CS 3
Introduction to Software Engineering
3: Exceptions
Questions?
Objectives

- Last Time: Procedural Abstraction
- This Time: Procedural Abstraction II
  Focus on Exceptions.

- Starting Next Time: Data Abstraction
What are Exceptions?

• Control-flow mechanism for unusual situations.
  – “Thrown” by methods/operators if no well-typed result makes sense.
  – “Caught” by handlers installed by callers.
  – Exception value (object) identifies the exceptional condition.

• In theory, doesn’t have to correspond to an “error”.
  – Lets method “return” to somewhere other than call site.
  – Can provide “escape hatch” for deep recursion, nested loops.
  – Textbook does this all over the place.
    But it’s a bad idea!
    • (Constructing exception objects is generally expensive.)
    • (try-catch interferes with compiler optimizations, etc.)
    • See online reading by Sestoft (2005).
Handling Exceptions

```
try
  block
catch (ExnType_1 x_1)
  catchblock_1
...
catch (ExnType_N x_N)
  catchblock_N
finally
  finallyblock
```

- If $N > 0$, `finally` part is optional.
- If `block` terminates with `exn`, execute first `catchblock_i` (if any) such that `exn` has type `ExnType_i`, binding `exn` to $x_i$.
  - If there is no such $i$, “rethrow” `exn`, possibly terminating method.
- Regardless of how `block` and/or `catchblock_i` terminates (normal, exception, `return`, `break`), execute `finallyblock`.
- (This is not a complete set of rules – look them up.)
Kinds of Exceptions

• In statement `throw e;` e must have type `java.lang.Throwable`.

• Three kinds of Throwables:
  – `java.lang.Exception`: thrown by libraries or user code.
    • `FileNotFoundException`, `MalformedURLException`.
  – `java.lang.RuntimeException`: Subclass of `Exception`, handled specially.
    • `NullPointerException`, `ArithmeticExpression`.
  – `java.lang.Error`: thrown by the JVM.
    • `NoClassDefFoundError`, `StackOverflowError`.
What’s the Difference?

• Answer two ways:
  – Treated differently by language.
  – Used differently by convention.

• Language difference?
  – Checked vs. unchecked
    – Checked = not Error, not RuntimeException.
    – Method’s throws clause must list all checked exceptions that might terminate method.
  – Whether thrown explicitly with throw or by callees
  – Exceptions handled by method don’t escape; no need to declare.
What’s the Difference?

• Answer two ways:
  – Treated differently by language.
  – Used differently by convention.

• Language difference?
  – Checked vs. unchecked.
    Checked = not Error, not RuntimeException.
  – Method’s throws clause must list all checked exns that might terminate method.
    • Whether thrown explicitly with throw or by callees.
    • Exceptions handled by method don’t escape; no need to declare.
What’s the Difference?

- **Usage differences?**

Basic Rules:
- Errors thrown by VM for severe problems.
- RuntimeExceptions signal bugs in programs.
- Things which should not have happened.
- Checked exceptions signal unusual conditions.
- Transient problems, like dropped net connections.
- User mistakes, like nonexistent filenames.

Something a caller may not expect but must prepare for.

Another Consideration:
Checked exceptions can be a nuisance. (C# doesn’t have them.)
What’s the Difference?

• Usage differences?

Basic Rules:
– Errors thrown by VM for severe problems.
– RuntimeExceptions signal bugs in programs.
  • Things which should not have happened.
– Checked Exceptions signal unusual conditions.
  • Transient problems, like dropped net connections.
  • User mistakes, like nonexistent filenames.
  Something a caller may not expect but must prepare for.

Another Consideration:
Checked exceptions can be a nuisance. (C# doesn’t have them.)
Exceptions and Abstractions

• Procedures should document all exceptions they may throw (checked or not).
  
  /** Computes the factorial of an integer.
   * @param n
   * @return the factorial of n
   * @throws IllegalArgumentException if n < 0.
   */

  int factorial(int n) {
    if (n < 0) {
      throw new IllegalArgumentException();
      ...
    }
  }
Exceptions and Abstractions

• Non-Redundancy Principle:
  If some condition is documented to cause an exception, it is NOT necessary to prohibit it with a “Requires” clause.

/** Computes the factorial of an integer.
 * Requires: n >= 0
 * @param n
 * @return n!
 */

/** Computes the factorial of an integer.
 * @param n
 * @return n!
 * @throws IllegalArgumentException if n < 0
 */
Exceptions and Abstractions

• Non-Redundancy Principle:
  If some condition is documented to cause an exception, it is NOT necessary to prohibit it with a “Requires” clause.

/** Computes the factorial of an integer.
 * <p> Requires: n >= 0
 * @param n
 * @return n!
 */

/** Computes the factorial of an integer.
 * @param n
 * @return n!
 * @throws IllegalArgumentException if n < 0
 */
Exceptions and Abstractions

• Non-Redundancy Principle:
  If some condition is documented to cause an exception, it is NOT necessary to prohibit it with a “Requires” clause.

```java
/** Computes the area of a triangle using Heron’s Formula.
 * <p> Requires: a, b and c are all positive
 * ... 
 * @return The area of a triangle with sides of lengths a, b and c.
 */

/** Computes the area of a triangle using Heron’s formula.
 * ... 
 * @return The area of a triangle with sides of lengths a, b and c.
 * @throws IllegalArgumentException if a <= 0 or b <= 0 or c <= 0.
 */
```
Exceptions and Abstractions

• Sometimes have a choice: “Requires” specification, or check and throw exception?

  • Two points of view:
    – Defensive: Shorter Requires clauses are better. Check for bad conditions and throw exceptions.
    – Passive-Aggressive: Too many exceptions make specifications confusing, too many checks make code confusing. Partial functions can be easier.

• Book (and I) favor a defensive approach:
  – Catch errors as soon as possible
  – Fail "gracefully" when something goes wrong
Exceptions and Abstractions

• Sometimes have a choice: “Requires” specification, or check and throw exception?

• Two points of view:
  – **Defensive**: Shorter Requires clauses are better. Check for bad conditions and throw exceptions.
  – “Passive-Aggressive”: Too many exceptions make specifications confusing, too many checks make code confusing. Partial functions can be easier.

  Book (and I) favor a defensive approach:
  – Catch errors as soon as possible
  – Fail gracefully when something goes wrong
Exceptions and Abstractions

• Sometimes have a choice: “Requires” specification, or check and throw exception?

• Two points of view:
  – **Defensive**: Shorter Requires clauses are better. Check for bad conditions and throw exceptions.
  – **“Passive-Aggressive”**: Too many exceptions make specifications confusing, too many checks make code confusing. Partial functions can be easier.

• Book (and I) favor a defensive approach:
  – Catch errors as soon as possible.
  – Fail “gracefully” when something goes wrong.
Local vs. Non-Local Use

• Public methods need to be as robust as possible.
  – Never trust your caller to call you right.
  – Behave “responsibly” even if caller does not.

• Not such a big deal in private methods.
  – All possible call sites are in same file!
• But still a good idea to check inputs before risky operations.
• Use Java’s assert statement.
  – E.g.: assert s != null : “String argument is null.”;
  – By default, assertions disabled, so does nothing.
  – If run with assertions enabled,
    Evaluates “s != null”. If false, throws AssertionError(“String…”)
• Never assert anything you think could possibly fail.
  Don’t try to catch AssertionError. Don’t document that you throw it.
• Assertions should be covered by Requires clause.
Assertion Demo
Propagate or Handle?

- If your procedure calls another that raises an exception, three choices:
  - Don’t catch; let it escape (add to your throws clause if checked).
  - Catch it and throw something else (“exception translation”).
    - Your spec says “throws IllegalArgumentException if string is not in table”.
    - Table lookup method throws NotFoundException.
      ```java
try {
    lookup(s);
} catch (NotFoundException nfe) {
    throw new IllegalArgumentException(s).initCause(nfe);
}
```
  - Include caught exception as “cause” of thrown exception.
    - Catch it, solve problem, caller never knows.
- Book calls first two “reflecting” and last “masking”.
Something the Book Doesn’t Say

• It’s OK to use Java’s built-in exception types if they make sense.

  /** Searches for a string in an array.
   * @param a – an array of strings
   * @param s – a string
   * @throws java.lang.NullPointerException if a == null or s == null.
   * … */

• Can avoid explicit null checks this way: Just try to use the object, and the right thing will happen.

• Other useful predefined exception types (all unchecked):
  – java.lang.IndexOutOfBoundsException
  – java.lang.IllegalArgumentException (very useful!)
  – java.lang.IllegalStateException ("you can’t do that now.")
“Failure Atomicity”

• Don’t forget: Exceptions you throw may be caught!
  – Throw exception ≠ give up and quit.
  – Caller may try again, or move on to something else.
  – That’s the whole point!
• If you must throw an exception, leave things in consistent state.
  – Try to test for bad cases before any side effects.
  – Clean up after yourself (close open files, etc.).
  – More on “consistent state” in Chapter 4.
Uncaught Exceptions

• What happens when an exception is thrown and not caught?
  Entire program crashes, with stack trace.
  (Even if there’s more than one thread.)

• Confusing twist:
  Event handlers in Swing programs:
  – Framework catches exception, prints stack trace,
  – program keeps going.
Uncaught Exceptions

• What happens when an exception is thrown and not caught?
  Entire program crashes, with stack trace.
  (Even if there’s more than one thread.)

• Confusing twist:
  Event handlers in GUI programs.
  – Framework catches exception, prints stack trace, program keeps going.
New Rule

- Code you write for CS 3 should *never* crash with an uncaught exception.
  - (Or elicit those messages from the AWT event thread.)

- Think of everything that might go wrong; Have a plan for every case!

- At the very least, inform user of problem in a civilized way before calling `System.exit(1)`.