CSE 143 Java

Exception Handling

Reading: Ch. 18

Overview

- Topics
  - Exceptions (review)
  - Exception handling
  - Use of exceptions

Exceptions as Error Signals (Review)

- When we discussed programming by contract, we described how to throw an exception to indicate an error (usually an invariant being violated)

```
if (argument == null) {
    throw new NullPointerException( );
}

if (index < 0 || index > size) {
    throw new IndexOutOfBoundsException("No such item");
}
```


Exception Handling

- Idea: exceptions can represent unusual events that client could handle
  - Finite data structure is full; can't add new element
  - Attempt to open a file failed
- Problem: the object that detects the error doesn't (and probably shouldn't) know how to handle it
- Problem: the client code could handle the error, but isn't in a position to detect it
- Solution: object detecting an error throws an exception; client code catches and handles it

return vs *throw*

- *return* and *throw* both end the execution of a method
- *return*: sends control back to the point where the method was called
- *throw*: sends control back to a specially designated point, if one exists
  - can call this the "catch point"
- The return point and the catch point are never the same
  - never, ever

try-catch

- Basic syntax

```
try {
    somethingThatMightBlowUp( );
    // return point for somethingThatMightBlowUp
    additional stuff;
} catch (Exception e) {
    recovery code – e, the exception object, is a "parameter"
}
```

- Semantics (control flow)
  - Execute statements of try block
  - If an exception is thrown by a called method:
    called method terminates
    catch block executes

catch Block

- catch is executed only if an exception occurs
- catch block may contain any statements whatsoever
- Usually catch block code will:
  - handle the error so the the method can continue
  - ignore the error (risky!)
  - Re-throw the exception

try-catch-catch-catch...

- Can have several catch blocks

```
try {
    attemptToReadFile( );
} catch (FileNotFoundException e) {
    …
} catch (IOException e) {
    …
} catch (Exception e) {
    …
}
```

  - Semantics: actual exception type compared to exception parameter types in order until a compatible match is found
  - No match – exception propagates to calling method

What if There is no *try/catch*?

- Suppose called method *somethingThatMightBlowUp* is not in a try/catch block:

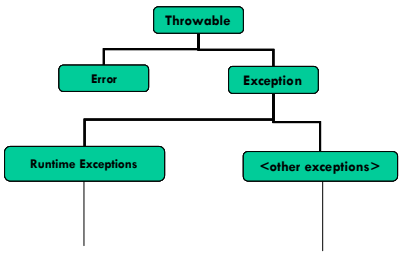
```
somethingThatMightBlowUp( );
// return point for somethingThatMightBlowUp()
additional stuff;
```

- What if it throws an exception?
  - Called method terminates
  - Calling method also terminates
  - The exception is automatically rethrown, to the method which called this one
  - If there is a catch block there, fine.
  - Otherwise, the process continues back up the chain

Exception Objects In Java

- Exceptions are regular objects in Java
  - Subclasses of the predefined Throwable/Exception
- Some predefined Java exception classes:
  - RuntimeException (a very generic kind of exception)
  - NullPointerException
  - IndexOutOfBoundsException
  - ArithmeticException (e.g. for divide by zero)
  - IllegalArgumentException (for any other kind of bad argument)
- Most exceptions have constructors that take a String argument

Throwable/Exception Hierarchy



Exceptions as Part of Method Specifications

- Generally a method must either handle an exception or declare that it can potentially throw it

```
void readSomeStuff( ) {
    try {
      readIt( );
    catch (IOException e) {
      //handle
    }
```

or

```
void readSomeStuff( ) throws IOException {  //declare
    readIt( );
}
```

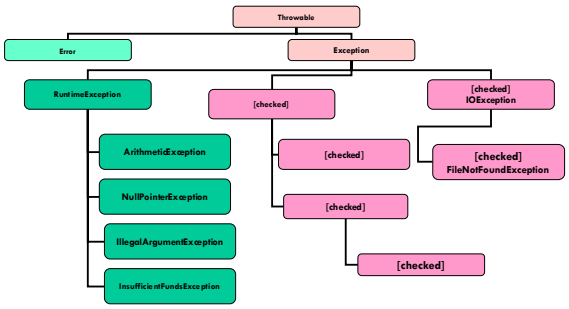
Checked vs Unchecked Exceptions

- Some types of exceptions can occur almost anywhere in any method
  - E.g. NullPointerException, IndexOutOfBoundsException, etc.
- Others are fairly specialized
  - MalformedURLException, FileNotFoundException, etc.
- Java exceptions are categorized as checked or unchecked
  - Unchecked: things like NullPointerException
  - Checked: things like IOException
- By definition:
  - unchecked exceptions are subclasses of RuntimeException
  - checked exceptions are all other exceptions
  - Footnote: The class Error is not checked

Checked vs Unchecked: The Rule

- Rule: method must either *handle* or *declare* all checked exceptions it might encounter
  - "handle" means "have a catch block for it"
  - "declare" means "have a throws clause for it"
- Do not need to declare anything about unchecked exceptions
- "Declaring an exception" means having a *throws* clause in the method header
- Compiler will enforce this

Throwable/Exception Hierarchy



Debugging Tip: Stack Traces

- Unhandled exception cause a "stack trace" to be printed
- Lists all active methods
  - First: the method where the exception occurred
  - Next: the method that called that method
  - Etc. etc.
- Stack trace entries also shows line number of each call
- Useful debugging information!

Guidelines: Exception Handling

- Intended for unusual or unanticipated conditions
  - Relatively expensive if thrown (free if not used)
  - Can lead to obfuscated code if used too much
- Guideline: Use in situations where *you* are in a position to detect an error, but *client* code would know how to react
- Guideline: Often appropriate in cases where a method's preconditions are met but the method isn't able to successfully establish postconditions (i.e., method can't do what is requested through no fault of the caller)