Thinking Like a Software Engineer

Armando Solar-Lezama
Spring 2010

Objectives
what you should expect to get out of this course

fundamental programming skills
how to specify, design, implement and test a program
proficiency in Java and use of Java APIs
use of standard development tools (Eclipse, Subversion, JUnit)

engineering sensibilities
capturing the essence of a problem
inventing powerful abstractions
appreciating the value of simplicity
awareness of risks and fallibilities

cultural literacy
familiarity with a variety of technologies (http, threads, sockets, etc)

Intellectual Structure
three paradigms
state machine programming
symbolic programming
object-based programming

pervasive themes
models and abstractions
interfaces and decoupling
analysis with invariants

incremental approach
concepts introduced as needed
deepening sophistication as ideas are revisited
Your Responsibilities

assignments
  • three 1-week explorations
  • writing a program we’ll use as a lecture example
  • three 1-week problem sets
  • written exercises to exercise concepts learned in class
  • three 2-3-week projects
  • in rotating teams of 3 people
  • three 3-hour project labs, one for each project
  • project labs prepare you to get started on the project
  • no quizzes

meetings
  • two lectures each week (Mon and Wed)
  • one recitation each week (Thursday)
  • one lab period each week (Fri), sometimes used for lab assignments and sometimes for meetings between project groups and TAs

laptops required
  • lab periods will require you to have a laptop
  • if you don’t have your own, you can borrow one from IS&T’s laptop loaner program
  • send mail to 6.005-prof@mit.edu to request a laptop

Grading Policy

collaboration
  • projects in teams of 3; must have different teams for each project
  • problem sets and explorations are done individually
    • discussion permitted but writing and code may not be shared

use of available resources
  • can use publicly available code, with proper attribution
    • but if an assignment says “implement X”, you cannot just reuse somebody else’s X
  • cannot reuse work done in 6.005 by another student (in this or past term)
  • cannot make your work available to other 6.005 students

grade breakdown
  • projects 40%
  • problem sets 30%
  • explorations 20%
  • participation 10%

Today’s Topics

basic Java
  • syntax & semantics

hacking vs. software engineering
  • hacking vs. software engineering

what makes software “good”
  • whether it works isn’t the only consideration
Why We Use Java in 6.005 (not Python)

safety
- static typing catches errors before you even run (unlike Python)
- strong typing and memory safety catch errors at run time (unlike C/C++)

ubiquity
- Java is widely used in industry and education

libraries
- Java has libraries and frameworks for many things

tools
- excellent, free tools exist for Java development (like Eclipse)

it's good to be multilingual
- knowing two languages paves the way to learning more (which you should)

why we regret using Java...
- wordy, inconsistent, freighted with legacy baggage from older languages,
  no interpreter, no lambda expressions, no continuations, no tail recursion,

Hailstone Sequences

start with some positive integer \( n \)
- if \( n \) is even, then next number is \( n/2 \)
- if \( n \) is odd, then next number is \( 3n+1 \)
- repeat these moves until you reach 1

examples
- \( 2, 1 \)
- \( 7, 22, 11, 34, 17, 52, 26, 13, 40, \ldots \)
- \( 3, 10, 5, 16, 8, 4, 2, 1 \)
- \( 4, 2, 1 \)
- \( 5, 16, 8, 4, 2, 1 \)

why “hailstone”? because hailstones in clouds also bounce up and down chaotically before finally falling to the ground

let’s explore this sequence
- open question: does every positive integer \( n \) eventually reach 1?

Computing a Hailstone Sequence

```java
int n = 3;
while (n != 1) {
    System.out.println(n);
    if (n % 2 == 0) {
        n = n / 2;
    } else {
        n = 3 * n + 1;
    }
}
System.out.println(n);
```

Java Syntax

statement grouping
- curly braces surround groups of statements
- semicolons terminate statements
- indentation is technically optional but essential for human readers

comments
- // introduce comment lines
- /* ... */ surround comment blocks

control statements
- while and if require parentheses around their conditions

operators
- mostly common with Python (+, -, *, /, <, >, <=, >=, ==)
- != means “not equal to”
- ! means “not”, so \( n!=1 \) is the same as \( !(n == 1) \)
- the `%` operator computes remainder after division
Declarations and Types
variables must be declared before being used
- a declaration includes the type of the variable
- two kinds of types, primitive and object
- primitive types include
  - int (integers up to +/- 2 billion)
  - long (integers up to +/- 2\(^{63}\))
  - boolean (true or false)
  - double (floating-point numbers)
  - char (characters)
- object types include
  - String (a sequence of characters, i.e. text)
- you can define new object types (using classes), but you can’t define new primitive types

Static Typing
static vs. dynamic
- static or compile-time means “known or done before the program runs”
- dynamic or run-time means “known or done while the program runs”

Java has static typing
- expressions are checked for type errors before the program runs
- Eclipse does it while you’re writing, in fact
  ```java
  int n = 1;
  n = n + "2"; // type error – Eclipse won’t let you run the program
  ```
- Python has dynamic typing – it wouldn’t complain about n + "2" until it reaches that line in the running program
A Complete Java Program

```java
public class Hailstone {
    public static void main(String[] args) {
        int n = 3;
        while (n != 1) {
            System.out.println(n);
            if (n % 2 == 0) {
                n = n / 2;
            } else {
                n = 3 * n + 1;
            }
            System.out.println(n);
        }
    }
}
```

Hacking vs. Software Engineering

So far we've been hacking

- Writing code without thought or plan
- Hacking is often marked by unbridled optimism
  - Writing lots of code before testing any of it
  - Keeping all the details in your head, assuming you'll remember forever
  - Assuming that bugs will be nonexistent, or else easy to find and fix

Software engineering is not hacking

- Think first, then code
- 6.005 gives you tools for thinking -- models and notations for expressing design problems
- Engineers are pessimists
  - Write a little bit at a time, testing as you go
  - Document the assumptions that the code depends on
  - Defend your code against stupidity – especially your own

Length of a Hailstone Sequence

```java
/*
* Returns the number of moves of the hailstone sequence needed to get from n to 1.
*/
public static int hailstoneLength(int n) {
    int moves = 0;
    while (n != 1) {
        if (isEven(n)) {
            n = n / 2;
        } else {
            n = 3 * n + 1;
        }
        ++moves;
    }
    return moves;
}
```

More Method Definitions

```java
/*
* Returns true if and only if n is even.
*/
public static boolean isEven(int n) {
    return n % 2 == 0;
}

/*
* Start of the program.
*/
public static void main(String[] args) { ... }
```
Recursive Method

public static int hailstoneLength(int n) {
    if (n == 1) {
        return 0;  // base case
    } else if (isEven(n)) {
        return 1 + hailstoneLength(n/2);  // recursive cases
    } else {
        return 1 + hailstoneLength(3*n + 1);
    }
}

Hailstone Sequence as a String

/*
 * Returns the hailstone sequence from n to 1
 * as a comma-separated string.
 * e.g. if n=5, then returns "5,16,8,4,2,1".
 */
public static String hailstoneSequence(int n) {
    // ...
}

Strings

- a String is an object representing a sequence of characters
- returning a List of integers would be better, but we need more machinery for Java Lists, so we'll defer it
- strings can be concatenated using +
  - "8" + "4" \(\rightarrow\) "84"
- String objects are immutable (never change), so concatenation creates a new string, it doesn't change the original string objects
- String objects have various methods
  - `String seq = "4,2,1";`
  - `seq.length() \(\rightarrow\) 5`
  - `seq.charAt(0) \(\rightarrow\) '4'`
  - `seq.substring(0, 2) \(\rightarrow\) "4,"
- use Google to find the Java documentation for String
- learn how to read the Java docs, and get familiar with them

Hailstone Sequence as a String

/*
 * Returns the hailstone sequence from n to 1
 * as a comma-separated string.
 * e.g. if n=5, then returns "5,16,8,4,2,1".
 */
public static String hailstoneSequence(int n) {
    String seq = String.valueOf(n);
    while (n != 1) {
        if (isEven(n)) {
            n = n / 2;
        } else {
            n = 3*n + 1;
        }
        seq += "," + n;
    }
    return seq;
}

Type error! Java requires you to convert the integer into a String object. This is a compile-time error.

But the + operator converts numbers to strings automatically

common shorthand for \(s = s + \"\", + n\)
Hailstone Sequence as an Array

```java
/**
 * Returns the hailstone sequence starting from n as an
 * array of integers, e.g. hailstoneArray(8) returns
 * the length-4 array [8,4,2,1].
 */
public static int[] hailstoneArray(int n) {
    int[] array = new int[hailstoneLength(n)+1];
    int i = 0;
    while (n != 1) {
        array[i] = n;
        ++i;
        if (isEven(n)) {
            n = n / 2;
        } else {
            n = 3 * n + 1;
        }
    }
    array[i] = n;
    return array;
}
```

Hailstone Sequence as a List

```java
import java.util.List;

/**
 * Returns the hailstone sequence starting from n as an
 * List of integers, e.g. hailstoneArray(8) returns
 * the list [8,4,2,1].
 */
public static List<Integer> hailstoneArray(int n) {
    ...
}
```

Arrays

- **array** is a fixed-length sequence of values
  - base type of an array can be any type (primitive, object, another array type)
  - array[] intArray;
  - char[] charArray;
  - String[] stringArray;
  - double[][] matrix; // array of arrays of floating-point numbers
  - fresh arrays are created with new keyword
  - intArray = new int[5]; // makes array of 5 integers
  - operations on an array
  - intArray[0] = 200; // sets a value
  - intArray[0] = 200 // gets a value
  - intArray.length = 5 // gets array's length
  - unlike a String, an array's elements can be changed
  - but once created, an array's length cannot be changed
    - so it's not like a Python list – a Java array can't grow or shrink

What happens if you omit this “+1”? The array is too short, and Java produces a runtime error when you try to write the last number. This is an example of an off-by-one error, a very common kind of bug.

Hailstone Sequence as a List

```java
import java.util.List;

 /**
 * Returns the hailstone sequence starting from n as an
 * List of integers, e.g. hailstoneArray(8) returns
 * the list [8,4,2,1].
 */
public static List<Integer> hailstoneArray(int n) {
    ...
}
```
Lists

List represents a variable-length sequence

- element type of a list can be any object type (not a primitive type)

  ```java
  List<String> stringList;
  List<Integer> intList;
  ```

- fresh lists are created with `new` keyword

  ```java
  intList = new ArrayList<Integer>(); // makes an empty list
  ```

Operations on a list

- ```java
  intList.add(200); // adds a value to the end of the list
  intList.get(0) ➞ 200 // gets a value
  intList.size() ➞ 5 // gets list's length
  ```

What Makes “Good” Software

- easy to understand
  - simple, short methods
  - well chosen, descriptive names
  - clear, accurate documentation
  - indentation

- ready for change
  - nonredundant: complex code or important design decisions appear in only one place
  - “decoupled”: changeable parts are isolated from each other

- safe from bugs
  - static typing helps find bugs before you run
  - assertions and runtime checking catch bugs quickly at runtime
  - testable in small parts
  - no hidden assumptions waiting to trap you or another programmer later

A Larger View of Good Software

- correct
  - gets the right answers

- economical
  - runs fast, uses minimal resources, doesn’t cost much to produce

- dependable
  - safe from bugs

- maintainable
  - easy to understand and ready for change

- usable
  - has an effective user interface

- secure
  - safe from malicious attacks

... all these properties matter in practice

Summary

- basic Java
  - control statements, expressions, operators
  - types and declarations
  - methods
  - strings, arrays, lists

- thinking like a software engineer
  - defensive programming
  - documenting your assumptions

- properties of good software
  - easy to understand
  - ready for change
  - safe from bugs

... all these properties matter in practice

- sometimes supporting each other, sometimes in conflict