JAVA REVIEW: EXCEPTIONS

17-Sep-15

Errors and Exceptions

- An error is a bug in your program
  - dividing by zero
  - going outside the bounds of an array
  - trying to use a null reference
- An exception is a problem whose cause is outside your program
  - trying to open a file that isn’t there
  - running out of memory

What to do about errors and exceptions

- An error is a bug in your program
  - It should be *fixed*
- An exception is a problem that your program may encounter
  - The source of the problem is outside your program
  - An exception is not the "normal" case, *but…*
  - …your program must be prepared to deal with it
- This is not a formal distinction–it isn't always clear whether something should be an error or an exception

Dealing with exceptions

- Most exceptions arise when you are handling files
  - A needed file may be missing
  - You may not have permission to write a file
  - A file may be the wrong type
- Exceptions may also arise when you use someone else’s classes (or they use yours)
  - You might use a class incorrectly
  - Incorrect use should result in an exception

The problem with exceptions

- Here’s what you might *like* to do:
  - *open a file*
  - *read a line from the file*
- But here’s what you might *have* to do:
  - *open a file*
  - *if the file doesn’t exist, inform the user*
  - *if you don’t have permission to use the file, inform the user*
  - *if the file isn’t a text file, inform the user*
  - *read a line from the file*
  - *if you couldn’t read a line, inform the user*
  - *etc., etc.*
- All this error checking really gets in the way of understanding the code

Three approaches to error checking

1. **Ignore all but the most important errors**
   - The code is cleaner, but the program will misbehave when it encounters an unusual error
2. **Do something appropriate for every error**
   - The code is cluttered, but the program works better
   - You might still forget some error conditions
3. **Do the normal processing in one place, handle the errors in another** (this is the Java way)
   - The code is at least reasonably uncluttered
   - Java tries to ensure that you handle every error

The try statement

- Java provides a new control structure, the try statement (also called the try-catch statement) to separate "normal" code from error handling:

```
try {
    do the "normal" code, ignoring possible exceptions
}
catch (some exception) {
    handle the exception
}
catch (some other exception) {
    handle the exception
}
```

Exception handling is *not* optional

- As in other languages, *errors* usually just cause your program to crash
- Other languages leave it up to you whether you want to handle *exceptions*
  - There are a lot of sloppy programs in the world
  - It's normal for human beings to be lazy
- Java tries to *force* you to handle exceptions
  - This is sometimes a pain in the neck, *but*...
  - the result is almost always a better program

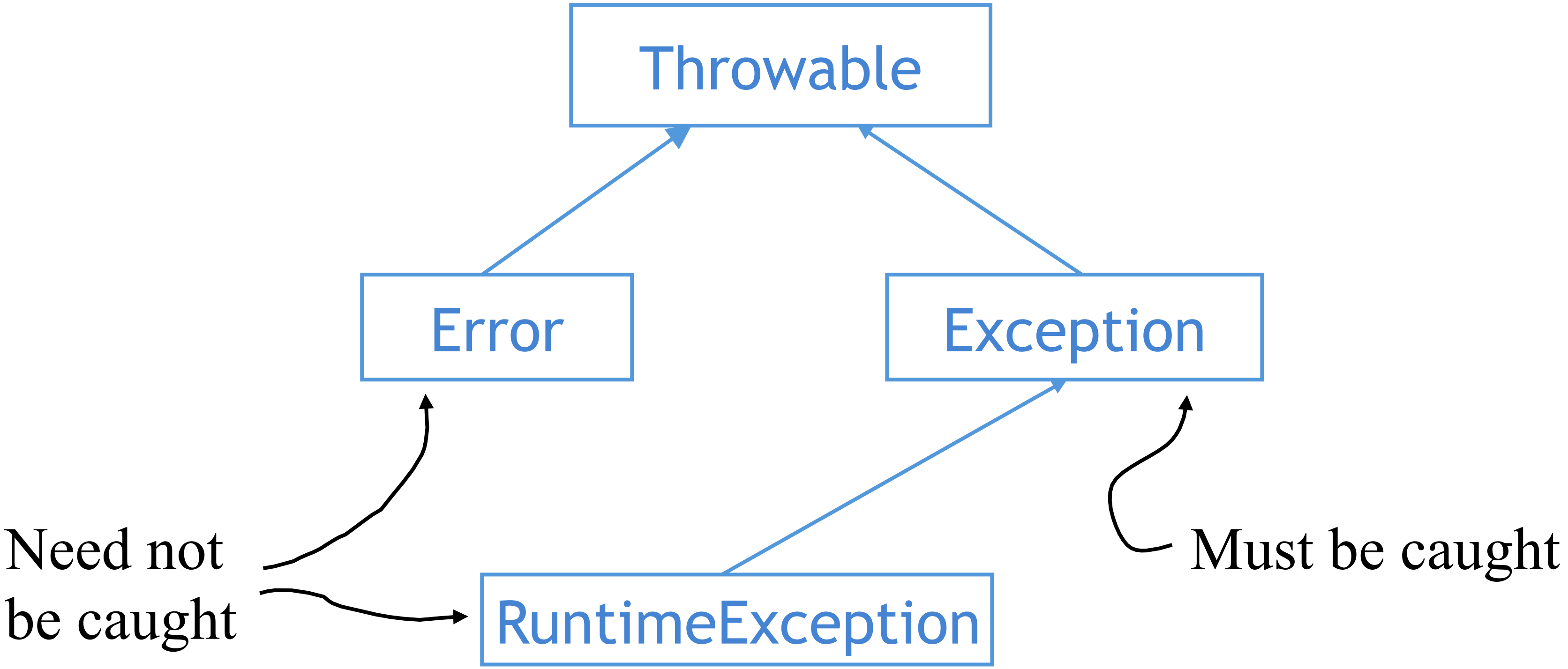
Error and Exception are Objects

- In Java, an error doesn't *necessarily* cause your program to crash
- When an *error* occurs, Java throws an Error object for you to use
  - You can catch this object to try to recover
  - You can *ignore* the error (the program will crash)
- When an *exception* occurs, Java throws an Exception object for you to use
  - You **cannot ignore** an Exception; you must catch it
  - You get a *syntax error* if you forget to take care of any possible Exception

The exception hierarchy

- Throwable: the superclass of “throwable” objects
  - Error: Usually should not be caught (instead, the bug that caused it should be fixed)
  - Exception: A problem that must be caught
    - RuntimeException: A special subclass of Exception that does *not* need to be caught
- Hence, it is the Exceptions that are most important to us (since we have to do something about them)

The Exception hierarchy II



A few kinds of Exceptions

- IOException: a problem doing input/output
  - FileNotFoundException: no such file
  - EOFException: tried to read past the End Of File
- NullPointerException: tried to use a object that was actually null (this is a RuntimeException)
- NumberFormatException: tried to convert a non-numeric String to a number (this is a RuntimeException)
- OutOfMemoryError: the program has used all available memory (this is an Error)
- There are about 200 predefined Exception types

What to do about Exceptions

- You have two choices:
  - You can "catch" the exception and deal with it
    - For Java's exceptions, this is usually the better choice
  - You can "pass the buck" and let some other part of the program deal with it
    - This is often better for exceptions that you create and throw
- Exceptions should be handled by the part of the program that is best equipped to do the right thing about them

What to do about Exceptions II

- You can catch exceptions with a `try` statement
  - When you catch an exception, you can try to repair the problem, or you can just print out information about what happened
- You can "pass the buck" by stating that the method in which the exception occurs "throws" the exception
  - Example:

    ```
    void openFile(String fileName) throws IOException { ... }
    ```
- Which of these you do depends on *whose responsibility it is* to do something about the exception
  - If the method "knows" what to do, it should do it
  - If it should really be up to the user (the method caller) to decide what to do, then "pass the buck"

How to use the try statement

- Put try {…} around any code that *might* throw an exception
  - This is a *syntax* requirement you cannot ignore
- For each Exception object that might be thrown, you must provide a catch phrase:

  catch (***exception_type name***) {…}

  - You can have as many catch phrases as you need
  - *name* is a formal parameter that holds the exception object
  - You can send messages to this object and access its fields

finally

- After all the catch phrases, you can have an *optional* finally phrase

-
  ```
  try { … }
  catch (AnExceptionType e) { … }
  catch (AnotherExceptionType e) { … }
  finally { … }
  ```
- Whatever happens in try and catch, *even if it does a return statement*, the finally code will be executed
  - If no exception occurs, the finally will be executed after the try code
  - If an exception does occur, the finally will be executed after the appropriate catch code

How the try statement works

- The code in the try {...} part is executed

- If there are no problems, the catch phrases are skipped
- If an exception occurs, the program jumps *immediately* to the first catch clause that can handle that exception

- Whether or not an exception occurred, the finally code is executed

Ordering the catch phrases

- A try can be followed by many catches
  - The first one that *can* catch the exception is the one that *will* catch the exception
- Bad:

```
catch(Exception e) { … }
catch(IOException e) { … }
```

- This is bad because IOException is a subclass of Exception, so any IOException will be handled by the *first* catch
  - The second catch phrase can never be used

Using the exception

- When you say catch(IOException e), e is a *formal parameter* of type IOException
  - A catch phrase is almost like a miniature method
  - e is an instance (object) of class IOException
  - Exception objects have methods you can use
- Here's an especially useful method that is defined for every exception type:

  e.printStackTrace();
  - This prints out what the exception was, and how you got to the statement that caused it

printStackTrace()

- printStackTrace() does *not* print on System.out, but on another stream, System.err
  - Eclipse writes this to the same Console window, but writes it in red
  - From the command line: both System.out and System.err are sent to the terminal window
- printStackTrace(***stream***) prints on the given stream
  - printStackTrace(System.out) prints on System.out, and this output is printed along with the "normal" output

Throwing an Exception

- If your method uses code that might throw an exception, and you don't want to handle the exception in this method, you can say that the method "throws" the exception
- Example:

```
String myGetLine( ) throws IOException { … }
```

- If you do this, then the method that calls this method must handle the exception

Constructing an Exception

- Exceptions are classes; you can create your own Exception with new
  - Exception types have two constructors: one with no parameters, and one with a String parameter
- You can subclass Exception to create your own exception type
  - But first, you should look through the predefined exceptions to see if there is already one that's appropriate

Throwing an Exception

- Once you create an Exception, you can throw it
  - throw new UserException("Bad data");
- You don’t *have* to throw an Exception; here’s another thing you can do with one:
  - new UserException("Bad data").printStackTrace();

Why create an Exception?

- If you are writing methods for someone else to use, you want to do something reasonable if they use your methods incorrectly
- Just doing the wrong thing isn't very friendly
- Remember, error messages are a good thing—much better than not having a clue what went wrong
  - Exceptions are even better than error messages, because they allow the user of your class to decide what to do