Introduction to Comp. Sci., Notes

Madeleine Thompson
May 8, 2013

Contents

1 Introduction **6**

2 Required software **6**
2.1 What is required . . . 6
2.2 Installing on Mac OS X . . . 6
2.3 Installing on Windows 7 or 8 . . . 7
2.4 Installing on something else . . . 7
2.5 Text editors . . . 7
2.6 Setting your editor on the Mac . . . 8
2.7 Setting your editor on Windows . . . 8

3 Version control **9**
3.1 Problems version control solves . . . 9
3.2 References . . . 9
3.3 Version control concepts . . . 10

4 Subversion demo # 1 **10**

5 Java development **15**
5.1 javac . . . 15
5.2 Running your programs . . . 15
5.3 Checkstyle . . . 16
5.4 Testing your programs . . . 16

6 Subversion demo #2 **17**
6.1 Preliminaries . . . 17
6.2 Creating a source file . . . 17
6.3 Compiling the program . . . 18
6.4 Running the program . . . 18
6.5 Running automatic tests . . . 19
6.6 Ignoring class files . . . 19
6.7 Committing to the repository . . . 20

7 Introduction to Java **21**
7.1 Basic Java template . . . 21
7.2 Exercise: First changes to the template . . . 21
7.3 Integer variables . . . 22
7.4 Expressions in Java . . . 23
7.5 While loops . . . 24

7.6 Exercise: Hello3, a second time . . . 25

8 Boolean variables and conditionals **26**
8.1 Boolean variables . . . 26
8.2 Exercises: boolean expressions . . . 26
8.3 Boolean expressions in while loop conditions . . . 26
8.4 If and if-else . . . 27
8.5 Nesting ifs . . . 28
8.6 Exercise: nesting ifs . . . 28

9 Characters and strings **29**
9.1 Characters . . . 29
9.2 Strings, briefly . . . 29

10 Reading and writing to the terminal **31**
10.1 Writing to the terminal . . . 31
10.2 printf . . . 31
10.3 Reading from the terminal . . . 32
10.4 Exercise: terminal IO . . . 33

11 More loops **34**
11.1 Overview . . . 34
11.2 break . . . 34
11.3 continue . . . 35
11.4 for . . . 36
11.5 Exercise: counting down . . . 36
11.6 Nested loops . . . 37

12 Static methods **39**
12.1 Introduction . . . 39
12.2 Example . . . 39
12.3 Purposes of methods . . . 39
12.4 Parameters . . . 40
12.5 Scope #1 . . . 40
12.6 Return values . . . 41
12.7 Calling methods in other classes . . . 43
12.8 Exercise: absolute value method . . . 43
12.9 Debugging errors using a backtrace . . . 44

13 Arrays **45**
13.1 What is an array? . . . 45
13.2 Creating arrays in Java . . . 45
13.3 Working with arrays . . . 46
13.4 Arrays are references . . . 46
13.5 Passing arrays to methods . . . 47
13.6 Exercise: containsElement . . . 48

14 Type conversions **49**

15 Classes **50**
15.1 Objects . . . 50
15.2 Classes . . . 50
15.3 Creating, accessing, and modifying objects . . . 51
15.4 Objects and primitive types . . . 51
15.5 Primitive types . . . 52
15.6 Exercise: area of a rectangle . . . 54

16 Non-static methods **55**
16.1 Introductory example . . . 55
16.2 Another example . . . 56

17 Nested objects **57**
17.1 Introduction . . . 57
17.2 null . . . 57
17.3 Uninitialized values . . . 57
17.4 Objects inside objects . . . 58
17.5 Objects inside arrays . . . 59
17.6 Exercise . . . 61

18 Arrays inside arrays **62**
18.1 Introduction . . . 62
18.2 Creating multidimensional arrays . . . 62
18.3 References in multidimensional arrays . . . 63
18.4 Ragged arrays . . . 63
18.5 Exercise . . . 65

19 Constructors **66**
19.1 Initializing objects and factory methods . . . 66
19.2 Constructor introduction . . . 67
19.3 Multiple constructors . . . 67
19.4 Exercise: representing money . . . 68

20 Inheritance **69**
20.1 Extending a class . . . 69
20.2 Exercise: Canadian addresses (part 1) . . . 70
20.3 Inheriting methods . . . 70
20.4 Overriding methods . . . 71
20.5 Exercise: Canadian addresses (part 2) . . . 72

21 Public and private **73**

22 Dynamic and static types **75**
22.1 Terms . . . 75
22.2 Static type . . . 75
22.3 Dynamic type . . . 75
22.4 When static and dynamic types differ . . . 75
22.5 A nontrivial example . . . 76
22.6 Casting . . . 77
22.7 Bad casts . . . 78
22.8 Exercise: Legal or illegal? (part 1) . . . 79
22.9 Exercise: Legal or illegal? (part 2) . . . 80

23 Dynamic method dispatch and overriding **81**
23.1 Multiple methods with the same name . . . 81
23.2 Dynamic dispatch . . . 82
23.3 java.lang.Object . . . 82
23.4 toString . . . 83

24 equals **84**
24.1 == versus equals . . . 84
24.2 A first try . . . 85
24.3 Dynamic dispatch and equals . . . 86
24.4 instanceof . . . 88
24.5 A final try at equals . . . 88
24.6 Exercise: equals . . . 89

25 Interfaces and Comparable **90**
25.1 Motivation . . . 90
25.2 Interfaces . . . 91
25.3 A full example . . . 92
25.4 Exercise . . . 94

26 Parametric types **95**
26.1 ArrayList . . . 95
26.2 ArrayList example . . . 96
26.3 ArrayList versus Object[] . . . 97
26.4 Parametric ArrayList . . . 98
26.5 Exercise: Adding parentheses to strings . . . 98
26.6 Boxed type motivation . . . 99
26.7 Using boxed types . . . 100
26.8 For more information . . . 100

27 Iteration **101**
27.1 For-each loops . . . 101
27.2 Iterators . . . 102
27.3 Iterator example . . . 102
27.4 Iterators and for loops . . . 104

27.5 Exercise: Range iteration . . . 105

28 Generic example **106**

29 Exceptions **107**
29.1 Old school error reporting . . . 107
29.2 Exceptions solve this problem . . . 107
29.3 What are exceptions? . . . 107
29.4 Exceptions in Java . . . 108
29.5 Java try-catch example . . . 108
29.6 A more complete example . . . 109
29.7 Checked versus unchecked exceptions . . . 110
29.8 Example: NumberFormatException . . . 110
29.9 Declaring exceptions . . . 110
29.10 Impossible exceptions . . . 111
29.11 Exercise: bonding . . . 113

30 Text files **114**
30.1 Classes for reading and writing files . . . 114
30.2 Example . . . 116

31 Recursion and binary search **117**
31.1 Recursion, obligatory factorial example . . . 117
31.2 Call stack in recursive functions . . . 118
31.3 Iterative searching of unsorted arrays . . . 119
31.4 Binary searching sorted arrays . . . 120
31.5 Recursive search . . . 122
31.6 Exercise: chain length . . . 123

32 Selection sort and mergesort **124**
32.1 Sorting . . . 124
32.2 Selection sort . . . 124
32.3 Recursive selection sort . . . 124
32.4 Efficiency of selection sort . . . 125
32.5 Interlude: sorting a deck of cards . . . 126
32.6 Mergesort . . . 126
32.7 Efficiency of mergesort . . . 128
32.8 Exercise . . . 128

33 Networking **130**
33.1 How do computers talk to each other? . . . 130
33.2 Protocol layers . . . 130
33.3 Internet addresses . . . 130
33.4 Transport-layer (TCP/UDP) addresses . . . 131
33.5 Application layer . . . 132
33.6 Talking to a server by hand . . . 132

33.7 Networking in Java 133

34 Concurrency **134**
34.1 Walking and chewing gum at the same time 134
34.2 Threading 134
34.3 Threading in Java 134
34.4 Race conditions 135
34.5 Locking 135
34.6 Locking in Java 136
34.7 Issues 136
34.8 Optional readings 136

1 Introduction

2013–01–28

- See the course information document for meta-information on the class.
- For ongoing updates, see the course web page.

2 Required software

2.1 What is required

You're going to need several pieces of software in this course. They are:

- Subversion: A version control system you will use to manage the code you write and submit it for grading.
- The Java Development Kit (JDK): Contains `javac`, the Java compiler, and the runtime environment necessary for running Java programs.
- A text editor; see below.
- JUnit, Checkstyle, and supporting test frameworks for testing your code. These will be provided through Subversion, so you don't need to do anything to install them.

2.2 Installing on Mac OS X

- These instructions were tested on OS X 10.8.2.
- Some versions of Mac OS X ship with Subversion. You can see if you already have it installed by opening Terminal and typing `svn --version`. If it says something like this:

```
$ svn --version
svn, version 1.7.6 (r1370777)
   compiled Sep  8 2012, 23:22:35
...
```

then you have it installed. If it says something like this:

```
$ svn --version
bash: svn: command not found
```

then you can get it as part of Apple's "Command Line Tools" package, which can be downloaded from Downloads for Apple Developers. That page has different links for Lion (10.7) and Mountain Lion (10.8).

- Make sure you have the JDK installed. Again in Terminal, type `javac -version`. If it says something like this:

```
$ javac -version
javac 1.6.0_37
```

then you're all set as long as the version is 1.5 or newer. The first time, it may ask you if you want to download and install it. Say yes.

2.3 Installing on Windows 7 or 8

- These instructions were tested on Windows 8 but should work on Windows 7.
- The best Subversion variant I know of on Windows is Slik Subversion.
- Version 7 of the JDK can be downloaded from Oracle's Java download page. Install the most recent version if you don't have any already, though everything in this class should work fine with JDK 6.
- To save a great deal of typing, follow these instructions to add the JDK to your path. As of Jan. 20, 2013, the right path component to add is `C:\Program Files\Java\jdk1.7.0_11\bin`.

2.4 Installing on something else

- If you're running Linux, you can probably install Subversion and the JDK with your distribution's package manager. And, if you're running Linux, you probably already have quasi-religious views about Emacs and Vi; I won't try to change your mind.
- Contact me if you do not own a laptop or it runs something else.

2.5 Text editors

- You're going to need a text editor that can edit plain text files for this course.
- You can use TextEdit (Mac) or Notepad (Windows), but those are not well suited to programming.

- IDEs like Eclipse, IntelliJ, and NetBeans are not allowed.
- Emacs (Aquamacs for Mac, GNU Emacs for Windows) and Vim are popular and free, but moderately difficult to learn. Vi For Smarties is a good introduction to Vim.
- On Mac, XCode (downloadable from Apple) and TextMate (costs money) may be easier to learn.
- On Windows, Notepad++ is free and popular. (Click the "Notepad++ Installer" link on the "download" page if you want to download it. Be careful of spammy ads on the site.)
- You should not be able to tell where your body ends and your editor begins. Don't settle for less.

2.6 Setting your editor on the Mac

On the Mac command line, programs you run will sometimes want you to edit a file. To tell command-line programs to use your graphical editor, open the file "`.bash_profile`" in your home directory and add this line at the end:

```
export EDITOR="open -t -W -n"
```

If there is no such file, create one with just that line. This tells command-line programs to use the default graphical text editor whenever you need to edit a file.

To specify a default text editor other than TextEdit, click on any plain text file in the Finder and select "File > Get Info." Then, select your editor under "Open with:" and click "Change All..."

If you do not do this, you'll get error messages like:

```
svn: E205007: None of the environment variables SVN_EDITOR,
VISUAL or EDITOR are set, and no 'editor-cmd' run-time
configuration option was found
```

2.7 Setting your editor on Windows

Just like you added the JDK to your `PATH` environment variable, you can set your editor with the `SVN_EDITOR` environment variable. (That's an underscore, there.) These instructions are for Windows 8 and Notepad++; other situations will be similar.

- Move the mouse to the top-right corner of the screen and click "Settings."
- Click "Control Panel > System and Security > System > Advanced system settings > Environment Variables..."

- Click the upper (user-specific) "New...", and enter "`SVN_EDITOR`" as the variable name and the following as the value:

```
"C:\Program Files (x86)\Notepad++\notepad++.exe" -nosession -multiInst
```

- Click "OK" three times to close the dialog boxes.
- Right-click on the tiles screen and click "All apps." Open "PowerShell."
- Enter the command: `(Get-Item Env:\SVN_EDITOR).Value`
- If it worked, it should look something like this:

```
PS C:\Users\Madeleine> (Get-Item Env:\SVN_EDITOR).Value
"C:\Program Files (x86)\Notepad++\notepad++.exe" -nosession -multiInst
```

3 Version control

3.1 Problems version control solves

- Keeps around every version of your code.
- Data loss is rare.
- Remotely accessible
- Share code (with me, not each other)

3.2 References

- Version Control by Example
  - Introduction to many version control systems.
  - Chapters 2 and 3 cover Subversion.
- Version Control with Subversion
  - Comprehensive reference; go to if you need an explanation of a feature.
- `svn help`
  - Type `svn help` at the command line for a list of sub-commands.
  - Type `svn help commit` (for example) at the command line for help on the `commit` sub-command.

3.3 Version control concepts

- Files are stored in two types of place, the *repository* and the *working copy*.
- Repository
  - Central location where every version of every file in the system is stored.
  - A hub through which all users who want to use any file can coordinate.
  - Accessible with `svn` commands and through a web interface.
  - For this class, the repository lives on a server in WWH.
- Working copy
  - Every user who wants to use files from the repository gets their own working copy (or even more than one working copy).
  - Can be stored on any machine, usually your laptop for this class.
  - File level operations like "move" or "delete" are performed with `svn` commands.
  - Files in the working copy can also be used with regular tools like text editors.
- Workflow
  - Copy ("check out") files from the repository to create a working copy.
  - Create, delete, and modify files in the working copy.
  - Synchronize ("commit") changes from the working copy to the repository.

4 Subversion demo # 1

Our goal is to create a text file on our personal machine and store it in a Subversion repository. Though it sounds like a simple task, there is a lot going on that you'll need to pick up. Don't worry; it gets easier.

This demo is on a Mac; similar commands work on Windows. On a Mac, you'd start by opening up the Terminal program. (It lives in "Utilities" in the "Applications" folder.) On Windows 8, right-click on the tiles screen and click "All apps," then "PowerShell." The Mac terminal will show you a prompt like this:

```
machine:~ name$
```

The `~` indicates that your current directory is your home directory. The `$` indicates that it is waiting for you to type a command. I'll abbreviate this to `$` for the rest of the demo.

PowerShell on Windows looks like this:

```
PS C:\Users\Name>
```

I'll abbreviate this to `>` for the rest of the demo. Unless otherwise noted, the commands are the same on Windows and Mac.

Suppose you want to work in your `Documents` directory (`My Documents` on Windows). Use the `cd` command ("change directory") to set the current working directory to `Documents`. (If the directory name has a space in it, put the whole directory name in double quotes.)

```
$ cd Documents
```

You can see the files in the current working directory with the `ls` ("list") command (`dir` on Windows).

```
$ ls
Microsoft User Data       notes                        receipts
```

You can see what the current working directory is on with the `pwd` ("print working directory") command. (You don't need this on Windows; it's in the prompt.)

```
$ pwd
/Users/name/Documents
```

First, we'll need to create a working copy from the repository. The repository we'll be using is `https://subversive.cims.nyu.edu/csci0101`. If your netid is `test1`, your files would be in the `/u/test1` subdirectory of the repository. Run a command like this:

```
$ svn checkout --username test1 https://subversive.cims.nyu.edu/csci0101/u/test1
Password for 'test1':
A    test1/readonly
```

It'll ask you for your password. Enter your NYU password and press "enter." It will not show up as you type it. The text `A test1/readonly` is a status message; I'll talk about that later.

The checkout command creates a `test1` directory in `Documents` (or wherever your current working directory is). `cd` into that directory so you can add some files.

```
$ cd test1
```

If you run `pwd` now, you'll see that you're in there now. If you run `ls` now, you'll see some files I put there for you that'll you'll be using later. Now would be a good time to try out the `cd`, `ls`, and `pwd` commands. If you `cd` into a directory and want to get back to the parent directory, type:

```
$ cd ..
```

In general, "`..`" means the parent directory of the current directory, and "`.`" means the current directory itself.

When you're done trying the directory commands out, get back to the working copy you checked out. If you run `pwd`, it should look something like this:

```
$ pwd
/Users/name/Documents/test1
```

Any time you're in a working copy, you can use the `svn info` command to figure out how the current directory relates to the repository. (If you're going to ask for help with Subversion over email, always send the output of this command. It's great for diagnosing problems.) The output looks like this:

```
$ svn info
Path: .
Working Copy Root Path: /Users/name/Documents/test1
URL: https://subversive.cims.nyu.edu/csci0101/u/test1
Repository Root: https://subversive.cims.nyu.edu/csci0101
...
```

This says that the current working directory (`.`) is part of a working copy whose top level is `/Users/name/Documents/test1` and that the working directory corresponds to the repository URL `https://subversive.cims.nyu.edu/csci0101/u/test1`.

Next, we want to create a subdirectory to put a file in. The command `mkdir` can be used to create a directory, like this:

```
$ mkdir inclass
```

You can open up working copy in the Finder to see what the commands are doing in a more familiar interface. Don't move, rename, or otherwise change the directory structure of a working copy in the Finder; Subversion will get confused. It's fine to look, though.

Next, run `svn status` to see how Subversion thinks your working copy relates to the repository. It'll look something like this:

```
$ svn status
?       inclass
```

The `?` before `inclass` indicates that a file or directory named `inclass` is in the working copy, but Subversion is not tracking it and it will not be synchronized with the repository. This is sometimes what we want (more on this later), but we want this directory (and the file we're about to put in it) synchronized to the repository, so we tell Subversion about it with the `svn add` command. For example:

```
$ svn add inclass
```

Now, if you run `svn status`, you'll get a different message:

```
$ svn status
A         inclass
```

The `A` indicates that the named file or directory is scheduled to be added to the repository in the next commit, but has not been synchronized yet. There are many status letters other than `?` and `A`; you can see the full list in the svn status documentation.

Next, create a plain text file named `hello.txt` with the string "Hello!" in it inside the `inclass` directory. Use whatever text editor you want. If you type `svn status` now from the `test1` directory, it should show something like this:

```
$ svn status
A       inclass
?       inclass/hello.txt
```

The `/` between `inclass` and `hello.txt` means that `hello.txt` is in a directory named `inclass`. (It is a `\` on Windows.) We want to synchronize `hello.txt` with the repository, so we use `svn add` again.

```
$ svn add inclass/hello.txt
```

(On Windows, either `/` or `\` work here.) If we run `svn status` again, it should show that `hello.txt` is scheduled to be sent to the server (with the `A` indicator).

Finally, we want to send our new directory and file to the repository. We do this with the `svn commit` command. When we commit anything to the repository, it is accompanied by a "commit message." Anyone looking at a record of changes to the repository will see this message. They traditionally describe what new code does, what bugs are fixed with changes, etc. They're not important when you're working on homework problems alone; they're much more useful in long-lived projects with many contributors.

You specify a commit message after a `-m` flag in quote marks. Double quotes work on both Mac and Windows. Here's how you commit to the repository:

```
$ svn commit -m "First version of hello.txt."
Adding         inclass
Adding         inclass/hello.txt
Transmitting file data .
Committed revision 973.
```

This output indicates that `inclass` and `hello.txt` were added to the repository. Unless you specify otherwise, `svn commit` commits all of the uncommitted files in the current directory and its subdirectories (the same ones you see when you run `svn status`).

You can verify that your changes made it to the repository by going to the repository URL (like `https://subversive.cims.nyu.edu/csci0101/u/test1`) in a web browser. If you can see your changes there, they're in the repository. If you can't, they're not, which means I can't see them and give you credit for them.

Next time, I'll cover details of using Subversion to manage source code and show you how to compile, run, test, and check the style of your Java programs.

5 Java development

2013–01–30

5.1 javac

- You can't run Java programs directly as you can in Python.
- First, you must convert them to class files with `javac` ("Java Compiler").
- Class files are not human-readable, but they are easier for a computer to work with.
- Class files are *platform-independent*: there is no difference between a "Mac class file" and a "Windows class file."
- `javac` turns Java source files (ending in "`.java`") into Java class files (ending in "`.class`").
- While doing this translation, `javac` detects many kinds of errors that would only be detected at run-time in Python or JavaScript if ever.
- Early detection of errors is possibly the biggest advantage that compiled languages (Java, C++) have over other languages (Python, JavaScript).

5.2 Running your programs

- Java programs are run with the `java` command.
- If you've created a Java program in `inclass/Program.java`, run it with:

```
$ java inclass.Program
```

- If your program depends on code written by someone else, you need to tell Java where to find it. Use `-classpath` for this.
- `-classpath` is a list of directories and `.jar` (a zipped collection of class files) to search for class files.
- Directories are colon-separated on Mac, semicolon separated on Windows.
- Classpath example:

```
$ java -classpath .:testing/lib/somecode.jar inclass.Program
```

- `javac` also takes a `-classpath` argument.

5.3 Checkstyle

- Checkstyle is a program used to identify aspects of Java programs that make the code hard to read or are likely to indicate a bug.
- Until we learn Java, there's no point to discussing the details of how it works. For now, I'll just show you how to run it.
- Suppose you've written a program in the Java file `inclass/Hello.java`.
- Mac

```
$ ./testing/checkstyle.sh inclass/Hello.java
Running: java -classpath ".:testing/lib/antlr-2.7.7.jar...
Starting audit...
Audit done.
```

- Windows
  - By default, Windows prevents you from running scripts written in PowerShell, the language used for some of the tools for this class. You can enable PowerShell scripts by right-clicking on PowerShell, selecting "Run as Administrator," and typing:

```
PS C:\Windows\system32> Set-ExecutionPolicy RemoteSigned
```

  - You can now run Checkstyle similarly to how you would run it on a Mac:

```
> .\testing\checkstyle.ps1 inclass\Hello.java
Starting audit...
Audit done.
```

5.4 Testing your programs

- I will often provide tests to help you determine whether your code is working.
- I use the same mechanism for marking homework assignments.
- I write tests in Java. For example, `testing/HelloTest.java` tests `inclass/Hello.java`.
- The tests use external libraries, so running `javac` or `java` requires a long `-classpath` argument.
- To save you some typing, I wrote `test.sh` (Mac) and `test.ps1` (Windows).
- Mac

```
$ testing/test.sh HelloTest
```

- Windows (assuming you set the execution policy):

```
> testing\test.ps1 HelloTest
```

6 Subversion demo #2

6.1 Preliminaries

This demo assumes you've already gone through the steps in Subversion demo #1. In particular, you should have already checked out a working copy from the repository, created the `inclass` directory, and committed it to the repository. If you get confused, check out the Subversion references section.

Multiple people can check out the same part of the repository, but the working copy does not update automatically. Since I will be constantly adding things to the `testing` directory, and you might make changes from multiple computers, it's a good idea to get updates from the repository when you start working. First, `cd` into your working copy. Then, type `svn update` to get changes from the server.

```
$ cd Documents/test1
$ svn update
Updating '.':
At revision 973.
```

Or, on Windows (with the prompt abbreviated to `>`):

```
>cd "My Documents\test1"

>svn update
Updating '.':
At revision 973.
```

6.2 Creating a source file

Next, let's create a Java program that prints the text "Hello, World!" In Python, this would be:

```
print "Hello, World!"
```

In Java, it's a bit more complicated. Here's the text:

```
package inclass;

public class Hello {
    public static void main(String[] args) {
        System.out.println( "Hello, World!" );
    }
}
```

We'll discuss the parts of it later. Open your text editor and create a file in the `inclass` directory called `Hello.java` containing exactly that text.

6.3 Compiling the program

We want to compile the source to generate a class file. We do this with `javac`:

```
$ javac inclass/Hello.java
```

Note that we put `inclass/` before `Hello.java` because our current directory is `test1` but the source file is in `test1/inclass`. We need to say "Compile `Hello.java`, which is located in the `inclass` directory inside the current directory."

If everything worked, `javac` won't output anything to the terminal. It will create a file `Hello.class` inside the `inclass` directory, though. Make sure it's there with the `ls` commmand (`dir` on Windows).

```
$ ls inclass
Hello.class     Hello.java      hello.txt
```

6.4 Running the program

Now, we want to run the Java program we wrote. For this, use the `java` command:

```
$ java inclass.Hello
Hello, World!
```

The argument to the `java` command is a class name prefixed by an optional package name. In this case, our Java class definition in `Hello.java` had the line "`package inclass`" and was in the `inclass` directory. Its class name was defined to be `Hello` with the text "`public class Hello`" and was in a file named `Hello.java`. So, its fully-qualified class name with the package is `inclass.Hello`. (The slash turns into a dot because, in Java, slash means "divide by.")

6.5 Running automatic tests

For each homework and some in-class exercises, I will provide automatic tests to help check whether your code works. To go along with the `Hello` class, I've provided a test named `HelloTest`. You can run it on a Mac with the wrapper `test.sh`:

```
$ testing/test.sh HelloTest
testStyle(testing.HelloTest)
java.lang.AssertionError: Starting audit...
.../inclass/Hello.java:6:28: '(' is followed by whitespace.
.../inclass/Hello.java:6:44: ')' is preceded with whitespace.
Audit done.
```

or `test.ps1` on Windows:

```
> testing\test.ps1 HelloTest
...
```

In this case, one of the tests, the one that called Checkstyle, failed. In Java, it is considered bad style to put space between parentheses and method arguments, so change this line:

```
        System.out.println( "Hello, World!" );
```

to:

```
        System.out.println("Hello, World!");
```

Now, run `test.sh` or `test.ps1` again, and it should not output any errors.

You should not rely exclusively on the tests I give you with an assignment. The ones I provide may only be smoke tests. The tests I withhold for grading will be more comprehensive than the ones I make public before the assignment is due.

6.6 Ignoring class files

First, let's check to see how the working copy relates to the repository with `svn status`.

```
$ svn status
?       inclass/Hello.class
?       inclass/Hello.java
X       testing

Performing status on external item at 'testing':
```

The question marks indicate that the named file is in the working copy but not the repository and is not scheduled to be added with a commit. (You can ignore the "`X testing`" and the message about the external item.) We want to add `Hello.java`, but `Hello.class` can be generated from `Hello.java`, so there's no reason to put it in the repository.

If you haven't set your text editor, do so now; things will start exploding if you don't. See the Text editors section for details.

Files and directories in a Subversion repository have "properties" attached to them. The `svn:ignore` property on a directory tells Subversion which files we don't want `svn status` to bug us about. Type:

```
$ svn propedit svn:ignore inclass
```

to edit the `svn:ignore` property on the `inclass` directory. If your editor is properly configured, Subversion should open up a new window in the editor. Enter a single line containing:

```
*.class
```

to indicate that Subversion should ignore files whose names end in `.class`. (The `*` means "any text.")

If we run `svn status` now, `inclass` will be indicated as modified with an `M` (since we changed a property), but it'll no longer bug us about the class file.

6.7 Committing to the repository

If we haven't added `Hello.java` yet, we should do that:

```
$ svn add inclass/Hello.java
A         inclass/Hello.java
```

Then, like last time, we can commit our changes with "`svn commit`."

```
$ svn commit -m "Added Hello.java and ignored class files."
Sending        inclass
Sending        inclass/Hello.java
Transmitting file data .
Committed revision 990.
```

If we'd omitted the commit message and just typed "`svn commit`," Subversion would open an editor window as it did with "`svn propedit`." Sometimes, with some versions of Subversion, when you edit a directory property, you'll get this error:

```
svn: E160042: Commit failed (details follow):
svn: E160042: File or directory '' is out of date; try updating
svn: E160024: resource out of date; try updating
```

If you get that, type "`svn update`" and try again.

At this point, your Java file should be in the repository. Confirm this in the web interface.

7 Introduction to Java

2013–02–04

7.1 Basic Java template

From last time, here's a simple Java program:

```
package inclass;

public class Hello {
    public static void main(String[] args) {
        System.out.println("Hello, World!");
    }
}
```

Ignore most of it, we'll treat it as boilerplate for now. The one line that actually does something is:

```
System.out.println("Hello, World!");
```

For the rest of today, we'll focus on changing this line to do other things, with the rest of the program treated as a wrapper for it.

7.2 Exercise: First changes to the template

- Java code is organized around individual actions called *statements*.
- Statements are always followed by a semicolon.
- "`System.out.println("Hello, World!");`" is a statement.
- We want to create a program that greets us three times, not just once.
- Copy `Hello.java` from last time to a new file, `inclass/Hello3.java`.
- Change the text "`class Hello`" to "`class Hello3`" to match the filename.
- Open it up in your text editor, and copy the "`System...`" line twice, so it appears three times total.
- Compile it with "`javac inclass/Hello3.java`", then run it with "`java inclass.Hello3`."

7.3 Integer variables

- A variable is a name we give to a place in a computer's memory that stores a value.
- In Java, every variable has a *type* indicating what kind of value a variable can have.
- One type is `int`, a 32-bit integer value.
- An `int` is stored as 32 zeros and ones, so it can have 2^{32} distinct values.
- These values in the range -2147483648 to 2147483647.
- We declare a variable in Java with code of the form:

```
type variable-name = initial-value;
```

- For example:

```
int n = 4;
```

  This creates a variable `n` of type `int` with the initial value 4.

- We can print out the value of `n` like this:

```
System.out.println(n);
```

  Altogether, this program:

```
package inclass;

public class PrintFour {
    public static void main(String[] args) {
        int n = 4;
        System.out.println(n);
    }
}
```

  outputs the a single line with the text "4." Note that if you'd put the `n` in quotes:

```
System.out.println("n");
```

  It would've output the text "n," instead.

- (Bonus: In the program above, `PrintFour`, `void`, `System`, and `String[]` are all types just like `int`, they just hold values other than numbers. We'll talk about these later.)

7.4 Expressions in Java

- There are a lot of different ways we can express values that we assign to variables in Java.
- For example, we can do math:

```
int n = 3 * 5;
```

  This assigns 15 to `n`.

- We can put use other variables in expressions:

```
int a = 3;
int b = 2 + a;
```

  Now `a` is 3 and `b` is 5.

- We can change the value of a variable (it is, after all, *variable*) after we create it by leaving off the type.

```
int a = 3;
a = -5;
```

  Now `a` has the value -5.

- You can only declare one variable with a given name in any part of the program, so this is an error:

```
int a = 3;
int a = -5;  // Error!
```

- Statements are executed in order, so the current value of the variable is the only one that matters. If we leave

```
int a = 3;
int b = 2 + a;
a = 2 * b;
```

  Now `a` is 10.

- Assignment to the variable name on the left side of a statement with an `=` is the last thing that happens, so you can use the same variable on both sides:

```
int a = 10;
a = a / 5;
```

Now `a` has the value 2.

- Parentheses can be used for grouping. If they're left out, the usual algebraic rules apply (multiplication and division before addition and subtraction). When in doubt, use parentheses.

```
int x = (1 + 2) * 10;
int y = 1 + (2 * 10);
int z = 1 + 2 * 10;
```

Now `x` has the value 30 and both `y` and `z` have the value 21.

7.5 While loops

- A `while` loop repeatedly executes a sequence of statements until a condition is true.
- The general form is:

```
while (condition) {
    statements to run while
    condition is true
}
```

- For example:

```
public class WhileDemo {
    public static void main(String[] args) {
        int n = 0;
        while (n < 2) {
            System.out.println(n);
            n = n + 1;
        }
        System.out.println("Done.");
    }
}
```

Running the program looks like this:

```
$ javac WhileDemo.java
$ java WhileDemo
0
1
Done.
```

The program runs like this:

1. The program initializes `n` to 0.
2. Zero is less than 2, so the program enters the `while` loop.
3. The program prints 0.
4. The program adds one to `n`.
5. The while loop statements are finished, so the condition is checked again. One is less than 2, so the loop is entered again.
6. The program prints the current value of `n`, 1.
7. The program adds one to `n`, making is value 2.
8. The while loop statements are finished, so the condition is checked again. The value of `n`, 2, is not less than 2, so the loop is not entered and the program goes to the statements after the `while` loop.
9. The program prints "Done."

7.6 Exercise: Hello3, a second time

- We want to a program that greets us three times, not just once, but repeating code is ugly.
- Change `inclass/Hello3.java` to use a while loop to print "Hello, World!" three times.
- Submit it to the Subversion repository.

8 Boolean variables and conditionals

2013–02–06

8.1 Boolean variables

- Booleans take the value `true` or `false`.
- Example:

```
boolean b = true;
```

- Can be the result of numeric comparisons:
  - `==` (equal-to), `!=` (not equal to)
  - `<`, `>`, `<=` (less than or equal to), `>=` (greater than or equal to)
  - `==` and `!=` work with almost any type, but `<`, etc. only work with numbers.

```
boolean b = 3 > 4;  // b is now false.
```

- Operations that take `boolean` arguments and produce `boolean` results:
  - `!` (not), `&&` (and), `||` (or)

```
boolean b = 3 <= 2 || !(1 > 1);  // b is now true
```

  - `&&` and `||` are "short circuit" operators.
  - If the left hand side of `&&` is `false` or the left hand side of `||` is `true`, there's no need to evaluate the right hand side.

```
boolean b1 = 3 > 2     || 5 / 0 > 0;  // b1 is true
boolean b2 = 5 / 0 > 0 || 3 > 2;      // divide-by-zero
```

8.2 Exercises: boolean expressions

What are the values of these expressions?

1. `3 <= 3 && (1 < 0 || !false)`
2. `(true == true) == (0 < -1)`

8.3 Boolean expressions in while loop conditions

Any expression that has a value of `true` or `false` can be used as the condition in a `while` loop.

```
// loops forever
while (true) {
    ...
}
```

```
boolean done = false;
// loops until something in the loop sets done to true
while (!done) {
    ...
}
```

8.4 If and if-else

`if` is like `while`, taking a single condition and a block of code, but the statements run once at most. This prints "Foo!"

```
int x = 3;
if (x < 4) {
    System.out.println("Foo!");
}
```

You can add an extra `else` block containing statements to execute if the condition is not true. This prints "Bar!"

```
boolean a = false;
if (a) {
    System.out.println("Foo!");
} else {
    System.out.println("Bar!");
}
```

Omitting the braces is sometimes legal but always confusing. In this, what is the value of `z` at the end?

```
int z = 0;
if (z > 0)
if (z > 2)
z = 1;
else
z = -1;
```

8.5 Nesting ifs

You can put `if` inside `while` and `while` inside `if`.

```
public class NestedIf {
    public static void main(String[] args) {
        int n = 0;
        while (n < 4) {
            if (n < 2) {
                System.out.println("n < 2");
            } else {
                System.out.println("n >= 2");
            }
            n = n + 1;
        }
    }
}
```

This outputs:

```
n < 2
n < 2
n >= 2
n >= 2
```

8.6 Exercise: nesting ifs

Write a program in `inclass/EvenOdd.java` that counts from 0 to 9 with a `while` loop. Each time through the loop, it should print "even" or "odd" if the current number is even or odd. You might find the "remainder" operator, `%`, useful. For example, "`15 % 4`" is 3, since the remainder of $15 \div 4$ is 3. When you are done, submit the program to Subversion.

9 Characters and strings

9.1 Characters

- A *character* is a single letter, number, or symbol.
- Java characters are represented by an integer in the range 0 to 65535, an unsigned 16-bit integer.
- They have the type `char`, which is used similarly to `int`.
- `char` and `int` are fundamentally similar, only differing in their range of possible values.
- A lowercase "a" corresponds to the number 97, so in this code:

```
char c1 = 97;
char c2 = 'a';
```

  `c1` and `c2` both hold the same value, and `c1 == c2` will be true.

- Single quoted symbols store fixed characters in Java source the same way sequences of digits store fixed integers.
- Sometimes, you need backslashes to escape characters with special meanings, like newlines, quotes, and backslashes.
- Newlines correspond to the number 10, so the following are the same:

```
char c3 = 10;
char c4 = '\n';
```

- The encoding that maps 97 to "a," 10 to newline, etc. is called *Unicode*.
- For more information, check out "The Absolute Minimum Every Software Developer Absolutely, Positively Must Know About Unicode and Character Sets (No Excuses!)," by Joel Spolsky.

9.2 Strings, briefly

- Not too much on strings today.
- A `String` is a sequence of characters.
- String constants are wrapped in double quotes in Java.

```
String s = "Hi";
```

- The string `"Hi"` is two characters, 72 for "H" and 105 for "i."

- You can use the same backslash escapes in `String` constants as `char constants`.

```
String s = "I said, \"Hi.\"";
```

  In this example, the contents of `s` are initialized to be:

```
I said, "Hi."
```

10 Reading and writing to the terminal

2013–02–11

10.1 Writing to the terminal

We've already seen that this prints a single line containing the text "Hello."

```
System.out.println("Hello.");
```

There is a variation on it, `print`, that does not end the line.

```
System.out.print("Hello, ");
System.out.println("World.");
```

Since the first is a `print`, the line does not end, so that prints:

```
Hello, World.
```

You can mix types this way:

```
int n = 3;
System.out.print("n = ");
System.out.println(n);
```

which prints:

```
n = 3
```

Remember how the character literal `\n` meant newline? These two statements are equivalent:

```
System.out.println("foo");
```

```
System.out.print("foo\n");
```

10.2 printf

There's a more convenient way to print out complicated strings, `printf`. If there are no `%` signs in its argument, it just works like `print`. Wherever it sees a special sequence starting with `%`, it does something special depending on what follows the `%`. Examples are best. This prints "`n = 5`."

```
System.out.printf("n = %d\n", 5);
```

Note how the `%d` is replaced by a decimal number, the argument after the format string. If there is more than one %-escape, the arguments after the format string are used in order. This prints "`Unicode character 50 is 2.`"

```
System.out.printf("Unicode character %d is %c.\n", 50, 50);
```

Note how the first escape, `%d`, is replaced by the number 50, and the second, `%c`, is replaced by 50 interpreted as a Unicode character, which is the digit 2.

Just like `%d` formats a decimal integer and `%c` formats a character, `%s` formats a `String`, and `%%` formats a literal percent sign. So, these two are equivalent:

```
String s = ...;
System.out.println(s);
```

```
String s = ...;
System.out.printf("%s\n", s);
```

Both print the contents of `s` followed by a newline.

There are many more things you can do with format strings. For example, the escape "`%+.3g`" prints a decimal fraction to three digits if possible, switching to scientific notation if necessary, including a `+` sign for positive numbers. For all the details, see the documentation for `Formatter`.

10.3 Reading from the terminal

- We'll cover input in depth later in the course.
- For now, we'd like to be able to set a variable to something a user types at the keyboard.
- Here's some code than uses `Scanner`:

```
public class ScanDemo {

    public static void main(String[] args) {
        java.util.Scanner scanner =
                new java.util.Scanner(System.in);
        System.out.print("What is your name? ");
        String name = scanner.next();
        System.out.printf("Hello, %s!\n", name);
        System.out.print("Enter a number: ");
        int n = scanner.nextInt();
        System.out.printf("%d + 2 = %d\n", n, n + 2);
    }
}
```

Running it looks like this:

```
$ java ScanDemo
What is your name? Madeleine
Hello, Madeleine!
Enter a number: 4
4 + 2 = 6
```

- Ignore the statement that begins `java.util.Scanner`; we'll talk about that later. Just remember to put it at the beginning of `main` if you want to read from the keyboard.
- Note how `scanner.next` returns a `String` from the user, and `scanner.nextInt` returns an `int`.
- `scanner.nextChar` and `scanner.nextBoolean` read the corresponding types.

10.4 Exercise: terminal IO

Write a program, `inclass/TermIO.java`, that reads a number from the terminal and prints the character corresponding to its Unicode code point. Here'e an example of how it should run:

```
$ java inclass.TermIO
Enter a number: 64
Character 64 is '@'.
```

11 More loops

11.1 Overview

- There are several more types of conditionals and loops in Java.
- Two kinds of `for` loop, regular and "for-each."
- We'll talk about the regular `for` today and the for-each loop later.
- We'll also talk about `break` and `continue`, statements used to escape from the middle of a loop block.
- You might see `switch` and `do-while` in a book, but they're mostly useless.

11.2 break

- Exits current loop.
- Often used when the loop statements are of the form "Read input, then do something with input," so that the loop can terminate if the input ends.

```
// Sum squares from the terminal until either ten have been
// read or the user enters a non-positive integer.

public class SumSqBreak {

    public static void main(String[] args) {
        java.util.Scanner scanner =
                new java.util.Scanner(System.in);
        System.out.println("Enter up to ten numbers:");
        int i = 0;
        int sumSq = 0;
        while (i < 10) {
            int x = scanner.nextInt();
            if (x <= 0) {
                break;  // ends the while loop
            }
            sumSq = sumSq + x * x;
            i = i + 1;
        }
        System.out.print("sum of squares: ");
        System.out.println(sumSq);
    }
}
```

11.3 continue

- Doesn't end the loop, just skips to the condition test.

```
// Sum squares from the terminal until ten positive
// integers have been read.

public class SumSqContinue {

    public static void main(String[] args) {
        java.util.Scanner scanner =
                new java.util.Scanner(System.in);
        System.out.println("Enter ten positive numbers:");
        int i = 0;
        int sumSq = 0;
        while (i < 10) {
            int x = scanner.nextInt();
            if (x <= 0) {
                continue;  // skips to the condition check
            }
            sumSq = sumSq + x * x;
            i = i + 1;
        }
        System.out.print("sum of squares: ");
        System.out.println(sumSq);
    }
}
```

- When looping over inputs, can be used to ignore invalid or inappropriate data without wrapping the entire body with "`if (valid) { ... }`."
- Be careful! `break` and `continue` exit statement blocks without a close-brace, so they can confuse people reading your code.

11.4 for

- Syntax:

```
for (initializer; condition; iteration) {
  statements to repeat while condition is true;
}
```

- Example:

```
int x = 0;
for (int k = 0; k < 3; k = k + 1) {
      x = x + k;
}
// x is now 0 + 1 + 2 = 3
```

- Almost the same as:

```
int x = 0;
int k = 0;
while (k < 3) {
    x = x + k;
    k = k + 1;
}
```

- Would only be different if there were a continue inside the body.
- The iteration statement (here, "`k = k + 1`") runs every time.
- `for` loops are a common pattern, so they're provided for brevity.
- There is also a "for-each" loop, which we will discuss later. It looks like:

```
for (int x : someSequenceOfInts) {
    ...
}
```

11.5 Exercise: counting down

2013–02–13

Write a program in the file `inclass/CountDown.java` that uses a `for` loop that counts downward to generate this output:

```
10
9
8
7
6
5
```

Submit it to the Subversion repository.

11.6 Nested loops

- Loops can go inside other loops. Each time though the outer loop, the inner loop starts over again. For example:

```
for (int i = 0; i < 3; i++) {
    for (int k = 0; k < 2; k++) {
        System.out.print(i);
        System.out.print(',');
        System.out.println(k);
    }
}
```

  ...which outputs:

```
0,0
0,1
1,0
1,1
2,0
2,1
```

- Inner loop conditions can depend on outer loop variable:

```
for (int i = 1; i <= 5; i++) {
    int factorial = 1;
    for (int k = 1; k <= i; k++) {
        factorial = factorial * k;
    }
    System.out.print(i);
    System.out.print("! =");
    System.out.println(factorial);
}
```

  ...generates:

```
1! = 1
2! = 2
3! = 6
4! = 24
5! = 120
```

- `break` and `continue` apply to the innermost loop.

```
public class NestedBreak {

    public static void main(String[] args) {
        for (int i = 10; i <= 15; i++) {
            int factorial = 1;
            boolean overflow = false;
            for (int k = 1; k <= i; k++) {
                int nextFactorial = factorial * k;
                // check for overflow
                if (nextFactorial / k != factorial) {
                    overflow = true;
                    break;
                }
                factorial = nextFactorial;
            }
            if (overflow) {
                System.out.printf("%d! is big.\n", i);
            } else {
                System.out.printf("%d! = %d\n", i,
                                  factorial);
            }
        }
    }
}
```

Output:

```
10! = 3628800
11! = 39916800
12! = 479001600
13! is big.
14! is big.
15! is big.
```

If `break` escaped from the outer `for` rather than the inner `for`, there would be no output after the 12! row.

12 Static methods

12.1 Introduction

A Java *method* (almost the same thing as a *function*) is a named sequence of statements that we can run from anywhere in the program by putting parentheses after its name in any Java expression.

12.2 Example

First, an example:

```
public class Method1 {

    static void testMethod() {
        System.out.println("bar");
    }

    public static void main(String[] args) {
        System.out.println("foo");
        testMethod();
        System.out.println("baz");
    }
}
```

This prints:

```
foo
bar
baz
```

The `main` method starts by printing "`foo`." The next statement is a call to `testMethod`, so the program jumps to the beginning of `testMethod`. The statement in `testMethod` prints "`bar`." There are no more statements, so the method returns, and the program continues where it left off in `main`. `main` prints "`baz`," and because there are no more statements in `main`, `main` returns, too, ending the program.

12.3 Purposes of methods

- Abstraction:
  - Replace code that does something with a name for what is being done.
  - Change how a value is computed or an action is performed without knowing why it is computed/performed.

  - Keep the code for performing conceptually different things separate.

- Eliminating redundancy:
  - Frequently, the same action needs to be repeated in more than one context, with different data.
  - Methods let you write the code in terms of parameters instead of repeating the same statements with minor variations.

12.4 Parameters

- The type of method call in `Method1` lets us break apart a program into separate sequences of statements, but it provides no way to communicate between them.

- Here is an example of using *parameters* to communicate:

```
public class Method2 {

    static void printSquare(int x) {
        int square = x * x;
        System.out.println(square);
    }

    public static void main(String[] args) {
        java.util.Scanner scanner =
                new java.util.Scanner(System.in);
        System.out.print("Enter a number: ");
        int n = scanner.nextInt();
        printSquare(n);
    }
}
```

- This reads a number and prints its square. When `printSquare` is called, it has one parameter in parentheses after it. This corresponds to the one variable in the definition of `printSquare`, which says that it takes one parameter named `x`, which is an `int`.

- When `printSquare` is called, `n` is copied into a variable named `x` that `printSquare` can use. It is a copy, so if `printSquare` changes `x`, the `n` in `main` is not changed.

12.5 Scope #1

- Variables are only visible in the *scope* in which they are defined.

- This does not work:

```
public class Method2b {

    static void computeSquare(int x) {
        int square = x * x;
    }

    public static void main(String[] args) {
        java.util.Scanner scanner =
                new java.util.Scanner(System.in);
        System.out.print("Enter a number: ");
        int n = scanner.nextInt();
        computeSquare(n);
        System.out.println(square);
    }
}
```

because `square` is only visible inside `computeSquare`, not `main`.

- This program prints "5," not "25":

```
public class Method2c {

    static void computeSquare(int n) {
        n = n * n;
    }

    public static void main(String[] args) {
        int n = 5;
        computeSquare(n);
        System.out.println(n);
    }
}
```

- Even though both `main` and `computeSquare` have an `n`, the one in `computeSquare` is a copy, not the original, so changing it has no effect on the one in `main`.
  - n.b.: Never change parameters; it's bad style.

12.6 Return values

- Parameters give us a way to communicate from a caller to a method it calls.
- *Return values* let the called method communicate back to the caller.

- Here's an example:

```
public class ReturnValue {

    static int square(int x) {
        int xSquared = x * x;
        return xSquared;
    }

    public static void main(String[] args) {
        java.util.Scanner scanner =
                new java.util.Scanner(System.in);
        System.out.print("Enter a number: ");
        int n = scanner.nextInt();
        int n2 = square(n);
        System.out.println(n2);
    }
}
```

- The `int` before `square` says that the return value of `square` is an `int`.
- Accordingly, the value after the `return` statement has type `int`.
- And, `n2`, the variable in the caller that the return value is copied into is also an `int`.
- You don't need to store everything in a temporary variable:

```
public class ReturnValue2 {

    static int square(int x) {
        return x * x;
    }

    public static void main(String[] args) {
        java.util.Scanner scanner =
                new java.util.Scanner(System.in);
        System.out.print("Enter a number: ");
        System.out.println(square(scanner.nextInt()));
    }
}
```

  For example, since `scanner.nextInt()` returns an `int`, you can use it in any context an `int` is expected, including the argument of `square`.

- The `void` return type we'd been using previously indicates that the method does not return anything.

12.7 Calling methods in other classes

Consider these two Java source files:

```
public class ReturnMain {

    public static void main(String[] args) {
        java.util.Scanner scanner =
                new java.util.Scanner(System.in);
        System.out.print("Enter a number: ");
        int sq = ReturnOther.square(scanner.nextInt());
        System.out.println(sq);
    }
}
```

```
public class ReturnOther {

    static int square(int x) {
        return x * x;
    }
}
```

See how `main` in `ReturnMain` calls `square` in `ReturnOther` by prefixing the method name with the class name? This technique can be used to group methods that are logically related into separate files and is used in Java libraries to provide methods that any program can use.

12.8 Exercise: absolute value method

Write a method `abs` that returns the absolute value of its argument in the file `inclass/AbsExercise.java`. Here's a skeleton:

```
package inclass;

public class AbsExercise {

    // Define abs method here.

    public static void main(String[] args) {
        java.util.Scanner scanner =
                new java.util.Scanner(System.in);
        System.out.print("Enter a number: ");
        int n = scanner.nextInt();
        int absN = abs(n);
        System.out.printf("|%d| = %d\n", n, absN);
    }
}
```

12.9 Debugging errors using a backtrace

- Consider this code:

```
1  public class Backtrace {
2
3      static void doSomethingIllegal() {
4          int x = 1 / 0;  // Divide by zero!
5      }
6
7      public static void main(String[] args) {
8          doSomethingIllegal();
9      }
10 }
```

- It contains a divide-by-zero error. When run, it gives this output:

```
Exception in thread "main" java.lang.ArithmeticException: / by zero
        at Backtrace.doSomethingIllegal(Backtrace.java:4)
        at Backtrace.main(Backtrace.java:8)
```

- The first line of the error says what kind of error it was, a divide-by-zero.
- The second says that it occurred at line 4 in `doSomethingIllegal`.
- The third says that `doSomethingIllegal` was called by line 8 of `main`.
- A good debugging strategy is to start at the top and read down until you get to some code you wrote and see if you see anything wrong.

13 Arrays

13.1 What is an array?

- Arrays hold a sequence of Java values.
- Can be any single type: `int`, `String`, whatever.
- Arrays let you operate on a collection of values as a whole without putting each of them into its own variable.
- For example, arrays let you write a method that adds a sequence of numbers without the code knowing how many numbers are to be added until the program runs.
- Unlike Python lists, all elements in an array need to be the same type.
- Unlike Python lists, the size of an array is fixed on creation.
- We've already talked about `String` a bit. A `String` is very similar to an array of `char` that can't be modified.

13.2 Creating arrays in Java

- An array of int is declared like this when you want to initialize it to a specific list of values:

```
int[] x = new int[]{21, 15, 9};
```

- You can also say:

```
int[] x = new int[3];
```

  to create an array of three elements with default values (0 for integer types).

- Use this syntax to make arrays of size you don't know when you're writing the program:

```
static void myMethod(int n) {
 int[] x = new int[n];
 // and then do stuff with x
}
```

- If you don't initialize a numeric variable, it defaults to zero.
- If you don't initialize an array, it defaults to the special value `null`.

13.3 Working with arrays

- The length of an array `x` is stored in `x.length`.
- Element `i` of array `x` is stored in `x[i]`.
- You can assign to `x[i]` but not `x.length`.
- Add the first two values of an array `x`:

```
int sum = x[0] + x[1];
```

- Add all of the elements of an array `x`:

```
int sum = 0;
for (int i = 0; i < x.length; i++) {
    sum += x[i];
}
```

- Change each element of `x` to the cumulative sum of all elements up to and including that element:

```
// x[0] starts correct
for (int i = 1 ; i < x.length; i++) {
    x[i] = x[i - 1] + x[i];
}
```

13.4 Arrays are references

- If an int is four bytes, does that mean that an int[3] is twelve?
- This code:

```
int[] x = new int[]{0, 1, 2};
int[] y = x;
System.out.println(x[1]);
y[1] = 17;
System.out.println(x[1]);
```

  will print:

```
1
17
```

- An array is a *reference type*.

- One way to think about it is that every array (of any type) is a single number identifying a location in memory where the array elements are stored.
- Assign one array to another, and you copy the reference.
- Change the underlying data through one reference, and you'll see the changes if you access them through the original array reference.
- The `new` operator claims a chunk of memory and returns its location.

13.5 Passing arrays to methods

- If you pass an array to a method, what is being passed is the reference.
- This code:

```
public class ArrayRef {

    static void changeArrayReference(int[] a) {
        a = new int[]{4};
    }

    static void changeArrayElement(int[] b) {
        b[0] = 4;
    }

    public static void main(String[] args) {
        int[] x = new int[]{1, 2, 3};
        changeArrayReference(x);
        System.out.println(x[0]);
        changeArrayElement(x);
        System.out.println(x[0]);
    }
}
```

  outputs this:

```
1
4
```

- `changeArrayReference` changes which chunk of memory `a` points to, but it does not change the contents of the original array, so `main` sees no changes to `x`.
- `changeArrayElement` changes an element of `b`. Since `b` and `x` point to the same chunk of memory, changing an element is visible to code that accesses the array through the other.

13.6 Exercise: containsElement

Write a method `containsElement` that takes two arguments, a `char` array and a `char`. It should return `true` if the array contains the `char`, and `false` if it does not. Put it in `inclass/Contains.java` and submit it with Subversion. Here's a skeleton with some test code:

```
package inclass;

public class Contains {

    static boolean containsElement(char[] a, char c) {
        // YOUR CODE HERE
    }

    public static void main(String[] args) {
        char[] a = new char[]{'a', 'b', 'c'};
        System.out.printf("contains 'b': %b\n",
                          containsElement(a, 'b'));
        System.out.printf("contains 'd': %b\n",
                          containsElement(a, 'd'));
    }
}
```

I recommend writing a `for` loop that runs from 0 up to but not including `a.length` and checking whether the element of `a` with that index is equal to `c`.

14 Type conversions

2013–02–25

- We can assign a `char` to an `int`:

```
char c = 28;
int k = c;
```

- We can always copy a `char` value into an `int` because every valid `char` is also a valid `int`.
- `char` values range from 0 to 65535; `int` values range from −2147483648 to 2147483647, a superset of `char` values.
- The same goes for assigning `int` to `double`, `byte` to `short`, etc.
- If we want to go the other way, information might be lost.
- What is would `d` be?

```
int m = -4;
char d = m;  // Compiler error!
```

- We can force the compiler to convert with an explicit type conversion:

```
int n = 4;
char e = (char) m;  // e is now 4.
```

- Whenever the `int` value is also a valid `char` value, the `char` gets that value.
- Otherwise, it is the remainder mod 2^{16}.
- Here's an example with `double` values:

```
double x = -5.0;
double y = 8.125;
int a = (int) x;
int b = (int) y;
```

  Here, `a` is −5, as we would hope. `y` is rounded towards zero to get 8.

15 Classes

15.1 Objects

- An *object* is a collection of data that can be referred to by reference.
- An array is a particular type of object, in which each element of data is the same type, but there can be any number of elements.
- Other objects can have different structure.
- For example, a `Rectangle` object may be composed of exactly two `int` elements, a width and a height.

```
class Rectangle {
    int width;
    int height;
}
```

- Or, a street address may be composed of an `int` street number, a `String` street name, and an `int` unit number.

```
class StreetAddress {
    int streetNumber;
    String streetName;
    int unitNumber;
}
```

15.2 Classes

- A *class* is a description of an object.
- A *class definition* like this describes what goes into an object:

```
class Rectangle {
    int width;
    int height;
}
```

- Arrays don't need explicit definitions because they only contain one kind of data.
- When we create an object, we say that it is an *instance* of a class.
- There is only one `Rectangle` class, which describes what a rectangle is.
- But, there can be any number of `Rectangle` objects, each of which has its own width and height.

- We call the individual parts of an object its *members*.
- An object member is like an array element.
- In fact, we can talk about array members and object elements and no one will wonder what we mean, it's just a bit more common to refer to the parts of arrays as elements and the parts of non-array objects as members.

15.3 Creating, accessing, and modifying objects

- Non-array objects work similarly to arrays.
- Objects are created with `new` (with parentheses after the class name):

```
Rectangle r = new Rectangle();
```

  This is much like creating an array:

```
int[] a = new int[2];
```

- Members can be read with the same "dot" notation used to read the length from an array:

```
System.out.printf("Width is %d.\n", r.width);
```

- Members can be changed this way, too:

```
r.width = 2;
```

  You can't do this with the `length` of an array—arrays are special in that way.

15.4 Objects and primitive types

- References to any kind of object work like array references.
- This program:

```
public class ClassRef {

    static void changeClassReference(Rectangle a) {
        a = new Rectangle();
        a.height = 1;
    }

    static void changeClassMember(Rectangle b) {
        b.height = 7;
    }

    public static void main(String[] args) {
        Rectangle r = new Rectangle();
        r.width = 3;
        r.height = 5;
        System.out.printf(
            "Initial: h=%d, w=%d\n", r.width, r.height);
        changeClassReference(r);
        System.out.printf(
            "After changeClassReference: h=%d, w=%d\n",
            r.width, r.height);
        changeClassMember(r);
        System.out.printf(
            "After changeClassMember: h=%d, w=%d\n",
            r.width, r.height);
    }
}
```

generates this output:

```
Initial: h=3, w=5
After changeClassReference: h=3, w=5
After changeClassMember: h=3, w=7
```

- Observe that `changeClassReference` changes its local `a` variable to point to a new `Rectangle`, so `r` in `main` is not changed.
- However, the reference `b` passed to `changeClassMember` is a copy of the `r` in `main`, so they refer to the same `Rectangle` object.
- Changes to the object in `changeClassMember` are visible when the object is accessed through the reference `r` in `main`.

15.5 Primitive types

- Some types don't have "contents."

- These are called *primitive types*.
- In Java, these include `int`, `char`, and `boolean`.
- The others are `double`, `float`, `long`, `short`, and `byte`, which we don't use much in this class.
- Primitive types begin with a lowercase letter.
- Non-array class names conventionally begin with an uppercase letter.
- The type of all Java data is either primitive or a class.
- Objects are created based on a class definition with `new` and are referred to by reference.
- Primitive variables are all pre-defined in Java.
- If you come up with a new type of thing, it has to be defined with a class.

15.6 Exercise: area of a rectangle

Create a Java program in the file `inclass/RectArea.java` that has a method `rectArea` that takes a `Rectangle` as an argument and returns the area of the rectangle as an `int`. You can use this definition of a `Rectangle`. (It should go in `inclass/Rectangle.java`.)

```
package inclass;

class Rectangle {
    int height;
    int width;
}
```

Since your program will be in the `inclass` package and `Rectangle` is in the `inclass` package, you can refer to it as just `Rectangle` from `RectArea`. You can start with this partial definition for `RectArea`. (It should go in `inclass/RectArea.java`.)

```
package inclass;

public class RectArea {

    // Define rectArea here.

    public static void main(String[] args) {
        Rectangle testRect = new Rectangle();
        testRect.height = 2;
        testRect.width = 9;
        System.out.printf("The area of testRect is %d.\n",
                          rectArea(testRect));
    }
}
```

16 Non-static methods

16.1 Introductory example

```
class RectangleWithMethod {

    int height;
    int width;

    int area() {
        return width * height;
    }
}
```

- If you understand as much of classes and objects as we've talked about so far, it's you can guess what this does.
- But, how and why?
- So far, we've put `static` in front of every method declaration.
- What does it mean to take it away?
- If `area` were a `static` method, we could call with `RectangleWithArea.area()`.
- That would be an error because `width` and `height` would not be defined.
- We can do this with the above class, though:

```
RectangleWithArea rect = new RectangleWithArea();
rect.width = 3;
rect.height = 4;
int a = rect.area();
```

- Instead of calling the method with the `class` name to the left of the dot, we call the method with the `object` name to the left of the dot.
- When we call a method this way, it can refer to all data fields of the object as if they were variables declared in the method.
- Changes to variables persist after the method returns.

```
class Counter {

    int n;

    void increment() {
        n = n + 1;
    }
}
```

```
public class CounterMain {

    public static void main(String[] args) {
        Counter c = new Counter();
        c.n = 4;
        c.increment();
        System.out.println(c.n);
    }
}
```

Output:

```
$ java CounterMain
5
```

- In `Counter.increment`, there is no type before `n`, so it must be referring to an already-declared variable.
- Changing it in a method changes is permanent; it changes the copy in the object, which lasts after the method exits.
- So, when `increment` changes `n`, the changed value is visible from `CounterMain.main`.

17 Nested objects

17.1 Introduction

You can put objects inside objects and arrays inside arrays and objects inside arrays and arrays inside objects. Here's a class definition we will use:

```
class Chain {
    int x;
    Chain next;
}
```

17.2 null

- `null` is like zero for objects.
- Any type of object can be set to `null`.

  ```
  Chain c = null;
  ```

- Objects are references; `null` points to no object at all.
- It is an error to access a member of a `null` object reference. Even though there is a name (the variable), there is no associated object, sort of like a disconnected phone number.
- This code:

  ```
  class ChainNull {
      public static void main(String[] args) {
          Chain c = null;
          c.x = 5;
      }
  }
  ```

  generates this error:

  ```
  Exception in thread "main" java.lang.NullPointerException
          at ChainNull.main(ChainNull.java:4)
  ```

17.3 Uninitialized values

- When you create an object and read a numeric value before writing it, it has the value zero. This:

  ```
  Chain c = new Chain();
  System.out.println(c.x);
  ```

outputs:

```
0
```

because `c.x` is initialized to 0.

- Since `null` is like zero for objects, this code:

```
Chain c = new Chain();
System.out.println(c.next);
```

outputs:

```
null
```

17.4 Objects inside objects

- We can reference objects from inside other objects:

```
class ChainLink {
    public static void main(String[] args) {
        Chain c = new Chain();
        Chain d = new Chain();
        c.next = d;
        c.x = 5;
        c.next.x = 8;
        System.out.printf("c.x = %d\nd.x = %d\n", c.x, d.x);
    }
}
```

This outputs:

```
c.x = 5
d.x = 8
```

because `d` and `c.next` refer to the same object.

- We can even make objects that reference themselves:

```
class ChainLoop {
    public static void main(String[] args) {
        Chain c = new Chain();
        c.next = c;
        c.x = 5;
        System.out.printf("c.x = %d\n", c.x);
        System.out.printf("c.next.x = %d\n", c.next.x);
        System.out.printf("c.next.next.next.next.x = %d\n",
            c.next.next.next.next.x);
    }
}
```

This outputs:

```
c.x = 5
c.next.x = 5
c.next.next.next.next.x = 5
```

17.5 Objects inside arrays

2013–03–04

- You can create an array of objects just like an array of primitive variables:

```
Chain[] chains = new Chain[2];
chains[0] = new Chain();
chains[1] = new Chain();
chains[0].x = 5;
chains[1].x = 7;
```

- If you don't initialize the values of an object array, they are `null`:

```
Chain[] chains = new Chain[2];
System.out.printf("chains[0] = %s\n", chains[0]);
```

outputs:

```
chains[0] = null
```

- Object array elements are references just like member variables and method local variables.
- More than one array element can point to the same place:

```
Chain[] chains = new Chain[2];
chains[0] = new Chain();
chains[1] = chains[0];
chains[0].x = 3;
System.out.println("chains[0].x = %d\n", chains[0].x);
System.out.println("chains[1].x = %d\n", chains[1].x);
```

outputs:

```
chains[0].x = 3
chains[1].x = 3
```

because both elements of `chains` refer to the same object.

17.6 Exercise

Write a method, `largest` that, given an array of `Chain` objects as defined above, returns the largest of their `x` fields. Here are some skeleton files to start you out.

`inclass/Chain.java`

```
package inclass;

class Chain {
    int x;
    Chain next;
}
```

`inclass/ChainLargest.java`

```
package inclass;

public class ChainLargest {

    public static int largest(Chain[] chains) {
        // Finish this.
        return Integer.MIN_VALUE;
    }

    public static void main(String[] args) {
        Chain[] chains =
            new Chain[]{new Chain(), new Chain(), new Chain()};
        chains[0].x = 0;
        chains[1].x = 7;
        chains[2].x = -2;
        int largestX = largest(chains);
        System.out.printf("The largest x is %d.\n", largestX);
        System.out.println("It should be 7.");
    }
}
```

18 Arrays inside arrays

18.1 Introduction

- Arrays are objects, so you can put arrays inside other arrays.
- Arrays themselves are objects, so they can be `null`.
- Array elements can only be `null` if it is array of some object type, not a primitive type like `boolean`.
- An `int[][]` is an array of `int[]`.
- A `String[][][][]` is an array of `String[][][]`!

18.2 Creating multidimensional arrays

- Recall that you can initialize a single dimensional array with braces:

```
int[] x = new int[]{ 21, 15, 9 };
```

- You can define a multidimensional array with nested braces.

```
int[][] x = new int[][]{
    new int[]{3, 21},
    new int[]{2, 15},
    new int[]{1, 9}
};
```

- This is the same as:

```
int[][] x = new int[3][2];
x[0][0] = 3;
x[0][1] = 21;
x[1][0] = 2;
x[1][1] = 15;
x[2][0] = 1;
x[2][1] = 9;
```

- If we just called `new` and didn't initialize the values, they would be zero (or `null`).
- There is a shorthand that hides the explicit `new`, but means the same thing:

```
int[][] x = { { 3, 21 }, { 2, 15 }, { 1, 9 } };
```

- As with other arrays, the dimensions can be computed at runtime:

```
int n = howManyArrays();
int v = howManyElementsInSubArray();
char[][] array = new int[n][v];
// and then do stuff with array
```

18.3 References in multidimensional arrays

In this example, you can see that by copying the reference `x[2]` to `x[1]`, changes to `x[1]` are visible through `x[2]` because they refer to the same object.

```
import java.util.Arrays;

public class MultiDimRef {

    public static void main(String[] args) {
        int[][] x = {{3, 21}, {2, 15}, {1, 9}};
        System.out.println("A: x[1] = " + Arrays.toString(x[1]));
        System.out.println("A: x[2] = " + Arrays.toString(x[2]));
        x[1] = x[2];
        x[1][0] = 4;
        System.out.println("B: x[1] = " + Arrays.toString(x[1]));
        System.out.println("B: x[2] = " + Arrays.toString(x[2]));
    }
}
```

Output:

```
A: x[1] = [2, 15]
A: x[2] = [1, 9]
B: x[1] = [4, 9]
B: x[2] = [4, 9]
```

18.4 Ragged arrays

In an array that contains other arrays, there is no reason that their lengths need be the same.

```
boolean[][] bb = {
    {false, false, true},
    {false, true},
    {true},
    {}
};
```

In this code, `bb.length` is 4, `bb[0].length` is 3, `bb[1].length` is 2, `bb[2].length` is 1, and `bb[3].length` is 0.

18.5 Exercise

Write a method, `transpose`, that transposes a matrix, switching the rows and columns:

$$\text{transpose} \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix} = \begin{bmatrix} 1 & 4 \\ 2 & 5 \\ 3 & 6 \end{bmatrix}$$

Here's a skeleton of some code, `inclass/MatrixTranspose.java`:

```
package inclass;

import java.util.Arrays;

public class MatrixTranspose {

    static double[][] transpose(double[][] matrix) {
        return matrix;  // Fix this!
    }

    public static void main(String[] args) {
        double[][] matrix = {
            {1, 2, 3},
            {4, 5, 6},
        };
        System.out.printf("matrix = %s\n",
                          Arrays.deepToString(matrix));
        System.out.printf("transpose(matrix) = %s\n",
                          Arrays.deepToString(transpose(matrix)));
    }
}
```

You can assume that the array passed to `transpose` is not null, that it has at least length one, that all its elements have the same length, and that that length is at least one.

19 Constructors

2013-03-06

19.1 Initializing objects and factory methods

- Consider this class for complex numbers:

```
class Complex0 {
    double real;
    double imag;
}
```

- It's awkward to have to initialize every field of a new object manually, like this:

```
Complex0 c = new Complex();
c.real = 4.0;
c.imag = -0.5;
```

- How do we know we remembered to initialize everything?
- How do we know our settings are consistent?
- We can write a static method that does it for us, as in:

```
class Complex1 {
    double real;
    double imag;

    static Complex1 makeComplex(double real, double imag) {
        Complex1 c = new Complex1();
        c.real = real;
        c.imag = imag;
        return c;
    }
}
```

  Then, we can create the new object with:

```
Complex1 c = makeComplex(4.0, -0.5);
```

- Such a method is called a *factory method*.
- They're pretty common.

19.2 Constructor introduction

- But, there's a more common pattern in Java, the *constructor*:

```
class Complex2 {
    double real;
    double imag;

    Complex2(double real, double imag) {
        this.real = real;
        this.imag = imag;
    }
}
```

- A constructor is like regular method, but it has the same name as the class and does not have a return value.
- The parameters of the constructor must be passed in the parameters of the `new` operator, like this:

```
Complex2 c = new Complex2(4.0, -0.5);
```

- If you don't define a constructor for a class, Java creates one for you that takes no parameters and does nothing.
- If you do define a constructor, this default constructor is not created, so in `Complex2`, this is illegal:

```
Complex2 c = new Complex2();
```

- Now we can see why we were always putting `()` after class names when we called `new`, as in:

```
Complex1 c = new Complex1();
```

  We were calling the automatically generated constructor, which has no arguments.

19.3 Multiple constructors

- We can define multiple constructors for the same class as long as they have different numbers or types of arguments so Java can figure out which one we're calling.
- For example, this class definition for `Complex3` lets you omit the `imag` argument if you want to create a complex number with no imaginary component.

```
class Complex3 {
    double real;
    double imag;

    Complex3(double real, double imag) {
        this.real = real;
        this.imag = imag;
    }

    Complex3(double real) {
        this.real = real;
        this.imag = 0;
    }
}
```

19.4 Exercise: representing money

Write a class called `inclass.Money`. It should have a single field, `value`, representing a quantity of money in cents. It should have a single constructor that takes two arguments, `dollars` and `cents`. The `value` member should be initialized to 100 times `dollars` plus `cents`. Don't worry about negative numbers.

20 Inheritance

2013–03–11

20.1 Extending a class

- Suppose you have a class `Address`:

```
class Address {
    String streetAddress;
    String city;
    String country;
}
```

  and you want to add a zipcode and state, but those only make sense in the US. We could do something like this:

```
class USAddress1 {
    String streetAddress;
    String city;
    String country;
    int zipcode;
    String state;
}
```

- A problem with this approach is that if we have to copy all the fields and methods from one definition to another.
- Keeping two mostly-identical classes synchronized is annoying and error prone.
- Instead, we can `extend` a class:

```
class USAddress2 extends Address {
    int zipcode;
    String state;
}
```

- Note the `extends Address` after the class name (`USAddress2` in the class definition.
- Since `USAddress2` extends `Address`, it has all of the fields of `Address`, too.
- If `Address` had methods, it would get those too. Since they would reference fields of `Address` and `USAddress` has all the fields of `Address`, they would still work fine.

20.2 Exercise: Canadian addresses (part 1)

Canadian addresses are like US addresses, but they use a six-character postal code instead of a five-number zip code. So, we can represent them with a `String`. Define a class `inclass.CanadaAddress` that extends `testing.predefined.Address` and adds `postalCode` and `province` members.

`testing.predefined.Address` is like the `Address` above, just in the `testing.predefined` package. You can make it available by running `svn update`.

I wrote a test class that will try your class out. You can run it with:

```
javac testing/inclass/TestCanada.java
java testing.inclass.TestCanada
```

Mark both the class `CanadaAddress` and its members `public` or else `TestCanada` will not be able to access them.

20.3 Inheriting methods

- Here is another variant on the address classes:

```
class AddressM {
    String streetAddress;
    String city;
    String country;

    public String toString() {
        return String.format("%s\n%s\n%s\n", streetAddress,
                             city, country);
    }
}
```

```
class USAddressM extends AddressM {
    int zipcode;
    String state;
}
```

- `USAddressM` extends `AddressM`, so any method that can be called on `AddressM` can also be called on `USAddressM`.

- For example:

```
public class AddressTestM {

    public static void main(String[] args) {
        USAddressM a = new USAddressM();
        a.streetAddress = "251 Mercer St.";
        a.city = "New York";
        a.state = "NY";
        a.zipcode = 10012;
        a.country = "USA";
        System.out.print(a.toString());
    }
}
```

This outputs:

```
251 Mercer St.
New York
USA
```

20.4 Overriding methods

- That doesn't look quite right. We need to provide the subclass with its own version of `toString`.

```
class USAddressM2 extends AddressM {
    int zipcode;
    String state;

    @Override public String toString() {
        return String.format(
            "%s\n%s, %s %05d\n%s\n", streetAddress, city,
            state, zipcode, country);
    }
}
```

- Now if we run this test program:

```
public class AddressTestM2 {

    public static void main(String[] args) {
        USAddressM2 a = new USAddressM2();
        a.streetAddress = "251 Mercer St.";
        a.city = "New York";
        a.state = "NY";
        a.zipcode = 10012;
        a.country = "USA";
        System.out.print(a.toString());
    }
}
```

we get this output:

```
251 Mercer St.
New York, NY 10012
USA
```

- Since the subclass provided a version of the method, that was called instead of the version from the superclass.
- Note the `@Override`. It is optional, but is good style.
- `@Override` causes the compiler to flag an error if the method does not override another method.
- This protects you from a misspelling or other minor difference causing the wrong version of the method to run.

20.5 Exercise: Canadian addresses (part 2)

Just like American addresses, Canadian addresses should be formatted to include the postal code. Override `CanadaAddress.toString()` so that `TestCanada` outputs:

```
2 Bloor Street East
Toronto, Ontario M4W 3H8
Canada
```

instead of:

```
2 Bloor Street East
Toronto
Canada
```

21 Public and private

- The Java modifiers `public`, `private`, and `protected` can be used to expose a stable public interface for others to use and while allowing you to freely change the implementation details.
- This helps you add features and fix bugs without breaking other code that uses yours.
- Mostly useful in much larger projects than this class, so we won't go into much detail.
- They keep coming up, so I'll just tell you what the modifiers mean.
- `public`
  - Any class can define a variable of a class marked `public`.
  - Any class can extend a class marked `public`.
  - Any class can call a method marked `public` (in a `public` class).
  - Any class can read and modify a field marked `public` (in a `public` class).
  - A `public` class with a `public` field:

```
package inclass;
public class A {
    public char c;
}
```

- `private`
  - Regular classes cannot be defined `private`.
  - Methods marked `private` can only be called by other methods of that class (including `public` ones).
  - Fields marked `private` can only be read or modified by other methods of that class.
  - It is good style to mark fields `private` and only access them through methods to make sure other classes don't make the state of an object inconsistent.
- "package private"
  - If you don't specify `public` or `private`, the access is "default," also called "package-private."
  - This is somewhere between `public` and `private`.
  - Classes with default access can be used and extended only by other classes in the same package.

  - Methods with default access can be called only by other classes in the same package.
  - Fields with default access can be accessed only by other classes in the same package.
  - A package-private/default class with a package-private/default field:

```
package inclass;
class B {
    char d;
}
```

- `protected`
  - `protected` is somewhere between `public` and default access.
  - `protected` methods and fields can also be accessed by the subclasses of a class.
  - In my opinion, it almost always indicates bad style.
- See Controlling Access to Members of a Class for details.
- General rule: If classes other people wrote should be able to access something, mark it `public`. Otherwise, use default or `private` (but be consistent).
- A class with a `main` method and the `main` method itself must always be `public`.

22 Dynamic and static types

2013–03–13

22.1 Terms

- Any piece of data in Java has two types:
  - Dynamic ⇔ runtime
  - Static ⇔ compile-time
- So far in this course, these two types have always been the same.

22.2 Static type

- An object's static type is the type of the variable holding it.
- No matter what is on the right side of the `=`, `v` has type `Foo`.

```
Foo v = ...;
```

22.3 Dynamic type

- An object's dynamic type is the type it is created as.
- No matter what the `...` is, `v` has type `Bar`.

```
... v = new Bar();
```

- Here, as usual:

```
Qux v = new Qux();
```

  we assign a value of dynamic type `Qux` into a variable of type `Qux`, so the static and dynamic types are the same.

22.4 When static and dynamic types differ

- Here are two boring classes:

```
class A {
}
```

```
class B extends A {
}
```

- You can always store a specific type in a more general variable:

```
A x = new B();  // legal, B is a subtype of A
B y = new A();  // compiler error, A is not a subtype of A
```

- `B` extends `A`, so every `B` is also an `A`, just a specific kind of `A`.
- Since a `B` is an `A`, we can put a reference to a `B` object in a variable with type `A`.
- Some `A` objects are `B` objects, but not all of them are, so we can't assign a `A` object reference (which may or may not be a `B`) to an `A` variable.

22.5 A nontrivial example

- Last time we used these two classes:

```
class AddressM {
    String streetAddress;
    String city;
    String country;

    public String toString() {
        return String.format("%s\n%s\n%s\n", streetAddress,
                             city, country);
    }
}
```

```
class USAddressM2 extends AddressM {
    int zipcode;
    String state;

    @Override public String toString() {
        return String.format(
            "%s\n%s, %s %05d\n%s\n", streetAddress, city,
            state, zipcode, country);
    }
}
```

- A `USAddressM2` is a kind of `AddressM`, so this works fine:

```
AddressM a1 = new USAddressM2();
```

- This doesn't, though:

```
USAddressM2 a2 = new AddressM();
```

- The `new` didn't create any space for `zipcode` or `state`, so what would this even do?

```
USAddressM2 a2 = new AddressM();
System.out.println(a2.zipcode);
```

22.6 Casting

- What if we're really sure we should be able to do the assignment?
- This ought to work, right?

```
USAddressM2 a2 = new USAddressM2();
a2.zipcode = 10003;
AddressM b2 = a2;  // Legal, a2 is a kind of AddressM.
System.out.println(b2.zipcode);
```

- `b2` has a `zipcode` field, but this is a compiler error because the static type of `b2`, `AddressM`, determines what is legal to do to it.
- We can promise the Java compiler that what we're doing will work:

```
1 USAddressM2 a2 = new USAddressM2();
2 a2.zipcode = 10003;
3 AddressM b2 = a2;  // Legal, a2 is a kind of AddressM.
4 USAddressM2 c2 = (USAddressM2) b2;  // Trust me!
5 System.out.println(c2.zipcode);
```

- This construct on line 4:

```
Type var = (Type) value;
```

  lets you turn a value of one static type to a value of another static type.

- On line 4, we're turning a value of static type `AddressM` to a value of static type `USAddressM2`.
- This is called "casting," and parentheses with a type name in them are called the "cast operator."
- It is a unary operator like "`!`"
- It is not the same as the similar-looking operation applied to primitive types.
- "`(int) k`" converts `k` to an `int`.
- "`(USAddressM2) x`" just changes the static type of `x`.

22.7 Bad casts

- Casting changes the static type of an object, not the dynamic type of an object.
- Static types are just a compile-time label.
- Dynamic types are actually what kind of object you have.
- So, this does not work:

```
1 AddressM x = new AddressM();
2 USAddressM2 y = (USAddressM2) x;  // Trust me!
3 System.out.println(b2.zipcode);
```

- We tried to get a variable of type `USAddressM2` when the underlying dynamic type was just `AddressM`.
- This will compile, since we told javac to trust us.
- When we run the program, we'll get a `ClassCastException` on line 2.
- Java will notice we asked it do something impossible, and crash.

22.8 Exercise: Legal or illegal? (part 1)

We have two classes:

```
class X {
    int j;
}
```

```
class Z extends X {
    int k;
}
```

Which are legal?

1. `X a = new Z();`
2. `Z b = new X();`
3. `X c = (X) new Z();`
4. `Z d = (Z) new X();`
5. `Z c = (X) new Z();`
6. `X c = (Z) new X();`

22.9 Exercise: Legal or illegal? (part 2)

2013–03–25

The same two classes as last time.

```
class X {
    int j;
}
```

```
class Z extends X {
    int k;
}
```

Which of these are legal?

1. `(new X()).j = 5;`
2. `(new Z()).j = 5;`
3. `(new X()).k = 5;`
4. `(new Z()).k = 5;`
5. `((X) new Z()).k = 5;`
6. `((Z) new X()).k = 5;`
7. `((X) new Z()).j = 5;`
8. `((Z) new X()).j = 5;`

23 Dynamic method dispatch and overriding

23.1 Multiple methods with the same name

- Let's look at three classes we've used before:

```
class AddressM {
    String streetAddress;
    String city;
    String country;

    public String toString() {
        return String.format("%s\n%s\n%s\n", streetAddress,
                             city, country);
    }
}
```

```
class USAddressM extends AddressM {
    int zipcode;
    String state;
}
```

```
class USAddressM2 extends AddressM {
    int zipcode;
    String state;

    @Override public String toString() {
        return String.format(
            "%s\n%s, %s %05d\n%s\n", streetAddress, city,
            state, zipcode, country);
    }
}
```

- Suppose we have these statements:

```
USAddressM a1 = new USAddressM();
...
System.out.println(a1.toString());
```

  Just like `a1.city` refers to the `city` field that `USAddressM` inherits from `AddressM`, `a1.toString()` refers to the `toString` that it inherits from `AddressM`.

- Now consider these statements:

```
USAddressM2 a2 = new USAddressM2();
...
System.out.println(a2.toString());
```

In this case, we have *overridden* `AddressM.toString()` with `USAddressM2.toString()`, so `a2.toString()` calls the one in `USAddressM2`.

23.2 Dynamic dispatch

- Now look at these:

```
AddressM a3 = new USAddressM2();
...
System.out.println(a3.toString());
```

- This legal Java. We can assign a `USAddressM2` to a variable of type `AddressM` because `USAddressM2` extends `AddressM`. And, `AddressM` provides a `toString`, so we can call `toString` on `a3`.
- The question is, which gets called, the `toString` in `AddressM`, the static type of `a3`, or the `toString` in `USAddressM2`, the dynamic type of `a3`?
- Different languages do it differently.
- In Java, it is always the dynamic type.
- The static type is for the compiler. It determines what methods are legal to call.
- The dynamic type is for the runtime. It determines what actual implementation gets called.
- A consequence of this is that type casting does not change which method gets called. It just makes compiler errors go away.

23.3 java.lang.Object

- If a class doesn't explicitly extend a class, it extends `java.lang.Object`.
- Every class extends `java.lang.Object` directly or indirectly.
- Even if a class doesn't define any methods at all, it inherits some from `java.lang.Object`.
- The two most interesting are `toString` and `equals`.
- We override these often since the provided ones don't work well.

23.4 toString

- `toString` is straightforward; we've already overridden it several times.
- A `toString` override always has the signature:

```
@Override public String toString() { ...
```

- It is called from many places in Java, such as `System.out.printf` with the `%s` specifier.
- The default returns garbage like "`AddressM@eb42cbf`."
- So, override `toString` often.

24 equals

2013–03–27

24.1 == versus equals

- `equals` is more interesting than `toString`.
- It has the signature:

```
@Override public boolean equals(Object obj) { ... }
```

- Object variables are really just references, and `==` compares references.
- This code:

```
import java.util.Arrays;

class EqEqDemo {
    public static void main(String[] args) {
        String s = new String("foo");
        String t = new String("foo");
        System.out.printf("s == t: %b\n", s == t);
        System.out.printf("s.equals(t): %b\n", s.equals(t));

        int[] a = new int[]{1, 2, 3};
        int[] b = new int[]{1, 2, 3};
        System.out.printf("a == b: %b\n", s == t);
        System.out.printf("a.equals(b): %b\n", a.equals(b));
        System.out.printf("Arrays.equals(a, b): %b\n",
                          Arrays.equals(a, b));
    }
}
```

  generates this output:

```
s == t: false
s.equals(t): true
a == b: false
a.equals(b): false
Arrays.equals(a, b): true
```

- When we create two distinct new objects, they are never "`==`" to each other.
  - As a result, `s == t` and `a == b` are `false`.
- The `equals` method of `java.lang.Object` just calls `equals`, so unless we override it, it's not any more useful.

- `String` overrides `equals` to return `true` if the argument to `equals` is a `String` and has the same characters as `this`.
  - As a result, `s.equals(t)` is `true`.
- Arrays do extend `Object`, but there's no way to override methods in an array, so they just get the `equals` from `Object`.
  - As a result, `a.equals(b)` is `false`.
- There is a utility class, `java.util.Arrays`, that provides static `equals` methods that take two arguments and returns `true` if they have the same number of elements and the elements compare equal to each other with `==` (for arrays of primitive types) and `equals` (for arrays of objects).
  - This behaves more like you might expect.
  - As a result, `Arrays.equals(a, b)` is `true`.

24.2 A first try

- Here's a try at overriding `equals`:

```
class AddressEqBad {
    String streetAddress;
    String city;

    public AddressEqBad(String streetAddress, String city) {
        this.streetAddress = streetAddress;
        this.city = city;
    }

    public boolean equals(AddressEqBad that) {
        return streetAddress.equals(that.streetAddress)
            && city.equals(that.city);
    }
}
```

- We can try it with this:

```
AddressEqBad a1 = new AddressEqBad("251 Mercer", "NY");
AddressEqBad a2 = new AddressEqBad("251 Mercer", "NY");
System.out.println(a1.equals(a2));
```

  This prints `true`, as we would hope.

- Let's try this one:

```
AddressEqBad b1 = new AddressEqBad("251 Mercer", "NY");
Object b2 = new AddressEqBad("251 Mercer", "NY");
System.out.println(b1.equals(b2));
```

This prints `false`!

- The `equals` method of `Object` is getting called.
- What's going on? Didn't we say that the dynamic type of an object determines what methods get called?
- The problem is `equals` should be:

```
public boolean equals(Object) { ... }
```

but we said:

```
public boolean equals(AddressEqBad) { ... }
```

- Now, there are *two* `equals` methods with different arguments.
- In `b1.equals(b2)`, the static type of `b2` is used to determine which set of arguments is right.
- The only one compatible with the static type of `b2` is the one that takes an `Object`, so that one gets called.

24.3 Dynamic dispatch and equals

- Method selection using dynamic types only happens when methods override each other.
- Here's a second try:

```
class AddressEqBad2 {
    String streetAddress;
    String city;

    AddressEqBad2(String streetAddress, String city) {
        this.streetAddress = streetAddress;
        this.city = city;
    }

    @Override public boolean equals(Object that) {
        AddressEqBad2 thatAddr = (AddressEqBad2) that;
        return streetAddress.equals(thatAddr.streetAddress)
            && city.equals(thatAddr.city);
    }
}
```

- `@Override` causes `javac` to give us an error if we are adding a new method instead of overriding an existing one.
- Use `@Override` to catch errors in method definitions.
- We take `Object` as an argument to `equals` so that it matches the definition of `Object.equals(...)`.
- Let's try it out:

```
AddressEqBad2 c1 = new AddressEqBad2("251 Mercer", "NY");
Object c2 = new AddressEqBad2("251 Mercer", "NY");
System.out.println(c1.equals(c2));
```

- This prints `true`!
- There is only one `equals`, so that gets called regardless of the static type of its argument.
- That is legal, since every class is a subclass of `Object`.
- Let's try the other way around:

```
Object d1 = new AddressEqBad2("251 Mercer", "NY");
AddressEqBad2 d2 = new AddressEqBad2("251 Mercer", "NY");
System.out.println(d1.equals(d2));
```

- This prints `true`, too.
- Since `Object` has an `equals` method that takes an `Object`, and `AddressEqBad2` is a subclass of `Object`, it is legal to write `d1.equals(d2)`.

- Since the dynamic type of `d1` is `AddressEqBad2`, the `equals` in `AddressEqBad2` is the actual method that gets called.

24.4 instanceof

- An interjection: The `instanceof` can be used to check the dynamic type of an object.
- Its left side is an object; its right side is a class.
- These are all true:

```
"foo" instanceof String
"foo" instanceof Object
new int[1] instanceof Object
```

- These are false:

```
"foo" instanceof java.util.Scanner
new int[1] instanceof String
```

24.5 A final try at equals

- That last one worked pretty well.
- One problem remains.
- This code:

```
Object e1 = new AddressEqBad2("251 Mercer", "NY");
System.out.println(e1.equals("foo"));
```

  generates this error:

```
Exception in thread "main" java.lang.ClassCastException:
java.lang.String cannot be cast to AddressEqBad2
```

- Even though `AddressEqBad2.equals()` takes an `Object` argument, it casts it to `AddressEqBad2`, so it crashes if given any other type of object.

- We can fix this with `instanceof`:

```
class AddressEqGood {
    String streetAddress;
    String city;

    AddressEqGood(String streetAddress, String city) {
        this.streetAddress = streetAddress;
        this.city = city;
    }

    @Override public boolean equals(Object that) {
        if (that instanceof AddressEqGood) {
            AddressEqGood thatAddr = (AddressEqGood) that;
            return streetAddress.equals(thatAddr.streetAddress)
                && city.equals(thatAddr.city);
        } else {
            return false;
        }
    }
}
```

- Now these work:

```
Object f1 = new AddressEqGood("251 Mercer", "NY");
Object f2 = new AddressEqGood("251 Mercer", "NY");
System.out.println(f1.equals("foo"));
System.out.println(f1.equals(f2));
```

- That code prints:

```
false
true
```

- Since `null instanceof Something` is always `false`, `AddressEqGood.equals(null)` returns `false`, as we would hope.

24.6 Exercise: equals

Write a class `OneInt` that contains a single `int` member, `n` and has an `equals` method that returns `true` if the argument is another `OneInt` with the same value for `n`.

```
package inclass;

class OneInt {
    int n;

    // Put equals method here.
}
```

25 Interfaces and Comparable

2013–04–01

25.1 Motivation

- Suppose we want to write a method that sorts objects, but we don't want to rewrite it for every type of object.
- To sort things, one needs to be able to determine whether one object is less than, equivalent to, or greater than another object.
- `Object` has `equals`, but no `lessThan` or `greaterThan`.
- `<`, `>`, `<=`, and `>=` do not work on references.
- How can we specify a generic type for objects with particular methods?
- We could force all of them to extend a common class with those methods.
  - Some languages do this!
- What happens when a subclass supports these methods but the parent does not?
  - Maybe "less than" makes sense for US addresses but not Japanese ones.
- What happens if there are more than one of these sets of methods? Which extends which?
- Can you extend multiple classes?
- If you can extend multiple classes, what happens when their methods or fields overlap?

25.2 Interfaces

- Java has a solution, *interfaces*.
- An interface is a type that specifies that the underlying object has particular methods.
- It looks like this:

```
interface Comparable {
    int compareTo(Object o);
}
```

- The documentation for `Comparable` says that it should return a negative number if `this` is less than `o`, zero if they are equivalent, or a positive number if `this` is greater than `o`.
- A class can *implement* an interface like this:

```
class OneInt implements Comparable {
    ...
}
```

- This says that the `OneInt` class contains all the methods declared inside `Comparable`, in this case a `compareTo` method that takes an `Object` and returns an `int`.
- Interfaces are types, so you could say this:

```
Comparable a = new OneInt(5);
```

- In this example, the static type of `a` is `Comparable` and the dynamic type is `OneInt`.
- We can call only the methods of `Comparable` on it, because the static type determines what can be called on an object.
- The code that actually gets run is from `OneInt` (or its superclasses), because the dynamic type determines what actually gets run.
- An interface cannot have fields or method implementations, so it does not make sense to create a `Comparable` with `new`.
- So, the dynamic type of an object can never be an interface type.

25.3 A full example

– Here's the full code to `OneInt`:

```
class OneInt implements Comparable {
    int n;

    OneInt(int n) {
        this.n = n;
    }

    @Override public int compareTo(Object o) {
        OneInt other = (OneInt) o;
        if (n < other.n) {
            return -1;
        } else if (n == other.n) {
            return 0;
        } else {
            return 1;
        }
        // Or we could've just said:
        //     return n - ((OneInt) o).n;
    }

    @Override public String toString() {
        return Integer.toString(n);
    }
}
```

- And here's some code to try it out:

```
import java.util.Arrays;

public class OneIntDemo {

    public static void main(String[] args) {
        Comparable v1 = new OneInt(4);
        Comparable v2 = new OneInt(9);
        Comparable v3 = new OneInt(-13);
        System.out.printf("v1.compareTo(v2) == %d\n",
                          v1.compareTo(v2));
        System.out.printf("v1.compareTo(v3) == %d\n",
                          v1.compareTo(v3));

        Comparable[] a = new Comparable[]{v1, v2, v3};
        System.out.printf("Before sort: %s\n",
                          Arrays.toString(a));
        Arrays.sort(a);
        System.out.printf("After sort: %s\n",
                          Arrays.toString(a));
    }
}
```

  which outputs:

```
v1.compareTo(v2) == -1
v1.compareTo(v3) == 1
Before sort: [4, 9, -13]
After sort: [-13, 4, 9]
```

- Note that the static types are all `Comparable`, but the `compareTo` and `toString` methods in `OneInt` are actually called.
- It's okay to call the methods of `Object` on an object whose static type is an interface since every class extends `Object`.
- When implementing a method from an interface, always specify `@Override` and always make the method `public`.
- If methods from interfaces did not have to be `public`, what would implementing an interface even mean?

25.4 Exercise

Write a class `Point` that implements `Comparable`. It should have two `int` fields, `x` and `y`. One `Point` is smaller than another if it is closer to the origin. That is, the `Point` with the smaller value of `x * x + y * y` is less than the other. Start with this:

```
package inclass;

public class Point implements Comparable {
    int x;
    int y;
    // Now define a comparison function.
}
```

26 Parametric types

2013–04–03

26.1 ArrayList

- Suppose you want an array, but don't know how many elements it should hold.
- For example, suppose you want to return all the primes smaller than `n`.
  - Unless you make a very large array, you won't be able to be sure you can fit all of them.
  - Even if you do, the array is too large when you're done.
- One solution:
  - Start with a small array.
  - If it fills up, create a bigger one, and copy all the elements from the small array.
- A better solution:
  - Use the code of someone who already did that.
  - ArrayList!

26.2 ArrayList example

- Example:

```
import java.util.ArrayList;
import java.util.Arrays;

public class ArrayListDemo {

    public static void main(String[] args) {
        // with an ArrayList:
        ArrayList a = new ArrayList();
        a.add("fee");
        a.add("fi");
        a.add("fo");
        a.add("fum");
        System.out.printf("a = %s\n", a);

        // with a String[]:
        String[] b = new String[] {
            "fee",
            "fi",
            "fo",
            "fum"
        };
        System.out.printf("b = %s\n", Arrays.toString(b));
    }
}
```

- This outputs:

```
a = [fee, fi, fo, fum]
b = [fee, fi, fo, fum]
```

26.3 ArrayList versus Object[]

- Here are some `ArrayList` operations that correspond to array operations:

```
ArrayList              Object[]
-------------------------------------------
a.get(3)               b[3]
a.set(3, "foo")        b[3] = "foo"
a.size()               b.length
```

- Unlike arrays, `ArrayList` starts out size zero and can be grown with `add` and shrunk with `remove`.
- `ArrayList` is just a class without builtin Java syntax like the `[]` operator, so `ArrayList.get(...)` needs a return type.
- That return type is `Object`, so you write code like this:

```
ArrayList a = new ArrayList();
a.add("foo");
String a0 = (String) a.get(0);
```

- If you only ever add, for example, `String` elements to a particular `ArrayList`, you can safely cast the return value of `get` to `String`.

26.4 Parametric ArrayList

- Java 1.5 introduced "parametric types."
- Instead of just an `ArrayList`, we can have an `ArrayList<String>`.
- Just like `ArrayList` is a variable-sized `Object[]`, `ArrayList<String>` is a variable-sized `String[]`.
- `ArrayList` and `ArrayList<Object>` are the same thing.
- The class name in angle-brackets can be any type of object (including any interface type).
- `add`, and `set` take the type of the parameter, and `get` returns it.
- So, this works now without a cast:

```
ArrayList<String> a = new ArrayList<String>();
a.add("foo");
String a0 = a.get(0);
```

- and this will generate a compiler error:

```
ArrayList<Double> a = new ArrayList<Double>();
a.add("foo");
```

  because `"foo"` is not a `Double`, it is a `String`, and the `add` in `ArrayList<Double>` expects a `Double`.

- Even if you plan to use specific parameterized instantiations of `ArrayList`, you only need to import `java.util.ArrayList`.

26.5 Exercise: Adding parentheses to strings

Write a static method, `parenthesize`, that takes an `ArrayList<String>` as an argument and modifies it so that each element is surrounded by parentheses. For example, if its argument is the list [`"foo"`, `"bar"`], then the list should, when `parenthesize` returns, be [`"(foo)"`, `"(bar)"`]. A skeleton program is in `testing/predefined/Parenthesize.java`. You can't change that file, so you should copy it to `inclass` and work from there.

You might find the `String.format()` method useful. It's like `System.out.printf()`, except that instead of printing the formatted `String` out, it returns it (so you could put it into an `ArrayList`, for example).

26.6 Boxed type motivation

- There is no such thing as `ArrayList<int>` or `ArrayList<double>`.
- The runtime type of all kinds of `ArrayList<Something>` is just `ArrayList`.
- The type parameter is only there for compile-time convenience and error checking.
- Everything that goes in or comes out has to be a reference type.
- What if we want a resizable array of some primitive type?
- We could create a type that holds one of them like this:

```
class BoxedInt {
    int x;

    BoxedInt(int x) {
        this.x = x;
    }
}
```

- Then, we could create an `ArrayList` of these:

```
ArrayList<BoxedInt> a = new ArrayList<BoxedInt>;
a.add(new BoxedInt(3));
a.add(new BoxedInt(4));
int a0 = a.get(0).x;  // would be 3.
```

26.7 Using boxed types

- Java provides boxed types for you with the same name as the primitive type, just capitalized and un-abbreviated.
- They are in the `java.lang` package, so they do not need to be imported.
- `Integer` (not `Int`), `Byte`, `Short`, `Long`, `Character` (not `Char`), `Boolean`, `Float`, and `Double`.
- Java will do its best to convert boxed types to primitive types for you:

```
ArrayList<Double> a = new ArrayList<Double>();
a.add(4.0);  // converts double -> Double
double x = a.get(0);  // converts Double -> double
a.set(0, Math.sqrt(a.get(0)));  // converts both ways
```

- Be careful, though, when an operation makes sense on a reference type as well; that interpretation will be preferred.
- These two `Boolean` objects are distinct, even though they have the same conceptual value:

```
new Boolean(true) == new Boolean(true)  // false!
new Boolean(true).equals(new Boolean(true))  // true
```

- In this way, boxed types behave similarly to `String`.

26.8 For more information

- There are lots of other classes that hold collections of other objects.
- CSCI-UA.0102, "Data Structures," will cover them in greater detail.
- Check out Joshua Bloch's Effective Java if you want a deep understanding of Java's parametric types.

27 Iteration

2013-04-08

27.1 For-each loops

- Java has an special syntax for looping over the elements of arrays, `ArrayList` objects, and other collections, the "for-each loop."
- n.b.: Never type `foreach` in Java. This isn't perl.
- Here's what it looks like:

```
import java.util.ArrayList;

public class ForEachDemo {

    public static void main(String[] args) {
        ArrayList<Double> list = new ArrayList<Double>();
        list.add(1.0);
        list.add(2.0);
        list.add(3.0);
        for (Double x : list) {
            System.out.printf("list element: %.3f\n", x);
        }

        double[] array = {4.0, 5.0, 6.0};
        for (double y : array) {
            System.out.printf("array element: %.3f\n", y);
        }
    }
}
```

  This outputs:

```
list element: 1.000
list element: 2.000
list element: 3.000
array element: 4.000
array element: 5.000
array element: 6.000
```

- The general form of such a loop is:

```
for (Type element : container) {
    // Statements that use the element variable.
}
```

- Here, `container` must be some collection of `Type` variables.

- `element` takes on the values of `container` in order.
- If `list` is an `ArrayList<Double>`, these two are roughly equivalent:

```
for (int i = 0; i < list.size(); i++) {
    Double d = list.get(i);
    // commands that use d
}
```

```
for (Double d : list) {
    // commands that use d
}
```

The latter is simpler, so it is preferable in most cases.

27.2 Iterators

- This seems like a simple-enough shorthand, but what is actually going on is more interesting.
- An *iterator* is like a reference to the inside of a sequence.
- It starts out pointing to the first element, then the second, and so on until there are no more elements left.
- In code, it is this:

```
package java.util;

public interface Iterator<T> {
    T next();           // return next element
    boolean hasNext();  // elements left?
    void remove();      // rarely used
}
```

27.3 Iterator example

- Suppose we want to iterate over the factors of a number.
- We could put them in an `ArrayList<Integer>`, then loop over them in a `for` loop.
- We can do better. We shouldn't need to compute them all and store them before we start the loop.
- Here's some code that does that with an `Iterator<Integer>`:

```
import java.util.NoSuchElementException;

public class FactorIterator implements Iterator<Integer> {

    private final int n;
    private int lastFactor;

    public FactorIterator(int n) {
        this.n = n;
        lastFactor = 0;
    }

    @Override public Integer next() {
        if (!hasNext()) {
            throw new NoSuchElementException();
        }
        int possibleFactor = lastFactor + 1;
        while (n % possibleFactor != 0) {
            possibleFactor++;
        }
        lastFactor = possibleFactor;
        return possibleFactor;
    }

    @Override public boolean hasNext() {
        return lastFactor < n;
    }

    @Override public void remove() {
        throw new UnsupportedOperationException();
    }
}
```

```
public class FactorMain {

    public static void main(String[] args) {
        FactorIterator iter = new FactorIterator(12);
        System.out.println("Factors of 12:");
        while (iter.hasNext()) {
            System.out.println(iter.next());
        }
    }
}
```

`FactorMain` outputs this:

```
Factors of 12:
1
2
3
4
6
12
```

27.4 Iterators and for loops

- An `Iterable` is anything that can be iterated over:

```
package java.lang;

public interface Iterable<T> {
    Iterator<T> iterator();
}
```

- That is, it is any object that has an `iterator` method that returns an iterator over its own elements.
- For example, `ArrayList<T>` implements `Iterable<T>`.
- Arrays don't, but let's not talk about that.
- If `foo` is an object of a class that implements `Iterable<Qux>`, then you can iterate over that object in a `for` loop:

```
for (Qux q : foo) {
    doSomeStuff(q);
}
```

- That is almost the same as:

```
Iterator<Qux> iter = foo.iterator();
while (iter.hasNext()) {
    Qux q = iter.next();
    doSomeStuff(q);
}
```

- Compared to a regular `for` loop, this provides a way to iterate over any sequence of objects, even if they aren't numbered.
- Many classes implement `Iterable`: collection classes like sets, sequences of rows in a database, lines in a file, etc.

27.5 Exercise: Range iteration

Create a class, `inclass.RangeIterator`, that implements `Iterator<Integer>`. It should have a constructor that takes two `int` variables, `first` and `last`. It should implement the `next` and `hasNext` methods so that the iterator returns the numbers from `first` to `last`, inclusive. To get you started, I provided the following classes in `testing/predefined`:

- `Range`, a class that implements `Iterable<Integer>` and creates `inclass.RangeIterator` objects when its `iterator` method is called.
- `RangeMain`, a class with a `main` method that will try to use your code in a `for` loop.
- `RangeIterator`, a skeleton of a solution you can copy to `inclass`.

28 Generic example

2013–04–15

On April 15 and 17, we covered the second midterm exam and talked about parametric types ("generics") through worked examples. There are no formal notes. Here is Oracle's generics tutorial. Here is an example from lecture: 2013–04–17

```
// Middle<T> computes the running median of a sequence of
// elements of type T.  The median method returns an element
// for which an equal number of smaller (or equal) and
// larger (or equal) elements have been passed to put.  If
// put has been called on an even number of elements, median
// returns an element larger than (or equal to) half the
// elements, so that it is larger than (or equal to) one
// more element than it is smaller than (or equal to).

import java.util.ArrayList;
import java.util.Collections;
import java.util.NoSuchElementException;

public class Middle<T extends Comparable<T>> {

    private ArrayList<T> seen =
        new ArrayList<T>();

    public void put(T obj) {
        seen.add(obj);
    }

    public T median() {
        if (seen.size() == 0) {
            throw new NoSuchElementException();
        }
        Collections.sort(seen);
        return seen.get(seen.size() / 2);
    }

    // Demonstration
    public static void main(String[] args) {
        Middle<String> middle = new Middle<String>();
        middle.put("foo");
        middle.put("bar");
        middle.put("baz");
        String s = middle.median();
        if (s.equals("baz")) {
            System.out.println("OK!");
        } else {
            System.out.println("Error! (s=" + s + ")");
        }
    }
}
```

29 Exceptions

2013–04–22

29.1 Old school error reporting

- How should a method indicate to its caller that something went wrong?
- All of these patterns are used by high-quality code.
- For example, suppose a method reads a file. What if the named file isn't there?
- It could abort the program.
  - What if the file is optional?
  - What if special error handling or logging is necessary?
- It could return a `boolean` indicating success or failure and return the contents of the file by modifying an argument.
  - This makes for long-winded code since methods can't be called with the return values of other methods.
  - What if the caller forgets to check the return value?
- It could return the desired return type and return an error code by modifying an argument.
  - Similarly long-winded code.
  - Similar problems with checking the return value.

29.2 Exceptions solve this problem

- Method gets to return a regular value.
- If something goes wrong, special error handling code is invoked.
- If there is no error handling code, the caller's caller gets a chance, and so on up the call stack.
- If no method handles the exception, the program aborts.

29.3 What are exceptions?

- Exceptions are alternate return paths for a method.
- Usually a method returns an X, but occasionally returns a Y.
  - "Usually a reciprocal method returns a number. Occasionally it returns a divide-by-zero error."
  - "Usually a file-reading method returns the contents of a file. Occasionally it returns a file was not found error."

  - "Usually a sorting method returns nothing at all and modifies its arguments. Occasionally it returns an an error indicating that the objects to be sorted are not comparable."

- Take a separate path through the code, so that caller can't ignore it.
- Should only be used for unexpected conditions.
  - Bad: reached the end of a file
  - Good: data corruption detected
  - Bad: sort function passed a zero-length array
  - Good: sort function passed a `null` array

29.4 Exceptions in Java

- `throw` is like `return` but returns an exception instead of a regular value.
- `try` and `catch` are used instead of assignment to handle the case where an exception is thrown.
- Just as a return type says what the type of the regular return value is, `throws` is used instead of a return type to indicate what kinds of exceptions a method may throw.
- `finally` lets you run some common code regardless of whether an exception is thrown.

29.5 Java try-catch example

```
try {
    System.out.print("Enter a positive number to ponder: ")
    int n = (new Scanner(System.in)).nextInt();
    // Do something with n here.
} catch (InputMismatchException e) {
    System.out.println("That does not look like a number.");
}
```

```
import java.util.InputMismatchException;
import java.util.NoSuchElementException;
import java.util.Scanner;

public class ExceptionDemo {

    // Reads a number from System.in, closes System.in,
    // and ponders the number.
    public static void ponderUserInput() {
        Scanner scanner = null;
        int n;
        try {
            scanner = new Scanner(System.in);
            while (true) {
                try {
                    System.out.print(
                        "Enter a number to ponder: ");
                    n = scanner.nextInt();
                    break;
                } catch (InputMismatchException e) {
                    System.out.println(
                        "That does not look like a number.");
                    scanner.nextLine();
                } catch (NoSuchElementException e) {
                    throw new RuntimeException(
                        "System.in is not readable.", e);
                }
            }
        } finally {
            scanner.close();
        }
        System.out.printf("Read n = %d.\n", n);
    }

    public static void main(String[] args) {
        ponderUserInput();    // Should work, eventually.
        ponderUserInput();    // Input is closed, will fail.
    }
}
```

29.7 Checked versus unchecked exceptions

- In Java, you can throw any object that is a subclass of `Throwable`.
- `Exception` and `Error` are the two main subclasses.
- `Error` indicates a Java-wide problem, like running out of memory.
  - You rarely want to catch these.
  - Let them propagate up to the to level and crash the program.
- `RuntimeException` is a subclass of `Exception`.
  - You occasionally want to catch these, but they're usually indicative of a bug.
  - Throw these if a caller might want to handle the exception, but probably doesn't.
- `Exception` is the superclass of all checked exceptions.
  - Throw these when it is critical that the caller have error-handling code.
  - Cannot throw a `FooException` with "`throw new FooException(...);`" unless the method has "`throws FooException`" in its definition.

29.8 Example: NumberFormatException

- `NumberFormatException` is thrown by `Integer.parseInt` if you pass it a `String` that is not a valid integer.
- If you generated the argument yourself, it indicates a bug, so you don't want to catch the exception.
- If the argument was user input, it may indicate bad user input, which is expected, so you want to catch it.
- `NumberFormatException` is a `RuntimeException`, so you get the choice.

29.9 Declaring exceptions

- Even though unchecked exceptions do not need to be explicitly declared, it's good form.
- Exceptions thrown are every bit as much of the function's contract as the return value.
- Can't expect a caller to ever handle an unchecked exception if they don't know about it.

- Example:

```
public static int parseInt(String s)
    throws NumberFormatException {
  ...
}
```

29.10 Impossible exceptions

You may be tempted to write something like this to get the Java compiler to shut up about checked exceptions that can't ever happen:

```
try {
    // some stuff that throws an exception
} catch (Exception e) {
}
```

Never do that. It's impossible to debug when the impossible things happen. Don't do this, either:

```
try {
    // some stuff that throws an exception
} catch (Exception e) {
    System.out.println("Error!");
}
```

- You lose all the debugging information.
- The program continues in an invalid state.
- You might not notice the error if you're not running in a terminal.

What about this?

```
try {
    // some stuff that throws an exception
} catch (Exception e) {
    System.out.println("Error:" + e);
    e.printStackTrace();
    System.exit(1);
}
```

- Not as bad, but still not good.
- You can't catch the exception higher up.

- Inside a test framework (like what `test.sh` uses), it will exit the entire test suite, not just the failing test case.

Do this instead:

```
try {
    // some stuff that throws an exception
} catch (IOException e) {
    throw new RuntimeException(e);
}
```

This turns the specific exception that can't happen into an unchecked exception, so if it does happen, your program aborts and you get an error message you can use for debugging.

29.11 Exercise: bonding

What does this do?

```
// Taken from http://xkcd.com/1188
// by Randall Munroe
// Attribution Non-Commercial CC Licensed

class Ball extends Throwable { }

class P {
    P target;

    P(P target) {
        this.target = target;
    }

    void aim(Ball ball) {
        System.out.println(this);
        try {
            throw ball;
        } catch (Ball b) {
            target.aim(b);
        }
    }

    public static void main(String[] args) {
        P parent = new P(null);
        P child = new P(parent);
        parent.target = child;
        parent.aim(new Ball());
    }
}
```

30 Text files

2013–04–24

Reading and writing text files is boring. Let's do this quickly.

30.1 Classes for reading and writing files

- The `java.io.File` class is a reference to a file or directory on disk.
- Creating one with `new` does not create the file, nor does it get deleted when the object goes away.
- `java.util.Scanner` has a constructor that takes a `java.io.File`.
  - Use this to read plain text files.
- `java.io.PrintWriter` has a constructor that takes a `java.io.File`.
  - Use this to write plain text files.
  - `PrintWriter` is very similar to `java.io.PrintStream`
  - `System.out`, which we have been using the whole course, is a `PrintStream`.

30.2 Example

```
// Copies all lines containing SUBSTRING from
// DICTIONARY_FILE to OUTPUT_FILE.

import java.io.File;
import java.io.FileNotFoundException;
import java.io.PrintWriter;
import java.util.Scanner;

public class FileDemo {
    private static final String DICTIONARY_FILE
        = "/usr/share/dict/words";
    private static final String OUTPUT_FILE = "ferrets.txt";
    private static final String SUBSTRING = "ferret";

    public static void main(String[] args) {
        Scanner wordsInput;
        try {
            File wordsFile = new File(DICTIONARY_FILE);
            wordsInput = new Scanner(wordsFile);
        } catch (FileNotFoundException e) {
            System.err.printf("Can't open %s. (%s)",
                              DICTIONARY_FILE, e);
            return;
        }

        PrintWriter output;
        try {
            File outputFile = new File(OUTPUT_FILE);
            output = new PrintWriter(outputFile);
        } catch (FileNotFoundException e) {
            System.err.printf("Can't open %s. (%s)",
                              OUTPUT_FILE, e);
            return;
        }

        while (wordsInput.hasNextLine()) {
            String line = wordsInput.nextLine();
            if (line.contains(SUBSTRING)) {
                output.println(line);
            }
        }
        output.flush();
    }
}
```

31 Recursion and binary search

2013–04–29

31.1 Recursion, obligatory factorial example

- It is a tradition to explain recursion in terms of factorials.
- Who am I to challenge tradition?
- The usual implementation looks like this:

```
static int factorial(int x) {
    int answer = 1;
    for (int k = 1; k <= x; k++) {
        answer *= k;
    }
    return answer;
}
```

- If `x >= 1`, it is the case that `factorial(x) == x * factorial(x - 1)`.
- We can use this to implement it:

```
static int recursiveFactorial(int x) {
    if (x == 0) {  // base case
        return 1;
    } else {       // inductive/recursive case
        return x * recursiveFactorial(x - 1);
    }
}
```

- In the "base case," the result is simple.
- In the "recursive case" or "inductive case," the result is expressed in terms of a simpler result.

31.2 Call stack in recursive functions

This program:

```
public class RecursiveCrash {

    public static int recursiveCrash(int x) {
        if (x == 0) {  // base case
            throw new IllegalStateException();
        } else {       // inductive/recursive case
            return x * recursiveCrash(x - 1);
        }
    }

    public static void main(String[] args) {
        recursiveCrash(5);
    }
}
```

generates the output:

```
Exception in thread "main" java.lang.IllegalStateException
        at RecursiveCrash.recursiveCrash(RecursiveCrash.java:5)
        at RecursiveCrash.recursiveCrash(RecursiveCrash.java:7)
        at RecursiveCrash.recursiveCrash(RecursiveCrash.java:7)
        at RecursiveCrash.recursiveCrash(RecursiveCrash.java:7)
        at RecursiveCrash.recursiveCrash(RecursiveCrash.java:7)
        at RecursiveCrash.recursiveCrash(RecursiveCrash.java:7)
        at RecursiveCrash.main(RecursiveCrash.java:12)
```

You can see how the base case is reached by `recursiveCrash` calling itself repeatedly.

31.3 Iterative searching of unsorted arrays

Iterative searching is the easiest way to find an object in an array, and it is the only way to do it if the array is not meaningfully ordered.

```
import java.util.Arrays;

public class ArrayContains1 {

    public static boolean arrayContains(
            Object[] array, Object elem) {
        for (int i = 0; i < array.length; i++) {
            if (array[i].equals(elem)) {
                return true;
            }
        }
        return false;
    }

    public static void main(String[] args) {
        String[] a = {"bar", "baz", "foo"};
        System.out.printf("%s contains \"foo\": %b\n",
                          Arrays.toString(a),
                          arrayContains(a, "foo"));
        System.out.printf("%s contains \"qux\": %b\n",
                          Arrays.toString(a),
                          arrayContains(a, "qux"));
    }
}
```

outputs:

```
[bar, baz, foo] contains "foo": true
[bar, baz, foo] contains "qux": false
```

31.4 Binary searching sorted arrays

```
import java.util.Arrays;

public class ArrayContains2 {

    public static boolean sortedArrayContains(
            String[] array, String elem) {
        int lower = 0;
        int length = array.length;
        while (length > 0) {
            int pivot = lower + length / 2;
            int c = elem.compareTo(array[pivot]);
            int elementsBeforePivot = pivot - lower;
            if (c < 0) {
                // Shrink to left.
                length = elementsBeforePivot;
            } else if (c > 0) {
                // Shrink to right.
                lower = pivot + 1;
                length = length - elementsBeforePivot - 1;
            } else {
                // Found it.
                return true;
            }
        }
        return false;
    }

    public static void main(String[] args) {
        String[] a = {"bar", "baz", "foo"};
        System.out.printf("%s contains \"foo\": %b\n",
                          Arrays.toString(a),
                          sortedArrayContains(a, "foo"));
        System.out.printf("%s contains \"qux\": %b\n",
                          Arrays.toString(a),
                          sortedArrayContains(a, "qux"));
    }
}
```

- This program uses *binary search* to search an array.
- Two local variables `lower` and `length` maintain a section of the array to search.
- `lower` is the smallest index that could match `elem`.

- `length` is the length of the sub-array starting at `lower` that could match `elem`.
- If `lower` is 3 and `length` is 4, then indices 3, 4, 5, and 6 could possibly match `elem`.
- We start `lower` at 0 and `length` at `array.length` indicating that any element of `array` could match.
- We pick an index, `pivot`, about halfway through, and compare `array[pivot]` to `elem`.
- If `elem` is ordered before `array[pivot]` (so `c` is negative), it must be in the part of the subarray before `pivot`.
- If `elem` is ordered after `array[pivot]` (so `c` is positive), it must be in the part of the subarray after `pivot`.
- Repeat until we find `elem` or `length` is zero, indicating that we have run out of possible matches.

31.5 Recursive search

Here's the same algorithm implemented with recursion:

```
import java.util.Arrays;

public class ArrayContains3 {

    public static boolean recursiveArrayContains(
             String[] array, String elem) {
        if (array.length == 0) {
            return false;
        }
        int pivot = array.length / 2;
        int c = elem.compareTo(array[pivot]);
        if (c == 0) {
            return true;
        }
        String[] subArray;
        if (c < 0) {
            // elem is before pivot, if anywhere
            subArray = Arrays.copyOfRange(
                array, 0, pivot);
        } else {
            // elem is after pivot, if anywhere
            subArray = Arrays.copyOfRange(
                array, pivot + 1, array.length);
        }
        return recursiveArrayContains(subArray, elem);
    }

    public static void main(String[] args) {
        String[] a = {"bar", "baz", "foo"};
        System.out.printf("%s contains \"qux\": %b\n",
                          Arrays.toString(a),
                          recursiveArrayContains(a, "qux"));
        for (String s : a) {
            System.out.printf("%s contains \"%s\": %b\n",
                              Arrays.toString(a), s,
                              recursiveArrayContains(a, s));
        }
    }
}
```

- `recursiveArrayContains` searches a range of an array for a particular element.

- It expresses searching an array in terms of searching an array, and is a bit simpler than the previous version.
- This particular implementation makes unnecessary copies; you wouldn't actually write it with `Arrays.copyOfRange`.

31.6 Exercise: chain length

Suppose you have a class, `Chain2`, that looks like this:

```
package inclass;

public class Chain2 {
    Chain2 link;
}
```

Link can be either another `Chain2` or `null`. Add a method to `Chain`, `length()`, that returns the length of the chain. The length of a chain with `link==null` is 1. The length of a chain with `link!=null` is one plus the length of `link`.

I have provided a skeleton of the code in `testing/predefined/Chain2.java`. Copy it to `inclass` and start from there.

32 Selection sort and mergesort

2013–05–01

32.1 Sorting

- Last time we talked about sequential search and binary search.
- In sequential search, we searched an array by looking at every element.
- In binary search, we assume the array is sorted.
- We take advantage of this to chop it in half successively until we find the part with the element we're looking for.
- But, how do the elements get sorted in the first place?

32.2 Selection sort

- Selection sort is the sorting analog of sequential search.
- Find the largest element, put it in the last position.
- Find the largest element remaining, put it in the second-to-last position.
- Repeat until done.
- Can be implemented recursively or iteratively. Let's do recursive.

32.3 Recursive selection sort

- Given `length` elements, find the largest and put it in position `length - 1`.
- Sort first `length - 1` elements.

```
import java.util.Arrays;

public class SelectionSort {

    public static void selectionSort(String[] array) {
        selectionSortRecurse(array, array.length);
    }

    static void selectionSortRecurse(
            String[] array, int length) {
        if (length < 2) {
            return;
        }
        // Find largest element.
        int indexOfLargest = 0;
        for (int k = 1; k < length; k++) {
            if (array[k].compareTo(array[indexOfLargest]) > 0) {
                indexOfLargest = k;
            }
        }
        // Swap largest and last.
        String tmp = array[length - 1];
        array[length - 1] = array[indexOfLargest];
        array[indexOfLargest] = tmp;
        // Sort first length - 1 elements.
        selectionSortRecurse(array, length - 1);
    }

    // Print out sorted command-line arguments.
    public static void main(String[] args) {
        selectionSort(args);
        System.out.println(Arrays.toString(args));
    }
}
```

32.4 Efficiency of selection sort

- There are approximately `array.length` calls to `selectionSortRecurse`.
- Each call calls `compare` a bunch of times, `array.length - 1` the first time through, only one when sorting the first two elements, and about `array.length / 2` when it's halfway done.
- On average, it has to look at half the elements each call to

`selectionSortRecurse`.

- So, there are about `array.length * array.length / 2` total calls to `compare`.
- We call this an $O(n^2)$ algorithm since to within a constant factor, it takes n^2 operations to handle a data set of size n.
- Similarly, sequential search is $O(n)$ and binary search is $O(\log n)$.

32.5 Interlude: sorting a deck of cards

- Bubble sort:
  - Flip through deck.
  - Every time you see a pair of cards out of order, switch them.
  - Repeat until all cards are in order.
- Selection sort:
  - Find the last card, put it aside.
  - Find the next-to-last card, put it on top of the last.
  - Repeat until all cards are in the sorted pile.
- Mergesort:
  - Split the cards in half.
  - Split the first pile in half.
  - Split the first half-pile in half again, repeating until the piles only have one card.
  - Combine the small piles by picking up the smallest card in a pile until all the cards have been combined.
  - Move on to the next pile.
  - Repeat until the two piles comprising the entire deck are merged.

32.6 Mergesort

- We can do better than selection sort or bubble sort!
- Let's divide up the problem like we did with binary search.
- Sort the first half of the array, sort the second half, and merge the results together.
- You can implement mergesort iteratively, but it is hard to get right.
- n.b. "Sequential search," "binary search," and "selection sort" are all two word names. "Mergesort" is a single word.

```
import java.util.Arrays;

public class Mergesort {

    // Assume a and b are sorted.  Merge them into m.
    private static void merge(
            String[] m, String[] a, String[] b) {
        if (m.length != a.length + b.length) {
            throw new IllegalArgumentException();
        }
        int nextA = 0;
        int nextB = 0;
        for (int nextM = 0; nextM < m.length; nextM++) {
            if (nextB == b.length
                || (nextA != a.length
                    && a[nextA].compareTo(b[nextB]) <= 0)) {
                m[nextM] = a[nextA];
                nextA++;
            } else {
                m[nextM] = b[nextB];
                nextB++;
            }
        }
    }

    public static void mergesort(String[] array) {
        if (array.length < 2) {
            return;
        }
        int pivot = array.length / 2;
        String[] a = Arrays.copyOf(array, pivot);
        String[] b = Arrays.copyOfRange(
            array, pivot, array.length);
        mergesort(a);
        mergesort(b);
        merge(array, a, b);
    }

    // Print out sorted command-line arguments.
    public static void main(String[] args) {
        mergesort(args);
        System.out.println(Arrays.toString(args));
    }
}
```

32.7 Efficiency of mergesort

- Mergesort splits the array in half until the arrays are only two long, so just like binary search, it calls itself to a depth of log base two of the number of elements in the array.
- Unlike binary search, it can't throw away half the array every time it calls itself.
- At every depth, each element gets copied once and merged once, so each layer of recursion requires about array.length operations.
- Multiplying the length times the log of the length gives us the total number of operations to within a constant factor.
- Or, concisely, mergesort is $O(n \log n)$.
- This is much better than selection sort, which is $O(n^2)$.
- As much better than selection sort as binary search is better than sequential search.
- If all you can do is compare elements, you can't sort in better than $O(n \log n)$ time.
- You can avoid taking up extra space with temporary arrays with *quicksort*, which will be covered in CSCI-UA.0102.
- Quicksort is really neat.

32.8 Exercise

There are a certain number of rabbits, $R(d)$, alive on day d. It takes two days after a rabbit reach maturity, and rabbits have one baby every day once they are mature. So, the number of rabbits on day d would normally be the number of rabbits on day $d-1$ (all of the living rabbits) plus the number on day $d-2$ (the number of babies of mature rabbits). However, a bear that eats one baby rabbit every day. On days 1 and 2, there are two rabbits. Write a Java function that takes one `int` argument, the day, and returns the number of rabbits alive on that day.

```
package inclass;

public class Rabbits {

    static int rabbitsOnDay(int day) {
        // Your code here.
        return 0;
    }

    public static void main(String[] args) {
        for (int day = 1; day < 10; day++) {
            int r = rabbitsOnDay(day);
            System.out.printf("%d rabbits on day %d\n",
                              r, day);
        }
    }
}
```

33 Networking

2013–05–06

33.1 How do computers talk to each other?

- A *network* is a collection of computers that can talk to each other over wires, radio, or other means.
- The internet is collection of connected networks run by different organizations.
- Networking protocols specify the conventions used by different devices on a network that wish to communicate with each other.

33.2 Protocol layers

- Networking protocols are specified at multiple levels, each wrapping the higher-level protocol.
- Physical: the voltages, connector types, etc. corresponding to the physical devices (Ethernet, Wi-Fi)
- Link: the conventions used for two directly-connected devices that wish to talk to each other (Ethernet, Wi-Fi)
- Internet: the conventions used for two indirectly-connected devices that wish to talk to each other (IP, IPv6)
- Transport: the conventions used for two applications that wish to talk to each other over a network reliably and without conflicting with other applications (TCP, UDP)
- Application: the conventions that apply to particular type of application communication (DNS, HTTP for downloading web pages, SMTP for sending email)

33.3 Internet addresses

- We're mostly going to talk about the internet layer and higher layers.
- Every computer that can talk to other computers over IP has a four byte address, usually written in the form "18.72.0.3," where each part is a number in the range 0 to 255.
- There are only four billion such numbers, and we ran out, so we're (very slowly) moving to IPv6, which has 128-bit addresses, usually written in the form "2001:0db8:85a3:0042:0000:8a2e:0370:7334."
- You sometimes also see 48-bit hex addresses like "08:00:07:A9:B2:FC." Those are "MAC addresses" or "Ethernet addresses" used at the link layer, not the internet layer.

- Link-layer addresses are fixed to a particular computer, but internet addresses change when you move to a new network.
- Otherwise, packets from computers not directly connected to you would not be able to find you.
- A protocol called DNS maps a name like "www.nyu.edu" to an internet address.
- You can do a DNS lookup yourself with the `host` command on Mac OS:

```
$ host www.nyu.edu
www.nyu.edu is an alias for WEB.nyu.edu.
WEB.nyu.edu has address 128.122.119.202
WEB.nyu.edu has IPv6 address 2607:f600:8002:1::7
WEB.nyu.edu mail is handled by 20 MX.nyu.edu.
```

or `Resolve-DnsName` on Windows:

```
> Resolve-DnsName www.nyu.edu

Name                Type   TTL   Section    NameHost
----                ----   ---   -------    --------
www.nyu.edu         CNAME  399   Answer     WEB.nyu.edu

Name       : WEB.nyu.edu
QueryType  : AAAA
TTL        : 399
Section    : Answer
IP6Address : 2607:f600:8002:1::7


Name       : WEB.nyu.edu
QueryType  : A
TTL        : 399
Section    : Answer
IP4Address : 128.122.119.202
```

33.4 Transport-layer (TCP/UDP) addresses

- At the internet level, computers send packets of data to other computers.
- At the transport level, program send continuous streams of data to other programs.
- At the transport level, addresses are the combination of an internet address and a *port*.

- A port is a 16-bit unsigned integer specifying which program on a computer is sending and receiving the messages.
- Servers listen on well-known ports.
  - The web server on a computer traditionally listens on port 80.
  - You sometimes see ports in URLs, e.g. `www.example.com:8080` means "look up `www.example.com` in DNS, then connect to the machine with that internet address on port 80."

33.5 Application layer

- Physical, link, internet, and transport layers are standardized.
- IPv4, in particular, has been in need of replacement for twenty years and hasn't been fixed because the standard is so widely implemented.
- Application layers, on the other hand, can be created and modified at will.
- This is a strong point of the internet: the "end-to-end" property.
- The internet takes care of getting data from point A to point B, with all of the interesting logic on the ends.
- It doesn't matter whether the intermediate link is a satellite or a fiber optic cable, who owns it, or how many individual packets are lost due to sunspots; it all looks the same to the program.
- In URLs, you can identify the application layer protocol from the prefix like `http://`.

33.6 Talking to a server by hand

A simple application protocol, "daytime," does not have any client data at all. A client connects, the server sends it the time, and the connection ends. If a computer is running a daytime server, it listens on port 13. (You can see all the well-known ports in `/etc/services`). You can talk to a server interactively and do the application-layer protocol yourself with the `nc` (netcat) command on Mac OS. (Versions for Windows are also available, but I can't recommend one.)

```
$ nc time.mit.edu 13
Sat Nov 10 21:51:33 2012
```

For a more complicated example, you can speak HTTP, the protocol for downloading web pages, yourself:

```
(you)    $ nc www.nyu.edu 80
(you)    GET / HTTP/1.1
(you)    Host: www.nyu.edu
(you)
(server) HTTP/1.1 200 OK
(server) Date: Sun, 11 Nov 2012 02:27:00 GMT
(server) Content-Length: 28903
(server) Content-Type: text/html
(server) ...
(server)     <title>New York University</title>
(server) ...
```

33.7 Networking in Java

2013–05–08

- The `InetAddress` class represents an internet-layer address.
- The `SocketAddress` class represents a transport-layer address.
- The `Socket` class represents a transport-layer connection.
- The `ServerSocket` class allows a server to listen for connections on a port. When a client connects to that port, `ServerSocket` gives back a `Socket` over which the client and server can communicate.
- Demonstration in `TimeClient.java` and `TimeServer.java`.

34 Concurrency

34.1 Walking and chewing gum at the same time

- *Concurrency* refers to doing more than one thing at once.
- In a computer with more than one processor, more than one program, function, or method may be executing at a given time.
- Even if there is only one processor, different bits of code can appear to run simultaneously by taking turns (or by the operating system forcing them to take turns).
- Even if two different chunks of code can't take turns and there is only one processor, input/output devices like disks and network adapters can be doing things while the main processor runs a program.
- More going on at once means more gets done.
- Especially improves responsiveness as perceived by users.

34.2 Threading

- One way to have concurrency is to run multiple programs at once.
- Another way is to have a single program do multiple things at once.
- In a concurrent program, a *thread* is one of the things the program is doing.
- Each thread gets its own local variables and its own call stack.
- But, objects can be referred to by multiple threads if multiple threads can get a reference to the same object.
- A *task*, as used in this class, refers to something a thread might do.
- Threads can run only one task at once, but they might run several tasks, one after another.

34.3 Threading in Java

- A thread in java is represented by the `Thread` class.
- A task is represented by the `Runnable` interface.
- The simplest way to write a concurrent program in java is to create `Runnable` objects for each thing you want the program to do, then create `Thread` objects for each of them, then start the threads.
- The threads will run concurrently.

- If any of them has to wait for IO, network, or user input, the rest will continue independently.
- If you have four processors, as long as at least four threads are not waiting for something external, all four will be in use at all times.
- See ConcurrentTimeServer.java.

34.4 Race conditions

- A *race condition* occurs when program behavior depends on timing in a way that makes its behavior unpredictable.
- See BadCountingTimeServer.java.
- Tasks in `BadCountingTimeServer` use `taskCount`, a variable that can change while the task runs.
- Not even `taskCount++` is safe.
- To increment a variable, you need to read it, increment it, and write it back out to memory.
- Two tasks can read the variable at the same time, increment it, and write it back out.
- Then, the variable will only have been incremented by one.

34.5 Locking

- A *lock* is used to coordinate multiple threads.
- A lock can be held by at most one thread at once.
- Each shared object is *guarded* by a lock.
- To read or write the shared object, you *acquire* the lock.
- When you are done with the shared object, you *release* the lock.
- If a thread tries to acquire a lock another thread has already acquired, it waits ("blocks") until the lock is released.
- If two objects only make sense when modified together, they should be guarded by the same lock.
  - Example: bank account balances: acquire lock, increase one, decrease other, release lock.

34.6 Locking in Java

- Any object can be used as a lock in Java.
- To acquire a lock and run some code with that lock held, put the code in a `synchronized` block.

```
// ...code run without lock...
...
// Next line blocks until lock on lockObject is acquired.
synchronized (lockObject) {
  // ...code to run with lock...
}  // Lock released here.
...
// ...code run without lock...
```

- See CountingTimeServer.java.

34.7 Issues

- Concurrency is hard.
- Locking does not solve all problems.
- Locking limits concurrency when multiple threads all need access to the same object at the same time.
- Worse, a program may *deadlock* if two threads need locks held by each other to proceed.
- Concurrency bugs are often non-deterministic and hard to identify and fix.

34.8 Optional readings

- Oracle Java Concurrency Tutorial: The minimum you need to know to write concurrent Java code.
- Java Concurrency in Practice, by Brian Goetz et al: More than you need to know, but quite good, and a pleasure to read even if you don't want to write concurrent Java code.