CSC152 2000S

Fundamentals of Computer Science II

Introductory Handouts

Monday, 24 January 2000

Samuel A. Rebelsky

Grinnell College

Introductory Handouts

These materials are also available online. The online versions may not precisely match the printed ones, since I update my Web pages regularly. The online versions are the definitive ones.

Catalog Description

An introduction to many of the fundamental concepts in computer science. Builds upon the programming knowledge from Computer Science 151 to study the design, analysis, and verification of algorithms. Includes a discussion of data abstractions and data structures. Also provides an overview of the field of computer science. Includes formal laboratory work.

Prerequisites: Computer Science 151.

Front Door

Welcome to the Spring 2000 session of Grinnell College’s CSC 152, *Fundamentals of Computer Science II*, which is described relatively well in the official blurb. My own take on this course is that we’ll be expanding your knowledge of Computer Science and of computer programming, while emphasizing the development and analysis of common data structures and algorithms. We will be using Java as our development language.

In an attempt to provide up-to-date information, and to spare a few trees, I am making this as much of a ‘‘paperless’’ course as I can. You may also want to read the basic instructions for using this course web.

Warning! Experience shows that CSC152 is a significantly more time-consuming and accelerated course than CSC151. Expect to spend about twice as much time on CSC152 as you spent on CSC151, and expect to go about twice as fast through the material.

Basics

Meets: MTuWF 11-11:50 a.m.

Instructor: Samuel A. Rebelsky, Science 2427. Office hours: Tu 2:15-4:15, W 3:15-4:15 (also feel free to stop by when my door is open)

Grading: Labs and attendance: 10%; Homework: 30% (3-5 graded assignments out of 6-10 total assignments); Project: 25%; Exams: 35% (3 graded take-home exams).

The final examination for this course is optional. It can be used as a makeup for one examination. Like the other examinations, it will be a take-home examination.

Labs: While you won’t do as many labs as you did in CS151 (if you took CS151), Labs are for your benefit, not mine, so I won’t be grading most of them (other than to check that you completed them).

Extra Credit: I will occasionally give you quizzes to ensure that you're keeping up with the reading. Correct answers on the quizzes will give you some amount of extra credit. I may also give some extra credit for corrections to *Java Plus Data Structures*.

Throughout the term, I may suggest other forms of extra credit.

Books and Other Readings

Rebelsky, Samuel (2000). *Experiments in Java*. Reading, MA: Addison Wesley Longman. This is the laboratory manual that we will use for the first few weeks of class. It is temporarily available online, but you are expected to purchase copies. (I receive no royalties.)

Dale, Nell; Rebelsky, Samuel; and Weems, Chip (2001). *Java Plus Data Structures*. This is also in draft form. I will distribute a chapter each week (more or less). You can find additional materials online.

Rebelsky, Samuel (2000). The CS152 2000S Course Web. The hypertext that you are currently reading. All of these materials are optional, but you may find them useful.

Class 01: Introduction to the Course

Held Monday, January 24, 2000

Overview

Today, we begin CS152 by considering the core subject matter of the course: computer science, data structures, algorithms, and object-oriented design.

Notes

- *Assignments:*
  - Read the introductory handouts.
  - Read chapter 1 of *Java Plus Data Structures* for tommorow. *Feel free to skip over the Java code.*
  - Introductory survey (due tomorrow). *Due before class Friday!*
  - Read lab J1 (due Wednesday).
- Make sure to complete the Introductory survey for Tuesday's class! (A few students always seem to miss this.)
- On Thursday, January 27, at noon, I'll be giving a presentation on summer opportunities in computing and computer science. Some opportunities are available for non-majors, and there's a chance that I'll take a first-year student for my research team, so feel free to come.
- Last-minute update (after the packet was printed): My co-authors and I have decided to rearrange *Java Plus Data Structures*. There will no longer be a Chapter 3 on Object-Oriented Programming. The various list chapters may also be combined into one.

Summary

- Course overview
- Definition of *computer science*
- Introduction to data structures and algorithms
- Introduction to programming paradigms
- *Handouts:*
  - Course Overview (taken from the Course Web
  - CSC152 at a Glance
  - Chapter 1 and Chapter 2 of *Java Plus Data Structures*.

About This Class

- CS152 is primarily a course in *Data Structures* and *Algorithms*. At some institutions, it has that name.
  - A *data structure* is a formalism for organizing and managing data. Often, the way you organize the information in your program permits or inhibits particular operations. Different structures may also lead to different costs (in time or space).
  - An *algorithm* expresses the steps involved in completing a task.
- CS152 is also a course in *procedural* and *object-oriented* programming.
  - Presumably, you've seen a little bit of each in CS151 (or whatever course you've taken previously). We will certainly talk more about both paradigms.
  - We will be doing our programming in **Java**. In the past, I've said that the language we use is less important than the concepts we learn. However, I must emphasize that you will not learn the concepts unless you also learn Java.
- Like most computer science courses, CS152 will have both theoretical and practical components. I hope you will enjoy relating the two.

Grounding Ourselves

- Before we delve too far into these issues, we should ground ourselves somewhat by asking ourselves a few questions (and I'll be asking these of the class).
  - What is Computer Science?
  - What is Computer Programming?
  - How are they similar? How are they different?
  - What is an algorithm?
  - What is a computer program?
- We also want to ask ourselves some practical questions.
  - What programming languages do we know?
  - What CS or programming concepts are we least comfortable with?
  - How comfortable are we with the workstations and Unix?
- Finally, I'd like you to reflect on the course (and you'll be doing this again on the introductory survey).
  - Why are you taking this course?
  - What do you expect to get out of this class?

Administrative Issues

- Please refer to the course web site and the introductory handout for details.
- Teaching philosophy: I support your learning
- Policies
  - Attendance: I expect you to attend every class. Let me know when you'll miss class and why.
  - Grading: I'm a hard grader. I don't grade everything.
  - Course web
  - Etc.

- The exams
  - Three take-home exams during the semester. Plan to spend ten hours on each one.
  - An *optional* final to make up for a bad exam grade. (Last semester, I didn't give a final and just dropped the lowest exam. This semester, I'm much more likely to give the final.)
- The lab manual
  - It may be available online (locally only); I'm still nego
  - Using the online labs
  - Yes, you must buy a copy.
  - I will rarely collect labs. When I do, you can just summarize your answers on a separate sheet of paper.
  - No, I don't receive royalties.
- The book
  - Under development.
  - I'll hand out chapters as I write them or determine that you need them (seven of the eleven chapters should be ready by the start of the semester; they cover the first seven weeks of the semester (although not in a one-to-one mapping).
  - I'd appreciate comments. If you feel that you've made a substantial contribution to the book through your comments, let me know and I'll put you in the acknowledgements section.
- Projects
  - The tradition is to do a large project
  - I'll list a few possibilities: Email client, Game, Image processing package, Four years at a glance, PseudoVax

An Introduction to Data Structures

- This semester, we'll talk about *data structures* and *abstract data types*. Many computer scientists treat them as equivalent terms.
- An *abstract data type* (ADT) is a collection of values and operations on those values. ADTs specify the *what* of data.
- A *data structure* is a structure designed to organize data.
- When we distinguish the two, we sometimes say that data structures implement abstract data types.
- Those of you coming from the Scheme-based 151 may already be familiar with two basic ADTs: the list and the vector.
- Both lists and vectors gather data into a *sequence*.
- More importantly, they provide facilities for manipulating the sequence.
  - You can extract an element from the sequence.
  - You can change an element in the sequence.
  - (Sometimes) you can insert or remove an element from the sequence.
- Vectors and lists differ in the operations they provide and the costs associated with each operation.
- In designing and building your own ADTs, you will be concerned with
  - The *specification* -- the description of the data structure and the operations it provides.
  - The *implementation* -- how you actually provide those operations (and how you store the data to provide those operations).
  - The *algorithms* used in the implementation.

    - The *efficiency* of your implementation -- how much it costs to provide those operations.
    - The *applications* of the data structure.
- Note that we want a clear barrier between specification and implementation, so that a client of one of your data structures need only know what you do and not how you do it. We often call this separation *encapsulation* or *information hiding*.
- This term, we will be looking at each of these aspects of a number of the key data structures in CS.

Programming Paradigms

- Computer Scientists have developed a number of strategies for looking at algorithm and data design, including
    - procedural / imperative
    - object-oriented
    - functional
    - logical
    - declarative
- While individual definitions of each category may differ, most definitions have some similarities.
- In CS151, you studied functional and imperative programming.
- In CS152, you will study object-oriented and imperative programming.

How to Use the Course Web

For a number of reasons, I have chosen to make many of the handouts for this course available only in electronic format on the World-Wide Web. I will not go over basic use of the Web, since you should know about it from other courses. You should make sure to ask me if you have any questions about using the World-Wide Web.

The course web can be found at
`http://www.math.grin.edu/~rebelsky/Courses/CS152/2000S/` You may want to bookmark that page.

A number of important pieces of information are in the course web, including assignments, readings, requirements, syllabus, and office hours. I assume that if I put information on the Web, you will (eventually) read it.

- At the bare minimum, you should read all the pieces of basic information about the course. Of particular interest is the syllabus, which lists all the readings. I will also hand out a copy of this information on the first day of class.
- I prepare a rough outline for each class. Most students find these useful, and you should feel free to refer to them before, during, and after class. Since this class is becoming more lab-based, it is likely that in laboratory sessions there will be more information in the outlines than I will cover in class.
- Each outline begins with some notes. You can find just the notes in a separate news page.

At the top and bottom of every page are a series of links to important components of the course web. These are

- Instructions. This set of instructions.
- Search. A simple search facility for the course web.
- Current. The outline of the current or next class. You may need to reload the page to get the appropriate version.
- News. The course news, taken from the outlines.
- Syllabus. The course syllabus.
- Glance. An abbreviated version of the syllabus.
- Links. A collection of links that you might find useful.
- Handouts. Handouts for the class.
- Project. Information on the course project, including code files once they become available.
- Outlines. The outlines of classes that have been held. You can sometimes access other outlines through the syllabus.
- Labs. Laboratory assignments not in *Experiments in Java*.
- Assignments. A list of the assignments for the class, accompanied by their due dates
- Quizzes. The quizzes and surveys for this course. I don’t always grade these.
- Examples. A list of examples generated for this class.
- EIJ *Experiments in Java*, the laboratory manual used for the first few weeks of class.
- JPDS. The textbook for the class (or a current draft thereof).
- Tutorial. A local online copy of *The Java Tutorial*. This is a useful introduction to the Java programming language. You can only access this through the MathLAN.

- API. Documentation for the standard Java libraries.

Administrative Information

On Teaching, Learning, and Grading

Introduction

I like to begin each course with a metacommentary on teaching and learning. Why? Because I care about the learning process, because I seem to have a different teaching style and personality than some students expect, because I want you to think not just about *what* you are learning, but also *how* you are learning, and, unfortunately, because in one of the first two courses I taught at Grinnell some students were clearly dissatisfied with the way I teach. (As people are getting used to me and my teaching style, this seems to be less of a problem.)

From my perspective, you are here to learn and I am here to support that learning. What will you be learning? The subject matter of the course, certainly. However, I expect that (or hope that) you will also be discovering new ways to think and learn or sharpening existing skills. In terms of subject matter, I tend to care more about the processes and concepts that you learn than about the ‘‘basic facts’’.

Learning is an interactive process. You learn by asking, discussing, and answering questions, by playing with ideas (in computer science, you also learn by playing with programs), and by working with others. I know from experience that computer science cannot be learned passively: you need to experiment with ideas (in your head, on paper, or on the computer) in order to fully grasp these ideas.

My Role

How do I try to support this learning? In a number of ways.

I *assign readings* to give you a basis for understanding the subject matter. Sometimes these readings will be from the textbook, sometimes I will distribute appropriate supplements.

I *lecture, lead discussions*, and *conduct recitations* on the topics of the course. Sometimes these will be based on readings and assignments, sometimes they will vary significantly from your readings. Why? Because I feel it wastes your time and mine to simply reiterate the readings. If you let me know that you’re confused about a reading, I will spend time going over that reading (either in person or in class).

To stimulate discussion and thinking, I regularly *call on students* in class. I know that not all of you are comfortable answering questions publically, but I strongly believe that you need to try. Please feel free to say ‘‘I’m not sure’’ when I call on you. At times, I’ll step through the class, asking each student in turn. At others, I’ll call on you individually. I tend to call more on students I interact with regularly.

I *assign work* because I find that most people learn by grounding concepts in particular exercises that allow them to better explore the details and implications of those concepts. I expect you to turn in work on the day it is due and will impose severe penalties on late assignments (including refusing to accept some late assignments).

Some of my assignments may involve *public presentation* of your work. Sometimes, the best way to learn a topic is to have to discuss it or present it to someone else. In addition, I've found that many students need some work on their presentation skills. Most often, presentations will be of papers that you've read.

In general, I expect you to spend about ten hours per week on this class outside of class time. If you find that you are spending more than that, let me know and I'll try to reduce the workload.

I *grade assignments* to help you identify some areas for improvement. Note that I believe that you learn more from doing an assignment than from receiving a grade on that assignment. This means that you may not receive a grade or comments on all your assignments. I will tell you when an assignment won't be graded, but not until after you hand it in. I will do my best to be prompt about returning grades on assignments. At times, I will use a grader to help speed the process.

I *give examinations* because I find that many students only attempt to master a concept when preparing for an exam. Because I care more about processes and concepts than about facts, I almost always give open-book examinations. Because I do not feel that time limits are helpful, this semester I will give you only take-home exams.

I *give quizzes* to ensure that you are doing the reading and that you are understanding what I expect you to understand from the readings and assignments. At times, I will give quizzes to help illustrate a particular point. This semester, all of my quizzes can only affect you positively: good work on quizzes will lead to extra credit.

I *build course webs* to organize my thoughts, to give you a resource for learning, and to help those of you who need to work on your note-taking skills. I do my best to make my notes for each lecture available on the Web, in outline format. In general, these notes will be available approximately five minutes before class. Warning: these are rough notes of what I expect to talk about; the actual class may not follow the notes. I will also attempt to update the notes after each class.

I *make myself available* to discuss problems and questions because I know that some of you will need personal attention. In general, if I'm in my office you should feel free to stop in. Most of the time, I'll be willing to help. Once in a while, I'll be working on a project and will ask you to come back later. Students always have first priority during office hours. You should also feel free to send me electronic mail, which I read regularly, and to call me. This semester, I am on partial parental leave, which means that I will be less available than normal. In particular, I will not be in the office on Tuesdays and Thursdays. Feel free to give me a call at home on those days, but understand that I may be busy.

At times, I *survey* my students to better understand how the class is going. Because I do research on the effects of computers on learning, I sometimes give surveys to gather data.

Grading

At the same time that you learn and I try to help you learn, Grinnell and the larger community expect me to assign a grade to your work in the class. I base grades on a number of components, but primarily on *assignments, examinations,* and *involvement in classroom discussions*.

Because I understand that not everyone gets everything right the first time, I will occasionally allow you to *substitute* an extra assignment for one that you did poorly on. Unfortunately, the time pressures of the semester are significant enough that I will not be able to permit you to make up assignments except through this mechanism.

In computer science, it is often possible to do the same problem in multiple ways. Hence, I typically reserve class days to discuss particularly significant assignments. This semester, each exam will be followed by a day of discussion relating to the exam. We may also take time from some classes to discuss particulars of assignments.

I will admit to a fairly strict grading scale. Grinnell notes that A and A- represent exceptional work. To me, ‘‘exceptional’’ means going beyond ‘‘solid’’, correct work. Exceptional work entails doing more than is assigned or doing what is assigned particularly elegantly. Work limited to mastery of the core materials is B-level work. To help you demonstrate exceptional understanding, I will occasionally suggest *extra credit work* (although truly exceptional students will often suggest such work on their own).

To help eliminate biases, I typically use a numerical grading scale. 94-100 is an A, 90-93 is an A-, 87-89 is a B+, 84-86 is a B, and so on and so forth.

Your Role

How should you participate as a member of my class? (Or, how do you do well in my class?) By being an active participant in your own learning. In part, this means doing all the work for the class. It also means a number of other things.

Come talk to me when you have questions or comments about subject matter, workload, or how the course is going in general. I will also set up an anonymous comment page for those who are uncomfortable talking to me directly.

Do the readings in advance of each class period and come prepared with a list of things that you don’t understand. I will try to spend time at the beginning of each class session answering these questions or will restructure the lecture to accommodate them.

Ask and answer questions and make comments during class periods. I consider active participation during class a particularly important part of the learning process.

Begin your assignments early. Students who begin assignments early have more opportunities to ask for help, to make sure that the assignment gets completed, and to sleep at night. Such students also do better in general.

Lecturing

I seem to have a different ‘‘lecturing’’ style than some students expect. As I mentioned earlier, I don’t think it is the purpose of lecture to reiterate the readings. I do, however, think lecture and readings can provide alternate perspectives on the subject matter. At times, I will also discuss issues not covered in any readings.

I see no point in going on with a lecture or example if many students don’t understand what’s going on. You are the first line of defense: stop me when you are confused. In addition, I will occasionally stop the class and ask for a show of hands to see who is confused. Don’t be embarrassed to raise your hand; if you are confused, it is likely that someone else is also confused. I realize that this show of hands leads to some ‘‘pressure for understanding’’. However, you won’t get much out of a class if you’re confused (and therefore just copying down what I’m writing without thinking about it).

I deem it important for students to be active participants in lecture. This means that I will often ask you to help develop algorithms, solve problems, and even critique each other’s answers. If I call on you and you’re not sure of an answers, feel free to say ‘‘I don’t know’’ or to venture a guess. I consider it very important for all of us to see the problem solving process, warts and all. Note that I often generate examples of discussion ‘‘on the fly’’ so that we can all be involved in the problem solving or development process.

Favoritism

For various reasons, I often get to know some students better than others. I tend to call on the students I know better, and sometimes respond slightly better to their questions because I have more context for the questions. I’m happy to make all of you ‘‘favorite’’ students. If you come to see me regularly and work enthusiastically on the material, you’ll probably end up being a student I know well.

Summary

As the prior discussion suggests, I expect a great deal from my students. I also use many different strategies to get the best out of you. Feel free to discuss any of this with me (anything from concerns about this perspective to suggestions on improving teaching and learning).

Academic Honesty

I expect you to follow the highest principles of academic honesty. Among other things, this means that any work you turn in should be your own or should have the work of others clearly documented. When you explicitly work as part of a group or team, you need not identify the work of each individual.

You should never ‘‘give away’’ answers to homework assignments or examinations. You may, however, work together in developing answers to most homework assignments. Except as specified on individual assignments, each student should develop his or her own final version of the assignment. On written assignments, each student should write up an individual version of the assignment and cite the discussion. On non-group programming assignments, each student should do his or her own programming, although students may help each other with design and debugging.

When working on examinations, you should not use other students as resources.

If you have a question as to whether a particular action may violate academic standards, please discuss it with me (preferably before you undertake that action).

Citing Program Code

Note that computer programming shares with normal writing a need to cite work taken from elsewhere. It is certainly acceptable practice to borrow other code for your assignments. However, you must cite any code that you use from elsewhere. Each piece of code you take from elsewhere must include a comment that specifies:

- the author of the original code;
- the date the original code was written and the version of the code (if available);
- the date you incorporated the code into your program;
- a summary of the modifications (if any) you made to the code;
- instructions for getting the original code.

This applies not only to the code you get from the Web and elsewhere; it also applies to code you get from me and from the textbook.

You do not need to cite the classes and libraries you use, as the command to include classes and libraries within a program provides sufficient citation.

Disabilities

I encourage those of you with disabilities, particularly ‘‘hidden disabilities’’ such as learning disabilities, to come see me about the accommodations that I can make to make your learning easier. If you have not already done so, you should also discuss your disability with academic advising. If you think you may have an undocumented learning disability, please speak to me and to academic advising.

In my experience, some learning difficulties can make computer science more difficult because of computers’ emphasis on small details. I also know that many of my favorite and best students have some learning disability, and have certainly succeeded. We’ll all do better if you talk to me about disabliities early.

Note that I generally feel that the ‘‘accommodations’’ that we are asked to make for those with learning disabilities are often appropriate for all students. Hence, I rarely give timed exams and I typically allow students to use computers during exams.

Project

Final Project

As part of your work in CSC152, you will be working on a large final project. You will work in small teams of 3-4 students, and each team will contribute a different part to the project. The subprojects will, of course, depend on the intended final project.

We first tried this large project in Spring 1999, and students reported that it was quite successful in that it: (1) demonstrated practical uses for many of the issues we covered in CSC152; (2) provided an appropriate mechanism for discussing and learning about issues in software design; (3) strengthened students skills at working in groups; and (4) provided a useful ‘‘line on the resume’’. In Fall 1999, students came to much the same conclusions, although we did not get as far int he project as we would have hoped.

Potential Projects

In Spring 1999, we built a distributed online auction system which will be used for art auctions at Science Fiction Conventions. In Fall 1999, we built most of the parts for an email client. This semester, we will choose a new project. (And yes, you can have input into the choice.)

Here are some of the suggestions I’ve come up with. Feel free to suggest your own.

Emailer. We could continue to work on the email package from last semester.

Image Manipulation. We can put together a package that allows one to manipulate digital images. This may include blurring, rotation, and other things you come up with. It might also be appropriate to provide a form of animation by applying multiple transformations in sequence. We may even choose to make this scriptable (so that people can write simple programs to transform an image).

A Nonviolent Game. I’ll admit that I don’t have any great plans here, but I thought I’d throw it out as a potential idea. We may also want to throw some form of networking into the mix (and may even need to do so). I think some form of Calvinball would be interesting.

Pseudo-Vax. An application that provides many of the features that students miss from the vax, including chat and plan files.

Four Years at a Glance. An application to help students (and their advisors) plan their careers at Grinnell. Given the new requirement that students fill out a four-year plan when they declare their major, this application could be particularly helpful.

A Web Search Engine.

Approximate Schedule

- Discuss possible projects: Week three
- Settle on a project topic: Start of week four
- Discuss architecture and break into teams: Week four

- Specifications due: Week six
- Final specifications due: Week seven
- Individual implementations due: Week ten
- Work on integration: Week eleven
- Public presentation: Week twelve
- Final projects due: Week thirteen

Course Syllabus

*This is a highly **approximate** syllabus. Expect topics, assignments, ordering, and almost everything else to change.*

- Week One: Introduction
- Week Two: Object Basics
- Week Three: Java Fundamentals
- Week Four: Building Graphical Programs
- Week Five: Algorithms and Recursion
- Week Six: Algorithms, Classic and Otherwise
- Week Seven: Data Structures
- Week Eight: Lists
- Break
- Week Nine: Linear Structures
- Week Ten: Dictionaries
- Week Eleven: Miscellaneous
- Week Twelve: Trees
- Week Thirteen: Graphs
- Week Fourteen: Wrapup
- Final

Week One: Introduction

Class 01 (Monday, January 24, 2000) **Introduction to the Course**

- Course overview
- Definition of *computer science*
- Introduction to data structures and algorithms
- Introduction to programming paradigms
- *Handouts:*
    - Course Overview (taken from the Course Web
    - CSC152 at a Glance
    - Chapter 1 and Chapter 2 of *Java Plus Data Structures*.

Class 02 (Tuesday, January 25, 2000) **Introduction to Java**

- Introduction to object-oriented programming
- An introduction to Java
- Java vs. Scheme
- *Due:*
    - Introductory survey
    - Reading: Chapter 1 of *Java Plus Data Structures*.
    - Reading: Introductory Handouts

Class 03 (Wednesday, January 26, 2000) **Lab: Getting Started with Java**

- Using Java in the MathLAN, revisited
- Laboratory J1: An Introduction to Java
- *Handouts:*
  - *Java Plus Data Structures*, Chapter 3 (I hope)
- *Due:*
  - Preparatory reading of Lab J1
  - Preparatory reading of Using Java in the MathLan

Class 04 (Friday, January 28, 2000) **Lab: Objects and Methods**

- The structure of a class, revisited
- Writing methods
- Lab J2: Objects and Methods
- *Due:*
  - Preparatory reading of Lab J2

Week Two: Object Basics

Class 05 (Monday, January 31, 2000) **Lab: Objects and Classes**

- Encapsulating object state with fields
- Constructors
- Overloading
- Lab J3: Building Your Own Classes
- *Distributed:*
  - *Java Plus Data Structures*, Chapters 4, 5, 6, and 11
- *Due:*
  - Preparatory reading of Lab J3
  - Homework 1: Completed Lab J2

Class 06 (Tuesday, February 1, 2000) **Reuse Through Inheritance and Polymorphism**

- Goals of reuse
- Inheritance
- Overloading
- Polymorphism
- *Due:*
  - Preparatory reading of Labs O2 and O3

Class 07 (Wednesday, February 2, 2000) **Lab: Polymorphism**

- Lab O3: Interfaces and Polymorphism
- *Distributed:*
  - Homework 2: The Design of a Student Info Class (Due Monday, February 7, 1999)

- *Due:*
    - Preparatory re-reading of Lab J2: Polymorphism

Class 08 (Friday, February 4, 2000) **Object-Oriented Design**

- (Re)Introduction to object-oriented design
    - Motivation
    - Key attributes
- Project discussion
- Sample object-oriented designs
- The six P's
- *Due:*
    - Reading of Lab O1: Object-Oriented Design (we will not do this lab)

Week Three: Java Fundamentals

Class 09 (Monday, February 7, 2000) **Lab: Primitive Types**

- The need for primitive types
- Java's primitive types
- Objects that correspond to the primitive types
- Lab X1: Primitive Types
- *Due:*
    - Preparatory reading of Lab X1
    - Homework 2: The Design of a Student Info Class

Class 10 (Tuesday, February 8, 2000) **Lab: Conditionals**

- Boolean values and expressions
- The `if` statement
- The `switch` statement
- Lab J4: Boolean Expressions and Conditionals
- *Due:*
    - Preparatory reading of Lab J4: Boolean Expressions and Conditionals

Class 11 (Wednesday, February 9, 2000) **Lab: Loops**

- The need for repetition
- Primary looping control structures:
    - `for` loops
    - `while` loops
- Lab J5: Control Structures for Repetition
- *Due:*
    - Preparatory reading of Lab J5

Class 12 (Friday, February 11, 2000) **When Things Go Wrong**

- Program design: How to handle and check errors
  - Test before execution
  - Test results
  - Catch exceptions
- Avoiding errors: Preconditions and postconditions
- *Handouts:*
  - Exam 1 (due Friday, February 18, 2000)
- *Due:*
  - Preparatory reading of Lab X3
  - Homework 3: A Simple Calculator

Week Four: Building Graphical Programs

Class 13 (Monday, February 14, 2000) **Lab: Java's Abstract Windowing Toolkit**

- GUI basics
- Java's Abstract Windowing Toolkit (AWT)
- Primary components: Frames, TextFields, etc.
- Event handling
- Extra discussion: Selecting a project
- Lab G2: Java's Abstract Windowing Toolkit
- *Due:*
  - Preparatory reading of Lab G2

Class 14 (Tuesday, February 15, 2000) **Lab: Java's AWT, Revisited**

- Organizing the frame
  - Layout managers
  - Panels
- Event Handling, Revisited
- Lab G3: Java's Abstract Windowing Toolkit, Continued
- *Due:*
  - Preparatory reading of Lab G3

Class 15 (Wednesday, February 16, 2000) **Image Processing**

- Modeling an image
- Colors in Java
- Java's `Image` class
- An improved image class

Class 16 (Friday, February 18, 2000) **Project Discussion**

- Discussion:
  - Project components and architecture
  - Working in a team
  - Selection of components
- *Due:*
  - Exam 1
  - Nominations for project teams

Week Five: Algorithms and Recursion

Class 17 (Monday, February 21, 2000) **Discussion of Exam 1**

- Further project discussion
- Summary of results of exam 1
- Particular problem areas
- *Distributed:*
  - Answer key for exam 1

Class 18 (Tuesday, February 22, 2000) **Algorithm Analysis**

- Two simple ‘‘find the smallest value’’ algorithms
- Basics of algorithm analysis
- Other searching algorithms
- *Due:*
  - *Java Plus Data Structures*, Section 5.5

Class 19 (Wednesday, February 23, 2000) **Lab: Recursion**

- Recursion
  - Purpose: looping
- The three question method:
  - Recursive case
  - Base case
  - Shrinking input
- Recursion vs. iteration
- Lab A1: Recursion
- *Due:*
  - Preparatory reading of Lab A1
  - *Java Plus Data Structures*, Chapter 6: Recursion
  - Homework 4: A Graphical Student Database

Class 20 (Friday, February 25, 2000) **Analyzing Recursive Algorithms**

- Examples
  - Three versions of exponentation
  - Two versions of Fibonacci
- Recursive functions and recurrence relations
- *Due:*
  - Project, Phase 1: Initial specifications for all your classes (written as interfaces)

Week Six: Algorithms, Classic and Otherwise

Class 21 (Monday, February 28, 2000) **Project Discussion: Specifications**

- Project Narrative

Class 22 (Tuesday, February 29, 2000) **Binary Search**

- Binary search
  - Iterativev vs. recursive
  - In Scheme
  - In Java
  - Examples
- Comparing objects
- *Due:*
  - *Java Plus Data Structures*, Sections 5.1 and 5.2

Class 23 (Wednesday, March 1, 2000) **Sorting Algorithms**

- The problem of sorting
- In-place vs. out-of-place sorting
- Stable sorts
- Three basic sorting methods:
  - Selection sort
  - Bubble sort
  - Insertion sort
- Choosing a sorting method
- *Due:*
  - *Java Plus Data Structures*, Chapter 11: Sorting
  - Homework 5: Algorithm Analysis

Class 24 (Friday, March 3, 2000) **More Efficient Sorting Algorithms**

- Divide and conquer sorts
  - Merge sort
  - Quick sort
- Other sorting techniques

- *Due:*
    - Project, Phase 2: Revised specifications for project classes

Week Seven: Data Structures

Class 25 (Monday, March 6, 2000) **Project Discussion: Revised Specifications**

- Project narrative, revisited
- Other comments on specifications

Class 26 (Tuesday, March 7, 2000) **Introduction to Data Structures**

- Introduction to data structures
    - Four key issues: functionality, implementation, efficiency, applications
- Compound data types
    - Collections
    - Iterated Collections
    - Keyed Tables

Class 27 (Wednesday, March 8, 2000) **Arrays**

- Arrays as a data structure
    - General model
    - Java's model
- Why use arrays?
- *Handouts:*
    - Exam 2 (due Friday, March 17, 2000)
- *Due:*
    - *Java Plus Data Structures*, Section 2.3 and Case Study: an Array of Strings
    - Homework 6: Quicksort

Class 28 (Friday, March 10, 2000) **Lab: Documentation and Testing**

- *I'll be at the SIGCSE conference, speaking and learning about CS education. Mr. Stone will teach today's class.*
- Why document?
- Audiences for documentation
- Javadoc: Java's documentation toolkit
- Short Javadoc lab

Week Eight: Lists

Class 29 (Monday, March 13, 2000) **Introduction to Lists**

- Introduction to collections
- Introduction to lists
- Iteration
- The design of lists
    - A data-oriented perspective
    - A functionality-oriented perspectivve
    - Other design issues
- *Due:*
    - Project, Phase 3: Compilable interfaces and stubs for all your project classes.
    - *Java Plus Data Structures*, Chapter 4: Implementing Lists with Arrays

Class 30 (Tuesday, March 14, 2000) **Implementing Lists**

- Primary implementation techniques:
    - Arrays
    - Linked lists
- *Due:*
    - *Java Plus Data Structures*, Chapter 7: Linked Lists

Class 31 (Wednesday, March 15, 2000) **Sorted Lists**

- Abstraction, revisited: What does it mean for a list to be sorted?
- Implementing sorted lists
- *Due:*
    - *Java Plus Data Structures*, Chapter 5: Sorted Lists

Class 32 (Friday, March 17, 2000) **Lab: Animation**

- It's the day before break, so we'll just have fun working with animations in Java
- *Handouts:*
    - *Java Plus Data Structues*, Chapter 8, Stacks and Queues
- *Due:*
    - Exam 2

Break

Break runs from 5:00 p.m. on Friday, March 17, 1998 to 8:00 a.m. on Monday, April 3.

Week Nine: Linear Structures

Class 33 (Monday, April 3, 2000) **Discussion of Exam 2**

- Summary of exam results
- Discussion of
- *Handouts:*
    - Answer key to exam 2

    - *Java Plus Data Structures*, Chapter 9: Dictionaries

Class 34 (Tuesday, April 4, 2000) **Introduction to Stacks and Queues**

- Linear Structures
    - Meaning
    - Core operations
- Three types of linear structures
    - Stacks
    - Queues
    - Priority queues
- *Due:*
    - *Java Plus Data Structures*, Chapter 8, Stacks and Queues

Class 35 (Wednesday, April 5, 2000) **Lab: Implementing Queues**

- An array-based implementation of queues
- Special Lab: Implementing Queues
- *Due:*
    - Homework 7: Critique of a List Design
    - Preparatory reading of special stack lab

Class 36 (Friday, April 7, 2000) **Lab: Stacks and Computation**

- Ambiguity in expressions
- Reverse Polish Notation
- Using stacks for computation
- Special Lab: Reverse Polish Notation

Week Ten: Dictionaries

Class 37 (Monday, April 10, 2000) **Dictionaries**

- Dictionaries, Defined
- Simple Implementation: Association Lists
- *Due:*
    - *Java Plus Data Structures*, Chapter 9, Dictionaries
    - Project, Phase 4: Draft implementations of classes

Class 38 (Tuesday, April 11, 2000) **Binary Search Trees**

- Modeling a data structure for binary search
- A ‘‘divide and conquer’’ data structure for searching
- Binary search trees
- Some tree terminology
- The problem of balancing trees

Class 39 (Wednesday, April 12, 2000) **Hash Tables**

- The elusive goal: O(1) lookup and retrieval
- Using objects as numeric indices
- Hash tables
- Java’s `java.util.Hashtable` class

Class 40 (Friday, April 14, 2000) **Implementing Hash Tables**

- Designing hash functions
- Handling conflicts in hashing
- Supporting deletion in hash tables

Week Eleven: Miscellaneous

Class 41 (Monday, April 17, 2000) **Priority Queues, Heaps, and Heap Sort**

- A ‘‘divide and conquer’’ implementation of priority queues
- Heaps: balanced implementations of priority queues
- Heapsort: Sorting using heaps
- *Due:*
    - Project, Phase 5: Working implementations of individual parts

Class 42 (Tuesday, April 18, 2000) **Project Discussion: Integration**

- Narratives, revisited
- Time to work with other groups
- Proposed order of presentation for Monday

Class 43 (Wednesday, April 19, 2000) **Pause for breath**

- *A little extra time to cover topics that need more time, or to spend more time on the project.*

Class 44 (Friday, April 21, 2000) **Introduction to Trees**

- Heaps and search trees, revisited
- Representing arithmetic expressions
    - Evaluating expressions
- Generalizing: a binary tree ADT
- Generalizing further: a tree ADT
- *Distributed:*
    - *Java Plus Data Structures*, Chapter 10, Trees

Week Twelve: Trees

Class 45 (Monday, April 24, 2000) **Project Discussion: Preparation for Presentation**

- We will give a public presentation of the project at lunch time. Class time will be devoted to preparation for that presentation. I'll pay for pizza and pop.
- *Due:*
    - Project, Phase 6: Descriptive handouts

Class 46 (Tuesday, April 25, 2000) **Automated Problem Solving with Linear Structures**

- Modeling puzzles
    - Modeling the solution space as a tree
    - An algorithm for solving puzzles
    - Using stacks
    - Using queues

Class 47 (Wednesday, April 26, 2000) **Traversing Trees**

- Generalizing the puzzle solution: How d you iterate, visit, or list the elements of a tree?
- Preorder vs. postorder
- Depth-first vs. breadth-first
- Special Lab: Travering trees
- *Due:*
    - *Java Plus Data Structures*, Chapter 10, Trees

Class 48 (Friday, April 28, 2000) **Implementing Trees**

- Implementation details
- *Handouts:*
    - Exam 3 (due Friday, May 5, 2000)
- *Due:*
    - Project, Phase 7: Final implementations

Week Thirteen: Graphs

Class 49 (Monday, May 1, 2000) **Introduction to Graphs**

- Modeling and pictures
- Sample modeling problems
- Introduction to graphs
    - A structure for modeling some common problems
    - Terminology
    - Methods
    - Common implementations

Class 50 (Tuesday, May 2, 2000) **Simple Graph Algorithms**

- Common graph problems
- The traveling salescritter problem
- Reachabliity: Can you get there from here?

Class 51 (Wednesday, May 3, 2000) **The Shortest Path Problem**

- Shortest path: How fast can you get there from here?
- A brute-force shortest-path algorithm
- Dijkstra’s algorithm

Class 52 (Friday, May 5, 2000) **Graphs, Concluded**

- Minimum spanning tree
- Design issues in graph problems
- *Handouts:*
  - Final examination
- *Due:*
  - Exam 3

Week Fourteen: Wrapup

Attendance is particularly important this week.

Class 53 (Monday, May 8, 2000) **Discussion of Exam 3**

- Summary of results of exam 3
- Special problems
- Comments on projects

Class 54 (Tuesday, May 9, 2000) **Course Summary and Evaluation**

- Course topics, Revisited
  - Object-Oriented Programming and Program Design
  - Algorithms: Analysis, Common, Design
  - Data Structures: Design issues, Common
  - Java Programming
- Time for official course evaluation

Class 55 (Wednesday, May 10, 2000) **What is Computer Science? Revisited**

- An overview of CS
  - Three threads: Mathematics, Science, Engineering
  - Many subtreads
- Topics/courses available at Grinnell
- *Due:*

    - Homework 8: Project Questions

Class 56 (Friday, May 12, 2000) **An Abbreviated History of Computer Science**

- Some definitions: Computing, Electronic Computing, Binary vs. Analog, Networking, Computer Science
- A timeline
- The impact of computing
- Other implications
- *Due:*
    - Course development questionaire

Final

You may turn in the *take-home* final any time during finals week.

Miscellaneous

Using Java on the MathLAN Network

There are two basic steps to running a Java program in any environment: you must *compile* the program and then *interpret* the compiled program. Compilation verifies the syntax of your program and translates your program into a language that the computer can more easily understand. The interpreter then executes this more easily understandable language.

To compile a program named `XXX.java` in the MathLAN, type

```
% /home/rebelsky/bin/jc XXX.java
```

The ‘‘`jc`’’ stands for ‘‘Java compiler’’. If there are no observable errors in your program, you will see another prompt after about one minute (yes, the MathLAN is slwo). If there are observable errors in your program, the compiler will print a list of line/error pairs. Most people find the error messages unreadable, so feel free to ask me for help understanding them. Usually, if you look closely at the line (or near the line), you’ll find the error.

When compilation succeeds, the compiler creates a file called `XXX.class`. You can confirm this by typing

```
% ls
```

To execute the compiled program, type

```
% /home/rebelsky/bin/ji XXX
```

The ‘‘`ji`’’ stands for ‘‘Java interpreter’’.

These commands are set up to work with reasonable variants. Hence, you can also leave off the `.java` when compiling or add it when running the program.

When you get sick of typing `/home/rebelsky/bin`, add the following line to your `.cshrc` file. This file is located in your home directory. Ask Sam or a tutor if you need help adding the line.

```
set path = (/home/rebelsky/bin $path)
```

Afterwards, close your terminal window and open another one. From then on, you can use just `ji` and `jc`.

Copying Classes

If you are importing classes such as `SimpleInput` and `SimpleOutput`, it is likely that you will need to make copies of those classes in the current directory. (Later in the semester we may talk about how to create your own library of standard classes.)

Warning!

Note that you are running special scripts designed for my class. If you would prefer to execute the actual Java compiler and interpreter, you'll need to make a few changes to your `.cshrc` file (this is only for more advanced students). The standard java compiler is called `javac`. The standard java interpreter is called `java`. Both can be found in `/usr/local/java/bin`

In order to use these, you will probably want to do the following.

- Add `/usr/local/java/bin` to your path. The standard command (which goes in your .cshrc) is `set path = (/usr/local/java/bin ${path})`

Unix Issues

A few helpful Unix hints:

- To redo the last command, use two exclamation points. For example, if you get an error when compiling a Java file, you can just type `!!` to do it again. (This special command is often pronounced ‘‘bang bang’’.)
- To redo the last of a particular kind of command, use an exclamation point and the beginning of that command. For example, to repeat the last `man` command, you would use `!man`.

Java Style Guide

This document is still under development. Any suggestions would be appreciated.

Programming, like writing, provides you some freedom to express the same concept in many different ways. Just as each writer develops his or her own personal style, so do programmers. Important aspects of programming style include

- the choise and capitalization of names (variables, function names, class names);
- the use of *whitespace* (carriage returns, indents) in coding;
- the ordering of components that may be presented in different orders; and
- the type and number of *comments* you use.

I expect you to follow some basic stylistic conventions in my class. While you will eventually develop your own style, few of you are experienced enough programs to make sufficiently informed decisions. In addition, a uniform style helps us discuss and share code. I will strive to use the same style, but may violate it ocassionally. If you use emacs as your editor, it will automatically enforce some of these rules.

Names

A typical program requires a significant amount of naming of variables, fields, functions, and classes. Strive for informative and distinguishable names. You should rarely use single-letter names (except, possibly, for loop counters). At the same time, you need not go overboard on the length of your names. For most purposes, five to ten characters suffice.

Sun (the developers of Java) recommends a particular capitalization style to help you distinguish the different things that you can name.

- *Class names* should begin with an uppercase letter and have mixed case. For example, `TimeZone` and *InputReader*.
- *Function names* (method names) should begin with a lowercase letter and have mixed case when appropriate. For example, `readInteger` of `getDefault`.
- *Variable names* (field names) should not include uppercase letters. They may include underscores to separate words.
- *Constants* (or variables that act like constants) should be in all capital letters. They may also use underscores and digits. For example, TESTING.

White Space

Java is whitespace-insensitive (at least in most cases). You can put as few or as many spaces, tabs, and carriage returns as you want in your programs. Hence, you should use whitespace to make your programs more readable, modifiable, and debugable.

- Include a blank line before each new function and between indpendent sections of a function. This helps the reader quickly skip between parts.
- For nested code (most often surrounded by curly braces), indent the nested code by two or three spaces (I use two; many people use three; emacs seems to use 3).

- Don’t put anything on the same line after an opening curly brace. Doing so makes it difficult to insert new code at the beginning of that block.
- Don’t put anything on the same line before a closing curly brace. Doing so makes it difficult to insert new code before the end of the block.
- Some people prefer to put opening curly brace on a line by themselves. You’ll see this style in *Java Plus Data Structures*. Much of the time, I prefer to put the opening curly brace after the thing that is being opened.
- Align your curly braces (this should be automatic if you indent according to my rules above, but this is a reminder). This makes it easier to understand the control flow of your program.
- When Java forces you to include curly braces around one command, you can put them on one line, as in

  ```
  try { line = input.readLine(); }
  catch (Exception e) { line = ""; }
  ```

Long Lines

Long lines of code are often hard to read, particularly when you print them out on a printer that only prints 80 columns. (Sometimes the extra text is lost; sometimes it’s wrapped to the next line.) Hence, you will often need to reformat or break long lines. Try to indent the continued line in such a way that makes the relationships clear. For example, you should indent arguments so that they match the open parenthesis for a method.

Comments

If people are to read your class (and yes, people will sometimes need to read your classes), you must provide *comments* to help them get through the morass of Java.

An *introductory javadoc class comment* describes the purpose of your class and may give examples of usage.

An *introductory programmer’s comment* follows the Javadoc comment, and gives details of the implementation for those who need it (primarily those who will want to extend or maintain your code).

Each field and method must be preceded by a *short javadoc comment* that describes their use. I recommend taht your comment include *the six P’s* (summarized below).

Within a method, you should provide *short, natural-language comments* that describe what you intend to do with your code in each ‘‘step’’ (a step need not be a line of code; some steps involve multiple lines).

If you use particularly tricky or confusing code (and yes, you probably know which pieces are tricky or confusing), please include an explanation of what’s going on.

The Six P’s

When you design and document methods, you need to think about a variety of things. Last semester, students found it useful to use what we ended up calling ‘‘the six Ps’’.

- The *purpose* of the method.
- The *parameters* of the method.
- The *preconditions* of the method. That is, requirements that must be met in order for the method to succeed.
- The *postconditions* of the method. That is, what you can guarantee will hold after the method finishes successfully.
- Potential *problems* that may arise. In Java, you may throw exceptions when problems arise.
- The type and value of results the method *produces*. (The return value and the return type. It took a little effort to fit this round peg into the square hole of ‘‘begins with p’’.)

Ordering

It is helpful to the reader if you order the code within your classes in a uniform way. For example, you might separate your program into

- Constants
- Fields
- Constructors
- Public methods, for external use
  - Methods that modify the object
  - Methods that just extract information
- Helper methods, for internal use

Most readers also find it useful if you provide clear divisions between the different sections.

This is not the only way to segment your classes. For example, if your class contains a large number of methods, you may want to group them according to functionality.

While some people will read your code in electronic form, others will need to read a printed copy. It helps if you alphabetize the methods or provide a "table of contents" at the beginning.

Examples

```
// Step through the list of people, printing out all the students.
for (int i = 0; i < REPETITIONS; ++i)
  {
    // If person i is a student then ...
    if (people[i].getRole().equals("student"))
      {
        // Print out the name of person i.
        System.out.println(people[i].getName() + " is a student");
      } // if
  } // for
```

Template

Here is a template you might use as you begin to write your classes. It is not the only possible structure, but it is a beginning.

```
/**
 * A description of the class.
 *
 * @author YOUR NAME HERE
 * @version XXX of DATE
 */
public class XXX {
/*
An introductory comment for the people who have to read your code,
giving an overall structure to the class.
*/

   // +--------+--------------------------------------------------
   // | Fields |
   // +--------+

   // +--------------+--------------------------------------------
   // | Constructors |
   // +--------------+

   // +--------------+--------------------------------------------
   // | Transformers |
   // +--------------+

   // +-----------+-----------------------------------------------
   // | Observers |
   // +-----------+

   // +----------------+------------------------------------------
   // | Helper Methods |
   // +----------------+
} // class XXX
```

Unix File Permissions

As many of you have noticed, Unix has a fairly complex schema for affecting the permissions of files. You can change permissions using the file manager or the `chmod` command.

- Defaults
- Categories of Users
- Types of Permissions
  - File Permissions
  - Directory Permissions
- Some Tasks

Defaults

In the MathLAN, the default is for your files to be unreadable and unmodifiable by anyone except yourself. It is also the default that no one can see what files are in your directories are use any files in your directories.

From my perspective, these defaults are too cumbersome. At the very least, you should make your home directory accessible, if not readable, with

```
% chmod +x /home/yourname
```

Categories of Users

At times, you will need to give others access to your files. For example, you may want me to look at your files so that I can make a copy. At present, Unix only has three categories of people you can affect: yourself, you and other general users, and any user of our system. These are conveniently referred to as **u**ser, **g**roup, and **o**ther.

Types of Permissions

Both files and directories have three basic kinds of permissions: read, write, and execute.

File Permissions

When you give someone **read** permission to a file, it means that they can look at the contents of the file *as long as they have appropriate access to the enclosing directories*. When you give some **write** permission to a file, it means that they can modify the file *as long as they have appropriate access to the enclosing directories*. Don’t worry about **execute** permission for files.

Directory Permissions

When you give some someone **read** permission to a directory, it means that they can list the contents of that directory *as long as they have appropriate access to the enclosing directories*. When you give someone **write** permission to a directory, it means that they can create and delete files in that directory *as long as they have appropriate access to the enclosing directories*.

What is appropriate access? It is **execute** access. Without execute access, there's not much anyone can do in or below a directory. I'd recommend that you make it a point to give execute access to any directories that others might use.

Some Tasks

I'd like other people to read a specific file, `zebra`, in `/home/student/stuff/animals`. They should not be able to tell what other files I have in that directory.

- All the directories you own should be executable by all.

```
% chmod go+x /home/student
% chmod go+x /home/student/stuff
% chmod go+x /home/student/stuff/animals
```

- That file should be readable by all

```
% chmod go+r /home/student/stuff/animals/zebra
```

- You want to make sure that `animals` is not readable.

```
% chmod go-r /home/student/stuff/animals
```

I'd like to create a ''drop box'' where people can put files but can't see them once they're there.

- All the enclosing directories should be executable.
- That directory should be executable and writiable.

```
% chmod go+wx /home/student/DropBox
```

- That directory should not be readable.

```
% chmod go-r /home/student/DropBox
```

- Unfortunately, not only is it possible for people to put files in that directory but also remove files (as long as they can guess the names of those files).
  - A particularly malicious person could write a program that simply removed every file name from your directory.

I'd like to create a file, `comments`, that anyone can modify. No one else should be able to add, remove, or even see files in its directory which I've called `restricted`.

- All the enclosing directories should be executable.
- That directory should be only executable

```
% chmod go-rw /home/student/restricted
% chmod go+x /home/student/restricted
```

- The file should be readable and writable.

```
% chmod go+rw /home/student/restricted/comments
```