Today’s Topics

What 6.005 will teach you

- Course content, logistics

A simple Java Example – Static vs Dynamic Typing

note that programming experience is a prerequisite for 6.005

- we assume you’ve used Python
- Basic Java is covered by nightly tutorials
  - *Yesterday, today and tomorrow*
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)
An ace programmer is a …

A Craftsman
- Create beautiful, elegant programs

An Architect
- Build cohesive, usable, extensible programs

An Engineer
- Create extremely complex programs with minimal bugs and deficiencies, on time in large teams

A Detective
- Examine and find bugs in programs

A Magician
- Innovate programs that no one thinks is possible to do
About 6.005

**lecturers**
- Saman Amarasinghe

**teaching assistants**
- Leonid Grinberg
- Eddie Francis Plaza
- Vlad Kontsevoi
- Sunila Saqib

**Lab assistants**
- TBD

- Shoaib Kamil
- Max Goldman
- Bryan Rosario
- Esha Sahai
- Hayden Metsky
- Joshua Oreman
Your Responsibilities

Assignments

- Five problem sets
  - programming components (including design, coding, specifications and tests)
  - Multi-step process
    - Beta turn-in
    - Code review (peers, staff and professional programmers)
    - Final turn-in

- Two small software development projects
  - Teams of 3 people

Two quizzes (March 13 & April 24 during-class time)

Meetings

- Three lectures each week (Mon, Wed, Fri)
- One recitation each week (Thu)
- project meetings with your team members and teaching staff
  - Lecture/recitation time will be made available for these meetings
Grading Policy

collaboration

- problem sets are done individually
  - discussion permitted but writing or code may not be shared
  - Can get inspiration from doing code reviews, but cannot copy code
- projects in teams of 3: you can choose your partners

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%
- quizzes 20%
- Participation 10%
Challenge Exercises

You don’t have to do them unless you want an A+

The exercises will not be part of your normal grade (up to an A)

However for you to get an A+

- Need an A in rest of the class
- Need to do well in the challenge exercises

Limited TA/LA support for challenge exercises

© 6.005 Staff 2007-09
Level of Language Abstraction

High level languages
- Compiler/interpreter do a lot of work for you
  - Manages your data
  - Hide details of the microprocessor
  - Reduces the possibilities for bugs
- But…
  - Less expressive
  - Less efficient

As you go down the abstraction
- Programmer gets more control/responsibility
  - You have to do more work, but…
  - You have a lot more power, ability to express yourself
  - More direct control over the performance of the program
- And… you get many more chances to mess-up!

At the lowest level
- You are one with the machine
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

Expressiveness
- Programmer has a lot of control of what is going on
- Java has libraries and frameworks for many things
- 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, ...
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

- 7, 22, 11, 34, 17, 52, 26, 13, 40, ...
- $2^n$, $2^{n-1}$, ..., 4, 2, 1

**why “hailstone”?** because hailstones in clouds also bounce up and down chaotically before finally falling to the ground
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  
7, 22, 11, 34, 17, 52, 26, 13, 40, ...

3, 10, 5, 16, 8, 4, 2, 1  
2^n, 2^{n-1}, ..., 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?
Hailstone Sequences
Lothar Collatz, 1937
Computing a Hailstone Sequence

Python

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

variables must be declared before being used

- a declaration includes the type of the variable

Types have values and operations

Two kinds of types, primitive and object
Primitive and Object types

**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**
- Primitive types are mainly mapped to special data types of the microprocessor, so it can do fast math/manipulation
Operations on Types

\[ a + b \quad +: \text{int} \times \text{int} \rightarrow \text{int} \]
\[ \text{bigint1.add(bigint2)} \quad \text{add: BigInteger} \times \text{BigInteger} \rightarrow \text{BigInteger} \]
\[ \text{Str.length()} \quad \text{length: String} \rightarrow \text{int} \]

Some operators are overloaded

- Same operation names can be used in different types
- Types will indicate with operator to use
- Examples: +, -, *
Static and Dynamic Typing

**static (compile-time) vs. dynamic (run-time)**

- **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
int n = 5;
if (n == true) { }
type error @ compile time
because integers can never be
equal to booleans
Locate the Bug II

double d = 1 / 5;

Not an error
But...
Programmer intended to get 0.2
but will get 0 assigned to d
Used the wrong / operator
as / is overloaded
Locate the Bug III

```java
int m = 5;
m = m/0;
```

Divide by zero exception @ runtime
Can use Java exception handler to catch this case.
double e = 5.0;
e = e/0;

Divide by zero is OK in floating point
Doubles can handle infinity!
int big = 2000000000;
big = big * 2;

overflow error
because an int can’t hold a
number bigger than 2 billion
But..
Java does not catch it and returns
-1651507200

Theory goes out of the window for expediency and efficiency.
The devil is in the details..
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);
    }
}
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);
    }
}
Length of a Hailstone Sequence

/*
   * Returns the number of moves of the hailstone sequence
   * needed to get from n to 1.
   */

public static int hailstoneLength(int n) {

}
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

/*
 * Returns true if and only if n is even.
 */

public static boolean isEven(int n) {
    return n % 2 == 0;
}
More Method Definitions

/*
 * Start of the program.
 */

public static void main(String[] args) {

}
More Method Definitions

/*
 * Start of the program.
 */

public static void main(String[] args) {
    for(int n=1; n < 10; n++) {
        int len = hailstoneLength(n);
        System.out.println(n + " : " + len);
    }
}

- **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” \(\rightarrow\) “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

```java
String seq = “4,2,1”;
seq.length() \(\rightarrow\) 5
seq.charAt(0) \(\rightarrow\) ‘4’
seq.substr(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 = n;
    while (n != 1) {
        if (isEven(n))
            n = n / 2;
        else
            n = 3 * n + 1;
        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
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) {
    int[] seq = new int[100];
    int i = 0;
    seq[i++] = n;
    while (n != 1) {
        if (isEven(n))
            n = n / 2;
        else
            n = 3 * n + 1;
        seq[i++] = n;
    }
    return seq;
}
Hailstone Sequence as a List

/**
 * Returns the hailstone sequence starting from n as an list of integers, e.g. hailstoneList(8) returns list of integers [8,4,2,1].
 */

public static List<Integer> hailstoneList(int n) {
    ...
}
Hailstone Sequence as a List

/**
 * Returns the hailstone sequence starting from n as an
 * list of integers, e.g. hailstoneList(8) returns
 * list of integers [8,4,2,1].
 */

public static List<Integer> hailstoneList(int n) {
    List<Integer> seq = new ArrayList<Integer>();
    seq.add(n);
    while (n != 1) {
        if (isEven(n))
            n = n / 2;
        else
            n = 3 * n + 1;
        seq.add(n);
    }
    return seq;
}
```java
public static List<Integer> hailstoneList(int n) {
    List<Integer> seq = new ArrayList<Integer>();
    seq.add(n);
    while (n != 1) {
        if (isEven(n))
            n = n / 2;
        else
            n = 3 * n + 1;
        seq.add(n);
    }
    return seq;
}
```

*static* means that the method doesn’t take a self parameter (which in Java is implicit).

Cannot have method parameters.
Hailstone Sequence as a List

/**
 * Returns the hailstone sequence starting from n as an
 * list of integers, e.g. hailstoneList(8) returns
 * list of integers [8,4,2,1].
 */

public static List<Integer> hailstoneList(int n) {
    List<Integer> seq = new ArrayList<Integer>();
    seq.add(n);
    while (n != 1) {
        if (isEven(n))
            n = n / 2;
        else
            n = 3 * n + 1;
        seq.add(n);
    }
    return seq;
}
Hailstone Sequence as a List

/**
 * Returns the hailstone sequence starting from n as an
 * list of integers, e.g. hailstoneList(8) returns
 * list of integers [8,4,2,1].
 */

public static List<Integer> hailstoneList(int n) {
    final List<Integer> seq = new ArrayList<Integer>();
    seq.add(n);
    while (n != 1) {
        if (isEven(n))
            n = n / 2;
        else
            n = 3 * n + 1;
        seq.add(n);
    }
    return seq;
}
Hailstone Sequence as a List

/**
 * Returns the hailstone sequence starting from n as an
 * list of integers, e.g. hailstoneList(8) returns
 * list of integers [8,4,2,1].
 */

public static List<Integer> hailstoneList(int n) {
    final List<Integer> seq = new ArrayList<Integer>();
    seq.add(n);
    while (n != 1) {
        if (isEven(n))
            n = n / 2;
        else
            n = 3 * n + 1;
        seq.add(n);
    }
    return seq;
}

List is a “generic type”. I.e. you can make a list
of other objects.
/**
 * Returns the hailstone sequence starting from n as an list of integers, e.g. hailstoneList(8) returns list of integers [8,4,2,1].
 */

public static List<Integer> hailstoneList(int n) {
    final List<Integer> seq = new ArrayList<Integer>();
    seq.add(n);
    while (n != 1) {
        if (isEven(n))
            n = n / 2;
        else
            n = 3 * n + 1;
        seq.add(n);
    }
    return seq;
}
Top 10 reasons to be in 6.005

10. I want to become a really good programmer
9. I want to write programs that won't crash all the time
8. I want to write my own iPhone app
7. I want to be a summer intern at Google so I can eat all that good food and get laundry done at work
6. I want to be the next Linus and write an operating system for the entire cloud
5. I want to do a startup with my roommate who is a Sloanie
4. I need a social life, hanging out and working on 6.005 problem sets is better than watching Star Trek reruns by myself
3. Everyone I know at MIT has declared as course-VI majors, I will have no friends left if I don’t do this
2. I watched Social Networking
1. I want to be a billionaire so fXXXg bad....