CSE 1720

Lecture 13

Exception Handling, Part II

Reminder

Midterm Exam

Thursday, Feb 16, 10-11:30

CLH J – Curtis Lecture Hall, Room J

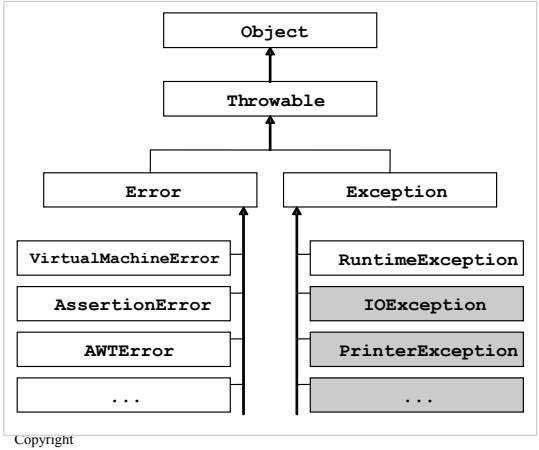
will cover all material up to and including Tues Feb 14th

- Lecture 12 (R, Feb 09) was our review and recap session
- Tues, Feb 14 –valentine's day celebration of continued coverage of the topic of exceptions

Topics

- various Q's in advance of the MT
- exception handling – sec 11.3

11.3.1 The Hierarchy



11.3.2 OO Exception Handling

- all instances of Exception inherit features from Throwable
- exceptions can be created, like any other object: Exception e = new Exception();
- exceptions can have methods invoked upon them, e.g. getMessage, printStackTrace, etc.
- all exceptions have the methods: toString, getClass
- Creating an exception does not simulate an exception. For that, use the throw keyword:

```
Exception e = new Exception("test");
throw e;
```


Example

Write an app that reads a string containing two slash-delimited integers the first of which is positive, and outputs their quotient using exception handling. Allow the user to retry indefinitely if an input is found invalid.

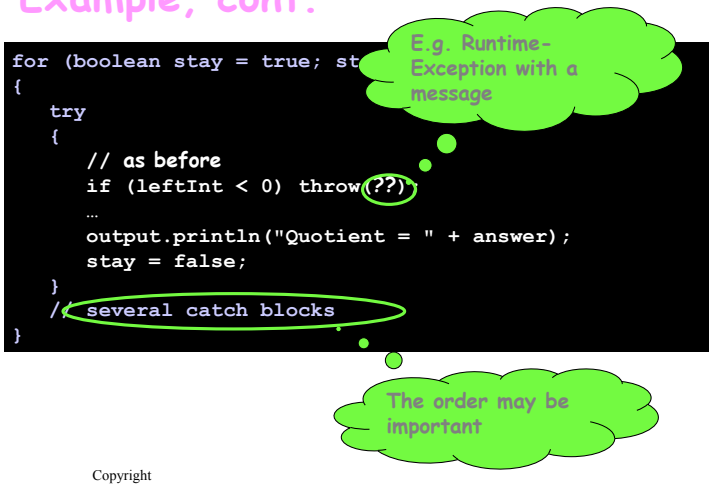
As before but:

- What if the first integer is not positive?
- How do you allow retrying?

Example, cont.

```
for (boolean stay = true; stay;)
{
   try
   {
      // as before
      if (leftInt < 0) throw(??);
      …
      output.println("Quotient = " + answer);
      stay = false;
   }
   // several catch blocks
}
```


Example, cont.



11.3.3 Checked Exceptions

- In the Exception hierarchy, there is an important distinction between
  - the branch of `RuntimeException`, and
  - the other branches, such as `IOException` and `ClassNotFoundException`
- the exceptions of type `RuntimeException`, in principle, ***can be avoided through defensive programming***
  - such exceptions can be prevented from arising in the first through careful programming
  - as such, programming language ideology suggests that app programmers should not be "forced" to handle such exceptions
    - the Java compiler does not impose any rules; if you want to use defensive programming, go knock yourself out!!!

11.3.3 Checked Exceptions

- Other exceptions such as `IOException` and `ClassNotFoundException`, however, cannot be easily validated **even in principle**
  - how can the app realistically monitor, at all times, the state of resources upon which it depends, such as network availability, media availability, the file system, the functioning of the OS and the WM ???
  - these conditions are, in essence, "un-validatable" (this is a made-up word ☺
  - as such, programming language ideology suggests that app programmers should be forced to handle such exceptions, *at least in some manner*.
  - the Java compiler ***does*** impose conditions in this case

11.3.3 Checked Exceptions

- For "un-validatable" exceptions, the compiler enforces **the acknowledgement rule**.
- The "un-validatable" exceptions are referred to as checked exceptions
  - App programmers *must* acknowledge their existence
  - The compiler ensures that the app either handles checked exceptions or use "throws" in its main.

Example

Write a program that opens a File, given a pathname, and then reads an object from that file

Hint: See L09App1 (reproduced as L12App01)

11.4 Building Robust Applications

Key points to remember:

- **Thanks to the compiler, checked exceptions are never "unexpected"; they are trapped or acknowledged**
- **Unchecked exceptions (often caused by the end user) must be avoided and/or trapped**
- **Defensive programming relies on validation to detect invalid inputs**
- **Exception-based programming relies on exceptions**
- **Both approaches can be employed in the same app**
- **Logic errors are minimized through early exposure, e.g. strong typing, assertion, etc.**