying any of the exception's superclasses in the catch statement. For example, to catch all array exceptions regardless of their specific type, an exception handler would specify an ArrayException argument:

```
catch (ArrayException e) { . . .  }
```

This handler would catch all array exceptions including `InvalidIndexException`, `ElementTypeException`, and `NoSuchElementException`. You can find out precisely which type of exception occurred by querying the exception handler parameter `e`. You could even set up an exception handler that handles any `Exception`, with this handler:

```
catch (Exception e) { .  . .  }
```

Exception handlers that are too general, such as the one shown here, can make your code *more* error prone, by catching and handling exceptions that you didn't anticipate, and therefore are not correctly handled within the handler. We don't recommend writing general exception handlers as a rule.

As you've seen, you can create groups of exceptions and handle exceptions in a general fashion, or you can use the specific exception type to differentiate exceptions and handle exceptions in an exact fashion.

21.2 Using exceptions

Now that you understand what exceptions are and the advantages of using exceptions in your Java programs, it's time to look in more detail at how to use them, and love them. You have undoubtedly already encountered exceptions, probably in the form of an unlovable error message from the compiler like this one:

```
InputFile.java:11: Exception java.io.FileNotFoundException
must be caught, or it must be declared in the throws clause
of this method.  in = new FileReader(filename); ^
```

This message indicates that the compiler found an exception that is not being handled. The Java language requires that methods either catch or specify all "checked" exceptions that can be thrown within the scope of that method. What *checked* means is covered in the next section. If the compiler detects a method, that doesn't meet this requirement, it issues an error message like the one shown above and refuses to compile the program.

Let's look at an example in more detail and see what's going on. The `InputFile` class wraps a `FileReader` and provides a method, `getWord`, for reading a word from the current position in the reader.

```
// Note: This class won't compile, by design!
import java.io.*;

public class InputFile {

   private FileReader in;

   public InputFile(String filename) {
      in = new FileReader(filename);
   }

   public String getWord() {
      int c = 0;
      StringBuffer buf = new StringBuffer();

      while (c != -1) {
         c = in.read();
         if (Character.isWhitespace((char)c)) {
         return buf.toString();
         } else {
            buf.append((char)c);
         }
      }

      return buf.toString();
   }
}
```

The compiler prints an error message because `InputFile` creates a new `FileReader` object and uses it to open a file whose name is passed into the `FileReader` constructor, but doesn't handle, or declare that it throws, the exceptions that `FileReader` may throw.

So what should the `FileReader` do if the named file does not exist on the file system? Well, that depends on what the program using the `FileReader` wants to do. The implementers of `FileReader` have no idea what the `InputFile` class wants to do if the file does not exist. Should the `FileReader` kill the program? Should it try an alternate filename? Should it just create a file of the indicated name?

There's no possible way the `FileReader` implementers could choose a solution that would suit every user of `FileReader`. So, they chose to throw an exception. If the

file named in the argument to the `FileReader` constructor does not exist on the file system, the constructor throws a `java.io.FileNotFoundException`. By throwing an exception, `FileReader` allows the calling method to handle the error in whatever way is most appropriate for it.

As you can see from the code, the `InputFile` class completely ignores the fact that the `FileReader` constructor can throw an exception. However, the Java language requires that a method either catch or specify all checked exceptions that can be thrown within the scope of that method. Because the `InputFile` class does neither, the compiler refuses to compile the program and prints an error message

In addition to the first error message, shown above, you also see the following similar error message when you try to compile the `InputFile` class:

```
InputFile.java:15: Warning: Exception java.io.IOException
must be caught, or it must be declared in throws clause of
this method.  while ((c = in.read()) != -1) { ^
```

The `InputFile` class's `getWord` method reads from the `FileReader` that was opened in `InputFile`'s constructor. The `FileReader` read method throws a `java.io.IOException` if for some reason it can't read from the file. Again, the `InputFile` class makes no attempt to catch or specify this exception. Thus you see the second error message.

At this point, you have two options. You can either arrange to catch the exceptions within the appropriate methods in the `InputFile` class, or the `InputFile` methods can "duck" and allow other methods further up the call stack to catch them. Either way, the InputFile methods must do something, either catch or specify the exceptions, before the `InputFile` class can be compiled.

21.3 Catch or Specify!

Java requires that a method must either catch or specify all checked exceptions that can be thrown within the scope of the method. This section discusses the reasoning behind this requirement and what it means to you and your Java programs. Several terms need further explanation: "catch", "specify," "checked exceptions," and "exceptions that can be thrown within the scope of the method."

Catch A method can catch an exception by providing an exception handler for that type of exception.

Specify If a method chooses not to catch an exception, the method must specify that it can throw that exception. Why did the Java designers make this requirement? Because any exception that can be thrown by a method is really part of the method's public programming interface: callers of a method must know about the exceptions that a method can throw in order to intelligently and consciously decide what to do about those exceptions. Thus, in the method signature you specify the exceptions that the method can throw.

Checked Exceptions Java has many different types of exceptions—I/O Exceptions, runtime exceptions, and exceptions of your own creation, to name but a few. Runtime exceptions are those exceptions that occur within the Java runtime system. This includes arithmetic exceptions (such as when dividing by zero), pointer exceptions (such as trying to access an object through a null reference), and indexing exceptions (such as attempting to access an array element through an index that is too large or too small).

Runtime exceptions can occur anywhere in a program and in a typical program can be very numerous. The compiler does not *require* that you catch or specify runtime exceptions, although you can. You should satisfy yourself that your code won't raise uncaught runtime exceptions. Unfortunately the compiler can't do this for you. Checking this requires thought—in general, it's too subtle an issue for the compiler to check automatically.

Checked exceptions are exceptions that are not runtime exceptions. These *are* checked by the compiler; the compiler checks that these exceptions are caught or specified. Some consider it a loophole in Java's exception handling mechanism, that runtime exceptions aren't checked, because programmers are tempted to make all exceptions runtime exceptions. This is not recommended.

Exceptions that can be thrown within the scope of the method The statement "exceptions that can be thrown within the scope of the method" may seem obvious at first: just look for the throw statement. However, this statement includes more than just the exceptions that can be thrown directly by the method: the key is in the phrase within the scope of. This phrase includes any exception that can be thrown while the flow of control remains within the method. This includes both exceptions that are thrown directly by the method with Java's **`throw`** statement, and exceptions that are thrown indirectly by the method through calls to other methods

21.4 Handling Exceptions

This section covers the three components of an exception handler—the **`try`**, **`catch`**, and **`finally`** blocks—by showing you how to use them to write an exception handler.

The `try` Block The first step in constructing an exception handler is to enclose the statements that might throw an exception within a try block.

In general, a try block looks like this:

```
try {
   Java statements
}
```

The segment of code labelled *`Java statements`* is composed of one or more legal Java statements that could throw an exception.

The **`try`** statement governs the statements enclosed within it and defines the scope of any exception handlers associated with it. In other words, if an exception occurs within the **`try`** statement, that exception is handled by the appropriate exception handler immediately following this **`try`** statement.

A **`try`** statement must be accompanied by at least one **`catch`** block, or one **`finally`** block.

The `catch` Block(s) The try statement defines the scope of its associated exception handlers. You associate exception handlers with a try statement by providing one or more catch blocks directly after the try block:

```
try {
    . . .
} catch ( . . . ) {
    . . .
} catch ( . . . ) {
    . . .
} . . .
```

There can be no intervening code between the end of the try statement and the beginning of the first catch statement. The general form of Java's catch statement is:

```
catch (SomeThrowableObject variableName) {
    Java statements
}
```

As you can see, the catch statement requires a single formal argument. The argument to the catch statement looks like an argument declaration for a method. The argument type, `SomeThrowableObject`, declares the type of exception that the handler can handle and must be the name of a class that inherits from the `Throwable` class defined in the java.lang package. When Java programs throw an exception they are really just throwing an object, and only objects that derive from `Throwable` can be thrown. You'll learn more about throwing exceptions later in this note.

You access the instance variables and methods of exceptions in the same manner that you access the instance variables and methods of other objects. For example, `getMessage` is a method provided by the `Throwable` class that prints additional information about the error that occurred. The `Throwable` class also implements two methods for filling in and printing the contents of the execution stack when the exception occurred. Subclasses of `Throwable` can add other methods or instance variables. To find out what methods an exception implements, check its class definition and definitions for any of its ancestor classes.

The catch block body, *`Java statements`*, contains a series of legal Java statements, which may use the parameter, `variableName`, to refer to the exception caught by the handler. These statements are executed if and when the exception handler is invoked. The runtime system invokes the exception handler when the handler is the first one in the call stack whose type matches that of the exception thrown.

Generally speaking, your exception handlers should be specialized to catch particular exception classes. Handlers that can catch most or all exceptions are typically useless for error recovery, because the handler has to determine what type of exception occurred anyway to determine the best recovery strategy. Also, exception handlers that are too general can make code *more* error prone by catching and handling, inappropriately, exceptions that weren't anticipated by the programmer and for which the handler was not intended.

The `finally` Block The final step in setting up an exception handler is providing a mechanism for cleaning up the state of the method before (possibly) allowing control to be passed to a different part of the program. You do this by enclosing the cleanup code within a finally block, which follows any catch blocks.

Here is a typical example of a **`finally`** block. It cleans up and closes a `PrintWriter`.

```
finally {
    if (out != null) {
        System.out.println("Closing PrintWriter");
        out.close();
    } else {
        System.out.println("PrintWriter not open");
    }
}
```

A **`finally`** block is often useful for such housekeeping. *However* control leaves a **`try`** block, whether by normal termination of the block, or because an exception is thrown, the **`finally`** block is always executed.

Specifying the Exceptions Thrown by a Method

If it is not appropriate for your method to catch and handle an exception thrown by a method that it calls, or if your method itself throws its own exception, you must specify in the method signature that the method throws the exception. The **`throws`** clause goes after the method name and argument list and before the curly bracket that defines the scope of the method.

21.5 Throwing Exceptions

The `throw` Statement Before you can catch an exception, some Java code somewhere must throw one. Any Java code can throw an exception: your code, code from a package written by someone else (such as the packages that come with the Java development environment), or the Java runtime system. Regardless of who (or what) throws the exception, it's always thrown with the Java throw statement. Here's an example of a throw statement:

```
throw someThrowableObject;
```

The throw statement requires a single argument: a throwable object. In the Java system, throwable objects are instances of any subclass of the `Throwable` class. If you attempt to throw an object that is not throwable, the compiler refuses to compile your program and displays an error message.

Creating Your Own Exception Classes As you have undoubtedly noticed in your travels (travails?) through the Java language, the packages that ship with the Java development environment provide numerous exception classes. All of these classes are descendants of the Throwable class and allow programs to differentiate between the

various types of exceptions that can occur during the execution of a Java program. You can create your own exception classes, as subclasses of `Exception`, to represent problems that can occur within the classes that you write.

Runtime Exceptions Although Java requires that methods catch or specify checked exceptions, they do not have to catch or specify runtime exceptions, that is, exceptions that occur within the Java runtime system. Because catching or specifying an exception is extra work, programmers may be tempted to write code that throws only runtime exceptions and therefore doesn't have to catch or specify them. This is "exception abuse" and is not recommended.

Why did the Java designers decide to force a method to specify all uncaught checked exceptions that can be thrown within its scope? Because any exception that can be thrown by a method is really part of the method's public programming interface: callers of a method must know about the exceptions that a method can throw in order to intelligently and conciously decide what to do about those exceptions. The exceptions that a method can throw are as much a part of that method's programming interface as its parameters and return value.

Checked exceptions represent useful information about the operation of a legally specified request that the caller may have had no control over and that the caller needs to be informed about–for example, the file system is now full, or the remote end has closed the connection, or the access privileges don't allow this action.

What does it buy you if you throw a RuntimeException or create a subclass of RuntimeException just because you don't want to deal with specifying it? Simply, you get the ability to throw an exception without specifying that you do so. In other words, it is a way to avoid documenting the exceptions that a method can throw. When is this good? Well, when is it ever good to avoid documenting a method's behavior? The answer is "hardly ever."

A Rule of Thumb Do not throw a runtime exception or create a subclass of RuntimeException simply because you don't want to be bothered with specifying the exceptions your methods can throw.

Create new exception classes as classes that extend `Exception`. Here's an example:

```
public class ParseException extends Exception{
   public String token;

   ParseException(String failed) {
      super("Parse failed with string" + failed);
      token = failed;
   }
}
```

Calling the superclass constructor in this way provides a string that can be accessed by the `getMessage` method.

21.6 Summary

- Exceptions are a language feature which provide a clean way of dealing with out-of-the-ordinary circumstances in program code.
- Exceptions may be *checked* or *unchecked*. The Java compiler enforces a **catch-or-specify** rule for checked exceptions. If a method might throw a checked exception `MyException`, then the exception must either be
  - *caught* inside the method, or
  - *specified* by writing **`throws`** `MyException` in the method header.

  The **`throws`** clause indicates to clients of the method that invoking the method may trigger the specified exception(s).
- Exceptions in Java are objects which are subclasses of `Throwable`. A new exception is created, capturing a snapshot of the callstack, by constructing an object of an exception class, *e.g.* by writing **`new`** `MyException();`.
- The `Throwable` class has three special subclasses which partition the Exceptions hierarchy:

Class	*Purpose*	*Examples*
`Exception`	ordinary checked exceptions	`FileNotFoundException` `FontFormatException`
`RuntimeException`	common exceptions (unchecked)	`ArrayIndexOutOfBoundsException` `NullPointerException`
`Error`	usually-fatal problems (unchecked)	`AssertionError` `LinkeageError`

  In fact, `RuntimeException` is a subclass of `Exception`. The examples in the third column are further subclasses of each class. Programmers usually create new subclasses of `Exception`.
- An exception is actually triggered by executing a **`throw`** statement, as **`throw`** `exn;` where `exn` is an exception object (typically created with **`new`** as above).
- To **`catch`** one or more exceptions and take remedial action, the **`try`**–**`catch`**–**`finally`** statement form is used. There may be zero or more **`catch`** blocks, and the **`finally`** clause is optional. The **`finally`** block is used for clean-up which may be necessary however a piece of code exits (*e.g.* closing files).
- Each **`catch`** block binds an exception name, *e.g.* **`catch`** `(MyException e)`, which names the exception which was thrown, as `e`, if the block is executed.
- If an exception object is thrown, the nearest **`catch`** block in the call-stack which catches that exception's class (or a superclass) will be executed. Exceptions propagate up the call-stack; exceptions which are never caught may terminate the Java Virtual Machine with an error message.

Note originally prepared by Michael Fourman, David Aspinall.
Murray Cole, 2000/10/16 17:10:35.