CSE 143 Java

Exception Handling

Reading: Ch. 15

Overview

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

Exceptions as Errors (Review)

- When we discussed programming by contract, we described how to throw an exception to indicate an error (precondition not met or other reason)

```
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 (as well as errors)
  - Finite data structure is full; can't add new element
  - Attempt to open a file failed
  - Network connection dropped in the middle of a transfer
- 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 the exception and handles it

try-catch

- Basic syntax

```
try {
    somethingThatMightBlowUp( );
} catch (Exception e) {
    recovery code – here e, the exception object, is a "parameter"
}
```

- Semantics
  - Execute try block
  - If an exception is thrown, terminate throwing method and all methods that called it, until reaching a method that catches the exception (has a catch with a matching parameter type)
  - Catch block can either process the exception, re-throw it, or throw another exception

try-catch

- Can have several catch blocks

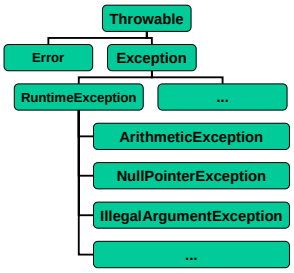
```
try {
    attemptToReadFile( );
} catch (FileNotFoundException e) {
    ...
} catch (IOException e) {
    ...
} catch (Exception e) {
    ...
}
```

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

Exception Objects In Java

- Exceptions are regular objects in Java
- Exception types must be subclasses (directly or indirectly) of the library class Throwable
- Some predefined Java exception classes:
  - RuntimeException (a very generic kind of exception)
  - NullPointerException
  - IndexOutOfBoundsException
  - ArithmeticException (e.g. integer divide by zero, etc.)
  - IllegalArgumentException (for any other kind of bad argument)
- Most exceptions have constructors that take a String argument – an error message, etc.

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 {
    readIt( );
}
```

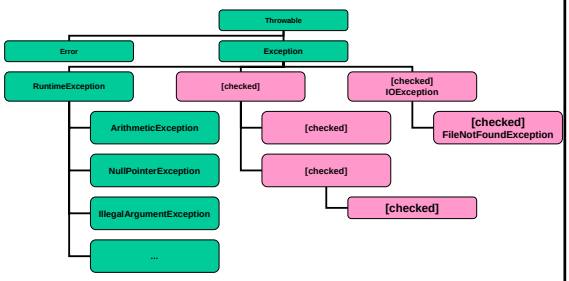

Checked vs Unchecked Exceptions (1)

- There's no point in declaring that methods can potentially throw NullPointerException, IndexOutOfBoundsException,...
  (Would wind up declaring this everywhere – useless clutter)
- Java exceptions are categorized as checked or unchecked
  - Unchecked: things like NullPointerException, ... (subclasses of RuntimeException)
  - Checked: things like IOException

Checked vs Unchecked Exceptions (2)

- Rule: a method must either handle (catch) all checked exceptions it might encounter, or declare that it might throw them
- No need to declare anything about unchecked exceptions
  - But often a good idea to declare unchecked exceptions that the method explicitly throws (e.g., IllegalArgumentException, ...) to make this part of the method documentation

Throwable/Exception Hierarchy



finally

- One last wrinkle: finally

```
try {
    ...
} catch (SomeException e) {
    ...
} catch (SomeOtherException e) {
    ...
} finally {
    ...
}
```

  - Semantics: code in the finally block is *always* executed, regardless of whether we catch an exception or not
  - Useful to guarantee execution of cleanup code no matter what

Use of 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 only 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)