Recursion
Testing with JUnit

CIS 36a, Lecture 12.2

April 15, 2015

Nathaniel Titterton

Questions?

Recursion

Recursion

An algorithmic technique where a function, in order to accomplish a task, calls itself with some part of the task.

Using recursive procedures

- Everyone thinks it's hard!
  - (well, it is… aha!-hard, not complicated-hard)

- Using repetition and loops to find answers

All recursion procedures need...

1. Base Case (s)
    - Where the problem is simple enough to be solved directly
2. Recursive Cases (s)
    1. Divide the Problem
        - into one or more smaller problems
    2. Invoke the function
        - Have it call itself recursively on each smaller part
    3. Combine the solutions
        - Combine each subpart into a solution for the whole

An example

```
// Find the number of characters in a nonnull
// (though possibly empty) string.

public static int charCount (String s) {
   if (s.equals("")) {
      return 0;
   } else {
      return 1 + charCount (s.substring(1));
   }
}
```

Recursion as cloning

- A recursive call inside a method essentially creates a "clone" of the method.
  - The clone has new arguments; if they haven't changed, the recursion is probably going to be infinite.
  - When the clone is executed, it may set up another clone with a copy of its arguments.
    This is recursion!

Tree Recursion

- Linear recursion is used to process a sequence (e.g., ArrayList)
  - At each step, one recursive call is made
  - Generally not useful in Java (or Python), where there are many ways to *iterate*
- In tree recursion, many recursive calls can be made at once.

File System

- Directories that can contain directories or files can be considered a tree.
- Recursively processing a directory (say, to list files) means recursing down any number of directories at any point.

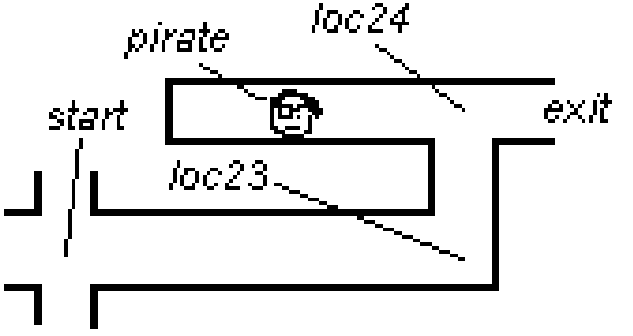
Counting or listing possibilities

- making Kindergarten words
  - given a list of consonants and a list of vowels, list all C-V-C words that can be made

- Regular expressions are a canonical example of tree recursion.

Solving A Maze

- For instance:
  - A maze consists of *locations*
  - A location contains *directions* which can be taken to reveal another location, or a special condition
  - **`walk(Location l, Direction d)`** →
    - a Location
    - the exit (a special location)
    - a deadend (or a pirate)



Finding exits

- Combining booleans is a common recursive technique

```
boolean leadToExit(Location l, Direction d)
```

  - first, walk that direction
  - if you find the exit, return **`True`**
  - if you find a deadend, return **`False`**
  - Otherwise, "and" the results to taking all the directions from the resulting location.

- Could this go on forever?

Testing with JUnit

JUnit is common testing framework

- A *unit* is a module, or chunk of functionality.
- A *suite* is a set of units.

- Generally, you write a test class for every class in your program.
- And, you write tests for each method, and often more (each "path" through your code)
- Together, you have a suite of tests you can quickly apply.

How many paths?

```
if ( something ) {
   // do stuff
   }
if ( something_else ) {
   // do other stuff
   }
```

Why?

- You should test your code. Right?
  - *ad hoc* testing means you test whatever occurs to you, and you'll obviously miss some things…
- It takes more time *upfront*
  - But, it is likely that it will save you time in the long run
  - And, it forces you to write more readable, maintainable code

Test driven development

- Your programming workflow:
  - Understand the features of the problem
  - Write tests on those features
  - Implement those features, incrementally testing

- Often promoted, rarely performed!

An individual test

- Should focus narrowly on one bit of functionality
- Shouldn't depend on "internal" functions, but should exercise a "contract"
- Setup, exercise, check
- check with *asserts*
  - These read "Ensure that X is true".

Assert methods

- `assertTrue(String message, Boolean test)`
- `assertFalse(String message, Boolean test)`
- `assertNull(String message, Object object)`
- `assertNotNull(String message, Object object)`
- `assertEquals(String message, Object expected, Object actual)`
  - (uses equals method)
- `assertSame(String message, Object expected, Object actual)`
  - (uses == operator)
- `assertNotSame(String message, Object expected, Object actual)`
- `assertArrayEquals(String msg, Array expected, Array actual)`

Black-box testing

- Treat a unit as something you can't see inside.
  - You just know the "contracts" : for given inputs, you expect certain kinds of outputs
- Should work if the code is rewritten
- Can be used by someone unfamiliar with the code
- As opposed to glass-box testing

JUnit is built into most IDEs