CMSC 132: OBJECT-ORIENTED PROGRAMMING II

Program Correctness

Department of Computer Science
University of Maryland, College Park
Announcements

• We update slides/example often. Always get class material from the web site

• Remember that you can work at school computers. See additional information at
  • http://www.cs.umd.edu/eclipse/launch.html#campus

• Submit your project often so you have a copy in the submit server
  • If something happens you have a back up (in addition to the one CVS provides)
Overview

• Program correctness is determined by the presence / absence of program defects (errors)

• Issues
  • Types of program errors
    • Compile-time
    • Run-time
    • Logic
  • Testing
  • Debugging
  • Exceptions
Program Errors (Compile-Time)

• Errors in code construction
  • Lexical (typographical), grammatical, types
• Detected during compilation
• Usually easy to correct quickly
• Examples
  • Misspelled keyword
  • Missing or misplaced symbol
  • Incorrect operator for variable type
Program Errors (Run-time)

- Operations illegal / impossible to execute
- Detected during program execution
  - But not detectable at compile time
- Treated as exceptions in Java
- Examples
  - Division by zero
  - Array index out of bounds
  - Using null pointer
  - Illegal format conversion
Program Errors (Logic)

• Logic errors
  • Operations leading to incorrect program state
  • May (or may not) lead to run-time errors
  • Problem in design or implementation of algorithm
• Examples
  • Computing incorrect arithmetic value
  • Ignoring illegal input (GIGO)
• Hardest error to handle
  • Detect by testing
  • Fix by debugging
Testing

- Run program (or part of program) under controlled conditions to verify behavior
  - Detects run-time error if exception thrown
  - Detects logic error if behavior is incorrect
  - Use of debugger is extremely important

- Issues
  - Selecting test cases
    - Think of them as you develop code or before
  - Testing different parts of program
  - Test coverage
  - Others
Test Coverage

- Whether code is executed by some test case
- Automatically calculated by submit server
  - For set of tests selected (from link)
    - E.g., student tests, public tests, student+public tests
  - For conditionals, reports X/Y where
    - $X = \#$ tests executing True
    - $Y = \#$ tests executing False
- Color
  - Green = executed by some test case
  - Pink = not executed
- In the submit server you can find results by selecting “view source” in “Submissions” report
Test Coverage Example

Source Code

Coverage information for public test #all:

<table>
<thead>
<tr>
<th>Source file</th>
<th>statements</th>
<th>conditionals</th>
<th>methods</th>
<th>total</th>
</tr>
</thead>
<tbody>
<tr>
<td>Utilities.java</td>
<td>4/10</td>
<td>1/5</td>
<td>1/2</td>
<td></td>
</tr>
</tbody>
</table>

```java
package utilities;

public class Utilities {

    public static String letterGrade(double numericGrade) {
        1/1
        if (numericGrade >= 90.0)
            return "A";
        1
        else if (numericGrade >= 80.0)
            return "B";
        1/0
        else if (numericGrade >= 70.0)
            return "C";
        0/0
        else if (numericGrade >= 60.0)
            return "D";
        else
            return "F";
    }

    public static boolean passingNumericGrade(double numericGrade) {
        0/0
        return numericGrade >= 70.0 ? true : false;
    }

};
```
About Testing

• JUnit
  • Review the information available at http://www.cs.umd.edu/eclipse/JUnitTesting.html
  • Notice the problem you may experience while using static and Junit
    • http://www.cs.umd.edu/eclipse/docs/JUnitExample.zip

• Submit Server
  • In addition to coverage information, the submit server provides feedback (warnings, etc.) regarding your code. Don’t ignore them.
Exceptions (Rare Events)

- Rare event outside normal behavior of code
  - Usually a run-time error
- Examples
  - Division by zero
  - Access past end of array
  - Out of memory
  - Number input in wrong format (float vs. integer)
  - Unable to write output to file
  - Missing input file
Dealing with Exceptions

• What to do when an exception occurs?
  • Ignore the problem
  • Print error message
  • Request data
  • Exit program
  • Exit method returning error code that caller must check.

Problems:
  • Calling method may forgot to check code
  • Agreement on error codes
  • Error handling code mixed with normal code

• Use **Exception Handling** (e.g., Java’s exception mechanism)
  • Preferred!
Exception Handling Advantages

- Compiler ensures exceptions are caught eventually
- No need to explicitly propagate exception to caller
  - Backtrack to caller(s) automatically
- Class hierarchy defines meaning of exceptions
  - No need for separate definition of error codes
- Exception handling code separate & clearly marked
Representing Exceptions in Java

- Exceptions represented as
  - Objects derived from class Throwable
- Code

```java
public class Throwable extends Object {
    Throwable() // No error message
    Throwable(String mesg) // Error message
    String getMessage() // Return error msg
    void printStackTrace() { … } // Record methods
    … // called & location
}
```
Java Exceptions

- Any code that can potentially throw an exception is enclosed in a
  - `try { }` block
- Exception handlers are specified using catch
  - `catch(ExceptionType e) { }`
- You can have several catch clauses associated with a try block
- When an exception is thrown
  - Control exits the try block
  - Proceeds to closest matching exception handler after the try block
    - Java Exceptions backtracks to caller(s) until matching catch block found
  - Execute code in exception handler
  - Execute code in finally block (if present)
- **Example:** Fundamentals.java
- Scope of try is dynamic
  - Includes code executed by methods invoked in try block (and their descendents)
Java Exceptions

- **Throwing exceptions**
  - In previous example the exception was thrown for you
  - You can throw exceptions too
    - throw <Object of class exception>
  - Example:
    ```java
    throw new UnsupportedOperationException("You must implement this method.");
    ```

- **Finally block**
  - Code that is executed no matter what
    - Regardless of which catch block
    - Even if no catch block is executed
    - Executed before transferring control to caller
  - Placed after try and all catch blocks
  - Tries to restore program state to be consistent, legal (e.g., closing files)
  - Example: Finally.java
Representing Exceptions

- Java Exception class hierarchy
- Two types of exceptions ⇒ checked & unchecked
Representing Exceptions

• Java Exception class hierarchy
Checked and Uncheck Exceptions

- **Unchecked**
  - Serious errors not handled by typical program
  - Usually indicate logic errors
  - Examples → NullPointerException, IndexOutOfBoundsException
  - Catching unchecked exceptions is optional (handled by JVM if not caught)

- **Checked**
  - Errors typical program should handle
  - Used for operations prone to error
  - Examples → IOException, ClassNotFoundException
  - Compiler requires “catch or declare”
    - Catch and handle exception in method, OR
    - Declare method can throw exception, forcing calling function to catch or declare exception in turn

- **Example**: Caught.java, Declared.java
Miscellaneous

• Use exceptions only for rare events
  • Not for common cases (e.g., checking end of loop)
  • High overhead to perform catch
• Use existing Java Exceptions if possible
• Avoid simply catching & ignoring exceptions
  • `catch (Exception e) { } // Nothing in between { }`
  • Poor software development style
• An exception can be rethrown
  `catch (ExceptionType e) {
    throw e;
  }
`