Throwing Exceptions

What is an Exception?

Chapter 11, Section 4

"When a program violates the semantic constraints of the Java programming language, the Java Virtual Machine signals this error to the program as an *exception*."

- While the JVM is interpreting your bytecode, things can go wrong
  - e.g. array bounds exceeded, divide by zero, etc.
- The JVM creates an exception and "throws" that exception
  - It may or may not be "caught"

Typical Exceptions…

- null pointer exceptions
- array bounds exceptions
- illegal argument exceptions
- arithmetic exceptions (e.g. divide by zero)
- illegal casting exceptions

Throwing Exceptions

throw *exception*;

- *exception* is an expression that resolves to a reference to an object which is in (or is derived from) the "Throwable" class
- The throw statement always stops the current execution flow
  - Even if you throw null (throws NullPointerException)
  - Even if your throw expression has a run-time problem

```
if (arg<0) throw new IllegalArgumentExpression(
    "Argument must be a positive number");
```

Creating a new Exception Object

- Look in the Java library for an appropriate exception class
  - There are hundreds and hundreds of options out there
- Create a new exception using the "new" keyword
  - Exceptions have a constructor with a single "String message" parameter Use that one… the others are for more complicated cases
  - The message string should describe what went wrong
- The "program stack" is captured by the exception when the object is created
  - Program stack tells which instruction you are running (see next slide)
  - Therefore, almost always "throw new…"

Program Stack Example

- Java invokes TestShapes main method
  - TestShapes.main invokes Rectangle.toString()
    - Rectangle.toString invokes super.toString which is Shape.toString
      - Shape.toString throws an exception

- Program Stack:
  - Shape.java:16 – null pointer exception
  - Rectangle.java:20 – invocation of super.toString()
  - TestShape.java:15 – implicit invocation of Rectangle.toString()

Uncaught Exceptions

- If an exception is thrown and not caught the program ends
- A message is printed that contains:
  - The name of the exception
  - The "message" associated with the exception
  - The program stack

```
Exception in thread "main" java.lang.NullPointerException
        at inherShapes.Shape.toString(Shape.java:16)
        at inherShapes.Rectangle.toString(Rectangle.java:20)
        at java.base/java.lang.String.valueOf(String.java:2951)
        at inherShapes.TestShapes.main(TestShapes.java:15).
```

Why throw exceptions?

- If you determine that your program cannot continue correctly, throw an exception
  - Causes program to end
  - Gives information to the user about what went wrong

- Alternative: every method returns a return code
  - If the method worked, return a good return code
  - If the method did not work, return a bad return code
  - Whenever that method is invoked, if return code is bad, handle it
  - Lots of work for things that (almost) never happen

Exceptions v. Input Verification

- Exceptions are not very user friendly
  - Program halts
  - Prints out lines of source code, which user has no clue about
- Don't use exceptions for user input verification
  - Verify user input in your own code
  - Make user friendly messages if check fails
  - Recover or exit gracefully from failed checks
- Use exceptions for programming errors

Overdoing Exceptions

- If something is truly exceptional, let Java handle it
- Avoid the temptation of coding for every possible contingency
- Consider throwing exceptions when assumptions are violated
  - IllegalArgumentException if the argument to your function is bad
  - Unsupported Method exception for abstract methods you are too lazy to implement
  - etc.