Designing with Exceptions

CSE 219: Computer Science III
while (!submission deadline) {
  code
  test
  if test is not successful
    debug
}
Testing

- Tells us if and when something is wrong (in each level)
  - Acceptance testing
  - System testing
  - Integration testing
  - Unit testing
- Does not tell us how to fix it!
Debugging

● Manual process of
  ○ Understanding the error, and correcting it

● First, we locate the problem
  ○ the line number in a certain method within a certain class

● Understand why that line is wrong
  ○ then, either fix the algorithm/process
  ○ or change the process altogether
Debugging is important

... but not always easy!

- Some bugs are not obvious
  - infinite loops

- Design to test, but **don’t design to debug**
  - Don’t rely on debugging to write your code
  - Correct implementation of the algorithms/processes/methods is far more important!
Debugging is important

... but not always easy!

- Some bugs are not obvious
  - infinite loops

- Design to test, but don’t design to debug
  - Don’t rely on debugging to write your code
  - Correct implementation of the algorithms/processes/methods is far more important!
  - But if there is still a bug, identify it. That’s half the battle!
Some common bugs

- Unconstructed objects
- Uninitialized variables (*NullPointerException*)
- Forgetting to re-initialize or update a loop-variable
- Improper iteration (*ArrayIndexOutOfBoundsException*)
- Incomplete changes
  - Regression testing is good for identifying this type of error
Bugs sorted by difficulty

- Some bugs can kill ... others are like a little cold that will just bother you for a day or two
- Syntax errors are the easiest to catch (compile-time errors)
- Runtime errors are harder
  - you may not realize that an object is null until the code crashes with a NullPointerException
- Logical errors are the worst
  - the code compiles and runs, but it just does something slightly different in some situations
Making life easier (while still being a programmer)

- Murphy’s Law reigns supreme!
  - Where there’s smoke, there’s fire ⇒ *where there’s code, there’s a bug*

- Solution
  - Design and document your code
  - avoid the “will deal with this later”
    - in extreme cases, when you must delay a bugfix, at least locate the bug, and then use a “todo” or “fixme” comment temporarily so that you fix it before product release!
The brute force approach to debugging

- The one who prints everything
  - easy to implement
- Makes code very messy
- Not always useful for
  - graphical programs
  - multithreaded applications
  - web and mobile apps
Debugger

- Any modern IDE provides a powerful debugger. Use this to
  - set breakpoints (where the code will pause for you to decide what to do next)
  - You decide among several options like
    - *step into*
    - *step out*
    - *step over*
    - *pause/resume*
Robust Programs

- In code, a method is like a mathematical function:
  - has a domain (arguments)
  - has a range (output)

- **Total Methods** are methods that are defined over any input arguments in the domain. These are, by definition, robust methods.

- Otherwise, a method is a *partial* method.
  - Partial methods lead to errors (think of the `setRadius` method from last lecture, with argument type `double`).
Robust Programs

- Robust programs continue to behave reasonably even in the presence of errors
- If an error occurs, robust programs behave in a well-defined way.
  - Providing some approximation of its behavior in the absence of an error
    - This is called “graceful degradation”
  - Halt with a meaningful error message without crashing or damage to permanent data or software systems
Exceptions

● Java’s “error handlers”

● Better than returning int values like -1 because error handling can give you much more detailed information about what happened, and how (stack trace)

● The java.lang.Exception class is an abstraction
  ○ often too general
  ○ need subclasses to handle specific types of errors
Exceptions

- An exception may be thrown because a method is called
  - that throws a checked exception (e.g., `FileNotFoundException`, `IOException`)
    - You can create your own class that extends `java.lang.Exception`
  - that throws an unchecked exception due to a programming error
  - these are runtime logical errors like `ArithmeticException` or `NullPointerException`

- An error may be thrown due to an internal error in the JVM or JRE
  - Such as `VirtualMachineError` or `OutOfMemoryError`
The Error/Exception Handling Hierarchy

- Error and its subclasses are always unchecked
- Exception and its subclasses may be checked as well as unchecked
Error handling in methods

- Must be done when the preconditions of a method are not satisfied
- The type of exception must be *problem-specific*
  - Don’t just throw Exception for everything.
  - Makes it very difficult to understand what exactly went wrong.
- Exceptions can have messages
  - Always provide as detailed a message as possible
  - Your future debugging self will thank you!
Checked and Unchecked Exceptions

- Checked exceptions are checked at compile time
- If some code throws a checked exception, then a method using that code must handle the exception using `try-catch`, or throw it further (using the `throws` keyword)
Checked and Unchecked Exceptions

- **Always use checked exceptions**
  
  - Other programmers (and your future-self) are aware of potential errors
  
  - Easier to anticipate potential errors
  
  - Easier to handle errors as deemed fit in an application

- **Why are there unchecked exceptions at all?**
  
  - Read more about it here: [http://docs.oracle.com/javase/tutorial/essential/exceptions/runtime.html](http://docs.oracle.com/javase/tutorial/essential/exceptions/runtime.html)