COMPSCI186-SEC01 Using Data Structures Spring 2022

Lecture notes 01-01

Overview

Welcome

Hello and welcome!

Welcome to our time of learning together.

Welcome to the many first-years in this classroom. Welcome to returning students.

Welcome to people of all ages, all colors, all cultures, abilities, sexual orientations and gender identities.

Welcome to those who identify as biracial, multi-racial, or multi-ethnic.

Welcome to people from Massachusetts, from other states, and from countries all around the world.

Welcome to people of all political persuasions – or who abstain from politics. Welcome to people of all religions and of no religion.

Welcome to military veterans.

Welcome to people who live with mental illness.

Welcome to those of you who are financially broke, or those broken in spirit.

It is my firm belief that *you all belong here*, and I want you to feel welcome. Whoever you, wherever you are on your journey in computer science, you are welcome here.

I'm Marc Liberatore liberato@cs.umass.edu and I'm your instructor for this course, COMPSCI 186. Please call me Marc.

Administrivia

The most important thing to know today: This courses uses Moodle! It links to the syllabus for this class and you are expected to read the syllabus in its entirety; it also includes all assignments, readings, and so on.

The course is not just me (of course). We have two fine TAs: Liza Bialik and Mehmet Savasci.

(show pictures)

TAs are graduate students in the department who also have the job of helping run courses. They hold office hours, run discussions and labs, do grading, answer questions online, and so on.

We also have several undergraduate course assistants, Ananya, Yan, Vedanth, who will also be answering questions, holding office hours, and grading some of your work. They don't start until next week, and their office hours will be online.

Public health

Face masks are required. Be sure to cover your nose and mouth with a well-fitting mask, and to wash your mask occasionally.

No eating or drinking in the classroom.

Get tested whenever UMass tells you to.

Complete your daily COVID-19 checklist *before* coming to class. We record lecture, so please do not come to class if you have any doubts about symptoms!

Wash and sanitize your hands frequently.

Other important announcements

There is written assignment due *before lecture* on Thursday!

And, there is a (short, and I hope simple) programming assignment due Friday!

Finally, there will be a start-of-semester quiz in lab next Monday!

(It appears that I love exclamation points!!!)

And, that's a typical week in 186 – Monday there is either a quiz or a lab meeting during your scheduled time. Tuesday and Thursday, lectures, with short written assignments due at the start of lecture.. Friday, problem programming assignment due. Get in the habit of thinking about 186 all the time!

Seriously, 186 is a lot of work for many people. Especially if you haven't done lots of programming before, this is a course where at least some of the training wheels come off. Set aside time – about eight or nine hours throughout the week, though not all at once!– to work on things, *especially* programming assignments.

What is this course?

What is this course? It's a course about the who, what, when, and why of commonly-used data structures – we'll learn their names and behavior well enough to know when to use which structures. We'll only briefly touch on the how – how data structures are implemented – as COMPSCI 187 concerns itself deeply with this topic and is the next course in the COMPSCI sequence.

Speaking of 187, let's confront the elephant in the room right now. 186 is an semi-optional course between 121 and 187. Why does it exist, and why does it exist now? (And by implication, why are you here?) Two big reasons and one small one:

- The "casualty rate" between 121 and 187 has in the past been unacceptably high. Students who get less than an A- in 121 or less than a 4 on the CS AP exam were not likely to succeed in 187 – by which I mean, they are likely to get less than a C on their first attempt at 187. The only option they then had was to re-take 187, which nobody (us or you) likes.
- Much of the content (the "how") in 187 is, perhaps, overkill for students intending to pursue the Informatics degree.
- Some first-year students come in with prior programming experience – enough that 121 is not a great use of their time, but not enough to prepare them for 187. In the past they just had to take 121, which again, not a great use of time.

186 is an attempt to kill these two-and-a-half birds with one stone.

First, we conjecture that many students who do OK in 121 – a B- is nothing to be ashamed of! – or who get a 3 on the CS AP exam, could pass 187 – if only they had a little more practice programming and exposure to various parts of the computer science ecosystem (especially the practical bits of Java). 186 is intended to guide you on a path toward programming mastery that's more gentle than the current trajectory of 187. 187 has a bit of what I call the "eat your vegetables first" problem. It also has a bit of the "pie eating contest, where first prize is more pie" problem. Together this is a recipe for a lot of spinach pies, which maybe isn't great if you've not been training in competitive spinach pie eating since you were a kid. 186 is an attempt to provide more reasonable portion sizes and to balance the diet.

Moving on from mixed metaphors, 187 is a prerequisite for all of the 200-level COMPSCI classes. Some upper-level COMPSCI classes are reasonable fits for Informatics majors (such as 326: Web Programming), but in the past had prerequisites of either 187 or 220/230. The reason for these prerequisites is, in some cases, programming maturity: we think you need more than just 121 to be ready for them. 186 is, we hope, an alternative to 187 in giving you the experience you need to be ready for these courses (this may change someday, but right now is true – INFO students can take some CS300s without other prep than 186).

So, what's our plan? We'll start off with a review of some of the material from 121 or AP CS – mostly Java and how it works. We'll spend a little time on Integrated Development Environments (IDEs); this semester, we'll be using Visual Studio Code. We'll begin with toy programs, and as the semester progresses, we'll work our way up to more complicated programs. We'll learn about and use many of the data structures available in the Java API to build these programs. Along the way, we'll learn about some of the tooling you can expect to use as a working Java programmer (in 187, or in Informatics courses, or in internships). This is the part where I'm also supposed to say, "and we'll have fun doing it," but let's not over-egg the pudding.

Java review

Bits and bytes

We'll spend most of the next few weeks on a review of some 121 material (and some IDE and testing how-tos). Let's get started.

The fundamental unit of information is the bit – a single, binary digit, of value 0 or 1.

Bits are organized into bytes: 8 bits in a byte. bits (the smaller unit) are abbreviated `b`; bytes (the bigger unit) are abbreviated `B`.

When we talk about computer memory and RAM, we're talking about a large number of bytes that we use to store data. *Data* is thing that enables nontrivial, "stateful" programs — without (changeable) state, behavior is predetermined. But computers are generally useful to us because we can vary their behavior: we need data, and we need to be able to manipulate it.

How many bytes are we talking about? How much memory does your PC or Mac have? 4 GB? What's a giga-?

A kilo / mega / giga are metric prefixes: multipliers of 10^3 (1,000), 10^6, and 10^9, respectively. Confusingly, in the land of computer science, they're multipliers of 2^10 (1,024), 2^20, and 2^30 – each slightly larger than the corresponding metric prefix. Even more confusingly, that only applies to memory; network bandwidth and disk sizes usually use the metric meaning. (This fact is one of the reasons why when you buy a 500GB drive it usually shows up as significantly less: 500 x 10^9 ~= 465 x 2^30.)

These several billion bytes are like an enormous canvas that you, the programmer, can paint upon. Except the computer isn't you, and can't see them all at once: you have to precisely name the place in memory you care about. Modern CPUs number the bytes, starting at zero ('cuz we start at zero in CS, of course) and working their way up to 4 x 2^30 - 1 (if you have 4GB of "addressable" memory). This number is called the byte's *address*.

Modern CPUs usually work on larger units of information, called a *word*. Words on modern CPUs are typically 32 or 64 bits (4 or 8 bytes), and some special instructions can work on larger units still. A computer can have as much *addressable memory* as fits in a word. Thus, the *address space* is usually equal to 2^(word size) of the computer.

Right about now you're probably like, "did I accidentally sign up for a Computer Systems Engineering course?" And the answer is no. But I do want to make sure you have some intuition for stuff that's going to come up later in the course and 187. But I promise this is about as deep as we'll go into computer organization.

Variables, data types, and assignment

OK, so another thing you might be thinking is, "Huh, I've never written any Java where I've worried about addressing memory directly." To which I respond, "Eff yes! Ain't it great?"

(Arguably) one of the greatest success stories of computer science is the development of high-level languages and runtimes to free programmers from worrying (too much) about nitty-gritty details like those above. Of course, sometimes you will need to do so, but for problems that don't push the boundaries of what a computer can do, and that don't need to scale to millions of machines, and so on, you can effectively ignore many little details and still be an extremely productive programmer.

For example, suppose we're writing a web app for the PVTA: http://bustracker.pvta.com/infopoint/

We don't need to worry about painstakingly laying out four bytes to represent a bus number, then eight bytes to represent a distance, and then remember their memory address each time we want to use them.

(on board)

Instead we might write:

```
int busNumber;
double distanceTraveled;
```

(on board)

And the computer (the compiler and the runtime) generate code that lays out memory for us, gives each *variable* a name – which is translated into a memory address for us, we never even have to think about it – and even knows something about the variable – its *type*. The compiler can do "typechecking" to prevent us from attempting some impossible things, like, say, adding together a boolean and an integer. It can also use type information to make our lives easier: adding a floating-point number and an integer is actually non-trivial, but it's transparent in Java. Similarly, we can "add" integers to strings to build a new string with the integer converted to a string and pre- or appended to the other string.

There are roughly two kinds of types in Java: primitive types, and objects. (Arrays are kinda in between, but are actually an Object.)

What are the primitive types you know about?

- byte: 8-bit signed
- short: 16-bit signed
- int: 32-bit signed
- long: 64-bit signed
- float: 32-bit floating point
- double: 64-bit floating point
- boolean: true or false
- char: 16-bit Unicode character

Whenever you declare a variable of one of these types, Java lays out memory of the correct size and remembers the address, using the value stored in that memory address whenever you reference the variable. We blur the difference sometimes when speaking, but keeping the idea of a variable (a particular location in memory) and a value (in this context, the contents of a memory location) separate is very important.

A fundamental thing you can do with variables is *assign* to them. You can assign a literal value, like `i = 3;` to write a value directly to a memory location. You can assign from one variable to another, like `i = j;`. But let's be clear about what's happening: the computer isn't "copying j into i," even though that's how you might say it. It's looking up the value stored at the address of j, then storing that value in the address of i. This is more clear if you think about the result of a computation, like "i = j + k;".

The primitive types support various kinds of computation using "operators", which are built into the language (things like addition and subtraction).

What happens in the following code?

```
int i = 2;
int j;
j = i;
j = j + 1;
```

(on board) A memory location to hold an `int` is allocated, and initialized with the value 0. Another (different!) memory location is allocated. The value from the first location (`i`) is looked up; it's then copied into the second location. Then, the value from the second location ('j') is looked up, incremented by 1, and written back into that location.

In-class exercise

```
int i = 5;
int j = i;
j = j + 2;
i = i + j
```

What are the values of `i` and `j` at the end of this code?

(on board) A memory location to hold an `int` is allocated, and initialized with the value 5. Another (different!) memory location is allocated. The value from the first location (`i`) is looked up; it's then copied into the second location. Then, the value from the second location (`j`) is looked up, incremented by 2, and written back into that location. Finally, the value of each of `i` and `j` are looked up, summed, and stored in the location of `i`.

More administrivia

Attendance

Instructors supposed to take attendance for the first two meetings of any class, so let's do that now. Please let me know the right way to pronounce your name or if you want me to call you something else. You can set up SPIRE to show your instructors what you want to be called and what pronouns you use. This is super helpful to us, please do it!

(show SPIRE settings)

(take attendance)

How to get help

My office hours are Thursday at 9 in CS 318. Please come! I'm happy to answer questions, chat, review something from the lectures you didn't understand, and so on. It's your time, so you should come and make the most of it!

For those of you who are new to college or UMass, office hours here are typically drop-in (no appointment necessary) and first-come, first-serve chances to come talk to a professor or TA / course assistant about anything in the course.

Usually, students come with specific questions, but general questions about course material are fine too. Come see us if something from lecture or an assignment is unclear, please! But don't expect to just open your laptop and say, "my program doesn't work, can you help me find the problem?" Be ready to tell us what you've tried and why you're stuck.

TA and UCA office hours we'll post online once they are finalized – UCAs don't even start until next week, so I don't have their hours yet.

There will also almost certainly be SI ("supplemental instruction") sessions, which are run in the LRC, located in the DuBois (the tall brick tower) library. Our SI leader this semester is Aadit Bhatia. Look into this if you need extra help in the course. Again, we'll post details when they're available.

Please use Campuswire for questions! You would be shocked (or maybe not) to know that many students will ask the same question; it saves everyone time if you check Campuswire to see if your question has already been asked and answered. And it saves us time if we only have to answer one question. Thus, we're able to answer more questions, which everyone likes.

More Java!

Java review

Arrays

Arrays are a built-in form of "container" type – the first (non-primitive) data structure you likely learned about, and perhaps the first reference type. In particular, arrays are a linear sequence of values, all of the same type. In our examples today, they'll be arrays of primitive-type values, but arrays can hold objects (actually refererences to objects) as well, as we'll see. An array type is denoted with the `[]` suffix after a type. For example, an array of `int`s might be declared as `int[] busNumbers;`

(on board) The array type doesn't tell you how many of the thing are in the array. This information exists only once the array is *instantiated* – the memory allocated for it: `int[] busNumbers = new int[5];` What's happening here? First, `new int[5]` creates a new space in memory for ten integers, one after another. Then, the *address* of this memory space is stored in `busNumbers` – arrays are a *reference type*, so they hold a reference (or address) to the thing, no the thing itself.

If you want to access a particular piece of the array, you must *address* it correctly, indexing from zero; the arrays indexing operator (`[]`) looks up the memory address of the array, then finds the offset to value you are interested in.

(on board) So, for example, `busNumbers[0] = busNumbers[3];` means, in English, to look up the memory address of the array, then look up the value stored in the third (starting from zero, which we call the "zero-eth", so zero-eth, first, second, third) slot of the array, and store it in the "zero-eth" slot. Or, perhaps more ()or less?) confusingly, look up fourth element in the array and store it in the first.

Arrays also support some useful methods and properties, such as `.length`.

You can have arrays of arrays, for example, `int sudoku[][]` is an array of arrays of ints; this kind of two (or more) dimensional array is sometimes a more intuitive way to represent a problem than a one-dimensional array.

In class exercise

What's true about the two variables int busNumber and int[] busNumbers and their associated values?

Do they store ints?
Do they *directly* or *indirectly* (that is, by reference) store ints?
Do they name values, or memory locations?
Do they contain values, or memory locations?
Are they primitive types, or reference types?

Methods and scope

Nothing lasts forever. Variables (the *names*, not the *values*) only live for as long as they are "in scope." Suppose we have two methods:

```
int add(int i, int j) {
  return i + j;
}

void print() {
  System.out.println(i + " " j);
}
```

Will this compile? No. Why not? Because `i` and `j` are not defined *within the print() method* – in other words, they are not *in scope*. Scope means the portion of the program where a *variable* (again, not necessarily the value) is valid.

Within methods, any parameters are in scope for the entire method. A variable that's declared inline:

```
void aMethod() {
  ...some stuff...
  int x = 12;
  ... some more stuff
}
```

is only valid from where it's declared until the end of its current *block* – which as you may recall from 121 is denoted by the next closing curly brace `}`.

Variables declared by some constructs (such as `for` loops) are only valid within the body of those constructs.

The stack and the heap

When the JVM is executing some code, for example:

```
int compute(int x) {
  return doubled(x);
}

int doubled(int x) {
  return 2 * x;
}
```

How does it keep track of which `x` is which, and how do values move around? I'm going to simplify somewhat here, but essentially there are two regions of memory used to store values (and whose sections are named by variables): the stack and the heap.

As the JVM executes the code, it goes line-by-line, expression-by-expression, evaluating each expression and performing each statement. If someone somewhere called `compute(3)`, the first thing that would happen is the JVM would lay out some memory in the stack for the result of the computation, and for each of the parameters, and for any variables that are in-scope for the whole method (none here, it turns out). Then the parameters would be copied into their spaces. Then the method would start.

Next, `doubled(x)` would be called. "On top" of the stack, more memory would be layed out, for the return value and for `x`, which would be copied in. Then `doubled` would execute, and copy the result into the right spot on the stack. Then control would `return` to `compute`, which would copy the value off the top of the stack and into the right spot. And so on. Notice that variables *and values* are automatically removed/reclaimed on the stack, and that we need only look at the top of the stack to find the current variable and value it holds.

"But Marc," you might be thinking, "can a value that's not the return value exist after a method ends?" Yes, those live on the heap, and the JVM is responsible for managing them dynamically via its garbage collection system. We'll talk about this (again, at a high level) later – but to a first approximation, whenever you write `new`, you're using the heap.

Stack allocation

```
double quadratic(double a, double b, double x) {
  double firstTerm = 0.0;
  double result = 0.0;

  firstTerm = a + Math.pow(x, 2); // first computation
  result = firstTerm + b; // second computation
  return result;
}
```

How many `double`s worth of space are allocated on the stack *in just this method*, including the parameters and return value?

Note our use of the `Math` class and its static method `pow` to square x.

More administrivia

Some words about assignments and grading.

There will be:

- written assignments, where you'll do short written and/or programming problems that reflect, reinforce, and explore things we talk about in lecture. You'll be able to do these with a small group, or alone if you prefer, they'll be assigned after each lecture and due at the start of the next. We'll use Gradescope for these. (25%)
- programming assignments, where you'll be asked to engage in the practice of programming, and will be able to submit your work to an online grader for immediate feedback. Some of these may also have some individual feedback from the TAs and UCAs on your code. These you can do with a single partner, or again, alone if you prefer. Again, weekly and due Fridays. These will be submitted through Gradescope. (30%)
- six or seven self-quizzes, where you'll *individually* work on a few small programming or short-answer questions. These are a way for you (and us!) to see how you're doing with the course content without having help from a friend. These will be give in lab on Mondays, approximately every other Monday. (20%)
- six or seven lab exercises. The other Mondays you will attend lab and do small group exercises that either introduce or develop further a concept we'll be covering in lecture. (10%)
- a final exam. There will be a written final exam! You cannot pass the course unless you pass the final! But the final will feel a lot like the quizzes and is not intended to be a stressful event. I have activated the "cannot pass the course unless" clause fewer than five times over nearly a thousand students. (15%)

There will not be any graded lecture exercises. I didn't know whether we'd be in person or remote, nor whether if we were in-person, how closely you could work together. And I would rather you not feel conflicted about coming to class for points if you are sick. So while I will be happy to interact with you and I'll be asking you to interact with one another, nothing graded will typically happen in lecture, and lecture recordings should be available shortly after each lecture – as will the notes.

Several things to note:

Lab attendance (and group work – you *must* work in groups in the lab) is not optional! And lecture attendance is expected – if you never come to lecture, you're probably also never going to watch lecture videos, and while it's in theory to do OK in that circumstance, it's not something I've seem much evidence for in practice.

Assignments (written and programming assignments) have a due date, clearly marked on the course web site. That is when they are due!

However, if you read the syllabus, you will see that there is a two-day, no-penalty extension available for you to grant to yourself for programming assignments if you need it; the flip side of this extension policy is that we *do not accept things after that date*. You already had two extra days! The exceptions will be at our discretion and with written documentation to support your reason.

This extension is only for programming assignments, *nothing else*.

There are more details to some of this. I expect you to read the syllabus in its entirety and ask questions if you have them.

If you have an accommodation due to a recognized disability, you need to contact me to tell me if and how you plan to use it (after getting it recognized by Disability Services and having them send me a letter.) In particular, things like extra time on assignments need to be requested in advance of, not after, the due date. Or another example: if you need extra time on the final exam, I need time to figure out how and where you'll take it. I'm 100% happy to figure out accomodations that work – I want you to succeed!

End of class reminders

Read the Syllabus!

Assignments and their due date will go up on Moodle/Gradescope as they become available. This includes the first programming assignment, and the first written assignment, both of which are now available, one of which is due in less than 48 hours!

Suggested reading and lecture notes will also be posted to Moodle.

Please ask questions on Campuswire (and check to make sure the question hasn't already been asked). We'll get office hours scheduled soon.

Finally, if you didn't receive email invitations to Campuswire and/or Gradescope to your @umass.edu email address, please email me and let me know *immediately*. (This is most likely because you added the class after I did the import into those tools last week.)

Last modified: Wednesday, February 2, 2022, 6:17 AM

◀ hamspam-student.zip

Jump to...

Lecture notes 01-02 ▶

© University of Massachusetts Amherst • Site Policies • Site Contact

Moodle Help for Students • Moodle Help for Instructors

You are logged in as Tanish Patel (Log out)
COMPSCI186-SEC01 SP22

Get Help
Help for Students
Help for Instructors

COMPSCI186-SEC01 Using Data Structures Spring 2022

Lecture notes 01-02

Annoucements

Remember, programming assignment 1 is due Friday at 5pm.

And, there is a quiz in person in lab on Monday. It's not intended to be super challenging – 121-level review material to get you used to taking code quizzes on paper, an important skill (sadly) for programming classes. Samples will be up on Moodle.

The next programming assignment will go up later on Friday.

If you didn't receive email invitations to Campuswire and/or Gradescope to your @umass.edu email address, please email me and let me know *immediately*. (This is most likely because you added the class after I did the import into those tools last week.)

If you added late and missed today's due date, let me know that, too.

Java and control flow

Control flow is the order in which statements in an imperative language are executed. By default, the program starts executing at the `main` method, and goes line by line, executing a sequence of statements. In Java, this default is overridden in one of several ways:

A method could be invoked. This unconditionally sends the flow of control into that method. Similarly, we can `return` control from a method to the method that invoked it.

Or, the flow of control could branch conditionally. Most commonly we think of `if[/then]` and `if[/then]/else`, but Java also also supports the `switch` statement as well. Probably you should never use the `switch` statement – let us never speak of them again! (Well, not much, anyway.)

Or, we could execute a sequence of statements zero, one, or more times, as in the `while` (or `do/while`, which we'll also neglect) or `for` statements. Within these statements, we might `break` out of the loop, `continue` back to the top of the loop, or exit the loop and `return` from the entire method.

Or, we could `throw` an exception, unwinding the stack of method calls until either the exception is caught (in other words, control resumes in the exception handler's `except` clause) or the `main` method is reached, in which case the JVM exits and outputs an error message. `NullPointerException` anyone?

There are a few other ways control flow can by modified, but the above hits most of the highlights.

Let's talk about each of them in turn.

Method invocation

Method invocations usually looks something like `method(parameter1, parameter2)`, in other words, the name of a method followed by its parameters (either variables or literal values). If there is no preceding variable name, e.g., `anObject.method()`, the method is resolved first in the context of the current object, then as a static method. Otherwise, the method is resolved in the context of the leading object name. You can also invoke static methods of other classes by prefixing with the class name; more about this in a bit.

`return`, no matter where it is, exits the current method. For example, you can `return` from within a loop and the loop will terminate immediately.

Conditional branches

`if[/then]/else` statements evaluate an expression as a boolean, then branch one way if it is true. If there's an `else`, the other way is followed – but with no `else` and an expression that evaluates to `false`, the statements are just skipped over.

How does Java delimit which statements are part of the `then` or `else`? It executes *only the following statement*. "But Marc!" Yes, I know, curly braces.

If you enclose a sequence of statements in curly braces, you have created a *block*. In terms of control flow treats a block as a single statement, so that you can, for example, branch to a sequence of statements after an `if`. Blocks also serve the purpose of defining a lexical scope, as we mentioned last class.

Speaking of, the fact you don't *need* to declare a block using `{}` after an `if` statement leads to one of the most common kinds of bugs that we see in first- and second-year programming classes. Students write code that looks like:

```
x = 5;
if (someCondition)
  x = x + 10;
else
  x = x - 10;
System.out.println(x);
```

And on its face, that's fine. The problem is when you later realize you need to modify the stuff that happens on the basis of `someCondition` to include more statements:

```
x = 5;
if (someCondition)
  x = x + 10;
else
  x = x - 10;
  x = x * 2
System.out.println(x);
```

Do you see the problem? Just because you indent the code, and things *look like* a block, it doesn't mean they are. Experienced programmers will insert the `{}` always, so that if later they modify the branch, things work correctly. Many style guides (e.g., Google's) require this, and IDEs like Visual Studio Code can be set to enforce it (either through warnings or errors, or to just reformat automatically every time you save).

In-class exercise

```
boolean someCondition = true;
int x = 10;
if (someCondition)
  x = x + 2;
else
  x = x - 2;
  x = x / 2;
System.out.println(x);
```

What's the output?

```
boolean someCondition = false;
int x = 10;
if (someCondition)
  x = x + 2;
else {
  x = x - 2;
  x = x / 2;
}
System.out.println(x);
```

What's the output?

Attendance

A brief pause to once again take attendance. Last time!

A little more on conditionals

So there's two basic times you might use a conditional. One is when you might want to do a thing, or not. That's a plain-old `if` statement.

```
if (isTime) {
  doTheThing();
}
```

But sometimes you want to do *either* one thing, *or* the other, but not both. That's `if`/`else`.

```
if (isThingA) {
  doThingA()
} else {
  doTheOtherThing();
}
```

And sometimes, you want to do one, and only one, of multiple options. That's a chained series of `if-else` statements – *not* just `if`s! (Why not just multiple `if`s? Because more than one could be, or become true – chained if-elses *force* only the first matching case to execute.)

```
if (caseA) {
  doCaseA();
} else if (caseB) {
  doCaseB()
}
} else if (caseC) {
  doCaseC()
} //... etc ...
else {
  // usually a good idea to have a catch-all -- if you should never
  // get here, then make that explicit with an error message
  throw new Exception("Programmer error, shouldn't get here!")
}
```

You could also use an equivalent `switch` statement, but they're basically a footgun (that is, they help you shoot yourself in the foot) and should generally be avoided in Java, unless you use the newer (Java 15+) `switch` expressions.

Why are they dangerous? A couple reasons, but most importantly, they have "fall-through", so if you forget your "break" more than one case could execute. This was the root cause of an enormous security error (https://heartbleed.com) from 2014 – the code was in C, but the syntax (and footgun) is basically the same in Java.

The new-style `switch` expressions have a slightly different syntax, and are both *exhaustive* (the compiler checks to make sure all possible inputs are handled) and *mutually exclusive* (only one case is ever activated). Most newer programming languages include *only* this kind of switch. They're way better!

Loops

Sometimes you want a statement to run zero or more times, as the result of a conditional. You want a `while` loop.

```
while (someCondition)
  doTheThing();
```

Of course, in the above example, `doTheThing()` had better change `someCondition` at some point, or the loop will never terminate (through normal means, anyway – it could throw an exception or otherwise exit the program, say, via `System.exit()`).

Also, why don't we check if `someCondition == true`? Because we don't need to: the if statement is checking if the condition is true already; there's no need to duplicate the code here.

Just like `if` statement branches, it's a super-good idea to always enclose loop bodies in a `{}` delimited block.

Note that there are three other ways, from within a loop, that we can change control flow. If we want to, we can immediately exit a loop with a `break` statement:

```
while (true) { // if you ever see this, you better see some way out of the loop, too
  boolean amDone = doTheThing();
  if (amDone) {
    break;
  }
}
```

In-class exercise

```
boolean done = false;
int x = 1;
while (!done) {
  x = x * 2;
  if (x > 10)
    break;
}
```

Will this loop terminate?

Loops, continued

`continue` returns to the top of the current loop; `return` exits loops (and the rest of the method!).

In addition, you can use the `continue` statement to return to the "top" of a loop (in a `while`, it forces re-evaluation of the condition):

```
int i = 0;
while (true) {
  i = i + 1;
  if (i < 10) {
    continue;
  }
  System.out.println(i);
  break;
}
```

This will repeatedly increment `i` and *not* reach the `println` until `i == 10`; then it will print `i` and exit the loop. What would happen if we removed the `break`?

What about this loop? What does it do?

```
int i = 0;
while (i < 10) {
  if (i % 2 == 0) {
    System.out.println(i);
    continue;
  }
  i = i + 1;
}
```

`for` loops

Finally, there's the `for` loop.

IMHO, *when possible*, the best way to use a `for` loop is with the *iterator* syntax. Arrays (and more importantly, container types that implement the `Iterable` interface, support the iterator protocol. If you care about the *contents* of an array (or any iterable), but not about the *indices*, then you can use the enhanced for loop.

Suppose you have an array you want to iterate over and do something to, like print its members. You can do it the "old fashioned" way:

```
int[] array = {1, 3, 3, 7};

for (int i = 0; i < array.length; i++) {
  System.out.println(array[i]);
}
```

Or you could use the enhanced `for` loop:

```
int[] array = {1, 3, 3, 7};

for (int e : array) {
  System.out.println(e);
}
```

Notably, the enhanced `for` loop prevents you from making indexing errors – it guarantees that it will visit each element of the array exactly once, assigning it to the iterator variable you provide each time through the loop.

The other way (the "old fashioned" way) is more general: you specify an initial state, a termination condition, and an update statement. Unfortunately this is prone to errors:

```
void printInts(int[] theInts) {
  for (int i = 0; i <= theInts.length + 1; i++) {
    System.out.println(theInt[i]);
  }
}
```

See the error(s)? Hard to follow exactly what's going on with that termination condition, huh?

In-class exercise

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

What will this method do?

More on `for` loops

But there are at least two major times when this "old-fashioned" style comes in handy.

The first is when you need access to the index, not just the element.

And the second (which is related) is when you need to compare different elements within the same array. For example, let's say you wanted to write a method to check if an array of `int`s was sorted in ascending order. You'd need to make sure each element was less than or equal to the element after it (and make sure you didn't check anything past the end of the array!):

```
boolean isSorted(int[] array) {
  for (int i = 0; i < array.length - 1; i++) {
    if (array[i] > array[i + 1]) {
      return false;
    }
  }
  return true;
}
```

In either style of `for` loop, you can, just as in `while` loops, modify the flow of control with `break`, `continue` (which performs the update and checks for termination), and `return`.

There are a few common patterns of `for` loops you should learn about and master, as they'll make writing `for` loops easier for you. We've seen some of them already, but let's talk about patterns. Here they are:

Doing something to every element of an iterable: Suppose we have an array of Things, and we want to doSomething to each one. The enhanced for loop lets us do so in a straightforward manner:

```
void doAll(Thing[] array) {
  for (Thing t: array) {
    doSomething(t);
  }
}
```

Testing that everything in an iterable satisfies a property:: Let's say we want to check if every element in an array is even. The general idea here is to check each element one-by-one. As soon as one thing violates the property (that is, is odd), we can return false. If we get to the end, we can return true:

```
boolean allEven(int[] array) {
  for (int e : array) {
    if (e % 2 == 1) {
      return false;
    }
  }
  return true;
}
```

Another example is a linear search, where we scan every element of an iterable and do a thing if an element is found. We do a different thing if it's not found:

```
boolean containsValue(int[] array, int value) {
  for (eint e : array) {
    if (e == value) {
      return true;
    }
  }
  return false;
}
```

The pattern is the same; the test and the return values differ.

Or if we wanted to see if a String contained a particular letter we can iterate over its contents. Probably the easiest way to do this is using the `charAt` member method of `String`, as `String`s don't directly support enhanced `for` loops:

```
boolean containsChar(String s, char c) {
  for (int i = 0; i < s.length(); i++) {
    if (s.charAt(i) == c) {
      return true;
    }
  }
  return false;
}
```

Again, the pattern is the same; the test and the return values differ.

Comparing elements in an array for some property: We saw this already in the `isSorted` method. Here, we cannot easily use the enhanced for loop, since we need access to the index (to check the current against the next value):

```
boolean isSorted(int[] array) {
  for (int i = 0; i < array.length - 1; i++) {
    if (array[i] > array[i + 1]) {
      return false;
    }
  }
  return true;
}
```

The pattern is similar to before, but we need to sure we access the right elements of the array, and we need to be careful with bounds. You could rewrite it as:

```
boolean isSorted(int[] array) {
  for (int i = 1; i < array.length; i++) {
    if (array[i - 1] > array[i]) {
      return false;
    }
  }
  return true;
}
```

but I find the former more natural.

Exceptions and other abnormal exits

We'll cover these more later – for the first part of this course, I won't be asking you to write code in this class that needs to do much with them. We'll return to them in more detail when the need arises.

End of class reminders, again!

Assignments and their due date will go up on Moodle as they become available. Next week's programming assignment will go up sometime after Friday morning.

Quiz in lab Monday! Sample quiz will be up soon if not already.

As always suggested reading and lecture notes will also be posted to Moodle.

Please ask questions on Campuswire (and check to make sure the question hasn't already been asked). We'll get the remaining office hours scheduled soon, once the UCAs start next week.

Finally, if you didn't receive email invitations to Campuswire and/or Gradescope to your @umass.edu email address, please email me and let me know *immediately*. (This is most likely because you added the class after I did the import into those tools last week.)

Last modified: Thursday, January 27, 2022, 1:25 PM

◀ Lecture notes 01-01

Jump to...

Sample quiz questions ▶

© University of Massachusetts Amherst • Site Policies • Site Contact

Moodle Help for Students • Moodle Help for Instructors

You are logged in as Tanish Patel (Log out)
COMPSCI186-SEC01 SP22

Get Help
Help for Students
Help for Instructors

COMPSCI186-SEC01 Using Data Structures Spring 2022

Lecture notes 02-01

Welcome

Welcome to people of all ages, all colors, all cultures, abilities, sexual orientations and gender identities.

Welcome to those who identify as biracial, multi-racial, or multi-ethnic.

Welcome especially to those of you who added the course recently, and to those of you coming from the 5-colleges!

Announcements

Are you a new add to the course? You need to get in touch with me immediately! Send me an email and I'll get you into Gradescope and CampusWire.

As ever, there are things that are due: I will post the WA shortly after lecture (if it's not up already). You also have a PA due – the "Bus simulator". This assignment is often reported by students as being "the most annoying" – it requires a careful case analysis and writing a sequence of if-then-elses correctly. There's no magical, elegant, one-line solution to it, you just have to specify the behavior of the bus correctly in code. Get started on this one early, as you will want time to think about it, and also to ask questions about it! Assignments are due Friday, so start soon if you haven't already!

Related to that, note that office hours are now up, and you have the option to talk to course staff every weekday. Come say hi if you have questions – we can't help if you don't ask!

Developing with Visual Studio Code

So, you've used Visual Studio Code (or just "Code" for short) in a very hand-holding way to complete HamSpam, but it hasn't really required you to understand the use of unit tests, or how you can use the debugger to help find errors in your code.

This week, we're going to do a little practice run through Code, demonstrating some of its features that you'll find useful for the rest of the semester.

In particular, we'll show you how you might write and use your own "unit tests" to check your progress toward completing an assignment. We provide tests for the projects early on, but you'll need to start writing your own at some point this semester, and sooner is probably better than later.

We'll also show you how you might use the debugger to figure out what's going on when things aren't working the way you'd expect. Sometimes "caveman debugging" (liberal use of "println") is good enough, but sometimes it's helpful to examine a program's state while it's running to figure out what's going on.

We'll use everyone's favorite game, tic-tac-toe (also known as "naughts and crosses") as our running example.

(demo the game on the board)

Creating a project

For this class we typically give you a project template to work with. One way to create such a template is to just use Code to make a new project. From the command palette, choose "Java: Create Java Project". In a moment, you'll be prompted to "Select Project Type"; choose "No Build Tools". Then you'll be prompted to pick a location; this is the folder where the project folder will go. Finally, choose the project name; the project will be stored in a folder of this name inside the location you chose in the previous step.

(Note that in actual lecture I fumble-fingered the above the first time, and re-did these steps when doing unit tests, later.)

By default Code creates a `README.md`, a `src/` directory (and a very short "Hello World" sort of thing called `App.java`) to store source files, and a `lib/` directory to store libraries."Libraries" are bundles (usually in the form of ".jar" files) of Java code written by others that this project will leverage. For example, this week's programming assignment uses the "Processing" jar to draw the bus simulator graphics and handle mouse and keyboard input. We won't use any libraries for graphics in our example, instead using simple text-based input and output. But we will use a pair of jar files to enable unit testing, as I'll show shortly.

OK, so now we have an new programming project project. A new project! What should you do?

a. immediately start coding whatever comes to mind, like a hyperactive gerbil that just had a quadruple espresso on an empty stomach!
b. maybe stop and think for a bit about what the program should do, and how it should do it.

Blank slate

Now, really, what should we do? Think about about the *data* our program is going to represent, and the *actions* we're going to perform upon that data.

What kinds of data is this program going to need to keep track of? (If we were meeting in person, I'd ask you in lecture). Think in terms of nouns here.

- a board / spaces
- which spaces have been claimed by which player
- winner/loser/draw (or game still in progress?)

And what kinds of actions are we going to be performing? This time, verbs.

- creating a new board
- claiming spaces
- checking winner
- checking end-of-game
- printing the board (actually, converting it to a string)

If we squint at this for a bit, we get a sense of what types of data we're going to have and the methods we might use to. We need something to represent the state of a space, and an array of these somethings to represent the board. And...that's about it, actually – all of the other things can be computed by looking at the board – we don't need to keep track of them, though we might choose to do so to keep the code simple or clear.

Keeping track of things we don't need to can be detrimental: we might make a mistake (and desynchronize between our board and, say, an `xHasWon` instance variable) which would lead to hard-to-track down bugs.

Whether to keep track of values, or to compute them from other values we are tracking, is a judgment call. If computing one of these properties was actually hard – that is, used a lot of *time* or *space*, or the code to do so was complicated and error prone, we might just track the thing in a variable. But if the code to update that variable was complicated and needed to be considered in many methods, we might compute the value, instead.

You'll develop a sense for this as you gain in programming experience (and as you learn how to accurately determine which things are actually costly in terms of time and space).

Getting started

Let's start by creating an empty board class, and put our state (instance variables) and behavior (methods) there. Notice I put the classes into a package; this is generally expected in "real" Java programs. Also notice I have to make a directory structure that corresponds to the package structure (that is, classes in a `tictactoe` package have to be in a `tictactoe` directory inside my source directory).

```
package tictactoe;

public class TicTacToeBoard {

}
```

Spaces and the board

How should we represent spaces? Two-state variables are naturally represented as booleans, but spaces can be in one of *three* states: empty, 'X' or 'O'. How can we deal with this?

One option is to use Strings ("empty" "x" "o"); another might be to use integers (0 = empty, 1 = x, 2 = 0), where we restrict our attention to the values of interest. These aren't bad choices per se, though they do come with some risks.

```
private String[] board;

// or we might use
// private int[] board;
```

Suppose we define our board as an array of strings, where each entry is `X`, `O` or (empty).

```
board[0] = "X";
// ... later ...
board[1] = "0";
```

Is this code correct? And, will the compiler catch the error?

For strings, if we make a typo in the string in our code somewhere ("X" vs "x") then comparisons will fail and some conditionals might not execute.

For ints, they aren't very readable, and again, we might slip with our fingers and accidentally assign the value 3 to a board space.

One way to mitigate these issues somewhat by using "named constants". We have to use two *keywords* to signal some things to the compiler:

- `final`: This variable's value is not variable at all, but instead constant. It's a compile-time error to attempt to change its value.
- `static`: There is only one copy of this variable, shared between all instances of the class. We don't need a separate copy per-instance. (Strictly speaking, this is optional, but since the values of the spaces won't vary between games, we might as well make that clear.)

```
public static final String X = "X";
public static final String O = "O";
public static final String EMPTY = "empty";
```

"Marc," you may be thinking, "this seems like a lot of extra work." Well, "a lot" is relative. But the goal here is to let the computer (in this case, the compiler) help you. If you only use these constant names in your code; it prevents you from making typos and is a good idea if you are working with constants. It also tells other people what the otherwise "magic numbers" or "magic strings" in your code mean. The vast majority of programming jobs in the real world don't let you write everything from scratch – you're often reading other people's code, too, so it pays to make your code legible.

Enumerations

We might also consider using an "enumeration type." Enumeration types are kinda like very short classes, which can only hold a small set of values, that have specific support in the Java language. Enums are *reference types*, and like all reference types, their default value is `null` – watch out for that!

An enum provides additional benefits over constants. Namely, it defines its own type so you can't accidentally compare it with another String. (Maybe someday take a deeper dive into a programming language that supports algebraic data types and pattern matching to see how you can *really* leverage the compiler to help you avoid mistakes.)

```
private enum Space {
    X, O, EMPTY;
}
```

(Note from Marc: In class we went with `char[][] board` – I'll switch back to that in the notes once we get past the `enum` things, but `enum` is here for you if you ever want it in the future.)

Back to the job at hand! Now that we've defined the types of values that can be stored in spaces, we need a place to put them. We'll create a Space array:

```
private Space[] board;
```

Now, let's see if we can see the board. Switch over to our driver and add this to the `main()` method:

```
    public static void main(String[] args) {
        TicTacToeBoard board = new TicTacToeBoard();

        System.out.println(board);
    }
```

And...you'll see something like `tictactoe.TicTacToeBoard@677327b6`! (It might look different each time you run the program.) What's this?

We're attempting to print `board`, which is an instance of `TicTacToeBoard`. What does `println` do when we pass it an object? It attempts to coerce it to a string, using the object's builtin `toString` method. But we didn't define one. Where is this string coming from?

Java classes use inheritance, and if a method isn't defined in the current class, the Java language says we should look up the inheritance hierarchy to find its definition. What's the superclass of TicTacToeBoard? We didn't define one (using `extends`), but there is a default for all objects: `Object`. What does it have to say about `toString`?

http://docs.oracle.com/javase/8/docs/api/java/lang/Object.html#toString--

So if we don't define a toString, we get the typename and the hash code. (And lookie there, the default value of the hashcode is the object's address! So they are, kinda, sorta, sometimes, available.)

Anyway, one of the important rules for development and debugging is that you can't fix what you can't see. So let's write our own toString method now. It should convert the current instance of the board into a string:

```
    public String toString() {
        String string = "";
        for (Space s : board) {
            string = string + s;
        }
        return string;
    }
```

What's the output?

And... Boom! Null pointer exception! We didn't initialize the board! Let's make a constructor and do that.

```
    public TicTacToeBoard() {
        board = new Space[9];
    }
```

And... not quite Boom! But still lotsa null. Eff null, seriously, what were language designers thinking? The guy who invented null has since claimed it was the billion dollar mistake, and that's before adjusting for inflation.

How do we fix it? By *fully* initializing the board:

```
    public TicTacToeBoard() {
        board = new Space[9];
        for (int i = 0; i < board.length; i++) {
            board[i] = Space.EMPTY;
        }
    }
```

Unfortunately we still get something ugly: `EMPTYEMPTYEMPTYEMPTYEMPTYEMPTYEMPTYEMPTYEMPTY`.

Let's fix this in two parts. First, let's teach the enumeration `toString()` itself:

```
        public String toString() {
            if (this == X) {
                return "X";
            } else if (this == O) {
                return "O";
            } else if (this == EMPTY) {
                return ".";
            }
            return "WTFBBQ";
        }
```

Annoyingly we can't prove to Java that we've covered all the cases here (again, languages with better type systems can handle this) so we need a default, obviously wrong value in case of programmer error. Choose your own if you like.

Then, while we don't need to, it might be nice to structure the board's `toString` output to be a little more human-readable. We want spaces between each item, and a newline at the end. And we want to trim the last newline. OK, here we go:

```
    public String toString() {
        String string = "";
        for (int i = 0; i < board.length; i++) {
            string = string + board[i];
            if (i % 3 == 2) {
                string = string + "\n";
            }
            else {
                string = string + " ";
            }
        }
        return string.substring(0, string.length() - 1);
    }
```

Well, not beautiful but it gets the job done. It might be marginally more efficient to use a `StringBuilder`, but enough premature optimization for now.

To Enumerate or Not

For small programs like this one, an `enum` is probably overkill; let's go back to named constants. But for larger programs (and, if you use the "better" version of `switch`, the *expression*, not the *statement*), they are quite useful.

Or Not to Enumerate

OK, so that's using `enum`. What if we went with a two-dimensional `char` array, instead? The code would then look like this:

```
package tictactoe;

public class Board {

    private char[][] board;
    public char X = 'X';
    public char O = 'O';
    public char EMPTY = '.';

    public Board() {
        board = new char[3][3];
        for (int i = 0; i < 3; i++) {
            for (int j = 0; j < 3; j++) {
                board[i][j] = EMPTY;
            }
        }
    }

    public String toString() {
        String s = "";
        for (int i = 0; i < 3; i++) {
            for (int j = 0; j < 3; j++) {
                s = s + board[i][j];
                if (j == 0 || j == 1) {
                    s = s + " ";
                }
                else {
                    s = s + "\n";
                }
            }
        }
        return s;
    }
}
```

Tests

So we've been manually running our driver after each change to see if the output matches what we want. You know what else is really good at running things repeatedly and checking for matches against expected results? A computer.

It turns out this is the basis of automated testing, and there's a system we can use in Java called JUnit to do this for us. We can still use our driver, of course, but we can also run *automated* tests to make sure that output is what we expect it to be.

In most class projects testing is already set up for you, and you can just edit the classes yourself if you like.

(Again, in class I fumbled this, but in Code, you open the "testing" on the Left, then "Enable" testing, and then select "JUnit" to be able to follow along here. This adds the relevant "jars" – which are self-contained Java libraries – to the project. You generally won't need to worry about doing this for class projects, but if you do your own Java development you might need to set up these (and other) jars.)

Now, let's add a test to capture the expected behavior of `toString` on an empty board:

```
@Test
public void testEmptyBoard() {
    Board b = new Board();
    assertEquals(". . .\n. . .\n. . .\n", b.toString());
}
```

First, note that the method is prefixed with an `@Test` decorator. This tells Code (all tools, that understand unit tests, really) to run this method as a test – not all methods in a "Test" class will be run as tests, only `@Test` methods.

Next, note we had to deal with the "newlines" or line breaks. Strings in Java source code can't contain explicit line breaks. So you "encode" them using an "escape character". Escape characters are short sequences of characters prefixed by a "backslash" `\`. `\n` means "don't actually treat this as backslash n, treat it as a newline." There are others (and they're used in other contexts, too) but this is enough for now.

Next, note the test returns void. We don't care about it's return value; instead, we use an *assertion* inside it.

Speaking of which, `assertEquals` checks that the first argument (the *expected* value, which we hardcode to the "right" answer) and the second argument (the *actual* value, which should be computed by the code being tested) are equal; if so, the test passes, if not, it fails. The test can also have an error, if it (for example) throws an unexpected exception.

This is a pretty common pattern, actually, where you write and test code in a driver class, then encode your test as a test case. This way, if you break something by accident later, you'll discover it immediately before you move on.

More on tests

Good tests are generally small – testing just one thing.

They usually use `assertEquals()`, where the first argument is what's expected and the second is what's being tested. (They might use `assertTrue` or `assertFalse` when checking boolean results, and there are a few others, but we'll come back to them later). It's important to write tests this way, so that tools (like, for example Code) show the right things to you in their UI.

TicTacToeSoFar

So what do we have so far?

```
package tictactoe;

import org.junit.Test;
import static org.junit.Assert.*;

public class Board {

    private char[][] board;
    public char X = 'X';
    public char O = 'O';
    public char EMPTY = '.';

    public Board() {
        board = new char[3][3];
        for (int i = 0; i < 3; i++) {
            for (int j = 0; j < 3; j++) {
                board[i][j] = EMPTY;
            }
        }
    }

    public String toString() {
        String s = "";
        for (int i = 0; i < 3; i++) {
            for (int j = 0; j < 3; j++) {
                s = s + board[i][j];
                if (j == 0 || j == 1) {
                    s = s + " ";
                }
                else {
                    s = s + "\n";
                }
            }
        }
        return s;
    }

    @Test
    public void testEmptyBoard() {
        Board b = new Board();
        assertEquals(". . .\n. . .\n. . .\n", b.toString());
    }

    public static void main(String[] args) {
        Board b = new Board();

        System.out.println(b);
    }

}
```

We could move the test out of `Board` and to their own file, for example:

```
package tictactoe;

import static org.junit.Assert.assertEquals;

import org.junit.Test;

public class TestBoard {
    @Test
    public void testEmptyBoard() {
        Board b = new Board();
        assertEquals(". . .\n. . .\n. . .\n", b.toString());
    }

}
```

This is a pretty common pattern, actually, where you write and test code in a driver class or a class's main method, then encode your test as a test case. This way, if you break something by accident later, you'll discover it immediately before you move on.

Output legibility

Remember we chose the `EMPTY toString` value as a single `.` Usually this sort of thing is a good idea – anything to make your life easier when reading your code (and its output) will pay dividends in the future, either to you or the programmer who's reading your code in six months.

Schools, especially intro classes, are very misleading about the code lifecycle. You spend most of your time in school writing your own code. But in the real world, you spend far more time reading, modifying, and fixing others' code. And once in a while, when you're reading and debugging some particularly gnarly code, you'll come to an awful realization: The programmer who wrote this mess was you, six months ago. Don't let this happen to you!

Back to the programming

Let's sketch out the driver so we know what we'll need to do. How is a typical game played?

- The board is empty
- Players take turns until the game is over, that is
  - one player wins
  - or it's a draw (tie game)
- Taking a turn means choosing a valid move (and updating the game state accordingly)
- The game ends, the result is announced

What might this look like in code? Here's a first cut.

```
package tictactoe;

import java.util.Scanner;

public class Game {
    public static void main(String[] args) {
        Board board = new Board();

        try (Scanner scanner = new Scanner(System.in)) {
            while (!board.isGameOver()) {
                System.out.println(board);
                System.out.println("Player " + board.getCurrentPlayer() + ", choose your move (0-8): ");
                int move = scanner.nextInt();
                // also, check move validity
                board.move(move);
            }
            System.out.println("Winner! Player " + board.getWinner());
        }
    }
}
```

Notice that we don't worry about various error conditions (like if the player chooses an invalid move); we'll get to that later. Also notice we haven't yet defined any of these methods – we'll do that soon.

Also notice that we've implicitly divided the methods into *observers*, which return a fact about the underlying object *but which don't change the object's state*, and *mutators*, which change something about the object. It's good practice to put methods into one or the other of these groups, to help us reason about our code.

We'll pick up here next class. I'll link the code so far on Moodle as a download so you can follow along next class if you want to!

Last modified: Tuesday, February 1, 2022, 4:45 PM

◀ bus-simulator-student.zip

Jump to...

tictactoe.zip ▶

© University of Massachusetts Amherst • Site Policies • Site Contact

Moodle Help for Students • Moodle Help for Instructors

You are logged in as Tanish Patel (Log out)
COMPSCI186-SEC01 SP22

Get Help
Help for Students
Help for Instructors

COMPSCI186-SEC01 Using Data Structures Spring 2022

Lecture notes 02-02

Welcome

Announcements

Late add? Come talk to me right after class!

Back to the programming

Let's sketch out the driver so we know what we'll need to do. How is a typical game played?

- The board is empty
- Players take turns until the game is over, that is
  - one player wins
  - or it's a draw (tie game)
- Taking a turn means choosing a valid move (and updating the game state accordingly)
- The game ends, the result is announced

What might this look like in code? Here's a first cut.

```
package tictactoe;

import java.util.Scanner;

public class Game {
    public static void main(String[] args) {
        Board board = new Board();

        try (Scanner scanner = new Scanner(System.in)) {
            while (!board.isGameOver()) {
                System.out.println(board);
                System.out.println("Player " + board.getCurrentPlayer() + ", choose your move (0-8): ");
                int move = scanner.nextInt();
                // also, check move validity
                board.move(move);
            }
            System.out.println("Winner! Player " + board.getWinner());
        }
    }
}
```

Notice that we don't worry about various error conditions (like if the player chooses an invalid move); we'll get to that later. Also notice we haven't yet defined any of these methods – we'll do that soon.

Observors vs Mutators

Notice that we've implicitly divided the methods into *observers*, which return a fact about the underlying object *but which don't change the object's state*, and *mutators*, which change something about the object. It's good practice to put methods into one or the other of these groups, to help us reason about our code.

Refactoring

As we program, sometimes it becomes clear we've named something poorly, and a better name would be more readable.

For example, what about renaming `s` to `string` in our `toString` method? Code lets you "refactor" code (what "refactoring" means is kinda complicated, but...). One simple refactoring is to rename a variable or type. You could just find-and-replace, but Code is smart enough to change only the variable you "mean" (for example, if you rename `x` to `count` in a method, it won't change any variable named `x` in other methods). Select the variable or type, then choose "Rename" in the command palette, or right click, to do this task.

Usually this sort of thing is a good idea – anything to make your life easier when reading your code (and its output – remember `toString`!) will pay dividends in the future, either to you or the programmer who's reading your code in six months.

Schools, especially intro classes, are very misleading about the code lifecycle. You spend most of your time in school writing your own code. But in the real world, you spend far more time reading, modifying, and fixing others' code. And once in a while, when you're reading and debugging some particularly gnarly code, you'll come to an awful realization: The programmer who wrote this mess was you, six months ago. Don't let this happen to you!

Ways to implement an observer

`getCurrentPlayer()` returns the current player. We can either explicitly keep track of whose turn it is (in an instance variable), or compute it by looking at the board (if there are an equal number of Xs and Os, it's X's turn; if there are more Xs, it's O's turn). Let's keep track explicitly for now by adding an instance variable and initializing it in the constructor.

```
    private char currentPlayer;
//...later, in the constructor
        currentPlayer = X;
```

Let's make sure we return it in our getter:

```
    public char getCurrentPlayer() {
        return currentPlayer;
    }
```

Notice we had to change the return type from the autogenerated `String` to `char` – but that's OK! It better reflects our intent at any rate.

A mutator

And, at the end of every move, it should be the other player's turn, so let's encode that, too:

```
    public void move(int move) {
        if (currentPlayer == X) {
            currentPlayer = O;
        } else { // currentPlayer == O
            currentPlayer = X;
        }
    }
```

Note that this method, which changes the object's state, does not return a value (it's `void`). Often mutators are void, since you generally don't want to mix mutation and observation.

Checking in the driver, it works, inasmuch as the players alternate.

Testing our mutator

We showed in the console that the player's turn will swap back and forth after each move(). Let's start by adding tests that captures this behavior:

```
    @Test
    public void firstPlayer() {
        Board b = new Board();
        assertEquals('X', b.getCurrentPlayer());
    }

    @Test
    public void secondTurnPlayer() {
        Board b = new Board();
        b.move(0);
        assertEquals('O', b.getCurrentPlayer());
    }
```

OK, and since we're getting crazy, how about we get move() working now, too?

```
        int row = move / 3;
        int col = move % 3;
        board[row][col] = X;
```

And test it in the console. Huh, there's a problem. Now, I get it, most of you see the problem immediately. But that's kinda on purpose here; some problems are easier than others to spot (and then fix). Sometimes they're simple problems you just can't see yourself; sometimes they're more subtle. Instead of caveman debugging this one (you know, adding "printlns" everywhere, let's try the debugger) so we can see what's going on.

First, we must set a breakpoint! Click the "gutter" and create a small red dot.

Then run under the debugger. You can do this in the command palette, or just click "Debug" right above the main method. Code executes until it reaches the breakpoint, and then pauses there. The yellow arrow shows you what line you're paused on.

The "debug" pane on the left is immediately useful – you can see all your breakpoints (and set other kinds), and you can examine the value of variables in the running program. You can also hover over things in the source to see their current value in the program.

We can "step" forward line-by-line. "over" means stay in this method; "into" means follow any method call in that line into another method; "out" means return to the calling method.

The second time through, we see that the wrong value is assigned to the board entry – whoops, we hardcoded it! Let's fix that:

```
        board[row][col] = currentPlayer;
```

And test it in the console with our driver.

And maybe a test now too:

```
    @Test
    public void firstPlayerMove() {
        Board b = new Board();
        b.move(0);
        assertEquals("X . .\n. . .\n. . .\n", b.toString());
    }
```

Again, (one of) the reason(s) we encode this stuff into tests is so that if we break something later, we'll spot it immediately and have a working test case to debug with.

Finding the end of the game

How about checking for the end of the game? In English, in `isGameOver()` we check to see if either player has won or if it's a draw. If any of those are true, the game is over, otherwise it's a draw. Similarly, our `getResult()` method needs to determine which player won, or if it's a draw.

Stopping and thinking for a second: we can express our draw check in terms of our winner check. If neither player has won, and the board is full, then it's a draw.

What does this look like?

```
    public boolean isGameOver() {
        return (isWinner(X) || isWinner(O) || isDraw());
    }
```

How do we check if a player has won? If they have three in a row. We could hardcode this, for example:

```
    private String getWinner() {
        if (board[0][0].equals(X) && board[0][1].equals(X) && board[0][2].equals(X)) {
            return X;
        }
        else if //...
    }
```

but we'd have to do 2 * (3 + 3 + 2) if clauses, mostly copy-pasted, making sure we didn't screw up the board index each time. That's OK, I guess, but it fails to capture the patterns we're looking for: rows, columns, diagonals. Maybe something like this would be better:

```
java private boolean isWinner(char player) { if (board[0][0] == player && board[0][1] == player && board[0][2] == player) { return true; } else if (board[1][0] == player && board[1][1] == player && board[1][2] == player) { return true; } else if (board[2][0] == player && board[2][1] == player && board[2][2] == player) { return true; } else if (board[0][0] == player && board[1][0] == player && board[2][0] == player) { return true; } else if (board[0][1] == player && board[1][1] == player && board[2][1] == player) { return true; } else if (board[0][2] == player && board[1][2] == player && board[2][2] == player) { return true; } else if (board[0][0] == player && board[1][1] == player && board[2][2] == player) { return true; } else if (board[0][2] == player && board[1][1] == player && board[2][0] == player) { return true; } return false; }
```

We could further condense the row/column checks with for loops (though it's not much cleaner).

How about a draw? If either player has won, it's not a draw. Or, if there are still empty spaces, it's not a draw. Otherwise, it's a draw:

```
    private boolean isDraw() {
        // has someone won?  NOTE WE SKIPPED THIS IN CLASS!
        if (isWinner(X) || isWinner (O)) {
            return false;
        }

        // otherwise, is the board full?
        for (int i = 0; i < 3; i++) {
            for (int j = 0; j < 3; j++) {
                if (board[i][j] == EMPTY) {
                    return false;
                }
            }
        }

        return true;
    }
```

This is the same pattern we saw in the past, where we check for a single violation of a property (and return false as soon as we find it); if we check everything and don't find a violation, we return true.

Notice I can freely call these "observer" methods (isDraw, isWinner, etc.): they only observe values, but they don't change anything. Having methods like this means I don't have to worry about "breaking" an object. A common beginner style error is to have a method that both answers a question about an object and changes its state (Imagine a method that, say, both makes a move and returns the current winner, if any. How do you then check winner without making a move, too? You can't, unless you add another method. But then there are two methods that check winner, which means you need to keep them in sync, which means it's easier to get later things wrong...).

Now we can also pretty easily return the game result, too:

```
    public String getWinner() {
        if (isWinner(X)) {
            return "X";

        }
        else if (isWinner(O)) {
            return "O";
        }
        else {
            return "nobody";
        }
    }
```

It appears to work (demo in console). One possible test might be:

```
@Test
public void testDiagonalXWinner() {
    Board board = new Board();
    board.move(0);
    board.move(1);
    board.move(4);
    board.move(3);
    board.move(8);
    assertEquals("X", board.getWinner());
}
```

Move validity

What does it mean for a move to be valid? Only that it's on the board, and that it's to a currently empty space. And, how will that change our driver? We'll loop until we get a valid move!

```
public static void main(String[] args) {
    Board board = new Board();

    try (Scanner scanner = new Scanner(System.in)) {
        while (!board.isGameOver()) {
            System.out.println(board);
            int move;
            do {
                System.out.println("Player " + board.getCurrentPlayer() + ", choose your move (0-8): ");
                move = scanner.nextInt();
            }
            while (!board.isValid(move));

            board.move(move);
        }
        System.out.println("Winner! Player " + board.getWinner());
    }
}
```

What does validity mean? That first, the move is on the board, and then, that it's to an empty space:

```
public boolean isValid(int move) {
    if (move < 0 || move > 8) {
        return false;
    }

    int row = move / 3;
    int col = move % 3;
    if (board[row][col] != EMPTY) {
        return false;
    }
    return true;
}
```

OK, enough tic-tac-toe. There's more you could do (like make sure moves are in the right range, and not over an existing player's move) but I'm sure you could figure it out (and test it!) yourself.

Code and testing note!

There's one last thing to tell you. Tests interact poorly with infinite loops. So at the top of most test files in this course's assignments, you'll see something like this:

```
@Rule
public Timeout globalTimeout = Timeout.seconds(10); // 10 seconds
```

This tells JUnit that if a test takes more than 10 seconds, it should just fail. If you don't do this and your code contains an infinite loop, the tests will "hang" on any test that calls that code. And your laptop will heat up and the fan will spin and you'll drain your battery, etc.

So the timeout stops this from happening. But it also stops the test if you're running it in the debugger. If you plan to use the debugger, you need to disable the timeout. Do this by commenting out *both* lines (the `@Rule` and the `public Timeout...` lines), and the debugger will no longer exit after 10 seconds – but if you have an infinite loop, tests will hang. (Gradescope takes 10 minutes to time out.)

Last modified: Thursday, February 3, 2022, 4:53 PM

◀ tictactoe.zip

Jump to...

anagrams.zip ▶

© University of Massachusetts Amherst • Site Policies • Site Contact

Moodle Help for Students • Moodle Help for Instructors

You are logged in as Tanish Patel (Log out)
COMPSCI186-SEC01 SP22

Get Help
Help for Students
Help for Instructors

COMPSCI186-SEC01 Using Data Structures Spring 2022

Lecture notes 03-01

Welcome

Announcements

Programming assignment 02 was due last Sunday. It was for some of you a doozy! As I said last class, often voted "most annoying" or "most tedious" in retrospect. This week's is different – it's mostly a series of stand-alone exercises to complete. This assignment will be easier to do if you *read the directions* on Moodle, and then *read the notes* in the Java code. Pro tip: Just like all assignments, you can get (small) partial credit just for submitting the project file as-is. Might as well do this early!

Also note that next week's PA04 will be shorter in terms of lines of code, but definitely conceptually a little more taxing than PA3. PA3 is a set of exercises; PA4 is a single, concrete problem you need to solve. The solution is spelled out in English in the writeup, but translating it to code will take some time and thought (like the bus simulator, but more thought, fewer `if` statements). Students often report that A02 was "the most annoying at the time" assignment, and that A04 was "the most challenging at the time". So start early and know that if you can handle A04, you will probably be able to handle everything else we plan to do this semester.

Lab this past Monday; quiz next Monday. Anything we've done so far is fair game; sample questions will go up Friday most likely.

If you have a grading question or regrade request: Start with the TAs (via regrade request on Gradescope). If you are unsatisfied with their answer, then ask me, but the first thing I'm going to do is check with them to see what their reasoning was.

This week's plan

We're gonna finish up a review on some key Java concepts today and Thursday, and then start motivating the idea of the List as an abstract data type. (We'll dive into the list in much more detail next week.)

Classes and objects

Java is said to be "object-oriented", which means that the language strongly encourages the use of objects as a paradigm for problem solving.

What's an object? A collection of related state and behavior. In Java, this means objects are typically variables and associated methods. The "blueprint" for objects is a class. Classes are also the place where we stick stuff that's not necessarily part of an object. (This falls out of Java's design: everything is part of a class.)

Instance and class variables

Local variables only exist within a method. But *instance* variables are the *state* that is part of a class. For each *instance* of a class (that is, for each object), there is a separate copy of the instance variable around.

```
public class Dog {
  private String name;

  public Dog(String n) {
    name = n;
  }

  public String getName() {
    return name;
  }
  // ... more code ...
}
```

Two different dogs can have different names:

```
  Dog dog1 = new Dog("Spot");
  Dog dog2 = new Dog("Ribsy");
  System.out.println(dog1.getName()); // prints "Spot"
  System.out.println(dog2.getName()); // prints "Ribsy"
```

Generally, things you do to one instance don't directly affect others – instances variables are separate across instances!

There are several kinds of instance variables: public, private, protected, and package come up most often. Public and private are denoted by keywords `public`, `protected`, and `private`, and package is denoted when you include no keyword at all. A public variable can be accessed (or "is visible") from anywhere. A private variable can be accessed only by instances of exactly this class. Package variables can be accessed only by classes in the same packages as this class. Protected variables can be accessed as package variables, and also by subclasses of this class.

The good news is that generally you don't have to worry too much about it! Generally, code you write will keep mutable variables private, and (sometimes) make constants and immutable variables public.

Packages? Mutable? What? Packages I'll talk about later. Mutability is kinda the opposite of immutability. Immutable variables are, in Java, variables that can be assigned to exactly once, and then are constant. (Constants are immutable variables that are initialized at the program start, and don't depend upon user input; immutable variables are more general, and can depend upon the program, like the `ham` and `spam` values in `HamSpam`.)

Generally, we mark an immutable variable with the `final` keyword. This helps to avoid errors – the compiler will now alert us if we try to change it (and in fact will treat it as an error).

Finally, we should mention *static* variables. Static variables are weird – they are *not* associated with an instance. Instead, there is *only one copy* of the variable, associated with the class, not the instance. So imagine we want to track the *species* of a `Dog`. All `Dog`s are of the same species, so we don't need an instance variable:

```
public class Dog {
  public static final String SPECIES = "Canis familiaris";
  private String name;
  // ... more code ...
}
```

(Here, I've also marked it as public, so anyone can access it; and final, since it's never going to change.)

Now, we can ask the dog *class*(!!) for this information:

```
System.out.println(Dog.SPECIES); // prints "Canis familiaris"
```

Mutable static variables, that is, non-final static variables, are the closest thing to "global variables" that Java has. They are typically considered a "code smell" – that is, an indication you're not doing something "the Java way" or even the right way. If you find yourself madly adding "static" keywords to resolve compiler errors, you should stop and ask course staff for help – it almost certainly means you've misinterpreted something. We'll be happy to get you on the right track.

Methods

So variables are state. Methods, on the other hand, are behavior.

In Java, methods are always attached to a class. A method consists of a declaration and a body, which is just a sequence of statements. Let's look at a declaration more closely:

```
public static void main(String[] args) {
  ...
}
```

WTF is going on here!?!?! is usually the reaction we get in 121 when people first start learning Java. But now, you probably know enough to understand most of it. Let's tackle it inside-out.

`String[] args` is the parameter to this method. One or more parameters are passed in, and they look like (and in many respects behave as) variable declarations. The difference is that their values are provided by the calling method (or in the very special case of `main`, by the JVM). Here, `main` gets an array of Strings, which are exactly what is passed on the command line to the java interpreter, e.g.,

```
public class PrintArgs {
  public static void main(String[] args) {
    int i = 0;
    for (String a : args) {
      System.out.println(i + ": " + a);
      i += 1;
    }
  }
}
```

```
> javac PrintArgs.java
> java PrintArgs Hello Marc!
0: Hello
1: Marc!
```

Next is the name of the method, which is by convention "camelCased," starting with a lowercase letter. Next is the method's return type, or `void` if it does not return anything.

Important note: methods that return `void` typically *do something*, like print a value, or delete an item from a list, or the like. They *affect state*, or are *stateful*. Sometimes but not always, methods that return a value *don't do something* (they are more like mathematical functions). The only way to be sure is to read the documentation (or the method code itself)! But a method's public API should describe what it does.

Next comes one or more method modifiers: either `abstract` or one or more of `static`, `final`, and `synchronized`. `static` methods are associated with a class, but not a particular instance of that class (in other words, not with an object). You've seen some of these in prior classes, most likely, such as `Math.min`. We'll talk more about the other modifiers later as they come up.

Finally there is at most one of `public`, `protected`, or `private`, which are member access modifiers. Which objects can invoke this method is determined by this modifier. `public` and `private` are probably familiar to you (any object and only objects of this class, respectively). `protected` and no modifier ("default access" or "package access") have special meanings we'll skip for now.

Classes in the JVM

(Note that like our discussion of the call stack, this is a simplified version to give you a mental model of how things work; the implementation in the JVM differs somewhat.)

A single copy of the class (note: the class! not an object of class!) lives in memory, annotated with its name, each method, and space for each `static` variable. There also some other stuff: a pointer to the class's superclass, and a list of interfaces it implements, and the like.

Objects that are instances of the class will be able to refer to these methods and variables here.

Instantiating objects

Objects are *defined* by a class. But an object does not exist until you *instantiate* it, that is, "make a new one" using the class as a template for the new object. For example:

```
Bus bus44 = new Bus();
```

There's a class called `Bus`, and we've instantiated an object of type `Bus`. The object is named by the variable `bus44`.

Thinking back to our first class, what's going on in memory when this happens?

Unlike *primitive types*, which are stored directly in the variable's allocated space, the variable "holding" an object isn't really the object's value. It's a *pointer* or *reference* to the place where the object really lives.

This is really important to understand, so I'll say it again: what Java stores in primitive type variables and object variables is different, conceptually: The former stores the value itself, the latter stores as its value the memory address of the object. Weird but true!

Let's do an example. What happens here?

```
String message = new String("Hi!");
String wassup = message;
```

First a new `String` is allocated. Then it's initialized on the heap. Then a new variable of type `String`, named `message`, is created. Its value is set to the *address* of the actual `String` object we created.

Note that this `String` object implicitly refers to a `String` class somewhere else. When you call methods on an object, it uses this reference to find the method – only one copy of the code for a method exists at a time – all objects of a class share it.

Next, another new variable, again of type `String`, named `wassup` is created. The *address* of `message` is looked up, and is then assigned to `wassup`. Both variables now "point to" or "refer to" the same object, which is a `String` object containing the data `"Hi!"`.

Having two variables refer to the same object is called "aliasing"; it is a common source of program bugs. Always think carefully about the *value* stored in a variable, not just the name of the variable.

So this leads to an important thing: `==` vs `.equals()`. With primitive types (`int` and so on), you only have one option: `==`. What does it do? It looks up the *values* stored in the variables, and returns `true` if and only if they are the same. `x = 3; y = 3; x==y;`

But what happens when we use `==` on variables that refer to objects? *Exactly the same thing!* Which might or might not be what we mean. Following from above, let's add `String hello = new String("Hi!");`, and ask, does `message == wassup`? Yes, because they *refer to the same object*.

Does `message == hello`? *No.* Even though the two `String` objects represent the same value ("Hi!"), they are stored in separate objects at different addresses. So the *variables* store the value of two *different addresses*.

But oftentimes we don't actually want to know if two variables point to the same objects. Instead, we want to know if the two objects' values are equivalent. In the case of `String`s. this means that they hold the same text; you can imagine that for more complicated objects, we might need a more complicated comparison of the two objects' instance variables. There is a method `.equals()` that a class can implement to provide a class-specific test of equality. (If you don't implement it, the `Object` class's `equals()` method, which defaults to `==`, is used.)

So we can write `message.equals(hello)` to check if the two objects store equivalent `String`s, rather than the two variables storing the same address as a value.

Example

(Note that in class we did this on Dogs – I love me some on-the-fly coding – but the principle is identical!)

Suppose we define a Bus class. Note we can use a Code plugin to write a semantically meaningful `equals` method for us (for now – later in the semester we'll see what all this means and how to do it ourselves, then go back to letting Code write the boilerplate).

```
public class Bus {
    private int number;

    public Bus(int n) {
        number = n;
    }

    public static void main(String[] args) {
        Bus busA = new Bus(44);
        Bus busB = new Bus(44);

        Bus busC = busA;

        System.out.println(busA == busB);
        System.out.println(busA.equals(busB));

        busC.setNumber(13);
        System.out.println(busA == busC);
        System.out.println(busA.equals(busC));

        busC = new Bus(13);
    }

    public int getNumber() {
        return number;
    }

    public void setNumber(int number) {
        this.number = number;
    }

    @Override
    public int hashCode() {
        final int prime = 31;
        int result = 1;
        result = prime * result + number;
        return result;
    }

    @Override
    public boolean equals(Object obj) {
        if (this == obj)
            return true;
        if (obj == null)
            return false;
        if (getClass() != obj.getClass())
            return false;
        Bus other = (Bus) obj;
        if (number != other.number)
            return false;
        return true;
    }
}
```

Using objects and methods

If the flow of control is executing within an object, and we access a variable, say `x`, what happens?

First the local scope is checked, inside out, for example:

```
class Bar {
  int x = 0;

  private void foo(int j) {
    for (int i = 0; i<x.length; i++) {
      x[i] = j;
    }
  }
}
```

`x` is not declared in the `for` loop, nor in the method body, nor as a parameter, so the next place that's checked is the object itself, where it's found. (The superclasses, etc.) This all happens at compile-time; you can't fail this lookup in a source that correctly compiles.

Relatedly, when methods are called on an object, the type of the object is examined, and the method is looked up in the corresponding class. If it's not found, the class's superclass is checked, and so on. Again, this is checked at compile-time. This is how the default "equals" method works; it's implemented in `Object`, which is by default the superclass of all classes that don't otherwise declare a superclass. In other words, it "inherits" the method from its superclass.

We'll do (slightly) more on inheritance, but best practices over the years have moved toward a relatively flat class hierarchy. Usually you won't see (or use) deep inheritance, with some exceptions for older codebases and large libraries (like, say, the Java Platform API).

Last modified: Thursday, February 10, 2022, 11:05 AM

◀ working-with-strings-and-lists-student.zip

Jump to...

Lecture notes 03-02 ▶

© University of Massachusetts Amherst • Site Policies • Site Contact

Moodle Help for Students • Moodle Help for Instructors

You are logged in as Tanish Patel (Log out)
COMPSCI186-SEC01 SP22

Get Help
Help for Students
Help for Instructors

COMPSCI186-SEC01 Using Data Structures Spring 2022

Lecture notes 03-02

Welcome

Announcements

Packages, namespaces, and the CLASSPATH

Namespaces

Many programming languages, including Java incorporate the idea of a "namespace". A namespace is a way to provide context to a particular name.

For a real-world analogy, you might think of a person's name, say, "John". There might be more than one in this class, so we add some context (a surname, or a student ID, or an address, or all of the above) to disambiguate which we mean.

This is very similar to the idea of a variable's scope in Java, but slightly different, as it's how we precisely name and identify classes.

For example, we all write `System.out.println()` all the time, and we all know that System is probably a class, since it starts with a capital letter. Where does it come from, though?

Packages

It's part of the `java.lang` *package*. Java organizes classes into packages; which are a hierarchical sequence of tokens (words), separated by dots. The built-in parts of the Java standard library all are part of the "java." package, though it's further subdivided.

For example, the aforementioned `java.lang` package defines the classes that are fundamental to the design of the language itself: things like `System` and `String` are defined here.

https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/lang/package-summary.html

By default, things in this namespace are automatically "imported" into the local namespace. That is, you don't have to type `java.lang.String` to declare a `String` (though you can); `String` suffices.

Interestingly, it's not against the rules to define your own `System` class. But it's like if you have both an instance variable named `x` and a local variable named `x`. By default, Java assumes you want to "more local" one.

```
int x = 5;

void test() {
  x = 3;
  System.out.println(x);
  System.out.println(this.x);
}
```

To get the "outer" one, you need to prefix it with `this.`, which tells Java you want the current instance variable with the same name.

Similarly, with a class, you need to *fully specify* the class if you want access to it. Inside your custom `System` class, if you want to access the "normal" `System.out.println`, you'll need to refer to it by its full name, `java.lang.System.out.println`.

`java` and `javax` are reserved by the JVM for built-in and extensions classes, but much like anyone can register a domain name on the Internet, anyone can declare a package namespace. There's a loose convention that you should use your reverse domain name as a prefix (for example, if our department released a package for autograding, we might put it in the `edu.umass.cs.autograder` package). But many modern java packages declare a top-level namespace – you see this in most assignments for this class, where we just define a similar top-level namespace. In practice, projects that do similar things don't usually have the same name, and/or agree to avoid namespace collisions.

Importing packages

Sometimes you want to use something not in the current namespace. Then you need to "import" it.

For example, if I want to print a random number between one and six:

```
public class Dice {
  public static void main(String[] args) {
    Random r = new Random();
    System.out.println(r.nextInt(6) + 1);
  }
}
```

it won't compile, because `Random` is not in the namespace. But I can use an import statement: `import java.util.Random;` to add it to the namespace.

Code can do this for you in one of its "quick fixes" but beware: there's sometimes more than one class with the same name! If you import the wrong one, it likely won't have the behavior you expect!

Finding packages

Where does Java look for packages? By default the JVM has access to a set of "built-in" packages, that form the Java Platform API:

https://docs.oracle.com/en/java/javase/11/docs/api/index.html

Again, mostly in the `java` and `javax` namespace, but also some others.

But where do the compiled classes, that is, the virtual machine code for them, actually live? On my machine, for Java8 a big chunk of them live in `/Library/Java/JavaVirtualMachines/jdk1.8.0_102.jdk/Contents/Home/jre/lib` in the JARs there. (The Java11 install I'm actually using has a more complicated system for loading JARs, so I'm going to show you the simpler one here.)

JARs are essentially ZIPfiles of compiled Java classes with some extra stuff (a Java-specific manifest, describing their contents, that the JVM knows how to read). Let's take a look in `rt.jar`. Hey look! Our friends `System` and `String`!

You may have noticed that the file, say `String.class` (which is the compiled representation of `String.java`) lives in a directory `java/lang/`. That looks a lot like `java.lang.`, doesn't it?

Not a coincidence! The JVM *requires* that packages *map* to (that is, *directly correspond to*) directories with the same name(s), and that classes map to `.class` files within those directories.

But there's still a piece of the puzzle missing. How does the JVM know where to look for these directories? How did it know, for example, that `/Library/Java/JavaVirtualMachines/jdk1.8.0_102.jdk/Contents/Home/jre/lib/rt.jar` was a place to search?

CLASSPATH and friends

There are three mechanisms the JVM uses. One is under your control: the "CLASSPATH". The other two you can't (easily) change – there is a "bootstrap CLASSPATH" and an "extensions directory".

The latter two are configured when your JVM is installed, and they contain classes that are part of the JRE, JDK platform, and vendor-distributed extensions.

But the CLASSPATH you do control. If you run from within code, you can add JARs to the CLASSPATH by adding them to the list of "Referenced libraries" in "Java Projects" under the explorer window; likewise, you can add directories by adding them to the "Java Source Path".

Putting your own classes into packages

If you want to put a class into a package, like, for example, you want to move our `Die` class into the `nerdy.gaming` package, you need to explicitly declare the package at the top of the file, *and* move it into an appropriate directory, in order for it to compile and be recognized by the JVM. You'll see a compiler error until you make the package and the directory strucure match.

Notice that now the top of the file has a `nerdy.gaming` package declaration; it lists just the package, not the classname (unlike `import`s, which list a full class name). Also notice that the package is rooted at a directory that's in our CLASSPATH; in this case, the `src/` directory in the project.

Another example: PA03

Sometimes this is more clear for people if I give an example based upon files they have. So let's do that now, using this week's programming assignment.

You can tell, by looking at the top of each `.java` file, that there are only two packages in this project: `list.exercises` and `string.exercises`.

Let's look at `ExtendedArrayList.java`. You can see its enclosing directories are `exercises` and `list` – in other words, part its *path* to it, on your disk's filesystem, is `list/exercises/ExtendedArrayList.java`. This is not a coincidence! Just like `.java` files have to have the same name as the class they define, the directory structure and the package structure have to "match" (with dots `.` being used in the package, and either slash or backslash in most filesystems).

So `ExtendedArrayList` (the class) is in the `list.exercises` packages, and correspondingly, `ExtendedArrayList.java` (the file) is in `list/exercises`. This is also true for `ListExercises`.

How does Java know where to look for classes and packages? It looks on the classpath. On my computer, for example, I have this project in `/Users/liberato/working-with-strings-and-lists-student/` Inside there is the project, including the `src/` directory, which contains `list`, which contains `exercises`, etc.

For this project, `/Users/liberato/working-with-strings-and-lists-student/src` is "on the classpath", that is, it's a place Java looks for packages. So it looks in there for `list/exercises/ListExercises.java` when it's trying to find the class `ListExercises` in the package `list.exercises`.

But there's more – you can have more than one directory on the classpath. For our projects, we typically have *three* directories on the classpath – `src`, `support`, and `test`. *All* directories on the classpath are searched when Java looks for classes. So the tests (in, say, `ListExercisesTest`) are in the same *package* as the things they test, even though they're in a different *directory* – because we look in `test` for packages, too, and there's a `list/exercises/` directory there, too.

Review: arrays

In our review of 121 material so far, we've exclusively used a single container type, the array. Container types are types that "hold" other types, and the array is probably the most basic: a fixed-size sequence of values (that are themselves either primitive types or references to objects) that can be read or written in approximately "constant time". We call these values "cells" or "elements" in the array.

Constant time means that no matter how big the array is, (to a first approximation) it takes the same amount of time to access (read or write) an element. We expect each of these statements to execute in about the same amount of time:

```
array[0] = 5;
System.out.println(array[1]);

array[1000000] = 12;
System.out.println(array[12345678]);
```

(modulo some caching effects, which are COMPSCI 230/335 material).

But arrays have some downsides, as well. For example, they're fixed size: you need to know how many elements you want in advance. You can cheat here by allocating a giant array, but that's wasteful for small inputs, and potentially won't work anyway if your data is of size (giant array + 1).

Instead, we might use a higher-level abstraction; that is, a more general "container type" than an array. In a bit, we'll describe the `List`:

- which is like (but not the same as) an array, and
- which can be implemented in terms of an array.
- we'll do an array-based implementation of the `List` in lecture (though we definitely won't do this for all implementations of all data structures – that's 187!)
- we'll briefly discuss but not fully implement a linked-list, an alternative to the array-based list, and talk about its pros and cons

We'll show how our `List` compares to the Java API's `List`, and this will lead into generics and container types, two topics that will come up again and again in material this semester.

Operations on arrays

To recap, what can you do with an array? You can declare it, allocate it, read or write individual elements, and determine its length at runtime.

Declare an array of `String`s called `strings`.

Allocate a `String` array of size 50 and assign it to `strings`.

Set the zeroth element of `strings` to "Hi".

Print the zeroth element of `strings`.

Print the length of `strings`.

```
String[] strings;
strings = new String[50];
strings[0] = "Hi";
System.out.println(strings[0]);
System.out.println(strings.length);
```

That's it for builtins of the array. If you want to do much else, you gotta build it yourself. (Note that there is a `java.util.Arrays` that has some helpful methods you can call on arrays, in particular the static `Arrays.toString` method is helpful when caveman debugging arrays of primitive types; otherwise the debugger can be helpful.)

So let's talk about the `List` abstract data type.

Properties of lists

Note I said "abstract data type." First we'll talk about the properties and assumptions we might expect from a `List`, in the abstract. Then we'll do an actual, concrete implementation of the data type and see how it measures up.

"List" is a very overloaded term; we'll simplify this by choosing a specific set of assumptions, that implicitly define an abstraction:

- lists are *unbounded*, that is they don't have a fixed size (if implemented with arrays, the arrays dynamically resize)
- *duplicate elements* are allowed (when searching for an element, one of several equal elements is as good as any other)
- lists can contain `null` elements (I hope you like `NullPointerException`s! though note some implementations might forbid `null` elements)
- lists support an `add` operation, either to the end of the list or to a specific place (sometimes called an "insert")
- lists support a `remove` operation, either of a specific value or of an element at a particular index
- lists support a `get` operation to return the element at a specific index
- lists support a `size` operation to determine how many elements are currently in the list
- lists can be in sorted order, but by default are not (you can imagine a `SortedList` that enforces this property)
- and many more, but we'll get to them later when we look at the full API that Java supplies.

Declare an `List` of `String`s called `strings`.

Allocate a `String List` and assign it to `strings`.

Insert the string "Hi" at the front of the list `strings`.

Append the string "Bye" at the *end* of the list `strings`.

Print the zeroth element of `strings`.

Print the length of `strings`.

```
List<String> strings;
strings = new ArrayList<>();
strings.add(0, "Hi");
strings.add("Bye");
System.out.println(strings.get(0));
System.out.println(strings.size());
```

Some things to notice:

The *type* of strings is `List<String>`. It's a variable of type `List`; we could also make it of type `ArrayList` or `LinkedList`, but what we care about is that it satisfies the `List` interface (see the javadocs).

Further, it's a *parameterized type*. Much like methods can take arguments, so can types! Usually, we see this with container classes (like Lists), where the argument (in `<>`s) is the type of thing it's holding. More on this later.

Sorta-like arrays, `List`s have an `add` operation that either appends, or inserts at a specified point – we illustrate both here. They also have a `set` operation, but it can only replace existing items, not insert new ones. And a `get`, which is much like array lookups.

Comparing lists and arrays

Lists are kinda like arrays – in that they're linear *container types* that can hold references to zero or more values of the same type.

They're kinda different, in that instead of special syntactic support (the `[]` operator), they are plain-old objects where we use their methods to "do things." And, their semantics differ somewhat: they aren't of fixed size, and we can add or delete items at arbitrary points, and anything "in the way" gets shifted out of the way.

For example, recall this question that some of you might have seen on the self-assessment:

> A sequence is said to be a Fibonaccish if each value is exactly the sum of the prior two values. For example, `{1, 2, 3, 5, 8, 13}` and `{3, 4, 7, 11, 18}` are Fibonaccish sequences, but `{1, 1, 2, 4, 8}` is not.
>
> Write a method `boolean isFibonaccish(int[] a)` that returns `true` if and only if the array `a` represents such a sequence. Assume the array `a` contains at least three values.

Your answer probably looked something like:

```
boolean isFibonaccish(int[] a) {
    for (int i = 2; i < a.length; i++) {
        if (a[i - 2] + a[i - 1]  != a[i]) {
            return false;
        }
    }
    return true;
}
```

What if, instead, we did this for a list? That is, a `List<Integer>` as the argument? What would it look like? Almost the same! But we'll need to adjust the code to access a List rather than an array.

```
boolean isFibonaccish(List<Integer> l) {
    for (int i = 2; i < l.size(); i++) {
        if (l.get(i - 2) + l.get(i - 1)  != l.get(i)) {
            return false;
        }
    }
    return true;
}
```

Normally we don't copy/paste code like this, but instead we just write it using the `List` methods.

Another example

For example, if you wanted to sum all of the numbers in a `List`, you'd write almost the same code as you would for an array, including the for-each loop:

```
int mystery (List<Integer> list) {
    int s = 0;
    for (int x : list) {
        s += x;
    }
    return s;
}
```

Why lists?

Why bother with lists at all? Because the abstraction is nicer, for many algorithms.

That is, because for many real-world problems:

- We don't know the size of the intermediate linear collection in advance. Each time we solve one of these problems, we could take the time to write a solution with arrays, and explicitly track the size, and make more space when we need to, but that's lots of busywork.
- Or, we can compute the size in advance, but doing so is costly.
- Or, we need to be able to move elements around easily. Much like the first point, we could explicitly move them around as needed, but it's a lot nicer just to say `theList.add(0, "the new front of the list!") to add an element to the front of the list.
- and so on.

As you're seeing in A03, lists, like `String`s, have many additional useful methods to use. We'll be seeing more of these methods (part of the "Collections API" in Java) as the semester progresses.

Last modified: Thursday, February 10, 2022, 4:35 PM

◀ Lecture notes 03-01

Jump to...

Programming assignment 04: DNA sequence assembly ▶

© University of Massachusetts Amherst • Site Policies • Site Contact

Moodle Help for Students • Moodle Help for Instructors

You are logged in as Tanish Patel (Log out)
COMPSCI186-SEC01 SP22

Get Help
Help for Students
Help for Instructors

COMPSCI186-SEC01 Using Data Structures Spring 2022

Lecture notes 04-01

Welcome

Announcements

Check your grades in Moodle please! That's the authoritative source I will be using to determine your grade in this class.

Reminder on `equals`

Writing a "proper" equals method is kind of a pain in the ass. First you have to check if the other object is non-null. Optionally you check if it's `==` to the current object. Then you check if it's of the same type as the current object. Then you compare each relevant field of the other object to the current one, using `==` or `.equals` as appropriate.

A04 requires that your code be able to compare fragments for semantic equality. You, the programmer, must decide if some "other" object is equal to the current `Fragment` – the "hint" here is that you must do exactly as above, and compare the relevant instance variable. There's only really going to be one – a `String` – so use `.equals` on that one. Ask for help on Campuswire or in office hour if it's not clear how to do it.

Aliasing and references in Lists (and other container types)

You probably noticed that this week's assignment forced you to make a copy of a list that's passed into a constructor. Some of you did it manually:

```
public class Assembler {
    private final List<Fragment> fragments;

    public Assembler(List<Fragment> fragments) {
        this.fragments = new ArrayList<Fragment>();
        for (Fragment f: fragments) {
            this.fragments.add(f);
        }
    }
```

But you can also do it in one line, using the `ArrayList` copy constructor, which copies one collection into another when that collection is passed as an argument:

```
        this.fragments = new ArrayList<Fragment>(fragments);
```

But it does leave the question: Why do I make you do this? Because it's a good idea! But why?

The real answer here is that if you don't, then the list you've "passed into" the `Assembler` is an alias of the list *stored* in the `Assembler`'s instance variable. So changing one changes the other, even though it "looks like" they are different lists!

This can be a huge source of bugs for programmers who haven't internalized that different *references can refer to the same thing*. When this bug bites, it's not usually when you use two references in the same scope, like we did in the `Bus` question on the problem set a week or so ago. It's because you made a copy of a reference and used both as though they were separate, independent things. But they're not! They're the same object! And that leads to problems if you don't realize they are the same object with different names in different scopes.

A `ListInterface`

How does `List` actually work? Like, what's going on behind the scenes? We're going to dive into it a little bit here in 186 today and next week, so that you have an intuition for how a `List` is implemented. If you choose to go to 187, you'll do a more thorough treatement of lists and several other, more complicated data structures.

OK, here we go!

As much as possible, it's a good idea to have the computer help us check our assumptions as we go. One way to do this is to leverage the type system. We'll declare a `StringList` for a list of `String`s as described above, then we'll implement it using arrays.

Now, normally, you never do this! You use the built-in `List` interface and `ArrayList` implementation. But we're going to do it here to peel back the curtain a little and show you that there's no magic, just straightforward application of computer science you already know. You can do this. (And you will do this, in 187!)

```
public interface StringList {
    public void add(String s);
    public void add(int i, String s);
    public String get(int i);
    public String remove(int i);
    public int size();
}
```

What happens if a user of this interface does something "bad," like attempting to remove an element that's not present, or to add an element past the end of the list? The type doesn't help us. We can add (optional) unchecked exception declarations to hint at this:

```
public interface StringList {
    public void add(String s);
    public void add(int i, String s) throws IndexOutOfBoundsException;
    public String get(int i) throws IndexOutOfBoundsException;
    public String remove(int i) throws IndexOutOfBoundsException;
    public int size();
}
```

..though note that these just give a programmer using your `StringList` a heads-up. The documentation comments for the class and the method define the actual contract that the class (or interface) offers. If there are no documentation comments, then only the type signature gives the contract. Usually it's not enough.

Example

```
    public int size() {
        return 12;
    }
```

Strictly speaking, does this this method obey its contract?

```
    /**
     * @return the number of elements stored in this list
     */
    public int size() {
        return numberOfElements;
    }
```

What about now?

Note that the compiler can't enforce all parts of this contract; it's up to the *programmer* implementing the `StringList` to do the right thing.

Also note that a full `StringList` would have many more methods (`set`, analogous to array assignment, `removeAll`, `equals`, and so on). I don't want to write all of them in lecture! (Though it should be straightforward to write most of them.)

Writing the `StringArrayList`

Let's think about what we need in our implementation. What instance variables do we need? Certainly an array of `String`s to hold the list elements. Anything else? The number of elements actually stored in the array. Remember, one of the reasons we're writing a `List` is that arrays are of fixed size, but a `List` can grow and shrink arbitrarily. (We'll see how soon.)

So let's declare an array `String[] array` and an `int size` to hold these values.

```
public class StringArrayList implements StringList {
    String[] array;
    int size;
}
```

(On board) Conceptually, strings will be added to, gotten, and removed from this array; it's the implementation's job to make sure they go in the right place, and that if a user of `StringArrayList` tries to access an element of the List (that is, of the underlying array) that is invalid, an exception is raised.

Let's start with a constructor that starts with a small empty array:

```
public StringArrayList() {
  array = new String[10];
  size = 0;
}
```

Now let's do the simple methods:

```
public int size() {
  return size;
}



@Override
public String get(int i) throws IndexOutOfBoundsException {
  return array[i];
}
```

But remember, while the *array* might be of size 10, there might be fewer than 10 (even no!) strings stored in the array. So a correct definition would instead read:

```
public String get(int i) throws IndexOutOfBoundsException {
  if (i >= size || i < 0) {
    throw new IndexOutOfBoundsException();
  }

  return array[i];
}
```

This is important to understand: the `List` acts like a list of elements with a size equal to the number that have been added (less the number removed). Even though there's an underlying array of a different size, *the user of the `List` interface cannot see it!* The details are said to be *encapsulated*. This is a very powerful concept that lets you use data structures (and generally any API) by reading their contract – you don't need to fully understand every detail of the implementation (though it can be helpful to do so!).

Now let's turn to some of the more complicated methods, like `add`:

```
public void add(String s) {
  array[size] = s;
  size++;
}
```

Making space

This sorta works, but what happens once we add the eleventh element? We'll overflow the array bounds, which we don't want to do – our list is supposed to be unbounded. Instead, we'll check to see if the array needs to be expanded, and do so:

```
public void add(String s) {
  if (size == array.length) {
    enlarge();
  }
  array[size] = s;
  size++;
}
```

What should `enlarge` do? It should allocate a new, larger array, copy the current array into it, then set the `strings` instance variable to point to this new array.

```
void enlarge() {
  String[] larger = new String[array.length * 2];
  for (int i = 0; i < array.length; i++) {
    larger[i] = array[i];
  }
  array = larger;
}
```

Why double, and not, say, just `+ 10`? The full answer is beyond the scope of this course, but in short: when you don't know anything else, doubling is the most efficient way to dynamically grow an array. If you do know other things, you might expose ways to grow (or shrink) the underlying array, but that's has its own problems (like: now users of your code are tied to your specific implementation, even if a better one comes along later).

Adding somewhere else

What about if we want to add in a particular place, rather than just at the end of the array? We need to move each element out of the way.

(On board) we have to move the last element forward one, then the previous element into the last element's place, and so on, to "make space" for the item we're inserting. We also need to make sure the index is valid, and that there's space.

In code:

```
public void add(int i, String s) throws IndexOutOfBoundsException {
  if (i >= size || i < 0) {
    throw new IndexOutOfBoundsException();
  }

  if (size == array.length) {
    enlarge();
  }
  for (int j = size; j > i; j--) {
    array[j] = array[j-1];
  }
  array[i] = s;
  size++;
}
```

Notice that we never worry about "checking" what's already in an array cell. We know that all cells at indexes less than `size` are in use; and all cells at indexes greater than or equal to `size` are not.

Finally, let's write the code to `remove` an element at index `i`. Similar to the above, we'll need to "move" any elements into the space we leave behind (on board). And by convention, return the value we removed.

```
public String remove(int i) throws IndexOutOfBoundsException {
  final String removed = array[i];
  if (i >= size || i < 0) {
    throw new IndexOutOfBoundsException();
  }

  for (int j = i; j < size - 1 ; j++) {
    array[j] = array[j+1];
  }

  // optional
  array[size-1] = null;

  size--;
  return removed;
}
```

Setting the unused space to null lets the garbage collector free the memory, but that reference will also be overwritten the next time we add to the list, so it's not strictly necessary.

Testing the implementation

Let's write a short, silly program demonstrating how our current code works. Let's write a method to print out the contents of a `StringList`:

```
public static void print(StringList l) {
  for (int i = 0; i < l.size(); i++) {
      System.out.print(l.get(i) + " ");
  }
  System.out.println();
}
```

Notice this is in terms of `StringList`: it knows nothing about the `ArrayStringList` and only depends upon methods in the interface.

Next, a silly test method:

```
public static void main(String[] args) {
    StringList l = new StringArrayList();

    l.add("Marc");
    l.add("not");
    l.add("so-good.");
    print(l);

    l.add(1, "is");
    print(l);

    l.remove(2);
    print(l);
}
```

OK, so our `StringList` works. What about if we build it a different way?

Last modified: Tuesday, February 15, 2022, 5:03 PM

◀ Sample quiz answers

Jump to...

Lecture notes 04-02 ▶

© University of Massachusetts Amherst • Site Policies • Site Contact

Moodle Help for Students • Moodle Help for Instructors

You are logged in as Tanish Patel (Log out)
COMPSCI186-SEC01 SP22

Get Help
Help for Students
Help for Instructors

COMPSCI186-SEC01 Using Data Structures Spring 2022

Lecture notes 04-02

Welcome

Announcements

So proud you're all working hard!

So glad you're finding people to work with!

Lab next week on Tuesday! (Tuesday is a Monday!), regular lecture / class Thursday.

Next – Linked lists!

Next, we're going to spend a little more time on an alternate implementation for `StringList`. This will be especially useful for those of you intending to go on to 187, but it's important regardless, since in some sense seeing-is-believing that you can have two very different ways of doing something that obey the same contract (but that might differ in, say, time or space efficiency). And it will give some context to the box-and-arrow diagrams that we'll be using later in the class to explain why various data structures work the way they do.

Lions, tigers, bears, and linked lists

Remember, we built our `ArrayStringList` using an array as our underlying container. We hid this detail from users of the class, so it's entirely possible we could have used a different way to build the list. Java's builtins (that is, language constructs not library classes or methods) that let you "connect" or collect multiple variables are limited to arrays and references, but it turns out — not coincidentally — that these are all you need to build basically every other data structure. We (definitely!) won't do them all, but we will do one more today: the *linked-list*. You'll also benefit from this when we talk about how some other data structures are built later (though again, not in this level of detail, except for very simple structures).

Arrays are a fixed-size container of a sequence of cells; linked lists circumvent this restriction by making the cells (or "nodes" as they're called in linked lists) explicit objects rather than an implicit property of an array. Here's a simple container for `String`s:

```
public class StringNode {
    public String data;
}
```

(On board). A simple object to hold a `String` (which we've made public – you might choose getters/setters in practice). So far so good, except there's no concept here of more than one `StringNode` being part of the same group of `StringNode`s. What do we do to fix this? Add a reference to a `next StringNode` to the current one!

```
public class StringNode {
    public StringNode next;
    public String data;
}
```

(Again on board) you can see if we want to add a node to the end of the list, it's pretty straightforward. Inserting into the middle (or removing) is a little more complicated, but we'll get there in a bit.

Let's use this `StringNode` to build an alternate implementation of `StringList`.

Building `StringLinkedList`

We'll need to keep a reference to the underlying list of nodes as an instance variable, and it turns out that's the only instance variable we need. By convention, this is called `head`, and it's initialized to null (an empty list) when we start:

```
private StringNode head;

public StringLinkedList() {
    head = null;
}
```

Let's add the "simple" methods from last lecture, starting with `size`. What do we need to do? There's no underlying array, so we can't just examine the `length` attribute and return it. Instead, we need to *traverse* the list. In other words, (on board) we will start at the head element (if it exists), and follow the `next` references until we reach the end, signaled by a `null` value. Don't forget that `head` is null, so your code should account for this:

```
public int size() {
    int size = 0;
    StringNode current = head;
    while (current != null) {
            size++;
            current = current.next;
    }
    return size;
}
```

Unlike arrays, where we can just jump to the node we want, a linked-list (almost) always requires that we traverse its elements until we get to the one we want.

Note that we don't *have* to traverse the whole list for `size`; we could maintain a private `size` instance variable and just return it instead if we wanted to (and then, just like in `StringArrayList`, we need to remember to updated whenever adding or removing.) Let's do so instead of the above:

```
    int size;
    StringNode head;

    public StringLinkedList() {
        size = 0;
        head = null;
    }

    @Override
    public int size() {
        return size;
    }
```

Now let's do `get`. The exceptions are the same as before, but now we must *traverse* the list to find the `i`th element:

```
public String get(int index) throws IndexOutOfBoundsException {
    if (index < 0 || index >= size) {
        throw new IndexOutOfBoundsException();
    }

    StringNode current = head;
    int i = 0;

    while (i != index) {
        // move to the next node
        current = current.next;

        // increment i
        i += 1;
    }

    return current.data;
}
```

More on StringLinkedList

Now let's look at `add`. If we want to add at the end of the list we can again traverse it to get to the end, and adjust the last element's `next` reference appropriately. Note that we have to stop our traversal *just before* we get to the end, not after we go past it, which also means we have to special-case the `head` (on board first):

```
public void add(String s) {
    StringNode n = new StringNode();
    n.data = s;

    // case 1: list is empty
    if (head == null) {
        head = n;
        size = 1;
        return;
    }

    // case 2: the list is not empty

    // find the last node in the list
    StringNode current = head;
    while (current.next != null) {
        current = current.next;
    }

    // attach a new node to the end
    current.next = n;
    size += 1;

}
```

Similarly, if we want to add somewhere in the middle of the list, we have to stop *just before* we get there, and do surgery on both the previous and current (added) item to make the links in the list line up, again with a special case for the first spot:

```
public void add(int index, String s) throws IndexOutOfBoundsException {
    if (index < 0 || index >= size) {
        throw new IndexOutOfBoundsException();
    }

    // case 1: insert at front of list
    if (index == 0) {
        // step 0: create the n
        StringNode n = new StringNode();
        n.data = s;

        // 1: point n's next pointer to current head
        n.next = head;

        // 2: point head to the new node
        head = n;

        size += 1;
        return;
    }



    // case 2: insert somewhere else

    // 0: create a new node n
    StringNode n = new StringNode();
    n.data = s;

    // 1: find the node before the node we want to insert
    StringNode current = head;
    int i = 0;
    while (i != index - 1) {
        current = current.next;
        i += 1;
    }

    // 2: update n's next pointer to point to the node after
    n.next = current.next;

    // 3: update the node before, to point to n
    current.next = n;

    size += 1;
}
```

Finally the `remove` method, which again has to do some surgery on the previous node (if it exists). Let's do some examples on the board first (end of list; front of list; middle of list)

Then here's the code for remove():

```
public String remove(int index) throws IndexOutOfBoundsException {
    if (index < 0 || index >= size) {
        throw new IndexOutOfBoundsException();
    }
    String result;

    // case 1: head
    if (index == 0) {
        result = head.data;
        head = head.next;
    }

    else { // not the head
        // find the node before the node we want to delete
        int i = 0;
        StringNode current = head;
        while (i != index - 1) {
            current = current.next;
            i += 1;
        }

        // store the value to be returned
        result = current.next.data;

        // point current to its next next node
        current.next = current.next.next;
    }
    size -= 1;
    return result;
}
```

And we'll run it in our toy program, switching `StringArrayList` to `StringLinkedList`. Notice the behavior doesn't change. (`StringList sli = new StringLinkedList();` is the only change.)

```
public class ListDriver {
    public static void main(String[] args) {
        StringList l = new StringLinkedList();

        l.add("Marc");
        l.add("appears");
        l.add("to");
        l.add("know");
        l.add("something.");

        l.remove(1);
        l.remove(1);
        l.add(2, "little");

        for (int i = 0; i < l.size(); i++) {
            System.out.println(l.get(i));
        }
    }
}
```

Implications

Now we've seen two different ways to implement a simple abstract data type (the `List`), using arrays, and using references. *All* other abstract data types can be implemented in terms of one or both of these mechanisms.

We've also seen that both implementations have the same results. Though, if you think about it, one or the other is more efficient (or at least different) in some ways.

For example, the array-based list uses up to twice the memory of its current size(). The linked list uses only what it needs, plus space for the link. It actually turns out they're about equivalent.

Array indexing (and assignment) are fast; about as fast as following a single link in a linked list. This means some things are going to be faster than others.

For example, Would you expect a call to `get(someLargeIndex)` in `ArrayList` or `LinkedList` to be faster?

For random access, the array-based implementation is better, since the `get` method is really a thin wrapper around array indexing, which is quite fast.

For addition or removal, it kind of depends. Adding something to the head of the linked list is really fast (make a node, set its next to head, set head to it), whereas adding a node to the front of an array-based list requires moving every single element in the array, and possibly enlarging the array, too.

On the other hand, adding an element to the end of the linked list is slow, since you have to traverse the entire list first. You could also keep a reference to the end of the list (called a `tail` pointer), but then you also have to update it and handle it in every method that modifies the list. On the third hand, only the implementor of the linked list needs to do this, not the user – as in our demo, both have the same interface, and neither expose their inner workings in terms of results (though runtime and memory usage might differ).

We won't go into this comprehensive level of detail for (most) data structure implementation again in this course, though we will definitely revisit the idea of using an array or references to build more complicated data structures. But that's about as far as we'll go, with diagrams rather than code. We'll be more interested in the *interface* of the abstract data types and what *behaviors* they provide, rather than the fine details of the implementations – that comes in 187 and later classes.

Last modified: Friday, February 18, 2022, 2:05 PM

◀ Lecture notes 04-01

Jump to...

Programming assignment 05: Scheduler ▶

© University of Massachusetts Amherst • Site Policies • Site Contact

Moodle Help for Students • Moodle Help for Instructors

You are logged in as Tanish Patel (Log out)
COMPSCI186-SEC01 SP22

Get Help
Help for Students
Help for Instructors

COMPSCI186-SEC01 Using Data Structures Spring 2022

Lecture notes 05-01

Generics

Motivation

Methods take parameters: rather than hardcoding all data and values into a method, we can make some parts of the data variable and parameterizable so that the method can be reused with different data and values. This make a lot of sense: Let's say we want to write a method that adds five to its argument:

```
int add5(int i) {
    return i + 5;
}
```

That's useful. But someday we want to also write a method that adds six:

```
int add6(int i) {
    return i + 6;
}
```

OK. Then `add7`, etc. Getting silly. We don't want to write a different method each time, since since (1) there's an infinite number of them! and (2) the operation of adding is mechanically the same each time. That is, there's a generalized algorithm we write, once, and then can use many times.

```
int add(int i, int j) {
    return i + j;
}
```

Example

The insight behind "generics" is that we can do the same thing with *types* – we can parameterize classes (and methods) with a type, too, and use it on different types of things. Our crappy `StringList`, for example, while it lets us hold any list of `String`s we want, was still limited to `String` data. But it turns out that instead of writing:

```
public interface StringList {
    public void add(String s);

    public void add(int index, String s) throws IndexOutOfBoundsException;

    public String get(int index) throws IndexOutOfBoundsException;

    public String remove(int index) throws IndexOutOfBoundsException;

    public int size();
}
```

you can *parameterize* the interface on a type using angle brackets:

```
public interface List<E> {
    public void add(E e);

    public void add(int index, E e) throws IndexOutOfBoundsException;

    public E get(int index) throws IndexOutOfBoundsException;

    public E remove(int index) throws IndexOutOfBoundsException;

    public int size();
}
```

The `List` now sports a *generic* type name in angle brackets. We've defined a family of possible types here; note that each method that used to operate on strings now operates on this mysterious `E`.

We can also type parameterize a class:

```
public class Node<E> {
    public Node<E> next;
    public E data;
}
```

...and the two together let us write *generic* code, that operates on *generic* types, based upon the type parameter.

Here's the fully converted `Node` and `LinkedList`:

```
public class StringLinkedList implements StringList {
    private int size;
    StringNode head;

    public StringLinkedList() {
        size = 0;
        head = null;
    }

    @Override
    public void add(String s) {
        StringNode n = new StringNode();
        n.data = s;

        // case 1: list is empty
        if (head == null) {
            head = n;
            size = 1;
            return;
        }

        // case 2: the list is not empty

        // find the last node in the list
        StringNode current = head;
        while (current.next != null) {
            current = current.next;
        }

        // attach a new node to the end
        current.next = n;
        size += 1;

    }

    @Override
    public void add(int index, String s) throws IndexOutOfBoundsException {
        if (index < 0 || index >= size) {
            throw new IndexOutOfBoundsException();
        }

        // case 1: insert at front of list
        if (index == 0) {
            // step 0: create the n
            StringNode n = new StringNode();
            n.data = s;

            // 1: point n's next pointer to current head
            n.next = head;

            // 2: point head to the new node
            head = n;

            size += 1;
            return;
        }



        // case 2: insert somewhere else

        // 0: create a new node n
        StringNode n = new StringNode();
        n.data = s;

        // 1: find the node before the node we want to insert
```

```
        StringNode current = head;
        int i = 0;
        while (i != index - 1) {
            current = current.next;
            i += 1;
        }

        // 2: update n's next pointer to point to the node after
        n.next = current.next;

        // 3: update the node before, to point to n
        current.next = n;

        size += 1;
    }

    @Override
    public String get(int index) throws IndexOutOfBoundsException {
        if (index < 0 || index >= size) {
            throw new IndexOutOfBoundsException();
        }

        StringNode current = head;
        int i = 0;

        while (i != index) {
            // move to the next node
            current = current.next;

            // increment i
            i += 1;
        }

        return current.data;
    }

    @Override
    public String remove(int index) throws IndexOutOfBoundsException {
        if (index < 0 || index >= size) {
            throw new IndexOutOfBoundsException();
        }
        String result;

        // case 1: head
        if (index == 0) {
            result = head.data;
            head = head.next;
        }

        else { // not the head
            // find the node before the node we want to delete
            int i = 0;
            StringNode current = head;
            while (i != index - 1) {
                current = current.next;
                i += 1;
            }

            // store the value to be returned
            result = current.next.data;

            // point current to its next next node
            current.next = current.next.next;
        }
        size -= 1;
        return result;
```

```
    }

    @Override
    public int size() {
        return size;
    }

}
```

And the `ArrayList`:

```java
public class ArrayList<E> implements List<E> {
    E[] array;
    int size;


    public ArrayList() {
        size = 0;
        array = (E[])new Object[10];
    }

    private void enlarge() {
        // create a new array
        E[] newArray = (E[])new Object[array.length * 2];

        // copy elements from old into new
        for (int i = 0; i < array.length; i++) {
            newArray[i] = array[i];
        }

        // update instance variable to point to the new array
        array = newArray;
    }

    @Override
    public void add(E s) {
        if (size == array.length) {
            enlarge();
        }

        array[size] = s;
        size += 1;
    }

    @Override
    public void add(int index, E s) throws IndexOutOfBoundsException {
        if (index < 0 || index >= size) {
            throw new IndexOutOfBoundsException();
        }

        if (size == array.length) {
            enlarge();
        }

        // shift things out of the way
        for (int i = size; i > index; i--) {
            array[i] = array[i - 1];
        }

        // insert the new element at the right index
        array[index] = s;
        size += 1;

    }

    @Override
    public E get(int index) throws IndexOutOfBoundsException {
        if (index < 0 || index >= size) {
            throw new IndexOutOfBoundsException();
        }

        return array[index];
    }

    @Override
    public E remove(int index) throws IndexOutOfBoundsException {
        if (index < 0 || index >= size) {
```

```
            throw new IndexOutOfBoundsException();
        }
        // keep a reference to the element
        E r = array[index];

        // shift things up to replace element at index
        for (int i = index; i < size; i++) {
            array[i] = array[i + 1];
        }

        // decrease size of the list
        size -= 1;

        // return the element
        return r;
    }

    @Override
    public int size() {
        return size;
    }

}
```

Note the shenanigans around generic arrays – we'll talk more about this later, but you don't need to worry too much about it right now.

Type parameters

The `E` is a type parameter – it says that the programmer who declares a variable of type `ListInterface` must also choose a particular type that the declared `ListInterface` will handle. `ListInterface`s of different type parameters are of different types. For example, you cannot assign one to another unless they have the same type parameter, any more than you can assign a `boolean` to a `String`:

```
boolean x = "banana"; // not allowed, fails at compile time

List<String> x;
List<Integer> y;

... // some code ...

x = y; // not allowed, fails at compile time
```

More on parameters

Type parameters are usually written as a single uppercase letter, and often that letter is an abbreviation. `E` stands for `E`lement of a collection; we'll also see `K`ey and `V`alue later in the course.

Type parameters, when instantiated (that is, when a generic is declared), must be a non-primitive type. But, Java does something called auto-boxing, so you can generally mix primitives and non-primitives freely using the associated *wrapper* types, like `Integer`. (`Integer` and friends also have many useful static methods.)

Another fun fact for today about type parameters: Usually we think of them as being declared on classes (and indeed, that's usually where they are declared). But if you write a particular method that would benefit from type parameterization, you can do so:

```
public class Pair<K, V> {
    public final K k;
    public final V v;

    public Pair(K k, V v) {
        this.k = k;
        this.v = v;
    }
}

public class Util<K, V> {

    public static <K, V> boolean compare(Pair<K, V> p1, Pair<K, V> p2) {
        return (p1.k.equals(p2.k) && p1.v.equals(p2.v));
    }

}
```

Note the type parameters come immediately *before* the return type. This is important, as `static` methods that operate on arguments (or have a return type) that's generic come up all the time, and you need to be able to tell Java what the generic type(s) in the method are. They're *independent* of the generic types, if any, that the class declares. For example, if the above class definition included `public class Util<K>`, the `K` in the `compare` method is *not* necessarily the same as the `K` in the method's declaration.

Why generics matter

They matter for the reasons listed above (generic re-usable code)! But also, in Java 5, the entire Collections library was re-written to use generics. Before then, all container types (`List`, etc.) only held things of type `Object`, and you, the programmer, had to laboriously cast them each time you used them:

```
List list = new ArrayList();
list.add("hello");
String s = (String) list.get(0);
```

Not only was this a pain, if you made a mistake:

```
        List l = new ArrayList();
        l.add("Zero");

        String s = (String)l.get(0);

        //...

        Integer i = (Integer)l.get(0); // throws exception at run-time!
        System.out.print(i);
```

you'd find out at run-time, not at compile-time. And while I know you hate compiler errors right now, you'll learn to love them when writing big programs — every error that the compiler catches is one you can fix at your leisure, while run-time errors are erratic, not always reproducible, and generally result in a (much bigger) headache for you.

Lists and sorting

Suppose you want to add items to a list (say, a `stores` attribute of type `List<Integer>` a store management system) and you want to keep it sorted. An aside: "sorted" is a funny word here. We really mean "in ascending (or descending) order", but it's a historical bit of vocabulary to say "sorted" that's stuck around.

How would you write the `public void addStore(int store)` method?

```
public void addStore(int store) {
    if (stores.isEmpty()) {
        stores.add(store);
        return;
    }
    for (int i = 0; i < stores.size(); i++) {
        if (stores.get(i).compareTo(store) >= 0) {
            stores.add(i, store);
            return;
        }
    }
    stores.add(store);
}
```

Automatic sorting

Here's an alternative?

```
stores.sort(null);
```

What's up with the `sort(null)` call? Let's look at the API: https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/util/List.html#sort(java.util.Comparator)-

The `Comparator` type parameter is an interface that describes how to compare two arbitrary objects (see also its simpler predecessor, the `Comparable` interface). You might implement it if you wanted to do something unusual with the sort, for example, put all odd numbers before all even numbers (which maybe sounds nonsensical, but think about mail delivery up and down each side of the street).

But you don't actually have to implement its abstract methods to use it here. Reading the documentation:

> If the specified comparator is null then all elements in this list must implement the `Comparable` interface and the elements' natural ordering should be used.

Do `Integer`s implement `Comparable`? Let's check: https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/lang/Integer.html

Yup. In general, you should start doing your best to read and follow links in the Java API if you see things in code you don't understand. You might not understand everything you read, but only by trying are you going to learn, and at some point you can't expect your instructors to spoon-feed you everything (though I will definitely cover the highlights).

So anyway, we can exercise option 2:

Option 2: Append the number then sort the list!

```
public void addStore(int store) {
    stores.add(store);
    stores.sort(null);
}
```

This may not be as *efficient* as inserting into the correct spot – we'll learn more about this later in the semester – but it's much easier *conceptually* and to write.

A worked example for `Comparator`

So we've been talking about `List`s and generics and reviewing 121 stuff like boolean conditions and flow control. Let's do another worked example.

We're going to design and write a class to represent simplified postal addresses (street numbers and names). Then we're going to dump several instances of it into a list. Finally, we're going to define a custom `Comparator` on a postal address that will let us sort the list in a special way.

Let's get going.

Given the problem statement, we know we're going to want to define a class that defines objects containing a street number and name:

```
public class PostalAddress {
    public final int number;
    public final String name;
}
```

What is up with Marc and his use of `public final`? Here's what's up: when you know a data type is going to remain fixed, and that any particular object isn't going to have its data change, there's no reason to deal with the pain of writing private instance variables and then turning around to write public accessors (get methods). "But Marc, what if you change how the value is stored, or how it is determined?" Then you'll be giving a different name (or writing an getter, etc.) and your IDE will flag all occurrences for you to fix. Or better yet, will allow you to *refactor* them all yourself.

(See "Effective Java, 2nd edition" which though over a decade old is still one of the best bits of reading you can do once you're an intermediate Java programmer.)

First steps

Anyway, now let's add a few constructors. First the obvious one:

```
public PostalAddress(int number, String name) {
    this.number = number;
    this.name = name;
}
```

Maybe one that does some simple parsing if we have time:

```
public PostalAddress(String textAddress) {
    String[] matches = textAddress.split("\\s+", 2);
    this.number = Integer.parseInt(matches[0]);
    this.name = matches[1];
}
```

OK! Now we've got a simple class. Let's try making a few of them and adding them to a list:

```
List<PostalAddress> addresses = new ArrayList<PostalAddress>();

for (int i = 1; i <= 10; i++) {
    addresses.add(new PostalAddress(i, "Maple St"));
}
```

and maybe printing them out:

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

Ugh, what's this `PostalAddress@677327b6` nonsense? Remember, when you print an object, Java tries to coerce it to a `String` using its `toString` method. We haven't written one, so we get the `Object` default, which is what you see. It is derived from the class name and the `hashCode()` method. Since we haven't defined the latter, we get its default, which is usually but not always the object's memory address. Yuck. Let's make it better:

```
public String toString() {
    return name + ", " + number;
}
```

Now it's a little better. Hey, let's see if 6 Maple Street is in our list:

```
System.out.println(addresses.contains(new PostalAddress("6 Maple St")));
```

What will this print?

`false`? What? Maybe our constructor is broken, let's try the other one:

```
System.out.println(addresses.contains(new PostalAddress(6, "Maple St")));
```

Nope, still false. Why? Let's look at the `List.contains` method javadoc. It uses `equals()`. Ahh, but we haven't written an `equals`, which means we're using the default one from Object. ...which checks `hashCode`, which will be different for different instances of the object. (Look it up in the java doc.) Let's fix this problem?

```
public boolean equals(PostalAddress o) {
    return number == o.number && name.equals(o.name);
}
```

But it still doesn't work? Note that `contains` (and `equals`) work on `Object`s, not `PostalAddress`es. So we need to change the signature. And then check the type:

```
public boolean equals(Object o) {
    if (!(o instanceof PostalAddress)) return false;
    PostalAddress p = (PostalAddress)o;
    return number == p.number && name.equals(p.name);
}
```

Now we'll implement `Comparable` and impose a natural ordering on `PostalAddress` that sorts only by number.

```
public class PostalAddress implements Comparable<PostalAddress>{
```

requires we add a `compareTo` method, so here it is:

```
public int compareTo(PostalAddress p) {
    if (this.number < p.number) {
        return -1;
    } else if (this.number == p.number) {
        return 0;
    } else if (this.number > p.number) {
        return 1;
    }

    // wont get here
    return 0;
}
```

We can control the "natural" order by changing this method, or we can develop a new order in a separate `Comparator` class. We'll see this next week!

Last modified: Friday, February 25, 2022, 9:42 AM

◀ scheduler-student.zip

Jump to...

Sample quiz questions ▶

© University of Massachusetts Amherst • Site Policies • Site Contact

Moodle Help for Students • Moodle Help for Instructors

You are logged in as Tanish Patel (Log out)
COMPSCI186-SEC01 SP22

Get Help
Help for Students
Help for Instructors

COMPSCI186-SEC01 Using Data Structures Spring 2022

Lecture notes 06-01

Review: Comparable and natural order

Last class we talked about implementing `Comparable` on a class. When you do so, you impose a "natural ordering" on objects of that class – given any two, you can decide which is greater than the other. You implement this in the `compareTo` method, which has semantics:

```
this    other    returns
      <            -1 (or any negative integer)
     ==             0
      >             1 (or any positive integer)
```

Most of the wrappers (like `Integer`) around primitive types (like `int`) implement the expected natural ordering.

Postal order

Now let's define a custom comparator for use in doing a "postal sort". That is, we still want to sort such that street names are alphabetical, but we want the numbers sorted as all odd first (in ascending order), then all even (in descending order). This is how the truck might go up and down the street (on board).

What does that look like? Let's declare a new `Comparator`:

```
public class PostalOrderComparator implements Comparator<PostalAddress> { ... }
```

Again, Code can helpfully fill it out with the method we need to implement, so let's do it.

It will be similar to but more complicated than the `compareTo` method we just wrote. A tip: `x % 2 == 0` if and only if `x` is even. `x % 2 == 1` iff it's false.

```
public int compare(PostalAddress p1, PostalAddress p2) {
    // first sort by street name
    // if they're different
    if (!p1.name.equals(p2.name)) {
        return p1.name.compareTo(p2.name);
    }

    // if they're the same, then sort by street number
    // odds before evens

    // first one odd, second one even, first one comes first
    if (p1.number % 2 == 1 && p2.number % 2 == 0) {
        return -1;
    }

    // first one is even, second one is odd, then second comes first
    if (p1.number % 2 == 0 && p2.number % 2 == 1) {
        return 1;
    }

    // otherwise, they're both even or both odd
    // if they're both odd
    if (p1.number % 2 == 1) {
        return Integer.compare(p1.number, p2.number);
    }

    // otherwise they're both even
    return -Integer.compare(p1.number, p2.number);

    // could also write
//        return Integer.compare(p2.number, p1.number);
}
```

And let's check it out:

```
    public static void main(String args[]) {
        List<PostalAddress> l = new ArrayList<>();

        l.add(new PostalAddress(10, "Main"));
        l.add(new PostalAddress(1, "Main"));
        l.add(new PostalAddress(13, "Main"));
        l.add(new PostalAddress(4, "Main"));
        l.add(new PostalAddress(50, "Main"));
        l.add(new PostalAddress(12, "Main"));

        l.add(new PostalAddress(101, "South"));
        l.add(new PostalAddress(12, "South"));
        l.add(new PostalAddress(13, "South"));
        l.add(new PostalAddress(42, "South"));
        l.add(new PostalAddress(5, "South"));
        l.add(new PostalAddress(1, "South"));

        l.sort(new PostalOrderComparator());
        System.out.println(l);
```

Sets: an introduction

Next, we're going to move on in our list of top-n abstract data types from the `List` to the `Set`. In order to give you some grounding in what a set is and how it differs from a list, we're going to turn to an arcane and little-known subject: Mathematics. To be clear, we're going to do a very gentle introduction to "simple" set theory; if you have already taken a discrete math course this will be review for you, and if you stay in CS, you'll see this again in a lot more detail in COMPSCI 250.

Simply put: a set is a collection of distinct objects.

The objects can be anything: people, numbers, shapes, colors, (or perhaps most topically, instances of Java objects). These objects are generally referred to as members or elements of a set.

In set theory, sets are named by a single uppercase letter: A or B, for example.

For our purposes, we'll usually write sets as a list of the elements. The list will be comma separated, and will be enclosed in curly braces. For example: A = {1, 3, -6} describes a set called "A" that has three integers as elements.

Sets contain a collection of *unique* items. That is, sets cannot have duplicate items.

While the items in a set might have an implicit, natural order (like the integers), the set itself doesn't define an order. So {1, 3, -6} = {-6, 1, 3}, that is, they're the same set. Order doesn't matter when comparing sets. (This is very different from our intuition with lists, where different orderings do matter.)

There are a few bits of notation I want you to have seen, so now I'm going to write them down for you.

First, how do we say a set contains an element? There's a symbol that looks like this: ∈ (kind of a funny "E") which is used to denote set membership. For example, "3 ∈ A" is pronounced "3 is an element of A." You can think of the funny E as standing for "Element of" to help remember it. Likewise, ∉ means "not an element of" or "does not contain", as in "10 ∉ A". How do we say a set is empty? We call it the empty set and write it as ∅ (or {}).

Next, sometimes we might want to talk about one set being "contained within another". For example, if B = {1, 3, -6, 10}, we might say A is a "subset" of B. This is written with the set containment symbol, which looks kinda like a curvy less-than-or-equal-to (or greater-than-or-equal-to) "A ⊆ B" and it "opens toward" the bigger set.

Sets are sometimes represented abstractly as "Venn diagrams". Here's what the above two sets might look like as a Venn diagram:

(on whiteboard)

Finally, there are a few operations on sets you should know about.

First is "union", written as a little ∪. The union of two sets contains all their elements. So if we have A = {1, 3, -6} and C = {9, 3, 4}, then A ∪ C = {1, 3, -6, 9, 4}. As a mnemonic, think of the "United States" as union of many things into one.

(on whiteboard)

Next is the "intersection". The intersection of two sets contains only the elements they have in common and is written as an upside-down u: ∩. Continuing our example from above, A ∩ C = {3}.

(on whiteboard)

Think of the "intersection of two roads": the intersection is just the part they share, not all of both roads.

Two more operations (I promise) then we're done with math and notation. First is set difference, sometimes called "relative complement" or "set-theoretic difference." It's written with a backslash `\` and refers to all the elements in one set that aren't in another. For example, A `\` C = {1, -6}. Note that in set difference, which set you write first matters (unlike union and intersection). Finally, there's symmetric difference of two sets, which is all the things in the union that aren't in the intersection. It can be written as **Δ**. A **Δ** C = {1, -6, 9, 4}.

Sets in Java

How does Java represent a set? As an abstract data type, specified by the `Set` interface. First we'll talk about the properties and assumptions we might expect from a `Set`, in the abstract. Then we'll talk about two concrete implementations of the data type provided by the Java API and see how they work.

- sets, like lists, are *unbounded*, that is they don't have a fixed size
- duplicate elements are *not* allowed (only new elements are added; attempts to re-add existing elements are ignored – no error, just ignored)
- sets are *unordered* (usually – though there is a subtype called an ordered set)
- sets, lists can contain a `null` element (I hope you like `NullPointerException`s! though note some implementations might forbid `null` elements)
- sets support an `add` (and an `addAll`) operation, which can *modify* the current set
- sets support a `remove` operation of a specific value
- sets support a `size` operation to determine how many elements are currently in the list
- sets support a `contains` (and a `containsAll`) operation to check membership
- and more, but we'll get to them later when we look at the full API that Java supplies.

Let's take a look at the interface: https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/util/Set.html

Not *too* different from `List`, though you'll note some things (like `remove` at an index, or `get`) are not present, as those operations don't make sense in the context of sets – they're unordered, so there is no index!

Pay special attention to a few things:

"...sets contain no pair of elements `e1` and `e2` such that `e1.equals(e2)`" – the `equals` method is very important to sets, and if you stick objects in that don't have an `equals` method, they'll use `Object`'s `equals` method. Make sure that's what you want if so!

Also note that "great care must be exercised if mutable objects are used as set elements. The behavior of a set is not specified if the value of an object is changed in a manner that affects equals comparisons while the object is an element in the set." In other words, if you have a setter that changes an instance variable in an object, and that instance variable is considered by the object's `equals` method, `Set` will have undefined (read: bad) behavior.

So putting relatively immutable things into sets is OK. Like `Integer`s or `String`s. Putting arbitrary objects that can be changed is not so good. Putting things that can be changed, but that you won't change is OK but dangerous – what if you accidentally do end up changing the object? The `Set` will almost certainly misbehave in a weird way.

Other than those two restrictions, you can use `Set`s almost like `List`s. Let's do some examples:

```
Set<Integer> s = new HashSet<Integer>();

s.add(1);
s.add(2);
System.out.println(s); // like lists, you can print them and their contents is printed

Set<Integer> t = new HashSet<Integer>();

t.add(2);
t.add(3);
t.add(4);

for (Integer i : t) {
    System.out.println(i); // like lists, you can iterate over them
}

s.addAll(t); // all elements in t are added to s; t is unchanged but s is not!
System.out.println(s);
System.out.println(t);

s.removeAll(t); // all elements in t are removed from s, as above
System.out.println(s);
System.out.println(t);
```

And you generally do want to use `Set`s when the set properties (of uniqueness and lack-of-intrinsic-order) apply to your data set, especially if your data set is going to be large.

Why? (you might ask.) Because sets have much, much better general performance for insertion, removal, and containment-testing than lists. How? (you might ask.) Well, now we have to talk a little about how the two most common implementations of `Set`s work: `HashSet`s and `TreeSet`s.

On `HashSet`s

One possible implementation of the set is the `HashSet`, which depends upon a correct `hashCode` method. Why? "This class implements the Set interface, backed by a hash table (actually a HashMap instance)." Now let's look at the documentation for `hashCode`: "This method is supported for the benefit of hash tables such as those provided by HashMap."

Wow, hash tables are so important that *every object* in Java must supply a `hashCode` method – it's built into `Object`.

`hashCode` returns an integer, and must obey the contract in its documentation. Let's look at each piece:

- It provides something like (but not exactly like!) equality: If two objects are equal according to the `equals(Object)` method, then calling the `hashCode` method on each of the two objects must produce the same integer result.

  This implies that if you use a field in an `equals` method, you should also use it in the `hashCode` method.

- It is consistent: Whenever it is invoked on the same object more than once during an execution of a Java application, the `hashCode` method must consistently return the same integer, provided no information used in `equals` comparisons on the object is modified.
- It is *not* an equality check, though: It is *not* required that if two objects are unequal according to the `equals(java.lang.Object)` method, then calling the `hashCode` method on each of the two objects must produce distinct integer results. However, the programmer should be aware that producing distinct integer results for unequal objects may improve the performance of hash tables.

So you could have a `hashCode` method that always returned the same integer, like `1`, and it would technically obey the contract. But usually, the `hashCode` of objects is not 1, but instead a large integer. Going back to our old example of (not) aliasing:

```
      String s = new String("x");
      String t = new String("x");

      System.out.println(s == t);
      System.out.println(s.equals(t));
      System.out.println(s.hashCode());
      System.out.println(t.hashCode());
```

Do you expect them to be `==`? No. Do you expect them to be `equals`? Yes. Do you expect them to have the same hash code? Yes, because of the first property above.

Why does this weird integer result in fast ("constant time") lookups?

Because you can use it as an *index* into an array.

In short, "hash tables" are arrays that store objects based upon their "hash code". If you want to put an element into the array, you figure out the right place to put it by checking its hash code. And if you want to see if an element is in the array, you look up its hash code, then jump to the right spot in the array.

In a perfect world, the array would be big enough to hold everything, and the hash codes would always be unique per-object, and this would all just work. In practice, sometimes there are *collisions* – more than one object ends up in the same spot in the array. We resolve these collisions in different ways (one way: each element of the array might be a short linked list of elements with the hash code corresponding to that element's index), and things usually work out with near-constant-time performance.

More on hash tables

Of course, if we have a hash table that's too small, or a hash code method that doesn't, in the words of the Java doc, "as much as is reasonably practical, ... return distinct integers for distinct objects", then everything ends up in just one or two lists and performance is bad.

So it's critical to write a `hashCode` method that works. And recall the contract: it should two objects' `hashCode` should produce the same value if the two objects are `equals`. How might we do this?

Writing a `hashCode` method

Just like when we wrote our `equals` method, we have to consider what instance variables comprise this object's identity. What makes it unique from other objects? Let's consider a version of our `PostalAddress`:

```
public class PostalAddress {
    public final String name;
    public final int number;
    public final LocalDateTime created;
    public PostalAddress(String name, int number) {
        this.name = name;
        this.number = number;
        created = LocalDateTime.now();
    }
}
```

Note that this has a `created` variable, but let's say we don't care about it when checking equality. Therefore we don't care about it when returning a `hashCode`, either. So how can we return a `hashCode` that's valid? One option is to just return an `int`:

```
public int hashCode() {
    return 1;
}
```

Valid hash code? Yes. Good hash code? No.

That's a terrible idea. We want it to depend upon the things that our `equals` method might depend upon. What are those? `name` and `number`. How might we create a `hashCode` on that basis? `number` is already an `int`. Can we get an `int` out of a `String`? Lots of methods return `int`. You can imagine all sorts of convoluted methods that might involve summing up the integer value of the characters stored in the string, and some of these would probably work great. You know what else would work great? The `String`'s `hashCode` method. So you might write:

```
public int hashCode() {
    return number + name.hashCode();
}
```

as a decent (though maybe not optimal) method. It turns out you need to know a tiny bit of number theory to understand why this isn't the best option; when you get to either 250 or 311 I guarantee you'll learn first-hand. But for now, know that this is OK, but not great.

But for now, you can use one of the classes built-in to Java11 to write hash code methods – we talked about this last week:

```
public int hashCode() {
  return Objects.hash(number, name);
}
```

This will do the optimal thing for you behind the scenes. And you'd still need to write a valid `equals` method. Just like before, you'd check to make sure the other `Object` is non-null, that it's an `instanceof PostalAddress`, then cast it to a `PostalAddress` and check that each relevant field is either `==` or `.equals()`.

Last modified: Tuesday, March 1, 2022, 5:52 PM

◀ Sample quiz answers

Jump to...

Programming assignment 06: Similarity detector ▶

© University of Massachusetts Amherst • Site Policies • Site Contact

Moodle Help for Students • Moodle Help for Instructors

You are logged in as Tanish Patel (Log out)
COMPSCI186-SEC01 SP22

Get Help
Help for Students
Help for Instructors

COMPSCI186-SEC01 Using Data Structures Spring 2022

Lecture notes 06-02

On to the trees

So from last class: that's what you need to know for `HashSet`s. If you can reasonably define an `equals` and `hashCode` method, you can get pretty good performance (near constant-time) if you use a `HashSet`. But there's another option that also gets pretty good performance, the `TreeSet`. To describe how it works, we need to define trees.

In computer science and mathematics, a *tree* is a kind of *graph*. What's a graph? It's a set (oh snap!) of *vertices* and *edges* between those vertices (sometimes also called *nodes*). (On board)

A tree is a particular kind of graph. It has a single vertex called a root "at the top", and it grows downward (weird, I know, like an upside-down tree). Each vertex in the tree can have "children", that is, nodes "below" them. For the sake of simplicity, let's say each vertex has zero, one, or two children, and the children are "left children" or "right children".

Trees that have at most two children per vertex are called *binary trees*.

Remember linked lists? It turns out you can model a tree in code, using something very similar:

```
class TreeNode<E> {
    E value;
    TreeNode<E> leftChild;
    TreeNode<E> rightChild;

    //...
}
```

But lucky for you, this is not 187, so we're just going to draw diagrams to give you an intuition, rather than make you code this up yourself.

Why trees?

OK, so now we've just implicitly created a new homework assignment for 187, but who cares, right? It's just a convoluted list, sort of, right?

Right, and wrong. Depending upon how you organize your tree, you can get very good or very bad performance. If you just stuff items into the tree willy-nilly, then yes, it's really no better than a linked list, as you'd have to traverse the entire tree to, for example, look to see if an element is there. In some ways it's worse, because you now also have to write the traversal code for a tree, which is more complicated than the same code for a list. But it turns out if you impose some constraints on the tree, you can do better.

Specifically, let's say that we require that a left child (and all grandchildren and additional descendants) of a node can only contain a value that's *less than* the current node's value. And a right child (and all descendants) can only contain a node that's *greater than* the current node's value. This is called a *binary search tree*.

Two trees!

How does this "Binary Search Tree" property help? Consider a tree that holds the values 1 through 7. Let's say I magically decide to insert them into the tree in this order: {4, 2, 6, 1, 3, 5, 7}:

```
   4
 2   6
1 3 5 7
```

This tree holds 7 values, and takes at most two comparisons to check whether a given value is in the tree or not. If we build an even bigger tree you can see that the tree's *height* (which is also how many comparisons are needed to search it) grows much more slowly than the tree's *size*, which is the total number of nodes in the tree.

Now, let's build a binary search tree containing the following values: {3, 2, 6, 10, 5, 1, 9}

Balanced? Not quite.

But what if we can magically keep the tree (mostly) balanced? You'll learn how in 187. It's not quite constant-time lookup (it's "logarithmic" overhead) but it's really fast nonetheless. The logarithm grows very slowly: https://en.wikipedia.org/wiki/Logarithm. So if we say a tree containing 10 elements has an overhead of 1, a tree containing 1000 elements (100 times as many!) has an overhead of only 3. And a tree containing 1,000,000 elements has an overhead of only 6. 10^9 elements? Overhead of 9. And so on.

Again, there are some details I'm skipping over, for example, how do you make sure your tree doesn't end up looking like a linked list? But you'll see them in 187 and 311.

`TreeSets`

OK, so now we see that in order to build a tree, we need to be able to see if values are less than or equal to other values. Have we seen this before? Sure we have. Java's `Comparable` and `Comparator` interfaces. So if we want to be able to place objects into a `TreeSet`, they'll have to either have a natural ordering (that is, implement `Comparable`), or we can create the `TreeSet` with a specific `Comparator` in order to decide how the tree is built. So let's add one.

Back to our `PostalAddress`:

```
public int compareTo(PostalAddress o) {
    if (name.compareTo(o.name) != 0) return name.compareTo(o.name);
    return Integer.compare(number, o.number);
}
```

Now we can instantiate a `TreeSet` of our `PostalAddress` (though we'll need to add back in our `toString`, first):

```
Set<PostalAddress> addresses = new TreeSet<PostalAddress>();
for (int i = 1; i <= 10; i++) {
    addresses.add(new PostalAddress(i, "Maple St"));
}
System.out.println(addresses);
}
```

Here's the whole "new" `PostalAddress.java` for your reference:

```java
package hashes;

import java.time.LocalDateTime;
import java.util.Objects;
import java.util.Set;
import java.util.TreeSet;

public class PostalAddress implements Comparable<PostalAddress> {
    public final int number;
    public final String name;
    public final LocalDateTime created;

    public PostalAddress(String name, int number) {
        this.number = number;
        this.name = name;
        created = LocalDateTime.now();
    }

    public boolean equals(Object o) {
        if (o == null) {
            return false;
        }
        if (!(o instanceof PostalAddress)) {
            return false;
        }

        PostalAddress p = (PostalAddress)o;

        if (number == p.number && name.equals(p.name)) {
            return true;
        }
        return false;
    }

    public int hashCode() {
        return Objects.hash(number, name);
    }

    public String toString() {
        return number +  " " + name;
    }

    public static void main(String[] args) {
        Set<PostalAddress> ts = new TreeSet<>();
        for (int i = 1; i <= 10; i++) {
            ts.add(new PostalAddress("Maple St", i));
        }
        System.out.println(ts);
    }

    @Override
    public int compareTo(PostalAddress o) {
        if (name.compareTo(o.name) != 0) return name.compareTo(o.name);
        return Integer.compare(number, o.number);
    }
}
```

Which should I use?

Generally, you should reach for `HashSet` when you want to use the `Set` interface, in the same way you can reach for `ArrayList`. Usually, objects have a custom `hashCode` method defined already, and if they don't, you can usually write one pretty easily using `Objects.hash()`.

`TreeSet`s are most useful when the objects in the set have a natural ordering, *and you care about using it*. In particular, `TreeSet` also implements the `NavigableSet` and `SortedSet` interfaces, which means you can the smallest or largest elements of the set, or the elements closest to a given value, in "logarithmic" time, which is about as good as it gets (only "constant time" is better). (See the respective JavaDocs for those interfaces.)

If the problem you're working on makes frequent use of computing these values, then a `TreeSet` might be a better choice than a `HashSet`. But 90% or more of the time, `HashSet` is what you're gonna want.

A worked example

We're going to walk through solving another toy problem that's made relatively straightforward with the `Set` abstraction. Then we'll talk about an extension to this problem that's harder to solve, and introduce a new abstraction – the `Map` – to solve it.

The problem is called "Santa's little helper" and is adapted from http://adventofcode.com/2015/day/3.

As an aside, if you're looking for nerdy little problems to polish your programming skills on, there are many such sites on line. I like Advent of Code because it's language-agnostic, there's a nice ramp-up as the days of the month go on, it's kind of a group activity you can do with your friends, and there's now a nice archive of these problems to work on. It's no 1337code – it's its own thing.

Santa's little helper

Santa is delivering presents to an infinite two-dimensional grid of houses.

He begins by delivering a present to the house at his starting location, and then an elf at the North Pole calls him via radio and tells him where to move next. Moves are always exactly one house to the north (`^`), south (`v`), east (`>`), or west (`<`). After each move, he delivers another present to the house at his new location.

However, the elf back at the north pole has had a little too much eggnog, and so his directions are a little off, and Santa ends up visiting some houses more than once. **How many houses receive at least one present?**

For example:

- `>` delivers presents to 2 houses: one at the starting location, and one to the east.
- `^>v<` delivers presents to 4 houses in a square, including twice to the house at his starting/ending location.
- `^v^v^v^v^v` delivers a bunch of presents to some very lucky children at only 2 houses.

Toward a solution

We're going to try to solve this in Java. Let's fire up Code and create a new package. Now let's write a `DeliverySimulator` class with a single method that takes a input of directions and returns the set of locations visited.

```
import java.util.Set;

public class DeliverySimulator {


    public static Set<Location> locationsVisited(String directions) {
        return null;
    }
}
```

Note we can use Code to "stub out" our program, and to create empty implementations wherever possible here.

Let's also add a method to compute the actual number of houses visited:

```
public static int housesVisited(Set<Location> visited) {
    return visited.size();
}
```

Hey, half done! Well, sorta.

What should a location look like? Let's give it an x and y coordinate:

```
public class Location {
    public final int x;
    public final int y;

    public Location(int x, int y) {
        this.x = x;
        this.y = y;
    }
}
```

Since we know we're going to be storing locations in a set, we should make sure we implement meaningful `equals` and `hashCode` methods. :

```
@Override
public int hashCode() {
  return Objects.hash(x, y);
}

@Override
public boolean equals(Object o) {
    if (o == null)
        return false;
    if !(o instanceof Location)
        return false;
    Location l = (Location) o;
    if (x == l.x && y == l.y)
        return true;
    return false;
}
```

and maybe:

```
public String toString() {
    return "(" + x + ", " + y + ")";
}
```

Does this work? We could add a main method to do some testing:

```
public static void main(String[] args) {
    Location x = new Location(0, 0);
    System.out.println(x);
    Location y = new Location(0, 0);

    System.out.println(x == y);

    System.out.println(x.equals(y));
    System.out.println(x.equals(new Location(1, 0)));
}
```

Sometimes it's nice to have automated tests.

A Unit Test

OK, now we'll write a test.

```
import static org.junit.Assert.*;

import org.junit.Test;

public class LocationTest {
    @Test
    public void testLocationEquals() {
        Location x = new Location(0,0);
        Location y = x;
        Location z = new Location(0,0);
        assertTrue(x == y);
        assertFalse(x == z);

        assertEquals(x, y);
        assertEquals(x, z);
    }

    @Test
    public void testLocationNotEquals() {
        assertFalse(new Location(0,0).equals(new Location(0,1)));
    }
}
```

A reminder about `assertEquals`: it uses the objects' built-in `equals()` method. If you have tests failing (say, in a project?), and the expected and actual look the same, maybe you forgot to define `.equals` on your objects?

Anyway, let's go back to `locationsVisited`. According to the problem statement, the house at (0, 0) always gets a present:

```
public static Set<Location> locationsVisited(String directions) {
    Set<Location> visited = new HashSet<Location>();

    visited.add(new Location(0,0));

    return visited;
}
```

Does this work? Let's add a test.

```
@Test
public void testEmptyDirections() {
    assertEquals(new HashSet<Location>(Arrays.asList(new Location(0,0))), DeliverySimulator.locationsVisited(""));
}
```

Then we need to think about where the sleigh goes, and deliver a present at each stop:

```
    public static Set<Location> locationsVisited(String directions) {
        Set<Location> visited = new HashSet<>();
        visited.add(new Location(0, 0));

        int x = 0;
        int y = 0;

        for (int i = 0; i < directions.length(); i++) {
            final char d = directions.charAt(i);
            if (d == '>') {
                x += 1;
            } else if (d == '<') {
                x -= 1;
            } else if (d == '^') {
                y += 1;
            } else if (d == 'v') {
                y -= 1;
            }
            visited.add(new Location(x, y));
        }

        return visited;
    }
```

And now we add some tests:

```
@Test
public void testSimple() {
    assertEquals(new HashSet<Location>(Arrays.asList(new Location(0, 0), new Location(1, 0))),
            DeliverySimulator.locationsVisited(">"));
}

@Test
public void testFour() {
    assertEquals(
            new HashSet<Location>(
                    Arrays.asList(new Location(0, 0), new Location(0, 1), new Location(1, 1), new Location(1, 0))),
            DeliverySimulator.locationsVisited("^>v<"));
}
```

You could also add checks for `housesVisited`, either to their own test cases (which would be more true to the spirit of unit tests) or within the above.

A new problem, and a new data structure: Map ADT

What if, each time Santa visited a house, he delivered a present, and we wanted to know the total number of presents delivered? How might you go about tracking this?

You could, for example, keep a `List` of visited locations (instead of a `Set`). Then you could count the number of times each location appeared. But how would you tabulate it? You want to build some kind of *table*, a way to connect a set of things, and for each thing, some information about it.

```
(0,0) | (0,1) | (1,1) | (1,0)
------|-------|-------|------
  2      1        1       1
```

You might create two `List`s, one of `Location` and one of `Integer`, where the `i`th element in each corresponded. But that's clunky, hard to program, and has the poor performance of a list for lookups. You might edit the `Location` to contain a `counter` variable. That could work; but it might cause problems in other contexts, when the thing you're storing is doesn't have a clear "has-a" relationship with whatever you're adding it to. Objects, like functions, should generally do a small set of things well.

What if I told you there was a data structure that solved this problem? What if I told you you've already (sorta) seen it? What if I told you it could be yours for no money down, and no monthly installments?

Introducing... the `Map`.

`Map` ADT

A `Map` "maps" keys to values. In other words, it associates one kind of thing with another kind of thing.

The first "thing" is the key. Keys are lookup keys; you can think of them as kinda analogous to an index card, or a page number, or a URL; they are the thing we can do lookups on (though they can also be useful data in and of themselves).

The second "thing" is a value – each value is associated with a key.

So `Map`s model, in essence, a table (on board). For our Santa problem, we might use `Location`s as keys and `Integer`s as values.

The keys in a map form a set – there can only be one of each. And each value is associated with exactly one key. Keys are unique (as they are a set), but the same value can be associated with more than one key (as shown in our table above).

And, the key values should be immutable, or at least, should not change with respect to `equals`, for exactly the same reason as in `Set`s – in fact, if you recall, `Set`s are implemented using `Map`s.

Like `Set`s, `Map`s come in two flavors you'll likely use: `HashMap`s and `TreeMap`s, with the same constraints (about efficient `hashCode` and `compareTo` methods).

The type signature is slightly different though: `Map<K, V>`; the key and value types are both parameterized. This should make sense: there's no particular reason a map would store keys and values of the same type, only that all keys are of the same type, and all values are of the same type.

There are many useful `Map` methods, but we'll start with the basics that are unique to `Map`:

- `containsKey` checks to see if a particular object is in the map's set of keys, by `equals` (this is fast, because it's just like set access)
- `containsValue` checks to see if a particular object is in the map (this is slow/linear, since generally it requires traversing each key's value until the value is found or all values are checked)
- `get`, which given a key returns the associated value (or `null` if not found!!!)
- `getOrDefault`, which given a key and a default returns the associated value for the key (or the default if not found)
- `put` which given a key and value, insert that key and value into the map (overwriting if previously in the map)
- `keySet` returns the `Set<K>` of keys in this map
- and so on; read the Javadoc for details.

Let's use these to come up with an answer to our previous question ("how many presents per house").

How many? Etc.

Let's parallel the code we wrote previously, but use a `Map` instead of a `Set`. We have to import some things, but then we can copy/paste our code and modify it in place.

```
    public static Map<Location, Integer> locationCount(String directions) {
        Map<Location, Integer> locationCount = new HashMap<>();

        locationCount.put(new Location(0, 0), 1);

        int x = 0;
        int y = 0;

        for (int i = 0; i < directions.length(); i++) {
            final char d = directions.charAt(i);
            if (d == '>') {
                x += 1;
            } else if (d == '<') {
                x -= 1;
            } else if (d == '^') {
                y += 1;
            } else if (d == 'v') {
                y -= 1;
            }

            Location current = new Location(x, y);

            // update code goes here

            return locationCount;
```

We have to declare a `Map` in our method, and translate the `Set.add` into `Map.put`. We also need to decide what to do when it's time to update our map. There are two cases:

```
            // case 1: not in the map
            if (!locationCount.containsKey(current)) {
                locationCount.put(current, 1);
            }
            // case 2: already in the map
            else {
                int j = locationCount.get(current);
                j++;
                locationCount.put(current, j);
            }
```

If the location isn't in the map, we put it in, along with a count of 1. If it is, we pull out the current value, increment it, and update it in the map. We can do this more concisely:

```
            int t = locationCount.getOrDefault(current, 0);
            t++;
            locationCount.put(current, t);
```

Here, we pull the count out of the map, but set it to 0 if it's not there. Then we increment it and put it back in.

Let's test it out in a `main` method:

```
public static void main(String[] args) {
    Map<Location, Integer> lc = locationCount("^>v<");
    System.out.println(lc);
}
```

What do we expect? Same as the table above. What do we get?

```
{(1, 0)=1, (0, 0)=2, (1, 1)=1, (0, 1)=1}
```

Bingo!

How might we compute the number of houses visited, given the code written so far (in the context of our `main` method)?

```
        // compute number of houses visited
        System.out.println(lc.size());
```

How might we find the set of houses visited, given the code written so far?

```
public Set<Location> locationsVisited() {
    // what goes here?
}
```

How might we find the number of presents delivered to a house? Let's define a method to compute it, given an already-filled-in `Map` and a `Location` to report on:

```
public static int numberPresents(Map<Location, Integer> locationCount, Location loc) {
    return locationCount.getOrDefault(loc, 0);
}
```

And then test it in our main method:

```
        // number of presents
        System.out.println(numberPresents(lc, new Location(0, 0)));
        System.out.println(numberPresents(lc, new Location(9, 9)));
```

On `Map` generality

`Map`s need not just be used to count things; they can be used to associate *any* (relatively immutable) set of keys with *any* values you like. For example, you could store:

- URLs and the results of retrieving those URLs
- filesystem paths and the sizes of those files
- student IDs and photos
- latin names and genomes
- basically, anything!

Last modified: Thursday, March 3, 2022, 4:22 PM

◀ similarity-detector-student.zip

Jump to...

hex.zip ▶

© University of Massachusetts Amherst • Site Policies • Site Contact

Moodle Help for Students • Moodle Help for Instructors

You are logged in as Tanish Patel (Log out)
COMPSCI186-SEC01 SP22

Get Help
Help for Students
Help for Instructors

COMPSCI186-SEC01 Using Data Structures Spring 2022

Lecture notes 07-01

Announcements

Reminder: you *should not* use `Map`s for the current programming assignment, or the one I am going to post Friday. You just need `Set`s and `List`s – in fact, `List`s would work, except they're too slow for large inputs – you need `Set`s for their faster operations.

Quiz 3 `sort` – you could just use `sort()`! We didn't want you to re-invent the wheel here. We will not overly penalize you if you got lost in the weeds there.

Another worked example: Maps and anagrams

We're going to do another worked example. In particular, we're going to write an anagram finder, that is, a program that given a word list, looks for words in that list that are anagrams of one another. An anagram of a word is a rearrangement of the letters in the word that results in a new (different) word, using each of the letters from the original word once. I'll go start-to-finish again today, building the project in Code.

Program sketch

What should our program do?

First, how might we recognize anagrams? For example, we could write a method to count the occurrences of each letter in a word, then store that in an array (or object), then check those arrays/objects for `equals`-style equality. And then we could perhaps write a `hashCode` method for those objects, etc.

You could do that, and it would work. But for purposes of illustration, I'm going to take advantage of a well-known trick, which is that if you sort the letters of two words and compare the sorted letters, they'll be equal iff the words are anagrams of one another. We can treat the sorted letters as *identifying the group of anagrams* for which this word is a member.

Here's an example. Consider the words "cat" "act" and "tac", all of which are anagrams of one another. If you sort their letters, you get "act", regardless of the word you started with. Here, "act" happens to be one of the words, but that's not necessarily the case. For example, "apple" has an sorted-letter identifier of "aelpp", which is not a word in the English langauge (as far as I know).

The idea then is for each word, find its identifier, and add it to the set of anagrams corresponding to the identifier. Note that the identifier may or may not be one of the words ("cat" vs "apple").

So now we've got the core of the algorithm. How can we group words into clusters of anagrams? I'm going to suggest a `Map<String, List<String>>`. The keys are going to be the sorted-letter version of the words, and the values will be lists of all the words that are associated with this sorted-letter version of the word.

(on board)

This structure, where a key is associated with many values, is sometimes called a "Multimap"; Java doesn't directly support multimaps, but associating a value with a collection type is an ad hoc version of this.

So what are we going to do? Something like the following:

- create a multimap
- read a list of words (a list? an array? or process one-by-one? up to us; the first is probably simplest, the last might be more memory efficient; it depends upon how big you expect the list to be)
- for each word:
  - compute its sorted identifier

- add it the multimap
  - the key is the identifier
  - the word must be appended to the value, which is a list

Then, we can write methods to query the multimap. Let's get started.

Coding up `AnagramFinder`

First, the instance variable that represents the map:

```
public class AnagramFinder {
    private final Map<String, List<String>> anagrams;

    public AnagramFinder() {
        anagrams = new HashMap<String, List<String>>();
    }
}
```

Next, the `add` method, to add a word to the `AnagramFinder`. What should `add` do?

It should look up the word in the map, and add it to the associated list. What if there is no list? It should make a new one and insert it into the map.

To look up the word, we'll need a method to return the letter-by-letter alphabetization of a `String`. There are several ways to do this. Here's one:

```
private static String alphabetized(String word) {
      char[] a = word.toCharArray();
      Arrays.sort(a);
      return new String(a);
  }
```

Why is this method static? It does not depend upon the instance in any way, so there is no need to make it an instance method. If it does later change to be part of the instance (that is, if we attempt to call an instance method from it, or use an instance variable from it), the type checker will alert us. Further, static methods could be moved (or copied) easily to another class if appropriate – this is part of a process called "refactoring".

OK, back to `add`. It has to handle two cases: when the alphabetized version of the word is in the multimap already, and when it's not.

(on board)

There are a couple of different ways you could write this. For example, you could handle the two cases completely separately:

```
public void add(String word) {
  String key = alphabetized(word);
  if (!anagrams.containsKey(key)) {
    List<String> l = new ArrayList<String>();
    l.add(word);
    anagrams.put(key, l);
  }
  else {
    List<String> l = anagrams.get(key);
    l.add(word);
  }
}
```

Or you could deal with the not-in-map problem first, and unify things otherwise:

```
public void add(String word) {
  String key = alphabetized(word);
  if (!anagrams.containsKey(key)) {
    anagrams.put(key, new ArrayList<>());
  }
  anagrams.get(key).add(word);
}
```

I find them both fairly readable, but things being otherwise equal, I will generally choose the shorter solution.

Let's write some code in our `main` method to test this out.

```
public static void main(String[] args) {
  AnagramFinder af = new AnagramFinder();
  af.add("bird");
  af.add("drib");

  af.add("and");
  af.add("nad");
  af.add("dan");

  af.add("it");}
```

OK, but we forgot to write methods to get anything out of the `AnagramFinder`! Let's do so now.

There's no real reason for this multimap to store `List`s as values, though – really it should store a `Set<String>`. Let's do that now, and see how Code and Java's type system show us what needs to be changed.

(demo refactoring from `List` to `Set`)

Finding the anagrams of a word

Let's write a method that returns the anagrams of a given word, or an empty list if there are no such anagrams.

```
public Set<String> anagramsOf(String word) {
  return // what?
}
```

`anagrams.getOrDefault(alphabetized(word), new HashSet<>())` – we can't just use `get()`, because we don't want to accidentally return `null`. In general, you don't want to return `null` because callers won't expect it and you'll cause `NullPointerException`s.

Does it work?

```
System.out.println(af.anagramsOf("and"));
System.out.println(af.anagramsOf("it"));
System.out.println(af.anagramsOf("boo"));
```

Reading words from a file

Let's add the ability to read from a file:

```
public void addFromFile(File f) throws FileNotFoundException {
    Scanner s = new Scanner(f);
    while (s.hasNext()) {
        add(s.next());
    }
    s.close();
}
```

(Note there are lots of ways to read from files, this is just one. It's what's done in AP CS last I checked, but you get the idea.)

Finding the most-anagram-words

Now let's write a method to find the word(s) with the most anagrams. There could be more than one, but let's just return any such set with the most anagrams (or an empty list if there are none yet). You'll probably want to use the `Map.values`.

```
public Set<String> mostAnagrams() {
    int longest = -1;
    Set<String> mostAnagrams = new HashSet<>();
    for (Set<String> set : anagrams.values()) {
        if (set.size() > longest) {
            longest = set.size();
            mostAnagrams = set;
        }
    }
    return mostAnagrams;
}
```

Now let's test it:

```
af.addFromFile(new File("/usr/share/dict/words")); // a common file on UNIX-like OSes
System.out.println(af.mostAnagrams());
```

Something to think about: What if we want to find the set of all sets of anagrams of maximum length? You should be able to do this on your own. A sketch:

- Find the size of the largest set (which we already did, above).
- Now create a `Set<Set> to hold the result.
- Iterate over the `anagrams.values()` just like we did above, but this time, each time you find a `Set<String>` that's of the largest size, add it to your result set. If you find a *larger* set of anagrams, through away your current result set and start a new one with this larger set of anagrams as its first member.
- Then return the result set.

I'll leave the actual implementation to you. I strongly suggest you take the time to do it, just for the practice! (If you do it correctly on my wordlist, you get three sets of anagrams. They are: `[[terse, tsere, ester, reest, reset, stree, steer, estre, stere], [creat, trace, caret, cater, carte, recta, react, crate, creta], [organ, groan, orang, angor, grano, goran, argon, nagor, rogan]]`).

Once more, double time, in Python

Python is another programming language. It's pretty popular in some circles, and for good reason: it makes writing easy programs ridiculously easy. There are various (somewhat hidden) costs involved though. For example, the lack of a static type system makes it harder to build large programs correctly. A good rule of thumb is that if the program is going to be more than a couple hundred lines or so, you probably shouldn't use Python. (Though, you can do a lot in 200 lines of Python!).

Unfortunately, perhaps, this gives students the impression that Python and similar languages are "the best" since most of what they're asked to program in school fits this description – but that's not generally the case for the real world, where programs are large, stick around for years, and are maintained by many different people (which is basically the worst case for Python).

And Python's bytecode is not terribly amenable to JIT compilation, so Python is notoriously "slow". Slow is relative, of course, but if you have a large, computationally-intensive job, plain-old-python without native extensions is not always the right choice.

Anyway, I'm going to talk more about Python near the end of the semester, but for now, let me very quickly demo a translation of the program we wrote above into Python now.

The code

```
def alphabetized(word):
    return ''.join(sorted(word))

def add(anagrams, word):
    key = alphabetized(word)
    # if key not in anagrams:
    #     anagrams[key] = [word]
    # else:
    #     anagrams[key].append(word)

    # or shorter
    anagrams.setdefault(key, []).append(word)

def anagrams_of(anagrams, word):
    return anagrams.get(alphabetized(word), [])

def add_from_file(anagrams, f):
    for line in f:
        add(anagrams, line.strip())

def most_anagrams(anagrams):
    n = -1
    most = []
    for l in anagrams.values():
        if len(l) > n:
            n = len(l)
            most = l
    return most

def main():
    a = {}
    with open('/usr/share/dict/words') as f:
        add_from_file(a, f)
    print(most_anagrams(a))

if __name__ == '__main__':
    main()
```

What we've done, and what to do

So far, we reviewed 121 material (including basic control flow, conditionals, expressions, statements, arrays, objects and classes, scope, and references).

We've introduced several foundational ADTs:

- lists
- sets
- maps

and covered their properties. We've also seen their implementations in the Java API.

We've seen their methods, and used them to iterate over, look up items within, and modify them.

And, guess what? For about 80% or more of the programs you're likely to write out in the real world, this is what you need to know, at least in terms of standard data structures. Lists, sets, maps, will let you represent most problems generally, and if not, you can take 187 to learn how to define your own data structures.

There are actually two or three data types and associated implementations that 187 covers that we haven't yet: stacks, queues, and priority queues / heaps. But they're pretty straightforward (and we'll get to them later this semester, at least the first two).

So what's left? Are we done for the semester? Of course not!

First, more practice. We're getting you ready for general programming out in the world (and for 187 in particular), so there will be more programming assignments, that may start to feel more difficult in various ways. Not all of them will involve new data structures concepts, but instead they'll serve to give you more practice. We'll also continue removing some of the training wheels you've had so far (full sets of test cases,

for example) so that you can start to get ready for the 187 experience (or, you know, the real world). We'll also do simplified versions of some previous 187 assignments to give you a running start in that class.

Second, more exposure to other topics in computer science and informatics. This course is a prerequisite for not just 187 but for various others as well. Some of our lectures and assignments will focus on things like: working with files (you'll see `CSVReader` if you haven't already in a programming assignment); simulated simple interactions over the network (with web servers or the like); text processing and data analysis (search engine stuff); and so on.

Finally, we're going to touch upon a few more core computer science concepts in detail. In particular, we're going to continue our study of *algorithms* – how we do certain tasks – and start to develop language to describe how efficient different approaches to the same problem might be. We'll continue to focus on "toy problems" here, things like sorting lists of numbers and searching simple graphs, but the algorithms we develop and the approaches we take will be useful to you later when tackling bigger problems. (Some of this you'll see in later assignments, I hope.)

Thinking about efficiency

So to think about how "efficient" an algorithm, or a piece of code, is, we need a way to quantify how long it takes to run. Our rule of thumb is as follows. Things take either:

- a small, constant amount of time, which we'll approximate as "about one unit," or
- they take an amount of time dependent upon some variable or variables

To simplify things, we say that almost all operators and keywords evaluate in a small, constant amount of time in Java: basic arithmetic, conditionals, assignment, array access, following the branch in control flow, and method invocation. So you might look at a method like:

```
int add(int x, int y) {
  int sum = x + y;
  return sum;
}
```

and say something like: well, when this method runs, first it adds x and y (1). Then it assigns to sum (1). Then it returns (1). So it takes "about" three units of time to execute.

```
int sub(int x, int y) {
  int diff = x - y;
  return diff;
}
```

About how long does this take to run?

Or you might look at:

```
void honkIfEven(int x) {
  if (x % 2 == 0) System.out.println("honk");
}
```

and say something like, well, first `x % 2` is computed. Then it's compared to zero. So the method takes at least two units. Then it might take a third to print "honk".

Does it?

Well, that depends on the implementation of `println()`. To do a "real" analysis, we have to drill down into any method that's called and check how it works, and look at methods it calls, and so on. For the purposes of this class, we'll just state that certain methods are roughly constant time (like `println`), even though that's not strictly true, in ways that will probably become clear to you as we go on.

OK, be that as it may, there's something important to note here, which is that both of these methods take a small, fixed amount of time that doesn't depend upon anything. Let's look at something different:

```
int sum(int[] a) {
  int s = 0;
  for (int i: a) {
    s += i;
  }
  return s;
}
```

How long does this method take to execute? Well, about one to declare and assign 0 to s.

Then about one to update i each time through the loop, and another to update s each time through the loop.

Then one for returning s.

So what's the answer? It depends upon the length of the array, right? **It depends upon the input** in other words, **it's not a constant**. Some parts of the runtime are (the initial setup and the return) are constant, but some are not (the loop). Here, we might say the runtime is about 2 + 2 * (a.length). In other words, the runtime here is a *function* (in the mathematical sense) of the length of a. It's *proportional* t the length of a.

Generally, any time you see a loop, you have the possibility of a non-constant runtime, that is, of a runtime that's a function of (some aspect of) the input.

```
void print(List<Integer> l) {
    for (int i : l) {
        System.out.println(i);
    }
}
```

About how long does this take to run? Again it depends. One would probably say "proportional to the length of l", though.

Early returns

What about if the loop can return early?

```
boolean containsOne(int[] a) {
  for (int i: a) {
    if (i == 1) return true;
  }
  return false;
}
```

Like before, the runtime here varies upon the input. But in a new and excitingly different way! When do we exit the loop? Who knows?!?

Since we can't know, we generally concern ourselves with the *worst case*, that is, what's the longest this loop could run?

Answer: it's a function of the length of a, again. Again, About 2 * `a.length`, given our previous analysis.

(The other kind of analysis we might do is an "average case" analysis, but we'll mostly leave that for COMPSCI 311.)

Also, just because you see a loop, it doesn't mean that a method runs in non-constant time. For example:

```
int sumFirstThree(int[] a) {
  int sum = 0;
  for (int i = 0; i < 3; i++) {
    sum += a[i];
  }
  return sum;
}
```

...runs in constant time. Nor does there have to be an array in the parameter list (or as an instance variable, etc.) to trigger non-constant time behavior.

Nested loops

Now consider the case of `for` loops within `for` loops. Suppose we had an algorithm for duplicate detection that looked like this:

```
boolean containsDuplicate(int[] a) {} {
  for (int i = 0; i < a.length; i++) {
    for (int j = 0; j < a.length; j++) {
      if (i == j) continue;
      if (a[i] == a[j]) return true;
    }
  }
  return false;
}
```

How does this algorithm operate? (On board.)

How long does it take to run? For each iteration of the outer loop, we have to go through the entire inner loop. So we have to run a.length * (cost of inner loop) * a.length.

This method's runtime is a function of its input, but it's no longer a linear (first-degree polynomial) function; it's "quadratic" – that is, it's runtime is proportional to a.length *squared*.

That's a *lot* worse, especially as a.length grows. Who cares, right? Computers are fast? 3 GHz = 3 billion operations a second, right?

Well, what if we're working with a big array? Say, a million elements? 10 ns each is only 10 ms total to run. Something that runs in time proportional to the array length will be manageable. What about quadratic? 1,000,000 x 1,000,000 = 1,000,000,000,000. That's a lot of zeroes! Even if each step only takes, say, 10 ns, we're still talking about 10,000 seconds (nearly three hours) to complete!

So generally, when we write methods or call them, and we suspect that they're going to be used with large inputs, we should be thinking about how much time they'll take to run. Many efficient algorithms are *linear* in the size of their input, though some are a little worse, and some are much worse.

OK, Marc, but we don't need to go through the entire array inside the inner loop; we could just go through "what's left" at the end, since we've already checked everything there, right?

Quadratic or not?

```
boolean containsDuplicate(int[] a) {} {
  for (int i = 0; i < a.length; i++) {
    for (int j = i + 1; j < a.length; j++) {
      if (a[i] == a[j]) return true;
    }
  }
  return false;
}
```

Well, again, how many steps does the inner loop take? It's not always a.length, but it's a function of a.length – the first time through, it is a.length - 1; the next time a.length - 2, and so on, down to 3, 2, 1, 0. What's that proportional to? It's *still a function of a.length* (about 1/2). Here's an illustration (on board), or you can run the sums if you like.

Asymptotes

We've been talking about running time (mostly) on the basis of the degree of the polynomial (linear, quadratic, etc.). But remember that each variable has a coefficient, and that depending upon your value of n and the coefficients, linear-time algorithms are not strictly better than quadratic. For a silly example, consider which is faster:

- an algorithm with runtime of 1,000,000 n
- an algorithm with runtime (n^2)/ 1,000,000

For "large enough" values of n, the first algorithm is faster. How large? (set them equal to one another and solve for n) "Large enough" in this case is 10^12. Normally the coefficients aren't quite this lopsided, but it is true in practice that sometimes, a small-coefficient quadratic algorithm is faster than a larger-coefficient (but better polynomial) algorithm for small but reasonable values of n.

Implementations matter

How about the following?

```
boolean allEven(List<Integer> list) {
  for (int i = 0; i < list.size(), i++) {
    if (list.get(i) % 2 == 1) return false;
  }
  return true;
}
```

Answer? *It depends.* This is where understanding *how* the implementation (`ArrayList`? `LinkedList`? Something else?) underneath a given abstraction works *matters*.

If you're using an `ArrayList`, this will be linear, just as when it was for an array. But remember that to get to the `i`th element in a linked list, you have to traverse the list. So if we're using a linked list, each time we call `get` here, we are doing something that's dependent upon the length of the list. So it will be quadratic!

To further muddle this mess, the enhanced for loop:

```
boolean allEven(List<Integer> list) {
  for (int i: list) {
    if (i% 2 == 1) return false;
  }
  return true;
}
```

is actually smart enough to "remember" where it was the next time through, so it *won't* be quadratic. But you wouldn't know this unless you knew how lists and their iterators were implemented. Take 187! :)

But, usually the Java Docs will help you here. Take a look at ArrayList to see that:

> The size, isEmpty, get, set, iterator, and listIterator operations run in constant time. The add operation runs in amortized constant time, that is, adding n elements requires O(n) time. All of the other operations run in linear time (roughly speaking). The constant factor is low compared to that for the LinkedList implementation.

Compare with the LinkedList:

> All of the operations perform as could be expected for a doubly-linked list. Operations that index into the list will traverse the list from the beginning or the end, whichever is closer to the specified index.

I guess you need to know what "as could be expected" means. Again, take 187 to be a better programmer!

Another example of implementation mattering

Remember that some things can be computed in different ways.

For example, to compute the sum of numbers from 1 to n,

- Sum from one to n, or
- compute n * (n + 1) / 2

```
int sumTo(int n) {
  int sum = 0;
  for (int i = 0; i <= n; i++) {
    sum += n;
  }
  return sum;
}
```

vs

```
int sumTo(int n) {
  return (n * (n + 1)) / 2;
}
```

Different algorithms *that accomplish the same goal* can have different running time behaviors.

Last modified: Wednesday, March 9, 2022, 8:38 AM

◀ hex.zip

Jump to...

Lecture notes 07-02 ▶

Lecture notes 07-02 ▶

© University of Massachusetts Amherst • Site Policies • Site Contact

Moodle Help for Students • Moodle Help for Instructors

You are logged in as Tanish Patel (Log out)
COMPSCI186-SEC01 SP22

Get Help
Help for Students
Help for Instructors

COMPSCI186-SEC01 Using Data Structures Spring 2022

Lecture notes 07-02

What we've done, and what to do

So far, we reviewed 121 material (including basic control flow, conditionals, expressions, statements, arrays, objects and classes, scope, and references).

We've introduced several foundational ADTs:

- lists
- sets
- maps

and covered their properties. We've also seen their implementations in the Java API.

We've seen their methods, and used them to iterate over, look up items within, and modify them.

And, guess what? For about 80% or more of the programs you're likely to write out in the real world, this is what you need to know, at least in terms of standard data structures. Lists, sets, maps, will let you represent most problems generally, and if not, you can take 187 to learn how to define your own data structures.

There are actually two or three data types and associated implementations that 187 covers that we haven't yet: stacks, queues, and priority queues / heaps. But they're pretty straightforward (and we'll get to them later this semester, at least the first two).

So what's left? Are we done for the semester? Of course not!

First, more practice. We're getting you ready for general programming out in the world (and for 187 in particular), so there will be more programming assignments, that may start to feel more difficult in various ways. Not all of them will involve new data structures concepts, but instead they'll serve to give you more practice. We'll also continue removing some of the training wheels you've had so far (full sets of test cases, for example) so that you can start to get ready for the 187 experience (or, you know, the real world). We'll also do simplified versions of some previous 187 assignments to give you a running start in that class.

Second, more exposure to other topics in computer science and informatics. This course is a prerequisite for not just 187 but for various others as well. Some of our lectures and assignments will focus on things like: working with files (you'll see `CSVReader` if you haven't already in a programming assignment); simulated simple interactions over the network (with web servers or the like); text processing and data analysis (search engine stuff); and so on.

Finally, we're going to touch upon a few more core computer science concepts in detail. In particular, we're going to continue our study of *algorithms* – how we do certain tasks – and start to develop language to describe how efficient different approaches to the same problem might be. We'll continue to focus on "toy problems" here, things like sorting lists of numbers and searching simple graphs, but the algorithms we develop and the approaches we take will be useful to you later when tackling bigger problems. (Some of this you'll see in later assignments, I hope.)

Thinking about efficiency

So to think about how "efficient" an algorithm, or a piece of code, is, we need a way to quantify how long it takes to run. Our rule of thumb is as follows. Things take either:

- a small, constant amount of time, which we'll approximate as "about one unit," or
- they take an amount of time dependent upon some variable or variables

To simplify things, we say that almost all operators and keywords evaluate in a small, constant amount of time in Java: basic arithmetic, conditionals, assignment, array access, following the branch in control flow, and method invocation. So you might look at a method like:

```
int add(int x, int y) {
  int sum = x + y;
  return sum;
}
```

and say something like: well, when this method runs, first it adds x and y (1). Then it assigns to sum (1). Then it returns (1). So it takes "about" three units of time to execute.

```
int sub(int x, int y) {
  int diff = x - y;
  return diff;
}
```

About how long does this take to run?

Or you might look at:

```
void honkIfEven(int x) {
  if (x % 2 == 0) System.out.println("honk");
}
```

and say something like, well, first `x % 2` is computed. Then it's compared to zero. So the method takes at least two units. Then it might take a third to print "honk".

Does it?

Well, that depends on the implementation of `println()`. To do a "real" analysis, we have to drill down into any method that's called and check how it works, and look at methods it calls, and so on. For the purposes of this class, we'll just state that certain methods are roughly constant time (like `println`), even though that's not strictly true, in ways that will probably become clear to you as we go on.

OK, be that as it may, there's something important to note here, which is that both of these methods take a small, fixed amount of time that doesn't depend upon anything. Let's look at something different:

```
int sum(int[] a) {
  int s = 0;
  for (int i: a) {
    s += i;
  }
  return s;
}
```

How long does this method take to execute? Well, about one to declare and assign 0 to s.

Then about one to update i each time through the loop, and another to update s each time through the loop.

Then one for returning s.

So what's the answer? It depends upon the length of the array, right? **It depends upon the input** in other words, **it's not a constant**. Some parts of the runtime are (the initial setup and the return) are constant, but some are not (the loop). Here, we might say the runtime is about 2 + 2 * (a.length). In other words, the runtime here is a *function* (in the mathematical sense) of the length of a. It's *proportional* t the length of a.

Generally, any time you see a loop, you have the possibility of a non-constant runtime, that is, of a runtime that's a function of (some aspect of) the input.

```
void print(List<Integer> l) {
    for (int i : l) {
        System.out.println(i);
    }
}
```

About how long does this take to run? Again it depends. One would probably say "proportional to the length of l", though.

Early returns

What about if the loop can return early?

```
boolean containsOne(int[] a) {
  for (int i: a) {
    if (i == 1) return true;
  }
  return false;
}
```

Like before, the runtime here varies upon the input. But in a new and excitingly different way! When do we exit the loop? Who knows?!?

Since we can't know, we generally concern ourselves with the *worst case*, that is, what's the longest this loop could run?

Answer: it's a function of the length of a, again. Again, About 2 * `a.length`, given our previous analysis.

(The other kind of analysis we might do is an "average case" analysis, but we'll mostly leave that for COMPSCI 311.)

Also, just because you see a loop, it doesn't mean that a method runs in non-constant time. For example:

```
int sumFirstThree(int[] a) {
  int sum = 0;
  for (int i = 0; i < 3; i++) {
    sum += a[i];
  }
  return sum;
}
```

...runs in constant time. Nor does there have to be an array in the parameter list (or as an instance variable, etc.) to trigger non-constant time behavior.

Nested loops

Now consider the case of `for` loops within `for` loops. Suppose we had an algorithm for duplicate detection that looked like this:

```
boolean containsDuplicate(int[] a) {} {
  for (int i = 0; i < a.length; i++) {
    for (int j = 0; j < a.length; j++) {
      if (i == j) continue;
      if (a[i] == a[j]) return true;
    }
  }
  return false;
}
```

How does this algorithm operate? (On board.)

How long does it take to run? For each iteration of the outer loop, we have to go through the entire inner loop. So we have to run a.length * (cost of inner loop) * a.length.

This method's runtime is a function of its input, but it's no longer a linear (first-degree polynomial) function; it's "quadratic" – that is, it's runtime is proportional to a.length *squared*.

That's a *lot* worse, especially as a.length grows. Who cares, right? Computers are fast? 3 GHz = 3 billion operations a second, right?

Well, what if we're working with a big array? Say, a million elements? 10 ns each is only 10 ms total to run. Something that runs in time proportional to the array length will be manageable. What about quadratic? 1,000,000 x 1,000,000 = 1,000,000,000,000. That's a lot of zeroes! Even if each step only takes, say, 10 ns, we're still talking about 10,000 seconds (nearly three hours) to complete!

So generally, when we write methods or call them, and we suspect that they're going to be used with large inputs, we should be thinking about how much time they'll take to run. Many efficient algorithms are *linear* in the size of their input, though some are a little worse, and some are much worse.

OK, Marc, but we don't need to go through the entire array inside the inner loop; we could just go through "what's left" at the end, since we've already checked everything there, right?

Quadratic or not?

```
boolean containsDuplicate(int[] a) {} {
  for (int i = 0; i < a.length; i++) {
    for (int j = i + 1; j < a.length; j++) {
      if (a[i] == a[j]) return true;
    }
  }
  return false;
}
```

Well, again, how many steps does the inner loop take? It's not always a.length, but it's a function of a.length – the first time through, it is a.length - 1; the next time a.length - 2, and so on, down to 3, 2, 1, 0. What's that proportional to? It's *still a function of a.length* (about 1/2). Here's an illustration (on board), or you can run the sums if you like.

Asymptotes

We've been talking about running time (mostly) on the basis of the degree of the polynomial (linear, quadratic, etc.). But remember that each variable has a coefficient, and that depending upon your value of n and the coefficients, linear-time algorithms are not strictly better than quadratic. For a silly example, consider which is faster:

- an algorithm with runtime of 1,000,000 n
- an algorithm with runtime (n^2)/ 1,000,000

For "large enough" values of n, the first algorithm is faster. How large? (set them equal to one another and solve for n) "Large enough" in this case is 10^12. Normally the coefficients aren't quite this lopsided, but it is true in practice that sometimes, a small-coefficient quadratic algorithm is faster than a larger-coefficient (but better polynomial) algorithm for small but reasonable values of n.

Implementations matter

How about the following?

```
boolean allEven(List<Integer> list) {
  for (int i = 0; i < list.size(), i++) {
    if (list.get(i) % 2 == 1) return false;
  }
  return true;
}
```

Answer? *It depends.* This is where understanding *how* the implementation (`ArrayList`? `LinkedList`? Something else?) underneath a given abstraction works *matters*.

If you're using an `ArrayList`, this will be linear, just as when it was for an array. But remember that to get to the `i`th element in a linked list, you have to traverse the list. So if we're using a linked list, each time we call `get` here, we are doing something that's dependent upon the length of the list. So it will be quadratic!

To further muddle this mess, the enhanced for loop:

```
boolean allEven(List<Integer> list) {
  for (int i: list) {
    if (i% 2 == 1) return false;
  }
  return true;
}
```

is actually smart enough to "remember" where it was the next time through, so it *won't* be quadratic. But you wouldn't know this unless you knew how lists and their iterators were implemented. Take 187! :)

But, usually the Java Docs will help you here. Take a look at ArrayList to see that:

> The size, isEmpty, get, set, iterator, and listIterator operations run in constant time. The add operation runs in amortized constant time, that is, adding n elements requires O(n) time. All of the other operations run in linear time (roughly speaking). The constant factor is low compared to that for the LinkedList implementation.

Compare with the LinkedList:

All of the operations perform as could be expected for a doubly-linked list. Operations that index into the list will traverse the list from the beginning or the end, whichever is closer to the specified index.

I guess you need to know what "as could be expected" means. Again, take 187 to be a better programmer!

Another example of implementation mattering

Remember that some things can be computed in different ways.

For example, to compute the sum of numbers from 1 to n,

- Sum from one to n, or
- compute n * (n + 1) / 2

```
int sumTo(int n) {
  int sum = 0;
  for (int i = 0; i <= n; i++) {
    sum += n;
  }
  return sum;
}
```

vs

```
int sumTo(int n) {
  return (n * (n + 1)) / 2;
}
```

Different algorithms *that accomplish the same goal* can have different running time behaviors.

Last modified: Sunday, March 13, 2022, 9:03 AM

◀ Lecture notes 07-01

Jump to...

COMPSCI186-SEC01 Using Data Structures Spring 2022

Lecture notes 08-01

Analyzing search, linear and binary

Let's look at one particular method on lists: `indexOf`. `indexOf` *searches* a list for an element, and returns its index (or -1) if not found. How long must a search take?

Well, knowing nothing else, we have to check every element of the list (or array, etc.). So? It's linear, right? Something like:

```
public int indexOf(String s, String[] array) {
    for (int i = 0; i < array.length; i++) {
        if (s.equals(array[i])) {
            return i;
        }
    }
    return -1;
}
```

Linear. But (and this is a big but and I cannot lie) if we know something more about the list, we can leverage that to not have to search the whole list.

For example, if the list is *sorted*. You know, like a telephone book, or a dictionary, or an index at the back of a book, or your phone's address book, or basically anything that's long and linear but where we want fast access to an arbitrary entry.

From Downey §12.8:

> When you look for a word in a dictionary, you don't just search page by page from front to back. Since the words are in alphabetical order, you probably use a binary search algorithm:
>
> - Start on a page near the middle of the dictionary.
> - Compare a word on the page to the word you are looking for. If you find it, stop.
> - If the word on the page comes before the word you are looking for, flip to somewhere later in the dictionary and go to step 2.
> - If the word on the page comes after the word you are looking for, flip to somewhere earlier in the dictionary and go to step 2.
>
> If you find two adjacent words on the page and your word comes between them, you can conclude that your word is not in the dictionary.

We can leverage this to write a faster search algorithm, called "binary search". It's called this because each time through the loop, it eliminates *half* of the possible entries, unlike a regular linear search that eliminates only one. It looks like this:

```
// note: only works on sorted arrays!
static int binarySearch(int i, int[] a) {

    int low = 0;
    int high = a.length - 1;

    while (low <= high) {
        int mid = (low + high) / 2;
        if (a[mid] == i) {
            return mid;
        } else if (a[mid] < i) {
            low = mid + 1;
        } else { // a[mid] > i
            high = mid - 1;
        }
    }
    return -1;
}
```

How long does this take to run?

Each time through the loop, we cut the distance between low and high in half. After k iterations, the number of remaining cells to search is `array.length` / 2^k. To find the number of iterations it takes to complete (in the worse case), we set array.length / 2^k = 1 and solve for k. The result is log_2 array.length. This is *sub-linear*. For example, for an array of 1,000 elements, it's about 10; a million elements, about 20; a billion elements, about 30, and so on.

The downside, of course, is that we have to keep the array sorted. How do we do that? (And more generally, we need things sorted all the time. How do we do that?)

Sorting

The binary search algorithm is a wonderful way to find an element in a list in less than n steps (log_2 n), even in the worst case, so long as the list is sorted.

Many operations can be performed more quickly on a sorted data set. Not to mention people often like to view sorted, rather than unsorted data (think about spreadsheets, indices, address books, etc.).

We're next going to turn our attention to three (well, two reasonable ones and one not-so-reasonable one) *sorting algorithms*, methods for transforming unsorted lists or array into sorted ones. We'll be using *comparison-based* sorts, where elements must be directly comparable (in Java: `Comparable`, or of primitive type that's naturally ordered). There are other approaches that you'll learn about in COMPSCI 311, like the radix sort.

We'll pay particular attention to running times of these algorithms, but also think a little about space requirements, access requirements (e.g., random access, like an array), and behavior in multiple cases (not just worst case, but perhaps best or average case). Again, more to come in 187.

A note

We'll think about runtimes in terms of swaps and comparisons.

We care about swaps as they are the basic way to reorder elements in an indexed list, like an array or `ArrayList`. (Note that some algorithms can be made to work on other abstractions.) A swap in an array or array-like structure usually requires small constant amount of space (equal to one element) to hold the swap variable, and a small constant amount of time:

```
public static void swap(int[] array, int i, int j) {
  int t = array[i];
  array[i] = array[j];
  array[j] = t;
}
```

As for comparisons: in the code I show, I'll use arrays of `int`s, because it's shorter. But in general, you could sort arrays of objects (that have an ordering) using `element.compareTo(other) < 0` rather than say `element < other`, or by instantiating and using an appropriate `Comparator`.

A first sorting algorithm: selection sort

It turns out that, like skinning a cat, there's more than one way to sort. See: https://en.wikipedia.org/wiki/Sorting_algorithm and https://www.toptal.com/developers/sorting-algorithms for example.

One way is to find the first (say, smallest) thing, and put it in the first position. Then find the next-smallest thing, and put it in the second position. Then find the third, and so on.

We find the thing using a simple linear search.

If we "put it in the ith position" using a swap, we don't need an entire list's (O(n)) worth of extra space, just a single element.

(on board with list 5 3 7 1)

This is called *selection sort*, because we *select* each element that we want, one-by-one, and put them where we want them.

```
static int indexOfMinimum(int[] array, int startIndex) {
  int minIndex = startIndex;
  for (int i = startIndex + 1; i < array.length; i++) {
    if (array[i] < array[minIndex]) {
      minIndex = i;
    }
  }
  return minIndex;
}

static void selectionSort(int[] array) {
  for (int i = 0; i < array.length - 1; i++) {
    // you could just:
    swap(array, i, indexOfMinimum(array, i));
    // but for maximum efficiency, instead:
    // int index = indexOfMinimum(array, i);
    // if (index != i) {
    //   swap(array, i, index);
    // }
    // this second approach avoids wasting time swapping
    // an element with itself, but at the cost of a if statement;
    // irrelevant optimization for asymptotic analysis though
  }
}
```

Let's also add some printing code to see this in action:

```
static void printArray(int[] array) {
  for (int i: array) {
    System.out.print(i + " ");
  }
  System.out.println();
}

public static void main(String[] args) {
  int[] array = new int[] {5, 3, 7, 1};
  printArray(array);
  selectionSort(array);
}
```

(We can add `printArray` to the search method's loop to see it work.)

Analyzing selection

How good or bad is selection sort, really? Let's think about comparisons and swaps.

There are exactly *n-1* comparisons the first time, *n-2* the second time, and so on. This sums up to *n(n-1)/2*.

There are exactly *n-1* swaps made (note that some could be no-ops, if an element `i == minIndex(array, i)` in other words, if it's in the right place already).

If comparisons and swaps are both about constant cost, then this algorithm is O(n^2) – the cost is dominated by the comparisons.

Even so, if swaps are much more expensive (a bigger constant) and n is not too large, then selection sort could be OK, since it bounds the number of swaps to be at most (n-1). But this case would be very unusual! Usually we just think about what's asymptotically most expensive, in this case, O(n^2) comparisons.

Bubble sort

Here's another sorting algorithm:

Start at the end of the list (element n-1). Then compare against the previous element (n-2). If the element at (n-1) is smaller, swap it with the element at (n-2). Then compare the element at (n-2) with the element at (n-3). And so on, all the way to the 0th element.

This will move the smallest element to the zeroth index of the list.

Now repeat, but stop at the 1st element. This will move the second-smallest element to the 1st index. Then repeat again, and so on, until the list is sorted.

Each time the algorithm repeats, the ith smallest element "bubbles up" to the front of the list; this is called a *bubble sort*.

```
static void bubbleUp(int[] array, int stopIndex) {
  for (int i = array.length - 1; i > stopIndex; i--) {
    if (array[i] < array[i - 1]) {
      swap(array, i, i - 1);
    }
  }
}

static void bubbleSort(int[] array) {
  for (int i = 0; i < array.length; i++) {
    bubbleUp(array, i);
  }
}
```

(on board with list 5 3 7 1)

Insertion sort

Now let's turn our attention to another sorting algorithm. This one is similar to how you might sort a handful of cards. Or maybe if you've ever volunteered at a library, it's kinda like how you might sort the books on a cart.

We break the hand up into two parts, sorted and unsorted. Then we add cards one-by-one from the unsorted part into the sorted part. (on board)

Let's say we start with our old friend 5 3 7 1, an unsorted list on the right, and a sorted (empty) list on the left:

```
| 5 3 7 1
```

"insert" the first card on the left of the unsorted array into the sorted array:

```
5 | 3 7 1
```

(Note we didn't actually do anything, just moved the index dividing the sorted part from the unsorted part). Now take the next element, 3.

```
5 3 | 7 1
```

We have to move it into the correct position by successively swapping it left until it's no longer smaller than its predecessor (or until there is no predecessor).

```
3 5 | 7 1
```

7 is easy:

```
3 5 7 | 1
```

and finally we need to successively move 1:

```
3 5 7 1 |
3 5 1 7 |
3 1 5 7 |
1 3 5 7 |
```

and we're done. This is called *insertion sorting*, since we take elements one one-by-one from the unsorted portion of the list and *insert* them into the sorted portion. Notice that when we're doing the "inserting" into the sorted position, it's exactly the same algorithm as our implementation of `ArrayList.add()` earlier in the semester used – we need to "move everything out of the way". The main difference here is we do both the moving and the finding-the-right-spot in the same loop, rather than being "given" the right spot (`i`) as a parameter to the `add()` method.

```
static void insertIntoSorted(int[] array, int sortedBoundary) {
  for (int i = sortedBoundary; i > 0; i--) {
    if (array[i] < array[i - 1]) {
      swap(array, i, i - 1);
    }
    else break; // you could omit this, but then you'd lose some non-worst-case performance
  }
}

static void insertionSort(int[] array) {
  for (int i = 1; i < array.length; i++) {
    insertIntoSorted(array, i);
  }
}
```

I'm going to ask you about the worst case in the problem set. But it turns out to be very fast in two particular cases:

- the constant factor for insertion sort is generally lower than some of the other O(n log n) algorithms you'll learn about later in 187, like merge sort and heap sort. Usually for n between 8 and 20 or so insertion's O(n^2) will outperform merge O(n log n) or quick sort (another n^2 sort that's got really good average-case performance) on a modern CPU.
- best case is an already sorted list: exactly n-1 comparisons and no swaps; partially-sorted lists are O(nk) where each element is no more than k from where it should be

Other sorting things

There are other sorting algorithms that can do better than n^2; the best algorithms run in "n log n" time ("mergesort" and "heapsort" are two you'll see in 187). They have tradeoffs, though, either requiring more than constant space or a higher constant factor (coefficient) than a simple sort like insertion sort.

In practice, most library `sort` methods, like `Arrays.sort` and `Collections.sort`, use a hybrid of the approaches, using the best algorithm for the task. Most common is the timsort, named after a guy named Tim (no joke!) Peters who first implemented it in the Python standard library.

There is also the legendary Bogosort, which will be familiar to you if you've ever played 52-card pickup.

Last modified: Tuesday, March 22, 2022, 5:07 PM

◀ Sample quiz answers

Jump to...

COMPSCI186-SEC01 Using Data Structures Spring 2022

Lecture notes 08-02

Graphs

Remember our discussion of trees, and how we talked about trees being a "kind" of graph? Graphs are this really useful thing, a kind of multipurpose data structure that let us represent so many things.

Notation

Recall G = {V, E}.

Edges can be undirected or directed – undirected connect vertices on both ends; directed have a directionality (from/to). Usually we speak of an entire *graph* being either directed or undirected.

Often, vertices and/or edges are annotated in some way. They might be named (typically vertices are named for convenience). Or they might have a value associated with them – often edges will, where the value is some notion of "cost" or "weight".

Vocabulary

Linear lists and trees are two ways to make objects out of *nodes* and *connections* from one node to another. There are many other ways to do it.

A *graph* consists of a set of nodes called *vertices* (one of them is a vertex) and a set of *edges* that connect the nodes. In an *undirected* graph, each edge connects two distinct vertices. In a *directed* graph, each edge goes from one vertex to another vertex.

Two vertices are *adjacent* (or *neighbors*) if there is an undirected edge from one to the other.

A *path* (in either kind of graph) is a sequence of edges where each edge goes to the vertex that the next edge comes from. A *simple* path is one that never reuses a vertex. In a *tree*, there is exactly one simple path from any one vertex to any other vertex.

A *complete* graph is one with every possible edge among its vertices – in other words, every vertex is adjacent to every other vertex.

A *connected component* is a set of vertices in which every vertex has a path to every other vertex (though not every vertex is adjacent).

A single graph might have two or more connected components! (on board)

Examples

maze

google maps

8-puzzle

tic-tac-toe

In-class thought experiment

Imagine you wanted to represent the first two years worth of COMPSCI courses (121, 186, 187, 220, 230, 240, 250) for majors (and their prerequisites) as a graph. What would it look like?

Graph abstraction and algorithms

Each of the previous examples can, if you squint at it correctly, be viewed as a graph. There is some "space" (finite or otherwise) of discrete points, and some points are connected to others.

This corresponds exactly to a graph. And what's interesting here is that there are many algorithms that operate generally on graphs, regardless of the underlying problem. So we can write them (once) and solve many kinds of problems. Most common are things like:

- search: start at a particular vertex, report true if there is a path to another vertex in the graph, or false otherwise)
- path search: (also shortest-path search) find the shortest path from one vertex to another vertex (this might be lowest number of edges, or if edges have a "weight", might be based upon sum of edge costs)
- minimax search in an adversarial game: given a state, look for the "winning-est" move
- all-pairs shortest path (which it turns out can be solved more efficiently than just doing each pairwise shortest-path search)
- min-cut: what's the fewest number of edges you can remove to partition the graph into two disjoint components (useful in disruption-tolerance algorithms)

and many, many more.

Total vs partial knowledge of a graph

You can know (or compute) the entire graph "ahead of time" when it's both small and knowable, for example, our earlier maze example. That is, you can create an ADT that allows you to set and access nodes and edges (and associated annotations) and instantiate it.

For some problems, the graph is too large to keep track of (e.g., the state space of chess is around 10^123). But obviously we have computer programs that can play chess. How do you do it? You generate a "partial view" of the state space, where you can find the "successors" of a particular state (on board) and their successors, and so on, up until you're "out of time" to think more or out of space to store more, and do the best you can with this partial view.

How might these two kinds of ADTs – total vs partial – look in practice?

ADT for graphs

We need to be able to add and query lists (or sets) of vertices and edges. Of course, edges are just links between two vertices, so we needn't have a separate data type. What might this look like in the simple case, where we don't worry about annotations? Something like:

```
public interface UndirectedGraph<V> {
  void addVertex(V v);
  boolean hasVertex(V v);
  Set<V> vertices();

  void addEdge(V u, V v);
  boolean hasEdge(V u, V v);
  Set<V> neighborsOf(V v);
}
```

What about if we are just concerned with a partial view of the graph? Maybe something like this:

```
public interface PartialUndirectedGraph<V> {
  List<V> neighborsOf(V v);
}
```

The implementation of a partial view would have to know quite a bit about the underlying problem in order to generate the neighbors, but on the other hand, you don't need to generate *everything*, just the bits of the graph you care about.

Searching a graph

How might we go about trying to solve a search problem? That is, suppose we had a graph that had been instantiated a particular way. We're given a start vertex, and we want to see if there's a path in that graph to the end vertex. As a human, we can just look at a small graph and decide, but larger graphs eventually can't just be glanced at. What's a methodical way to check?

Let's work through an example:

(on board, graph S,1,2,3,4,G where 1,2,3 are strongly connected and 4 is only connected to 3).

The idea behind searching a graph is that we want to systematically examine it, starting at one point, looking for a path to another point. We do so by keeping track of a list of places to be explored (the "frontier"). We repeat the following steps until the frontier is empty or our goal is found:

- Pick and remove a location from the frontier.
- Mark the location as explored (visited) so we don't "expand" it again.
- "Expand" the location by looking at its neighbors. Any neighbor we haven't seen yet (not visited, not already on the frontier) is added to the frontier.

We need to be able to add and query lists (or sets) of vertices and edges. Of course, edges are just links between two vertices, so we needn't have a separate data type. What might this look like in the simple case, where we don't worry about annotations? Something like:

```
public interface UndirectedGraph<V> {
  void addVertex(V v);
  boolean hasVertex(V v);
  Set<V> vertices();

  void addEdge(V u, V v);
  boolean hasEdge(V u, V v);
  Set<V> neighborsOf(V v);
}
```

The idea behind searching a graph is that we want to systematically examine it, starting at one point, looking for a path to another point. We do so by keeping track of a list of places to be explored (the "frontier").

We start by marking the start location as seen and adding it to the frontier.

We then repeat the following steps until the frontier is empty or our goal is found:

- Pick and remove a location from the frontier.
- "Expand" the location by looking at its neighbors. Any neighbor we haven't seen yet is added to the frontier; mark those locations as seen as you add them to the frontier.

Coding the search algorithm

What might this look like in code? (NOTE: This is slightly different than the in-class code – I will go over the changes next class!)

```
static <V> boolean isPath(UndirectedGraph<V> graph, V start, V goal) {
  Queue<V> frontier = new LinkedList<>();
  frontier.add(start);

  Set<V> seen = new HashSet<>();
  seen.add(start);

  while (!frontier.isEmpty()) {
    V current = frontier.remove();
    if (current.equals(goal)) return true;
    for (V next : graph.neighborsOf(current)) {
      // note: could put check for goal here instead;
      // if so, we need to be careful of the case where start == goal
      // we'd have to check for that outside the loop
      if (!seen.contains(next)) {
        frontier.add(next);
        seen.add(next);
      }
    }
  }
  return false;
}
```

Note I used a `Queue` here; this first in, first-out behavior enforces a breadth-first search. (on board) `Queue`s are lists, but you can only add on one end, and only remove from the other; like waiting in line at Disney or some such. (You could also use a `List` if you wanted to, but how you add and removes vertices from the frontier controls how the search runs.)

Using a `Queue`, this search will visit all vertices adjacent to the start (that is, one hop away from the start) before it visits their neighbors (two hops away from the start), and so on, like ripples in a pond. This is called a "breadth-first" search.

Depending upon the order of vertices returned from our `frontier`, the search will progress in different ways; most notably is a depth-first when the frontier is a stack – last in, first out. You'll see this in more detail in 187.

Last modified: Saturday, March 26, 2022, 11:33 AM

◀ Lecture notes 08-01

Jump to...

Programming assignment 08: Image mosaic ▶

© University of Massachusetts Amherst • Site Policies • Site Contact

Moodle Help for Students • Moodle Help for Instructors

You are logged in as Tanish Patel (Log out)
COMPSCI186-SEC01 SP22

Get Help
Help for Students
Help for Instructors

COMPSCI186-SEC01 Using Data Structures Spring 2022

Lecture notes 09-01

Recall

To search a graph, we need to be able to represent it.

We need to be able to add and query lists (or sets) of vertices and edges. Of course, edges are just links between two vertices, so we needn't have a separate data type. What might this look like in the simple case, where we don't worry about annotations? Something like:

```
public interface UndirectedGraph<V> {
  void addVertex(V v);
  boolean hasVertex(V v);
  Set<V> vertices();

  void addEdge(V u, V v);
  boolean hasEdge(V u, V v);
  Set<V> neighborsOf(V v);
}
```

The idea behind searching a graph is that we want to systematically examine it, starting at one point, looking for a path to another point. We do so by keeping track of a list of places to be explored (the "frontier").

We start by marking the start location as seen and adding it to the frontier.

We then repeat the following steps until the frontier is empty or our goal is found:

- Pick and remove a location from the frontier.
- "Expand" the location by looking at its neighbors. Any neighbor we haven't seen yet is added to the frontier; mark those locations as seen as you add them to the frontier.

Coding the search algorithm

What might this look like in code?

```
static <V> boolean isPath(UndirectedGraph<V> graph, V start, V goal) {
  Queue<V> frontier = new LinkedList<>();
  frontier.add(start);

  Set<V> seen = new HashSet<>();
  seen.add(start);

  while (!frontier.isEmpty()) {
    V current = frontier.remove();
    if (current.equals(goal)) return true;
    for (V next : graph.neighborsOf(current)) {
      // note: could put check for goal here instead;
      // if so, we need to be careful of the case where start == goal
      // we'd have to check for that outside the loop
      if (!seen.contains(next)) {
        frontier.add(next);
        seen.add(next);
      }
    }
  }
  return false;
}
```

Note I used a `Queue` here; this first in, first-out behavior enforces a breadth-first search. (on board) `Queue`s are lists, but you can only add on one end, and only remove from the other; like waiting in line at Disney or some such. (You could also use a `List` if you wanted to, but how you add and removes vertices from the frontier controls how the search runs.)

Using a `Queue`, this search will visit all vertices adjacent to the start (that is, one hop away from the start) before it visits their neighbors (two hops away from the start), and so on, like ripples in a pond. This is called a "breadth-first" search.

Depending upon the order of vertices returned from our `frontier`, the search will progress in different ways; most notably is a depth-first when the frontier is a stack – last in, first out. You'll see this in more detail in 187.

Finding the path

`isPath` doesn't actually find the path, it just checks to see if there is one.

One way to find the path is to change `seen` slightly.

Instead of keeping track only of whether or not a vertex has been seen, we can keep track of where we "came from" to get to that vertex. In other words, we can track the "predecessor" of that vertex. (on board)

Here's the updated code:

```
static <V> List<V> findPath(UndirectedGraph<V> graph, V start, V goal) {
  Queue<V> frontier = new LinkedList<>();
  frontier.add(start);


  Map<V, V> predecessor = new HashMap<>();
  predecessor.put(start, null);

  List<V> path = new ArrayList<>();

  while (!frontier.isEmpty()) {
    V current = frontier.remove();

    // have we found the goal?
    if (current.equals(goal)) {
      // if so, reconstruct the path
      path.add(current);
      V previous = predecessor.get(current);
      while (previous != null) {
        path.add(0, previous);
        previous = predecessor.get(previous);
      }
      // and exit the loop / search
      break;
    }

    // otherwise, continue searching
    for (V next : graph.neighborsOf(current)) {
      if (!predecessor.containsKey(next)) {
        frontier.add(next);
        predecessor.put(next, current);
      }
    }
  }
  return path;
}
```

As before, we could do the goal check inside the inner `for` loop to save a few frontier expansions; I broke it out here to make it more clear, but either way works.

OK, great! What does this look like generally? Again, we search each vertex one hop away before we get to *any* of the vertices two hops away, and so on. This behavior, the choice of which vertices to search, is entirely a function of how we store and return vertices from the frontier. When it's a queue, we get this "breadth-first", ripples-in-a-pond behavior. You can imagine the form of the search a tree, where each level of the tree is the distance, in hops, from the start node. We search this tree level-by-level in a breadth first search. (on board)

Depth first search (DFS)

The other way to search a graph is "depth-first" search, where we fully explore one branch before backtracking to the next. (on whiteboard)

What does the code look like? There's a couple ways to structure it. One way, that you explored in lab, is recursive; the other is iterative. Both use a stack – the recursive method uses the call stack, while the iterative method uses an explicit stack to perform the search:

```
static <V> List<V> findPathDFS(UndirectedGraph<V> graph, V start, V goal) {
    Stack<V> frontier = new Stack<>();
    frontier.push(start);

    Map<V, V> predecessor = new HashMap<>();
    predecessor.put(start, null);

    List<V> path = new ArrayList<>();
    while (!frontier.isEmpty()) {
        V current = frontier.pop();
        if (current.equals(goal)) {
            path.add(current);
            V previous = predecessor.get(current);
            while (previous != null) {
                path.add(0, previous);
                previous = predecessor.get(previous);
            }
            break;
        }

        for (V next: graph.neighborsOf(current)) {
            if (!predecessor.containsKey(next)) {
                frontier.push(next);
                predecessor.put(next, current);
            }
        }
    }
    return path;
}
```

Notice the only code changes to support the use of a stack, rather than a queue, to hold the frontier.

BFS vs DFS: What kinds of paths are found?

BFS always finds (one of) the paths to the goal that has the shortest number of edges, since it always searches all paths of n edges before searching paths of n+1 edges. But it requires that you remember the entire search tree as you go.

DFS, interestingly, does not need to remember the entire tree; only the vertices along a path along with their neighbors. Once a vertex has been seen it can be forgotten, if you're willing to do some minor bookkeeping (less than is required in BFS, which requires tracking every previously seen node). But DFS might not find the shortest path path. Remember that if this were a depth-first search, we'd search the tree *as far as possible* down one path before backtracking. (on board)

Again, to get this DFS, behavior, all we need to do is switch the queue to a stack. (Correctly "forgetting" nodes to keep space requirements low is more complex code-wise to implement than just switching to a stack, though.)

Informed search

Both breadth- and depth-first search are said to be "uninformed" search. That is, they explore the frontier systematically, but with no knowledge of where the goal is relative to the current position. There's only so much you can do to optimize them (there's a hybrid algorithm called "iterative deepening DFS" that sorta gets you the best of both worlds; you might analyze this in more depth in 311 or 383).

But if you *know something* about the problem domain, you can do better.

For example, suppose you have a graph where the vertices represent places (say, on campus), and the edges represent paths between those places. Each edge has a cost associated with it (say, the distance), and you're trying to find a least-cost (aka shortest) path between a start and a goal.

If you just pick edges according to BFS, you'll find the shortest path number-of-edges-wise but not cost-wise. How might we do better?

Well, one way would be to *order* the frontier, from least-cost to highest-cost, and examine vertices in order from least-to-highest cost. Of course, we are probably working with estimates, since if we really knew the true cost, it wouldn't be a search: we'd just follow the least-coast path like a homing missile.

How do estimate their costs? We say that each vertex's cost is defined by a function f(x).

One definition for f(x) is a heuristic, say h(x). A heuristic is an "approximation" or guess of a true value. In our campus graph example, a heuristic might be the straight-line distance from the vertex in question to the goal; usually, paths are roughly straight lines, though of course buildings or the campus pond might make this heuristic not-quite-correct. We can compute this by looking at the map.

So, one approach is to do a "greedy" search, where we always choose the closest node. How can we do this? We could *sort* the frontier after each iteration of the loop, which would require time proportional to about (n log n), where n is the number of vertices, if we used an efficient sorting algorithm. And that's fine. It would produce a "priority queue," that is, a queue that returns things not in first-in, first-out order, but "best-out" order.

It turns out Java implements this for us.

Priority queues

The `PriorityQueue` will act exactly as we want, allowing items to be added in arbitrary order, and returning them in lowest-cost order. Priority queues internally are not a `List`, but instead a heap. We won't implement heaps in this course (wait for it... but you will in 187!). Heaps are "not-quite-sorted"; they maintain another property (the "heap property") which lets them remove and return the current smallest item in (log n) time, and add new items in (log n) time.

In any event, we need to define an ordering on items to create a useful priority heap. Just like we've seen several times before, when there's additional context we need to compare two items (for example, in our campus navigation example, we'd need to know about the map, not just the location), we can define a `Comparator` to hold this additional state and use it in its `compare` method. This `Comparator` gets passed to the `PriorityQueue` constructor, and then we have a "greedy" search. This is basically a one-line change to the method, just like going from BFS to DFS.

```
static <V> List<V> findPath(UndirectedGraph<V> graph, V start, V goal,
    Comparator<V> comp) {
  Queue<V> frontier = new PriorityQueue<>(comp);
  frontier.add(start);

  Map<V, V> predecessor = new HashMap<>();
  predecessor.put(start, null);

  List<V> path = new ArrayList<>();

  while (!frontier.isEmpty()) {
    V current = frontier.remove();
    for (V next : graph.neighborsOf(current)) {
      if (!predecessor.containsKey(next)) {
        frontier.add(next);
        predecessor.put(next, current);
      }
    }
    if (current.equals(goal)) {
      path.add(current);
      V previous = predecessor.get(current);
      while (previous != null) {
        path.add(0, previous);
        previous = predecessor.get(previous);
      }
      break;
    }
  }
  return path;
}
```

Defining the comparator is problem-specific; we can assume it's been passed in as above.

Greedy search can "get it wrong" and find a sub-optimal path, especially if the heuristic is inaccurate. As you'll learn in later courses, an optimal informed search algorithm is called "A*", and its f(x) = g(x) + h(x). g(x) is just the known lowest-cost to get to vertex x so far; h(x) is a heuristic that must obey certain conditions – an "admissible" heuristic. Again, you'll see this in future courses.

Last modified: Wednesday, March 30, 2022, 3:08 PM

◀ image-mosaic-student.zip

Jump to...

Lecture notes 09-02 ▶

© University of Massachusetts Amherst • Site Policies • Site Contact

Moodle Help for Students • Moodle Help for Instructors

You are logged in as Tanish Patel (Log out)
COMPSCI186-SEC01 SP22

Get Help
Help for Students
Help for Instructors

COMPSCI186-SEC01 Using Data Structures Spring 2022

Lecture notes 09-02

Review of Graph Representations

Recall that we can represent the Graph ADT in any implementation we want: in this course, we'll (briefly) sketch the Adjacency Matrix and Adjacency List implementations.

Consider a (very) simplified graph, where V = {0, 1, 2, ... n-1}.

The adjacency matrix representation just creates an n x n 2D array of booleans, representing the edge from-to relationship. A given entry in the array is `true` iff there exists an edge from-to the corresponding indices of the array.

The adjacency list representation is an array of lists. The array is n elements long; each element points to a list of outgoing edge destination corresponding to that element's edges (or an empty list, if it has no outgoing edges).

(on board)

Implementing the abstraction

Next we're going to talk about how you'd implement those interfaces.

Here's a naive implementation:

```java
public class AdjacencyMatrixUndirectedGraph<V> implements UndirectedGraph<V> {

    private List<V> vertices;
    private final boolean[][] edges;

    public AdjacencyMatrixUndirectedGraph(int maxVertices) {
        vertices = new ArrayList<>();
        edges = new boolean[maxVertices][maxVertices];
    }


    @Override
    public void addVertex(V v) {
        // what if the vertex is already in the graph?
        vertices.add(v);
    }

    @Override
    public boolean hasVertex(V v) {
        return vertices.contains(v);
    }

    @Override
    public Set<V> vertices() {
        return new HashSet<>(vertices);
    }

    @Override
    public void addEdge(V u, V v) {
        // order of edges?
        // u,v in graph?
        edges[vertices.indexOf(u)][vertices.indexOf(v)] = true;
    }

    @Override
    public boolean hasEdge(V u, V v) {
        // order of edges?
        // u,v in graph?
        return edges[vertices.indexOf(u)][vertices.indexOf(v)];
    }

    @Override
    public Set<V> neighborsOf(V v) {
        // order of edges?
        // v in graph?
        Set<V> neighbors = new HashSet<>();
        int index = vertices.indexOf(v);
        for (int i = 0; i < vertices.size(); i++) {
            if (edges[index][i]) {
                neighbors.add(vertices.get(i));
            }
        }
        return neighbors;
    }
}
```

Note that upon reflection, there are some problems here (repeated vertices! order of vertices in edges! are vertices even in the graph?). Some of this we can fix in code (by having, say, a canonical ordering, or being sure to set both spots in the matrix); some of this implies we need to add to our API (methods that take arbitrary vertices as parameters should throw an exception).

Why adjacency matrices?

Remember, the main advantage of adjacency matrices is that they're lightning fast in terms of checking if an edge is in the graph; it's not just constant time, it's constant time with a very low constant. Except our crappy implementation above requires a call to `List.indexOf` first; so it's actually linear in the number of vertices. But a better-optimized version of an adjacency matrix representation of a graph would not do this (it would instead use just `int`s for vertices) and would be "supah-fast", that is, constant-time.

Adjacency lists

The main downside to adjacency matrices is that they consume a lot of space: the implementation above uses (maxVertices)^2 space, that is, space quadratic in the number of vertices. In the worst case, a graph actually needs this much space – an "almost-complete" graph is called a "dense" graph. But if most vertices are not connected to most other vertices, that is, if we have a "sparse" graph, a more efficient implementation is the adjacency list.

Let's write one now using our by-now old friend the `Map`:

```
public class AdjacenyListUndirectedGraph<V> implements UndirectedGraph<V> {
    Map<V, List<V>> adjacencyList;

    public AdjacenyListUndirectedGraph() {
        adjacencyList = new HashMap<>();
    }

    @Override
    public void addVertex(V v) {
        // duplicate vertex?
        adjacencyList.put(v, new ArrayList<>());
    }

    @Override
    public boolean hasVertex(V v) {
        return adjacencyList.containsKey(v);
    }

    @Override
    public Set<V> vertices() {
        // modification?
        return adjacencyList.keySet();
    }

    @Override
    public void addEdge(V u, V v) {
        // order?
        // u, v in adjacencyList?
        adjacencyList.get(u).add(v);
    }

    @Override
    public boolean hasEdge(V u, V v) {
        return adjacencyList.get(u).contains(v);
    }

    @Override
    public Set<V> neighborsOf(V v) {
    return new HashSet<>(adjacencyList.get(v));
    }
}
```

Again some problems here, including that we need to be careful of returning `Set`s that share structure with the graph. The caller might mutate the `Set`, and thus change the graph! If that's not what we want (and it usually isn't), then we should return *copies* of the structures that represent parts of the graph, not the original structures themselves.

Why adjacency lists?

Is this "slower" than an adjacency matrix? Yes. In particular, any time we need to iterate over the list (`contains`), we are, worst case, linear in the number of vertices. But we only need exactly as much space as is required to store each edge/vertex. In the worst case this is quadratic in the number of vertices, so we're no better off than an adjacency matrix. But in a sparse graph, we come out ahead space-wise. And, saying a graph is sparse is roughly equivalent to saying that each vertex has a small constant number of edges, so `contains` is usually OK in this case. (You'll explore this more in 311).

"But Marc," you might be thinking, "why not make it a `Map<V, Set<V>>` and get the best of both worlds?" You can! And you would (mostly!). But while hash lookups are constant time, they're not as quite as small a constant as array lookups. If you're really, really worried about speed, and space is not an issue, you may end up using the adjacency matrix representation anyway. But enough about that – the details of graph representation in data structures go deep, and this isn't a class about that. And frankly, just like lists, sets, and maps, if you are using graphs in a real-world problem, you likely are going to use a pre-written graph library that's optimized for your use case.

Last modified: Thursday, March 31, 2022, 6:25 PM

◀ Lecture notes 09-01

Jump to...

Sample quiz questions ▶

© University of Massachusetts Amherst • Site Policies • Site Contact

Moodle Help for Students • Moodle Help for Instructors

You are logged in as Tanish Patel (Log out)
COMPSCI186-SEC01 SP22

Get Help
Help for Students
Help for Instructors

COMPSCI186-SEC01 Using Data Structures Spring 2022

Lecture notes 10-01

Specialized linear ADTs

The "standard" linear ADTs (in Java) are the array and the (generic) `List`. Arrays are a simple type, with very fast random access but the limitation of fixed sizes. Lists are more flexible, and their underlying implementation is generally written in terms of (resizing) arrays or (sometimes) in terms of a linked list.

But as we've mentioned, there are other linear data structures that one might use; they are similar to lists but restrict themselves in various ways. We're going to revisit them so you're ready when you see them again ("for the first time") in 187. We'll start with behavior, then do implementations.

Stacks

Stacks are a last-in, first-out data structure. They are like lists, but instead of allowing for random access (via `get`, `remove`, `add`), they restrict a user to adding items on one end (the "top") and removing from that same position. These operations are sometimes called `push` (add an item), `pop` (remove an item), and `peek` (look at but do not remove the top item).

Modern Java suggests we use the `Deque` interface, which is short for *double-ended queue*, and use the `addFirst`, `removeFirst`, and `peekFirst` methods. In either case, though, the behavior is the same, LIFO.

```
s.push("a");
s.push("b");
s.pop();
s.push("c");
s.push("d");
s.peek();
```

(top on right)

- After the first operation, the stack contains ["a"]
- After the second, the stack contains ["a", "b"].
- removes and returns "b", then stack contains, ["a"]
- ["a", "c"]
- ["a", "c", "d"]
- `peek` returns "d"

Queues

Queues are a first-in, first-out data structure. Java has a `Queue` interface you can use, or you can (also) use `Deque`, as described in its documentation. In a `Queue`, we typically talk about `add` (always at one end) `remove` (always from the other), and sometimes `peek` (just like a stack, returns but does not remove the next element that `remove` would return).

```
q.add("a");
q.add("b");
q.remove();
q.add("c");
q.add("d");
q.peek();
```

(front on left, rear on right)

- ["a"]
- ["a", "b"]
- removes and returns "a", queue contains ["b"]
- ["b", "c"]
- ["b", "c", "d"]
- returns "b"

A side note: over/underflow

Stacks and queues can *underflow*. If you call `pop` or `remove` on an empty stack/queue, this will generate an exception.

Some stacks and queues are *bounded*, which means they have an explicit capacity. If you try to `push` or `add` to a stack/queue that is already at capacity, then you will *overflow* the structure and generate an exception.

Priority queues

A priority queue is like a queue, but it returns the next "smallest" (in Java) thing, rather than the first-in thing, when `remove` or `peek` is called.

It's important to note that the exact order of the items stored in the priority queue *is not visible* to the user; you can only see the "next" / "top" item (that will be returned by `peek` or `remove`). Internally, priority queues are implemented as "heaps", which are a tree-based structure similar to, but different from, the binary search trees we talked about briefly earlier this semester. Heaps allow for efficient (log n) insertion and removal of the smallest item.

How do we define "smallest"? The usual way, by either depending upon the "natural" ordering of the elements stored in the `PriorityQueue<E>` (that is, they must implement `Comparable`) or by passing in a `Comparator` when constructing the `PriorityQueue`.

Suppose then we do the following with a priority queue:

```
pq.add("b");
pq.add("a");
pq.remove();
pq.add("c");
pq.add("d");
pq.peek();
```

- ["b"]
- ["a", "b"]
- removes and returns "a", contents are ["b"]
- ["b", "c"]
- ["b", "c", "d"] ; note we don't know whether "c" or "d" comes first; all we know is "b" is up next to be removed
- returns "b"

Implementing a stack

The *stack*, a last-in, first-out data structure. How might we go about building one? Today we'll build two implementations of a stack, based upon the ADT specified by a Java interface.

As we go through this, see how well you can follow along. This is the first of several "barometer" tasks we'll be doing: if you find them easy (or at least, doable), then you should feel reasonable optimistic about 187 next semester. If you find them difficult or confusing, again, maybe have a fallback plan (or plan to study hard over the break, perhaps practicing on 186 assignments you had trouble with).

The interface

```
package linearadts;

public interface Stack<E> {
    void push(E e) throws StackOverflowException;
    E pop() throws StackUnderflowException;
    E peek() throws StackUnderflowException;
    boolean isEmpty();
    boolean isFull();
    int size();
}
```

This is a minimal interface; of course you can add more if you like. Note we include thrown exceptions, which is optional depending upon whether the exceptions are "checked" or "unchecked"; the former inherit from `Exception` and the latter from `RuntimeException`. See https://docs.oracle.com/javase/tutorial/essential/exceptions/runtime.html for more details.

We'll trivially define (empty) classes for each exception ourselves.

Array-based stack

Now let's look at an array-based implementation of a stack. We'll do a "bounded" stack, that is, a stack that has a fixed maximum capacity. (You can imagine an unbounded, growable array, like the one we talked about for lists earlier in the semester, but we'll skip that for now.)

What other state, that is, instance variables, does the stack require, other than an array? An index into that array, that indicates where the top of the stack "lives". You have some discretion here – it could index the current top of the stack, or it could index the spot where the next top will be. This is a stylistic choice, but I find pointing it to the current top (or -1, if the stack is empty) to be more natural. Up to you.

(on board)

```java
package linearadts;

public class BoundedArrayStack<E> implements Stack<E> {
    private E[] array;
    private int top;

    public BoundedArrayStack(int capacity) {
        top = -1;
        array = (E[]) new Object[capacity]; // references to this array must not escape this class
    }

    @Override
    public void push(E e) throws StackOverflowException {
        if (isFull()) {
            throw new StackOverflowException();
        }
        top += 1;
        array[top] = e;
    }

    @Override
    public E pop() throws StackUnderflowException {
        if (isEmpty()) {
            throw new StackUnderflowException();
        }
        E temp = array[top];
        array[top] = null;
        top--;
        return temp;
    }

    @Override
    public E peek() throws StackUnderflowException {
        if (isEmpty()) {
            throw new StackUnderflowException();
        }
        return array[top];
    }

    @Override
    public boolean isEmpty() {
        return top == -1;
    }

    @Override
    public int size() {
        return top + 1;
    }

    @Override
    public boolean isFull() {
        return top == array.length - 1;
    }

}
```

Note 1: You can't instantiate generic, typesafe arrays in Java, for historical and technical reasons that are beyond the scope of the course. Note the workaround: `array = (E[]) new Object[capacity];` works, but generates a warning. As long as no references to the array escape the enclosing object (that is, as long as we never return the `E[]` array to a caller) we'll be OK.

Note 2: When we pop, we explicitly set the array cell to `null`. Why? (on board) we are removing the reference (from the stack) to the object so that it can be garbage collected later. This is a minor potential memory leak but it's worth plugging.

Linked-list based stack

Recall there are two basic ways to agglomerate data in Java: you can use arrays, or you can use references. For this second implementation, let's consider using linked lists. Again, as we showed earlier in the semester, we'll need a simple `Node` structure to link together into the list.

```
package linearadts;

// note this is a super-discount Node class; a "real" one would probably have at
// least a constructor (that took an `E data` argument)
public class Node<E> {
    public Node<E> next;
    public E data;
}
```

And since we're only interested in the "top" of the stack, it's pretty easy to do.

```
package linearadts;

public class UnboundedLinkedStack<E> implements Stack<E> {
    private Node<E> head;
    private int size;

    @Override
    public void push(E e) throws StackOverflowException {
        Node<E> node = new Node<>();
        node.data = e;
        node.next = head;
        head = node;
        size++;
    }

    @Override
    public E pop() throws StackUnderflowException {
        if (isEmpty()) {
            throw new StackUnderflowException();
        }
        E temp = head.data;
        head = head.next;
        size--;
        return temp;
    }

    @Override
    public E peek() throws StackUnderflowException {
        if (isEmpty()) {
            throw new StackUnderflowException();
        }
        return head.data;
    }

    @Override
    public boolean isEmpty() {
        return head == null;
    }

    @Override
    public boolean isFull() {
        return false;
    }

    @Override
    public int size() {
        return size;
    }

}
```

Implementing queues using linked lists

Recall to implement a stack with linked lists, we used just a head pointer. This made sense, because we always manipulated the same end of the list.

To review, to add to the head of the list (like we just did for stacks), we did the following:

- create a new node
- set its next reference to the current head
- set head to the new node

To remove the head node, we did the following:

- remember the head node's data
- set head to it's next reference (now there are no refs into the former head node, so it's deleted)

Queues have to manipulate both ends of the list, so we'll need a head and tail pointer. Should we enqueue at the head or the tail? And where should we dequeue?

We know that both adding and removing at the head are pretty straightforward, since that's what we did with push/pop for stacks. What about adding or removing at the tail?

(On board). Suppose we keep a "tail" pointer which always points at the last node in the list. To add, we:

- create a new node
- set tail node's next point to this new node
- update tail to point at the new node

How do remove? We need a pointer to the node before the tail...we'd have to have both forward and backward pointers in the list (which then is a different data structure, a doubly-linked list), or to traverse the entire list to the node-before:

- traverse to node-before, keep reference to it
- remember the tail node's data
- set tail to the node-before

That's linear-time behavior, which is terrible if we don't need to do it.

So, what have we learned? Adding at the tail is trivial, but removing the tail in a singly-linked list is not (linear to seek to the node before the tail). So let's enqueue onto the tail, and dequeue from the head.

Last modified: Tuesday, April 5, 2022, 4:00 PM

◀ information-retrieval-student.zip

Jump to...

COMPSCI186-SEC01 Using Data Structures Spring 2022

Lecture notes 10-02

Implementing queues using linked lists

Recall to implement a stack with linked lists, we used just a head pointer. This made sense, because we always manipulated the same end of the list.

To review, to add to the head of the list (like we just did for stacks), we did the following:

- create a new node
- set its next reference to the current head
- set head to the new node

To remove the head node, we did the following:

- remember the head node's data
- set head to it's next reference (now there are no refs into the former head node, so it's deleted)

Queues have to manipulate both ends of the list, so we'll need a head and tail pointer. Should we enqueue at the head or the tail? And where should we dequeue?

We know that both adding and removing at the head are pretty straightforward, since that's what we did with push/pop for stacks. What about adding or removing at the tail?

(On board). Suppose we keep a "tail" pointer which always points at the last node in the list. To add, we:

- create a new node
- set tail node's next point to this new node
- update tail to point at the new node

How do remove? We need a pointer to the node before the tail...we'd have to have both forward and backward pointers in the list (which then is a different data structure, a doubly-linked list), or to traverse the entire list to the node-before:

- traverse to node-before, keep reference to it
- remember the tail node's data
- set tail to the node-before

That's linear-time behavior, which is terrible if we don't need to do it.

So, what have we learned? Adding at the tail is trivial, but removing the tail in a singly-linked list is not (linear to seek to the node before the tail). So let's enqueue onto the tail, and dequeue from the head.

What does the code look like? It's similar to the stack, but with the tail reference come a few extra points of complication.

```
package linearadts;

public interface Queue<E> {
    void add(E e);
    E remove(); // note: we would probably declare over/underflow in a "real" implementation
    E peek();
    boolean isEmpty();
    boolean isFull();
    int size();
}
```

```
public class LinkedQueue<E> {
    private Node<E> head;
    private Node<E> tail;
    private int size;

    public LinkedQueue() {
        // optional to initialized head and tail, as their default values are null
        // same for size = 0
    }
...
```

When we enqueue, we make a new node and add it to the rear of the queue. Again, how do we do this? By setting the tail's next node equal to the new node, and then setting the tail equal to the new node.

But there's a special case: if the queue is empty, our new node will be both the head and the tail. (on board)

```
    public void add(E e) {
        Node<E> node = new Node<>();
        node.data = e;

        if (isEmpty()) { // it's OK to use isEmpty here, because we're not yet changing the list.
            tail = node;
            head = node;
        } else {
            tail.next = node;
            tail = node;
        }
        size++;
    }
```

alternatively, a slightly shorter arrangement (though I think it's maybe less clear):

```
    public void add(E e) {
        Node<E> node = new Node<>();
        node.data = e;

        if (isEmpty()) {
            head = node;
        } else {
            tail.next = node;
        }
        tail = node;
        size++;
    }
```

Dequeuing is similar. We remove the head node from the list and return its value, just like popping from a stack.

But just like `add()`, there's a special case is if the node is the last node. If so, we need to remember to also set tail to null. (on board)

```
public E remove() {
    E value = head.data;
    head = head.next; // note this is ok for both cases; if the queue
                      // is down to one element, head.next == null

    if (size == 1)  { // is it OK to use isEmpty() here?
                           // We just changed the list.
                           // (Answer: It depends on your implementation of isEmpty().
                           // If it's just head == null, it's fine.
                           // But if it's size == 0, then maybe not;
                           // depends when you size--!)
        tail = null;
    }
    size--;
    return value;
}
```

I'm leaving out the check for underflow (and I didn't define a `QueueUnderflowException`) but I hope you see how to do that – it's the same as for the stack.

The rest is pretty short:

```
@Override
public E peek() {
    return head.data;
}

@Override
public boolean isEmpty() {
    return head == null; // also consider size == 0
}

@Override
public boolean isFull() {
    return false;
}

@Override
public int size() {
    return size;
}
```

Implementing queues using arrays

You'd need to finish the above off with some other details to have a full and correct implementation (throwing over/underflow exceptions, for example) but it should be enough to convey the idea.

Now, suppose we want to use arrays to back our queue, like we did with stacks. With stacks, we fixed the element at index 0 as the "bottom" of the stack, and maintained a top pointer. What if we do this with queues?

(on board)

We can easily enqueue elements by keeping a pointer to the last element enqueued. But what about dequeuing? We'd have to remove the element at position zero, then move the element from position 1 into 0, 2 in to 1, and so on. That's linear in the size of the queue (and thus terrible).

The right thing to do parallels the linked list implementation. Instead of a head and tail pointer, we'll maintain two indices into the array, and allow both the front and rear of the queue to move through the array.

There is a trick, though: we need to now thing of the array as a circle, not as a linear array, that "wraps around". This is kind of like a clock: (on board)

Now, instead of incrementing our front/rear indices, we use `front = (front + 1) % capacity)` – this is modular arithmetic, and "wraps around" exactly as we want.

We'll define `rear` as the last occupied index (and initialize to what? virtual position "-1"). Thus the next place to enqueue is `rear + 1`. We'll define `front` as the next occupied index to `dequeue` and initialize it to 0.

We have to keep track of the `size`, since both an empty queue and a full queue have rear just before front.

```
package linearadts;

public class CircularQueue<E> implements Queue<E> {
    E[] array;
    int front;
    int rear;
    int size;

    public CircularQueue(int capacity) {
        array = (E[]) new Object[capacity];
        size = 0;
        front = 0;
        rear = -1;
    }

    @Override
    public void add(E e) {
        // check for overflow!

        rear = (rear + 1) % array.length;
        array[rear] = e;
        size++;
    }

    @Override
    public E remove() {
        // check for underflow!

        E value = array[front];
        array[front] = null;
        front = (front + 1) % array.length;
        size--;
        return value;
    }

    @Override
    public E peek() {
        // check for underflow!
        return array[front];
    }

    @Override
    public boolean isEmpty() {
        return size == 0;
    }

    @Override
    public boolean isFull() {
        return size == array.length;
    }

    @Override
    public int size() {
        return size;
    }
}
```

How you might implement priority queues using heaps

So finally, we're going to do an implementation of priority queues.

It usually comes up in lecture that you could do a simple implementation by just using an array or list or the like, and re-`sort()`ing it after each call to `add` or `remove` – and that's true! It would be functionally correct! The problem is related to *performance*. In particular, the comparison-based sorts we've seen so far are quadratic in the length of the list. That's terrible performance to add or remove single elements! And even the best such sorts are log-linear, that is, (n log n), which while not quadratic is still more-than-linear, which means touching a single element still requires, in the worst case, more than a linear number of operations! Ugh.

So it turns out we can do better by using a particular kind of data structure – a *heap*. Now, just like the implementation details of stacks and queues, you're unlikely to need to build heaps yourself in day-to-day applications programming (though, you will definitely appreciate this if you go to 187, and/or if you go onto other lower-level systems programming in the future).

A *heap* is, conceptually, another kind of binary tree, kinda like a binary search tree.

You'll remember that BSTs are used to implement certain kinds of maps and sets, and are binary trees with a very specific property: each node's N's left-descendants (that is, their children, and their children's children, etc.) contain a value less than N's value, and right-descendants contain values greater than N's value.

A BST is, then, a *very organized* sort of tree. A heap is, as you might guess from the name, a *somewhat less organized* sort of tree. Instead of obeying the BST property, it obeys a *weaker* property, called, not coincidentally, the *heap property.* The heap property is as follows: In a min-heap, each node N's value is strictly less than it's children's values. That's it. (In a max-heap, it's strictly greater-than. And, you can relax this to be "less-than-or-equal" but I'm going to skip that as it makes the code slightly more complicated.)

The idea here is that, for a priority queue, you use a heap, and the "next" thing is always at the root. That's it, that's the idea.

Let's show an example:

(draw a min-heap with the numbers 1-7 on the board, something like)

```
        1
   2        4
 6   3    7   5
```

Every node's value is less than that of its children, so this is a heap.

Now, suppose we want to add a node to the heap. How might we do so? Well, we could try to guess the right spot and hope for the best. But the "right way" is to add it at the first open space at the "bottom" of the tree, here, as a child of 6. Let's add the value 8. If we do, then great! It's greater than 6 (or in other words, 6 is less than the new node 8), and we're done. We know, because of the heap property, that nothing above six is greater than six.

Let's add another node, zero, also a child of 6. Uh-oh! 0 is less than 6. So, the heap is currently broken – it doesn't obey the "heap property." So whenever we add a node, we have to check and "fix" the heap. How? By comparing with its parent, and if needed, swapping it – this is called "sifting up" (sometimes also called "bubbling up" or "trickling up"). If we don't need to sift up, like when we added 8, then we're done. If we do need to sift up, then, importantly, we need to "sift up" on the new parent node, too! We keep doing it until we no longer need to do it, or we reach the root.

So here, we'd sift 0 up to 6. Then we'd sift up to 2, then up to 1. The new tree would look like this:

```
        0
   1        4
 2   3    7   5
8 6
```

So that's how we add things, by appending to the bottom and then sifting up.

How do we remove things? We can't just take the top node, because then it's not clear what happens. Do we move an entire limb of the tree up? What if that breaks the heap property? It turns out, again, that there's a right answer here. The right answer is to remove and return the top (root) element, but before you return, do a little bit more. What is that little bit more? You move the "bottom" element of the tree to the root, and then "sift down" the element into the right place.

Here, the comparison is slightly more complicated: you look at a node N's value, and it's two children's values, C1 and C2. If node N is the smallest (in a min-heap), you're done! If not, then you swap N with the *smaller* of C1 or C2 – so the smallest of the three is the new parent of this group of three. Then, you repeat the process with the child you swapped N down into – you sift down there again. Just like sifting up, you repeat until you can stop (because N is smallest) or must stop (because N has no children!).

Now, why is this better than re-sorting an entire list each time? Because you touch, at most, one node on each level of the tree – and the tree only has (log n) levels. In other words, it's strictly sub-linear in the number of nodes in the tree! So this is way better than (n log n)!

How might you actually build this? I'm not going to spend the time on that in class – take 187 or read basically any data structures textbook – but I'll give you the sketch here. The naive way to do it is with a `Node`-like structure, like linked lists. To do it, you need not just `next` references, but three in total: `left` and `right` children (for sifting down), and `parent`s (for sifting up).

But in practice, this is often implemented using expandable arrays. In particular, you number the nodes of a tree like this:

```
   0
 1   2
3 4 5 6
```

etc. And use those numbers as indices into an array. Then, `parent` is always `(index - 1) / 2`, `left` is always `index * 2 + 1` and `right` is always `index * 2 + 2`. `size` is the index of the "last" element of the heap (just like the `ArrayList`). You can `add` by placing the next element at `size + 1`, just like arraylists, and you can expand, just like arraylists. In fact, this general construction always works for storing trees in arrays, and you'll definitely see it again in 187.

Last modified: Friday, April 8, 2022, 6:59 AM

◀ Lecture notes 10-01

Jump to...

Programming assignment 10: Search ▶

© University of Massachusetts Amherst • Site Policies • Site Contact

Moodle Help for Students • Moodle Help for Instructors

You are logged in as Tanish Patel (Log out)
COMPSCI186-SEC01 SP22

Get Help
Help for Students
Help for Instructors

COMPSCI186-SEC01 Using Data Structures Spring 2022

Lecture notes 11-01

Recursion

In common language, a "circular definition" is not useful. When you define something in terms of itself, it conveys no meaningful information: "Good music is music that's good" tells you nothing.

But math (and computer science) often uses *recursive definition* to define concepts in terms of themselves. WTF?

For example:

- directories contain files and/or directories
- an if statement in Java is made of `if`, an expression that evaluates to a boolean, and a statement (plus an optional `else` and another statement). Any of the other statements could be `if`s!
- a postfix expression is either an operand or two postfix expressions followed by an operator

and so on.

How do the above differ from our bad definition of good music? While they are defined in terms of themselves, the circularity is only partial – there's a way out of the circle.

Recursion and fractals

(This particular exercise is directly inspired by a talk Dan Garcia gave at a recent SIGCSE about part of their CS 10 (CS Principles) class, The Beauty and Joy of Computing.)

Many things can be expressed "recursively" that is, partly or wholly in terms of themselves. This is usually where your CS teacher immediately brings up factorial. BUT I WILL MANFULLY RESTRAIN MYSELF! (or maybe you'll see it in lab or later this week 😀)

I'll point out two things to get us started:

First, that recursion is a beautiful and simple (though maybe not easy) way to think about computation. The dual of the Turing machine is the lambda calculus, and it's elegantly powered by recursion.

Second, that recursion is all around us in nature, in the form of fractals, which when used colloquially means "things that are similar to themselves over extended, but finite, scales." (See Wikipedia for some example of fractals: https://en.wikipedia.org/wiki/Fractal#Natural_phenomena_with_fractal_features)

We're going to explore the concept of recursion graphically today, rather than with factorials and Fibonacci and the like. To be clear, we will also do a little bit of factorial stuff and the like, too. Well, more than a little bit.

Drawing trees

So, we're going to learn to draw beautiful trees that look like this (on board / slides).

But we'll start with something simpler.

The important thing to see is that this tree is self-similar; in other words, it's fractal: each part contains smaller versions of itself. (on board)

We're going to draw this in Processing. But first, some preliminaries.

Installing Processing

Processing is a Java-based tool for sketching, computer-based art, and learning to code in the context of the visual arts. There are also versions that run in Python and in JavaScript in your browser.

The Java-based Processing is available as both a stand-alone program, and a `.jar` you can integrate into your Java projects.

Unfortunately, the support for Java versions greater then Java8 is in alpha and still somewhat sketchy (see what I did there? dad jokes amirite?), so we're going to use the standalone version. If you want to follow along, you can download and install it pretty easily from their website. Alternatively, you can download the "Processing4" .jar and include it in a Code project for Java11. You'll also have to do a little bit more work to write full classes rather than just the relevant methods. But the ideas are the same (as are the syntax and method names).

Aside: setting up Code

If you do this in Code, you'll need to do it in a project that has the Processing JAR set up. You'll need to write a class that extend `PApplet` like so, with a very specific `main` method:

```
import processing.core.PApplet;

public class Trees extends PApplet {

    public static void main(String[] args) {
        PApplet.main("processing.Trees"); // or whatever package.Class is the name of this file
    }
}
```

Processing lets you skip this step and directly start coding methods and variables, without worrying about the class, which is what we're going to do.

Turtle / pen graphics

One (old!) way to introduce programming to people is via physical computing. Some of you may have had a robotics club or the like in high school; this approach is a very specialized version of that idea.

In short, a "turtle" (which is really just a robot with a pen attached) could be programmed to draw things on paper. You can also do this virtually, which is what we're going to do, but I thought you might appreciate the history.

(show slide)

A turtle (or "pen") minimally has three pieces of state: its x and y coordinates, and the direction it's facing. You could add others (like the pen color or thickness, etc.) but let's just start there.

We're going to implement a turtle pen, and then use it to explore tree-drawing and recursion.

Setting up processing

Processing lets you do pretty arbitrary things (it's Java, after all). We're going to tame the complexity somewhat by doing "turtle graphics" aka "pen graphics." In other words, we're going to draw everything in terms of pen. The pen has three pieces of state: X, Y, and direction. This is similar to a turtle with a pen attached crawling around on a piece of paper, or my hand on the board, sorta.

```
float penX;
float penY;
float direction;
```

Note that we're using `float` instead of double because much of the Processing API uses `float`; using `float` prevents various warning messages.

OK. So far, so nothing. Let's set up the sketching area:

```
void setup() {
  size(600, 600);
  penX = width / 2;
  penY = height / 2;
  background(255);
  noLoop();
}
```

(Note: because reasons, if you are doing this in Code, you'll need to make `setup public`, and move `size()` to a `public void settings()` method.)

Line by line, what's happening here?

`size(600, 600);` creates a viewable (x, y) rectangle to draw on – it's the sketch, in Processing lingo. The x coordinates increase left to right, and the y coordinates increase top to bottom. This is the opposite of algebra y-axes! But it is common in graphics.

`penX = width / 2;` and the following line center the pen on the x/y coordinates.

`background(255);` sets the entire background to white. If you pass a value on the range 0...255 as a color, Processing interprets it as a shade of grey (0 = black, 255 = white).

`noLoop();` tells Processing that once we start to draw, call draw() only one time; otherwise (by default) it is called repeatedly in an endless loop.

Now, let's teach processing how to move the pen:

```
void move(float amount) {
  float newX = penX + sin(radians(direction)) * amount;
  float newY = penY - cos(radians(direction)) * amount;

  line(penX, penY, newX, newY);

  penX = newX;
  penY = newY;
}
```

This is a three-step process. First we have to compute where we're going, on the basis of where we are. We use trig to figure out how far away in the x and y directions we need to go (remember, y is flipped, hence the minus). We draw a line from the pen's current location to its new location. Then we update the pen's location.

Let's test it out in a `draw()` method:

```
void draw() {
    move(50);
}
```

OK, that works. What about if we want to turn?

```
void turn(float degrees) {
  direction += degrees;
}

void draw() {
    move(50);
    turn(90);
    move(50);
    turn(90);
    move(50);
    turn(90);
    move(50);
    turn(90);
}
```

Great. How about a random walk?

```
void draw() {
  turn(random(360));
  stroke(random(255), random(255), random(255)); // random pen color each time, RGB when you pass three values
  move(random(20));
}
```

And how about we use the mouse to turn it on and off?

```
void mousePressed() {
  loop();
}

void mouseReleased() {
  noLoop();
}
```

OK, now we're cooking!

Drawing a tree

So, back to our drawing: notice that each tree is made up of smaller trees. We're going to write a `tree` method that draws a tree. It'll start with a `move()` call to draw the trunk, then it will turn left, draw the smaller tree, turn right, draw the smaller true, and go back to where it started.

In other words, it'll start with a `move()` call to draw the trunk, then it will turn left, draw the smaller tree *using* `tree()`, and so on.

"Hey dawg, I heard you liked trees inside of your trees, so can you show me how I can use `tree()` inside of `tree()` to draw a tree?" That's today's magic show!

No recursion here

First write `tree1`. All it does is draw the trunk by moving forward `size` and then backward `size`:

```
void tree1(float size) {
  move(size);
  move(-size);
}
```

Make sure it works:

```
void draw() {
    tree1(100);
}
```

Maybe also move the root of the tree down a bit:

```
// in setup()
penY = height * 0.9
```

Now, let's draw a two-level tree.

```
void tree2a(float size) {
  move(size);
  turn(-15);
  move(size * 0.75);
  move(-size * 0.75);
  turn(30);
  move(size * 0.75);
  move(-size * 0.75);
  turn(-15);
  move(-size);
}
```

At this rate, it's gonna take a lot of code to draw a beautiful tree. Don't worry, though, we'll get better.

Does everyone understand what's happening above? (Trace through on board.)

More levels

OK, let's simplify a bit. Do you see that `tree2a` contains `tree1` if you stop and squint at it a bit? Let's rewrite it to use `tree1`:

```
void tree2(float size) {
  move(size);
  turn(-15);
  tree1(size * 0.75);
  turn(30);
  tree1(size * 0.75);
  turn(-15);
  move(-size);
}
```

Now, I want you to write a `tree3` that uses `tree2`. It should *not* use `tree1`. (Hint: it looks almost exactly like `tree2`; copy/paste and change three characters).

```
void tree3(float size) {
  move(size);
  turn(-15);
  tree2(size * 0.75);
  turn(30);
  tree2(size * 0.75);
  turn(-15);
  move(-size);
}
```

How would `tree4` look? Exactly the same, except it would call `tree3`. So, here's the big idea: we can write a `tree` method that *uses itself* to draw a tree, so long as it knows how many levels it's expected to draw.

A Recursive `tree`

`tree` will need a `levels` parameter in addition to size. This tells us what "level" we are on, where level 1 is the last level (the outermost branches):

```
void tree(int levels, float size) {
    move(size);
    turn(-15);
    tree(levels - 1, size * 0.75);
    turn(30);
    tree(levels - 1, size * 0.75);
    turn(-15);
    move(-size);
}
```

What happens when we run this in our `draw()` method?

As Processing helpfully explains, there's a problem with our *recursion*. Let's think about what's going on. (on board, trace operation with levels = 2)

The problem is that the methods we wrote before aren't all exactly the same: `tree1` in particular just drew a trunk, and no branches. So, we need a special *case* to handle this:

```
void tree(int levels, float size) {
  if (levels == 1) {
    move(size);
    move(-size);
  }
  else {
    move(size);
    turn(-15);
    tree(levels - 1, size * 0.75);
    turn(30);
    tree(levels - 1, size * 0.75);
    turn(-15);
    move(-size);
  }
```

And now it works. Let's try a larger levels for a prettier tree. (show 9)

This general code pattern, with a simple base case that doesn't call the method itself, and a less simple case that does call itself, is typical of recursive code. Let's break it down:

```
void tree(int levels, float size) {
  // the base case; a problem so simple we know how to handle it
  // completely
  if (levels == 1) {
    move(size);
    move(-size);
  }
  // the recursive case
  // do a small part here, and delegate the rest to somebody else
  else {
    // draw the trunk
    move(size);
    // position the pen for the left sub-tree
    turn(-15);
    // delegate drawing the left sub-tree to someone else
    tree(levels - 1, size * 0.75);
    // re-position for the right sub-tree
    turn(30);
    // delegate drawing the right sub-tree to someone else
    tree(levels - 1, size * 0.75);
    // re-position to go back to the start
    turn(-15);
    // retrace our steps back to where we started
    move(-size);
  }
```

Notice that in the recursive case, we are always making progress toward the base case – here, `levels` is guaranteed to be decremented by one each time the method calls itself, so we're guaranteed (mostly) to hit `levels == 1` and stop. If we don't do this, the method calls itself indefinitely, and we end up with a `StackOverflow` (at least in Java).

What would happen if we started with `levels` < 1?

Sometimes a recursive program is more elegant if you have a smaller base case. What would a "zero level" tree mean? Can you rewrite the program to have a base case where `levels == 0`?

An exercise

Try to draw a more "beautiful" tree. Two things that might help:

```
strokeWeight(5);
```

makes the stroke (the ink left by the pen) five pixels wide instead of one.

And you can change the color by red green blue components. Google has a nice color picker to help.

```
stroke(75, 160, 15); // kinda greenish
```

Last modified: Tuesday, April 12, 2022, 4:11 PM

◀ search-student.zip

Jump to...

Processing tree drawing code ▶

© University of Massachusetts Amherst • Site Policies • Site Contact

Moodle Help for Students • Moodle Help for Instructors

You are logged in as Tanish Patel (Log out)
COMPSCI186-SEC01 SP22

Get Help
Help for Students
Help for Instructors

COMPSCI186-SEC01 Using Data Structures Spring 2022

Lecture notes 11-02

Recursive algorithms: the mathy example

It turns out some algorithms are most naturally expressed recursively – usually they correspond to solving problems or implementing abstractions that are also defined recursively. A big chunk of COMPSCI 250 explores the relationship between recursion and inductive proofs.

For example, consider the factorial function over non-negative integers. n-factorial (usually written *n!*) is the product of the numbers 1 through n: (1 * 2 * 3 * ... * n)

4! = 1 * 2 * 3 * 4 = 24.

What is 0!? 1. Why? The intuition is that you're multiplying together "no numbers", which is like not multiplying at all. What's the same as not multiplying at all? Multiplying by 1. The real reason for this has to do with the algebra of integers (which differs from the algebra you learned in middle/high school); take an abstract algebra or number theory course if you want to learn more.

We can recursively define the factorial function.

```
0! = 1
n! = n * (n-1)!
```

If we use this definition to compute 4!, we get 4 * 3!. Applied again, we get 4 * 3 * 2!, finally, we get 4 * 3 * 2 * 1, which is the same as the non-recursive definition.

The non-recursive definition is relatively easy to transcribe into Java (we'll not handle invalid inputs, like -1):

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

What about the recursive one? Baby steps. Let's define `factorial0`, then `factorial1`, and so on:

```
static int factorial0() }
  return 1;
}

static int factorial1() }
  return 1 * factorial0();
}

static int factorial2() }
  return 2 * factorial1();
}

static int factorial3() }
  return 3 * factorial2();
}

static int factorial4() }
  return 4 * factorial3();
}
```

`factorial4` calls `factorial3`; once `factorial3` returns an answer, `factorial4` returns its answer. How does `factorial3` work? The same way, using `factorial2`. In fact, everything except `factorial0` works the same way. So why not capture this pattern in a single method:

```
static int factorial(int n) }
  return n * factorial(n - 1);
}
```

That's not quite right, because it won't "stop" when it gets to zero. We could check for the zero case and call `factorial0`:

```
static int factorial(int n) }
  if (n == 0) {
      return factorial0();
  } else {
    return n * factorial(n - 1);
  }
}
```

or just get rid of it entirely:

```
static int factorial(int n) }
  if (n == 0) {
      return 1;
  } else {
    return n * factorial(n - 1);
  }
}
```

What actually happens when we call the recursive `factorial` with an argument of 4?

That call makes a call to `factorial(3)`, which calls `factorial(2)`, which calls `factorial(1)`, which calls `factorial(0)`, which returns 1 to `factorial(1)`, which returns 1 to `factorial(2)`, which returns 2 to `factorial(3)`, which return 6 to `factorial(4)`, which returns 24, which is the right answer, which is the same as what happened when we had different (`factorial1`, `factorial2`, etc.) methods. But we do it all from one method!

The values are passed along on the *call stack*, which grows to a height of five while this recursive method operates. Why doesn't it just blow up? Because eventually the recursion terminates. How do we know it will terminate, and how do we know the answer is right? Glad you asked.

Three questions (about recursion)

> Stop! Who would cross the Bridge of Death must answer me these questions three, ere the other side he see.
> — Bridgekeeper, *Monty Python and the Holy Grail*

There are three questions we must answer affirmatively about a recursive algorithm if we expect it to (a) terminate and (b) produce the correct answer.

1. Does it have a base case?
2. Does every recursive call (more generally: every bounded sequence of recursive calls) make progress toward (or arrive at) the base case?
3. Does the algorithm get the right answer if we assume the recursive calls get the right answer?

A *base case* is the end of the line, when there is no more recursion. `factorial`'s base case is when n=0.

Guaranteeing progress is trickier. If our parameter is numerical, and each recursive call brings us a constant amount closer toward the base case (as is the case of `factorial`) then we can be certain we'll eventually get there. But sometimes you need two or more recursive calls to make progress (see `foo` below).

Justifying correctness is something you'll do more in COMPSCI 250, but in short, if you are proceeding from a correct recursive definition (which factorial is) then your recursive algorithm will be correct.

Recursion for control flow

Recursion is a mechanism for control flow, similar to loops in its behavior but quite dissimilar in terms of its appearance.

We talked about how, if you can define the solution to some problem recursively, you can generally rewrite that recursive definition into a recursive method in Java.

Generally, any time we can write a recursive definition for a problem, we can then write the recursive method pretty easily. But how do we write these recursive definitions? It's pretty easy if you're a Math major and you're used to this style of thinking, or if you're a CS major who has passed through the purifying flame of COMPSCI 250, or you're just a nerdy-nerd.

But what if you're more comfortable thinking about things iteratively? And why wouldn't you be, since you've been conditioned by AP CS and/or 121 and/or this class to always think in terms of `for` (and sometimes, `while`) loops?

That's OK. Next we're going to spend some time practicing translating loops from their iterative to their recursive form, so that you can see (some of) the symmetry between these two forms of control flow.

So, first things first: it turns out that while `for` is generally very useful for the iteratively-minded, it obscures some of the connection between iteration and recursion in a way that `while` does not. But good news: we can trivially rewrite any (non-enhanced) `for` loop as a `while` loop. To refresh your memory, here's an example:

```
int sumToNFor(int n) {
    int sum = 0;
    for (int i = 1; i <= n; i++) {
        sum += i;
    }
    return sum;
}

int sumToNWhile(int n) {
    int sum = 0;

    int i = 1;
    while (true) {
        if (i > n) {
            break;
        }
        else {
            sum += i;
        }
        i++;
    }
    return sum;
}
```

Notice I separated out the `break` to make the cases clear.

More generally:

```
for (init; condition; increment)
    doStuff
```

is equivalent to

```
init
while (true):
    if !condition:
        break
    else:
        doStuff
    increment
```

There's still not an exact correspondence to the recursive form, though:

```
int sumToN(int n) {
    if (n == 0) {
        return 0;
    }
    else {
        return n + (sumToN(n - 1));
    }
}
```

Why not? Because our loop control variable (i) is still being tested against an *input* to the method (a variable), rather than an end point that's independent of the input. If we rewrite our `for` loop to terminate based upon a constant rather than a variable:

```
for (int i = n ; i >= 0; i--)
```

and rewrite our `while` loop the same way:

```
int sumToNWhile(int n) {
    int sum = 0;

    int i = 1;
    while (true) {
        if (i == 0) {
            break;
        }
        else {
            sum += i;
        }
        i--;
    }
    return sum;
}
```

It's maybe now a little easier to see how the base case and the recursive case of the recursive solution relate to the branches within the `while` loop of the iterative solution.

There's a general observation here about translating iterative solutions to recursive ones to make: Recursive methods must "make progress toward the base case." For methods where "the thing that's making progress" is a counter, that means the most natural way to write them is usually *counting down* rather than up, even though that's the opposite way we typically write our `for` loops.

Why? Because we need to know "when to stop" our recursion. It's easy to know when you've hit a constant (in other words, when `n == 0`), since there's no external state to keep track of. But it's less clear "how to stop" if your recursion starts at 0 and counts up to n. It's certainly possible (we'll get there next week).

Last modified: Friday, April 15, 2022, 12:43 PM

◀ Processing tree drawing code

Jump to...

Sample quiz questions ▶

© University of Massachusetts Amherst • Site Policies • Site Contact

Moodle Help for Students • Moodle Help for Instructors

You are logged in as Tanish Patel (Log out)
COMPSCI186-SEC01 SP22

Get Help
Help for Students
Help for Instructors

COMPSCI186-SEC01 Using Data Structures Spring 2022

Lecture notes 12-01

More examples of recursion

Sums, again

Let's move away from simple numerical examples and start looking at recursion over data structures. In particular, let's look at operations over arrays and over linked lists.

Suppose you want to sum the elements of an array iteratively. Easy-peasy:

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

OK, now suppose you want to sum the integers contained in a linked list. Remember, a linked list is a set of `Node<E>` objects (each with a `next` reference and a `E data` variable). Lists are identified by their `head`, which is just a node. The iterative way to sum up a list of integers in an linked list would look like:

```
int sum(Node<Integer> head) {
    int s = 0;
    for (Node<Integer> current = head; current != null; current = current.next) {
        s += current.data;
    }
    return s;
}
```

Recursion on linked lists

How might we write this recursively? Well, how would we define it recursively?

The sum of the elements in an empty list is zero.

The sum of the elements in a list of one element is just that element's value.

For two (or more) elements, it's the two elements values...or put another way, it's the sum of the current value and the sum of the rest of the list's values.

This is similar to how we might think of `sumToN`, where we say the sum of the first n is equal to n + the sum of (n-1). But in this case, we say the sum of the values is equal to the current node's value plus the sum of the remaining nodes.

In other words:

sum(node) =

- 0 if node is null, or
- the current node's value + the sum of the remaining nodes' values (node.data + sum(node.next)) otherwise.

It's an exact parallel to our `sumToN` code. We'd write it the same way:

```
int sum(Node<Integer> node) {
    if (node == null) {
        return 0;
    }
    else {
        return node.data + sum(node.next);
    }
}
```

Code-wise, this approach is not horribly worse than an iterative solution, even if it feels unfamiliar to the typical Java programmer. Linked structures (which are defined partially recursively) tend to be pretty OK to write recursive algorithms for. In some languages, you get more syntax support, and writing recursive algorithms is the preferred way to do many things. And on some data structures and problems, recursive solutions, even in a language that favors iteration, like Java, "feel" more natural. But more about that later.

Recursion on arrays

OK. What about the array case for summing entries, could you do it recursively?

You need to keep track of both the array and the index to the array. But the `sum` method only takes an array as a parameter, and doesn't know anything about an index. For reasons I'm not going to bore you with, the Land of Recursion frowns upon the use of state that exists outside of the method calls themselves (it's not disallowed, but used injudiciously it can lead to recursive methods that are impossible to reason about). Instead, all state gets passed into methods as arguments, and out of methods as return values. In that sense, the methods are "functions" in the mathematical sense.

So anyway, you would need to write what's called a "helper method." The purpose of the helper method is to give you access, through a new argument, to some state you wish to propagate through the recursion. What state do we need? The index of the array currently being considered. What does this look like?

```
int sumRecursive(int[] a) {
    return sumRecursive(a, 0);
}

int sumRecursive(int[] a, int i) {
    if (i == a.length) {
        return 0;
    }
    else {
        return a[i] + sumRecursive(a, i + 1);
    }

}
```

What's going on here? `sum(a)` calls `sum(a, i)`; only the latter is recursive. What's it saying?

sum(a, i) =

- 0 if the index i is past the end of the array, or
- the current element's value plus the sum of the values of the elements after it (a[i] + sum(a, i+1)) otherwise

Here, we're passing some state – the index in question – along on the recursive calls. And we need to do so because we need to keep track of "where we are" in the recursion. In linked lists, this happens naturally (because the `node` reference tracks where we are and the structure itself), but arrays require us to track it explicitly in a variable.

Again: adding this helper method lets us pass state through the recursion.

The utility of writing out your recursion

It's a good idea to get into the habit of writing out your recursion in text before coding it. Remember this?

```
int sum(Node<Integer> node) {
    if (node == null) {
        return 0;
    }
    else {
        return node.data + sum(node.next);
    }
}
```

translates to

The `sum` of a list :

- zero if the list is empty,
- the value in the head, plus the `sum` of the rest of the list otherwise

Or think of `factorial`, for example. The `factorial` of a number is:

- 1 if the number is zero,
- the number times (the factorial of the number minus one) otherwise

And so on.

Suppose we want to make a copy of a linked list. Iteratively what do we do? Something like:

```
static <E> Node<E> copyIterative(Node<E> head) {
    if (head == null) {
        return null;
    }

    Node<E> newHead = new Node<>();
    newHead.data = head.data;

    Node<E> oldRef = head.next;
    Node<E> newRef = newHead;
    while (oldRef != null) {
        Node<E> n = new Node<>();
        n.data = oldRef.data;
        newRef.next = n;

        newRef = newRef.next;
        oldRef = oldRef.next;
    }
    return newHead;
}
```

In the code above, we maintain two references: one (`oldRef`) points to the next node in the old list to be copied; the other (`newRef`) points to the most recently copied node in the new list (that is, the new list's current tail). We traverse the old list until we reach the end, copying each node into a new node and attaching it to the end of the new list. (on board)

What about doing it recursively? Let's write out the definition first.

copy(node) =

- null if the list is empty,
- otherwise it's a copy of the current node, whose next pointer points to a copy of the rest of the list:

```
static <E> Node<E> copyRecursive(Node<E> node) {
    if (node == null) {
        return null;
    }
    else {
        Node<E> n = new Node<>();
        n.data = node.data;
        n.next = copyRecursive(node.next);
        return n;
    }
}
```

How's it look? Base case seems fine. Let's do some examples on the board.

I find this version a little easier to read than the iterative case, but maybe that's just me.

Passing values through the recursion

In all of the examples we've seen so far, the result has been implicitly accumulated – it's been computed from and returned as the result of a chain of recursive calls. Sometimes doing so is awkward, and we want to explicitly "accumulate" a result into a variable as we go. Then, we return that variable (which contains the accumulated result) at the end of the recursion, in the base case.

We've already seen a mechanism to do this – when we used recurison to iterate over an array, we passed the index down through the recursion. We can do the same thing here, and pass the result through the recursion (building it up step-by-step).

Reversing a linked list, recursively

The classic example of this is making a copy of a list in reverse order.

reversed(node) =

- null if the list is empty,
- otherwise it's a reversed copy of the rest of the list, with the current node at its end.

But how do we append *to the end*? We don't have a tail pointer. We'd have to traverse the list (recursively?) each time. That seems (and is) crap, that is, quadratic.

Let's take a different approach. Again, remember the array-summing?

```
    int sum(int[] a) {
        return sum(a, 0);
    }

    int sum(int[] a, int i) {
        if (i == a.length) {
            return 0;
        } else {
            return a[i] + sum(a, i + 1);
        }
```

Here, we had to pass the value `i` into the "helper method" in order to "keep track" of where we were in the array – it was an extra parameter. We can also pass around a partial result into the recursion. Each step of the recursion builds a little more of the answer, and when we reach the "end" (base case), we return the built-up result.

`reversed(node, rev_so_far)` =

- `rev_so_far` if the list is empty (`rev_so_far` is the "accumulated" result!),
- otherwise put the head of the list at the head of `rev_so_far`, and return `reversed` on the *rest* of the list and the *new* `rev_so_far`

In other words, we'll build up the reversed list as we go, passing it in as a parameter, and returning it when we're done:

```
static <E> Node<E> reversed(Node<E> node) {
    return reversed(node, null);
}

static <E> Node<E> reversed(Node<E> node, Node<E> rev) {
    if (node == null) {
        return rev;
    }
    else {
        Node<E> n = new Node<>();
        n.data = node.data;
        n.next = rev;
        return reversed(node.next, n);
    }
}
```

(show example on board)

Passing state, either to control the recursion (like array summing) or to build a result (like reversing the list) is common in recursive programming.

Last modified: Tuesday, April 19, 2022, 4:44 PM

◀ crawler-student.zip

Jump to...

Lecture notes 12-02 ▶

© University of Massachusetts Amherst • Site Policies • Site Contact

Moodle Help for Students • Moodle Help for Instructors

You are logged in as Tanish Patel (Log out)
COMPSCI186-SEC01 SP22

Get Help
Help for Students
Help for Instructors

COMPSCI186-SEC01 Using Data Structures Spring 2022

Lecture notes 12-02

Recursion on binary trees

Remember earlier in the semester when we talked about binary trees? They're like lists: structures composed of linked nodes. But in a binary tree, each node has a left and right "child" rather than a single next node:

```
class Node<E> {
    public E data;
    public Node<E> left;
    public Node<E> right;
}
```

Some types of binary trees have additional constraints. For example, binary *search* trees might require that the values stored in the left subtree always be less than or equal to the current node's value, and the values stored in the right subtree always be greater.

As you'll see in 187, you can operate on these trees either recursively or iteratively. Some operations are about as hard to write iteratively or recursively (not unlike writing `contains` in a linked list). Others are more clear one way or the other. Let's do a few examples. The basic binary search tree looks like this:

```
// note we need the Comparable here if it's a BST, so we can order the Nodes by E
// but a regular (non-search_ tree doesn't require it.
public class BinarySearchTree<E extends Comparable<E>> {
    private Node<E> root;
}
```

Suppose we have an existing tree. How do we find an element in the tree? Let's write a `contains()` method both ways (iterative and recursive). This method bears a strong resemblance to the `contains` method of a linked list, but has to check the correct child of a node (notably, not both children!).

Iterative first: (1) We start at the root. then, we (2) repeat the following: If the node contains the data, (3) return true; otherwise, (4) descend to the appropriate child. If we reach the end without finding the value, (5) return false.

```
public boolean containsIterative(E e) {
    //1
    Node<E> node = root;

    //2
    while (node != null) {

        //3
        int cmp = e.compareTo(node.data);

        if (cmp == 0) {
            return true;
        } else if (cmp < 0) {
            node = node.left;
        } else {
            node = node.right;
        }
    }

    return false;
}
```

What about recursively? We'll need a helper method to be able to pass the "current" node in, but otherwise it's much the same.

Our textual description is something like:

contains(node, e) =

- false if we've reached the end of a branch, or
- true if the current node's value is the value we're looking for, or
- it's equal to contains(left child, e) if the value is less than the current node, or contains(right child, e) otherwise

```
public boolean contains(E e) {
    return contains(root, e);
}

private boolean contains(Node<E> node, E e) {
    if (node == null) {
        return false;
    }
    if (node.data.equals(e)) {
        return true;
    }
    if (e.compareTo(node.data) < 0) {
        return contains(node.left, e);
    } else {
        return contains(node.right, e);
    }
}
```

Two more quick binary tree examples, this time of methods that recursion makes easier. Suppose your binary tree contains `Integer`s. How would you find the sum of the values? If this were a list, you would traverse the list and sum them up as you went. But if you "iterate" through a tree, you must go either left or right. What about if you recurse? Then you can define sum() as follows:

The sum of the values of the tree rooted at a given node is equal to:

- zero, if the node is null
- otherwise, it's equal to the current node's value, plus sum(left child) + sum(right child).

(on board)

That's a clear recursion, the code would look something like:

```
int sum(Node<Integer> node) {
    if (node == null) {
        return 0;
    }
    else {
        return node.data + sum(node.left) + sum(node.right);
    }
}
```

Suppose we wanted to just count the number of nodes in the tree?

```
int count(Node<Integer> node) {
    if (node == null) {
        return 0;
    }
    else {
        return 1 + count(node.left) + count(node.right);
    }
}
```

Here's another example. What if we want to calculate the height of a tree? Remember, the height is the distance from the root to the node furthest from the root (on board).

Again, we don't *know* in advance which child of a given node leads toward the "deeper" subtree, so we have to explore both. We can define the height of the subtree rooted at a node as:

- zero, if the current node has no children
- one plus the larger of the heights of the left and right subtrees

Again, in Java, though this one's slightly messier due to the null check:

```
int height(Node<E> node) {
    if (node.left == null && node.right == null) {
        return 0;
    }
    if (node.left == null) {
        return 1 + height(node.right);
    }
    if (node.right == null) {
        return 1 + height(node.left);
    }
    return 1 + Math.max(height(node.left), height(node.right));
}
```

Can you write these (`sum`, `count`, and `height`) iteratively? Sure, but not without (at least) one of two things:

- You could modify the tree to include `parent` references, which would allow backtracking.
- Or, you could use auxiliary storage to keep track of the "pending" parts of the tree. This approach exactly parallels how search works: you need to keep track of the "frontier" of nodes, removing one at a time from it, and adding all children of a current node to it. (You don't need to track the visited set, since there are by definition no loops in a tree.)

Recursion is equivalent to the latter – the program's call stack is your auxiliary storage, and if you use a stack to track the frontier, your iterative program can be written to visit the nodes in exactly the same order as the recursive version.

Hanoi

Tower of Hanoi, a simple puzzle where you move disks from one peg to another, the goal being to move all the disks from the first peg to the last. The constraints are that you may only move one disk at a time, and you must place a disk on either an empty peg, or atop a disk larger than itself.

A recursive procedure to solve the puzzle is as follows. Label the three pegs the source (that is, the starting) peg, the target (the place we want the disks all to go), and the spare (the other peg). To move `m` disks from the source to the target, use the following `solve(m, source, target, spare)` procedure:

1. If we are being asked to move zero pegs, do nothing (base case).

2. Move m – 1 disks from the source to the spare peg using `solve(m - 1, source, spare, target)` This leaves the disk m as a top disk on the source peg and all the other disks on the spare peg.
3. Move the disk m from the source to the target (which is empty).
4. Move the m - 1 disks from the spare to the target, using the now-empty source peg as the spare: `solve(m - 1, spare, target, source)`

What does this look like in code? Let's declare a Hanoi class to handle this:

```
public class Hanoi {
    int n;
    private List<Character> a;
    private List<Character> b;
    private List<Character> c;

    public Hanoi(int n) {
        this.n = n;
        a = new ArrayList<>();
        b = new ArrayList<>();
        c = new ArrayList<>();

        for (char i = (char)('1' + n - 1); i >= '1'; i--) {
            a.add(i);
        }
    }

    public String toString() {
        return a.toString() + "\n" + b.toString() + "\n" + c.toString();
    }

        public static void main(String[] args) {
        int n = 3;
        Hanoi h = new Hanoi(n);
        System.out.println(h);
    }
```

And translate our recursive solver into code, including printouts each time it moves a disk:

```
private void solve(int m, List<Character> source, List<Character> target, List<Character> spare) {
    if (m == 0) {
        return;
    }
    solve(m - 1, source, spare, target);
    target.add(source.remove(source.size() - 1));
    System.out.println("----");
    System.out.println(this);
    solve(m - 1, spare, target, source);
}
```

And finally, set up to call it:

```
public void solve() {
    solve(n, a, c, b);
}
```

Et voilà:

```
    public static void main(String[] args) {
        int n = 3;
        Hanoi h = new Hanoi(n);
        System.out.println(h);
        h.solve();
    }
```

How does it work? It works well. 😎

You'll get the tools you need to prove it's correct in COMPSCI 250.

Other minor notes on recursion

Some other languages make use of recursion and recursive definitions much more natural than Java. For example, here's a snippet of OCaml to computes the factorial:

```
let rec factorial = function
  | 0 -> 1
  | x -> x * factorial (x - 1)
;;

print_int (factorial 10)
```

It's like a direct transcription of the mathematical description! We might talk more about OCaml (and a couple of other languages) later as part of the course wrap-up.

Last modified: Friday, April 22, 2022, 3:18 PM

◀ Lecture notes 12-01

Jump to...

Programming assignment 12: War ▶

© University of Massachusetts Amherst • Site Policies • Site Contact

Moodle Help for Students • Moodle Help for Instructors

You are logged in as Tanish Patel (Log out)
COMPSCI186-SEC01 SP22

Get Help
Help for Students
Help for Instructors