Introduction

Spring 2009
Today’s Topics

what makes software “good” and what makes writing good software hard

- whether it works isn’t the only consideration

What 6.005 will teach you

- Course content, logistics

getting up to speed with Java

- note that programming experience is a prerequisite for 6.005
- we assume you’ve used Python
- these initial lectures will show the Java way to do things you should already be able to do in Python (or some other language)
How Good is Our Personal Software?

**Software Warranty 1987**

**Interactive EasyFlow:** “We don't claim Interactive EasyFlow is good for anything ... if you think it is, great, but it's up to you to decide. If Interactive EasyFlow doesn't work: tough. If you lose a million because Interactive EasyFlow messes up, it's you that's out of the million, not us. If you don't like this disclaimer: tough. We reserve the right to do the absolute minimum provided by law, up to and including nothing. This is basically the same disclaimer that comes with all software packages, but ours is in plain English and theirs is in legalese.”

**Software Warranties 2008**

**Apple:** “Except for the limited warranty on media ... software is provided “as is”, with all faults and without warranty of any kind...”

**Google:** “as is, with no warranties whatsoever”

**Microsoft:** “substantially in accordance with the accompanying materials, for a period of 90 days...”
Software Failures

Is your PC secure?

Typical security patch size
  ▪ 100MB

Typical time to download
  ▪ 10 minutes

average time to infection for unprotected PC *
  ▪ 4 minutes

* [Windows XP, default firewall settings] Unprotected PCs Fall To Hacker Bots In Just Four Minutes
Gregg Keizer; Nov 30, 2004; http://www.techweb.com/wire/security/54201306
From: Security Absurdity: The Complete, Unquestionable, And Total Failure of Information Security
Noam Eppel; http://securityabsurdity.com
What About Critical Software?

Runaway Cannons in South Africa, October 2007

- antiaircraft cannon kills 9 soldiers and injures 14 others
- cause not known, but software suspected


Air Traffic Control

A radar system that was supposed to warn low-flying planes of nearby obstacles was plagued with problems and fixed nationwide only after a 1997 fatal airplane crash on Guam

In some cases system did not produce alarms.

In other cases false alarms were so numerous that air traffic controllers placed cardboard over warning speakers to silence the noise.
How Did We Get Here?

Magnetic Disks, US $/Gigabyte
How Did We Get Here?

Operating System Growth in millions of lines of code

<table>
<thead>
<tr>
<th>Version</th>
<th>Year</th>
</tr>
</thead>
<tbody>
<tr>
<td>NASA space shuttle ctrl</td>
<td>1992</td>
</tr>
<tr>
<td>Windows 3.1</td>
<td>1992</td>
</tr>
<tr>
<td>NT 95</td>
<td>1998</td>
</tr>
<tr>
<td>Solaris 98</td>
<td>1998</td>
</tr>
<tr>
<td>Windows NT 5.0</td>
<td>1998</td>
</tr>
<tr>
<td>Red Hat Linux 6.2</td>
<td>2000</td>
</tr>
<tr>
<td>Red Hat Linux 7.1</td>
<td>2001</td>
</tr>
<tr>
<td>Windows XP</td>
<td></td>
</tr>
<tr>
<td>Vista</td>
<td></td>
</tr>
</tbody>
</table>
Software as a component

A software system is a component of a larger system

- interacts with physical environment
- and organizational context of operators & users

Sources of defects

- < 3% of software failures due to bugs in code
- >90% from poor understanding of requirements

Consequences

- requirements analysis is critical
- not just function, also assumptions, e.g., environmental
  - Airbus disaster Warsaw 1993
  - Space shuttle disaster
State Space Complexity

Software systems have huge state space

- in lifetime, small proportion covered
- in testing, hardly any covered

Implications

- “Program testing can be used to show the presence of bugs, but never to show their absence!”* 
- often running in uncharted territory

*E.W. Dijkstra, Structured programming (EWD268)
http://www.cs.utexas.edu/users/EWD/
Coupling

What is coupling?
- when components of a system affect each other
- damages reliability, makes changes hard

Physical components
- coupled in simple and predictable ways

Software components
- coupled in complex and unpredictable ways
## Taming Software

<table>
<thead>
<tr>
<th>Requirements</th>
<th>State space</th>
<th>Decoupling</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pay attention to context</td>
<td>Explicit modeling Simplicity</td>
<td>Automated Testing</td>
</tr>
<tr>
<td>End-to-end arguments</td>
<td>Model-based exploration</td>
<td>Safe languages</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Dependency management</td>
</tr>
</tbody>
</table>
What Makes “Good” Software

safe from bugs
- static typing helps find bugs before you run
- testable in small parts
- no hidden assumptions waiting to trap you or another programmer later

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

ready for change
- Non-redundant: complex code or important design decisions appear in only one place
- “decoupled”: changeable parts are isolated from each other
About 6.005

**lecturers**

- Srini Devadas and Martin Rinard

**teaching assistants**

- Alex Bakst, Harold Cooper, Hank Huang, Aleks Milicevic, Joe Near, Alex Schwendner, Kuat Yessenov

**lab assistants**

- Steven Bartel, Nicole Bieber, Lance Collins, Igor Kopylov, José Navarro, Fernando Shao, Yod Watanaprakornkul, Angela Yen

**web site on Stellar**

- http://stellar.mit.edu/S/course/6/sp09/6.005/
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, SVN, 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, postscript, 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 2-week **problem sets**
  - both written and programming components
- three 2-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

**meetings**

- two **lectures** each week (Mon, Wed, sometimes Fri)
- one **recitation** each week
- **project meetings** with your team members and teaching staff
  - lecture time will often be made available for these meetings
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 or code may not be shared

**use of available resources**
- can use publicly available code, designs, specs, except when explicitly disallowed
- 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%
Why We Use Java in 6.005

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 having to use Java...
- wordy, inconsistent, freighted with legacy baggage from older languages,
  no interpreter, no lambda expressions, no continuations, no tail recursion, ...

First 2 weeks are an intro to Java. 6.005 really starts in Week 3!
Hailstone Sequences
Lothar Collatz, 1937

**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
- 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

// hailstone sequence from n
while (n != 1) {
    if (n % 2 == 0) {
        n = n / 2;
    } else {
        n = 3 * n + 1;
    }
}

Python

# hailstone sequence from n
while n != 1:
    if n % 2 == 0:
        n = n / 2
    else:
        n = 3 * n + 1
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
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);
```

- **declares** the integer variable `n`
- prints a value to the console (useful for debugging)
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
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);
    }
}
/*
 * 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

/*
 * 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) {

† void means the method has no return type (so no return statement is required)

† String [] is an array of String objects (in this case, these strings are the arguments given to the program on the Unix/Windows/Mac command line)
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” ➞ “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() ➞ 5
   seq.charAt(0) ➞ ‘4’
   seq.substr(0, 2) ➞ “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 = n;
    // Type error! Java requires you to convert the integer into a String object. This is a compile-time error.
    String seq = String.valueOf(n);
    while (n != 1) {
        if (isEven(n)) {
            n = n / 2;
        } else {
            n = 3 * n + 1;
        }
        // But the + operator converts numbers to strings automatically
        seq += "," + n;
    }
    return seq;
}
Hailstone Sequence as an Array

/**
 * 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) {
    ...
}
Arrays

array is a fixed-length sequence of values

- base type of an array can be any type (primitive, object, another array type)
  ```java
  int[] intArray;
  char[] charArray;
  String[] stringArray;
  double[][] matrix;  // array of arrays of floating-point numbers
  ```

- fresh arrays are created with `new` keyword
  ```java
  intArray = new int[5];  // makes array of 5 integers
  ```

- operations on an array
  ```java
  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
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;
}
Maximum Value of an Array

/**
 * Returns the maximum value of an array of positive integers.
 * Returns 0 if the array is empty.
 */

public static int maxValue(int[] array) {
    int max = 0;
    for (int i = 0; i < array.length; ++i) {
        if (array[i] > max) max = array[i];
    }
    return max;
}

The for loop is commonly used for iterating through a collection. for (init; test; update) ... is roughly equivalent to init; while (test) ... ; update
Summary

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

properties of good software
- easy to understand
- ready for change
- safe from bugs
What You Should Do

today
➢ sign up for a recitation on the 6.005 web site

tomorrow
➢ go to the recitation you’ve been assigned to

Friday
➢ read Lab 1 before coming to lab
➢ go to your assigned lab location for Lab 1