Hug61B Spring 2019 Edition

This book is the companion to Josh Hug's version of CS61B, UC Berkeley's Data Structures course. The spring 2019 version of the course can be found at https://sp19.datastructur.es/. There may be newer versions. See https://datastructur.es.

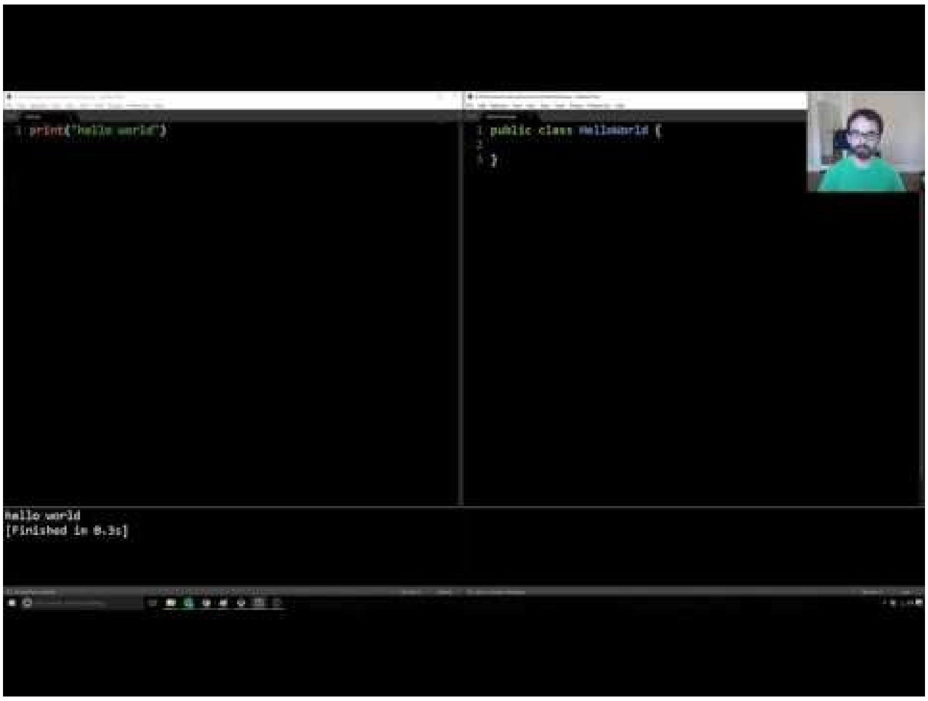
This course presumes that you already have a strong understanding of programming fundamentals. At the very least, you should be comfortable with the ideas of object oriented programming, recursion, lists, and trees. You should also understand how to use a terminal in the operating system of your choice. If you don't have such experience, I encourage you to check out CS61A, UC Berkeley's introductory programming course.

If you already have experience with Java, you might consider skipping straight to chapter 2, though you might still get something by skimming the first chapter.

Licensing:

Table of Contents

Hello World



[Video link]

Let's look at our first Java program. When run, the program below prints "Hello world!" to the screen.

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

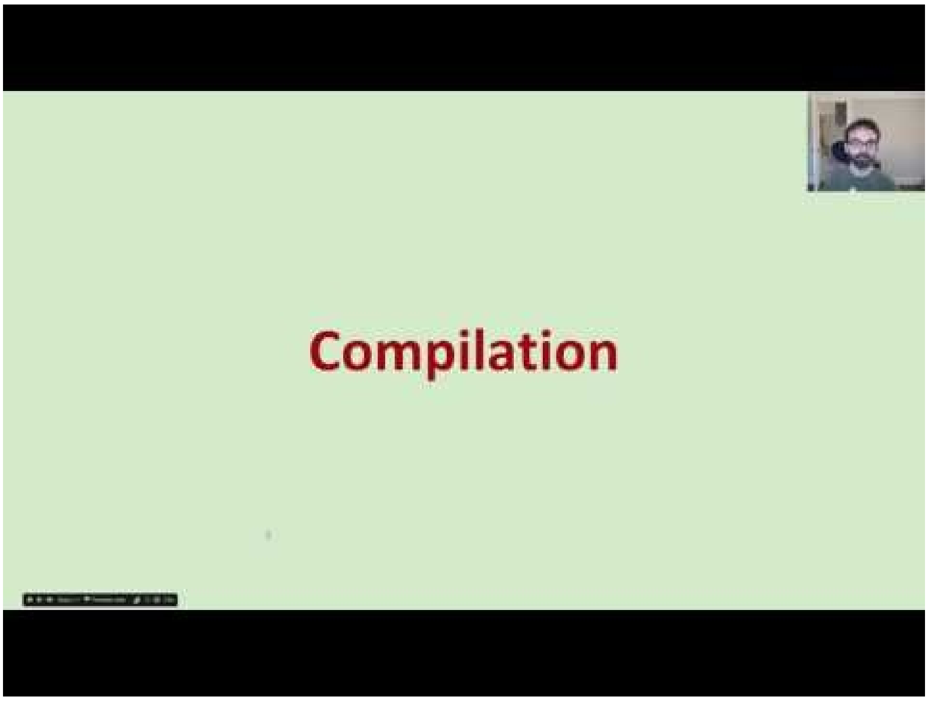
For those of you coming from a language like Python, this probably seems needlessly verbose. However, it's all for good reason, which we'll come to understand over the next couple of weeks. Some key syntactic features to notice:

- The program consists of a class declaration, which is declared using the keywords `public class`. In Java, all code lives inside of classes.
- The code that is run is inside of a method called `main`, which is declared as `public static void main(String[] args)`.
- We use curly braces `{` and `}` to denote the beginning and the end of a section of code.
- Statements must end with semi-colons.

This is not a Java textbook, so we won't be going over Java syntax in detail. If you'd like a reference, consider either Paul Hilfinger's free eBook A Java Reference, or if you'd like a more traditional book, consider Kathy Sierra's and Bert Bates's Head First Java.

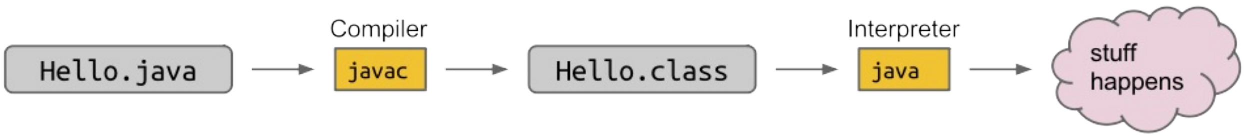
For fun, see Hello world! in other languages.

Running a Java Program



Video link

The most common way to execute a Java program is to run it through a sequence of two programs. The first is the Java compiler, or `javac`. The second is the Java interpreter, or `java`.



For example, to run `HelloWorld.java`, we'd type the command `javac HelloWorld.java` into the terminal, followed by the command `java HelloWorld`. The result would look something like this:

```
$ javac HelloWorld.java
$ java HelloWorld
Hello World!
```

In the figure above, the $ represents our terminal's command prompt. Yours is probably something longer.

You may notice that we include the '.java' when compiling, but we don't include the '.class' when interpreting. This is just the way it is (TIJTWII).

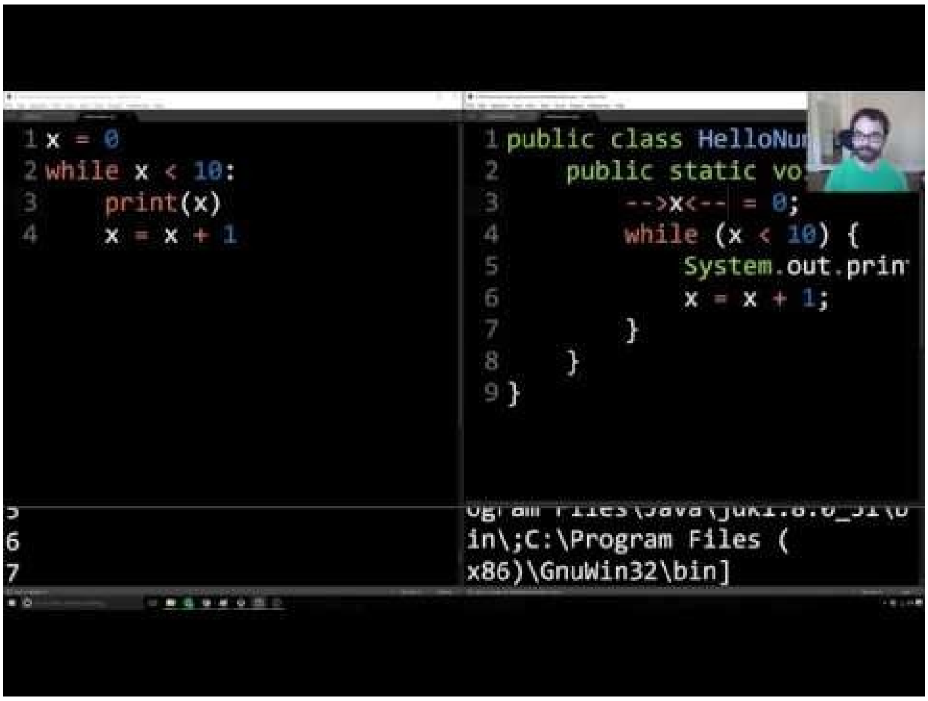
Exercise 1.1.1. Create a file on your computer called HelloWorld.java and copy and paste the exact program from above. Try out the `javac HelloWorld.java` command. It'll look like nothing happened.

However, if you look in the directory, you'll see that a new file named HelloWorld.class was created. We'll discuss what this is later.

Now try entering the command `java HelloWorld`. You should see "Hello World!" printed in your terminal.

Just for fun. Try opening up HelloWorld.class using a text editor like Notepad, TextEdit, Sublime, vim, or whatever you like. You'll see lots of crazy garbage that only a Java interpreter could love. This is Java bytecode, which we won't discuss in our course.

Variables and Loops



Video link

The program below will print out the integers from 0 through 9.

```java
public class HelloNumbers {
    public static void main(String[] args) {
        int x = 0;
        while (x < 10) {
            System.out.print(x + " ");
            x = x + 1;
        }
    }
}
```

When we run this program, we see:

```
$ javac HelloNumbers.java
$ java HelloNumbers
$ 0 1 2 3 4 5 6 7 8 9
```

Some interesting features of this program that might jump out at you:

- Our variable x must be declared before it is used, *and it must be given a type!*
- Our loop definition is contained inside of curly braces, and the boolean expression that is tested is contained inside of parentheses.
- Our print statement is just `System.out.print` instead of `System.out.println`. This means we should not include a newline (a return).
- Our print statement adds a number to a space. This makes sure the numbers don't run into each other. Try removing the space to see what happens.
- When we run it, our prompt ends up on the same line as the numbers (which you can fix in the following exercise if you'd like).

Of these features the most important one is the fact that variables have a declared type. We'll come back to this in a bit, but first, an exercise.

Exercise 1.1.2. Modify `HelloNumbers` so that it prints out the cumulative sum of the integers from 0 to 9. For example, your output should start with 0 1 3 6 10... and should end with 45.

Also, if you've got an aesthetic itch, modify the program so that it prints out a new line at the end.

Gradescope

The work in this course is graded using a website called gradescope. If you're taking the University of California class that accompanies this course, you'll be using this to submit your work for a grade. If you're just taking it for fun, you're welcome to use gradescope to check your work. To sign up, use the entry code 93PK75. For more on gradescope and how to submit your work, see the gradescope guide (link coming later).

Static Typing

One of the most important features of Java is that all variables and expressions have a so-called `static type`. Java variables can contain values of that type, and only that type. Furthermore, the type of a variable can never change.

One of the key features of the Java compiler is that it performs a static type check. For example, suppose we have the program below:

```
public class HelloNumbers {
    public static void main(String[] args) {
        int x = 0;
        while (x < 10) {
            System.out.print(x + " ");
            x = x + 1;
        }
        x = "horse";
    }
}
```

Compiling this program, we see:

```
$ javac HelloNumbers.java
HelloNumbers.java:9: error: incompatible types: String cannot be converted to int
        x = "horse";
                ^
1 error
```

The compiler rejects this program out of hand before it even runs. This is a big deal, because it means that there's no chance that somebody running this program out in the world will ever run into a type error!

This is in contrast to dynamically typed languages like Python, where users can run into type errors during execution!

In addition to providing additional error checking, static types also let the programmer know exactly what sort of object he or she is working with. We'll see just how important this is in the coming weeks. This is one of my personal favorite Java features.

To summarize, static typing has the following advantages:

- The compiler ensures that all types are compatible, making it easier for the programmer to debug their code.
- Since the code is guaranteed to be free of type errors, users of your compiled programs will never run into type errors. For example, Android apps are written in Java, and are typically distributed only as .class files, i.e. in a compiled format. As a result, such

  applications should never crash due to a type error since they have already been checked by the compiler.
- Every variable, parameter, and function has a declared type, making it easier for a programmer to understand and reason about code.

However, we'll see that static typing comes with disadvantages, to be discussed in a later chapter.

Extra Thought Exercise

In Java, we can say `System.out.println(5 + " ");`. But in Python, we can't say `print(5 + "horse")`, like we saw above. Why is that so?

Consider these two Java statements:

```
String h = 5 + "horse";
```

and

```
int h = 5 + "horse";
```

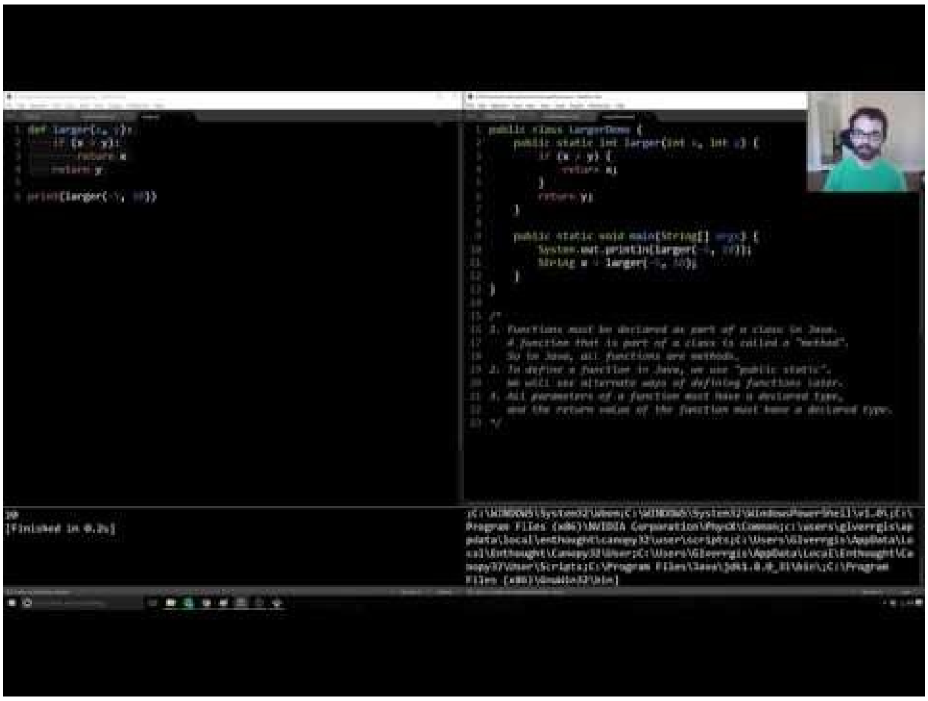
The first one of these will succeed; the second will give a compiler error. Since Java is strongly typed, if you tell it `h` is a string, it can concatenate the elements and give you a string. But when `h` is an `int`, it can't concatenate a number and a string and give you a number.

Python doesn't constrain the type, and it can't make an assumption for what type you want. Is `x = 5 + "horse"` supposed to be a number? A string? Python doesn't know. So it errors.

In this case, `System.out.println(5 + "horse");`, Java interprets the arguments as a string concatentation, and prints out "5horse" as your result. Or, more usefully, `System.out.println(5 + " ");` will print a space after your "5".

What does `System.out.println(5 + "10");` print? 510, or 15? How about `System.out.println(5 + 10);`?

Defining Functions in Java



Video link

In languages like Python, functions can be declared anywhere, even outside of functions. For example, the code below declares a function that returns the larger of two arguments, and then uses this function to compute and print the larger of the numbers 8 and 10:

```
def larger(x, y):
    if x > y:
        return x
    return y

print(larger(8, 10))
```

Since all Java code is part of a class, we must define functions so that they belong to some class. Functions that are part of a class are commonly called "methods". We will use the terms interchangably throughout the course. The equivalent Java program to the code above is as follows:

```java
public class LargerDemo {
    public static int larger(int x, int y) {
        if (x > y) {
            return x;
        }
        return y;
    }

    public static void main(String[] args) {
        System.out.println(larger(8, 10));
    }
}
```

The new piece of syntax here is that we declared our method using the keywords `public static`, which is a very rough analog of Python's `def` keyword. We will see alternate ways to declare methods in the next chapter.

The Java code given here certainly seems much more verbose! You might think that this sort of programming language will slow you down, and indeed it will, in the short term. Think of all of this stuff as safety equipment that we don't yet understand. When we're building small programs, it all seems superfluous. However, when we get to building large programs, we'll grow to appreciate all of the added complexity.

As an analogy, programming in Python can be a bit like Dan Osman free-soloing Lover's Leap. It can be very fast, but dangerous. Java, by contrast is more like using ropes, helmets, etc. as in this video.

Code Style, Comments, Javadoc

Code can be beautiful in many ways. It can be concise. It can be clever. It can be efficient. One of the least appreciated aspects of code by novices is code style. When you program as a novice, you are often single mindedly intent on getting it to work, without regard to ever looking at it again or having to maintain it over a long period of time.

In this course, we'll work hard to try to keep our code readable. Some of the most important features of good coding style are:

- Consistent style (spacing, variable naming, brace style, etc)
- Size (lines that are not too wide, source files that are not too long)
- Descriptive naming (variables, functions, classes), e.g. variables or functions with names like `year` or `getUserName` instead of `x` or `f`.
- Avoidance of repetitive code: You should almost never have two significant blocks of code that are nearly identical except for a few changes.
- Comments where appropriate. Line comments in Java use the `//` delimiter. Block

(a.k.a. multi-line comments) comments use `/*` and `*/`.

The golden rule is this: Write your code so that it is easy for a stranger to understand.

Here is the course's official style guide. It's worth taking a look!

Often, we are willing to incur slight performance penalties, just so that our code is simpler to grok. We will highlight examples in later chapters.

Comments

We encourage you to write code that is self-documenting, i.e. by picking variable names and function names that make it easy to know exactly what's going on. However, this is not always enough. For example, if you are implementing a complex algorithm, you may need to add comments to describe your code. Your use of comments should be judicious. Through experience and exposure to others' code, you will get a feeling for when comments are most appropriate.

One special note is that all of your methods and almost all of your classes should be described in a comment using the so-called Javadoc format. In a Javadoc comment, the block comment starts with an extra asterisk, e.g. `/**`, and the comment often (but not always) contains descriptive tags. We won't discuss these tags in this textbook, but see the link above for a description of how they work.

As an example without tags:

```
public class LargerDemo {
    /** Returns the larger of x and y. */
    public static int larger(int x, int y) {
        if (x > y) {
            return x;
        }
        return y;
    }

    public static void main(String[] args) {
        System.out.println(larger(8, 10));
    }
}
```

The widely used javadoc tool can be used to generate HTML descriptions of your code. We'll see examples in a later chapter.

What Next

At the end of each chapter, there will be links letting you know what exercises (if any) you can complete with the material covered so far, listed in the order that you should complete them.

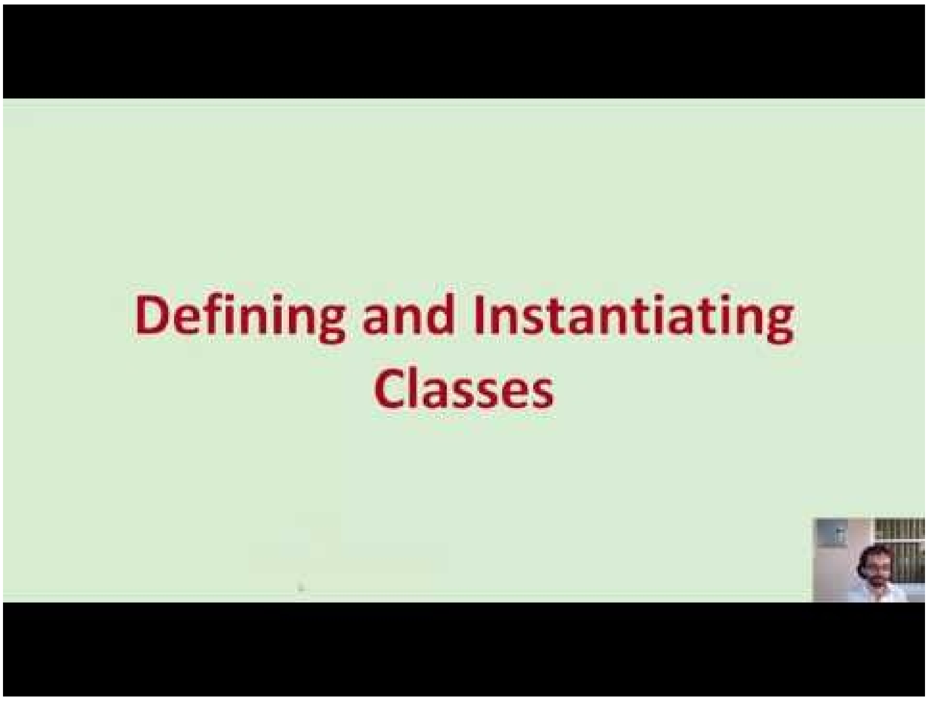
- Homework 0
- Lab 1b
- Lab 1
- Discussion 1

Defining and Using Classes

If you do not have prior Java experience, we recommend that you work through the exercises in HW0 before reading this chapter. It will cover various syntax issues that we will not discuss in the book.

Static vs. Non-Static Methods

Static Methods



Video link

All code in Java must be part of a class (or something similar to a class, which we'll learn about later). Most code is written inside of methods. Let's consider an example:

```
public class Dog {
    public static void makeNoise() {
        System.out.println("Bark!");
    }
}
```

If we try running the `Dog` class, we'll simply get an error message:

```
$ java Dog
Error: Main method not found in class Dog, please define the main method as:
       public static void main(String[] args)
```

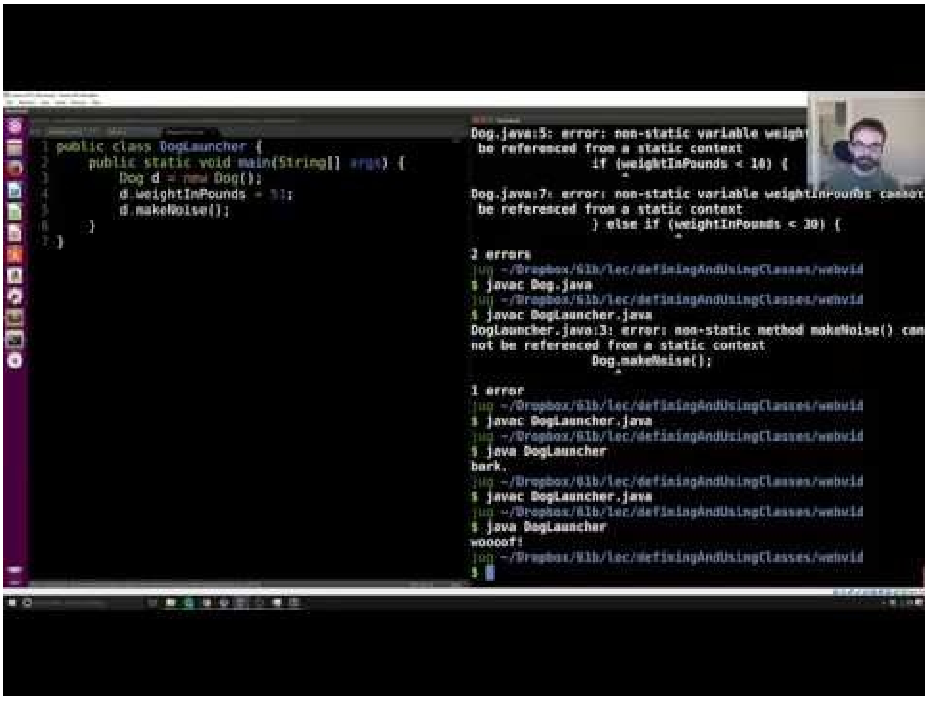
The `Dog` class we've defined doesn't do anything. We've simply defined something that `Dog` **can** do, namely make noise. To actually run the class, we'd either need to add a main method to the `Dog` class, as we saw in chapter 1.1. Or we could create a separate `DogLauncher` class that runs methods from the `Dog` class. For example, consider the program below:

```
public class DogLauncher {
    public static void main(String[] args) {
        Dog.makeNoise();
    }
}
```

```
$ java DogLauncher
Bark!
```

A class that uses another class is sometimes called a "client" of that class, i.e. `DogLauncher` is a client of `Dog`. Neither of the two techniques is better: Adding a main method to `Dog` may be better in some situations, and creating a client class like `DogLauncher` may be better in others. The relative advantages of each approach will become clear as we gain additional practice throughout the course.

Instance Variables and Object Instantiation



[Video link]

Not all dogs are alike. Some dogs like to yap incessantly, while others bellow sonorously, bringing joy to all who hear their glorious call. Often, we write programs to mimic features of the universe we inhabit, and Java's syntax was crafted to easily allow such mimicry.

One approach to allowing us to represent the spectrum of Dogdom would be to create separate classes for each type of Dog.

```
public class TinyDog {
    public static void makeNoise() {
        System.out.println("yip yip yip yip");
    }
}

public class MalamuteDog {
    public static void makeNoise() {
        System.out.println("arooooooooooooooo!");
    }
}
```

As you should have seen in the past, classes can be instantiated, and instances can hold data. This leads to a more natural approach, where we create instances of the `Dog` class and make the behavior of the `Dog` methods contingent upon the properties of the specific `Dog`. To make this more concrete, consider the class below:

```java
public class Dog {
    public int weightInPounds;

    public void makeNoise() {
        if (weightInPounds < 10) {
            System.out.println("yipyipyip!");
        } else if (weightInPounds < 30) {
            System.out.println("bark. bark.");
        } else {
            System.out.println("woof!");
        }
    }
}
```

As an example of using such a Dog, consider:

```java
public class DogLauncher {
    public static void main(String[] args) {
        Dog d;
        d = new Dog();
        d.weightInPounds = 20;
        d.makeNoise();
    }
}
```

When run, this program will create a `Dog` with weight 20, and that `Dog` will soon let out a nice "bark. bark.".

Some key observations and terminology:

- An `Object` in Java is an instance of any class.
- The `Dog` class has its own variables, also known as *instance variables* or *non-static variables*. These must be declared inside the class, unlike languages like Python or Matlab, where new variables can be added at runtime.
- The method that we created in the `Dog` class did not have the `static` keyword. We call such methods *instance methods* or *non-static methods*.
- To call the `makeNoise` method, we had to first *instantiate* a `Dog` using the `new` keyword, and then make a specific `Dog` bark. In other words, we called `d.makeNoise()` instead of `Dog.makeNoise()`.
- Once an object has been instantiated, it can be *assigned* to a *declared* variable of the appropriate type, e.g. `d = new Dog();`
- Variables and methods of a class are also called *members* of a class.
- Members of a class are accessed using *dot notation*.

Constructors in Java

As you've hopefully seen before, we usually construct objects in object oriented languages using a *constructor*:

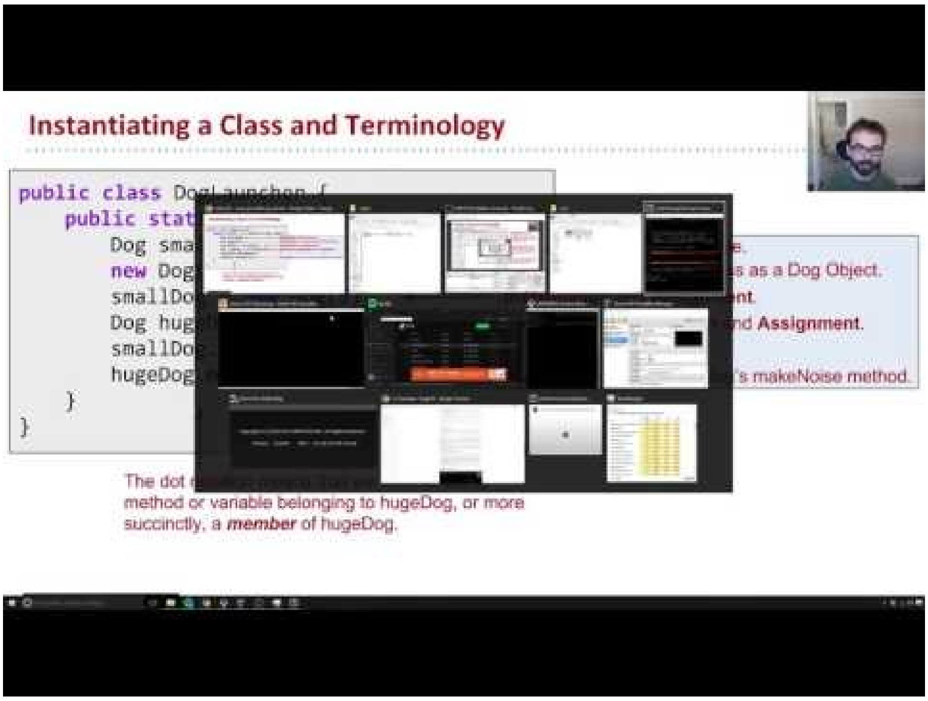
```
public class DogLauncher {
    public static void main(String[] args) {
        Dog d = new Dog(20);
        d.makeNoise();
    }
}
```

Here, the instantiation is parameterized, saving us the time and messiness of manually typing out potentially many instance variable assignments. To enable such syntax, we need only add a "constructor" to our Dog class, as shown below:

```
public class Dog {
    public int weightInPounds;

    public Dog(int w) {
        weightInPounds = w;
    }

    public void makeNoise() {
        if (weightInPounds < 10) {
            System.out.println("yipyipyip!");
        } else if (weightInPounds < 30) {
            System.out.println("bark. bark.");
        } else {
            System.out.println("woof!");
        }
    }
}
```

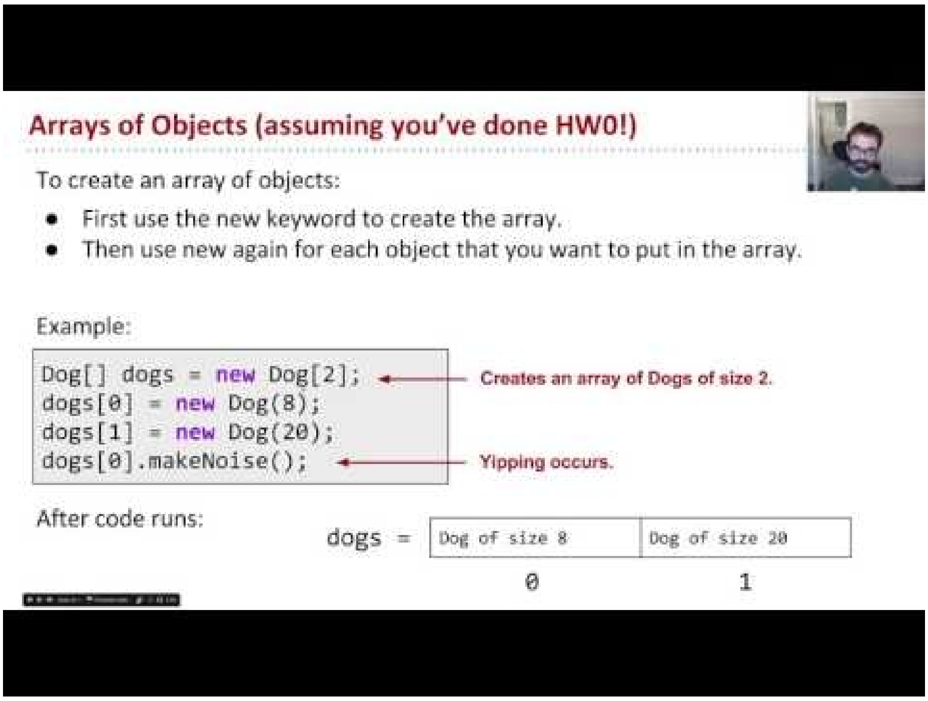
The constructor with signature `public Dog(int w)` will be invoked anytime that we try to create a `Dog` using the `new` keyword and a single integer parameter. For those of you coming from Python, the constructor is very similar to the `__init__` method.

Terminology Summary



[Video link]

Array Instantiation, Arrays of Objects



[Video link]

As we saw in HW0, arrays are also instantiated in Java using the new keyword. For example:

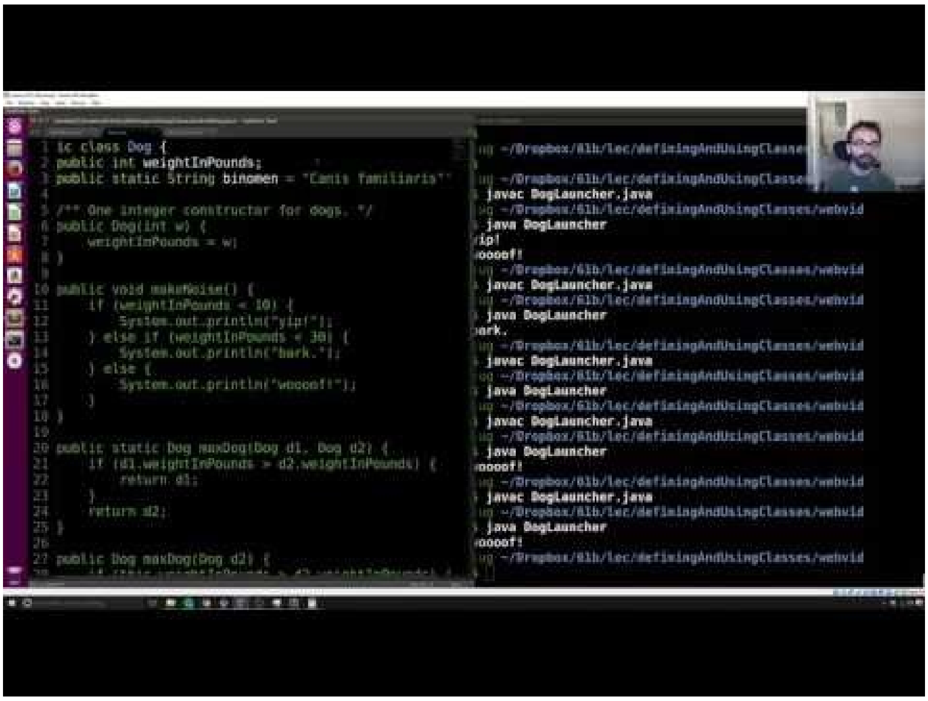
```java
public class ArrayDemo {
    public static void main(String[] args) {
        /* Create an array of five integers. */
        int[] someArray = new int[5];
        someArray[0] = 3;
        someArray[1] = 4;
    }
}
```

Similarly, we can create arrays of instantiated objects in Java, e.g.

```java
public class DogArrayDemo {
    public static void main(String[] args) {
        /* Create an array of two dogs. */
        Dog[] dogs = new Dog[2];
        dogs[0] = new Dog(8);
        dogs[1] = new Dog(20);

        /* Yipping will result, since dogs[0] has weight 8. */
        dogs[0].makeNoise();
    }
}
```

Observe that new is used in two different ways: Once to create an array that can hold two `Dog` objects, and twice to create each actual `Dog` .

Class Methods vs. Instance Methods



[Video link]

Java allows us to define two types of methods:

- Class methods, a.k.a. static methods.
- Instance methods, a.k.a. non-static methods.

Instance methods are actions that can be taken only by a specific instance of a class. Static methods are actions that are taken by the class itself. Both are useful in different circumstances. As an example of a static method, the `Math` class provides a `sqrt` method. Because it is static, we can call it as follows:

```
x = Math.sqrt(100);
```

If `sqrt` had been an instance method, we would have instead the awkward syntax below. Luckily `sqrt` is a static method so we don't have to do this in real programs.

```
Math m = new Math();
x = m.sqrt(100);
```

Sometimes, it makes sense to have a class with both instance and static methods. For example, suppose want the ability to compare two dogs. One way to do this is to add a static method for comparing Dogs.

```
public static Dog maxDog(Dog d1, Dog d2) {
    if (d1.weightInPounds > d2.weightInPounds) {
        return d1;
    }
    return d2;
}
```

This method could be invoked by, for example:

```
Dog d = new Dog(15);
Dog d2 = new Dog(100);
Dog.maxDog(d, d2);
```

Observe that we've invoked using the class name, since this method is a static method.

We could also have implemented `maxDog` as a non-static method, e.g.

```
public Dog maxDog(Dog d2) {
    if (this.weightInPounds > d2.weightInPounds) {
        return this;
    }
    return d2;
}
```

Above, we use the keyword `this` to refer to the current object. This method could be invoked, for example, with:

```
Dog d = new Dog(15);
Dog d2 = new Dog(100);
d.maxDog(d2);
```

Here, we invoke the method using a specific instance variable.

Exercise 1.2.1: What would the following method do? If you're not sure, try it out.

```
public static Dog maxDog(Dog d1, Dog d2) {
    if (weightInPounds > d2.weightInPounds) {
        return this;
    }
    return d2;
}
```

Static Variables

It is occasionally useful for classes to have static variables. These are properties inherent to the class itself, rather than the instance. For example, we might record that the scientific name (or binomen) for Dogs is "Canis familiaris":

```
public class Dog {
    public int weightInPounds;
    public static String binomen = "Canis familiaris";
    ...
}
```

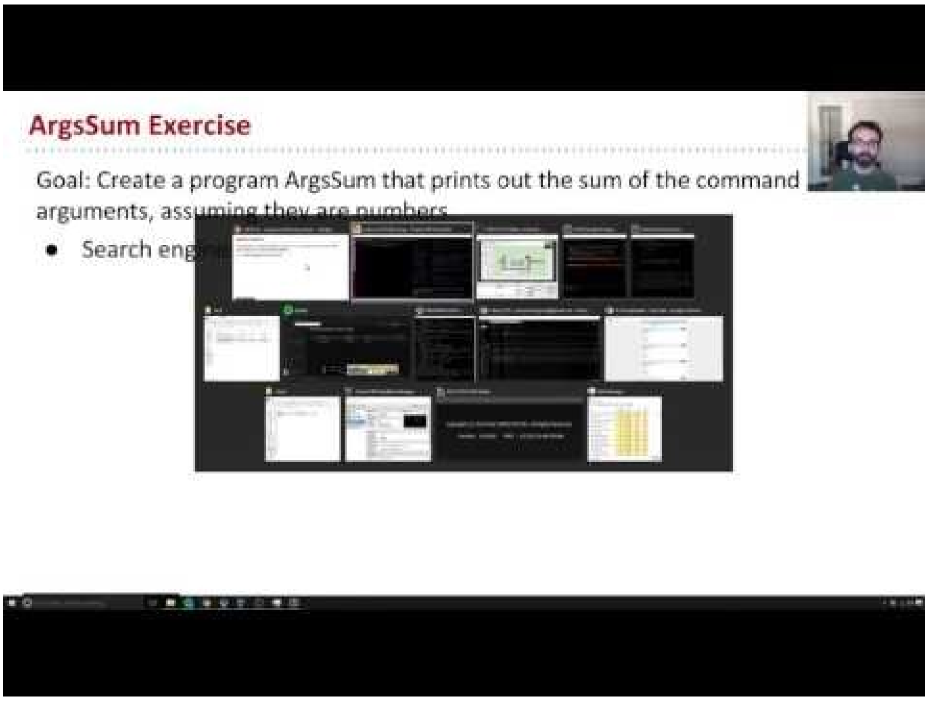
Static variables should be accessed using the name of the class rather than a specific instance, e.g. you should use `Dog.binomen`, not `d.binomen`.

While Java technically allows you to access a static variable using an instance name, it is bad style, confusing, and in my opinion an error by the Java designers.

Exercise 1.2.2: Complete this exercise:

- Video: link
- Slide: link
- Solution Video: link

public static void main(String[] args)



Video link

With what we've learned so far, it's time to demystify the declaration we've been using for the main method. Breaking it into pieces, we have:

- `public` : So far, all of our methods start with this keyword.
- `static` : It is a static method, not associated with any particular instance.
- `void` : It has no return type.
- `main` : This is the name of the method.
- `String[] args` : This is a parameter that is passed to the main method.

Command Line Arguments

Since main is called by the Java interpreter itself rather than another Java class, it is the interpreter's job to supply these arguments. They refer usually to the command line arguments. For example, consider the program `ArgsDemo` below:

```java
public class ArgsDemo {
    public static void main(String[] args) {
        System.out.println(args[0]);
    }
}
```

This program prints out the 0th command line argument, e.g.

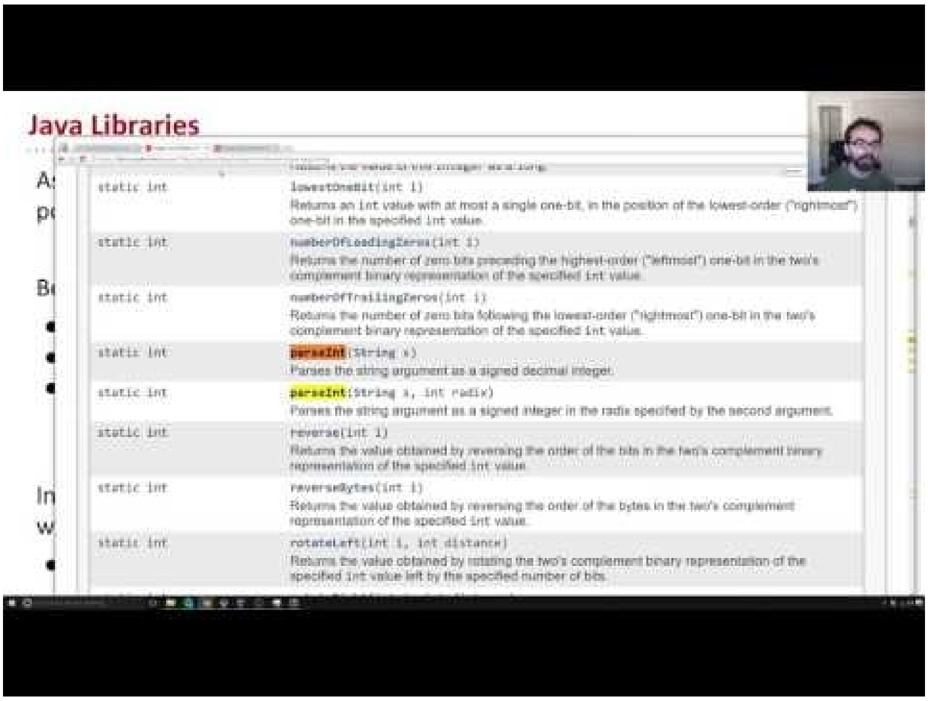
```
$ java ArgsDemo these are command line arguments
these
```

In the example above, `args` will be an array of Strings, where the entries are {"these", "are", "command", "line", "arguments"}.

Summing Command Line Arguments

Exercise 1.2.3: Try to write a program that sums up the command line arguments, assuming they are numbers. For a solution, see the webcast or the code provided on GitHub.

Using Libraries



Video link

One of the most important skills as a programmer is knowing how to find and use existing libraries. In the glorious modern era, it is often possible to save yourself tons of work and debugging by turning to the web for help.

In this course, you're welcome to do this, with the following caveats:

- Do not use libraries that we do not provide.
- Cite your sources.
- Do not search for solutions for specific homework or project problems.

For example, it's fine to search for "convert String integer Java". However, it is not OK to search for "nbody project berkeley".

For more on collaboration and academic honesty policy, see the course syllabus.

What Next

- Project 0
- Discussion 2

Lists

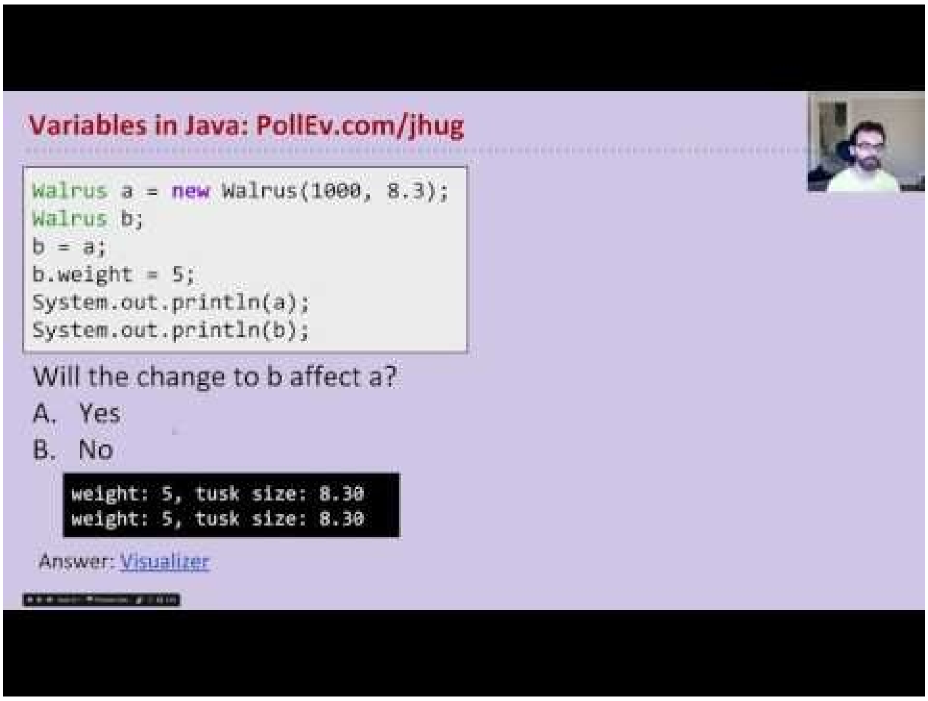
In Project 0, we use arrays to track the positions of N objects in space. One thing we would not have been able to easily do is change the number of objects after the simulation had begun. This is because arrays have a fixed size in Java that can never change.

An alternate approach would have been to use a list type. You've no doubt used a list data structure at some point in the past. For example, in Python:

```
L = [3, 5, 6]
L.append(7)
```

While Java does have a built-in List type, we're going to eschew using it for now. In this chapter, we'll build our own list from scratch, along the way learning some key features of Java.

The Mystery of the Walrus



Video link

To begin our journey, we will first ponder the profound Mystery of the Walrus.

Try to predict what happens when we run the code below. Does the change to b affect a? Hint: If you're coming from Python, Java has the same behavior.

```
Walrus a = new Walrus(1000, 8.3);
Walrus b;
b = a;
b.weight = 5;
System.out.println(a);
System.out.println(b);
```

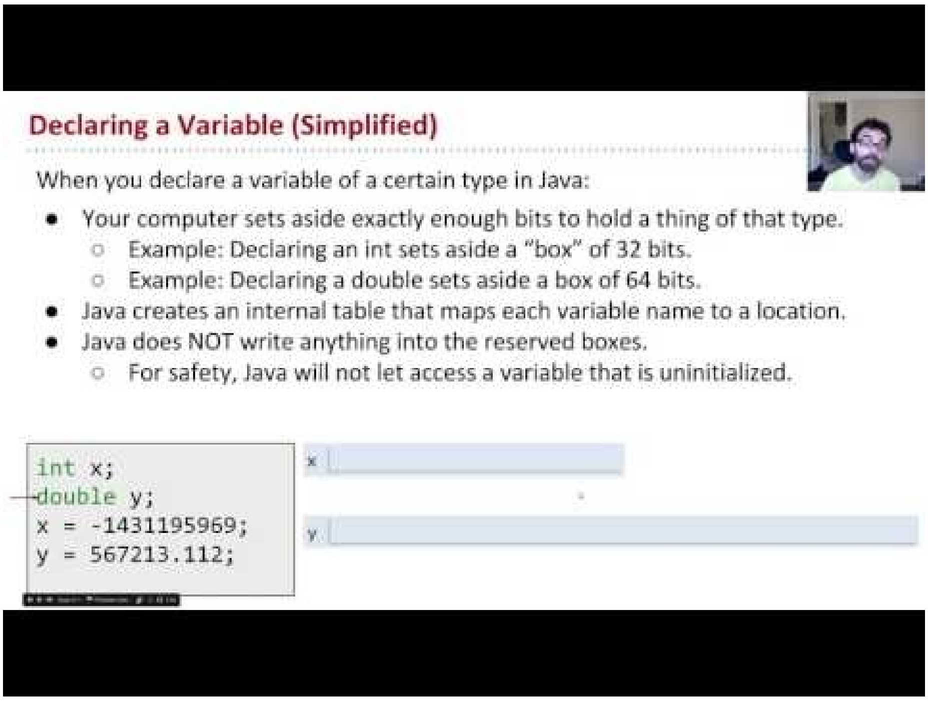
Now try to predict what happens when we run the code below. Does the change to x affect y?

```
int x = 5;
int y;
y = x;
x = 2;
System.out.println("x is: " + x);
System.out.println("y is: " + y);
```

The answer can be found here.

While subtle, the key ideas that underlie the Mystery of the Walrus will be incredibly important to the efficiency of the data structures that we'll implement in this course, and a deep understanding of this problem will also lead to safer, more reliable code.

Bits



Video link

All information in your computer is stored in *memory* as a sequence of ones and zeros. Some examples:

- 72 is often stored as 01001000
- 205.75 is often stored as 01000011 01001101 11000000 00000000
- The letter H is often stored as 01001000 (same as 72)
- The true value is often stored as 00000001

In this course, we won't spend much time talking about specific binary representations, e.g. why on earth 205.75 is stored as the seemingly random string of 32 bits above. Understanding specific representations is a topic of CS61C, the followup course to 61B.

Though we won't learn the language of binary, it's good to know that this is what is going on under the hood.

One interesting observation is that both 72 and H are stored as 01001000. This raises the question: how does a piece of Java code know how to interpret 01001000?

The answer is through types! For example, consider the code below:

```
char c = 'H';
int x = c;
System.out.println(c);
System.out.println(x);
```

If we run this code, we get:

```
H
72
```

In this case, both the x and c variables contain the same bits (well, almost...), but the Java interpreter treats them differently when printed.

In Java, there are 8 primitive types: byte, short, int, long, float, double, boolean, and char. Each has different properties that we'll discuss throughout the course, with the exception of short and float, which you'll likely never use.

Declaring a Variable (Simplified)

You can think of your computer as containing a vast number of memory bits for storing information, each of which has a unique address. Many billions of such bits are available to the modern computer.

When you declare a variable of a certain type, Java finds a contiguous block with exactly enough bits to hold a thing of that type. For example, if you declare an int, you get a block of 32 bits. If you declare a byte, you get a block of 8 bits. Each data type in Java holds a different number of bits. The exact number is not terribly important to us in this class.

For the sake of having a convenient metaphor, we'll call one of these blocks a "box" of bits.

In addition to setting aside memory, the Java interpreter also creates an entry in an internal table that maps each variable name to the location of the first bit in the box.

For example, if you declared `int x` and `double y`, then Java might decide to use bits 352 through 384 of your computer's memory to store x, and bits 20800 through 20864 to store y. The interpreter will then record that int x starts at bit 352 and y starts at bit 20800. For example, after executing the code:

```
int x;
double y;
```

We'd end up with boxes of size 32 and 64 respectively, as shown in the figure below:

The Java language provides no way for you to know the location of the box, e.g. you can't somehow find out that x is in position 352. In other words, the exact memory address is below the level of abstraction accessible to us in Java. This is unlike languages like C where you can ask the language for the exact address of a piece of data. For this reason, I have omitted the addresses from the figure above.

This feature of Java is a tradeoff! Hiding memory locations from the programmer gives you less control, which prevents you from doing certain types of optimizations. However, it also avoids a large class of very tricky programming errors. In the modern era of very low cost computing, this tradeoff is usually well worth it. As the wise Donald Knuth once said: "We should forget about small efficiencies, say about 97% of the time: premature optimization is the root of all evil".

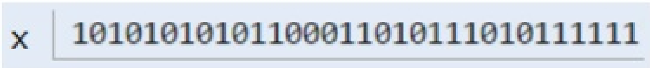
As an analogy, you do not have direct control over your heartbeat. While this restricts your ability to optimize for certain situations, it also avoids the possibility of making stupid errors like accidentally turning it off.

Java does not write anything into the reserved box when a variable is declared. In other words, there are no default values. As a result, the Java compiler prevents you from using a variable until after the box has been filled with bits using the `=` operator. For this reason, I have avoided showing any bits in the boxes in the figure above.

When you assign values to a memory box, it is filled with the bits you specify. For example, if we execute the lines:

```
x = -1431195969;
y = 567213.112;
```

Then the memory boxes from above are filled as shown below, in what I call **box notation**.



The top bits represent -1431195969, and the bottom bits represent 567213.112. Why these specific sequences of bits represent these two numbers is not important, and is a topic covered in CS61C. However, if you're curious, see integer representations and double representations on wikipedia.

Note: Memory allocation is actually somewhat more complicated than described here, and is a topic of CS 61C. However, this model is close enough to reality for our purposes in 61B.

Simplified Box Notation

While the box notation we used in the previous section is great for understanding approximately what's going on under the hood, it's not useful for practical purposes since we don't know how to interpret the binary bits.

Thus, instead of writing memory box contents in binary, we'll write them in human readable symbols. We will do this throughout the rest of the course. For example, after executing:

```
int x;
double y;
x = -1431195969;
y = 567213.112;
```

We can represent the program environment using what I call **simplified box notation**, shown below:

The Golden Rule of Equals (GRoE)

Now armed with simplified box notation, we can finally start to resolve the Mystery of the Walrus.

It turns out our Mystery has a simple solution: When you write `y = x`, you are telling the Java interpreter to copy the bits from x into y. This Golden Rule of Equals (GRoE) is the root of all truth when it comes to understanding our Walrus Mystery.

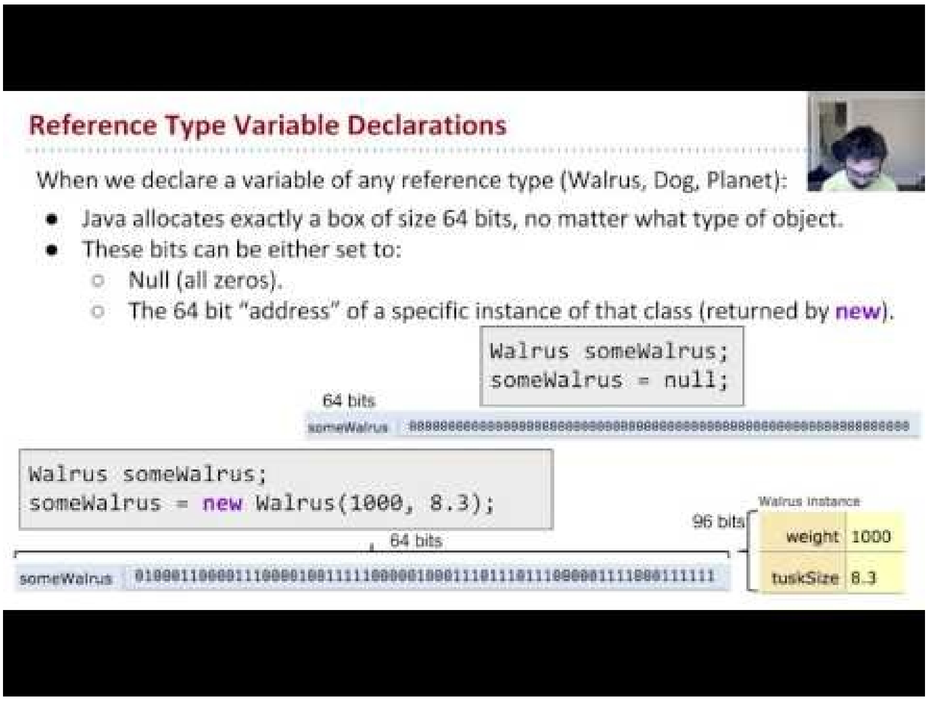
```
int x = 5;
int y;
y = x;
x = 2;
System.out.println("x is: " + x);
System.out.println("y is: " + y);
```

This simple idea of copying the bits is true for ANY assignment using `=` in Java. To see this in action, click this link.

Reference Types

Above, we said that there are 8 primitive types: byte, short, int, long, float, double, boolean, char. Everything else, including arrays, is not a primitive type but rather a `reference type`.

Object Instantiation



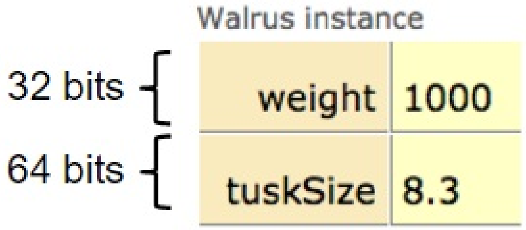
[Video link]

When we *instantiate* an Object using `new` (e.g. Dog, Walrus, Planet), Java first allocates a box for each instance variable of the class, and fills them with a default value. The constructor then usually (but not always) fills every box with some other value.

For example, if our Walrus class is:

```java
public static class Walrus {
    public int weight;
    public double tuskSize;

    public Walrus(int w, double ts) {
          weight = w;
          tuskSize = ts;
    }
}
```

And we create a Walrus using `new Walrus(1000, 8.3);`, then we end up with a Walrus consisting of two boxes of 32 and 64 bits respectively:



In real implementations of the Java programming language, there is actually some additional overhead for any object, so a Walrus takes somewhat more than 96 bits. However, for our purposes, we will ignore such overhead, since we will never interact with it directly.

The Walrus we've created is anonymous, in the sense that it has been created, but it is not stored in any variable. Let's now turn to variables that store objects.

Reference Variable Declaration

When we *declare* a variable of any reference type (Walrus, Dog, Planet, array, etc.), Java allocates a box of 64 bits, no matter what type of object.

At first glance, this might seem to lead to a Walrus Paradox. Our Walrus from the previous section required more than 64 bits to store. Furthermore, it may seem bizarre that no matter the type of object, we only get 64 bits to store it.

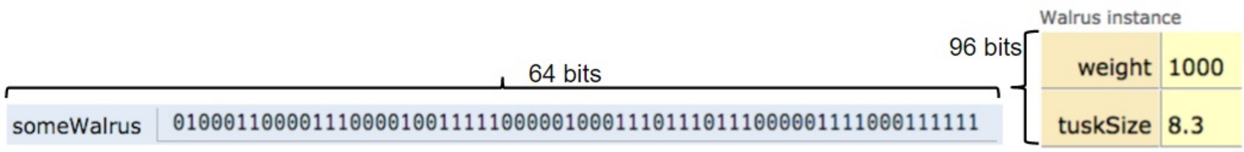
However, this problem is easily resolved with the following piece of information: the 64 bit box contains not the data about the walrus, but instead the address of the Walrus in memory.

As an example, suppose we call:

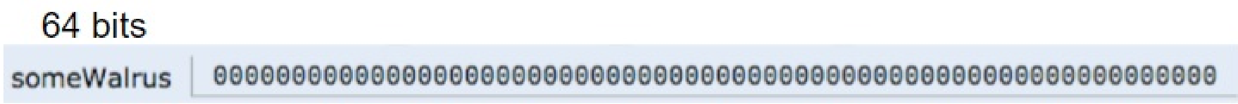
```
Walrus someWalrus;
someWalrus = new Walrus(1000, 8.3);
```

The first line creates a box of 64 bits. The second line creates a new Walrus, and the address is returned by the `new` operator. These bits are then copied into the `someWalrus` box according to the GRoE.

If we imagine our Walrus weight is stored starting at bit `5051956592385990207` of memory, and tuskSize starts at bit `5051956592385990239`, we might store `5051956592385990207` in the Walrus variable. In binary, `5051956592385990207` is represented by the 64 bits `0100011000011100001001111100000100011101110111000001111000111111`, giving us in box notation:



We can also assign the special value `null` to a reference variable, corresponding to all zeros.



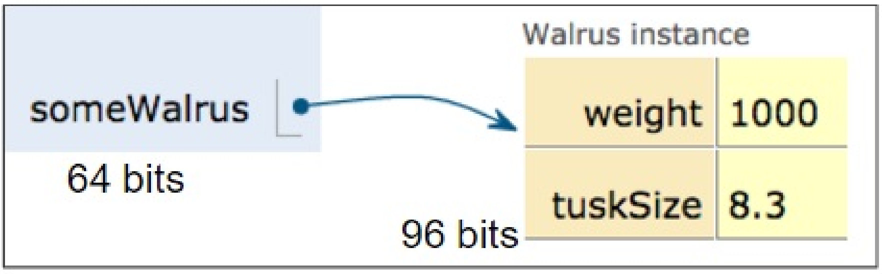
Box and Pointer Notation

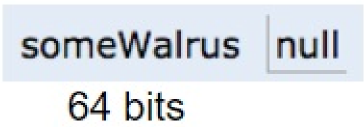
Just as before, it's hard to interpret a bunch of bits inside a reference variable, so we'll create a simplified box notation for reference variable as follows:

- If an address is all zeros, we will represent it with null.
- A non-zero address will be represented by an **arrow** pointing at an object instantiation.

This is also sometimes called "box and pointer" notation.

For the examples from the previous section, we'd have:



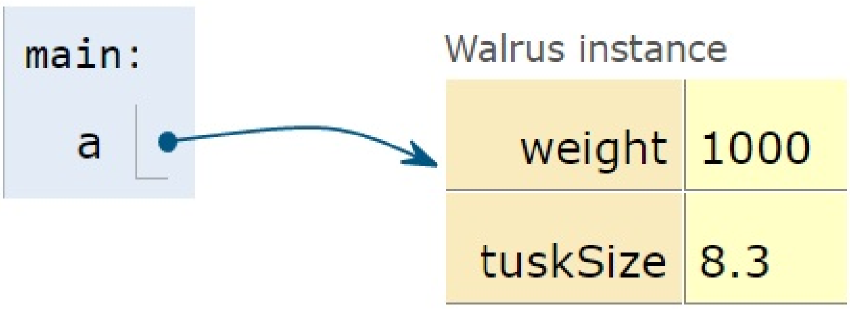


Resolving the Mystery of the Walrus

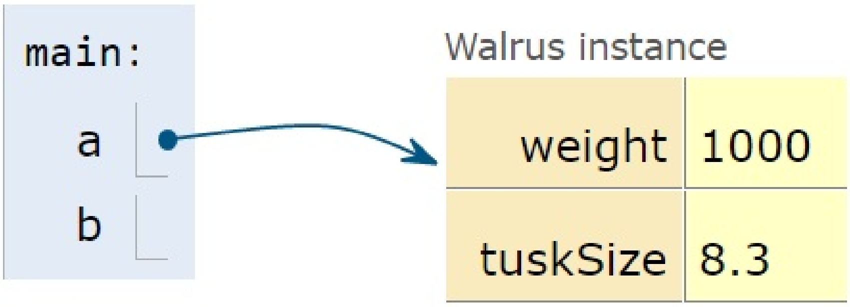
We're now finally ready to resolve, fully and completely, the Mystery of the Walrus.

```
Walrus a = new Walrus(1000, 8.3);
Walrus b;
b = a;
```

After the first line is executed, we have:

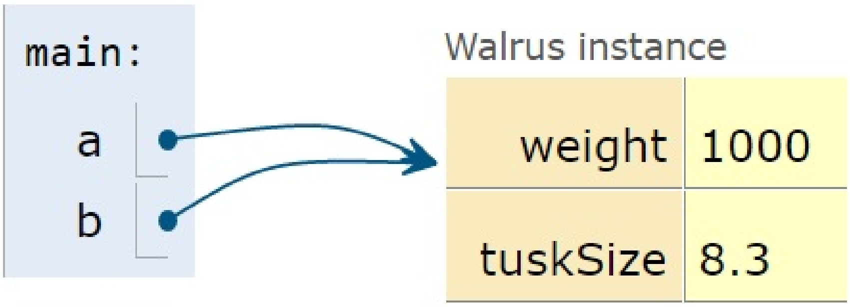


After the second line is executed, we have:



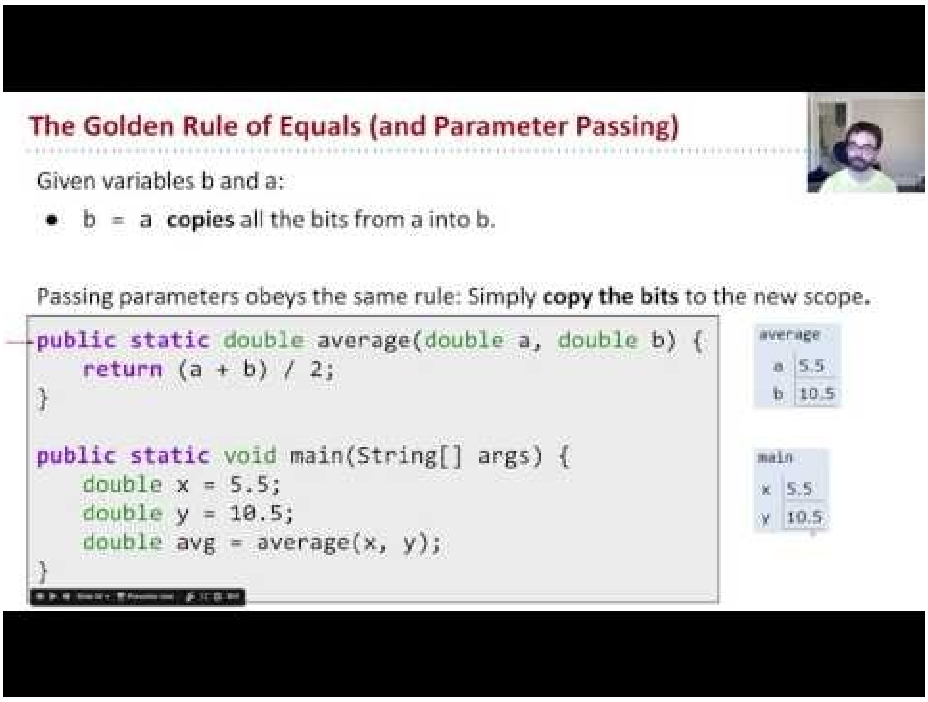
Note that above, b is undefined, not null.

According to the GRoE, the final line simply copies the bits in the `a` box into the `b` box. Or in terms of our visual metaphor, this means that b will copy exactly the arrow in a and now show an arrow pointing at the same object.



And that's it. There's no more complexity than this.

Parameter Passing



Video link

When you pass parameters to a function, you are also simply copying the bits. In other words, the GRoE also applies to parameter passing. Copying the bits is usually called "pass by value". In Java, we **always** pass by value.

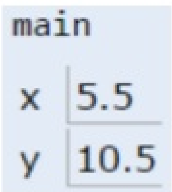
For example, consider the function below:

```
public static double average(double a, double b) {
    return (a + b) / 2;
}
```

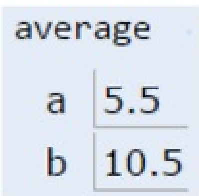
Suppose we invoke this function as shown below:

```
public static void main(String[] args) {
    double x = 5.5;
    double y = 10.5;
    double avg = average(x, y);
}
```

After executing the first two lines of this function, the main method will have two boxes labeled `x` and `y` containing the values shown below:

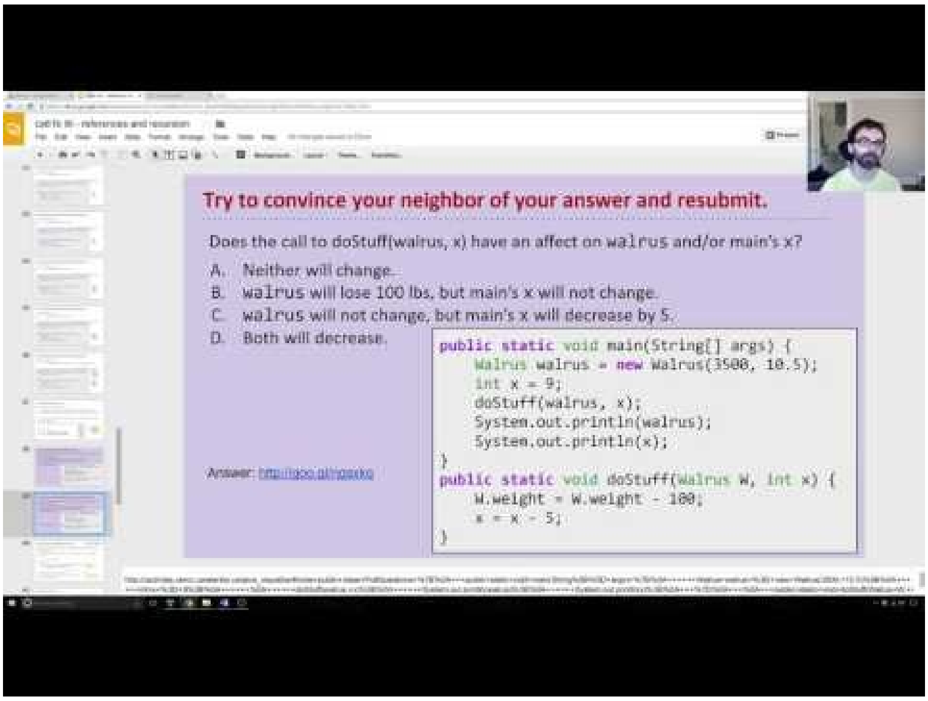


When the function is invoked, the `average` function has its **own** scope with two new boxes labeled as `a` and `b`, and the bits are simply *copied* in. This copying of bits is what we refer to when we say "pass by value".



If the `average` function were to change `a`, then `x` in main would be unchanged, since the GRoE tells us that we'd simply be filling in the box labeled `a` with new bits.

Test Your Understanding



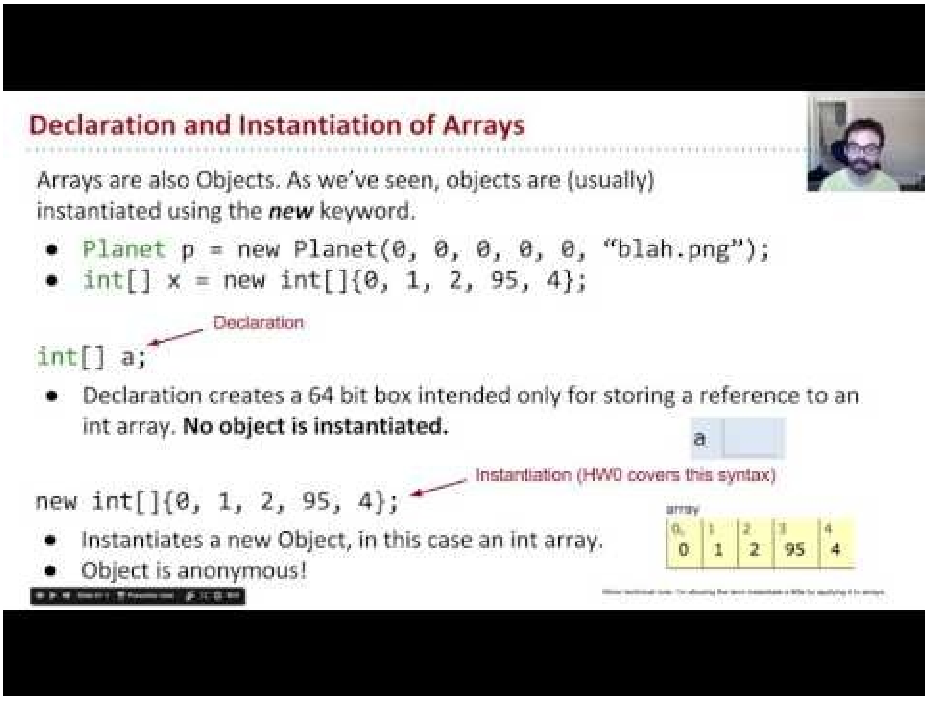
[Video link](#)

Exercise 2.1.1: Suppose we have the code below:

```
public class PassByValueFigure {
    public static void main(String[] args) {
           Walrus walrus = new Walrus(3500, 10.5);
           int x = 9;

           doStuff(walrus, x);
           System.out.println(walrus);
           System.out.println(x);
    }

    public static void doStuff(Walrus W, int x) {
           W.weight = W.weight - 100;
           x = x - 5;
    }
}
```

Does the call to `doStuff` have an effect on walrus and/or x? Hint: We only need to know the GRoE to solve this problem.

Instantiation of Arrays



[Video link]

As mentioned above, variables that store arrays are reference variables just like any other. As an example, consider the declarations below:

```
int[] x;
Planet[] planets;
```

Both of these declarations create memory boxes of 64 bits. `x` can only hold the address of an `int` array, and `planets` can only hold the address of a `Planet` array.

Instantiating an array is very similar to instantiating an object. For example, if we create an integer array of size 5 as shown below:

```
x = new int[]{0, 1, 2, 95, 4};
```

Then the `new` keyword creates 5 boxes of 32 bits each and returns the address of the overall object for assignment to x.

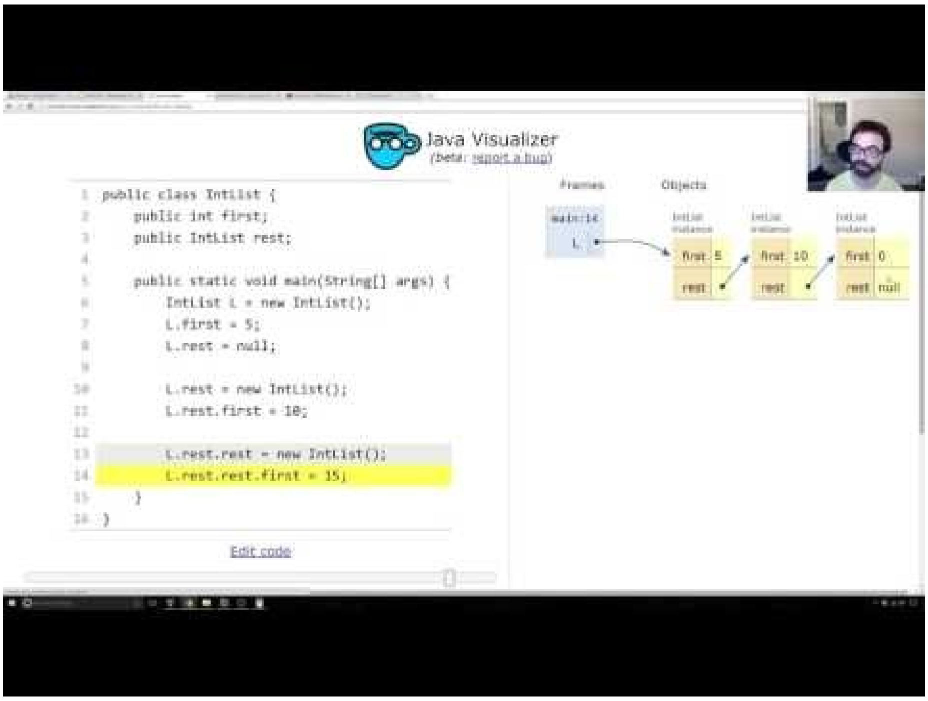
Objects can be lost if you lose the bits corresponding to the address. For example if the only copy of the address of a particular Walrus is stored in `x`, then `x = null` will cause you to permanently lose this Walrus. This isn't necessarily a bad thing, since you'll often decide you're done with an object, and thus it's safe to simply throw away the reference. We'll see this when we build lists later in this chapter.

The Law of the Broken Futon

You might ask yourself why we spent so much time and space covering what seems like a triviality. This is probably especially true if you have prior Java experience. The reason is that it is very easy for a student to have a half-cocked understanding of this issue, allowing them to write code, but without true comprehension of what's going on.

While this might be fine in the short term, in the long term, doing problems without full understanding may doom you to failure later down the line. There's a blog post about this so-called Law of the Broken Futon that you might find interesting.

IntLists



Video link

Now that we've truly understood the Mystery of the Walrus, we're ready to build our own list class.

It turns out that a very basic list is trivial to implement, as shown below:

```
public class IntList {
    public int first;
    public IntList rest;

    public IntList(int f, IntList r) {
        first = f;
        rest = r;
    }
}
```

You may remember something like this from 61a called a "Linked List".

Such a list is ugly to use. For example, if we want to make a list of the numbers 5, 10, and 15, we can either do:

```
IntList L = new IntList(5, null);
L.rest = new IntList(10, null);
L.rest.rest = new IntList(15, null);
```

Alternately, we could build our list backwards, yielding slightly nicer but harder to understand code:

```
IntList L = new IntList(15, null);
L = new IntList(10, L);
L = new IntList(5, L);
```

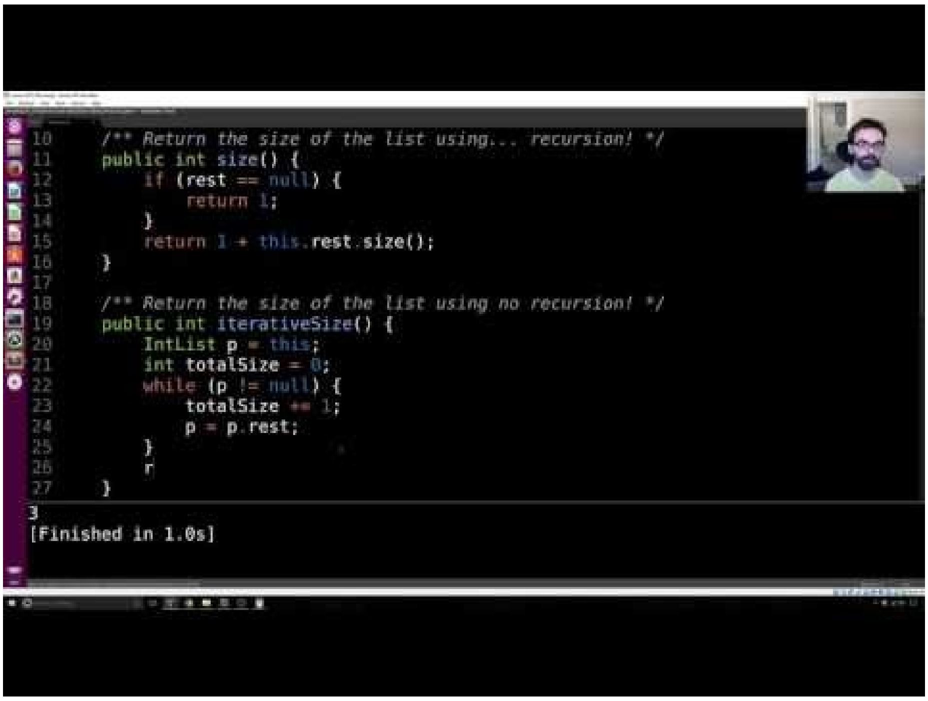
While you could in principle use the IntList to store any list of integers, the resulting code would be rather ugly and prone to errors. We'll adopt the usual object oriented programming strategy of adding helper methods to our class to perform basic tasks.

size and iterativeSize

We'd like to add a method `size` to the `IntList` class so that if you call `L.size()`, you get back the number of items in `L`.

Consider writing a `size` and `iterativeSize` method before reading the rest of this chapter. `size` should use recursion, and `iterativeSize` should not. You'll probably learn more by trying on your own before seeing how I do it. The two videos provide a live demonstration of how one might implement these methods.

[Video link]



[Video link]

My `size` method is as shown below:

```java
/** Return the size of the list using... recursion! */
public int size() {
    if (rest == null) {
        return 1;
    }
    return 1 + this.rest.size();
}
```

The key thing to remember about recursive code is that you need a base case. In this situation, the most reasonable base case is that rest is `null`, which results in a size 1 list.

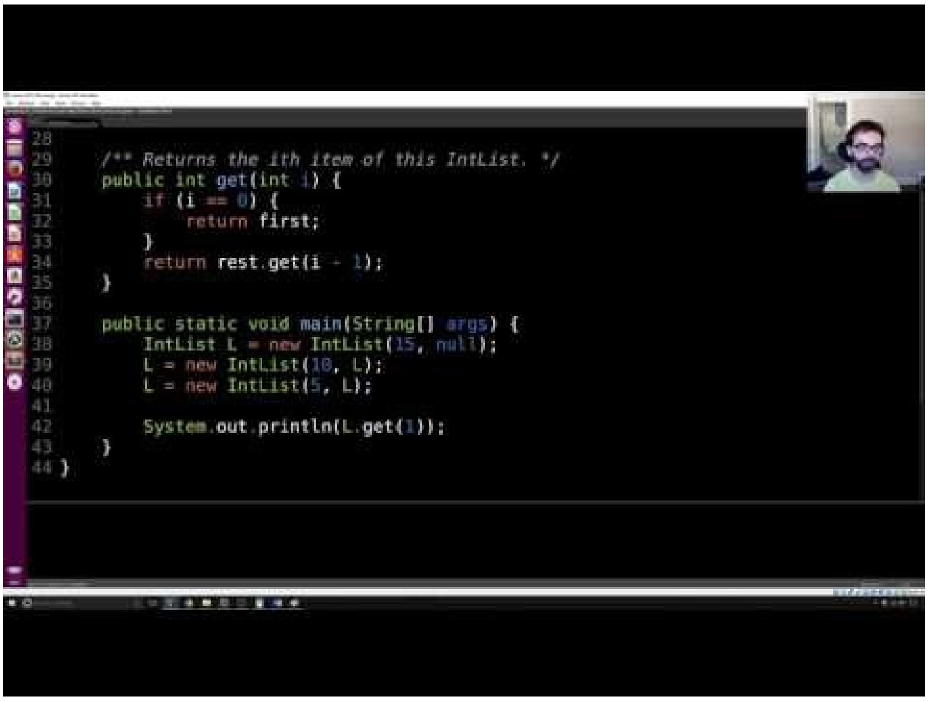
Exercise: You might wonder why we don't do something like `if (this == null) return 0;`. Why wouldn't this work?

Answer: Think about what happens when you call size. You are calling it on an object, for example L.size(). If L were null, then you would get a NullPointer error!

My `iterativeSize` method is as shown below. I recommend that when you write iterative data structure code that you use the name `p` to remind yourself that the variable is holding a pointer. You need that pointer because you can't reassign "this" in Java. The followups in this Stack Overflow Post offer a brief explanation as to why.

```java
/** Return the size of the list using no recursion! */
public int iterativeSize() {
    IntList p = this;
    int totalSize = 0;
    while (p != null) {
        totalSize += 1;
        p = p.rest;
    }
    return totalSize;
}
```

get



[Video link]

While the `size` method lets us get the size of a list, we have no easy way of getting the ith element of the list.

Exercise: Write a method `get(int i)` that returns the ith item of the list. For example, if `L` is 5 -> 10 -> 15, then `L.get(0)` should return 5, `L.get(1)` should return 10, and `L.get(2)` should return 15. It doesn't matter how your code behaves for invalid `i`, either too big or too small.

For a solution, see the lecture video above or the lectureCode repository.

Note that the method we've written takes linear time! That is, if you have a list that is 1,000,000 items long, then getting the last item is going to take much longer than it would if we had a small list. We'll see an alternate way to implement a list that will avoid this problem in a future lecture.

What Next

- Lab setup
- Lab 2

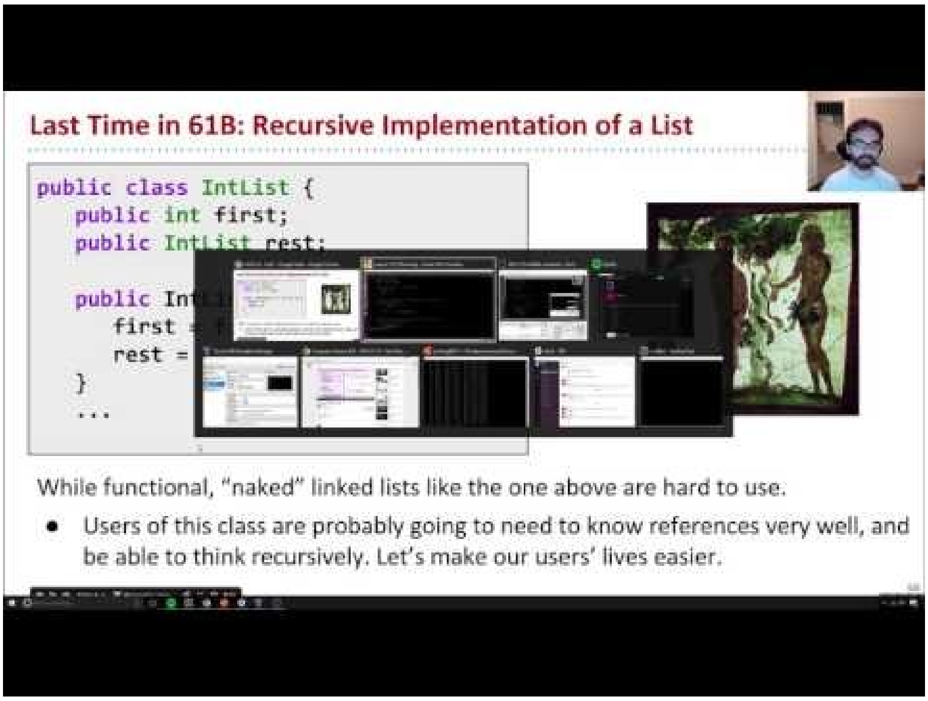
SLLists

In Chapter 2.1, we built the `IntList` class, a list data structure that can technically do all the things a list can do. However, in practice, the `IntList` suffers from the fact that it is fairly awkward to use, resulting in code that is hard to read and maintain.

Fundamentally, the issue is that the `IntList` is what I call a **naked recursive** data structure. In order to use an `IntList` correctly, the programmer must understand and utilize recursion even for simple list related tasks. This limits its usefulness to novice programmers, and potentially introduces a whole new class of tricky errors that programmers might run into, depending on what sort of helper methods are provided by the `IntList` class.

Inspired by our experience with the `IntList`, we'll now build a new class `SLList`, which much more closely resembles the list implementations that programmers use in modern languages. We'll do so by iteratively adding a sequence of improvements.

Improvement #1: Rebranding



Video link

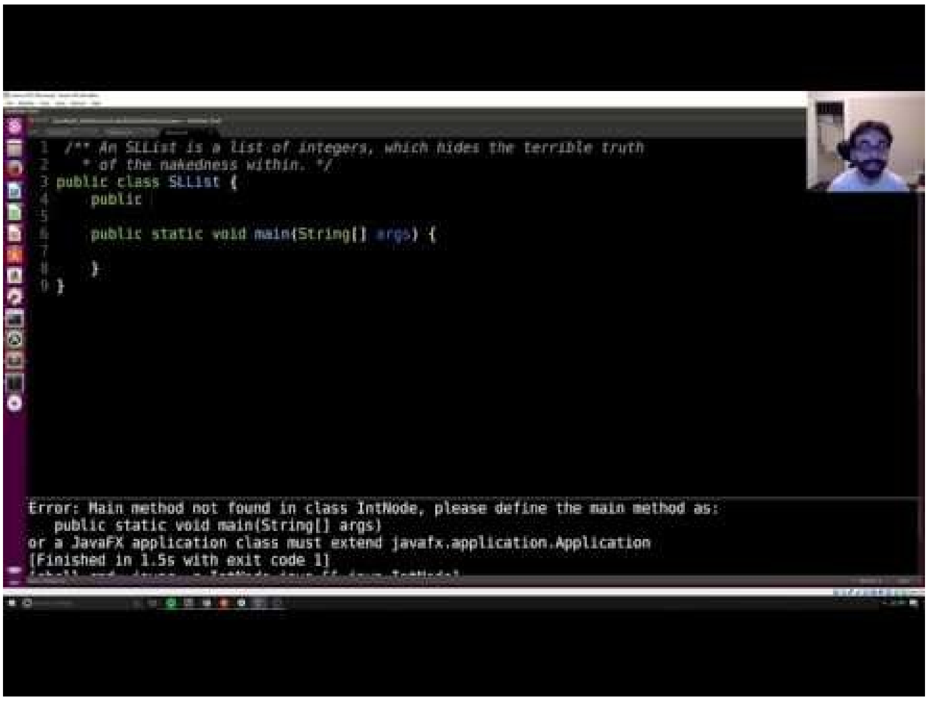
Our `IntList` class from last time was as follows, with helper methods omitted:

```java
public class IntList {
    public int first;
    public IntList rest;

    public IntList(int f, IntList r) {
        first = f;
        rest = r;
    }
...
```

Our first step will be to simply rename everything and throw away the helper methods. This probably doesn't seem like progress, but trust me, I'm a professional.

```java
public class IntNode {
    public int item;
    public IntNode next;

    public IntNode(int i, IntNode n) {
        item = i;
        next = n;
    }
}
```

Improvement #2: Bureaucracy



Video link

Knowing that `IntNodes` are hard to work with, we're going to create a separate class called `SLList` that the user will interact with. The basic class is simply:

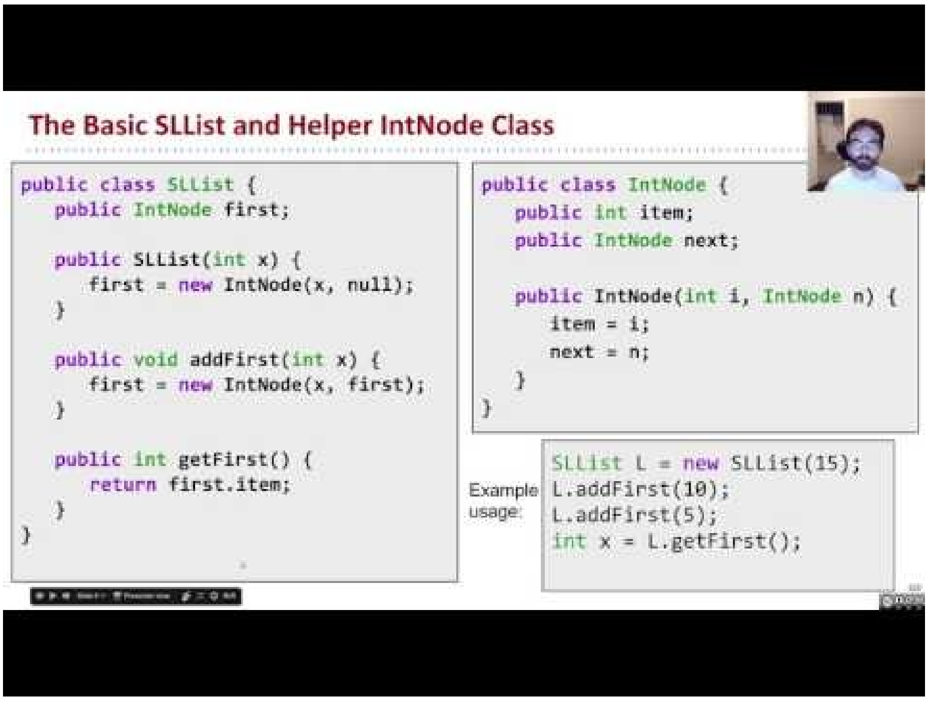
```
public class SLList {
    public IntNode first;

    public SLList(int x) {
        first = new IntNode(x, null);
    }
}
```

Already, we can get a vague sense of why a `SLList` is better. Compare the creation of an `IntList` of one item to the creation of a `SLList` of one item.

```
IntList L1 = new IntList(5, null);
SLList L2  = new SLList(5);
```

The `SLList` hides the detail that there exists a null link from the user. The `SLList` class isn't very useful yet, so let's add an `addFirst` and `getFirst` method as simple warmup methods. Consider trying to write them yourself before reading on.

addFirst and getFirst



[Video link]

`addFirst` is relatively straightforward if you understood chapter 2.1. With `IntLists`, we added to the front with the line of code `L = new IntList(5, L)`. Thus, we end up with:

```java
public class SLList {
    public IntNode first;

    public SLList(int x) {
        first = new IntNode(x, null);
    }

    /** Adds an item to the front of the list. */
    public void addFirst(int x) {
        first = new IntNode(x, first);
    }
}
```

`getFirst` is even easier. We simply return `first.item` :

```java
/** Retrieves the front item from the list. */
public int getFirst() {
    return first.item;
}
```

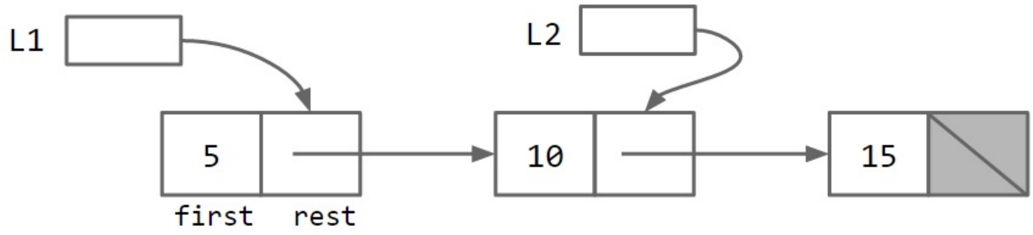
The resulting `SLList` class is much easier to use. Compare:

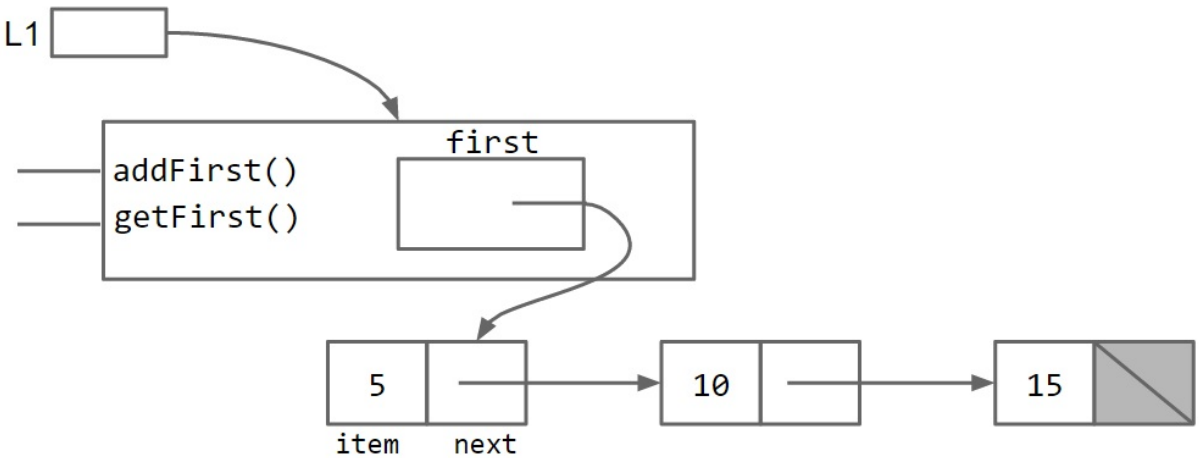
```java
SLList L = new SLList(15);
L.addFirst(10);
L.addFirst(5);
int x = L.getFirst();
```

to the `IntList` equivalent:

```java
IntList L = new IntList(15, null);
L = new IntList(10, L);
L = new IntList(5, L);
int x = L.first;
```

Comparing the two data structures visually, we have: (with the `IntList` version on top and `SLList` version below it)

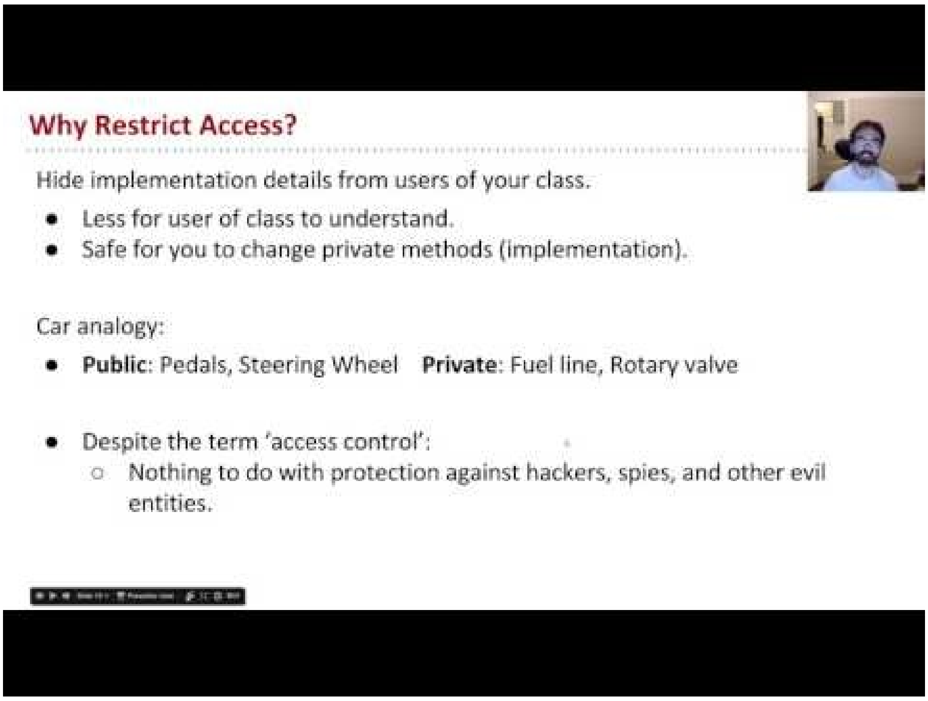




Essentially, the `SLList` class acts as a middleman between the list user and the naked recursive data structure. As suggested above in the `IntList` version, there is a potentially undesireable possibility for the `IntList` user to have variables that point to the middle of the `IntList`. As Ovid said: Mortals cannot look upon a god without dying, so perhaps it is best that the `SLList` is there to act as our intermediary.

Exercise 2.2.1: The curious reader might object and say that the `IntList` would be just as easy to use if we simply wrote an `addFirst` method. Try to write an `addFirst` method to the `IntList` class. You'll find that the resulting method is tricky as well as inefficient.

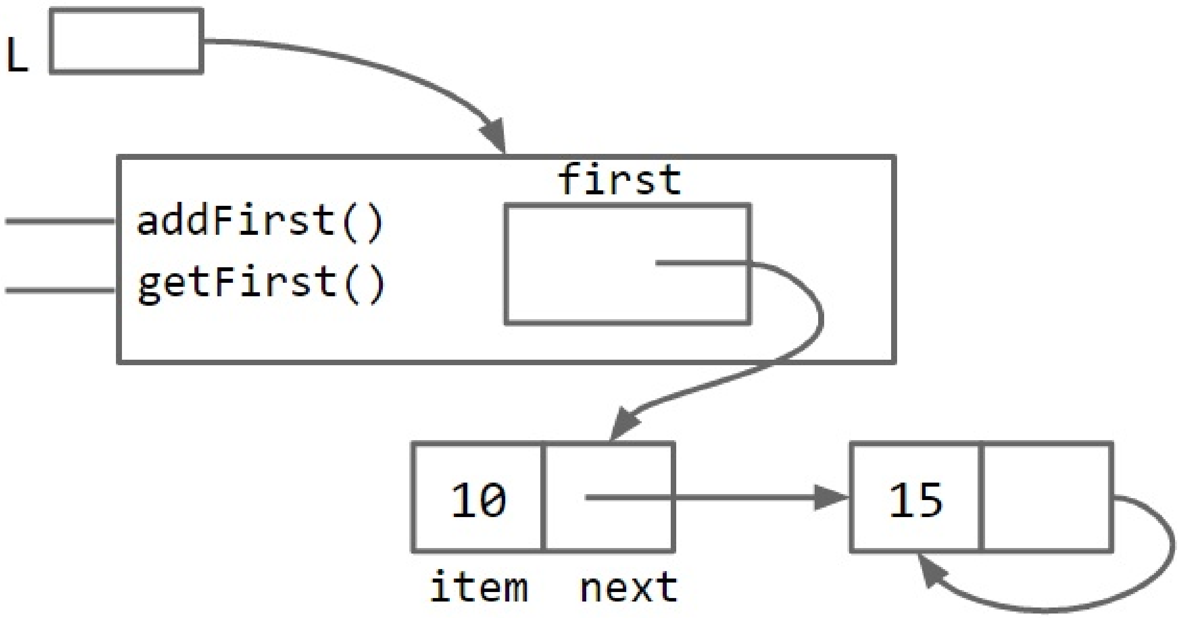
Improvement #3: Public vs. Private



Video link

Unfortunately, our `SLList` can be bypassed and the raw power of our naked data structure (with all its dangers) can be accessed. A programmer can easily modify the list directly, without going through the kid-tested, mother-approved `addFirst` method, for example:

```
SLList L = new SLList(15);
L.addFirst(10);
L.first.next.next = L.first.next;
```



This results in a malformed list with an infinite loop. To deal with this problem, we can modify the `SLList` class so that the `first` variable is declared with the `private` keyword.

```java
public class SLList {
    private IntNode first;
...
```

Private variables and methods can only be accessed by code inside the same `.java` file, e.g. in this case `SLList.java`. That means that a class like `SLLTroubleMaker` below will fail to compile, yielding a `first has private access in SLList` error.

```java
public class SLLTroubleMaker {
    public static void main(String[] args) {
        SLList L = new SLList(15);
        L.addFirst(10);
        L.first.next.next = L.first.next;
    }
}
```

By contrast, any code inside the `SLList.java` file will be able to access the `first` variable.

It may seem a little silly to restrict access. After all, the only thing that the `private` keyword does is break programs that otherwise compile. However, in large software engineering projects, the `private` keyword is an invaluable signal that certain pieces of code should be ignored (and thus need not be understood) by the end user. Likewise, the `public` keyword should be thought of as a declaration that a method is available and will work **forever** exactly as it does now.

As an analogy, a car has certain `public` features, e.g. the accelerator and brake pedals. Under the hood, there are `private` details about how these operate. In a gas powered car, the accelerator pedal might control some sort of fuel injection system, and in a battery powered car, it may adjust the amount of battery power being delivered to the motor. While the private details may vary from car to car, we expect the same behavior from all accelerator pedals. Changing these would cause great consternation from users, and quite possibly terrible accidents.

When you create a `public` member (i.e. method or variable), be careful, because you're effectively committing to supporting that member's behavior exactly as it is now, forever.

Improvement #4: Nested Classes

At the moment, we have two `.java` files: `IntNode` and `SLList`. However, the `IntNode` is really just a supporting character in the story of `SLList`.

Java provides us with the ability to embed a class declaration inside of another for just this situation. The syntax is straightforward and intuitive:

```
public class SLList {
       public class IntNode {
            public int item;
            public IntNode next;
            public IntNode(int i, IntNode n) {
                item = i;
                next = n;
            }
       }

       private IntNode first;

       public SLList(int x) {
           first = new IntNode(x, null);
       }
...
```

Having a nested class has no meaningful effect on code performance, and is simply a tool for keeping code organized. For more on nested classes, see Oracle's official documentation.

If the nested class has no need to use any of the instance methods or variables of `SLList`, you may declare the nested class `static`, as follows. Declaring a nested class as `static` means that methods inside the static class can not access any of the members of the enclosing class. In this case, it means that no method in `IntNode` would be able to access `first`, `addFirst`, or `getFirst`.

```
public class SLList {
       public static class IntNode {
            public int item;
            public IntNode next;
            public IntNode(int i, IntNode n) {
                item = i;
                next = n;
            }
       }

       private IntNode first;
...
```

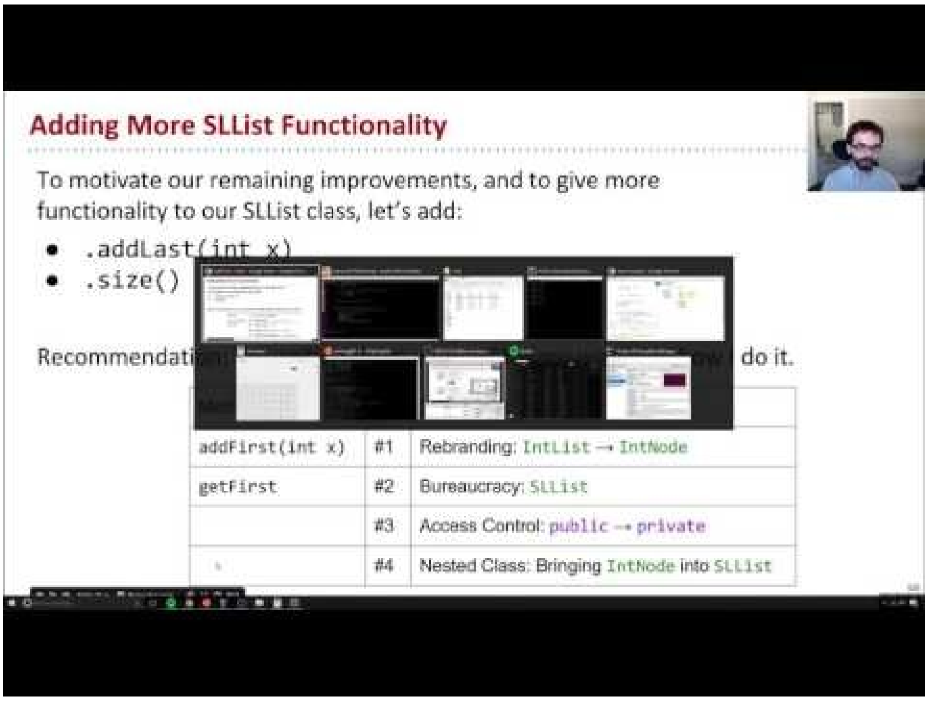
This saves a bit of memory, because each `IntNode` no longer needs to keep track of how to access its enclosing `SLList`.

Put another way, if you examine the code above, you'll see that the `IntNode` class never uses the `first` variable of `SLList`, nor any of `SLList` 's methods. As a result, we can use the static keyword, which means the `IntNode` class doesn't get a reference to its boss, saving us a small amount of memory.

If this seems a bit technical and hard to follow, try Exercise 2.2.2. A simple rule of thumb is that *if you don't use any instance members of the outer class, make the nested class static*.

Exercise 2.2.2 Delete the word `static` as few times as possible so that this program compiles (Refresh the page after clicking the link and making sure the url changed). Make sure to read the comments at the top before doing the exercise.

addLast() and size()



Video link

To motivate our remaining improvements and also demonstrate some common patterns in data structure implementation, we'll add `addLast(int x)` and `size()` methods. You're encouraged to take the starter code and try it yourself before reading on. I especially encourage you to try to write a recursive implementation of `size`, which will yield an interesting challenge.

I'll implement the `addLast` method iteratively, though you could also do it recursively. The idea is fairly straightforward, we create a pointer variable `p` and have it iterate through the list to the end.

```java
/** Adds an item to the end of the list. */
public void addLast(int x) {
    IntNode p = first;

    /* Advance p to the end of the list. */
    while (p.next != null) {
        p = p.next;
    }
    p.next = new IntNode(x, null);
}
```

By contrast, I'll implement `size` recursively. This method will be somewhat similar to the `size` method we implemented in section 2.1 for `IntList`.

The recursive call for `size` in `IntList` was straightforward: `return 1 + this.rest.size()`. For a `SLList`, this approach does not make sense. A `SLList` has no `rest` variable. Instead, we'll use a common pattern that is used with middleman classes like `SLList` -- we'll create a private helper method that interacts with the underlying naked recursive data structure.

This yields a method like the following:

```java
/** Returns the size of the list starting at IntNode p. */
private static int size(IntNode p) {
    if (p.next == null) {
        return 1;
    }

    return 1 + size(p.next);
}
```

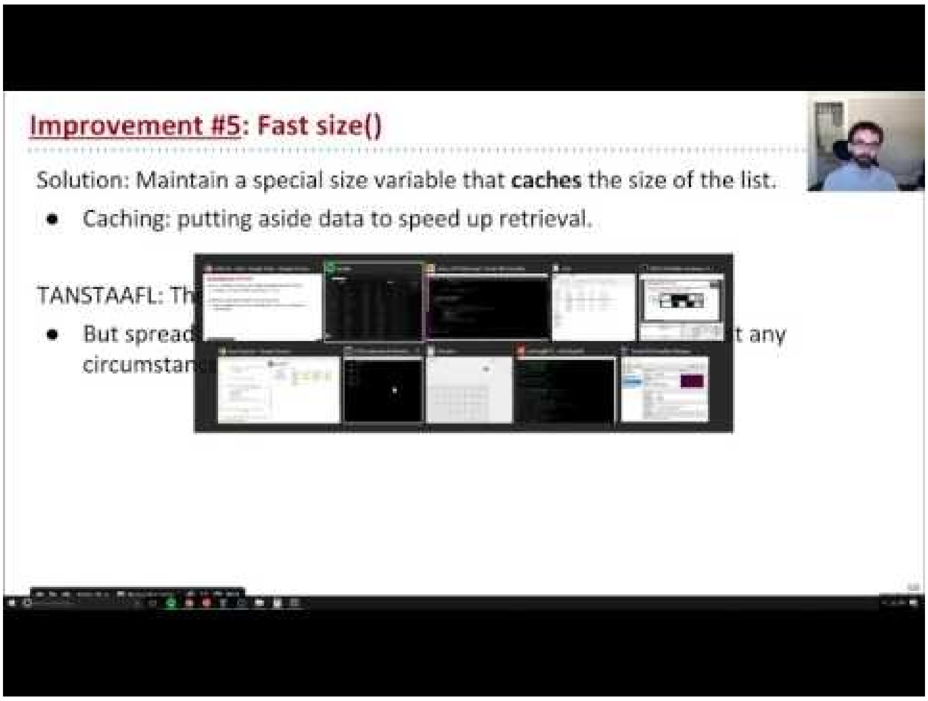
Using this method, we can easily compute the size of the entire list:

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

Here, we have two methods, both named `size`. This is allowed in Java, since they have different parameters. We say that two methods with the same name but different signatures are **overloaded**. For more on overloaded methods, see Java's official documentation.

An alternate approach is to create a non-static helper method in the `IntNode` class itself. Either approach is fine, though I personally prefer not having any methods in the `IntNode` class.

Improvement #5: Caching



Video link

Consider the `size` method we wrote above. Suppose `size` takes 2 seconds on a list of size 1,000. We expect that on a list of size 1,000,000, the `size` method will take 2,000 seconds, since the computer has to step through 1,000 times as many items in the list to reach the end. Having a `size` method that is very slow for large lists is unacceptable, since we can do better.

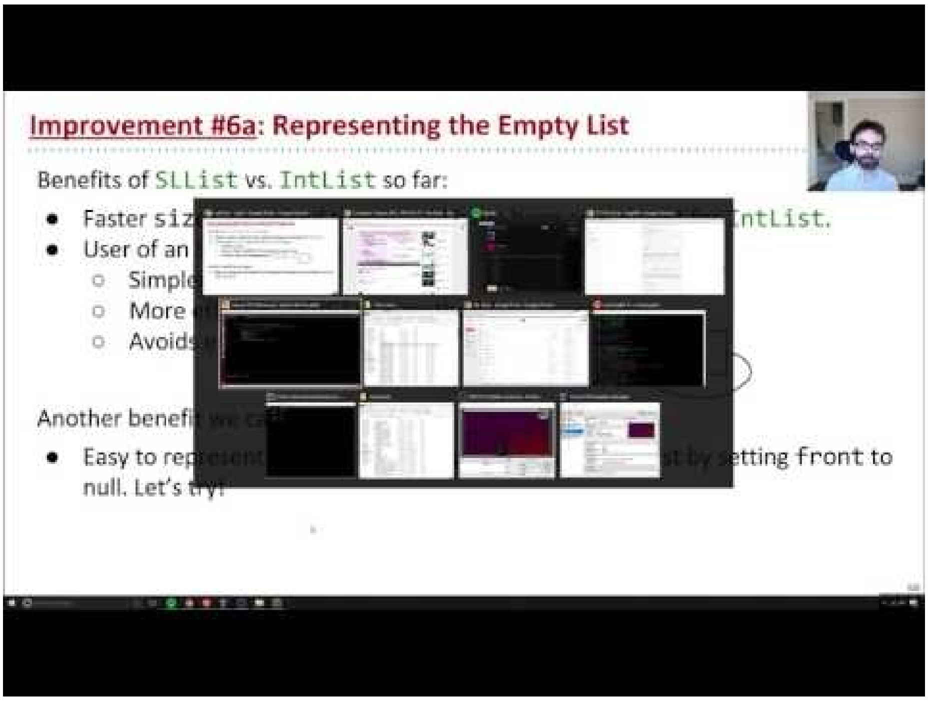
It is possible to rewrite `size` so that it takes the same amount of time, no matter how large the list.

To do so, we can simply add a `size` variable to the `SLList` class that tracks the current size, yielding the code below. This practice of saving important data to speed up retrieval is sometimes known as **caching**.

```
public class SLList {
    ... /* IntNode declaration omitted. */
    private IntNode first;
    private int size;

    public SLList(int x) {
        first = new IntNode(x, null);
        size = 1;
    }

    public void addFirst(int x) {
        first = new IntNode(x, first);
        size += 1;
    }

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

This modification makes our `size` method incredibly fast, no matter how large the list. Of course, it will also slow down our `addFirst` and `addLast` methods, and also increase the memory of usage of our class, but only by a trivial amount. In this case, the tradeoff is clearly in favor of creating a cache for size.

Improvement #6: The Empty List



Video link

Our `SLList` has a number of benefits over the simple `IntList` from chapter 2.1:

- Users of a `SLList` never see the `IntList` class.
    - Simpler to use.
    - More efficient `addFirst` method (exercise 2.2.1).
    - Avoids errors or malfeasance by `IntList` users.
- Faster `size` method than possible with `IntList` .

Another natural advantage is that we will be able to easily implement a constructor that creates an empty list. The most natural way is to set `first` to `null` if the list is empty. This yields the constructor below:

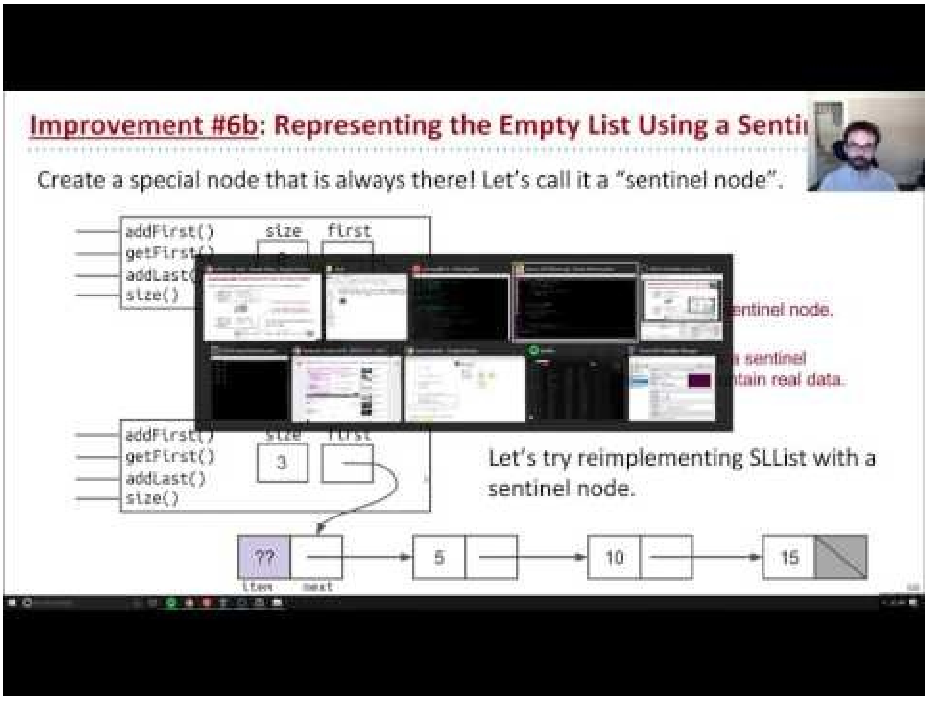
```
public SLList() {
    first = null;
    size = 0;
}
```

Unfortunately, this causes our `addLast` method to crash if we insert into an empty list. Since `first` is `null` , the attempt to access `p.next` in `while (p.next != null)` below causes a null pointer exception.

```
public void addLast(int x) {
    size += 1;
    IntNode p = first;
    while (p.next != null) {
        p = p.next;
    }

    p.next = new IntNode(x, null);
}
```

Exercise 2.2.3 Fix the `addLast` method. Starter code here.

Improvement #6b: Sentinel Nodes



Video link

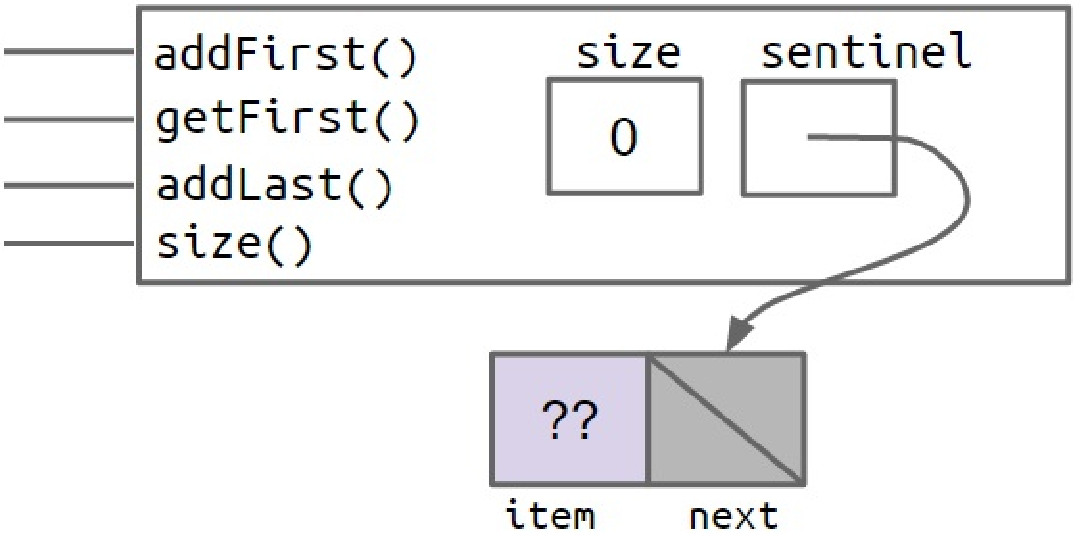
One solution to fix `addLast` is to create a special case for the empty list, as shown below:

```
public void addLast(int x) {
    size += 1;

    if (first == null) {
        first = new IntNode(x, null);
        return;
    }

    IntNode p = first;
    while (p.next != null) {
        p = p.next;
    }

    p.next = new IntNode(x, null);
}
```

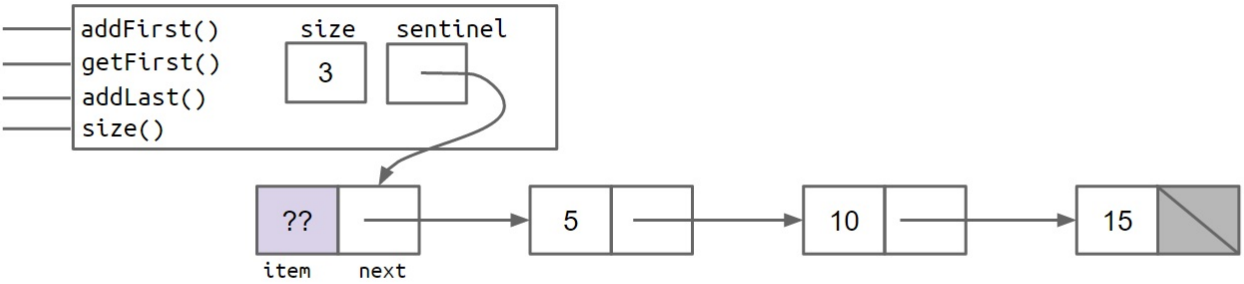
This solution works, but special case code like that shown above should be avoided when necessary. Human beings only have so much working memory, and thus we want to keep complexity under control wherever possible. For a simple data structure like the `SLList`, the number of special cases is small. More complicated data structures like trees can get much, much uglier.

A cleaner, though less obvious solution, is to make it so that all `SLLists` are the "same", even if they are empty. We can do this by creating a special node that is always there, which we will call a **sentinel node**. The sentinel node will hold a value, which we won't care about.

For example, the empty list created by `SLList L = new SLList()` would be as shown below:



And a `SLList` with the items 5, 10, and 15 would look like:



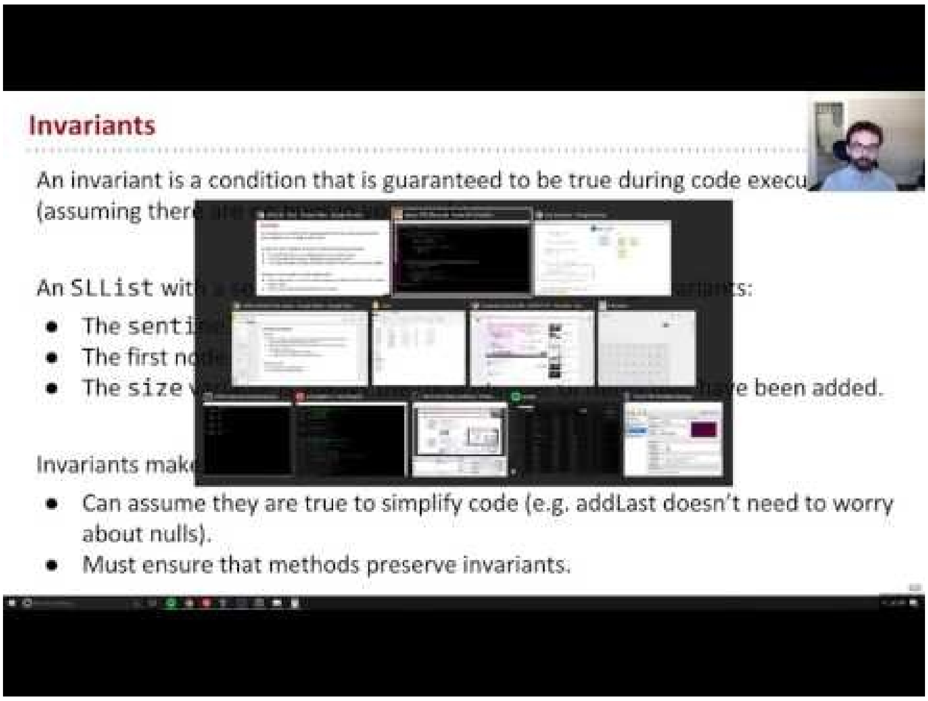
In the figures above, the lavender ?? value indicates that we don't care what value is there. Since Java does not allow us to fill in an integer with question marks, we just pick some abitrary value like -518273 or 63 or anything else.

Since a `SLList` without a sentinel has no special cases, we can simply delete the special case from our `addLast` method, yielding:

```
public void addLast(int x) {
    size += 1;
    IntNode p = sentinel;
    while (p.next != null) {
        p = p.next;
    }

    p.next = new IntNode(x, null);
}
```

As you can see, this code is much much cleaner!

Invariants



[Video link]

An invariant is a fact about a data structure that is guaranteed to be true (assuming there are no bugs in your code).

A `SLList` with a sentinel node has at least the following invariants:

- The `sentinel` reference always points to a sentinel node.
- The front item (if it exists), is always at `sentinel.next.item`.
- The `size` variable is always the total number of items that have been added.

Invariants make it easier to reason about code, and also give you specific goals to strive for in making sure your code works.

A true understanding of how convenient sentinels are will require you to really dig in and do some implementation of your own. You'll get plenty of practice in project 1. However, I recommend that you wait until after you've finished the next section of this book before beginning project 1.

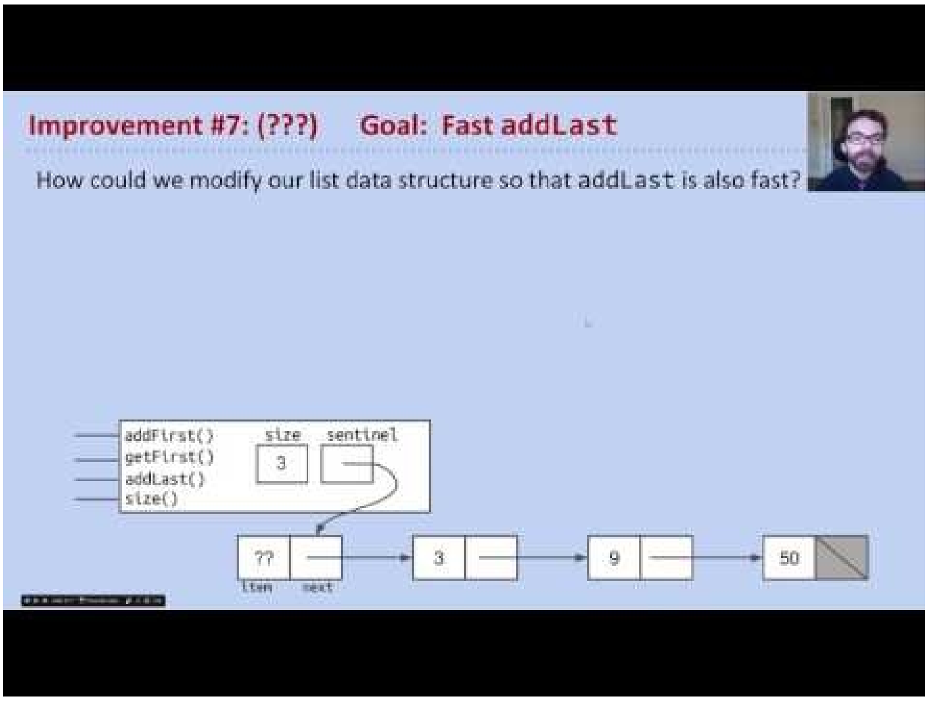
What Next

Nothing for this chapter. However, if you're taking the Berkeley course, you're welcome to now begin Lab 2.

DLLists

In Chapter 2.2, we built the `SLList` class, which was better than our earlier naked recursive `IntList` data structure. In this section, we'll wrap up our discussion of linked lists, and also start learning the foundations of arrays that we'll need for an array based list we'll call an `AList`. Along the way, we'll also reveal the secret of why we used the awkward name `SLList` in the previous chapter.

addLast



Video link

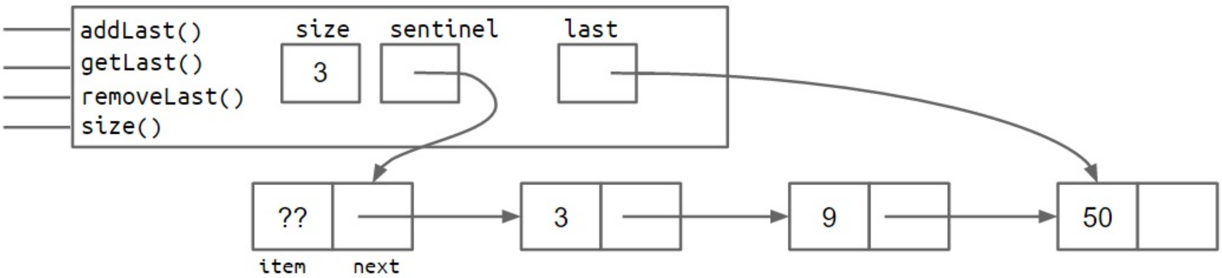
Consider the `addLast(int x)` method from the previous chapter.

```
public void addLast(int x) {
    size += 1;
    IntNode p = sentinel;
    while (p.next != null) {
        p = p.next;
    }

    p.next = new IntNode(x, null);
}
```

The issue with this method is that it is slow. For a long list, the `addLast` method has to walk through the entire list, much like we saw with the `size` method in chapter 2.2. Similarly, we can attempt to speed things up by adding a `last` variable, to speed up our code, as shown below:

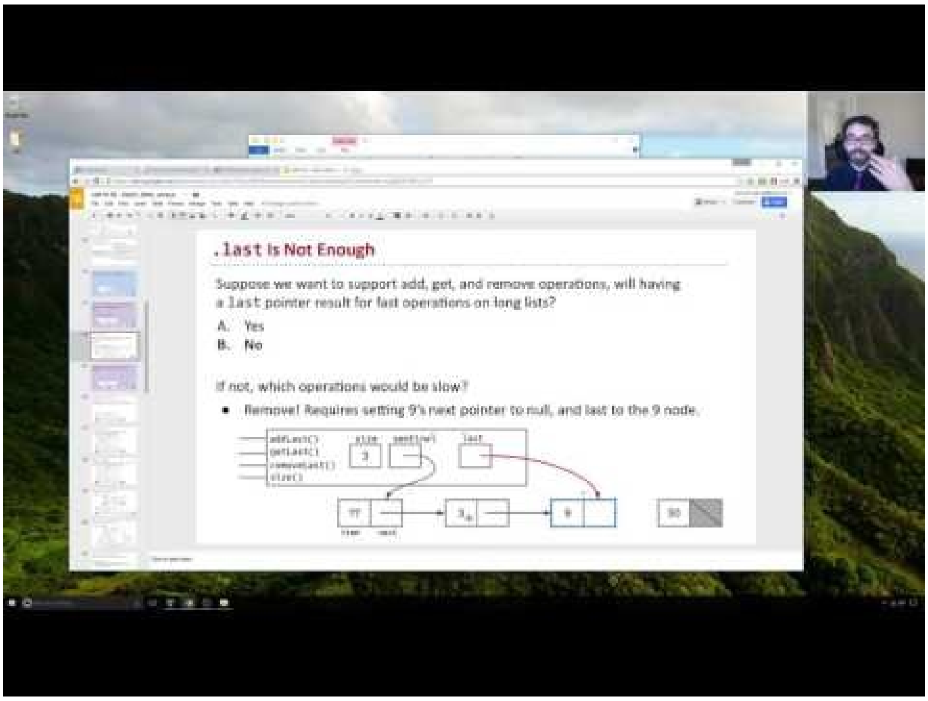
```
public class SLList {
    private IntNode sentinel;
    private IntNode last;
    private int size;

    public void addLast(int x) {
        last.next = new IntNode(x, null);
        last = last.next;
        size += 1;
    }
    ...
}
```

Exercise 2.3.1: Consider the box and pointer diagram representing the `SLList` implementation above, which includes the last pointer. Suppose that we'd like to support `addLast` , `getLast` , and `removeLast` operations. Will the structure shown support rapid `addLast` , `getLast` , and `removeLast` operations? If not, which operations are slow?



Answer 2.3.1: `addLast` and `getLast` will be fast, but `removeLast` will be slow. That's because we have no easy way to get the second-to-last node, to update the `last` pointer, after removing the last node.

SecondToLast



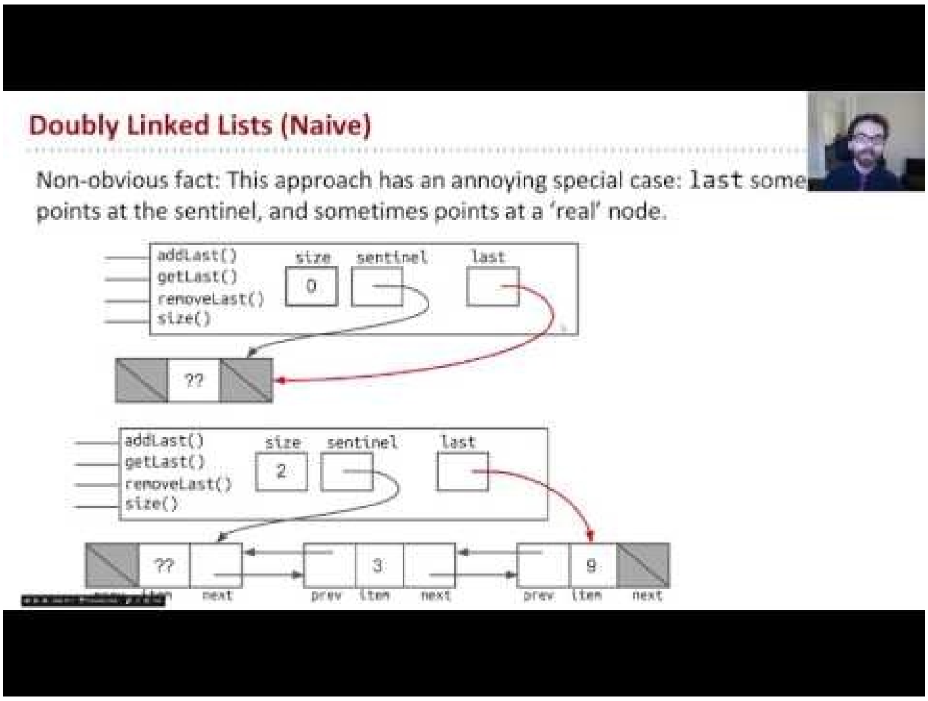
Video link

The issue with the structure from exercise 2.3.1 is that a method that removes the last item in the list will be inherently slow. This is because we need to first find the second to last item, and then set its next pointer to be null. Adding a `secondToLast` pointer will not help either, because then we'd need to find the third to last item in the list in order to make sure that `secondToLast` and `last` obey the appropriate invariants after removing the last item.

Exercise 2.3.2: Try to devise a scheme for speeding up the `removeLast` operation so that it always runs in constant time, no matter how long the list. Don't worry about actually coding up a solution, we'll leave that to project 1. Just come up with an idea about how you'd modify the structure of the list (i.e. the instance variables).

We'll describe the solution in Improvement #7.

Improvement #7: Looking Back



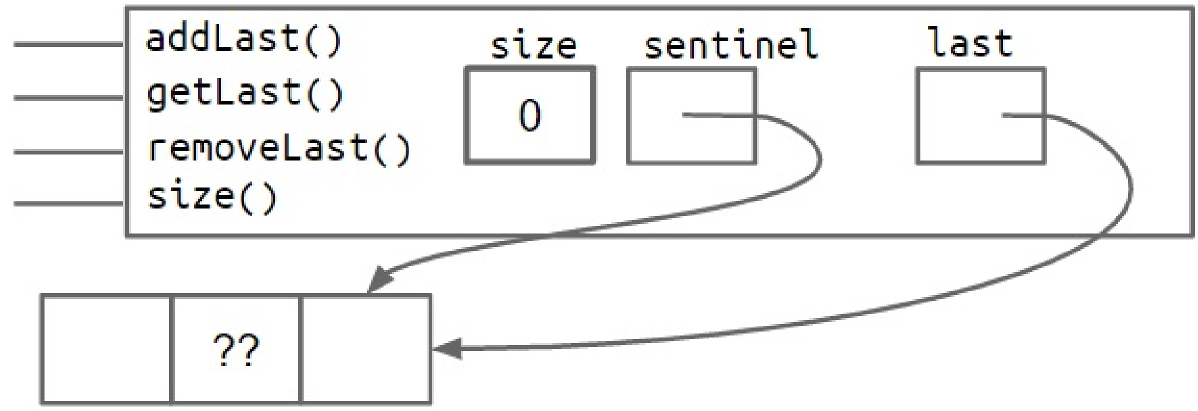
[Video link]()

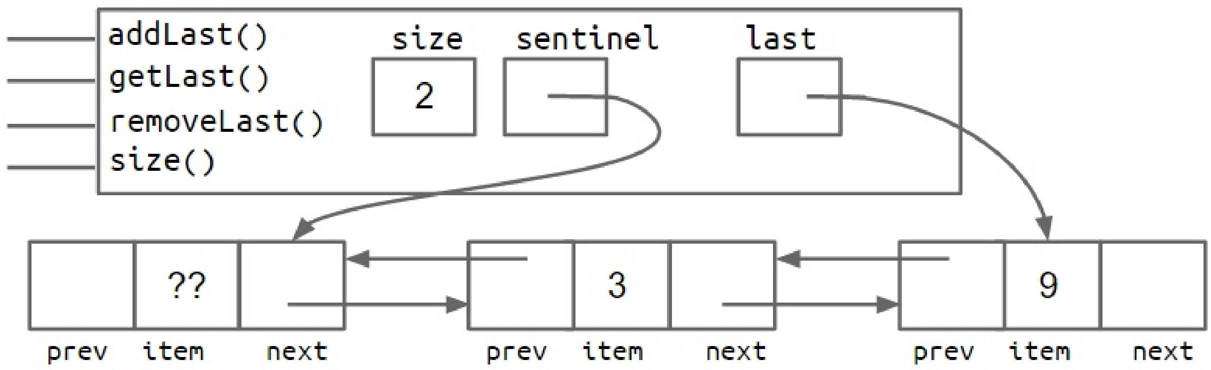
The most natural way to tackle this issue is to add a previous pointer to each `IntNode`, i.e.

```
public class IntNode {
    public IntNode prev;
    public int item;
    public IntNode next;
}
```

In other words, our list now has two links for every node. One common term for such lists is the "Doubly Linked List", which we'll call a `DLList` for short. This is in contrast to a single linked list from chapter 2.2, a.k.a. an `SLList`.

The addition of these extra pointers will lead to extra code complexity. Rather than walk you through it, you'll build a doubly linked list on your own in project 1. The box and pointer diagram below shows more precisely what a doubly linked list looks like for lists of size 0 and size 2, respectively.

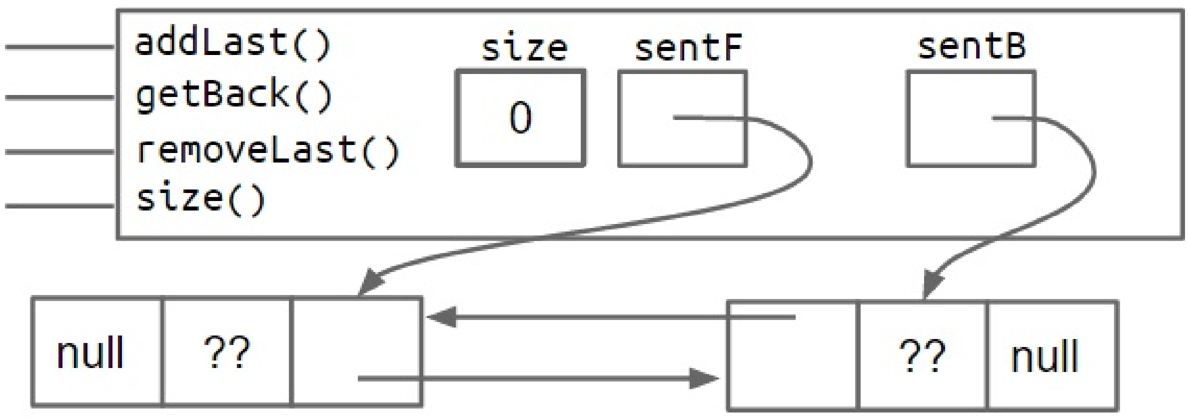


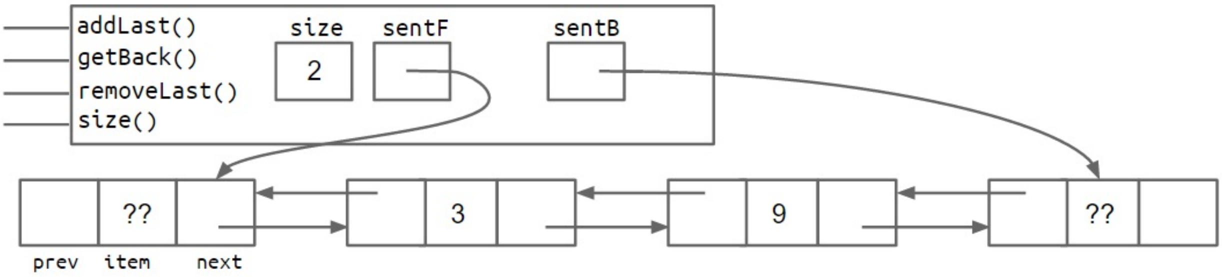


Improvement #8: Sentinel Upgrade

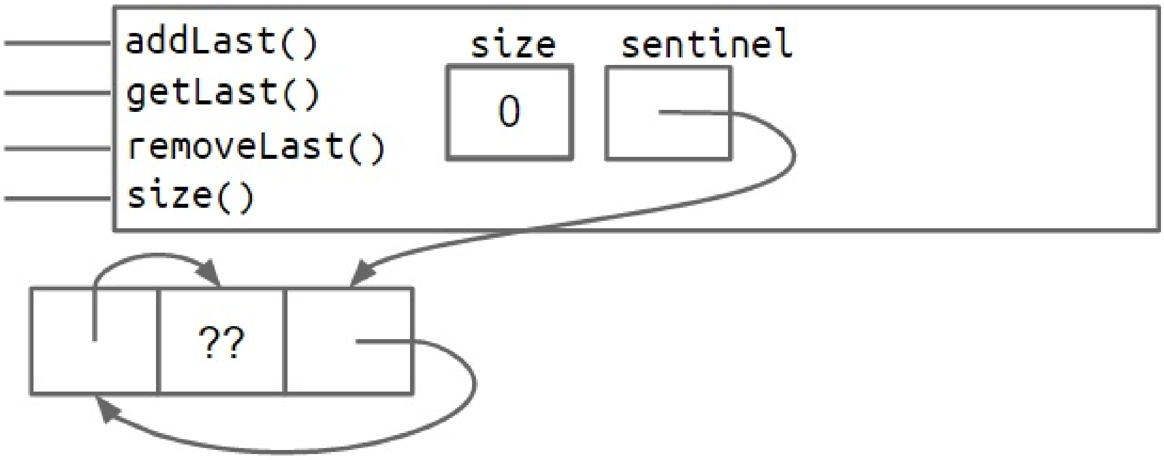
Back pointers allow a list to support adding, getting, and removing the front and back of a list in constant time. There is a subtle issue with this design where the `last` pointer sometimes points at the sentinel node, and sometimes at a real node. Just like the non-sentinel version of the `SLList`, this results in code with special cases that is much uglier than what we'll get after our 8th and final improvement. (Can you think of what `DLList` methods would have these special cases?)

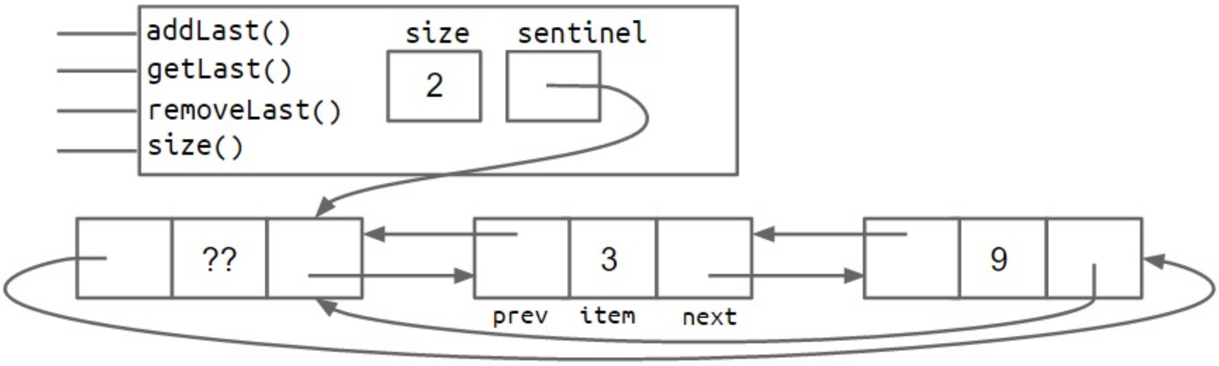
One fix is to add a second sentinel node to the back of the list. This results in the topology shown below as a box and pointer diagram.





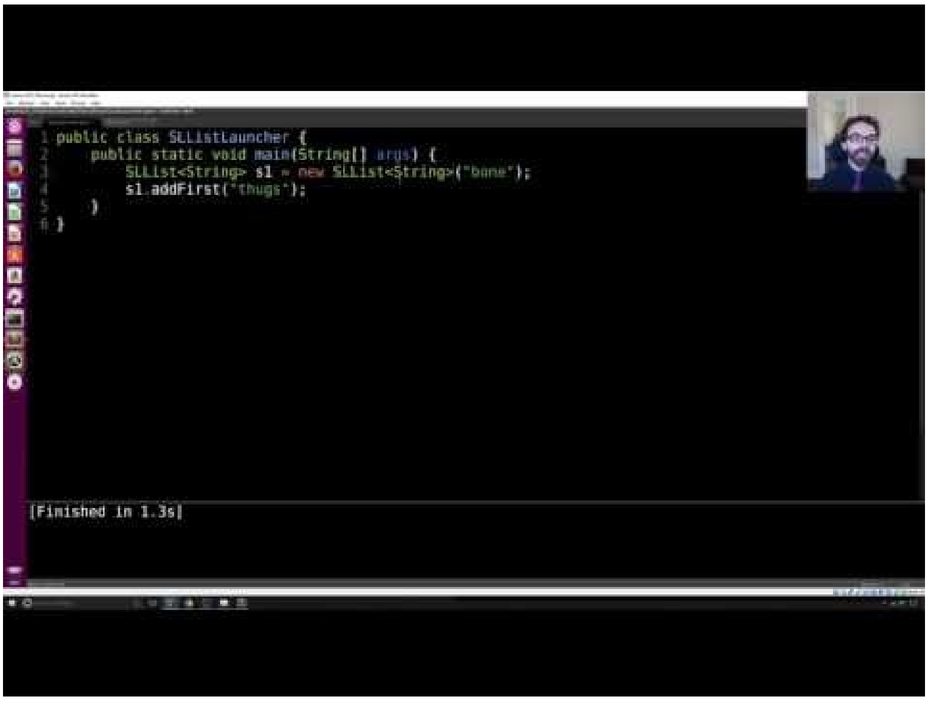
An alternate approach is to implement the list so that it is circular, with the front and back pointers sharing the same sentinel node.





Both the two-sentinel and circular sentinel approaches work and result in code that is free of ugly special cases, though I personally find the circular approach to be cleaner and more aesthetically beautiful. We will not discuss the details of these implementations, as you'll have a chance to explore one or both in project 1.

Generic DLLists



Video link

Our DLLists suffer from a major limitation: they can only hold integer values. For example, suppose we wanted to create a list of Strings:

```
DLList d2 = new DLList("hello");
d2.addLast("world");
```

The code above would crash, since our `DLList` constructor and `addLast` methods only take an integer argument.

Luckily, in 2004, the creators of Java added **generics** to the language, which will allow you to, among other things, create data structures that hold any reference type.

The syntax is a little strange to grasp at first. The basic idea is that right after the name of the class in your class declaration, you use an arbitrary placeholder inside angle brackets: `<>`. Then anywhere you want to use the arbitrary type, you use that placeholder instead.

For example, our `DLList` declaration before was:

```
public class DLList {
    private IntNode sentinel;
    private int size;

    public class IntNode {
        public IntNode prev;
        public int item;
        public IntNode next;
        ...
    }
    ...
}
```

A generic `DLList` that can hold any type would look as below:

```
public class DLList<BleepBlorp> {
    private IntNode sentinel;
    private int size;

    public class IntNode {
        public IntNode prev;
        public BleepBlorp item;
        public IntNode next;
        ...
    }
    ...
}
```

Here, `BleepBlorp` is just a name I made up, and you could use most any other name you might care to use instead, like `GloopGlop` , `Horse` , `TelbudorphMulticulus` or whatever.

Now that we've defined a generic version of the `DLList` class, we must also use a special syntax to instantiate this class. To do so, we put the desired type inside of angle brackets during declaration, and also use empty angle brackets during instantiation. For example:

```
DLList<String> d2 = new DLList<>("hello");
d2.addLast("world");
```

Since generics only work with reference types, we cannot put primitives like `int` or `double` inside of angle brackets, e.g. `<int>` . Instead, we use the reference version of the primitive type, which in the case of `int` case is `Integer` , e.g.

```
DLList<Integer> d1 = new DLList<>(5);
d1.insertFront(10);
```

There are additional nuances about working with generic types, but we will defer them to a later chapter of this book, when you've had more of a chance to experiment with them on your own. For now, use the following rules of thumb:

- In the .java file **implementing** a data structure, specify your generic type name only once at the very top of the file after the class name.
- In other .java files, which use your data structure, specify the specific desired type during declaration, and use the empty diamond operator during instantiation.
- If you need to instantiate a generic over a primitive type, use `Integer` , `Double` , `Character` , `Boolean` , `Long` , `Short` , `Byte` , or `Float` instead of their primitive equivalents.

Minor detail: You may also declare the type inside of angle brackets when instantiating, though this is not necessary, so long as you are also declaring a variable on the same line. In other words, the following line of code is perfectly valid, even though the `Integer` on the right hand side is redundant.

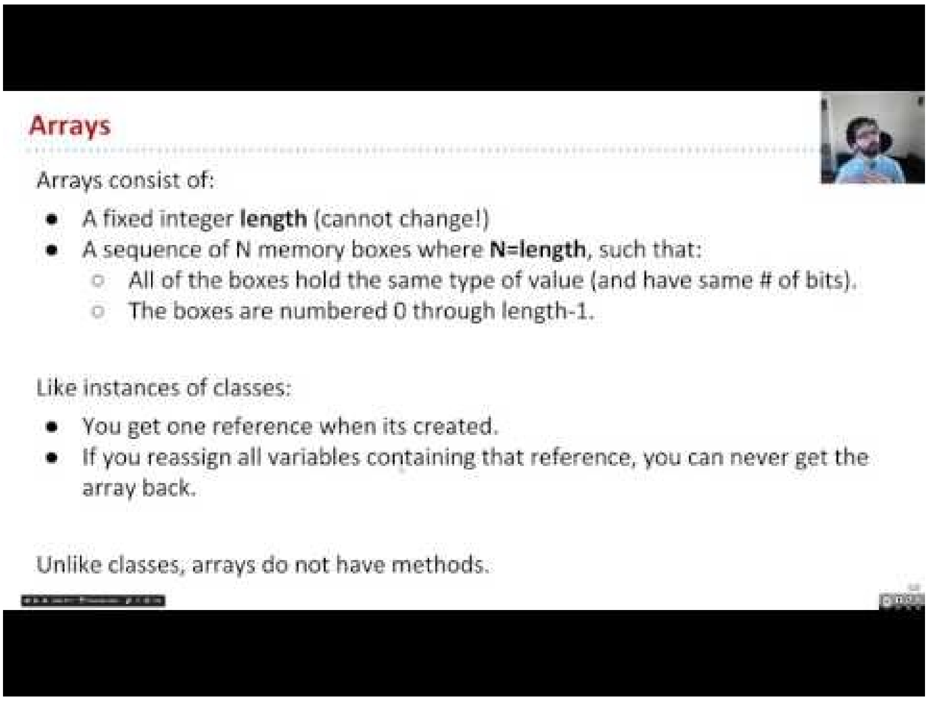
```
DLList<Integer> d1 = new DLList<Integer>(5);
```

At this point, you know everything you need to know to implement the `LinkedListDeque` project on project 1, where you'll refine all of the knowledge you've gained in chapters 2.1, 2.2, and 2.3.

What Next

- The first part of Project 1, where you implement `LinkedListDeque.java` .

Arrays



Video link

So far, we've seen how to harness recursive class definitions to create an expandable list class, including the `IntList`, `SLList`, and `DLList`. In the next two sections of this book, we'll discuss how to build a list class using arrays.

This section of this book assumes you've already worked with arrays and is not intended to be a comprehensive guide to their syntax.

Array Basics

To ultimately build a list that can hold information, we need some way to get memory boxes. Prevously, we saw how we could get memory boxes with variable declarations and class instantiations. For example:

- `int x;` gives us a 32 bit memory box that stores ints.
- `Walrus w1;` gives us a 64 bit memory box that stores Walrus references.
- `Walrus w2 = new Walrus(30, 5.6);` gets us 3 total memory boxes. One 64 bit box that stores Walrus references, one 32 bit box that stores the int size of the Walrus, and a 64 bit box that stores the double tuskSize of the Walrus.

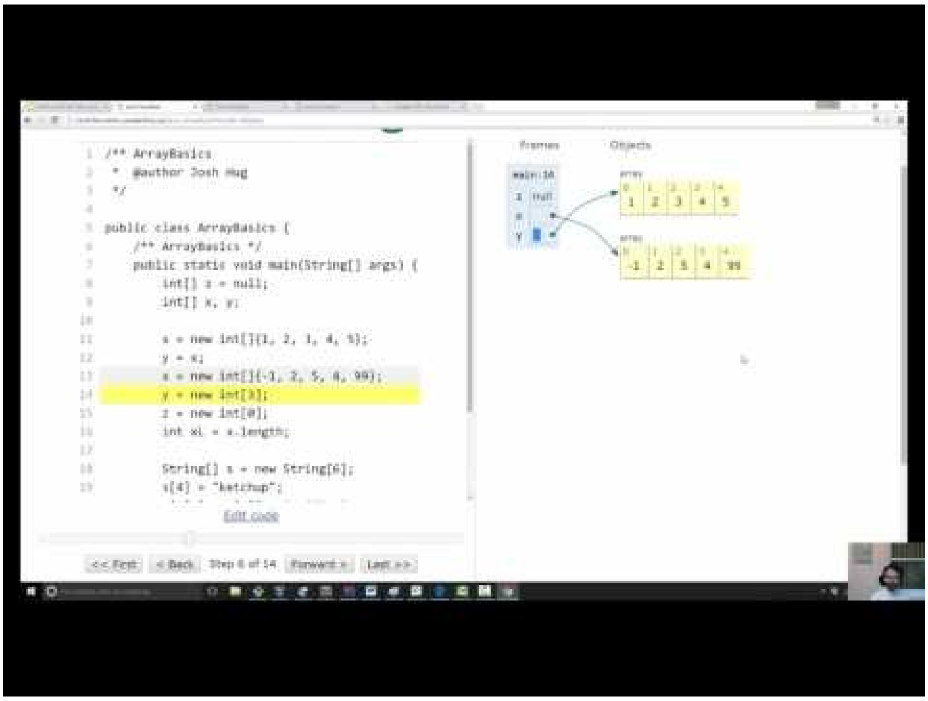
Arrays are a special type of object that consists of a numbered sequence of memory boxes. This is unlike class instances, which have named memory boxes. To get the ith item of an array, we use bracket notation as we saw in HW0 and Project 0, e.g. `A[i]` to get the `i` th element of A.

Arrays consist of:

- A fixed integer length, N
- A sequence of N memory boxes (N = length) where all boxes are of the same type, and are numbered 0 through N - 1.

Unlike classes, arrays do not have methods.

Array Creation



Video link

There are three valid notations for array creation. Try running the code below and see what happens. Click here for an interactive visualization.

- `x = new int[3];`
- `y = new int[]{1, 2, 3, 4, 5};`
- `int[] z = {9, 10, 11, 12, 13};`

All three notations create an array. The first notation, used to create `x`, will create an array of the specified length and fill in each memory box with a default value. In this case, it will create an array of length 3, and fill each of the 3 boxes with the default `int` value `0`.

The second notation, used to create `y`, creates an array with the exact size needed to accommodate the specified starting values. In this case, it creates an array of length 5, with those five specific elements.

The third notation, used to declare **and** create `z`, has the same behavior as the second notation. The only difference is that it omits the usage of `new`, and can only be used when combined with a variable declaration.

None of these notations is better than any other.

Array Access and Modification

The following code showcases all of the key syntax we'll use to work with arrays. Try stepping through the code below and making sure you understand what happens when each line executes. To do so, click here for an interactive visualization. With the exception of the final line of code, we've seen all of this syntax before.

```
int[] z = null;
int[] x, y;

x = new int[]{1, 2, 3, 4, 5};
y = x;
x = new int[]{-1, 2, 5, 4, 99};
y = new int[3];
z = new int[0];
int xL = x.length;

String[] s = new String[6];
s[4] = "ketchup";
s[x[3] - x[1]] = "muffins";

int[] b = {9, 10, 11};
System.arraycopy(b, 0, x, 3, 2);
```

The final line demonstrates one way to copy information from one array to another. `System.arraycopy` takes five parameters:

- The array to use as a source
- Where to start in the source array
- The array to use as a destination
- Where to start in the destination array
- How many items to copy

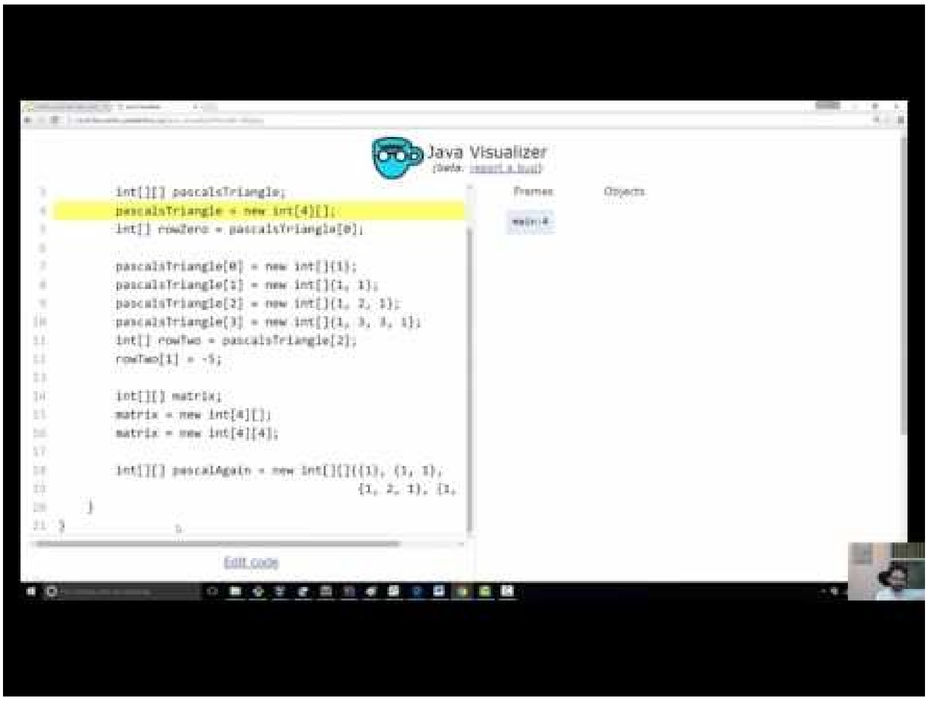
For Python veterans, `System.arraycopy(b, 0,x, 3, 2)` is the equivalent of `x[3:5] = b[0:2]` in Python.

An alternate approach to copying arrays would be to use a loop. `arraycopy` is usually faster than a loop, and results in more compact code. The only downside is that `arraycopy` is (arguably) harder to read. Note that Java arrays only perform bounds checking at runtime. That is, the following code compiles just fine, but will crash at runtime.

```
int[] x = {9, 10, 11, 12, 13};
int[] y = new int[2];
int i = 0;
while (i < x.length) {
    y[i] = x[i];
    i += 1;
}
```

Try running this code locally in a java file or in the visualizer. What is the name of the error that you encounter when it crashes? Does the name of the error make sense?

2D Arrays in Java



Video link

What one might call a 2D array in Java is actually just an array of arrays. They follow the same rules for objects that we've already learned, but let's review them to make sure we understand how they work.

Syntax for arrays of arrays can be a bit confusing. Consider the code `int[][] bamboozle = new int[4][]`. This creates an array of integer arrays called `bamboozle`. Specifically, this creates exactly four memory boxes, each of which can point to an array of integers (of unspecified length).

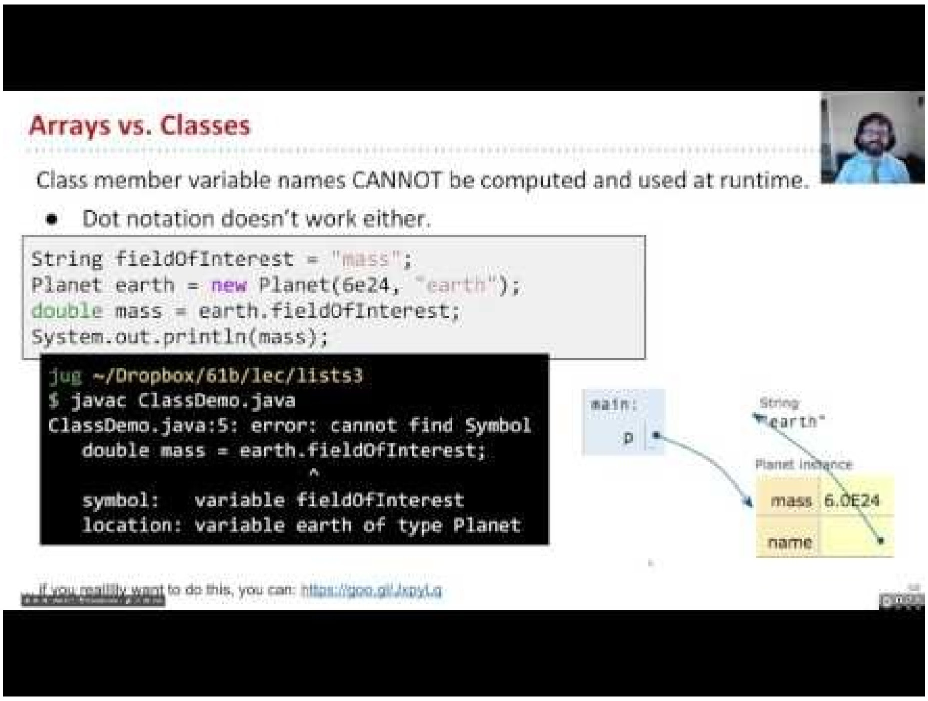
Try running the code below line-by-lines, and see if the results match your intuition. For an interactive visualization, click here.

```
int[][] pascalsTriangle;
pascalsTriangle = new int[4][];
int[] rowZero = pascalsTriangle[0];

pascalsTriangle[0] = new int[]{1};
pascalsTriangle[1] = new int[]{1, 1};
pascalsTriangle[2] = new int[]{1, 2, 1};
pascalsTriangle[3] = new int[]{1, 3, 3, 1};
int[] rowTwo = pascalsTriangle[2];
rowTwo[1] = -5;

int[][] matrix;
matrix = new int[4][];
matrix = new int[4][4];

int[][] pascalAgain = new int[][]{{1}, {1, 1},
                                  {1, 2, 1}, {1, 3, 3, 1}};
```

Exercise 2.4.1: After running the code below, what will be the values of x[0][0] and w[0][0]? Check your work by clicking here.

```
int[][] x = {{1, 2, 3}, {4, 5, 6}, {7, 8, 9}};

int[][] z = new int[3][];
z[0] = x[0];
z[1] = x[1];
z[2] = x[2];
z[0][0] = -z[0][0];

int[][] w = new int[3][3];
System.arraycopy(x[0], 0, w[0], 0, 3);
System.arraycopy(x[1], 0, w[1], 0, 3);
System.arraycopy(x[2], 0, w[2], 0, 3);
w[0][0] = -w[0][0];
```

Arrays vs. Classes



[Video link](#)

Both arrays and classes can be used to organize a bunch of memory boxes. In both cases, the number of memory boxes is fixed, i.e. the length of an array cannot be changed, just as class fields cannot be added or removed.

The key differences between memory boxes in arrays and classes:

- Array boxes are numbered and accessed using `[]` notation, and class boxes are named and accessed using dot notation.
- Array boxes must all be the same type. Class boxes can be different types.

One particularly notable impact of these difference is that `[]` notation allows us to specify which index we'd like at runtime. For example, consider the code below:

```
int indexOfInterest = askUserForInteger();
int[] x = {100, 101, 102, 103};
int k = x[indexOfInterest];
System.out.println(k);
```

If we run this code, we might get something like:

```
$ javac arrayDemo
$ java arrayDemo
What index do you want? 2
102
```

By contrast, specifying fields in a class is not something we do at runtime. For example, consider the code below:

```
String fieldOfInterest = "mass";
Planet p = new Planet(6e24, "earth");
double mass = p[fieldOfInterest];
```

If we tried compiling this, we'd get a syntax error.

```
$ javac classDemo
FieldDemo.java:5: error: array required, but Planet found
        double mass = earth[fieldOfInterest];
                               ^
```

The same problem occurs if we try to use dot notation:

```
String fieldOfInterest = "mass";
Planet p = new Planet(6e24, "earth");
double mass = p.fieldOfInterest;
```

Compiling, we'd get:

```
$ javac classDemo
FieldDemo.java:5: error: cannot find symbol
        double mass = earth.fieldOfInterest;
                           ^
  symbol:   variable fieldOfInterest
  location: variable earth of type Planet
```

This isn't a limitation you'll face often, but it's worth pointing out, just for the sake of good scholarship. For what it's worth, there is a way to specify desired fields at runtime called *reflection*, but it is considered very bad coding style for typical programs. You can read more about reflection here. **You should never use reflection in any 61B program**, and we won't discuss it in our course.

In general, programming languages are partially designed to limit the choices of programmers to make code simpler to reason about. By restricting these sorts of features to the special Reflections API, we make typical Java programs easier to read and interpret.

Appendix: Java Arrays vs. Other Languages

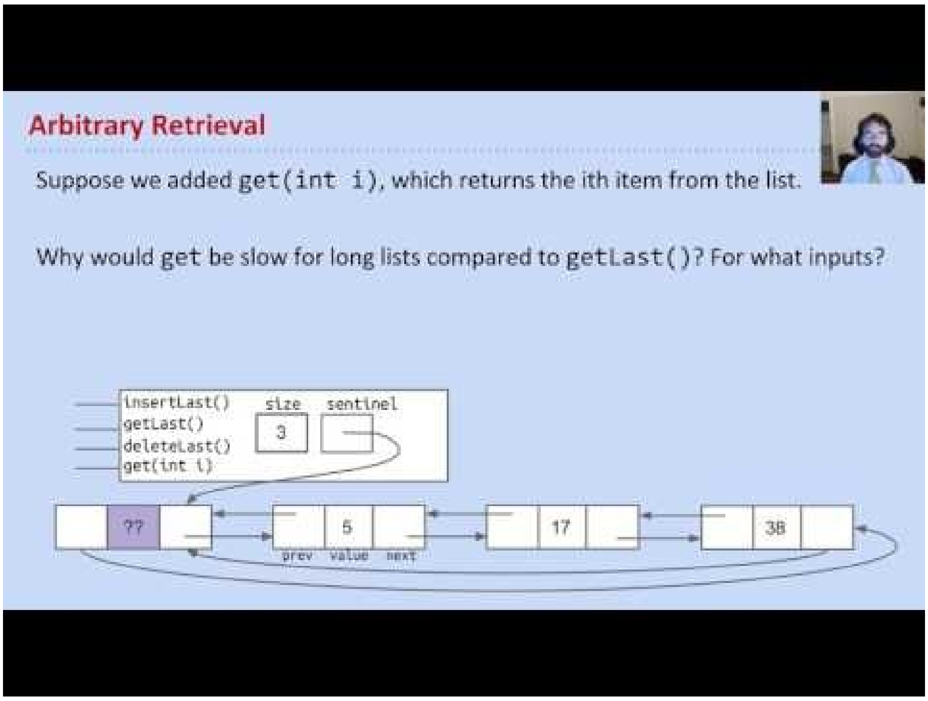
Compared to arrays in other languages, Java arrays:

- Have no special syntax for "slicing" (such as in Python).

- Cannot be shrunk or expanded (such as in Ruby).
- Do not have member methods (such as in Javascript).
- Must contain values only of the same type (unlike Python).

In this section, we'll build a new class called `AList` that can be used to store arbitrarily long lists of data, similar to our `DLList`. Unlike the `DLList`, the `AList` will use arrays to store data instead of a linked list.

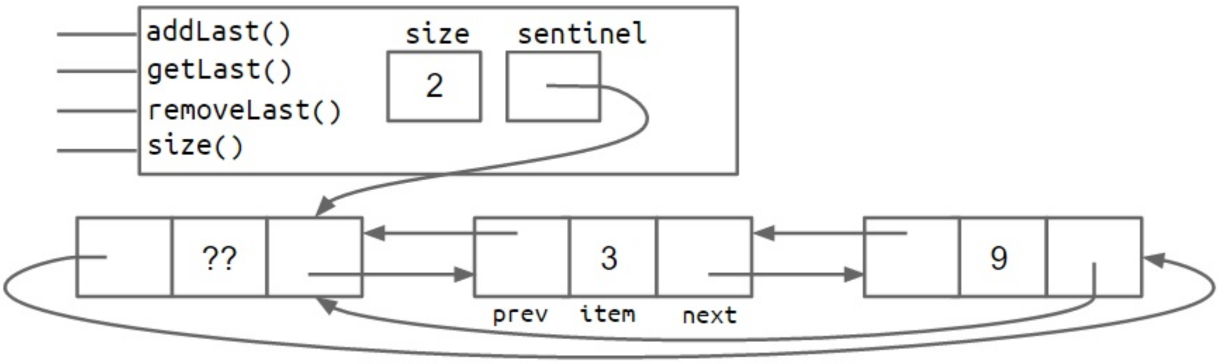
Linked List Performance Puzzle



[Video link]

Suppose we wanted to write a new method for `DLList` called `int get(int i)`. Why would `get` be slow for long lists compared to `getLast`? For what inputs would it be especially slow?

You may find the figure below useful for thinking about your answer.



Linked List Performance Puzzle Solution

It turns out that no matter how clever you are, the `get` method will usually be slower than `getBack` if we're using the doubly linked list structure described in section 2.3.

This is because, since we only have references to the first and last items of the list, we'll always need to walk through the list from the front or back to get to the item that we're trying to retrieve. For example, if we want to get item #417 in a list of length 10,000, we'll have to walk across 417 forward links to get to the item we want.

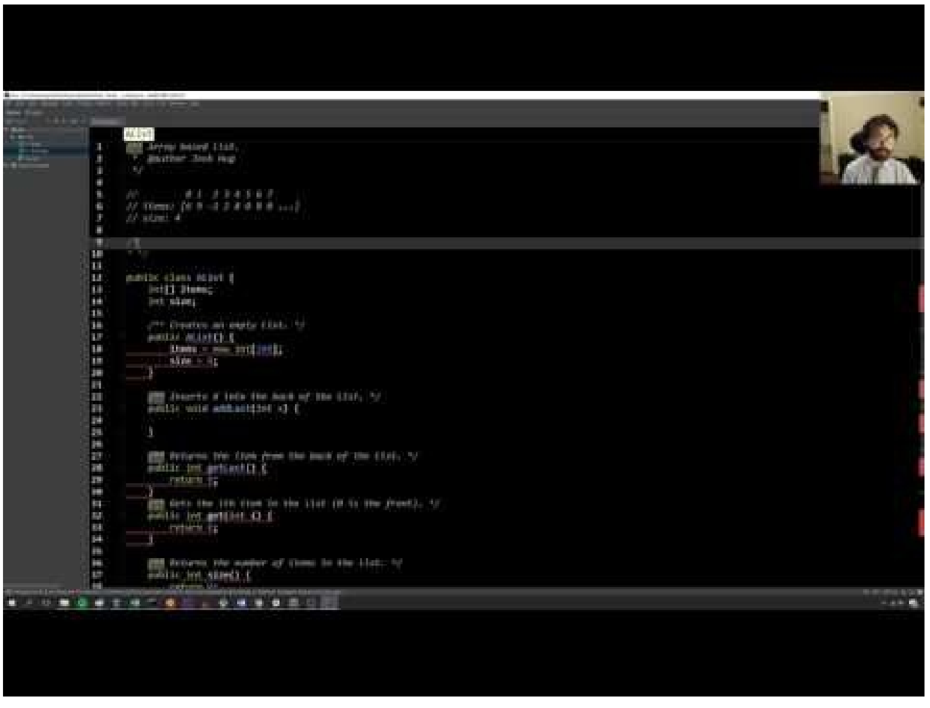
In the very worst case, the item is in the very middle and we'll need to walk through a number of items proportional to the length of the list (specifically, the number of items divided by two). In other words, our worst case execution time for `get` is linear in the size of the entire list. This in contrast to the runtime for `getBack`, which is constant, no matter the size of the list. Later in the course, we'll formally define runtimes in terms of big O and big Theta notation. For now, we'll stick to an informal understanding.

Our First Attempt: The Naive Array Based List

Accessing the `i` th element of an array takes constant time on a modern computer. This suggests that an array-based list would be capable of much better performance for `get` than a linked-list based solution, since it can simply use bracket notation to get the item of interest.

If you'd like to know **why** arrays have constant time access, check out this Quora post.

Optional Exercise 2.5.1: Try to build an AList class that supports `addLast`, `getLast`, `get`, and `size` operations. Your AList should work for any size array up to 100. For starter code, see https://github.com/Berkeley-CS61B/lectureCode/tree/master/lists4/DIY.



Video link

My solution has the following handy invariants.

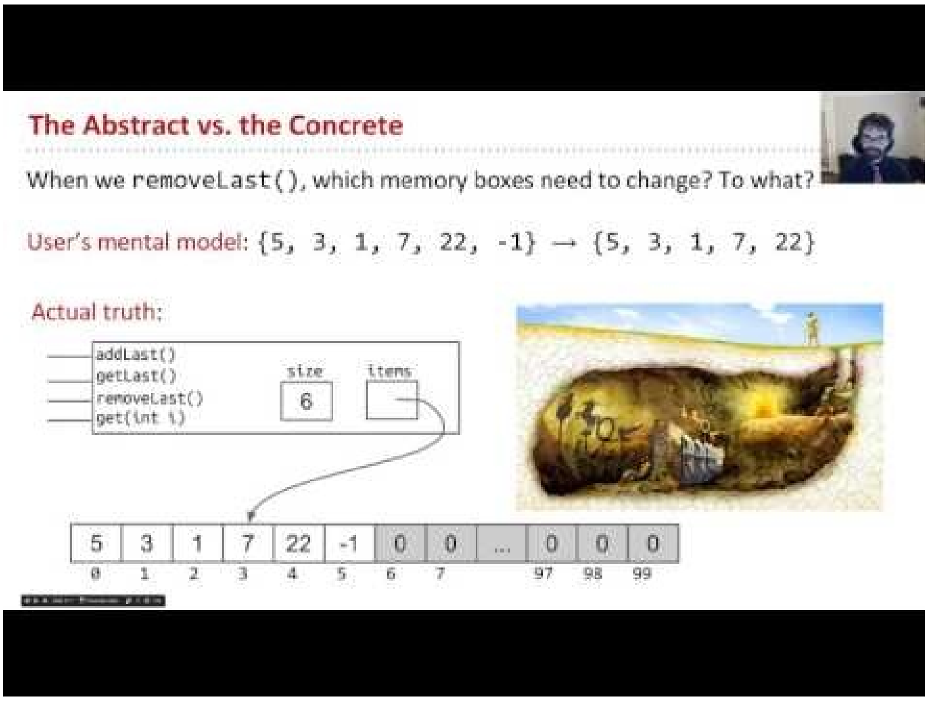
- The position of the next item to be inserted (using `addLast`) is always `size` .
- The number of items in the AList is always `size` .
- The position of the last item in the list is always `size - 1` .

Other solutions might be slightly different.

removeLast

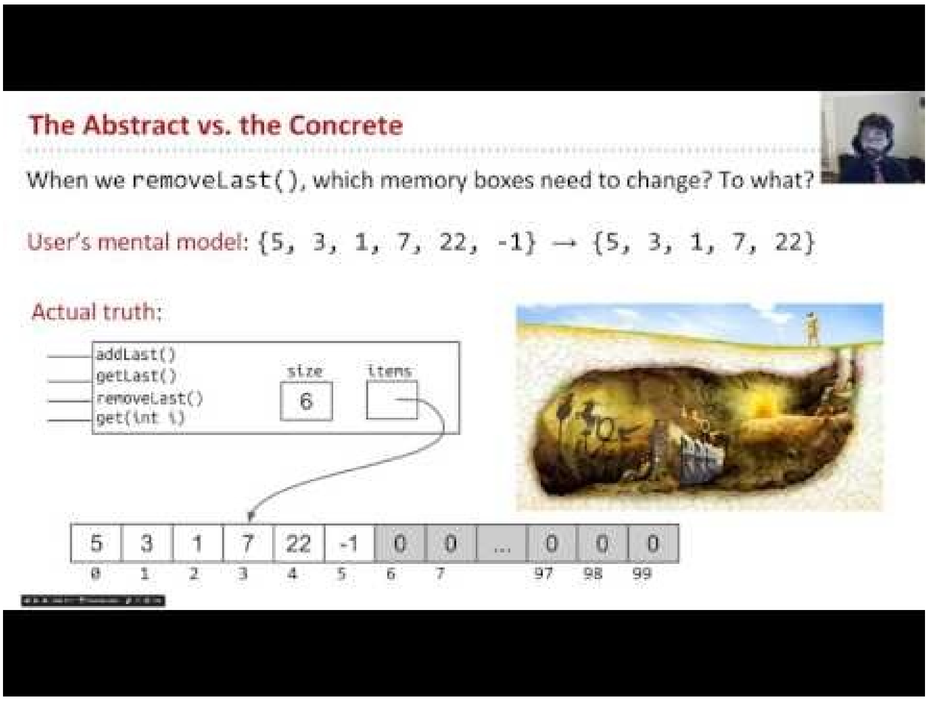
The last operation we need to support is `removeLast` . Before we start, we make the following key observation: Any change to our list must be reflected in a change in one or more memory boxes in our implementation.

This might seem obvious, but there is some profundity to it. The list is an abstract idea, and the `size` , `items` , and `items[i]` memory boxes are the concrete representation of that idea. Any change the user tries to make to the list using the abstractions we provide (`addLast` , `removeLast`) must be reflected in some changes to these memory boxes in a way that matches the user's expectations. Our invariants provide us with a guide for what those changes should look like.



[Video link]

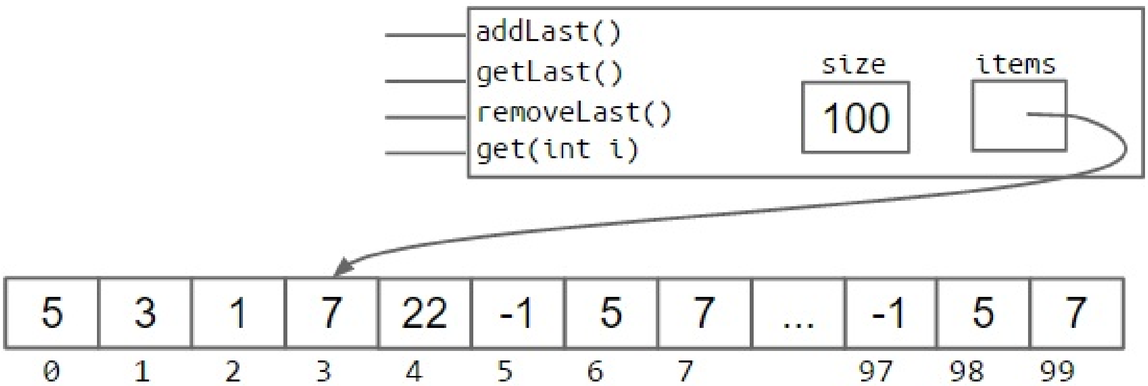
Optional Exercise 2.5.2: Try to write `removeLast` . Before starting, decide which of `size` , `items` , and `items[i]` needs to change so that our invariants are preserved after the operation, i.e. so that future calls to our methods provide the user of the list class with the behavior they expect.



Video link

Naive Resizing Arrays

Optional Exercise 2.5.3: Suppose we have an AList in the state shown in the figure below. What will happen if we call `addLast(11)`? What should we do about this problem?



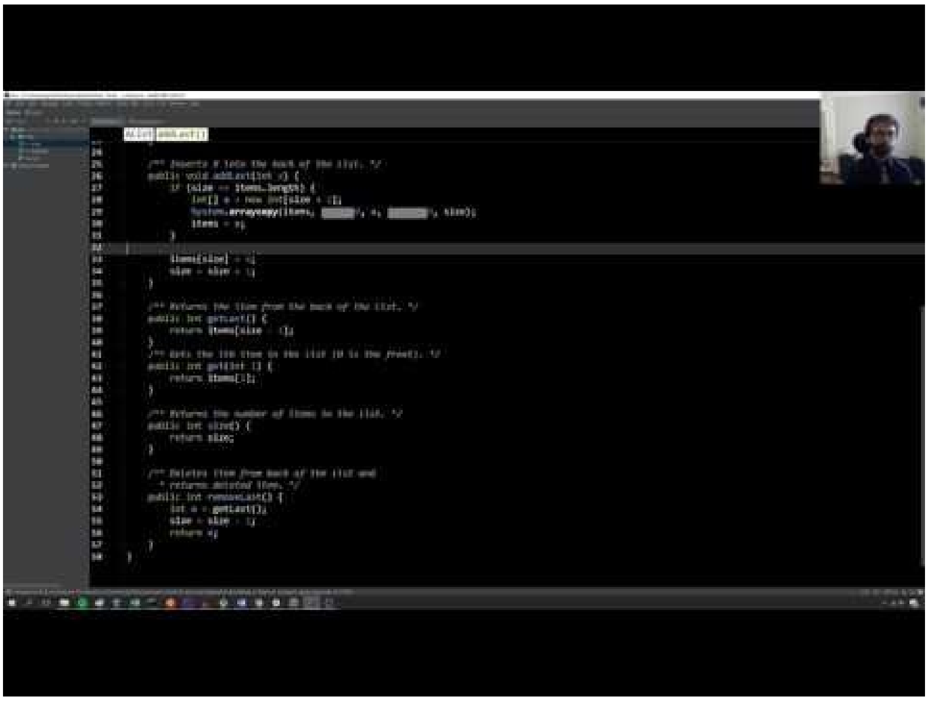
Video link

The answer, in Java, is that we simply build a new array that is big enough to accomodate the new data. For example, we can imagine adding the new item as follows:

```
int[] a = new int[size + 1];
System.arraycopy(items, 0, a, 0, size);
a[size] = 11;
items = a;
size = size + 1;
```

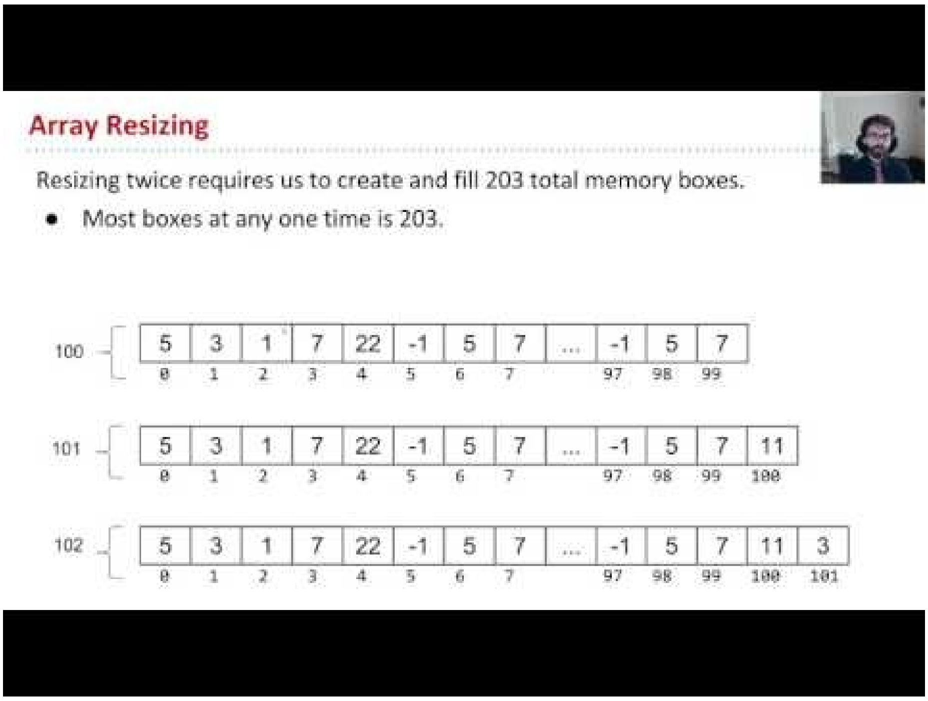
The process of creating a new array and copying items over is often referred to as "resizing". It's a bit of a misnomer since the array doesn't actually change size, we are just making a **new** one that has a bigger size.

Exercise 2.5.4: Try to implement the `addLast(int i)` method to work with resizing arrays.



Video link

Analyzing the Naive Resizing Array

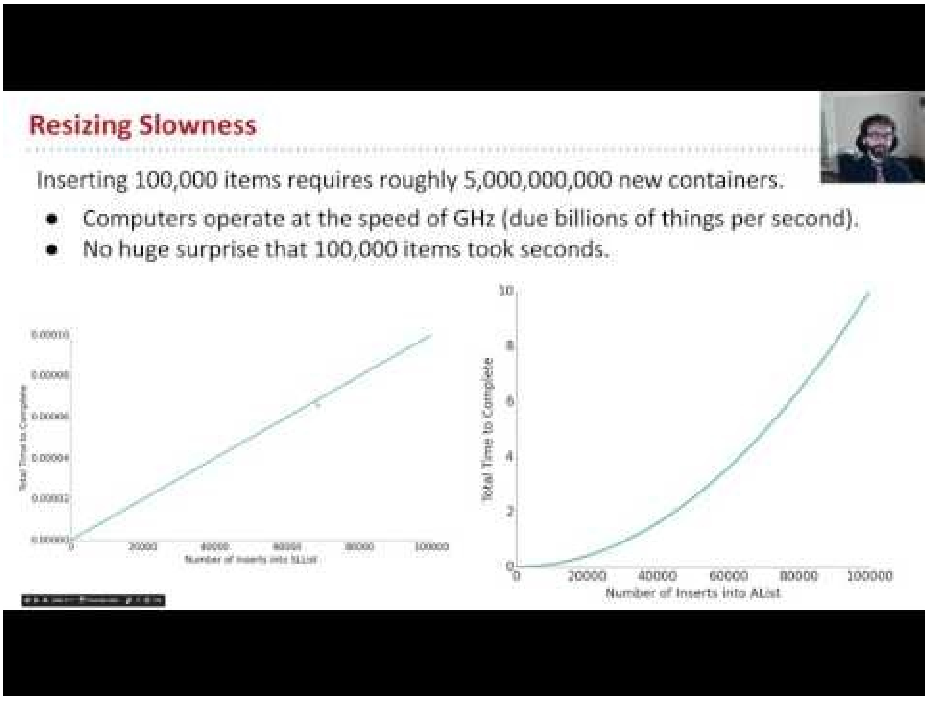


Video link

The approach that we attempted in the previous section has terrible performance. By running a simple computational experiment where we call `addLast` 100,000 times, we see that the `SLList` completes so fast that we can't even time it. By contrast our array based list takes several seconds.

To understand why, consider the following exercise:

Exercise 2.5.5: Suppose we have an array of size 100. If we call insertBack two times, how many total boxes will we need to create and fill throughout this entire process? How many total boxes will we have at any one time, assuming that garbage collection happens as soon as the last reference to an array is lost?

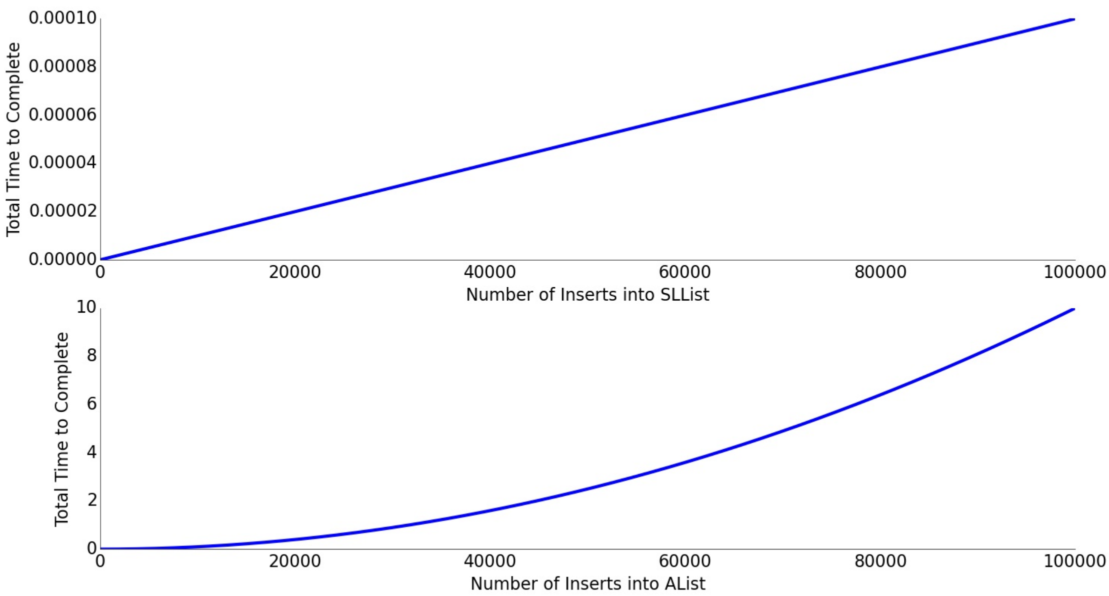


[Video link](#)

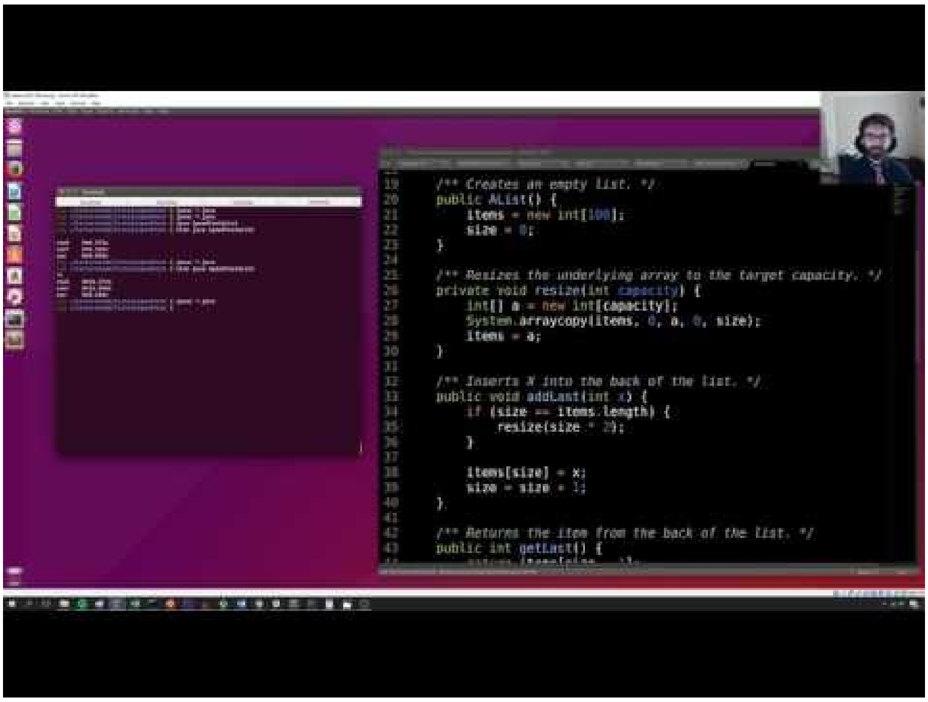
Exercise 2.5.6: Starting from an array of size 100, approximately how many memory boxes get created and filled if we call `addLast` 1,000 times?

Creating all those memory boxes and recopying their contents takes time. In the graph below, we plot total time vs. number of operations for an SLList on the top, and for a naive array based list on the bottom. The SLList shows a straight line, which means for each `add` operation, the list takes the same additional amount of time. This means each single operation takes constant time! You can also think of it this way: the graph is linear, indicating that each operation takes constant time, since the integral of a constant is a line.

By contrast, the naive array list shows a parabola, indicating that each operation takes linear time, since the integral of a line is a parabola. This has significant real world implications. For inserting 100,000 items, we can roughly compute how much longer by computing the ratio of N^2/N. Inserting 100,000 items into our array based list takes (100,000^2)/100,000 or 100,000 times as long. This is obviously unacceptable.



Geometric Resizing



[Video link]

We can fix our performance problems by growing the size of our array by a multiplicative amount, rather than an additive amount. That is, rather than **adding** a number of memory boxes equal to some resizing factor `RFACTOR`:

```
public void insertBack(int x) {
    if (size == items.length) {
           resize(size + RFACTOR);
    }
    items[size] = x;
    size += 1;
}
```

We instead resize by **multiplying** the number of boxes by `RFACTOR`.

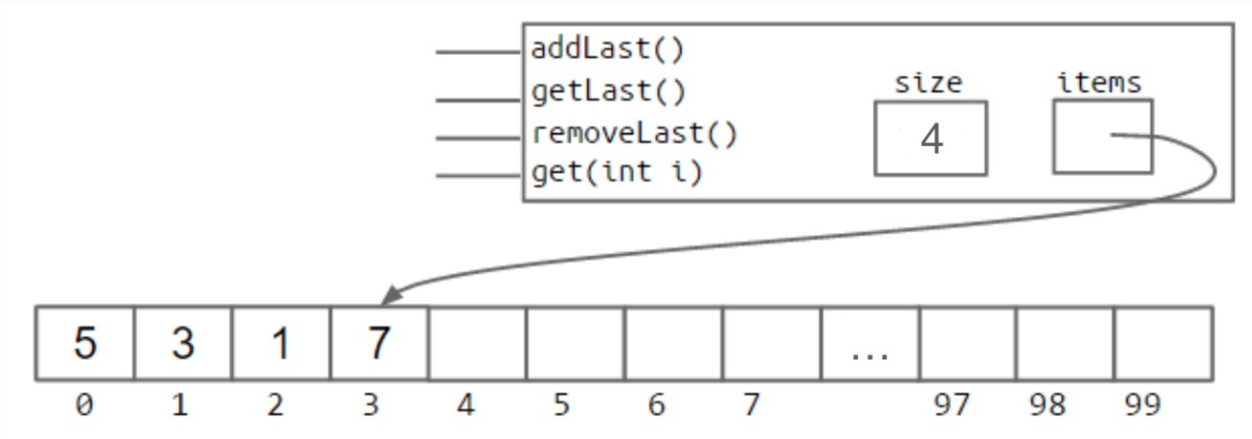
```
public void insertBack(int x) {
    if (size == items.length) {
           resize(size * RFACTOR);
    }
    items[size] = x;
    size += 1;
}
```

Repeating our computational experiment from before, we see that our new `AList` completes 100,000 inserts in so little time that we don't even notice. We'll defer a full analysis of why this happens until the final chapter of this book.

Memory Performance

Our `AList` is almost done, but we have one major issue. Suppose we insert 1,000,000,000 items, then later remove 990,000,000 items. In this case, we'll be using only 10,000,000 of our memory boxes, leaving 99% completely unused.

To fix this issue, we can also downsize our array when it starts looking empty. Specifically, we define a "usage ratio" R which is equal to the size of the list divided by the length of the `items` array. For example, in the figure below, the usage ratio is 0.04.



In a typical implementation, we halve the size of the array when R falls to less than 0.25.

Generic ALists

Video link

Just as we did before, we can modify our `AList` so that it can hold any data type, not just integers. To do this, we again use the special angle braces notation in our class and substitute our arbitrary type parameter for integer wherever appropriate. For example, below, we use `Glorp` as our type parameter.

There is one significant syntactical difference: Java does not allow us to create an array of generic objects due to an obscure issue with the way generics are implemented. That is, we cannot do something like:

```
Glorp[] items = new Glorp[8];
```

Instead, we have to use the awkward syntax shown below:

```
Glorp[] items = (Glorp []) new Object[8];
```

This will yield a compilation warning, but it's just something we'll have to live with. We'll discuss this in more details in a later chapter.

The other change we make is that we null out any items that we "delete". Whereas before, we had no reason to zero out elements that were deleted, with generic objects, we do want to null out references to the objects that we're storing. This is to avoid "loitering". Recall that Java only destroys objects when the last reference has been lost. If we fail to null out the reference, then Java will not garbage collect the objects that have been added to the list.

This is a subtle performance bug that you're unlikely to observe unless you're looking for it, but in certain cases could result in a significant wastage of memory.

What's next:

- Discussion 3
- Lab 2
- Project 1A

Testing and Selection Sort

One of the most important skills an intermediate to advanced programmer can have is the ability to tell when your code is correct. In this chapter, we'll discuss how you can write tests to evaluate code correctness. Along the way, we'll also discuss an algorithm for sorting called Selection Sort.

A New Way

Video link

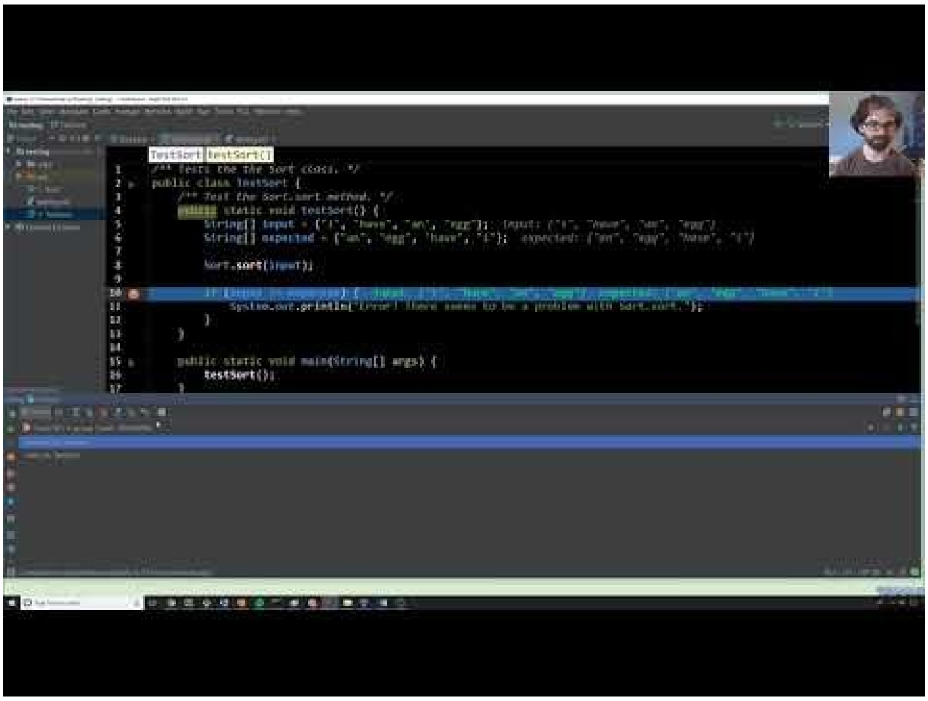
When you write a program, it may have errors. In a classroom setting, you gain confidence in your code's correctness through some combination of user interaction, code analysis, and autograder testing, with this last item being of the greatest importance in many cases, particularly as it is how you earn points.

Autograders, of course, are not magic. They are code that the instructors write that is fundamentally not all that different from the code that you are writing. In the real world, these tests are written by the programmers themselves, rather than some benevolent Josh-Hug-like third party.

In this chapter, we'll explore how we can write our own tests. Our goal will be to create a class called `Sort` that provides a method `sort(String[] x)` that destructively sorts the strings in the array `x`.

As a totally new way of thinking, we'll start by writing `testSort()` first, and only after we've finished the test, we'll move on to writing the actual sorting code.

Ad Hoc Testing



[Video link]

Writing a test for `Sort.sort` is relatively straightforward, albeit tedious. We simply need to create an input, call `sort`, and check that the output of the method is correct. If the output is not correct, we print out the first mismatch and terminate the test. For example, we might create a test class as follows:

```java
public class TestSort {
    /** Tests the sort method of the Sort class. */
    public static void testSort() {
        String[] input = {"i", "have", "an", "egg"};
        String[] expected = {"an", "egg", "have", "i"};
        Sort.sort(input);
        for (int i = 0; i < input.length; i += 1) {
            if (!input[i].equals(expected[i])) {
                System.out.println("Mismatch in position " + i + ", expected: " + expe
cted + ", but got: " + input[i] + ".");
                break;
            }
        }
    }

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

We can test out our test by creating a blank `Sort.sort` method as shown below:

```java
public class Sort {
    /** Sorts strings destructively. */
    public static void sort(String[] x) {
    }
}
```

If we run the `testSort()` method with this blank `Sort.sort` method, we'd get:

```
Mismatch in position 0, expected: an, but got: i.
```

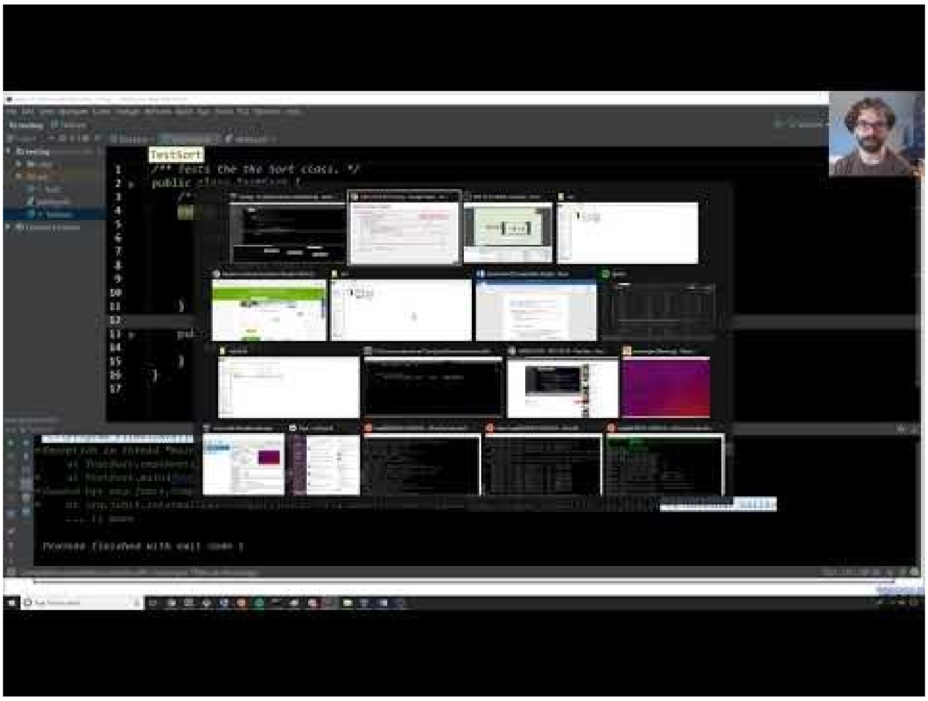
The fact that we're getting an error message is a good thing! This means our test is working. What's very interesting about this is that we've now created a little game for ourselves to play, where the goal is to modify the code for `Sort.sort` so that this error message no longer occurs. It's a bit of a psychological trick, but many programmers find the creation of these little mini-puzzles for themselves to be almost addictive.

In fact, this is a lot like the situation where you have an autograder for a class, and you find yourself hooked on the idea of getting the autograder to give you its love and approval. You now have the ability to create a judge for your code, whose esteem you can only win by completing the code correctly.

Important note: You may be asking "Why are you looping through the entire array? Why don't you just check if the arrays are equal using `==` ? ". The reason is, when we test for equality of two objects, we cannot simply use the `==` operator. The `==` operator compares the literal bits in the memory boxes, e.g. `input == expected` would test whether or not the addresses of `input` and `expected` are the same, not whether the values in the arrays are the same. Instead, we used a loop in `testSort` , and print out the first mismatch. You could also use the built-in method `java.util.Arrays.equals` instead of a loop.

While the single test above wasn't a ton of work, writing a suite of such *ad hoc* tests would be very tedious, as it would entail writing a bunch of different loops and print statements. In the next section, we'll see how the `org.junit` library saves us a lot of work.

JUnit Testing



Video link

The `org.junit` library provides a number of helpful methods and useful capabilities for simplifying the writing of tests. For example, we can replace our simple *ad hoc* test from above with:

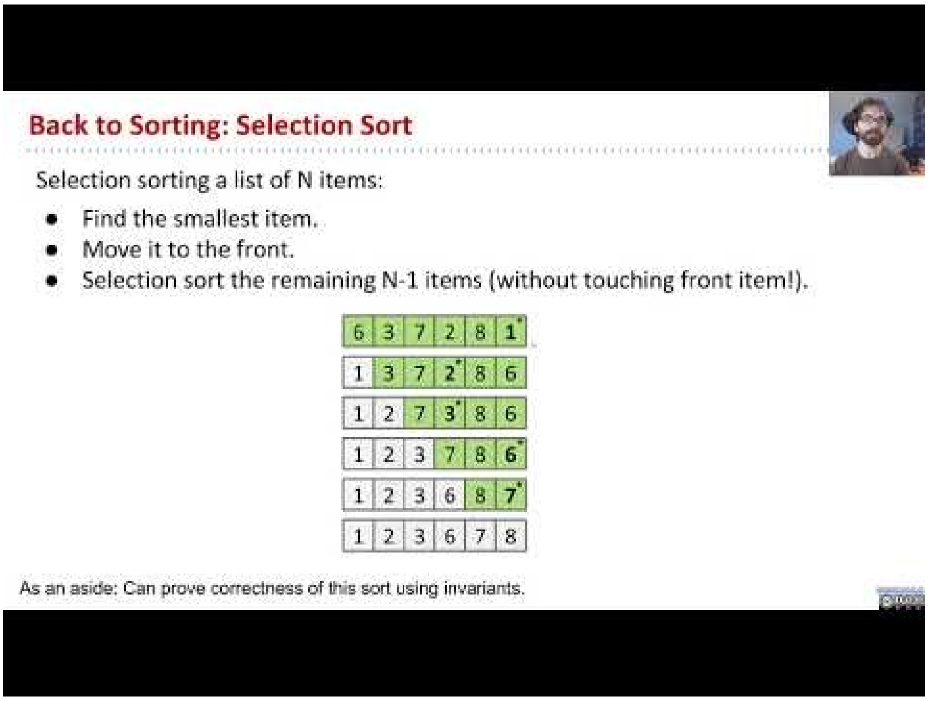
```
public static void testSort() {
    String[] input = {"i", "have", "an", "egg"};
    String[] expected = {"an", "egg", "have", "i"};
    Sort.sort(input);
    org.junit.Assert.assertArrayEquals(expected, input);
}
```

This code is much simpler, and does more or less the exact same thing, i.e. if the arrays are not equal, it will tell us the first mismatch. For example, if we run `testSort()` on a `Sort.sort` method that does nothing, we'd get:

```
Exception in thread "main" arrays first differed at element [0]; expected:<[an]> but w
as:<[i]>
    at org.junit.internal.ComparisonCriteria.arrayEquals(ComparisonCriteria.java:55)
    at org.junit.Assert.internalArrayEquals(Assert.java:532)
    ...
```

While this output is a little uglier than our *ad hoc* test, we'll see at the very end of this chapter how to make it nicer.

Selection Sort



[Video link]

Before we can write a `Sort.sort` method, we need some algorithm for sorting. Perhaps the simplest sorting algorithm around is "selection sort." Selection sort consists of three steps:

- Find the smallest item.
- Move it to the front.
- Selection sort the remaining N-1 items (without touching the front item).

For example, suppose we have the array `{6, 3, 7, 2, 8, 1}`. The smallest item in this array is `1`, so we'd move the `1` to the front. There are two natural ways to do this: One is to stick the `1` at the front and slide all the numbers over, i.e. `{1, 6, 3, 7, 2, 8}`. However, the much more efficient way is to simply swap the `1` with the old front (in this case `6`), yielding `{1, 3, 7, 2, 8, 6}`.

We'd simply repeat the same process for the remaining digits, i.e. the smallest item in `...3, 7, 2, 8, 6}` is `2`. Swapping to the front, we get `{1, 2, 7, 3, 8, 6}`. Repeating until we've got a sorted array, we'd get `{1, 2, 3, 7, 8, 6}`, then `{1, 2, 3, 6, 8, 7}`, then finally `{1, 2, 3, 6, 7, 8}`.

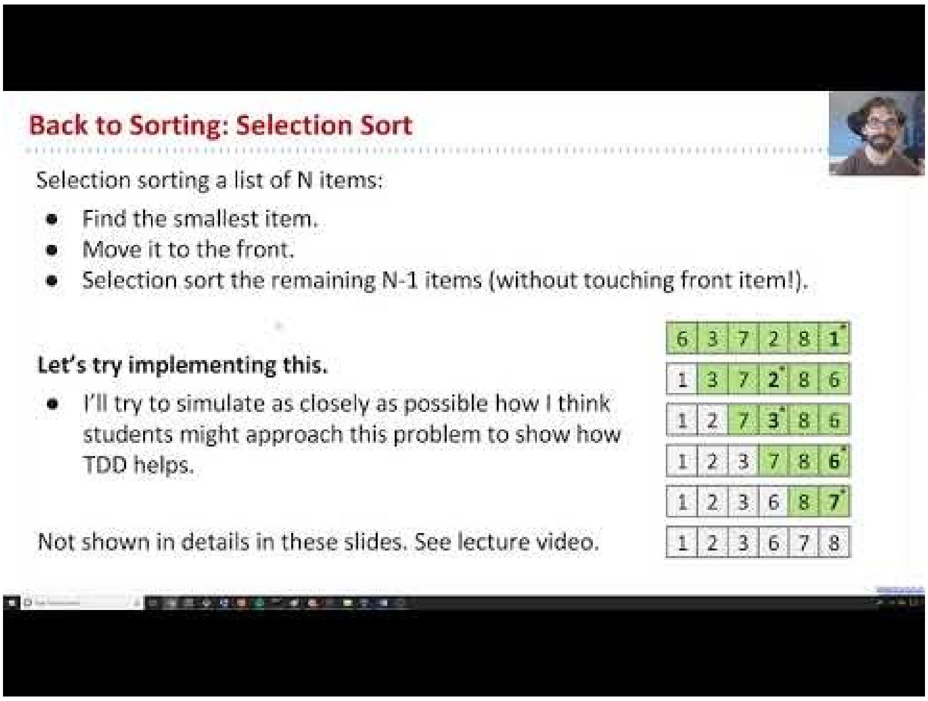
We could mathematically prove the correctness of this sorting algorithm on any arrays by using the concept of invariants that was originally introduced in chapter 2.4, though we will not do so in this textbook. Before proceeding, try writing out your own short array of numbers and perform selection sort on it, so that you can make sure you get the idea.

Now that we know how selection sort works, we can write in a few short comments in our blank `Sort.sort` method to guide our thinking:

```
public class Sort {
    /** Sorts strings destructively. */
    public static void sort(String[] x) {
           // find the smallest item
           // move it to the front
           // selection sort the rest (using recursion?)
    }
}
```

In the following sections, I will attempt to complete an implementation of selection sort. I'll do so in a way that resembles how a student might approach the problem, so **I'll be making a few intentional errors along the way**. These intentional errors are a good thing, as they'll help demonstrate the usefulness of testing. If you spot any of the errors while reading, don't worry, we'll eventually come around and correct them.

findSmallest



[Video link]

The most natural place to start is to write a method for finding the smallest item in a list. As with `Sort.sort`, we'll start by writing a test before we even complete the method. First, we'll create a dummy `findSmallest` method that simply returns some arbitrary value:

```
public class Sort {
    /** Sorts strings destructively. */
    public static void sort(String[] x) {
           // find the smallest item
           // move it to the front
           // selection sort the rest (using recursion?)
    }

    /** Returns the smallest string in x. */
    public static String findSmallest(String[] x) {
        return x[2];
    }
}
```

Obviously this is not a correct implementation, but we've chosen to defer actually thinking about how `findSmallest` works until after we've written a test. Using the `org.junit` library, adding such a test to our `TestSort` class is very easy, as shown below:

```
public class TestSort {
    ...
    public static void testFindSmallest() {
        String[] input = {"i", "have", "an", "egg"};
        String expected = "an";

        String actual = Sort.findSmallest(input);
        org.junit.Assert.assertEquals(expected, actual);
    }

    public static void main(String[] args) {
        testFindSmallest(); // note: we changed this from testSort!
    }
}
```

As with `TestSort.testsort`, we then run our `TestSort.testFindSmallest` method to make sure that it fails. When we run this test, we'll see that it actually passes, i.e. no message appears. This is because we just happened to hard code the correct return value `x[2]`. Let's modify our `findSmallest` method so that it returns something that is definitely incorrect:

```
/** Returns the smallest string in x. */
public static String findSmallest(String[] x) {
    return x[3];
}
```

After making this change, when we run `TestSort.testFindSmallest`, we'll get an error, which is a good thing:

```
Exception in thread "main" java.lang.AssertionError: expected:<[an]> but was:<[null]>
    at org.junit.Assert.failNotEquals(Assert.juava:834)
    at TestSort.testFindSmallest(TestSort.java:9)
    at TestSort.main(TestSort.java:24)
```

As before, we've set up for ourselves a little game to play, where our goal is now to modify the code for `Sort.findSmallest` so that this error no longer appears. This is a smaller goal than getting `Sort.sort` to work, which might be even more addictive.

Side note: It might have seem rather contrived that I just happened to return the right value `x[2]`. However, when I was recording this lecture video, I actually did make this exact mistake without intending to do so!

Next we turn to actually writing `findSmallest`. This seems like it should be relatively straightforward. If you're a Java novice, you might end up writing code that looks something like this:

```
/**  Returns the smallest string in x. */
public static String findSmallest(String[] x) {
    String smallest = x[0];
    for (int i = 0; i < x.length; i += 1) {
        if (x[i] < smallest) {
            smallest = x[i];
        }
    }
    return smallest;
}
```

However, this will yield the compilation error "< cannot be applied to 'java.lang.String'". The issue is that Java does not allow comparisons between Strings using the < operator.

When you're programming and get stuck on an issue like this that is easily describable, it's probably best to turn to a search engine. For example, we might search "less than strings Java" with Google. Such a search might yield a Stack Overflow post like this one.

One of the popular answers for this post explains that the `str1.compareTo(str2)` method will return a negative number if `str1 < str2`, 0 if they are equal, and a positive number if `str1 > str2`.

Incorporating this into our code, we might end up with:

```
/** Returns the smallest string in x.
  * @source Got help with string compares from https://goo.gl/a7yBU5. */
public static String findSmallest(String[] x) {
    String smallest = x[0];
    for (int i = 0; i < x.length; i += 1) {
        int cmp = x[i].compareTo(smallest);
        if (cmp < 0) {
            smallest = x[i];
        }
    }
    return smallest;
}
```

Note that we've used a `@source` tag in order to cite our sources. I'm showing this by example for those of you who are taking 61B as a formal course. This is not a typical real world practice.

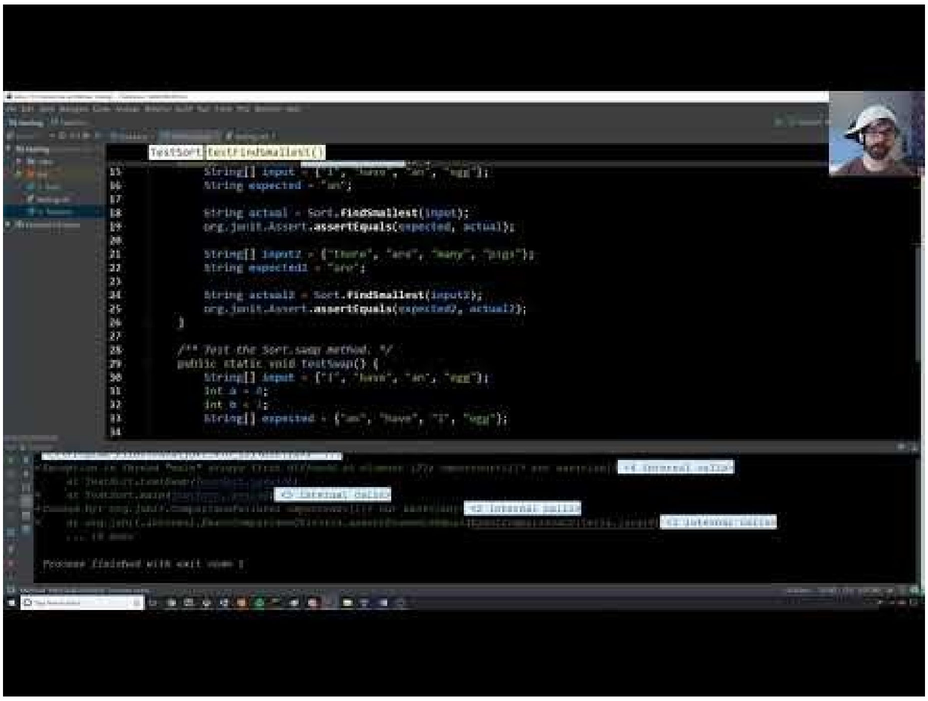
Since we are using syntax features that are totally new to us, we might lack confidence in the correctness of our `findSmallest` method. Luckily, we just wrote that test a little while ago. If we try running it, we'll see that nothing gets printed, which means our code is probably correct.

We can augment our test to increase our confidence by adding more test cases. For example, we might change `testFindSmallest` so that it reads as shown below:

```
public static void testFindSmallest() {
    String[] input = {"i", "have", "an", "egg"};
    String expected = "an";

    String actual = Sort.findSmallest(input);
    org.junit.Assert.assertEquals(expected, actual);

    String[] input2 = {"there", "are", "many", "pigs"};
    String expected2 = "are";

    String actual2 = Sort.findSmallest(input2);
    org.junit.Assert.assertEquals(expected2, actual2);
}
```

Rerunning the test, we see that it still passes. We are not absolutely certain that it works, but we are much more certain that we would have been without any tests.

Swap



Video link

Looking at our `sort` method below, the next helper method we need to write is something to move an item to the front, which we'll call `swap`.

```java
/** Sorts strings destructively. */
public static void sort(String[] x) {
       // find the smallest item
       // move it to the front
       // selection sort the rest (using recursion?)
}
```

Writing a `swap` method is very straightforward, and you've probably done so before. A correct implementation might look like:

```java
public static void swap(String[] x, int a, int b) {
    String temp = x[a];
    x[a] = x[b];
    x[b] = temp;
}
```

However, for the moment, let's introduce an intentional error so that we can demonstrate the utility of testing. A more naive programmer might have done something like:

```java
public static void swap(String[] x, int a, int b) {
    x[a] = x[b];
    x[b] = x[a];
}
```

Writing a test for this method is quite easy with the help of JUnit. An example test is shown below. Note that we have also edited the main method so that it calls `testSwap` instead of `testFindSmallest` or `testSort`.

```java
public class TestSort {
    ...

    /** Test the Sort.swap method. */
    public static void testSwap() {
        String[] input = {"i", "have", "an", "egg"};
        int a = 0;
        int b = 2;
        String[] expected = {"an", "have", "i", "egg"};

        Sort.swap(input, a, b);
        org.junit.Assert.assertArrayEquals(expected, input);
    }

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

Running this test on our buggy `swap` yields an error, as we'd expect.

```
Exception in thread "main" arrays first differed in element [2]; expected:<[i]> but wa
s:<[an]>
    at TestSort.testSwap(TestSort.java:36)
```

It's worth briefly noting that it is important that we call only `testSwap` and not `testSort` as well. For example, if our `main` method was as below, the entire `main` method will terminate execution as soon as `testSort` fails, and `testSwap` will never run:

```
public static void main(String[] args) {
    testSort();
    testFindSmallest();
    testSwap();
}
```

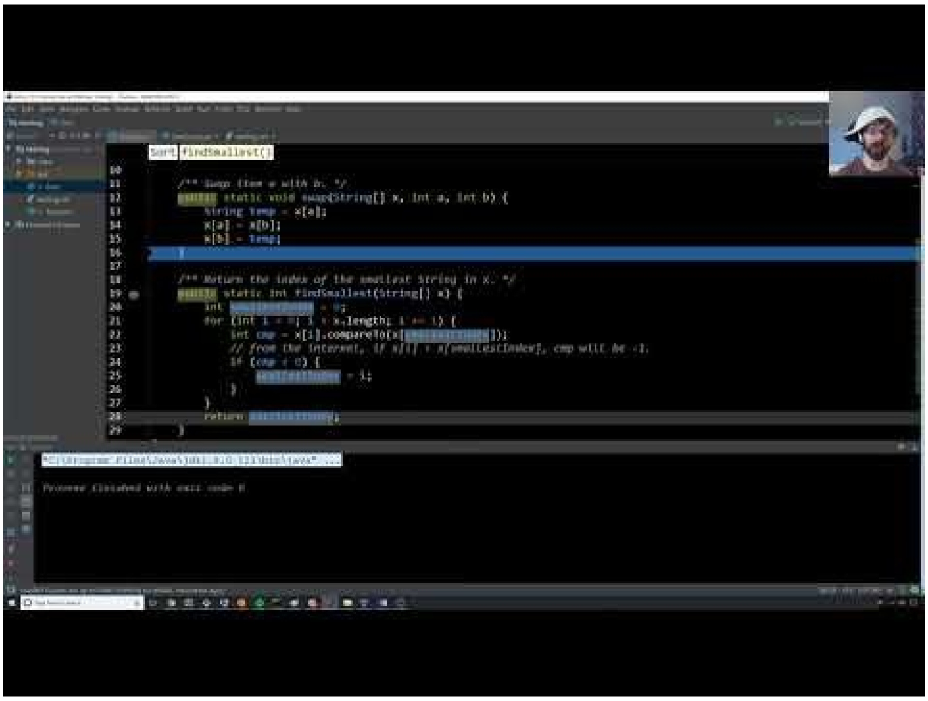
We will learn a more elegant way to deal with multiple tests at the end of this chapter that will avoid the need to manually specify which tests to run.

Now that we have a failing test, we can use it to help us debug. One way to do this is to set a breakpoint inside the `swap` method and use the visual debugging feature in IntelliJ. If you would like more information about and practice on debugging, check out Lab3. Stepping through the code line-by-line makes it immediately clear what is wrong (see video or try it yourself), and we can fix it by updating our code to include a temporary variable as that the beginning of this section:

```
public static void swap(String[] x, int a, int b) {
    String temp = x[a];
    x[a] = x[b];
    x[b] = temp;
}
```

Rerunning the test, we see that it now passes.

Revising findSmallest



Video link

Now that we have multiple pieces of our method done, we can start trying to connect them up together to create a `Sort` method.

```
/** Sorts strings destructively. */
public static void sort(String[] x) {
       // find the smallest item
       // move it to the front
       // selection sort the rest (using recursion?)
}
```

It's clear how to use our `findSmallest` and `swap` methods, but when we do so, we immediately realize there is a bit of a mismatch: `findSmallest` returns a `String`, and `swap` expects two indices.

```
/** Sorts strings destructively. */
public static void sort(String[] x) {
       // find the smallest item
       String smallest = findSmallest(x);

       // move it to the front
       swap(x, 0, smallest);

       // selection sort the rest (using recursion?)
}
```

In other words, what `findSmallest` should have been returning is the index of the smallest String, not the String itself. Making silly errors like this is normal and really easy to do, so don't sweat it if you find yourself doing something similar. Iterating on a design is part of the process of writing code.

Luckily, this new design can be easily changed. We simply need to adjust `findSmallest` to return an `int`, as shown below:

```
public static int findSmallest(String[] x) {
    int smallestIndex = 0;
    for (int i = 0; i < x.length; i += 1) {
        int cmp = x[i].compareTo(x[smallestIndex]);
        if (cmp < 0) {
            smallestIndex = i;
        }
    }
    return smallestIndex;
}
```

Since this is a non-trivial change, we should also update `testFindSmallest` and make sure that `findSmallest` still works.

```
public static void testFindSmallest() {
    String[] input = {"i", "have", "an", "egg"};
    int expected = 2;

    int actual = Sort.findSmallest(input);
    org.junit.Assert.assertEquals(expected, actual);

    String[] input2 = {"there", "are", "many", "pigs"};
    int expected2 = 1;

    int actual2 = Sort.findSmallest(input);
    org.junit.Assert.assertEquals(expected2, actual2);
}
```

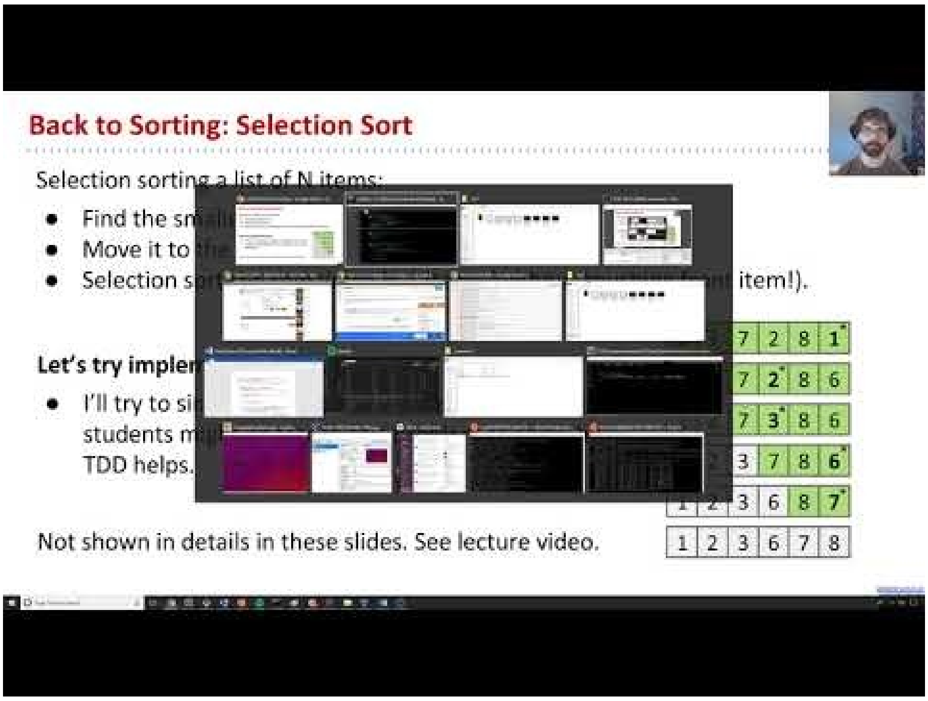
After modifying `TestSort` so that this test is run, and running `TestSort.main`, we see that our code passes the tests. Now, revising sort, we can fill in the first two steps of our sorting algorithm.

```java
/** Sorts strings destructively. */
public static void sort(String[] x) {
   // find the smallest item
   // move it to the front
   // selection sort the rest (using recursion?)
   int smallestIndex = findSmallest(x);
   swap(x, 0, smallestIndex);
}
```

All that's left is to somehow selection sort the remaining items, perhaps using recursion. We'll tackle this in the next section.

Reflecting on what we've accomplished, it's worth noting how we created tests first, and used these to build confidence that the actual methods work before we ever tried to use them for anything. This is an incredibly important idea, and one that will serve you well if you decide to adopt it.

Recursive Helper Methods



Video link

To begin this section, consider how you might make the recursive call needed to complete `sort`:

```java
/** Sorts strings destructively. */
public static void sort(String[] x) {
   int smallestIndex = findSmallest(x);
   swap(x, 0, smallestIndex);
   // recursive call??
}
```

For those of you who are used to a language like Python, it might be tempting to try and use something like slice notation, e.g.

```java
/** Sorts strings destructively. */
public static void sort(String[] x) {
   int smallestIndex = findSmallest(x);
   swap(x, 0, smallestIndex);
   sort(x[1:])
}
```

However, there is no such thing in Java as a reference to a sub-array, i.e. we can't just pass the address of the next item in the array.

This problem of needing to consider only a subset of a larger array is very common. A typical solution is to create a private helper method that has an additional parameter (or parameters) that delineate which part of the array to consider. For example, we might write a private helper method also called `sort` that consider only the items starting with item `start`.

```java
/** Sorts strings destructively starting from item start. */
private static void sort(String[] x, int start) {
    // TODO
}
```

Unlike our public sort method, it's relatively straightforward to use recursion now that we have the additional parameter `start`, as shown below. We'll test this method in the next section.

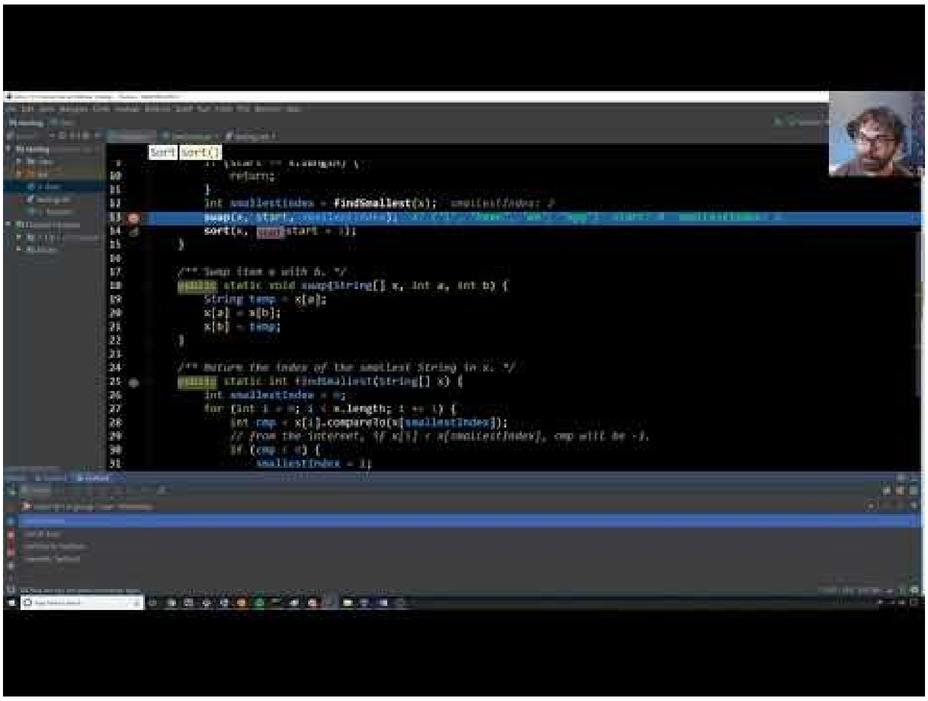
```java
/** Sorts strings destructively starting from item start. */
private static void sort(String[] x, int start) {
   int smallestIndex = findSmallest(x);
   swap(x, start, smallestIndex);
   sort(x, start + 1);
}
```

Now that we have a helper method, we need to set up the correct original call. If we set the start to 0, we effectively sort the entire array.

```java
/** Sorts strings destructively. */
public static void sort(String[] x) {
   sort(x, 0);
}
```

This approach is quite common when trying to use recursion on a data structure that is not inherently recursive, e.g. arrays.

Debugging and Completing Sort



[Video link](#)

Running our `testSort` method, we immediately run into a problem:

```
Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException: 4
    at Sort.swap(Sort.java:16)
```

Using the Java debugger, we see that the problem is that somehow `start` is reaching the value 4. Stepping through the code carefully (see video above), we find that the issue is that we forgot to include a base case in our recursive `sort` method. Fixing this is straightforward:

```
/** Sorts strings destructively starting from item start. */
private static void sort(String[] x, int start) {
   if (start == x.length) {
       return;
   }
   int smallestIndex = findSmallest(x);
   swap(x, start, smallestIndex);
   sort(x, start + 1);
}
```

Rerunning this test again, we get another error:

```
Exception in thread "main" arrays first differed at element [0];
   expected<[an]> bit was:<[have]>
```

Again, with judicious use of the IntelliJ debugger (see video), we can identify a line of code whose result does not match our expectations. Of note is the fact that I debugged the code at a higher level of abstraction than you might have otherwise, which I achieve by using `Step Over` more than `Step Into`. As discussed in lab 3, debugging at a higher level of abstraction saves you a lot of time and energy, by allowing you to compare the results of entire function calls with your expectation.

Specifically, we find that when sorting the last 3 (out of 4) items, the `findSmallest` method is giving as the 0th item (`"an"`) rather than the 3rd item (`"egg"`) when called on the input `{"an", "have", "i", "egg"}`. Looking carefully at the definition of `findSmallest`, this behavior is not a surprise, since `findSmallest` looks at the entire array, not just the items starting from position `start`. This sort of design flaw is very common, and writing tests and using the debugger is a great way to go about fixing them.

To fix our code, we revise `findSmallest` so that it takes a second parameter `start`, i.e. `findSmallest(String[] x, int start)`. In this way, we ensure that we're finding the smallest item only out of the last however many are still unsorted. The revision is as shown below:

```
public static int findSmallest(String[] x, int start) {
    int smallestIndex = start;
    for (int i = start; i < x.length; i += 1) {
        int cmp = x[i].compareTo(x[smallestIndex]);
        if (cmp < 0) {
            smallestIndex = i;
        }
    }
    return smallestIndex;
}
```

Given that we've made a significant change to one of our building blocks, i.e. `findSmallest`, we should ensure that our changes are correct.

We first modify `testFindSmallest` so that it uses our new parameter, as shown below:

```
public static void testFindSmallest() {
    String[] input = {"i", "have", "an", "egg"};
    int expected = 2;

    int actual = Sort.findSmallest(input, 0);
    org.junit.Assert.assertEquals(expected, actual);

    String[] input2 = {"there", "are", "many", "pigs"};
    int expected2 = 2;

    int actual2 = Sort.findSmallest(input2, 2);
    org.junit.Assert.assertEquals(expected2, actual2);
}
```

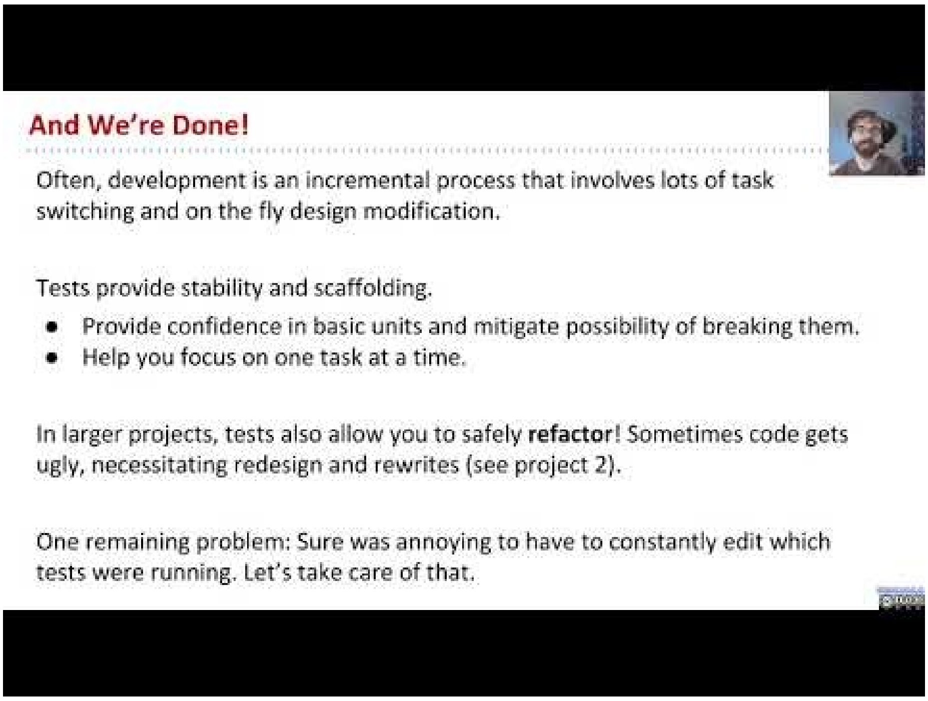
We then modify `TestSort.main` so that it runs `testFindSmallest`. This test passes, strongly suggesting that our revisions to `findSmallest` were correct.

We next modify `Sort.sort` so that it uses the new `start` parameter in `findSmallest`:

```
/** Sorts strings destructively starting from item start. */
private static void sort(String[] x, int start) {
   if (start == x.length) {
       return;
   }
   int smallestIndex = findSmallest(x, start);
   swap(x, start, smallestIndex);
   sort(x, start + 1);
}
```

We then modify `TestSort` so that it runs `TestSort.sort` and voila, the method works. We are done! You have now seen the "new way" from the beginning of this lecture, which we'll reflect on for the remainder of this chapter.

Reflections on the Development Process



Video link

When you're writing and debugging a program, you'll often find yourself switching between different contexts. Trying to hold too much in your brain at once is a recipe for disaster at worst, and slow progress at best.

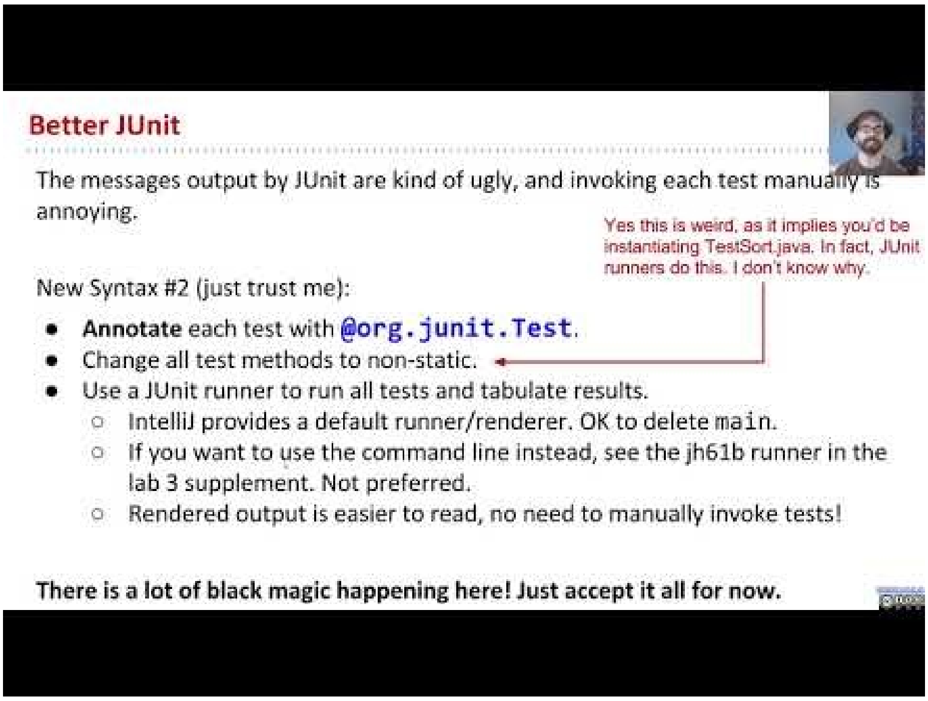
Having a set of automated tests helps reduce this cognitive load. For example, we were in the middle of writing `sort` when we realized there was a bug in `findSmallest`. We were able to switch contexts to consider `findSmallest` and establish that it was correct using our `testFindSmallest` method, and then switch back to `sort`. This is in sharp contrast to a more naive approach where you would simply be calling `sort` over and over and trying to figure out if the behavior of the overall algorithm suggests that the `findSmallest` method is correct.

As an analogy, you could test that a parachute's ripcord works by getting in an airplane, taking off, jumping out, and pulling the ripcord and seeing if the parachute comes out. However, you could also just pull it on the ground and see what happens. So, too, is it unnecessary to use `sort` to try out `findSmallest`.

As mentioned earlier in this chapter, tests also allow you to gain confidence in the basic pieces of your program, so that if something goes wrong, you have a better idea of where to start looking.

Lastly, tests make it easier to refactor your code. Suppose you decide to rewrite `findSmallest` so that it is faster or more readable. We can safely do so by making our desired changes and seeing if the tests still work.

Better JUnit



Video link

First, let's reflect on the new syntax we've seen today, namely `org.junit.Assert.assertEquals(expected, actual)`. This method (with a very long name) tests that `expected` and `actual` are equal, and if they are not, terminates the program with a verbose error message.

JUnit has many more such methods other than `assertEquals`, such as `assertFalse`, `assertNotNull`, `fail`, and so forth, and they can be found in the official JUnit documentation. JUnit also has many other complex features we will not describe or teach in 61B, though you're free to use them.

While JUnit certainly improved things, our test code from before was a bit clumsy in several ways. In the remainder of this section, we'll talk about two major enhancements you can make so that your code is cleaner and easier to use. These enhancements will seem very mysterious from a syntax point of view, so just copy what we're doing for now, and we'll explain some (but not all) of it in a later chapter.

The first enhancement is to use what is known as a "test annotation". To do this, we:

- Precede each method with `@org.junit.Test` (no semi-colon).
- Change each test method to be non-static.
- Remove our `main` method from the `TestSort` class.

Once we've done these three things, if we re-run our code in JUnit using the Run->Run command, all of the tests execute without having to be manually invoked. This annotation based approach has several advantages:

- No need to manually invoke tests.
- All tests are run, not just the ones we specify.
- If one test fails, the others still run.
- A count of how many tests were run and how many passed is provided.
- The error messages on a test failure are much nicer looking.
- If all tests pass, we get a nice message and a green bar appears, rather than simply getting no output.

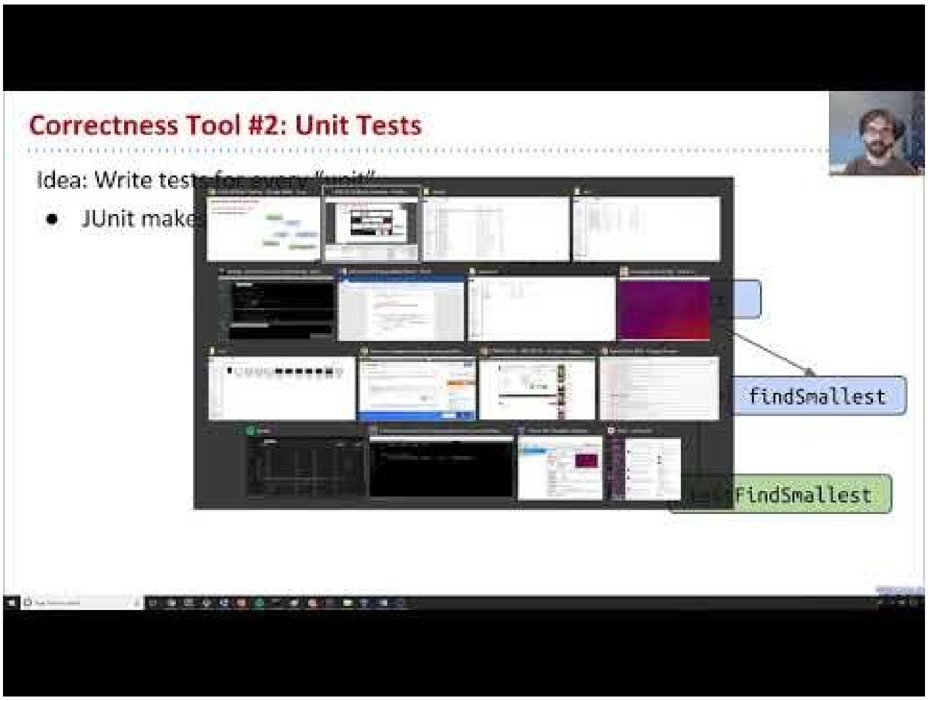
The second enhancement will let us use shorter names for some of the very lengthy method names, as well as the annotation name. Specifically, we'll use what is known as an "import statement".

We first add the import statement `import org.junit.Test;` to the top of our file. After doing this, we can replace all instances of `@org.junit.Test` with simply `@Test`.

We then add our second import statement `import static org.junit.Assert.*`. After doing this, anywhere we can omit anywhere we had `org.junit.Assert.`. For example, we can replace `org.junit.Assert.assertEquals(expected2, actual2);` with simply `assertEquals(expected2, actual2);`

We will explain exactly why import statements are in a later lecture. For now, just use and enjoy.

Testing Philosophy



[Video link]

Correctness Tool #1: Autograder

Let's go back to ground zero. The autograder was likely the first correctness tool you were exposed to. Our autograder is in fact based on JUnit plus some extra custom libraries.

There are some great benefits to autograders. Perhaps most importantly, it verifies correctness for you, saving you from the tedious and non-instructive task of writing all of your own tests. It also gamifies the assessment process by providing juicy points as an incentive to acheiving correctness. This can also backfire if students spend undue amounts of time chasing final points that won't actually affect their grade or learning.

However, autograders don't exist in the real world and relying on autograders can build bad habits. One's workflow is hindered by sporadically uploading your code and waiting for the autograder to run. *Autograder Driven Development* is an extreme version of this in which students write all their code, fix their compiler errors, and then submit to the autograder. After getting back errors, students may try to make some changes, sprinkle in print statements, and submit again. And repeat. Ultimately, you are not in control of either your workflow or your code if you rely on an autograder.

Correctness Tool #2: JUnit Tests

JUnit testing, as we have seen, unlocks a new world for you. Rather than relying on an autograder written by someone else, you write tests for each piece of your program. We refer to each of these pieces as a unit. This allows you to have confidence in each unit of your code - you can depend on them. This also helps decrease debugging time as you can isolate attention to one unit of code at a time (often a single method). Unit testing also forces you to clarify what each unit of code should be accomplishing.

There are some downsides to unit tests, however. First, writing thorough tests takes time. It's easy to write incomplete unit tests which give a false confidence to your code. It's also difficult to write tests for units that depend on other units (consider the `addFirst` method in your `LinkedListDeque`).

Test-Driven Development (TDD)

TDD is a development process in which we write tests for code before writing the code itself. The steps are as follows:

1. Identify a new feature.
2. Write a unit test for that feature.
3. Run the test. It should fail.
4. Write code that passes the test. Yay!
5. Optional: refactor code to make it faster, cleaner, etc. Except now we have a reference to tests that should pass.

Test-Driven Development is not required in this class and may not be your style but unit testing in general is most definitely a good idea.

Correctness Tool #3: Integration Testing

Unit tests are great but we should also make sure these units work properly together (unlike this meme). Integration testing verifies that components interact properly together. JUnit can in fact be used for this. You can imagine unit testing as the most nitty gritty, with integration testing a level of abstraction above this.

The challenge with integration testing is that it is tedious to do manually yet challenging to automate. And at a high level of abstraction, it's easy to miss subtle or rare errors.

As a summary, you should **definitely write tests but only when they might be useful!** Taking inspiration from TDD, writing your tests before writing code can also be very helpful in some cases.

What's next?

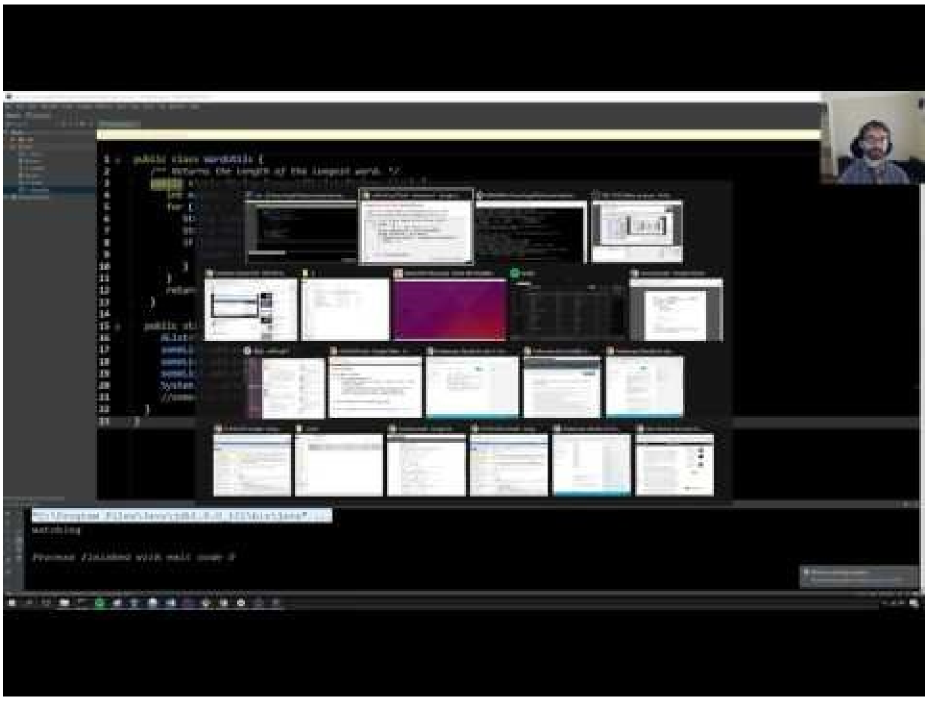
- Project 1b

The Problem

Recall the two list classes we created last week: SLList and AList. If you take a look at their documentation, you'll notice that they are very similar. In fact, all of their supporting methods are the same!

Suppose we want to write a class `WordUtils` that includes functions we can run on lists of words, including a method that calculates the longest string in an SLList.

Exercise 4.1.1. Try writing this method by yourself. The method should take in an SLList of strings and return the longest string in the list.



Video link

Here is the method that we came up with.

```
public static String longest(SLList<String> list) {
    int maxDex = 0;
    for (int i = 0; i < list.size(); i += 1) {
        String longestString = list.get(maxDex);
        String thisString = list.get(i);
        if (thisString.length() > longestString.length()) {
            maxDex = i;
        }
    }
    return list.get(maxDex);
}
```

How do we make this method work for AList as well?

All we really have to do is change the method's signature: the parameter

```
SLList<String> list
```

should be changed to

```
AList<String> list
```

Now we have two methods in our `WordUtils` class with exactly the same method name.

```
public static String longest(SLList<String> list)
```

and

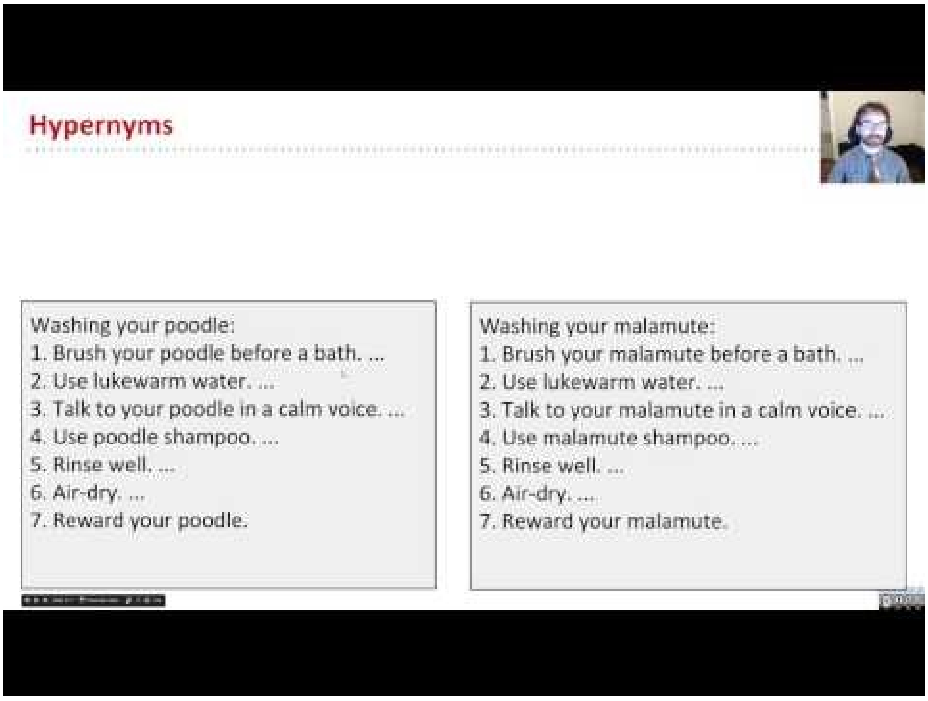
```
public static String longest(AList<String> list)
```

This is actually allowed in Java! It's something called *method overloading*. When you call WordUtils.longest, Java knows which one to run according to what kind of parameter you supply it. If you supply it with an AList, it will call the AList method. Same with an SLList.

It's nice that Java is smart enough to know how to deal with two of the same methods for different types, but overloading has several downsides:

- It's super repetitive and ugly, because you now have two virtually identical blocks of code.
- It's more code to maintain, meaning if you want to make a small change to the `longest` method such as correcting a bug, you need to change it in the method for each type of list.
- If we want to make more list types, we would have to copy the method for every new list class.

Hypernyms, Hyponyms, and Interface Inheritance



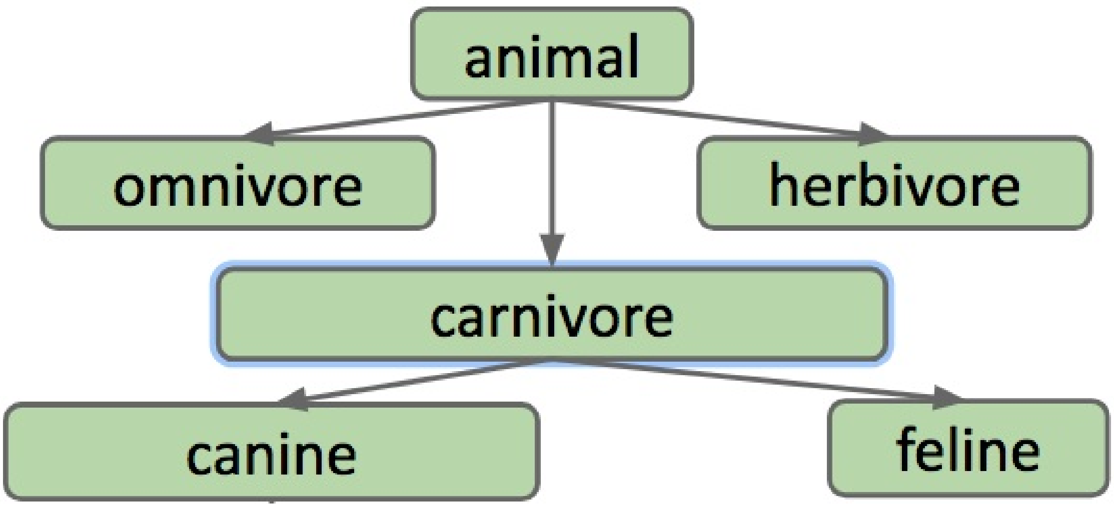
Video link

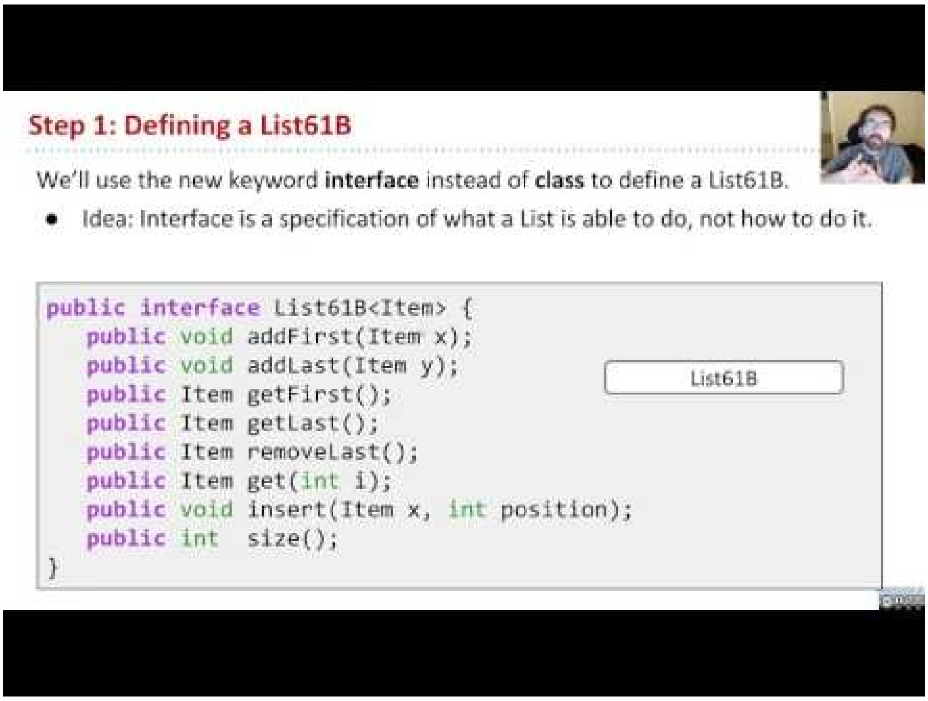
In the English language and life in general, there exist logical hierarchies to words and objects.

Dog is what is called a *hypernym of* poodle, malamute, husky, etc. In the reverse direction, poodle, malamute, and husky, are *hyponyms* of dog.

These words form a hierarchy of "is-a" relationships:

- a poodle "is-a" dog
- a dog "is-a" canine
- a canine "is-a" carnivore
- a carnivore "is-an" animal





[Video link](#)

The same hierarchy goes for SLLists and ALists! SLList and AList are both hyponyms of a more general list.

We will formalize this relationship in Java: if a SLList is a hyponym of List61B, then the SLList class is a **subclass** of the List61B class and the List61B class is a **superclass** of the SLList class.

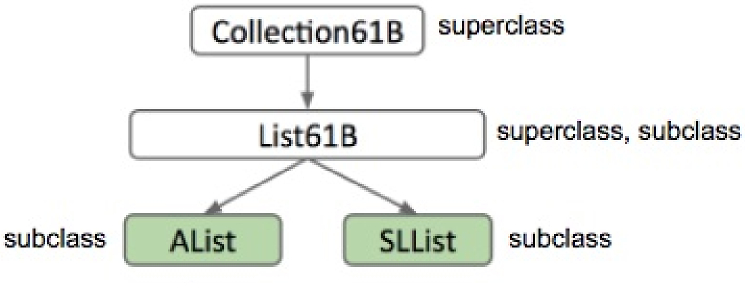


Figure 4.1.1

In Java, in order to *express* this hierarchy, we need to do **two things**:

- Step 1: Define a type for the general list hypernym -- we will choose the name List61B.
- Step 2: Specify that SLList and AList are hyponyms of that type.

The new List61B is what Java calls an **interface**. It is essentially a contract that specifies what a list must be able to do, but it doesn't provide any implementation for those behaviors. Can you think of why?

Here is our List61B interface. At this point, we have satisfied the first step in establishing the relationship hierarchy: creating a hypernym.

```
public interface List61B<Item> {
    public void addFirst(Item x);
    public void add Last(Item y);
    public Item getFirst();
    public Item getLast();
    public Item removeLast();
    public Item get(int i);
    public void insert(Item x, int position);
    public int size();
}
```

Now, to complete step 2, we need to specify that AList and SLList are hyponyms of the List61B class. In Java, we define this relationship in the class definition.

We will add to

`public class AList<Item> {...}`

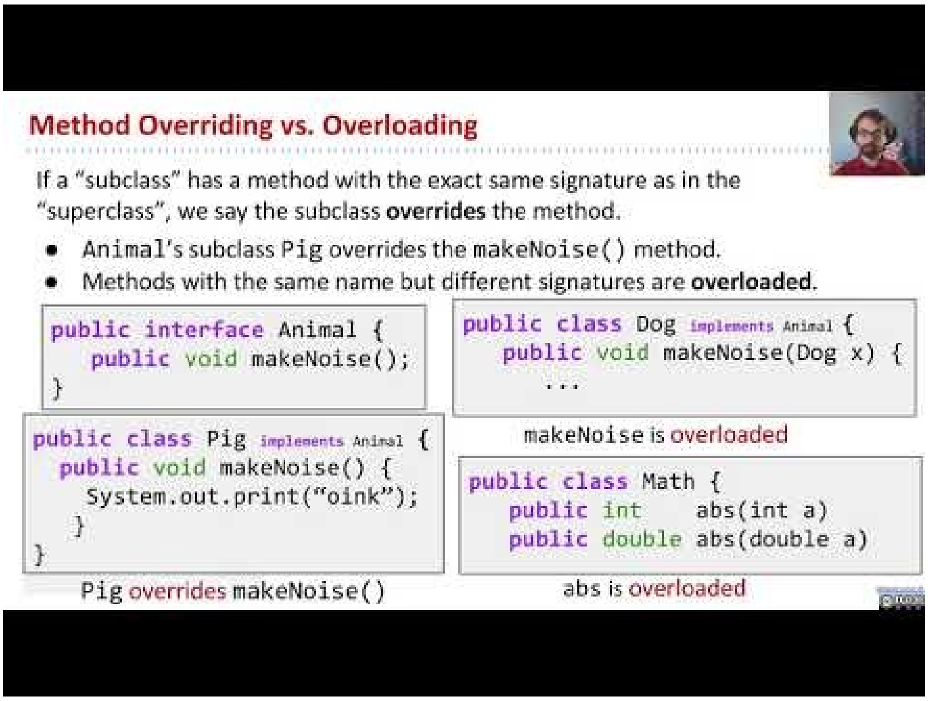
a relationship-defining word: implements.

`public class AList<Item> implements List61B<Item>{...}`

`implements List61B<Item>` is essentially a promise. AList is saying "I promise I will have and define all the attributes and behaviors specified in the List61B interface"

Now we can edit our `longest` method in `WordUtils` to take in a List61B. Because AList and SLList share an "is-a" relationship.

Overriding



Video link

We promised we would implement the methods specified in List61B in the AList and SLList classes, so let's go ahead and do that.

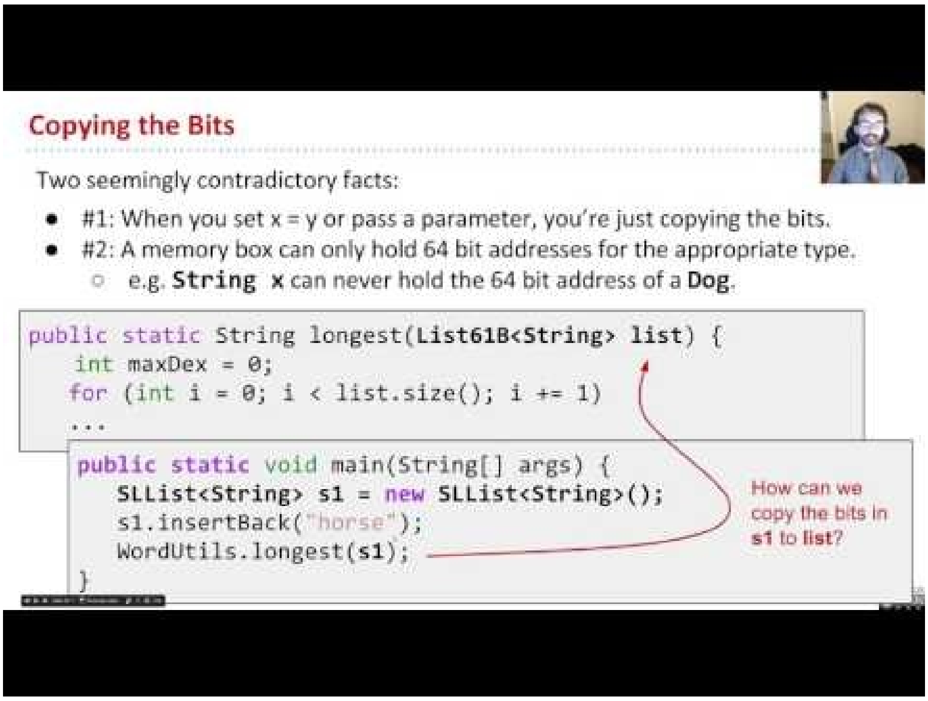
When implementing the required functions in the subclass, it's useful (and actually required in 61B) to include the `@Override` tag right on top of the method signature. Here, we have done that for just one method.

```
@Override
public void addFirst(Item x) {
    insert(x, 0);
}
```

It is good to note that even if you don’t include this tag, you *are* still overriding the method. So technically, you don't *have* to include it. However, including the tag acts as a safeguard for you as the programmer by alerting the compiler that you intend to override this method. Why would this be helpful you ask? Well, it's kind of like having a proofreader! The compiler will tell you if something goes wrong in the process.

Say you want to override the `addLast` method. What if you make a typo and accidentally write `addLsat` ? If you don't include the @Override tag, then you might not catch the mistake, which could make debugging a more difficult and painful process. Whereas if you include @Override, the compiler will stop and prompt you to fix your mistakes before your program even runs.

Interface Inheritance



[Video link](#)

Interface Inheritance refers to a relationship in which a subclass inherits all the methods/behaviors of the superclass. As in the List61B class we defined in the **Hyponyms and Hypernyms** section, the interface includes all the method signatures, but not implementations. It's up to the subclass to actually provide those implementations.

This inheritance is also multi-generational. This means if we have a long lineage of superclass/subclass relationships like in **Figure 4.1.1**, AList not only inherits the methods from List61B but also every other class above it all the way to the highest superclass AKA AList inherits from Collection.

GRoE

Recall the Golden Rule of Equals we introduced in the first chapter. This means whenever we make an assignment `a = b` , we copy the bits from b into a, with the requirement that b is the same type as a. You can't assign `Dog b = 1` or `Dog b = new Cat()` because 1 is not a Dog and neither is Cat.

Let's try to apply this rule to the `longest` method we wrote previously in this chapter.

`public static String longest(List61B<String> list)` takes in a List61B. We said that this could take in AList and SLList as well, but how is that possible since AList and List61B are different classes? Well, recall that AList shares an "is-a" relationship with List61B, Which means an AList should be able to fit into a List61B box!

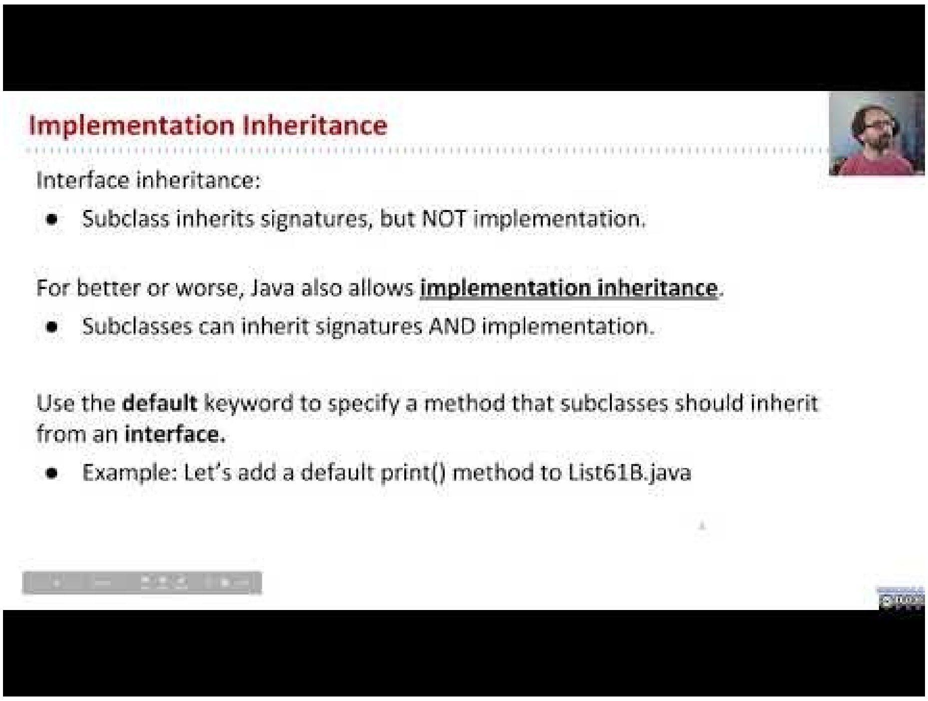
Exercise 4.1.2 Do you think the code below will compile? If so, what happens when it runs?

```java
public static void main(String[] args) {
    List61B<String> someList = new SLList<String>();
    someList.addFirst("elk");
}
```

Here are possible answers:

- Will not compile.
- Will compile, but will cause an error on the **new** line
- When it runs, an SLList is created and its address is stored in the someList variable, but it crashes on someList.addFirst() since the List class doesn't implement addFirst;
- When it runs, and SLList is created and its address is stored in the someList variable. Then the string "elk" is inserted into the SLList referred to by addFirst.

Implementation Inheritance



Video link

Previously, we had an interface List61B that only had method headers identifying **what** List61B's should do. But, now we will see that we can write methods in List61B that already have their implementation filled out. These methods identify **how** hypernyms of List61B should behave.

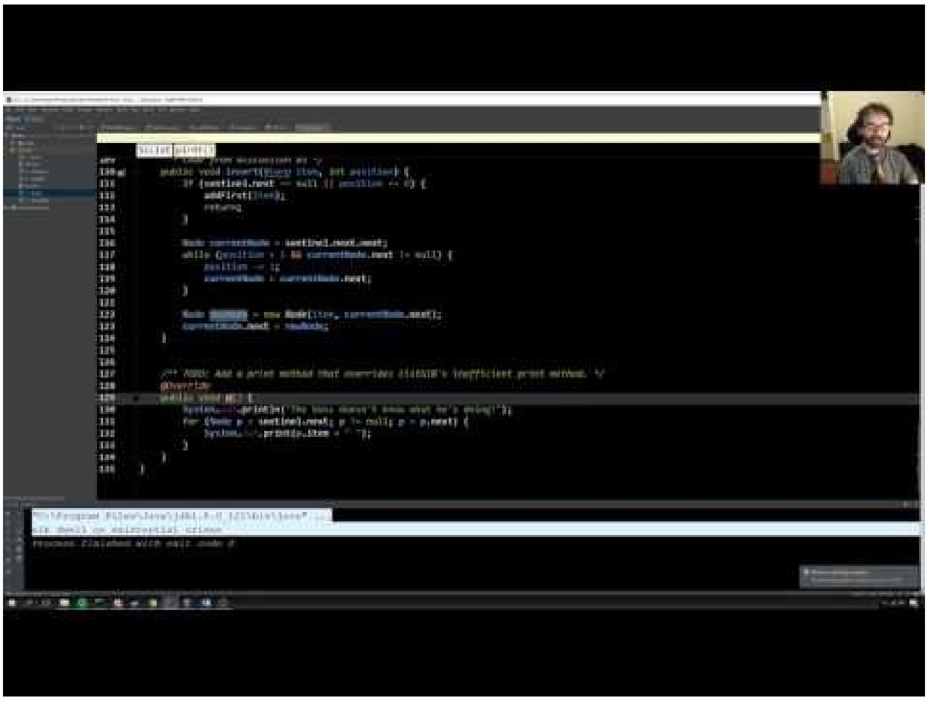
In order to do this, you must include the `default` keyword in the method signature.

If we define this method in List61B

```
default public void print() {
    for (int i = 0; i < size(); i += 1) {
        System.out.print(get(i) + " ");
    }
    System.out.println();
}
```

Then everything that implements the List61B class can use the method!

However, there is one small inefficiency in this method. Can you catch it?



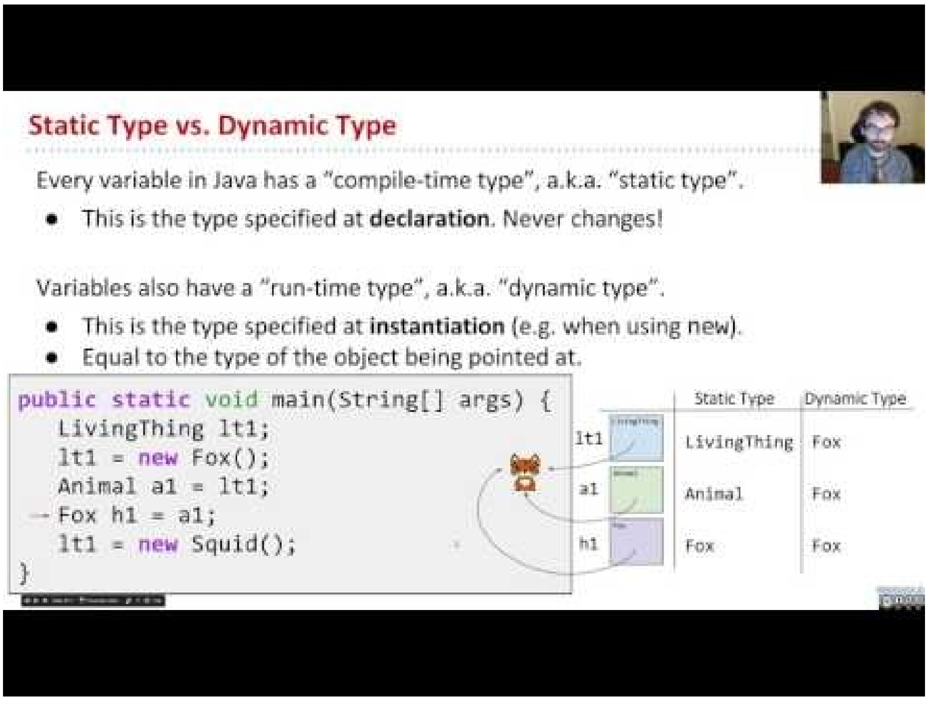
Video link

For an SLList, the `get` method needs to jump through the entirety of the list. during each call. It's much better to just print while jumping through!

We want SLList to print a different way than the way specified in its interface. To do this, we need to override it. In SLList, we implement this method;

```
@Override
public void print() {
    for (Node p = sentinel.next; p != null; p = p.next) {
        System.out.print(p.item + " ");
    }
}
```

Now, whenever we call print() on an SLList, it will call this method instead of the one in List61B.



Video link

You may be wondering, how does Java know which print() to call? Good question. Java is able to do this due to something called **dynamic method selection**.

We know that variables in java have a type.

```
List61B<String> lst = new SLList<String>();
```

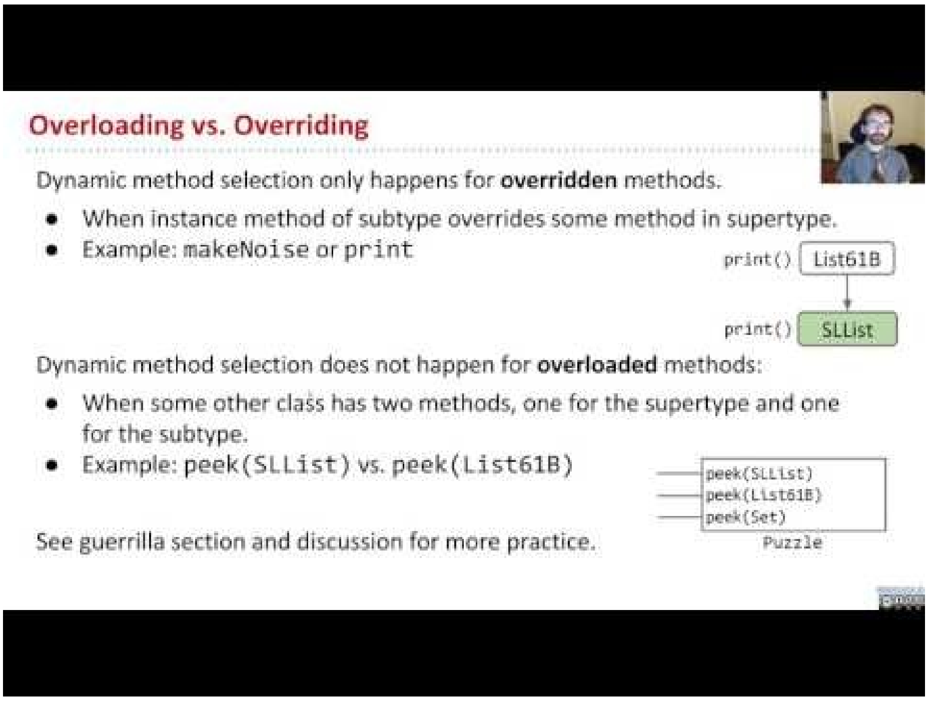
In the above declaration and instantiation, lst is of type "List61B". This is called the "static type"

However, the objects themselves have types as well. the object that lst points to is of type SLList. Although this object is intrinsically an SLList (since it was declared as such), it is also a List61B, because of the “is-a” relationship we explored earlier. But, because the object itself was instantiated using the SLList constructor, We call this its "dynamic type".

Aside: the name “dynamic type” is actually quite semantic in its origin! Should lst be reassigned to point to an object of another type, say a AList object, lst’s dynamic type would now be AList and not SLList! It’s dynamic because it changes based on the type of the object it’s currently referring to.

When Java runs a method that is overriden, it searches for the appropriate method signature in it's **dynamic type** and runs it.

IMPORTANT: This does not work for overloaded methods!



[Video link]

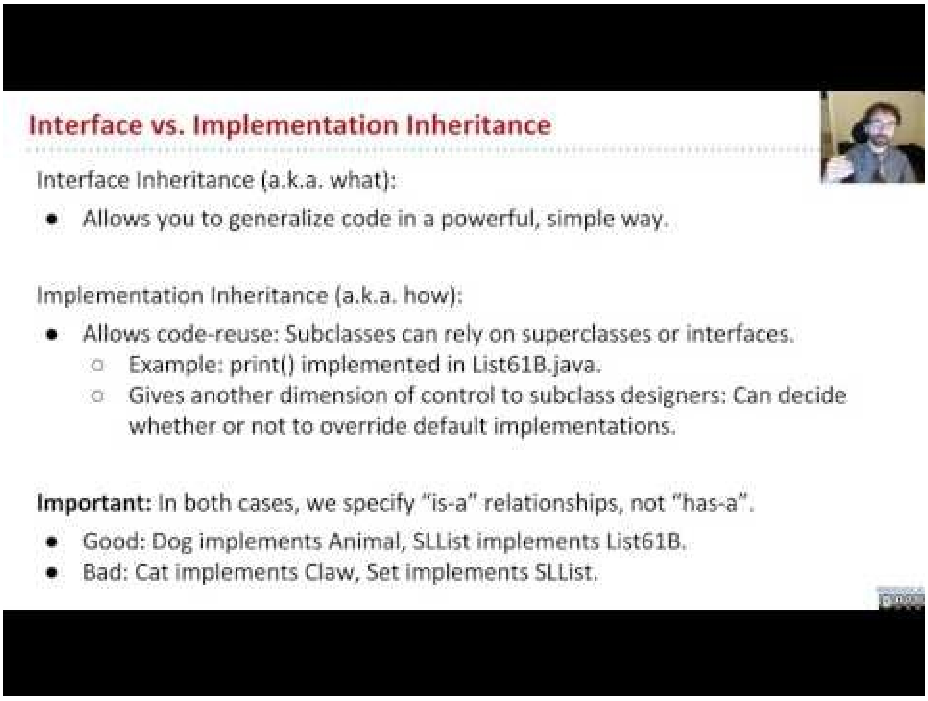
Say there are two methods in the same class

```
public static void peek(List61B<String> list) {
    System.out.println(list.getLast());
}
public static void peek(SLList<String> list) {
    System.out.println(list.getFirst());
}
```

and you run this code

```
SLList<String> SP = new SLList<String>();
List61B<String> LP = SP;
SP.addLast("elk");
SP.addLast("are");
SP.addLast("cool");
peek(SP);
peek(LP);
```

The first call to peek() will use the second peek method that takes in an SLList. The second call to peek() will use the first peek method which takes in a List61B. This is because the only distinction between two overloaded methods is the types of the parameters. When Java checks to see which method to call, it checks the **static type** and calls the method with the parameter of the same type.



Video link

Interface Inheritance vs Implementation Inheritance

How do we differentiate between "interface inheritance" and "implementation inheritance"? Well, you can use this simple distinction:

- Interface inheritance (what): Simply tells what the subclasses should be able to do.
  - EX) all lists should be able to print themselves, how they do it is up to them.
- Implementation inheritance (how): Tells the subclasses how they should behave.
  - EX) Lists should print themselves exactly this way: by getting each element in order and then printing them.

When you are creating these hierarchies, remember that the relationship between a subclass and a superclass should be an "is-a" relationship. AKA Cat should only implement Animal Cat **is an** Animal. You should not be defining them using a "has-a" relationship. Cat **has-a** Claw, but Cat definitely should not be implementing Claw.

Finally, Implementation inheritance may sound nice and all but there are some drawbacks:

- We are fallible humans, and we can't keep track of everything, so it's possible that you overrode a method but forgot you did.
- It may be hard to resolve conflicts in case two interfaces give conflicting default methods.
- It encourages overly complex code

What's Next?

- Discussion 4 Inheritance

Extends

Now you've seen how we can use the `implements` keyword to define a hierarchical relationship with interfaces. What if we wanted to define a hierarchical relationship between classes?

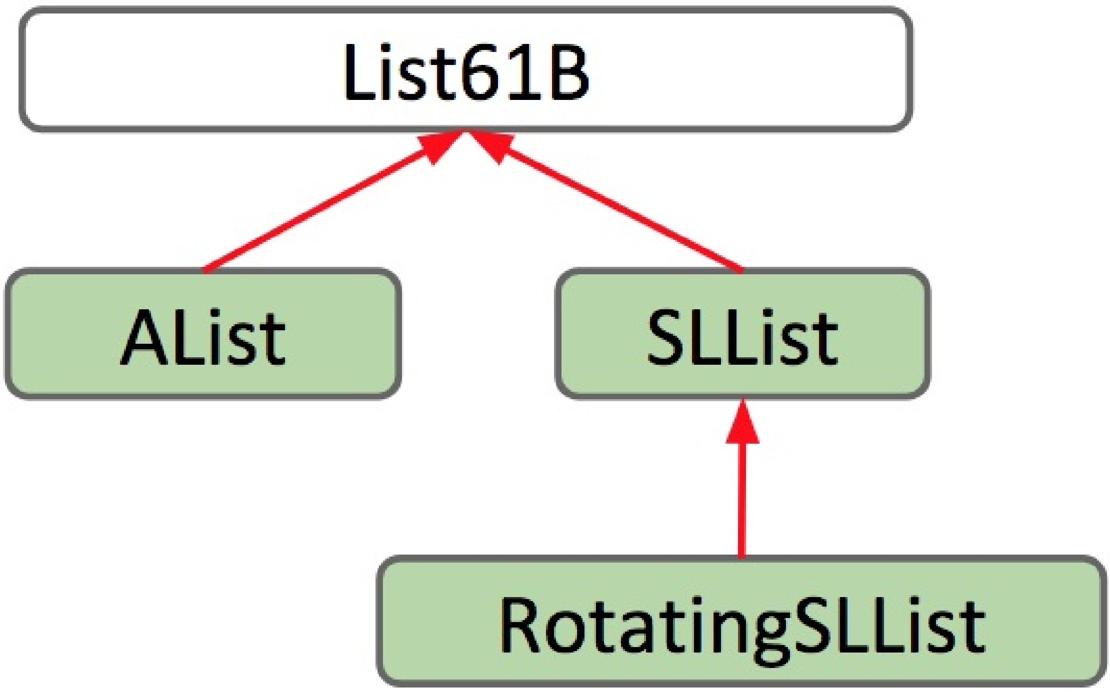
Suppose we want to build a RotatingSLList that has the same functionality as the SLList like `addFirst`, `size`, etc., but with an additional `rotateRight` operation to bring the last item to the front of the list.

One way you could do this would be to copy and paste all the methods from SLList and write `rotateRight` on top of it all - but then we wouldn't be taking advantage of the power of inheritance! Remember that inheritance allows subclasses to *reuse* code from an already defined class. So let's define our RotatingSLList class to inherit from SLList.

We can set up this inheritance relationship in the class header, using the `extends` keyword like so:

```
public class RotatingSLList<Item> extends SLList<Item>
```

In the same way that AList shares an "is-a" relationship with List61B, RotatingSLList shares an "is-a" relationship SLList. The `extends` keyword lets us keep the original functionality of SLList, while enabling us to make modifications and add additional functionality.

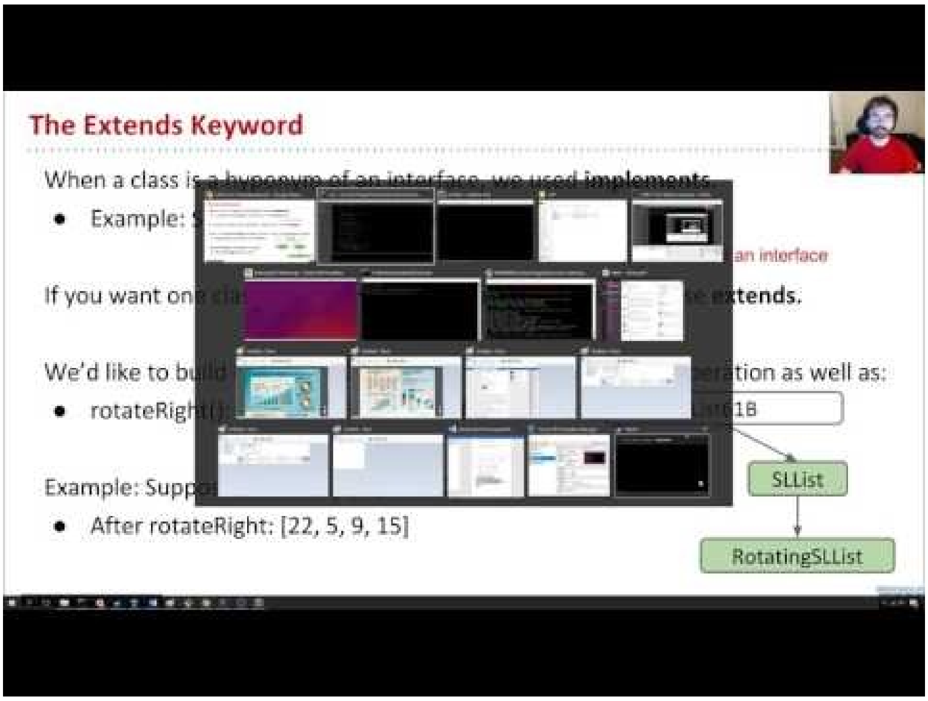


Now that we've defined our RotatingSLList to extend from SLList, let's give it its unique ability to rotate.

Exercise 4.2.1. Define the `rotateRight` method, which takes in an existing list, and rotates every element one spot to the right, moving the last item to the front of the list.

For example, calling `rotateRight` on [5, 9, 15, 22] should return [22, 5, 9, 15].

Tip: are there any inherited methods that might be helpful in doing this?



Video link

Here's what we came up with.

```
public void rotateRight() {
    Item x = removeLast();
    addFirst(x);
}
```

You might have noticed that we were able to use methods defined outside of RotatingSLList, because we used the `extends` keyword to inherit them from SLList. That gives rise to the question: What exactly do we inherit?

By using the `extends` keyword, subclasses inherit all **members** of the parent class. "Members" includes:

- All instance and static variables
- All methods
- All nested classes

Note that constructors are not inherited, and private members cannot be directly accessed by subclasses.

VengefulSLList

Notice that when someone calls `removeLast` on an SLList, it throws that value away - never to be seen again. But what if those removed values left and started a massive rebellion against us? In this case, we need to remember what those removed (or rather defected >:() values were so we can hunt them down and terminate them later.

We create a new class, VengefulSLList, that remembers all items that have been banished by `removeLast` .

Like before, we specify in VengefulSLList's class header that it should inherit from SLList.

```
public class VengefulSLList<Item> extends SLList<Item>
```

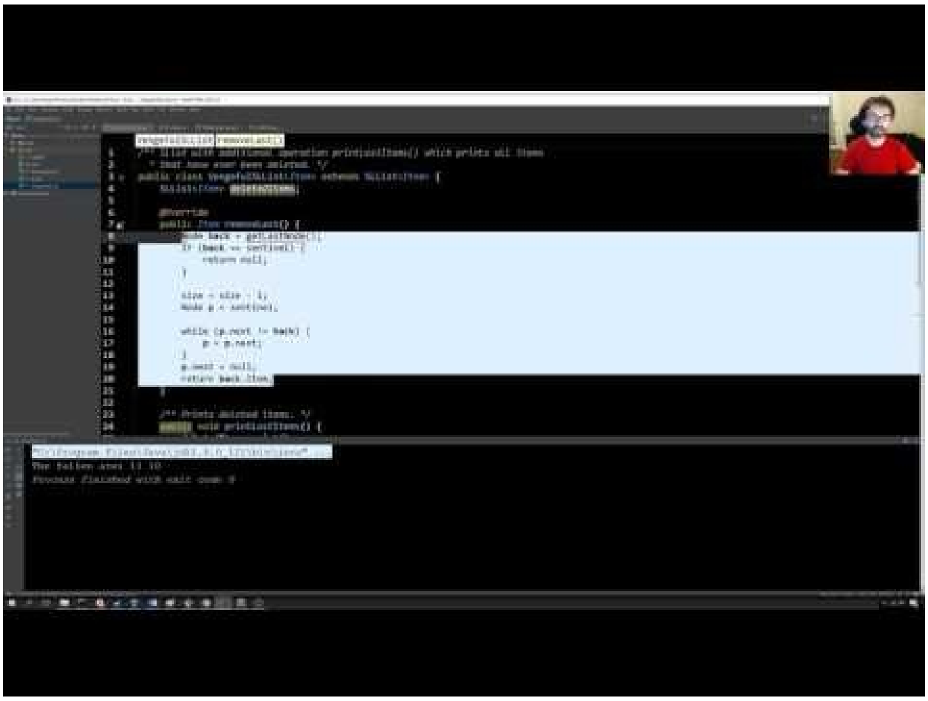
Now, let's give VengefulSLList a method to print out all of the items that have been removed by a call to the `removeLast` method, `printLostItems()` . We can do this by adding an instance variable that can keep track of all the deleted items. If we use an SLList to keep track of our items, then we can simply make a call to the `print()` method to print out all the items.

So far this is what we have:

```
public class VengefulSLList<Item> extends SLList<Item> {
    SLList<Item> deletedItems;

    public void printLostItems() {
        deletedItems.print();
    }
}
```

VengefulSLList's `removeLast` should do exactly the same thing that SLList's does, except with one additional operation - adding the removed item to the `deletedItems` list. In an effort to *reuse* code, we can **override** the `removeLast` method to modify it to fit our needs, and call the `removeLast` method defined in the parent class, SLList, using the `super` keyword.

Exercise 4.2.2. Override the `removeLast` method to remove the last item, add that item to the `deletedItems` list, then return it.



Video link

Finally, VengefulSLList remembers all items deleted from it, as intended.

```
public class VengefulSLList<Item> extends SLList<Item> {
    SLList<Item> deletedItems;

    public VengefulSLList() {
        deletedItems = new SLList<Item>();
    }

    @Override
    public Item removeLast() {
        Item x = super.removeLast();
        deletedItems.addLast(x);
        return x;
    }

    /** Prints deleted items. */
    public void printLostItems() {
        deletedItems.print();
    }
}
```

Constructors Are Not Inherited

As we mentioned earlier, subclasses inherit all members of the parent class, which includes instance and static variables, methods, and nested classes, but does *not* include constructors.

While constructors are not inherited, Java requires that all constructors **must start with a call to one of its superclass's constructors**.

To gain some intuition on why that it is, recall that the `extends` keywords defines an "is-a" relationship between a subclass and a parent class. If a VengefulSLList "is-an" SLList, then it follows that every VengefulSLList must be set up like an SLList.

Here's a more in-depth explanation. Let's say we have two classes:

```
public class Human {...}
```

```
public class TA extends Human {...}
```

It is logical for TA to extend Human, because all TA's are Human. Thus, we want TA's to inherit the attributes and behaviors of Humans.

If we run the code below:

`TA Christine = new TA();`

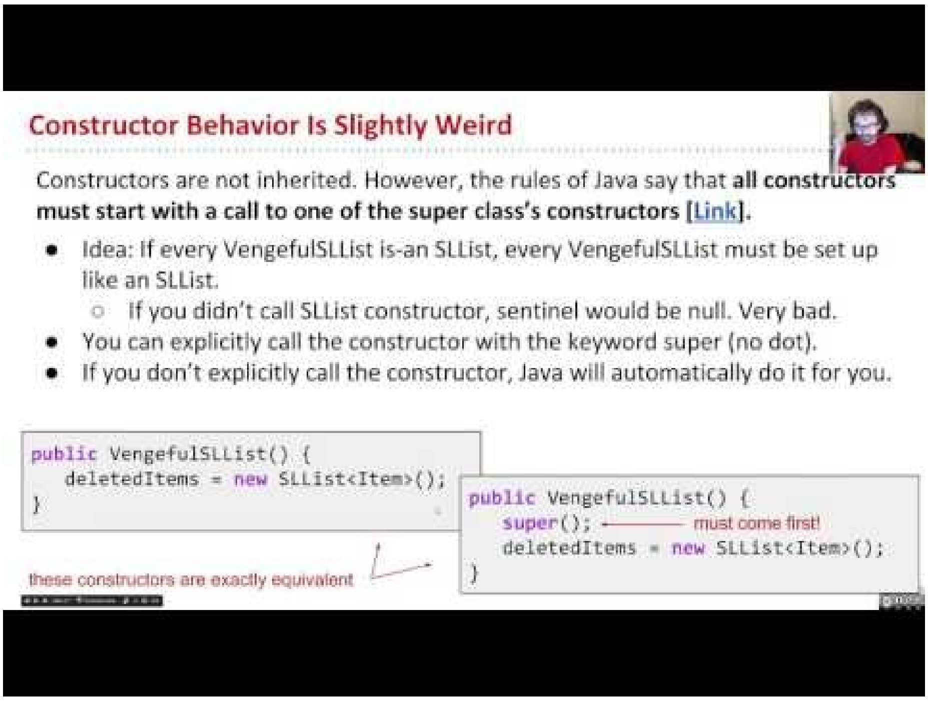
Then first, a Human must be created. Then, that Human can be given the qualities of a TA. It doesn't make sense for a TA to be constructed without first creating a Human first.

Thus, we can either explicitly make a call to the superclass's constructor, using the `super` keyword:

```
public VengefulSLList() {
    super();
    deletedItems = new SLList<Item>();
}
```

Or, if we choose not to, Java will automatically make a call to the superclass's *no-argument* constructor for us.

In this case, adding `super()` has no difference from the constructor we wrote before. It just makes explicit what was done implicitly by Java before. However, if we were to define another constructor in VengefulSLList, Java's implicit call may not be what we intend to call.



Video link

Suppose we had a one-argument constructor that took in an item. If we had relied on an implicit call to the superclass's *no-argument* constructor, `super()`, the item passed in as an argument wouldn't be placed anywhere!

So, we must make an explicit call to the correct constructor by passing in the item as a parameter to super.

```
public VengefulSLList(Item x) {
    super(x);
    deletedItems = new SLList<Item>();
}
```

The Object Class

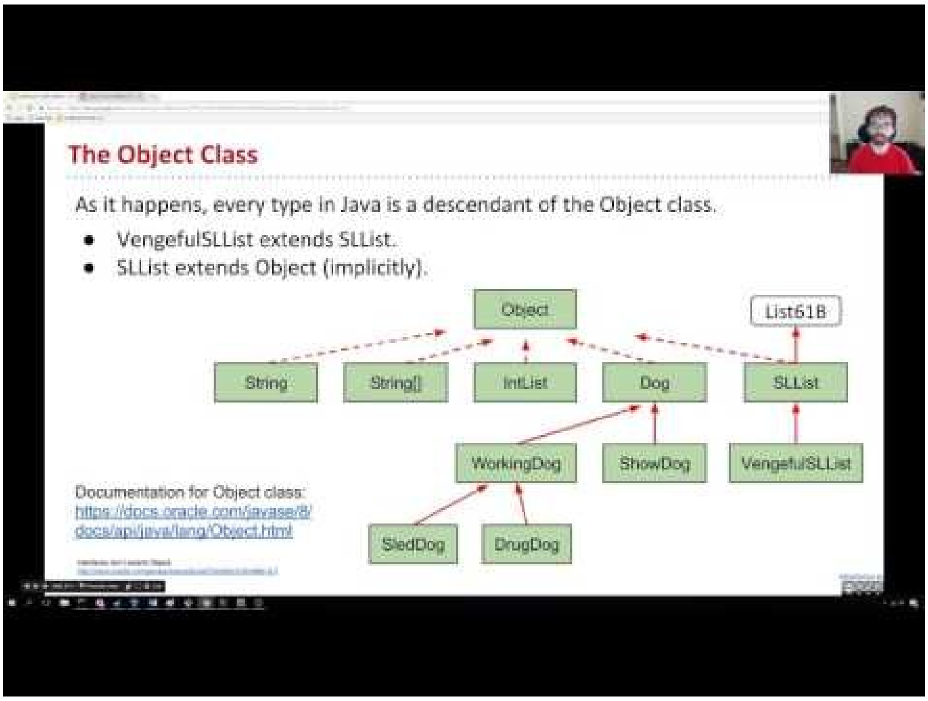
Every class in Java is a descendant of the Object class, or `extends` the Object class. Even classes that do not have an explicit `extends` in their class still *implicitly* extend the Object class.

For example,

- VengefulSLList `extends` SLList explicitly in its class declaration
- SLList `extends` Object implicitly

This means that since SLList inherits all members of Object, VengefulSLList inherits all members of SLList *and* Object, transitively. So, what is to be inherited from Object?

As seen in the documentation for the Object class, the Object class provides operations that every Object should be able to do - like `.equals(Object obj)`, `.hashCode()`, and `toString()`.



Video link

Is-a vs. Has-a

Important Note: The `extends` keyword defines "is-a", or hypernymic relationships. A common mistake is to instead use it for "has-a", or meronymic relationships.

When extending a class, a wise thing to do would be to ask yourself if the "is-a" relationship makes sense.

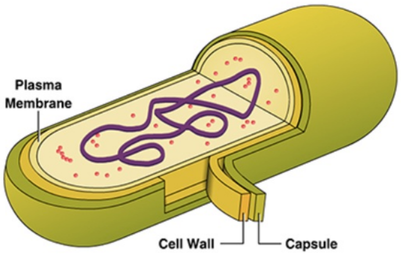
- `Shower` is a `Bathroom`? No!
- `VengefulSLList` is a `SLList`? Yes!

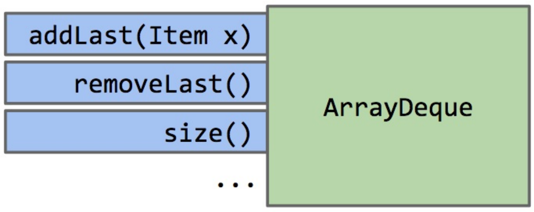
Encapsulation

Encapsulation is one of the fundamental principles of object oriented programming, and is one of the approaches that we take as programmers to resist our biggest enemy: *complexity*. Managing complexity is one of the major challenges we must face when writing large programs.

Some of the tools we can use to fight complexity include hierarchical abstraction (abstraction barriers!) and a concept known as "Design for change". This revolves around the idea that programs should be built into modular, interchangeable pieces that can be swapped around without breaking the system. Additionally, **hiding information** that others don't need is another fundamental approach when managing a large system.

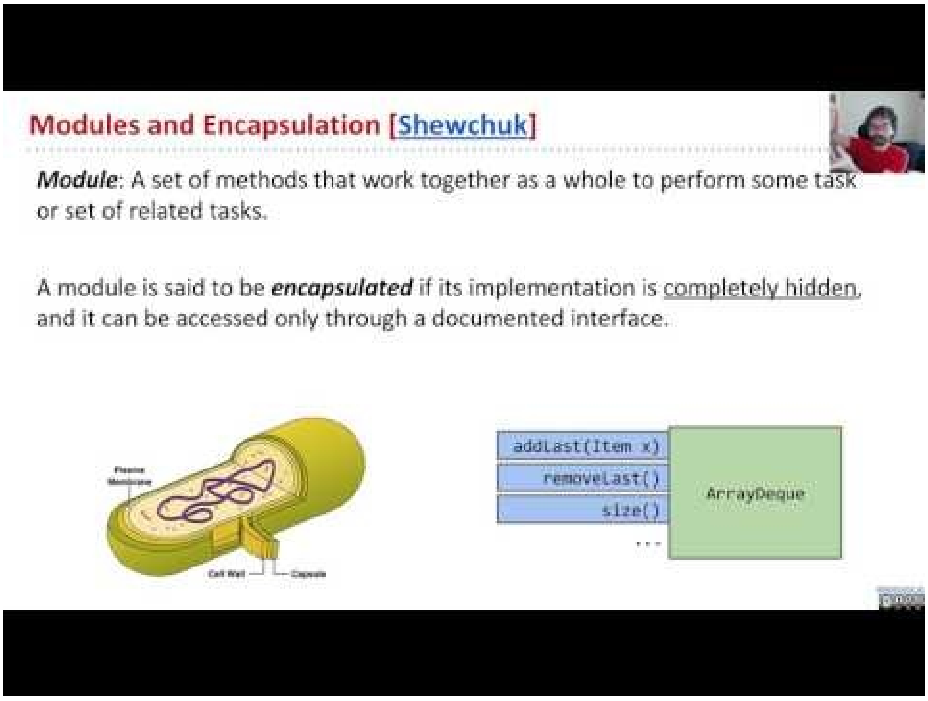
The root of encapsulation lies in this notion of hiding information from the outside. One way to look at it is to see how encapsulation is analogous to a Human cell. The internals of a cell may be extremely complex, consisting of chromosomes, mitochondria, ribosomes etc., but yet it is fully encapsulated into a single *module* - abstracting away the complexity inside.





In computer science terms, a module can be defined as a set of methods that work together as a whole to perform a task or set of related tasks. This could be something like a class that represents a list. Now, if the implementation details of a module are kept internally hidden and the only way to interact with it is through a documented interface, then that module is said to be encapsulated.

Take the `ArrayDeque` class, for example. The outside world is able to utilize and interact with an `ArrayDeque` through its defined methods, like `addLast` and `removeLast`. However, they need not understand the complex details of how the data structure was implemented in order to be able to use it effectively.



Video link

Abstraction Barriers

Ideally, a user should not be able to observe the internal workings of, say, a data structure they are using. Fortunately, Java makes it easy to enforce abstraction barriers. Using the `private` keyword in Java, it becomes virtually impossible to look inside an object - ensuring that the underlying complexity isn't exposed to the outside world.

How Inheritance Breaks Encapsulation

Suppose we had the following two methods in a Dog class. We could have implemented `bark` and `barkMany` like so:

```
public void bark() {
    System.out.println("bark");
}

public void barkMany(int N) {
    for (int i = 0; i < N; i += 1) {
        bark();
    }
}
```

Or, alternatively, we could have implemented it like so:

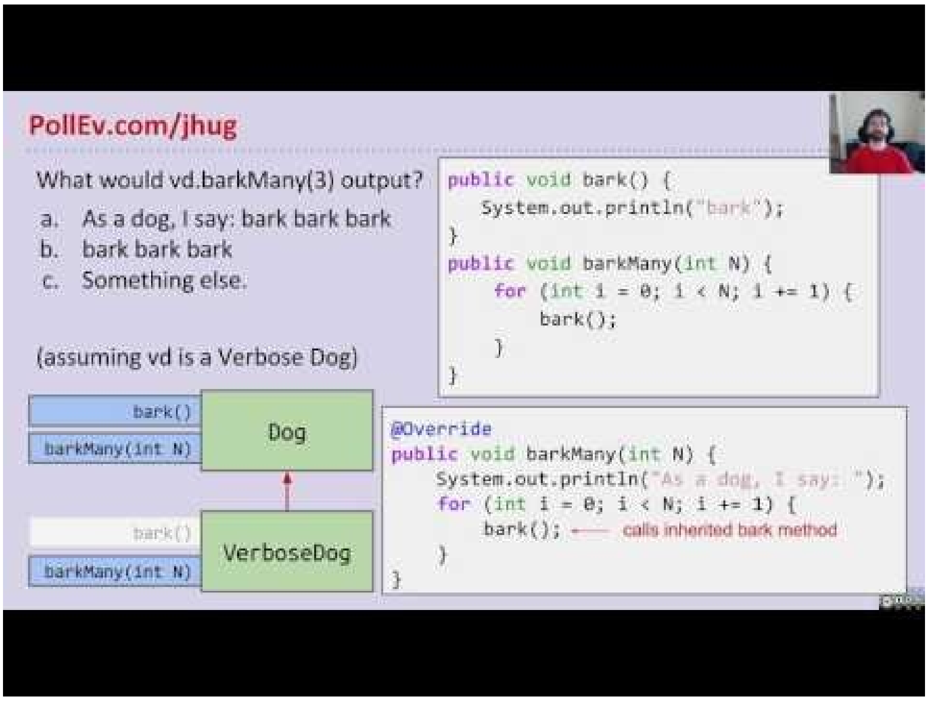
```
public void bark() {
    barkMany(1);
}

public void barkMany(int N) {
    for (int i = 0; i < N; i += 1) {
        System.out.println("bark");
    }
}
```

From a user's perspective, the functionality of either of these implementations is exactly the same. However, observe the effect if we were to define a a subclass of `Dog` called `VerboseDog`, and override its `barkMany` method as such:

```
@Override
public void barkMany(int N) {
    System.out.println("As a dog, I say: ");
    for (int i = 0; i < N; i += 1) {
        bark();
    }
}
```

Exercise 4.2.3. Given a VerboseDog `vd`, what would `vd.barkMany(3)` output, given the first implementation above? The second implementation?

- a: As a dog, I say: bark bark bark
- b: bark bark bark
- c: Something else



[Video link]

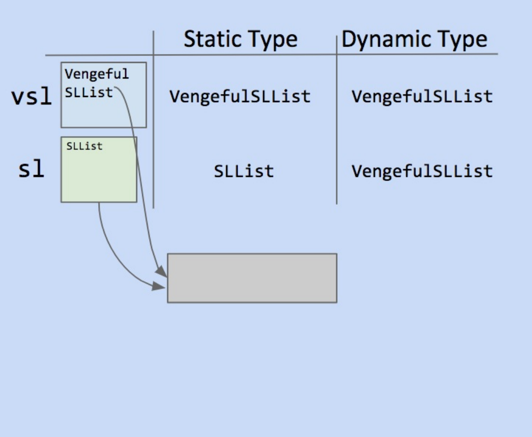
As you have seen, using the first implementation, the output is A, while using the second implementation, the program gets caught in an infinite loop. The call to `bark()` will call `barkMany(1)`, which makes a call to `bark()`, repeating the process infinitely many times.

Type Checking and Casting

Before we go into types and casting, let's review dynamic method selection. Recall that dynamic method lookup is the process of determining the method that is executed at runtime based on the dynamic type of the object. Specifically, if a method in SLList is overridden by the VengefulSLList class, then the method that is called at runtime is determined by the run-time type, or dynamic type, of that variable.

Exercise 4.2.4. For each line of code below, decide the following:

- Does that line cause a compilation error?
- Which method uses dynamic selection?




```java
public static void main(String[] args) {
    VengefulSLList<Integer> vsl =
          new VengefulSLList<Integer>(9);
    SLList<Integer> sl = vsl;

    sl.addLast(50);
    sl.removeLast();

    sl.printLostItems();
    VengefulSLList<Integer> vsl2 = sl;
}
```

Let's go through this program line by line.

```
VengefulSLList<Integer> vsl = new VengefulSLList<Integer>(9);
SLList<Integer> sl = vsl;
```

These two lines above compile just fine. Since `VengefulSLList` "is-an" SLList, it's valid to put an instance of the `VengefulSLList` class inside an `SLList` "container".

```
sl.addLast(50);
sl.removeLast();
```

These lines above also compile. The call to `addLast` is unambiguous, as VengefulSLList did not override or implement it, so the method executed is in SLList. The `removeLast` method is overridden by VengefulSLList, however, so we take a look at the dynamic type of `sl`. Its dynamic type is VengefulSLList, and so dynamic method selection chooses the overridden method in the VengefulSLList class.

```
sl.printLostItems();
```

This line above results in a compile-time error. Remember that the compiler determines whether or not something is valid based on the static type of the object. Since `sl` is of static type SLList, and `printLostItems` is not defined in the SLList class, the code will not be allowed to run, *even though* `sl` 's runtime type is VengefulSLList.

```
VengefulSLList<Integer> vsl2 = sl;
```

This line above also results in a compile-time error, for a similar reason. In general, the compiler only allows method calls and assignments based on compile-time types. Since the compiler only sees that the static type of `sl` is SLList, it will not allow a VengefulSLList "container" to hold it.

Expressions

Like variables as seen above, expressions using the `new` keyword also have compile-time types.

```
SLList<Integer> sl = new VengefulSLList<Integer>();
```

Above, the compile-time type of the right-hand side of the expression is VengefulSLList. The compiler checks to make sure that VengefulSLList "is-a" SLList, and allows this assignment,

```
VengefulSLList<Integer> vsl = new SLList<Integer>();
```

Above, the compile-time type of the right-hand side of the expression is SLList. The compiler checks if SLList "is-a" VengefulSLList, which it is not in all cases, and thus a compilation error results.

Further, method calls have compile-time types equal to their declared type. Suppose we have this method:

```
public static Dog maxDog(Dog d1, Dog d2) { ... }
```

Since the return type of `maxDog` is Dog, any call to `maxDog` will have compile-time type Dog.

```
Poodle frank = new Poodle("Frank", 5);
Poodle frankJr = new Poodle("Frank Jr.", 15);

Dog largerDog = maxDog(frank, frankJr);
Poodle largerPoodle = maxDog(frank, frankJr); //does not compile! RHS has compile-time
 type Dog
```

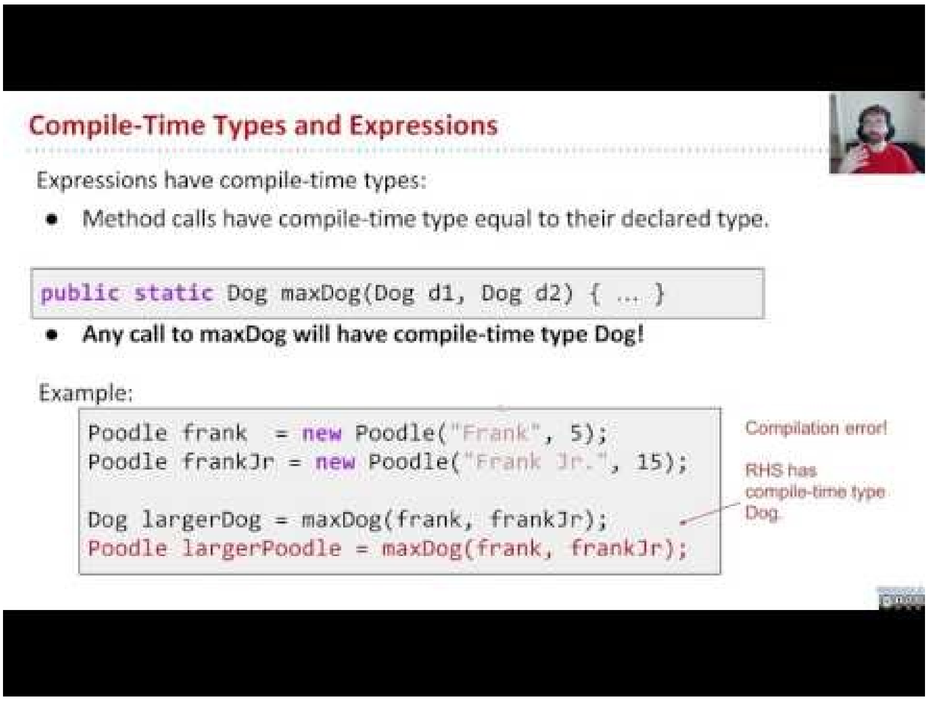
Assigning a Dog object to a Poodle variable, like in the SLList case, results in a compilation error. A Poodle "is-a" Dog, but a more general Dog object may not always be a Poodle, even if it clearly is to you and me (we know that `frank` and `frankJr` are both Poodles!). Is there any way around this, when we know for certain that assignment would work?

Casting

Java has a special syntax where you can tell the compiler that a specific expression has a specific compile-time type. This is called "casting". With casting, we can tell the compiler to view an expression as a different compile-time type.

Looking back at the code that failed above, since we know that `frank` and `frankJr` are both Poodles, we can cast:

```
Poodle largerPoodle = (Poodle) maxDog(frank, frankJr); // compiles! Right hand side ha
s compile-time type Poodle after casting
```



[Video link](#)

Caution: Casting is a powerful but dangerous tool. Essentially, casting is telling the compiler not to do its type-checking duties - telling it to trust you and act the way you want it to. Here's a possible issue that could arise:

```
Poodle frank = new Poodle("Frank", 5);
Malamute frankSr = new Malamute("Frank Sr.", 100);

Poodle largerPoodle = (Poodle) maxDog(frank, frankSr); // runtime exception!
```

In this case, we compare a Poodle and a Malamute. Without casting, the compiler would normally not allow the call to `maxDog` to compile, as the right hand side compile-time type would be Dog, not Poodle. However, casting allows this code to pass, and when `maxDog` returns the Malamute at runtime, and we try casting a Malamute as a Poodle, we run into a runtime exception - a `ClassCastException`.

Higher Order Functions

Taking a little bit of a detour, we are going to introduce higher order functions. A higher order function is a function that treats other functions as data. For example, take this Python program `do_twice` that takes in another function as input, and applies it to the input `x` twice.

```
def tenX(x):
    return 10*x

def do_twice(f, x):
    return f(f(x))
```

A call to `print(do_twice(tenX, 2))` would apply tenX to 2, and apply tenX again to its result, 20, resulting in 200. How would we do something like this in Java?

In old school Java (Java 7 and earlier), memory boxes (variables) could not contain pointers to functions. What that means is that we could not write a function that has a "Function" type, as there was simply no type for functions.

To get around this we can take advantage of interface inheritance. Let's write an interface that defines any function that takes in an integer and returns an integer - an `IntUnaryFunction`.

```
public interface IntUnaryFunction {
    int apply(int x);
}
```

Now we can write a class which `implements IntUnaryFunction` to represent a concrete function. Let's make a function that takes in an integer and returns 10 times that integer.

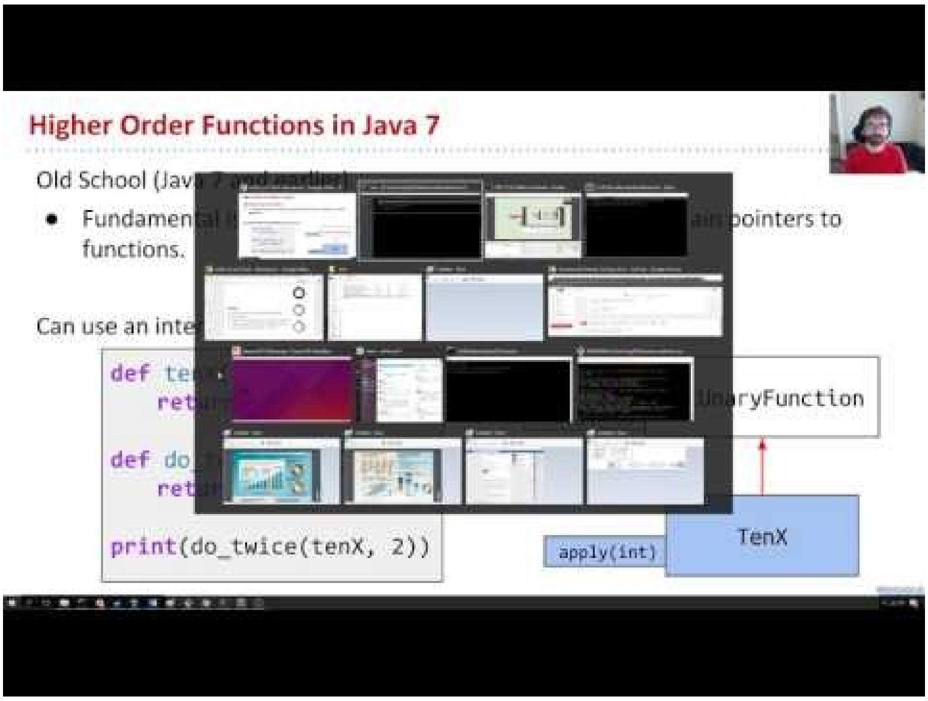
```
public class TenX implements IntUnaryFunction {
    /* Returns ten times the argument. */
    public int apply(int x) {
        return 10 * x;
    }
}
```

At this point, we've written in Java the Python equivalent of the `tenX` function. Let's write `do_twice` now.

```
public static int do_twice(IntUnaryFunction f, int x) {
    return f.apply(f.apply(x));
}
```

A call to `print(do_twice(tenX, 2))` in Java would look like this:

```
System.out.println(do_twice(new TenX(), 2));
```



[Video link](#)

Inheritance Cheatsheet

`VengefulSLList extends SLList` means VengefulSLList "is-an" SLList, and inherits all of SLList's members:

- Variables, methods nested classes
- Not constructors Subclass constructors must invoke superclass constructor first. The `super` keyword can be used to invoke overridden superclass methods and constructors.

Invocation of overridden methods follows two simple rules:

- Compiler plays it safe and only allows us to do things according to the static type.
- For overridden methods (*not overloaded methods*), the actual method invoked is based on the dynamic type of the invoking expression
- Can use casting to overrule compiler type checking.

Subtype Polymorphism

We've seen how inheritance lets us reuse existing code in a superclass while implementing small modifications by overriding a superclass's methods or writing brand new methods in the subclass. Inheritance also makes it possible to design general data structures and methods using *polymorphism*.

Polymorphism, at its core, means 'many forms'. In Java, polymorphism refers to how objects can have many forms or types. In object-oriented programming, polymorphism relates to how an object can be regarded as an instance of its own class, an instance of its superclass, an instance of its superclass's superclass, and so on.

Consider a variable `deque` of static type Deque. A call to `deque.addFirst()` will be determined at the time of execution, depending on the run-time type, or dynamic type, of `deque` when `addFirst` is called. As we saw in the last chapter, Java picks which method to call using dynamic method selection.

Suppose we want to write a python program that prints a string representation of the larger of two objects. There are two approaches to this.

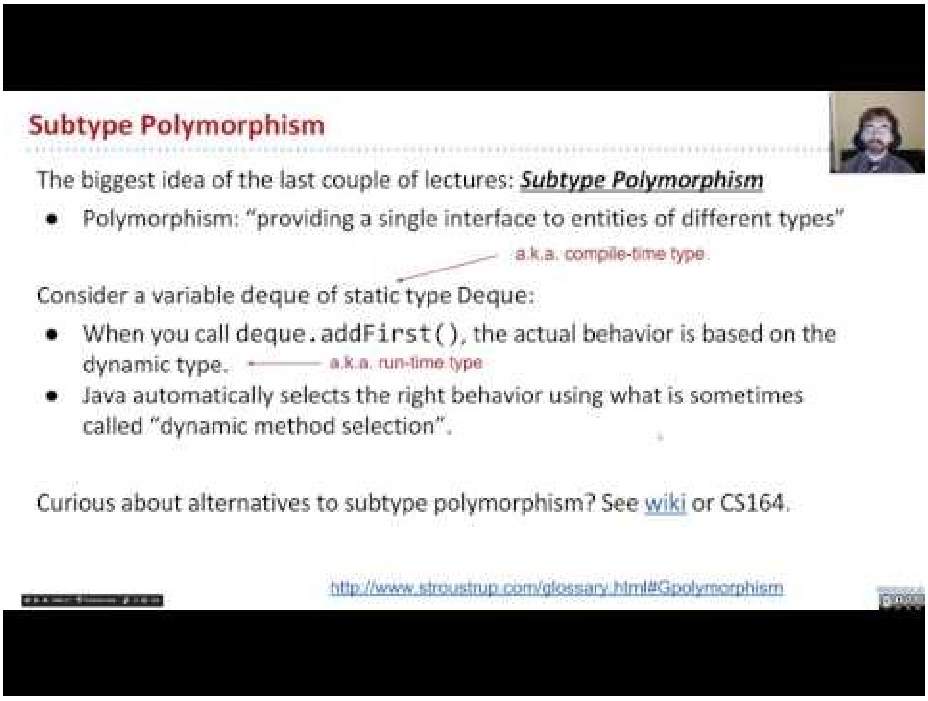
1. Explicit HoF Approach

```
def print_larger(x, y, compare, stringify):
    if compare(x, y):
        return stringify(x)
    return stringify(y)
```

1. Subtype Polymorphism Approach

```
def print_larger(x, y):
    if x.largerThan(y):
        return x.str()
    return y.str()
```

Using the explicit higher order function approach, you have a common way to print out the larger of two objects. In contrast, in the subtype polymorphism approach, the object *itself* makes the choices. The `largerFunction` that is called is dependent on what x and y actually are.



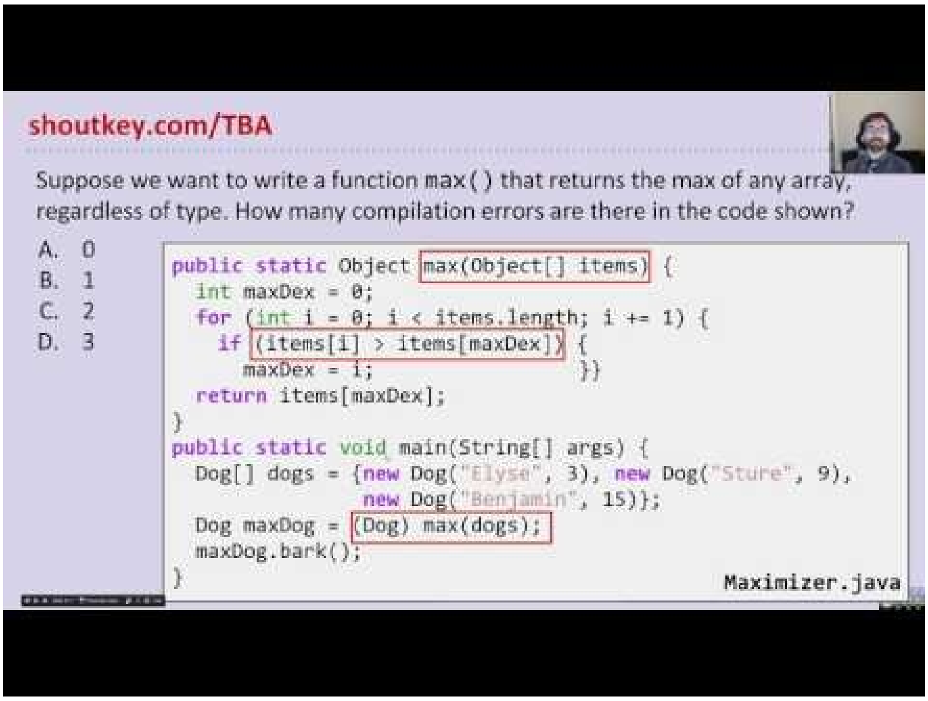
[Video link]

Max Function

Say we want to write a `max` function which takes in any array - regardless of type - and returns the maximum item in the array.

Exercise 4.3.1. Your task is to determine how many compilation errors there are in the code below.

```
public static Object max(Object[] items) {
    int maxDex = 0;
    for (int i = 0; i < items.length; i += 1) {
        if (items[i] > items[maxDex]) {
            maxDex = i;
        }
    }
    return items[maxDex];
}

public static void main(String[] args) {
    Dog[] dogs = {new Dog("Elyse", 3), new Dog("Sture", 9), new Dog("Benjamin", 15)};
    Dog maxDog = (Dog) max(dogs);
    maxDog.bark();
}
```



[Video link]

In the code above, there was only 1 error, found at this line:

```
if (items[i] > items[maxDex]) {
```

The reason why this results in a compilation error is because this line assumes that the `>` operator works with arbitrary Object types, when in fact it does not.

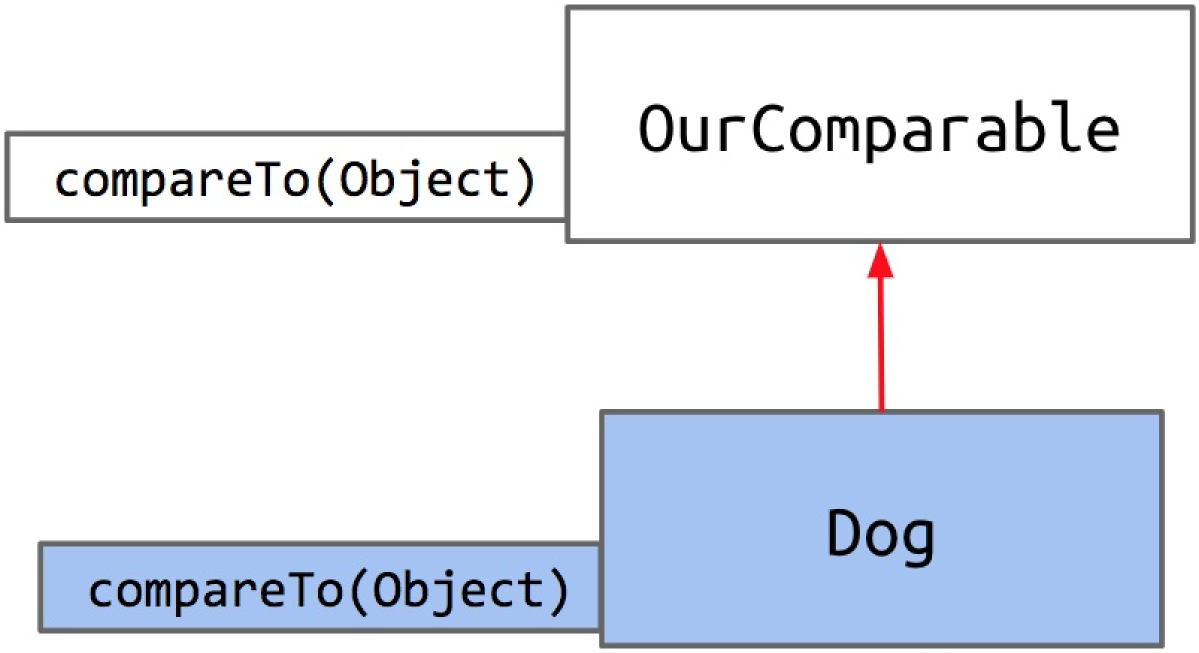
Instead, one thing we could is define a `maxDog` function in the Dog class, and give up on writing a "one true max function" that could take in an array of any arbitrary type. We might define something like this:

```
public static Dog maxDog(Dog[] dogs) {
    if (dogs == null || dogs.length == 0) {
        return null;
    }
    Dog maxDog = dogs[0];
    for (Dog d : dogs) {
        if (d.size > maxDog.size) {
            maxDog = d;
        }
    }
    return maxDog;
}
```

While this would work for now, if we give up on our dream of making a generalized `max` function and let the Dog class define its own `max` function, then we'd have to do the same for any class we define later. We'd need to write a `maxCat` function, a `maxPenguin` function, a `maxWhale` function, etc., resulting in unnecessary repeated work and a lot of redundant code.

The fundamental issue that gives rise to this is that Objects cannot be compared with `>`. This makes sense, as how could Java know whether it should use the String representation of the object, or the size, or another metric, to make the comparison? In Python or C++, the way that the `>` operator works could be redefined to work in different ways when applied to different types. Unfortunately, Java does not have this capability. Instead, we turn to interface inheritance to help us out.

We can create an interface that guarantees that any implementing class, like Dog, contains a comparison method, which we'll call `compareTo`.



Let's write our interface. We'll specify one method `compareTo`.

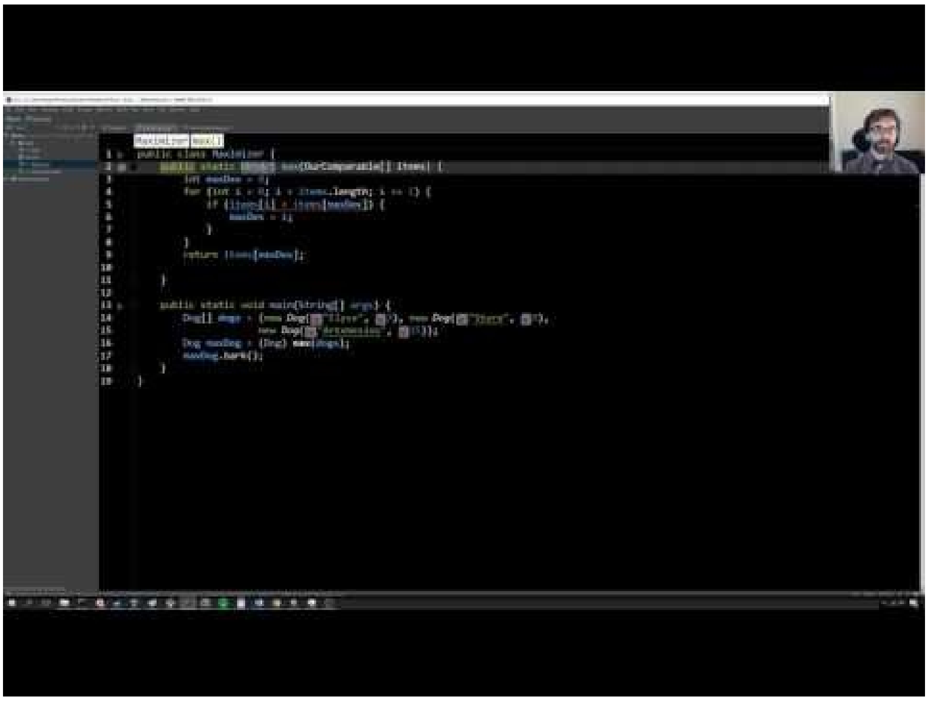
```
public interface OurComparable {
    public int compareTo(Object o);
}
```

We will define its behavior like so:

- Return -1 if `this` < o.
- Return 0 if `this` equals o.
- Return 1 if `this` > o.

Now that we've created the `OurComparable` interface, we can require that our Dog class implements the `compareTo` method. First, we change Dog's class header to include `implements OurComparable`, and then we write the `compareTo` method according to its defined behavior above.

Exercise 4.3.2. Implement the `compareTo` method for the Dog class.



Video link

We use the instance variable `size` to make our comparison.

```
public class Dog implements OurComparable {
    private String name;
    private int size;

    public Dog(String n, int s) {
        name = n;
        size = s;
    }

    public void bark() {
        System.out.println(name + " says: bark");
    }

    public int compareTo(Object o) {
        Dog uddaDog = (Dog) o;
        if (this.size < uddaDog.size) {
            return -1;
        } else if (this.size == uddaDog.size) {
            return 0;
        }
        return 1;
    }
}
```

Notice that since `compareTo` takes in any arbitrary Object o, we have to *cast* the input to a Dog to make our comparison using the `size` instance variable.

Now we can generalize the `max` function we defined in exercise 4.3.1 to, instead of taking in any arbitrary array of objects, takes in `OurComparable` objects - which we know for certain all have the `compareTo` method implemented.

```
public static OurComparable max(OurComparable[] items) {
    int maxDex = 0;
    for (int i = 0; i < items.length; i += 1) {
        int cmp = items[i].compareTo(items[maxDex]);
        if (cmp > 0) {
            maxDex = i;
        }
    }
    return items[maxDex];
}
```

Great! Now our `max` function can take in an array of any `OurComparable` type objects and return the maximum object in the array. Now, this code is admittedly quite long, so we can make it much more succinct by modifying our `compareTo` method's behavior:

- Return negative number if `this` < o.
- Return 0 if `this` equals o.

- Return positive number if `this` > o.

Now, we can just return the difference between the sizes. If my size is 2, and uddaDog's size is 5, `compareTo` would return -3, a negative number indicating that I am smaller.

```
public int compareTo(Object o) {
    Dog uddaDog = (Dog) o;
    return this.size - uddaDog.size;
}
```

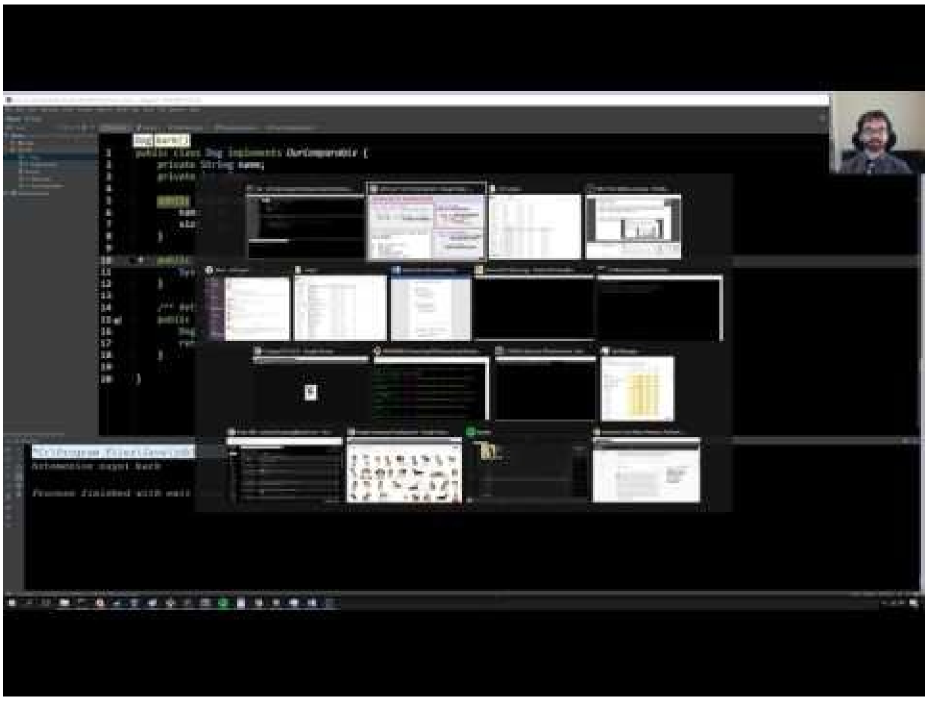
Using inheritance, we were able to generalize our maximization function. What are the benefits to this approach?

- No need for maximization code in every class(i.e. no `Dog.maxDog(Dog[])` function required
- We have code that operates on multiple types (mostly) gracefully

Interfaces Quiz

Exercise 4.3.3. Given the `Dog` class, `DogLauncher` class, `OurComparable` interface, and the `Maximizer` class, if we omit the compareTo() method from the Dog class, which file will fail to compile?

```
public class DogLauncher {
    public static void main(String[] args) {
        ...
        Dog[] dogs = new Dog[]{d1, d2, d3};
        System.out.println(Maximizer.max(dogs));
    }
}

public class Dog implements OurComparable {
    ...
    public int compareTo(Object o) {
        Dog uddaDog = (Dog) o;
        if (this.size < uddaDog.size) {
            return -1;
        } else if (this.size == uddaDog.size) {
            return 0;
        }
        return 1;
    }
    ...
}

public class Maximizer {
    public static OurComparable max(OurComparable[] items) {
        ...
        int cmp = items[i].compareTo(items[maxDex]);
        ...
    }
}
```



Video link

In this case, the `Dog` class fails to compile. By declaring that it `implements OurComparable`, the Dog class makes a claim that it "is-an" OurComparable. As a result, the compiler checks that this claim is actually true, but sees that Dog doesn't implement `compareTo`.

What if we were to omit `implements OurComparable` from the Dog class header? This would cause a compile error in DogLauncher due to this line:

```
System.out.println(Maximizer.max(dogs));
```

If Dog does not implement the OurComparable interface, then trying to pass in an array of Dogs to Maximizer's `max` function wouldn't be approved by the compiler. `max` only accepts an array of OurComparable objects.

Comparables

The `OurComparable` interface that we just built works, but it's not perfect. Here are some issues with it:

- Awkward casting to/from Objects
- We made it up.
    - No existing classes implement OurComparable (e.g. String, etc.)
    - No existing classes use OurComparable (e.g. no built-in max function that uses OurComparable)

The solution? We'll take advantage of an interface that already exists called `Comparable`. `Comparable` is already defined by Java and is used by countless libraries.

`Comparable` looks very similar to the OurComparable interface we made, but with one main difference. Can you spot it?

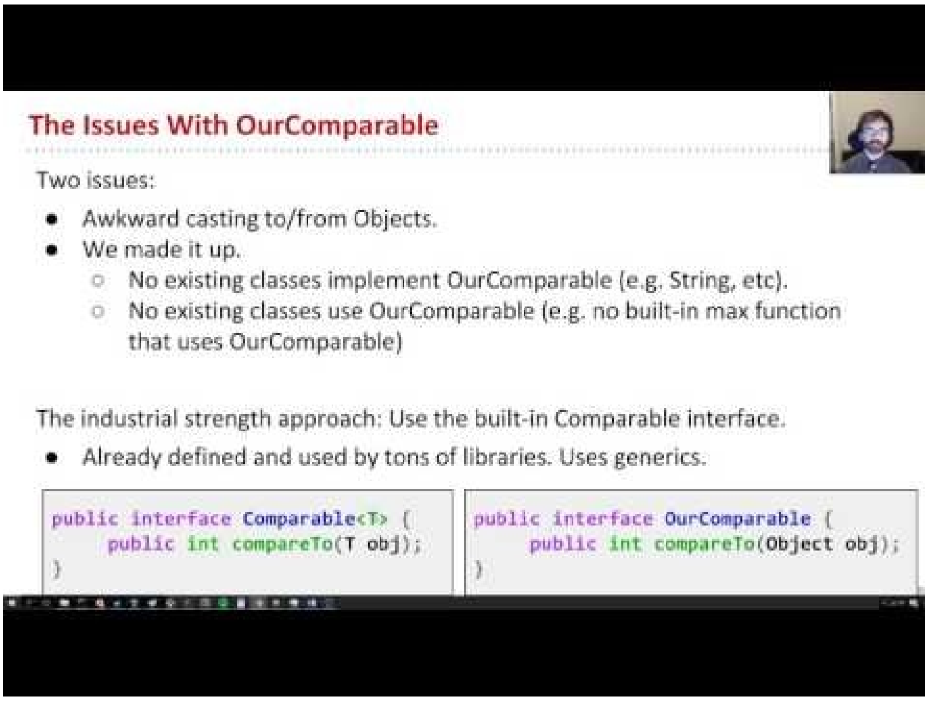
```
public interface Comparable<T> {
    public int compareTo(T obj);
}
```

```
public interface OurComparable {
    public int compareTo(Object obj);
}
```

Notice that `Comparable<T>` means that it takes a generic type. This will help us avoid having to cast an object to a specific type! Now, we will rewrite the Dog class to implement the Comparable interface, being sure to update the generic type `T` to Dog:

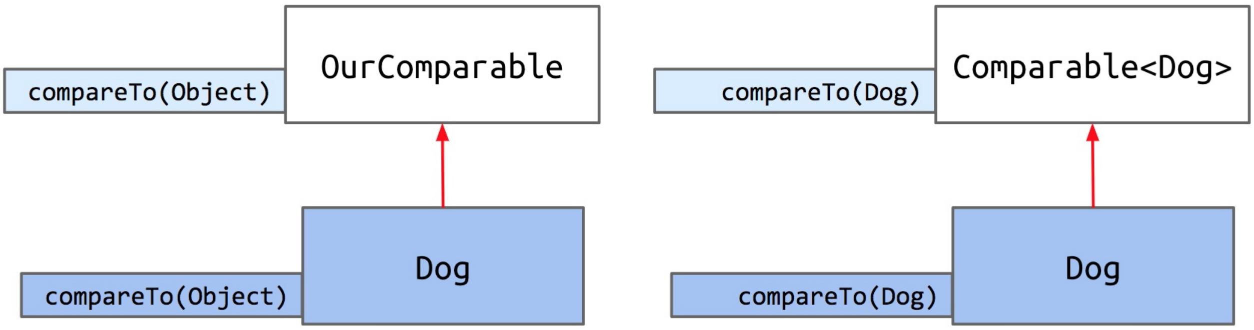
```
public class Dog implements Comparable<Dog> {
    ...
    public int compareTo(Dog uddaDog) {
        return this.size - uddaDog.size;
    }
}
```

Now all that's left is to change each instance of OurComparable in the Maximizer class to Comparable. Watch as the largest Dog says bark:



Video link

Instead of using our personally created interface `OurComparable`, we now use the real, built-in interface, `Comparable`. As a result, we can take advantage of all the libraries that already exist and use `Comparable`.



Comparator

We've just learned about the comparable interface, which imbeds into each Dog the ability to compare itself to another Dog. Now, we will introduce a new interface that looks very similar called `Comparator` .

Let's start off by defining some terminology.

- Natural order - used to refer to the ordering implied in the `compareTo` method of a particular class.

As an example, the natural ordering of Dogs, as we stated previously, is defined according to the value of size. What if we'd like to sort Dogs in a different way than their natural ordering, such as by alphabetical order of their name?

Java's way of doing this is by using `Comparator` 's. Since a comparator is an object, the way we'll use `Comparator` is by writing a nested class inside Dog that implements the `Comparator` interface.

But first, what's inside this interface?

```
public interface Comparator<T> {
    int compare(T o1, T o2);
}
```

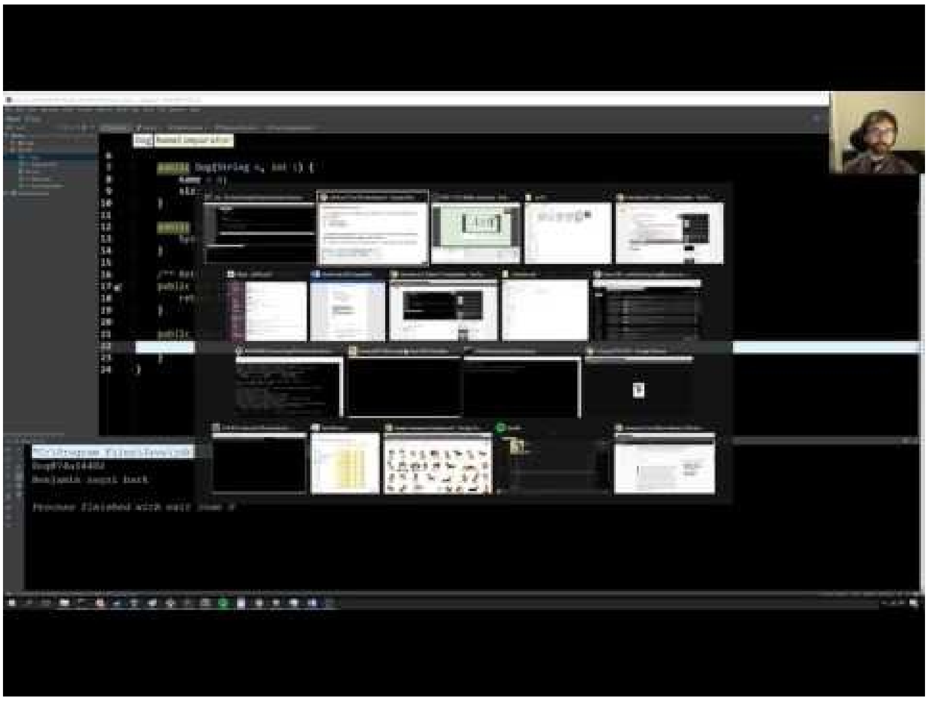
This shows that the `Comparator` interface requires that any implementing class implements the `compare` method. The rule for `compare` is just like `compareTo` :

- Return negative number if o1 < o2.
- Return 0 if o1 equals o2.
- Return positive number if o1 > o2.

Let's give Dog a NameComparator. To do this, we can simply defer to `String` 's already defined `compareTo` method.

```java
import java.util.Comparator;

public class Dog implements Comparable<Dog> {
    ...
    public int compareTo(Dog uddaDog) {
        return this.size - uddaDog.size;
    }

    private static class NameComparator implements Comparator<Dog> {
        public int compare(Dog a, Dog b) {
            return a.name.compareTo(b.name);
        }
    }

    public static Comparator<Dog> getNameComparator() {
        return new NameComparator();
    }
}
```

Note that we've declared NameComparator to be a static class. A minor difference, but we do so because we do not need to instantiate a Dog to get a NameComparator. Let's see how this Comparator works in action.



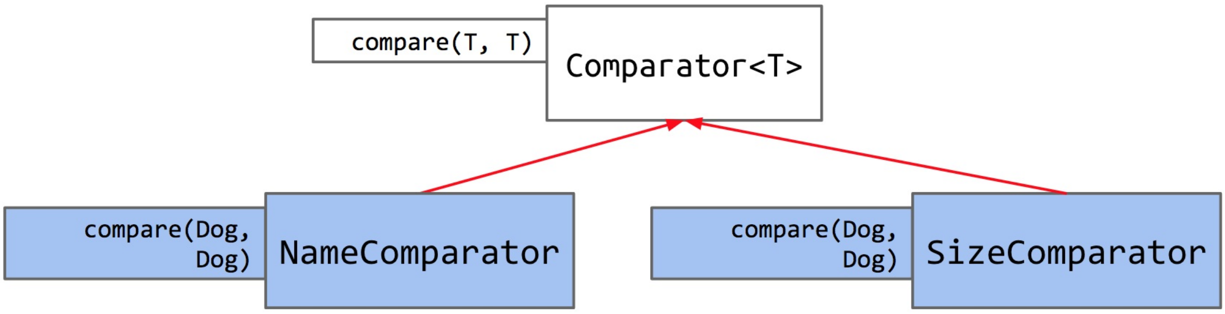
Video link

As you've seen, we can retrieve our NameComparator like so:

```java
Comparator<Dog> nc = Dog.getNameComparator();
```

All in all, we have a Dog class that has a private NameComparator class and a method that returns a NameComparator we can use to compare dogs alphabetically by name.

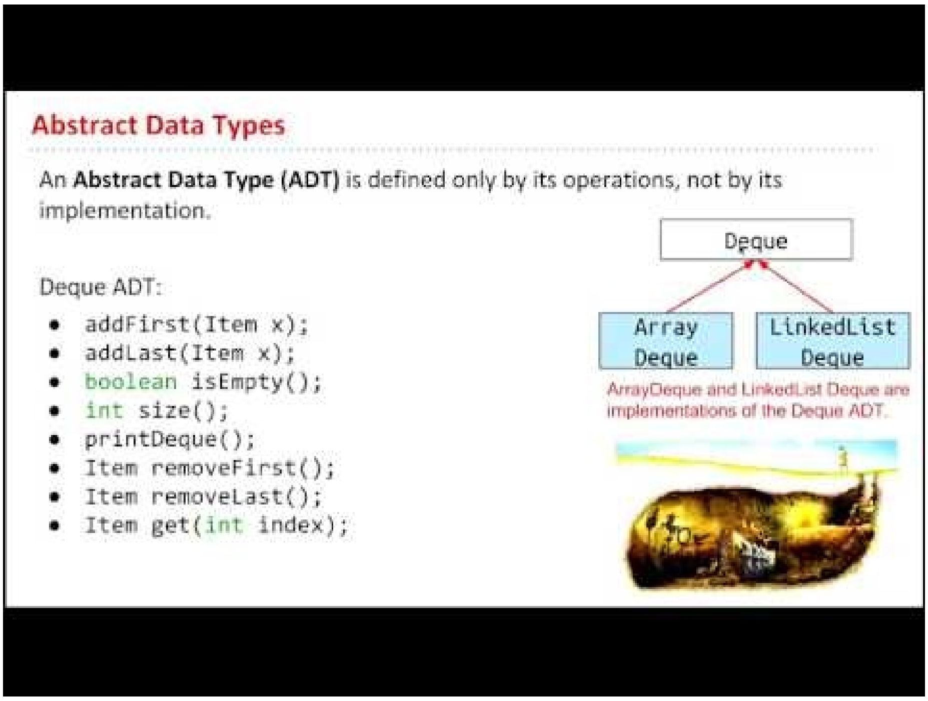
Let's see how everything works in the inheritance hierarchy - we have a Comparator interface that's built-in to Java, which we can implement to define our own Comparators (`NameComparator` , `SizeComparator` , etc.) within Dog.



To summarize, interfaces in Java provide us with the ability to make **callbacks**. Sometimes, a function needs the help of another function that might not have been written yet (e.g. `max` needs `compareTo`). A callback function is the helping function (in the scenario, `compareTo`). In some languages, this is accomplished using explicit function passing; in Java, we wrap the needed function in an interface.

A Comparable says, "I want to compare myself to another object". It is imbedded within the object itself, and it defines the **natural ordering** of a type. A Comparator, on the other hand, is more like a third party machine that compares two objects to each other. Since there's only room for one `compareTo` method, if we want multiple ways to compare, we must turn to Comparator.

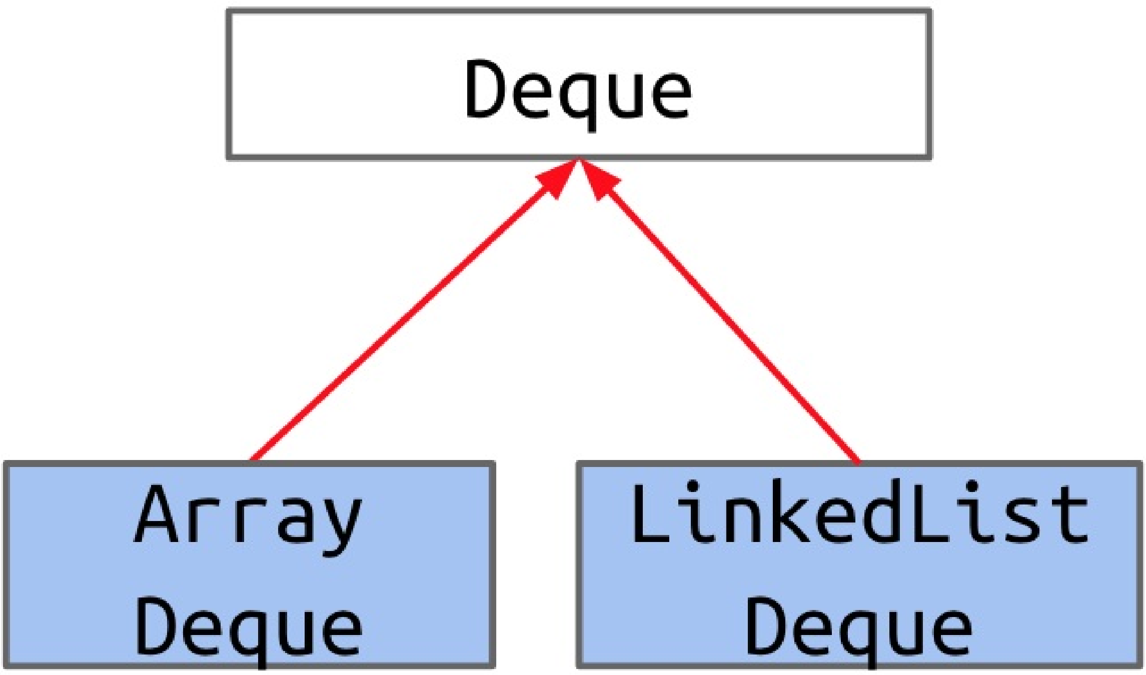
Abstract Data Types (ADTS)



[Video link]

Despite not talking about them explicitly, we have actually seen a few Abstract Data types already in class!

Two examples are List61B and Deque. Let's hone in on Deque.



We have this interface `deque` that both ArrayDeque and LinkedListDeque implement. What is the relationship between Deque and its implementing classes? Well, deque simply provides a list of methods (behaviors):

```
public void addFirst(T item);
public void addLast(T item);
public boolean isEmpty();
public int size();
public void printDeque();
public T removeFirst();
public T removeLast();
public T get(int index);
```

These methods are actually **implemented** by ArrayDeque and LinkedListDeque.

In Java, Deque is called an interface. Conceptually, we call deque an **Abstract data type**. Deque only comes with behaviors, not any concrete ways to exhibit those behaviors. In this way, it is abstract.

Java Libraries

Java has certain built-in Abstract data types that you can use. These are packaged in Java Libraries.

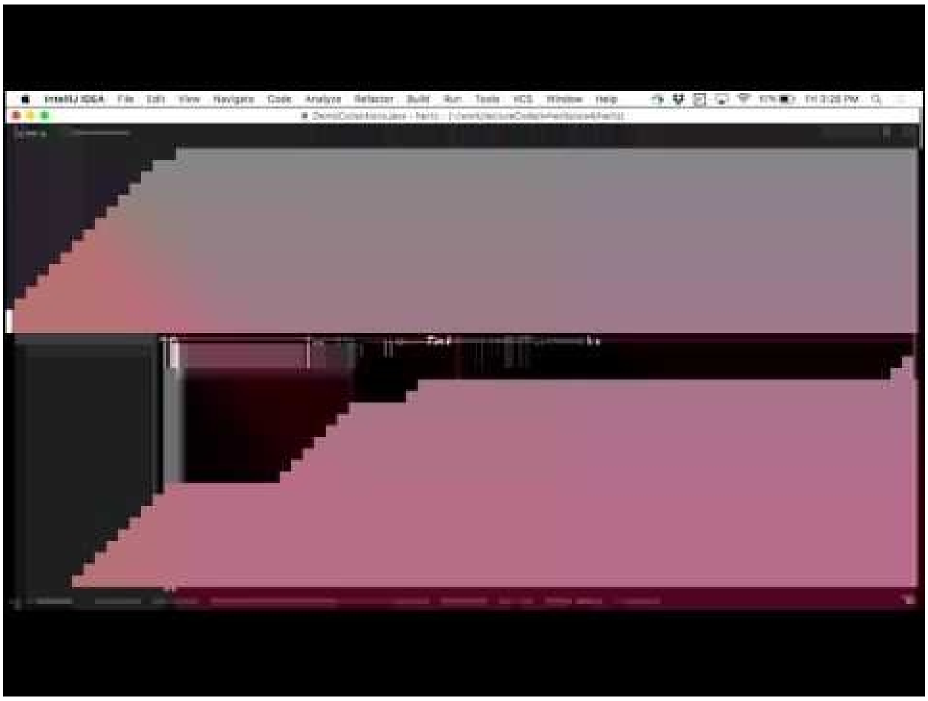
The three most important ADTs come in the java.util library:

- List: an ordered collection of items
  - A popular implementation is the ArrayList
- Set: an unordered collection of strictly unique items (no repeats)
  - A popular implementation is the HashSet
- Map: a collection of key/value pairs. You access the value via the key.
  - A popular implementation is the HashMap

Finish the exercises below by using the above three ADT's. Reading the documentations linked above will help immensely.

Exercise 4.4.1 Write a method `getWords` that takes in a `String inputFileName` and puts every word from the input file into a list. Recall how we read words from a file in proj0. **Hint*: use `In`

Exercise 4.4.2 Write a method `countUniqueWords` that takes in a `List<String>` and counts how many **unique** words there are in the file.

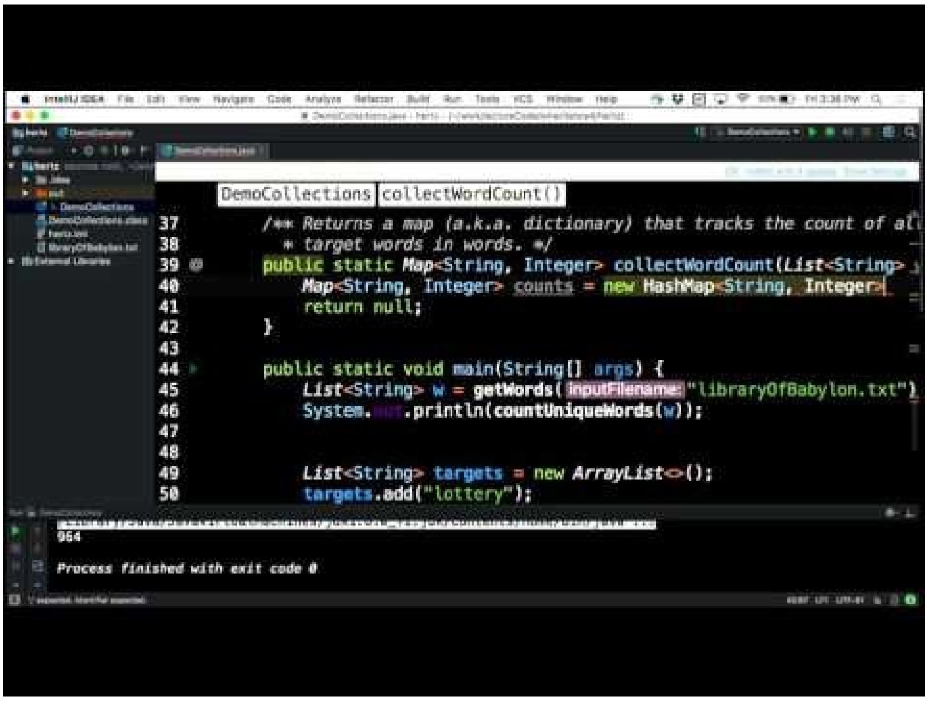


Video link

We used a list for the first exercise and set for the second.

```
public static List<String> getWords(String inputFileName) {
    List<String> lst = new ArrayList<String>();
    In in = new In();
    while (!in.isEmpty()) {
        lst.add(in.readString()); //optionally, define a cleanString() method that cleans the string first.
    }
    return lst;
}

public static int countUniqueWords(List<String> words) {
    Set<String> ss = new HashSet<>();
    for (String s : words) {
           ss.add(s);
    }
    return ss.size();
}
```

Exercise 4.4.3 Write a method `collectWordCount` that takes in a `List<String> targets` and a `List<String> words` and finds the number of times each target word appears in the word list.

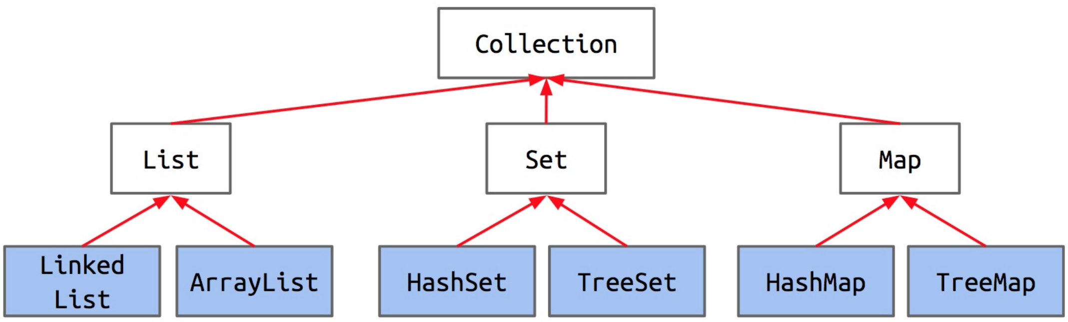


Video link

```
public static Map<String, Integer> collectWordCount(List<String> words) {
    Map<String, Integer> counts = new HashMap<String, Integer>();
    for (String t: target) {
        counts.put(s, 0);
    }
    for (String s: words) {
        if (counts.containsKey(s)) {
            counts.put(word, counts.get(s)+1);
        }
    }
    return counts;
}
```

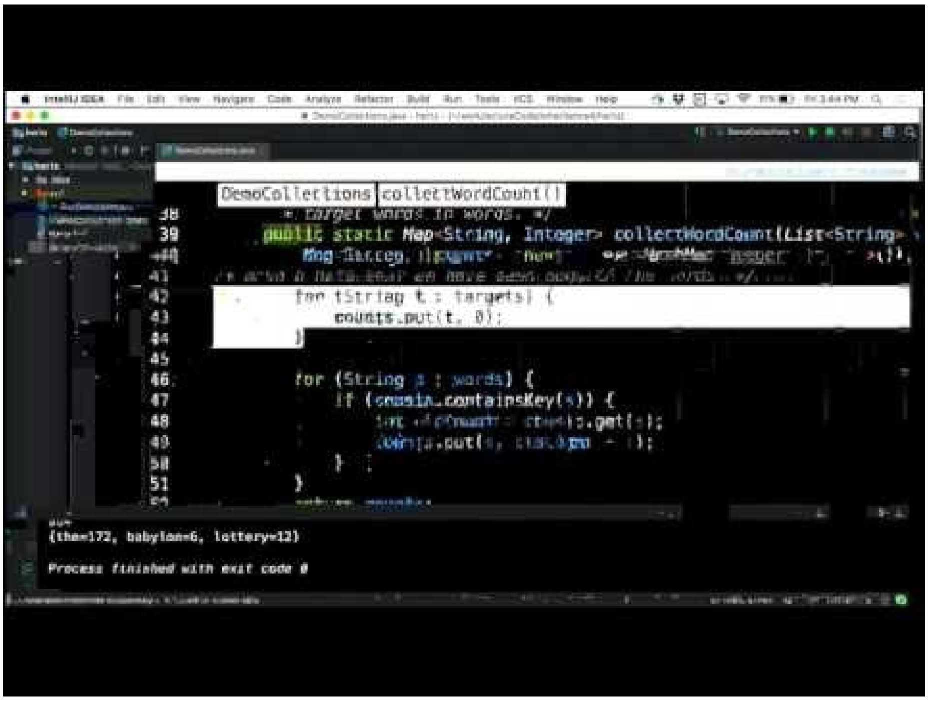
We used a map because it makes an association between two things. In our case, we need an association between word and number.

These three ADT's all extend from the Collection Interface. The collection interface is super vague. Java says collections "represent a group of objects, known as its elements".



In the diagram above, the white boxes are interfaces. The blue boxes are concrete classes.

Java vs Python



Video link

Java is pretty verbose. The java code below looks a lot more cumbersome than the corresponding python code.

```
public static Map<String, Integer>
        collectWordCount(List<String> words, List<String> targets) {
    Map<String, Integer> wordCounts = new HashMap<>();
    for (String s : targets) {
        wordCounts.put(s, 0);
    }
    for (String s : words) {
        if (wordCounts.containsKey(s)) {
            int oldCount = wordCounts.get(s);
            wordCounts.put(s, oldCount + 1);
        }
    }
    return wordCounts;
}
```

```python
def find_word_count(words, targets):
    word_counts = {}
    for s in targets:
        word_counts[s] = 0

    for s in words:
        if s in word_counts:
            word_counts[s] += 1

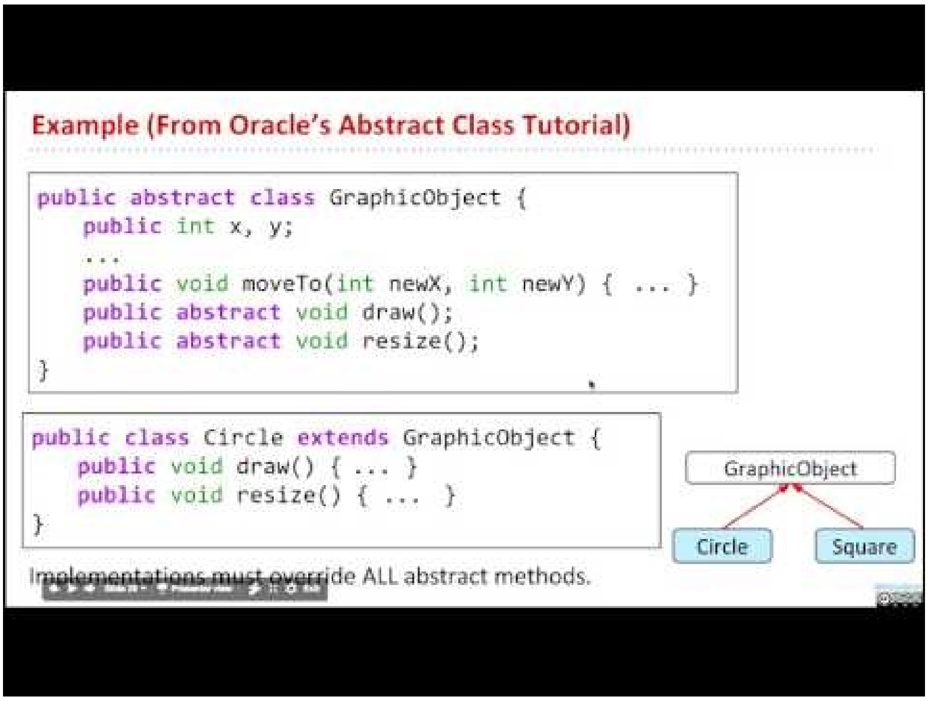
    return word_counts
```

But, Java has its upsides too! It gives you a lot of choices and freedom. For example, python only has one dictionary type which is declared using curly brackets {}. With Java, if you want to use an ADT like a Map, you can choose what kind of map you want: a Hashmap? a Treemap? etc.

We like Java in 61B! Here are some reasons why:

- Arguably, takes less time to write programs, due to features like:
    - Static types (provides type checking and helps guide programmer).
    - Bias towards interface inheritance leading to cleaner subtype polymorphism.
    - Access control modifiers make abstraction barriers more solid.
- More efficient code, due to features like:
    - Ability to have more control over engineering tradeoffs.
    - Single valued arrays lead to better performance.
- Basic data structures more closely resemble underlying hardware:
    - Would be weird to do ArrayDeque in Python, since there is no need for array resizing. However, in hardware (see 61C), variable length arrays don't exist.

Abstract classes



[Video link](#)

We've seen interfaces that can do a lot of cool things! They allow you to take advantage of interface inheritance and implementation inheritance. As a refresher, these are the qualities of interfaces:

- All methods must be public.
- All variables must be public static final.
- Cannot be instantiated
- All methods are by default abstract unless specified to be `default`
- Can implement more than one interface per class

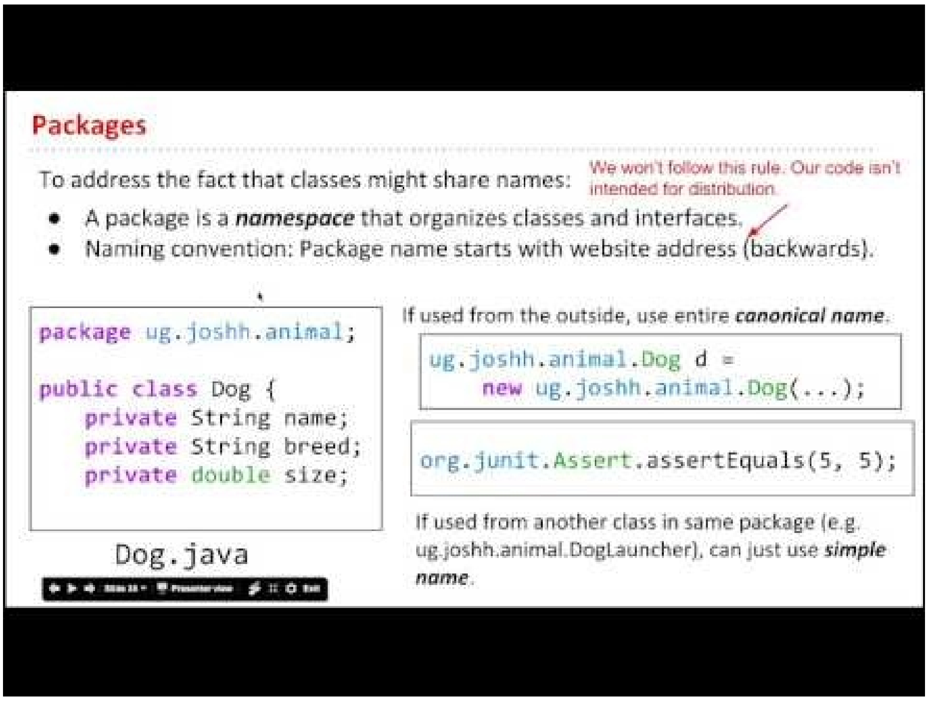
We will now introduce a new class that lies somewhere in between interfaces and concrete classes: the abstract class. Below are the characteristics of abstract classes:

- Methods can be public or private
- Can have any types of variables
- Cannot be instantiated
- Methods are by default concrete unless specified to be `abstract`
- Can only implement one per class

Basically, abstract classes can do everything interfaces can do and more.

When in doubt, try to use interfaces in order to reduce complexity.

Packages



[Video link]

Package names give **give a canonical name for everything**. Canonical means a *unique representation* for a thing.

Why? Well, in Java, we could have multiple classes with the same name. We need a way to differentiate between these different classes. In industry, this differentiation happens by appending the class to a website address (backwards) like below:

```
But... this means we have to type out that entire name every time we want to instantia
te something of that class.

```ug.joshh.animal.Dog d = new ug.joshh.animal.Dog()
```

This is annoying. We can remedy this by importing the package.

`import ug.joshh.animal`

Now we can use dogs as we please.

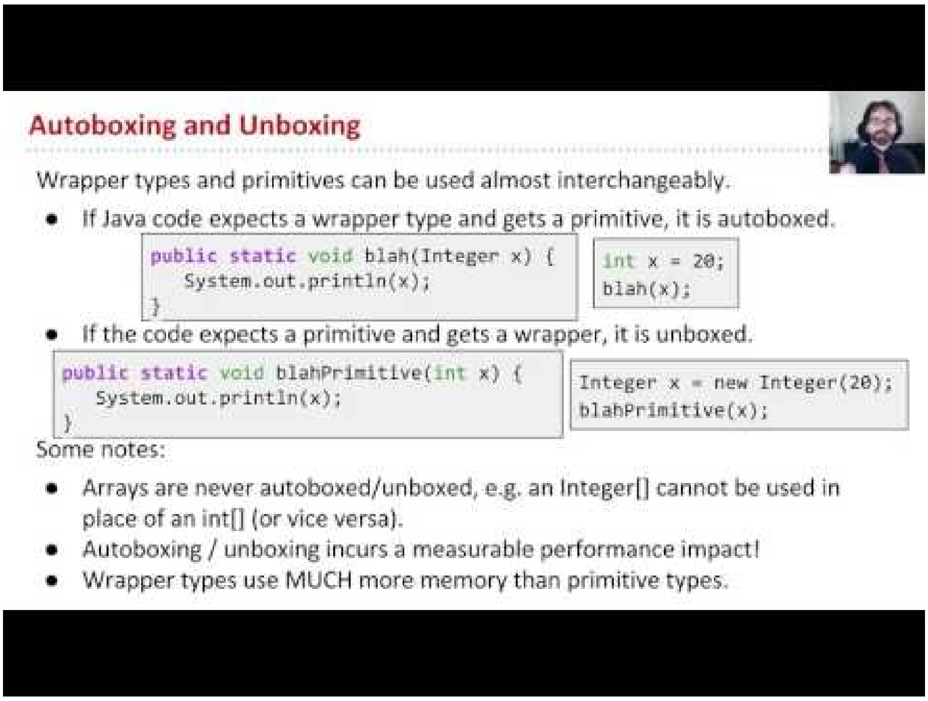
This is just a brief preview of packages. We will get to more of this during later weeks of the course.
```

Industrial Strength Syntax

In the previous parts of this book, we've talked about various data structures and the way that Java supports their implementation. In this chapter, we'll discuss a variety of supplementary topics that are used in industrial strength implementations of Java programs.

This is not meant to be a comprehensive guide to Java, but rather a highlight of features that are likely to be useful to you while working on this course.

Automatic Conversions



Video link

Autoboxing and Unboxing

As we saw in the previous chapter, we can define classes which have generic type variables using the `<>` syntax, e.g. `LinkedListDeque<Item>` and `ArrayDeque<Item>`. When we want to instantiate an object whose class uses generics, we have to substitute the generic with a concrete class, i.e. specify what type of items are going to go into that class.

Recall that Java has 8 primitive types -- all other types are reference types. One particular feature of Java is that we cannot provide a primitive type as an actual type argument for generics, e.g. `ArrayDeque<int>` is a syntax error. Instead, we use `ArrayDeque<Integer>`. For each primitive type, we use the corresponding reference type as shown in the table below. These reference types are called "wrapper classes".

Primitive	Class
byte	Byte
short	Short
int	Integer
long	Long
float	Float
double	Doube
boolean	Boolean
char	Character

Naively, we'd assume that this would result in having to manually convert between primitive and reference types when using a generic data structure. For example, we might imagine having to do the following:

```
public class BasicArrayList {
    public static void main(String[] args) {
      ArrayList<Integer> L = new ArrayList<Integer>();
      L.add(new Integer(5));
      L.add(new Integer(6));

      /* Use the Integer.valueOf method to convert to int */
      int first = L.get(0).valueOf();
    }
}
```

Writing code like above can be a bit annoying. Luckily, Java can implicitly convert between primitive and wrapper types, so the code below works just fine:

```
public class BasicArrayList {
    public static void main(String[] args) {
      ArrayList<Integer> L = new ArrayList<Integer>();
      L.add(5);
      L.add(6);
      int first = L.get(0);
    }
}
```

The reason this works is that Java will automatically "box" and "unbox" values between a primitive type and its corresponding reference type. That is, if Java expects a wrapper type, like Integer, and you provide a primitive type, like int, it will "autobox" the integer. For example, if we have the function:

```
public static void blah(Integer x) {
    System.out.println(x);
}
```

And we call it using:

```
int x = 20;
blah(x);
```

Then Java implicitly creates a new `Integer` with value 20, resulting in a call to equivalent to calling `blah(new Integer(20))`. This process is known as autoboxing.

Likewise, if Java expected a primitive:

```
public static void blahPrimitive(int x) {
    System.out.println(x);
}
```

but you give it a value of the corresponding wrapper type:

```
Integer x = new Integer(20);
blahPrimitive(x);
```

It will automatically unbox the integer, equivalent to calling the `Integer` class's `valueOf` method.

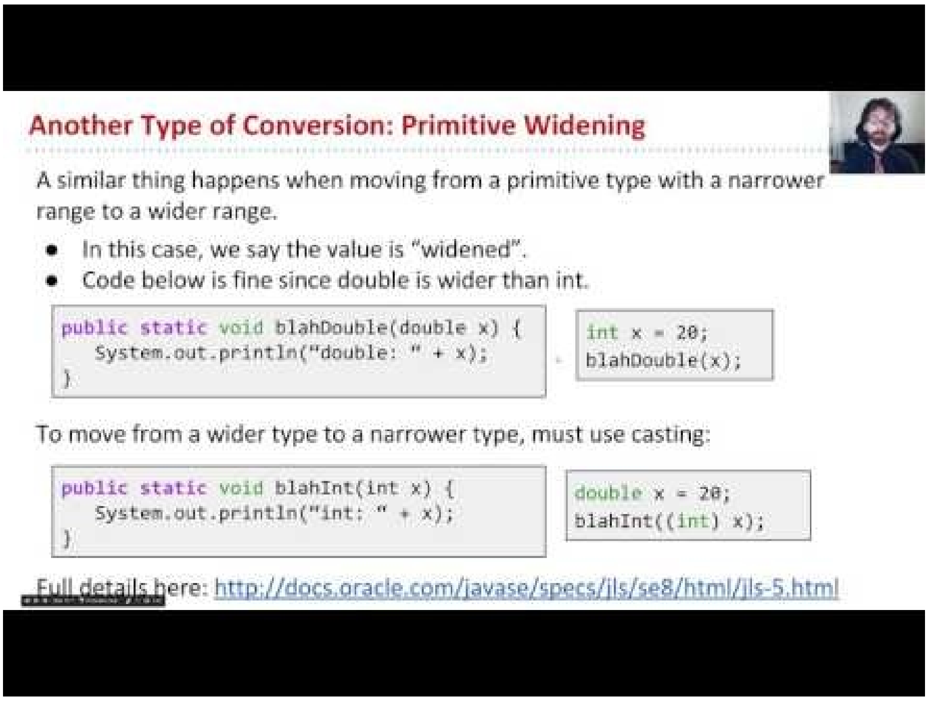
Caveats

There are a few things to keep in mind when it comes to autoboxing and unboxing:

- Arrays are never autoboxes or auto-unboxed, e.g. if you have an array of integers `int[] x`, and try to put its address into a variable of type `Integer[]`, the compiler will not allow your program to compile.
- Autoboxing and unboxing also has a measurable performance impact. That is, code that relies on autoboxing and unboxing will be slower than code that eschews such automatic conversions.

- Additionally, wrapper types use much more memory than primitive types. On most modern comptuers, not only must your code hold a 64 bit reference to the object, but every object also requires 64 bits of overhead used to store things like the dynamic type of the object.
    - For more on memory usage, see this link or this link.

Widening



Video link

Similar to the autoboxing/unboxing process, Java will also automatically widen a primitive if needed. Specifically, if a program expects a primitive of type T2 and is given a variable of type T1, and type T2 can take on a wider range of values than T1, the the variable will be implicitly cast to type T2.

For example, doubles in Java are wider than ints. If we have the function shown below:

```
public static void blahDouble(double x) {
    System.out.println("double: " + x);
}
```

We can call it with an int argument:

```
int x = 20;
blahDouble(x);
```

The effect is the same as if we'd done `blahDouble((double) x)`. Thanks Java!

If you want to go from a wider type to a narrower type, you must manually cast. For example, if you have the method below:

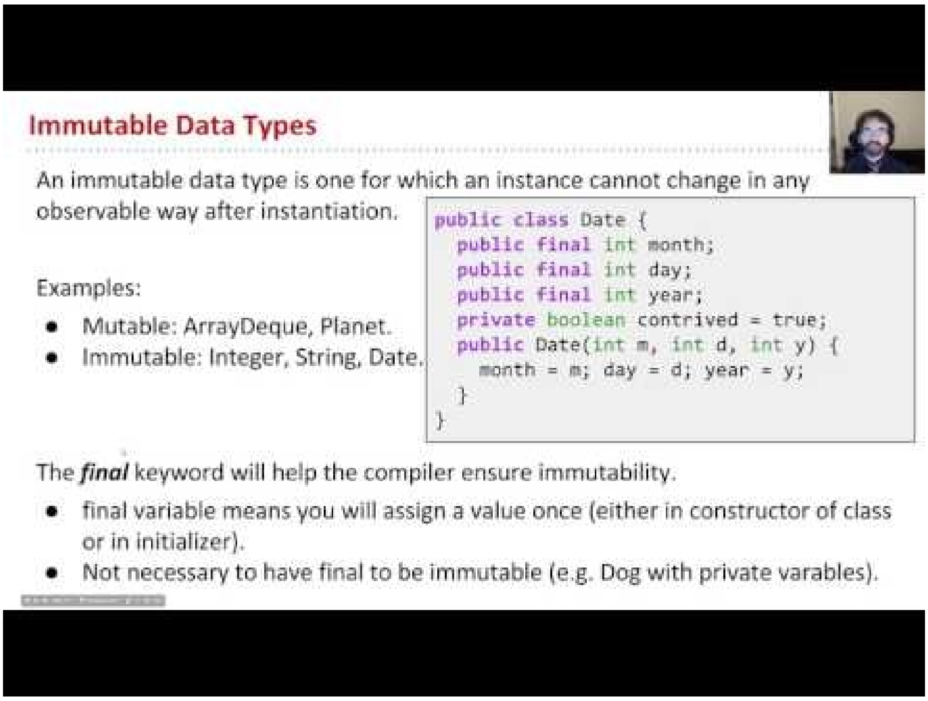
```
public static void blahInt(int x) {
    System.out.println("int: " + x);
}
```

Then we'd need to use a cast if we want to call this method using a double value, e.g.

```
double x = 20;
blahInt((int) x);
```

For more details on widening, including a full description of what types are wider than others, see the official Java documentation.

Immutability



Video link

The notion of immutability is one of the things you might never have known existed, but that can greatly simplify your life once you realize it's a thing (sort of like the realization you get as an adult that nobody *really* knows what they're doing, at least when they first start doing something new).

An immutable data type is a data type whose instances cannot change in any observable way after instantiation.

For example, `String` objects in Java are immutable. No matter what, if you have an instance of `String`, you can call any method on that `String`, but it will remain completely unchanged. This means that when `String` objects are concatenated, neither of the original Strings are modified -- instead, a totally new `String` object is returned.

Mutable datatypes include objects like `ArrayDeque` and `Planet`. We can add or remove items from an `ArrayDeque`, which are observable changes. Similarly, the velocity and position of a `Planet` may change over time.

Any data type with non-private variables is mutable, unless those variables are declared `final` (this is not the only condition for mutability -- there are many other ways of defining a data type so that it is mutable). This is because an outside method can change the value of non-private variables, leading to observable change.

The `final` keyword is a keyword for variables that prevents the variable from being changed after its first assignment. For example, consider the `Date` class below:

```
public class Date {
    public final int month;
    public final int day;
    public final int year;
    private boolean contrived = true;
    public Date(int m, int d, int y) {
        month = m; day = d; year = y;
    }
}
```

This class is immutable. After instantiating a `Date`, there is no way to change the value of any of its properties.

Advantages of immutable data types:

- Prevents bugs and makes debugging easier because properties cannot change ever
- You can count on objects to have a certain behavior/trait

Disadvantages:

- You need to create a new object in order to change a property

Caveats:

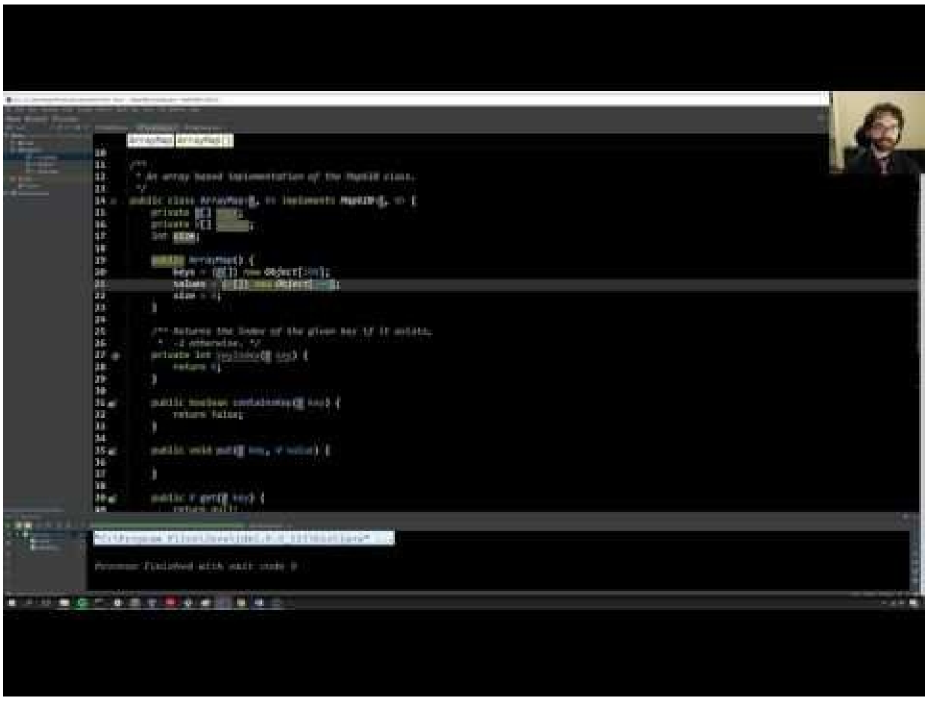
- Declaring a reference as **final** does not make the object that reference is pointing to immutable! For example, consider the following code snippet:

  ```
  public final ArrayDeque<String>() deque = new ArrayDeque<String>();
  ```

  The `deque` variable is final and can never be reassigned, but the array deque object its pointing to can change! ArrayDeques are always mutable!

- Using the Reflection API, it is possible to make changes even to private variables! Our notion of immutability assumes that we're not using any of the special capabilities of this library.

Creating Another Generic Class



Video link

Now that we've created generic lists, such as `DLLists` and `ALists`, lets move on to a different data type: maps. Maps let you associate keys with values, for example, the statement "Josh's score on the exam is 0" could be stored in a Map that associates students to their exam scores. A map is the Java equivalent of a Python dictionary.

We're going to be creating the `ArrayMap` class, which implements the `Map61B` Interface, a restricted version of Java's built-in `Map` interface. `ArrayMap` will have the following methods:

```
put(key, value): Associate key with value.
containsKey(key): Checks if map contains the key.
get(key): Returns value, assuming key exists.
keys(): Returns a list of all keys.
size(): Returns number of keys.
```

For this exercise, we will ignore resizing. One thing to note about the `Map61B` interface (and the Java `Map` interface in general) is that each key can only have one value at a time. If Josh is mapped to 0, and then we say "Oh wait, there was a mistake! Josh actually got 100 on the exam," we erase the value 0 that's Josh maps to and replace it with 100.

Feel free to try building an `ArrayMap` on your own, but for reference, the full implementation is below.

```java
package Map61B;

import java.util.List;
import java.util.ArrayList;

/***
 * An array-based implementation of Map61B.
 ***/
public class ArrayMap<K, V> implements Map61B<K, V> {

    private K[] keys;
    private V[] values;
    int size;

    public ArrayMap() {
        keys = (K[]) new Object[100];
        values = (V[]) new Object[100];
        size = 0;
    }

    /**
    * Returns the index of the key, if it exists. Otherwise returns -1.
    **/
    private int keyIndex(K key) {
        for (int i = 0; i < size; i++) {
            if (keys[i].equals(key)) {
            return i;
        }
        return -1;
    }

    public boolean containsKey(K key) {
        int index = keyIndex(key);
        return index > -1;
    }

    public void put(K key, V value) {
        int index = keyIndex(key);
        if (index == -1) {
            keys[size] = key;
            values[size] = value;
            size += 1;
        } else {
            values[index] = value;
        }
    }

    public V get(K key) {
        int index = keyIndex(key);
        return values[index];
    }
```

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

    public List<K> keys() {
        List<K> keyList = new ArrayList<>();
        for (int i = 0; i < keys.length; i++) {
            keyList.add(keys[i]);
        }
        return keyList;
    }
}
```

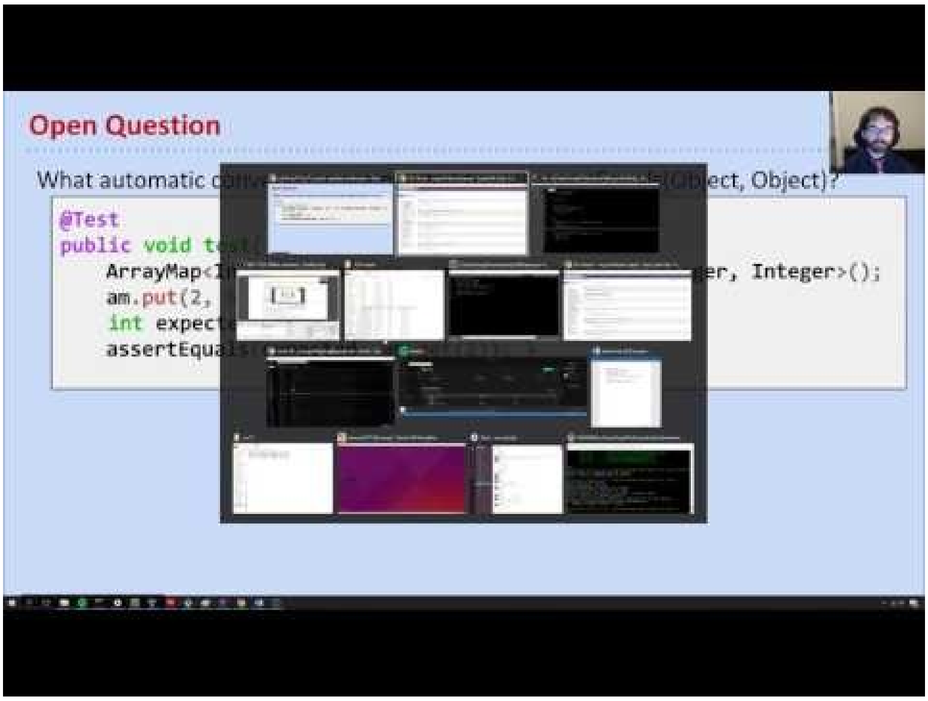
Note: the decision to name the generics `K` and `V` is arbitrary (but meant to be intuitive). We could have just as well replaced these generics with `Potato` and `Sauce`, or any other name. However, it's quite common to see generics in Java represented as a single uppercase letter, in this course and elsewhere.

There were a few interesting things here; looking at the top of the code, we stated `package Map61B;`. We will go over this a bit later, but for now just know that it means we are putting our `ArrayMap` class within a folder called Map61B. Additionally, we import `List` and `ArrayList` from `java.utils`.

Exercise 5.2.1: In our current implementation of ArrayMap, there is a bug. Can you figure out what it is?

Answer: In the `keys` method, the for loop should be iterating until `i == size`, not `keys.length`.

ArrayMap and Autoboxing Puzzle



Video link

If we write a test as shown below:

```
@Test
public void test() {
    ArrayMap<Integer, Integer> am = new ArrayMap<Integer, Integer>();
    am.put(2, 5);
    int expected = 5;
    assertEquals(expected, am.get(2));
}
```

You will find that we get a compile-time error!

```
$ javac ArrayMapTest.java
ArrayMapTest.java:11: error: reference to assertEquals is ambiguous
    assertEquals(expected, am.get(2));
    ^
    both method assertEquals(long, long) in Assert and method assertEquals(Object, Obj
ect) in Assert match
```

We get this error because JUnit's `assertEquals` method is overloaded, eg. `assertEquals(int expected, int actual)`, `assertEquals(Object expected, Object actual)`, etc. Thus, Java is unsure which method to call for `assertEquals(expected, am.get(2))`, which requires one argument to be autoboxed/unboxed.

Excercise 5.2.2 What would we need to do in order to call `assertEquals(long, long)`? A.) Widen `expected` to a `long` B.) Autobox `expected` to a `Long` C.) Unbox `am.get(2)` D.) Widen the unboxed `am.get(2)` to long

Answer A, C, and D all work.

Excercise 5.2.3 How would we make it work with `assertEquals(Object, Object)` ?

Answer Autobox `expected` to an `Integer` because `Integers` are `Objects` .

Excercise 5.2.4 How do we make the code compile with casting?

Answer Cast `expected` to `Integer` .

Generic Methods

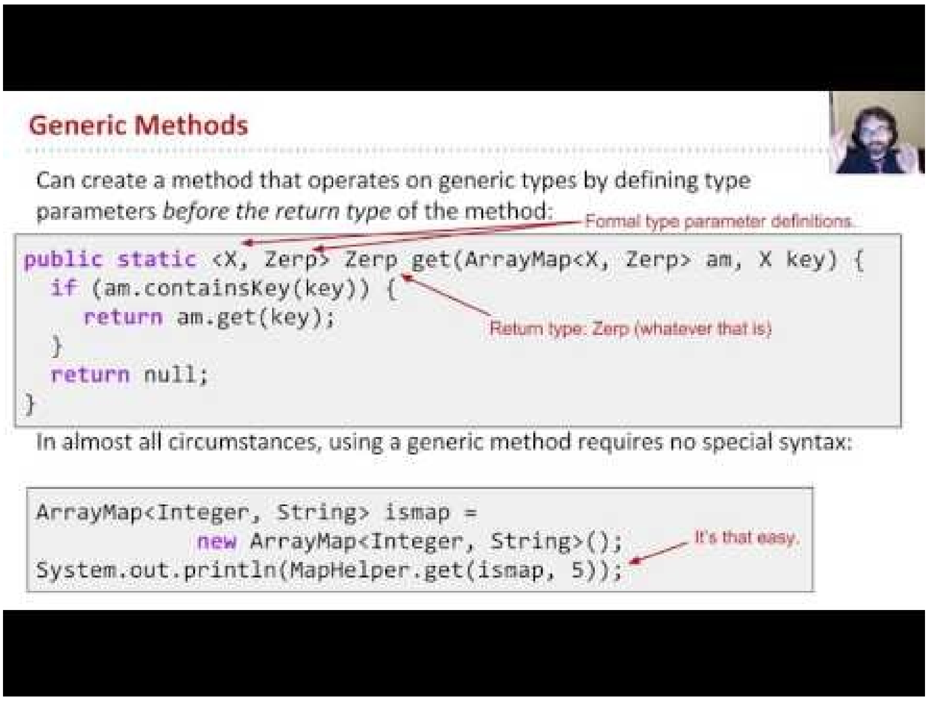
The goal for the next section is to create a class `MapHelper` which will have two methods:

- `get(Map61B, key)` : Returns the value corresponding to the given key in the map if it exists, otherwise null.
    - This is useful because `ArrayMap` currently has a bug where the get method throws an ArrayIndexOutOfBoundsException if we try to get a key that doesn't exist in the `ArrayMap` .
- `maxKey(Map61B)` : Returns the maximum of all keys in the given `ArrayMap` . Works only if keys can be compared.

Implementing get

`get` is a static method that takes in a Map61B instance and a key and returns the value that corresponds to the key if it exists, otherwise returns null.

Excercise 5.2.5 Try writing this method yourself!



Video link

As you see in the video, we could write a very limited method by declare the parameters as String and Integer like so:

```
public static Integer get(Map61B<String, Integer> map, String key) {
    ...
}
```

We are restricting this method to only take in `Map61B<String, Integer>`, which is not what we want! We want it to take any kind of `Map61B`, no matter what the actual types for the generics are. However, the following method header produces a compilation error:

```
public static V get(Map61B<K, V> map, String key) {
    ...
}
```

This is because with generics defined in class headers, Java waits for the user to instantiate an object of the class in order to know what actual types each generic will be. However, here we'd like a generic specific to this method. Moreover, we do not care what actual types `K` and `V` take on in our `Map61B` argument -- the important part is that whatever `V` is, an object of type `V` is returned.

Thus we see the need for generic methods. To declare a method as generic, the formal type parameters must be specified before the return type:

```
public static <K,V> V get(Map61B<K,V> map, K key) {
    if map.containsKey(key) {
        return map.get(key);
    }
    return null;
}
```

Here's an example of how to call it:

```
ArrayMap<Integer, String> isMap = new ArrayMap<Integer, String>();
System.out.println(mapHelper.get(isMap, 5));
```

You don't need any explicit declaration of what type you are inserting. Java can infer that isMap is an `ArrayMap` from `Integers` to `Strings`.

Implementing maxKey

Exercise 5.2.6 Try writing this method yourself!

Here's something that looks OK, but isn't quite correct:

```
public static <K, V> K maxKey(Map61B<K, V> map) {
    List<K> keylist = map.keys();
    K largest = map.get(0);
    for (K k: keylist) {
        if (k > largest) {
            largest = k;
        }
    }
    return largest;
}
```

Exercise 5.2.7 Can you spot what's wrong with this method?

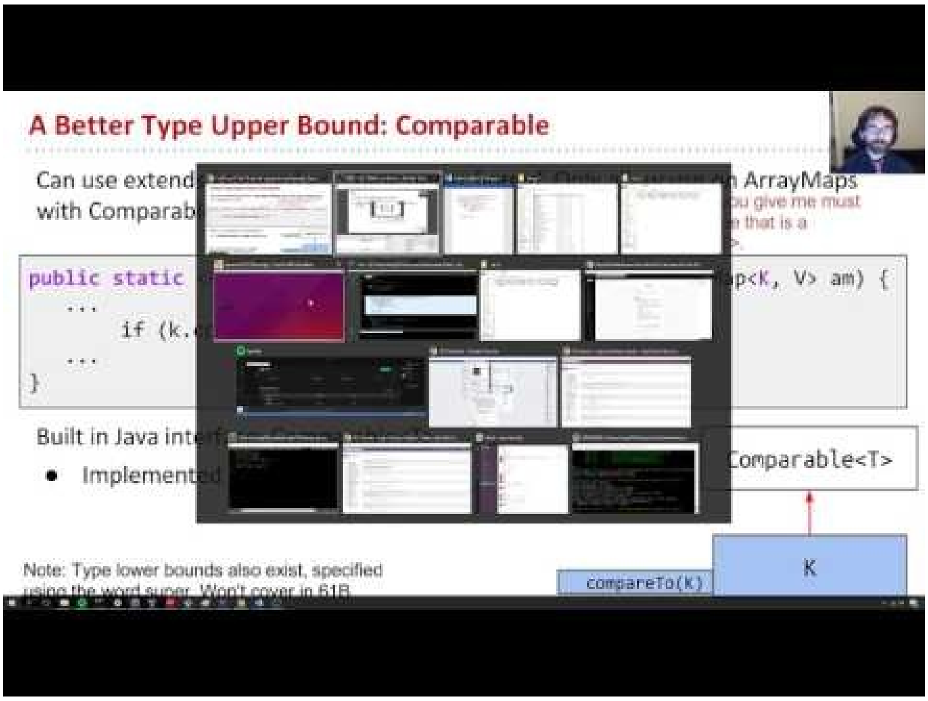
Answer: The `>` operator can't be used to compare `K` objects. This only works on primitives and `map` may not hold primitives

We will rewrite this method as such:

```
public static <K, V> K maxKey(Map61B<K, V> map) {
    List<K> keylist = map.keys();
    K largest = map.get(0);
    for (K k: keylist) {
        if (k.compareTo(largest)) {
            largest = k;
        }
    }
    return largest;
}
```

Exercise 5.2.8 This is also wrong, why?

Answer Not all objects have a `compareTo` method!



[Video link]

We will introduce a little more syntax for generic methods in the header of the function.

```
public static <K extends Comparable<K>, V> K maxKey(Map61B<K, V> map) {...}
```

The `K extends Comparable<K>` means keys must implement the comparable interface and can be compared to other K's. We need to include the `<K>` after `Comparable` because `Comparable` itself is a generic interface! Therefore, we must specify what kind of comparable we want. In this case, we want to compare K's with K's.

Type upper bounds

You might be wondering, why does it "extend" comparable and not "implement"? Comparable is an interface after all.

Well, it turns out, "extends" in this context has a different meaning than in the polymorphism context.

When we say that the Dog class extends the Animal class, we are saying that Dogs can do anything that animals can do and more! We are **giving** Dog the abilities of an animal. When we say that K extends Comparable, we are simply stating a fact. We aren't **giving** K the abilities of a Comparable, we are just saying that K **must be** Comparable. This different use of `extends` is called type upper bounding. Confusing? That's okay, it *is* confusing. Just remember, in the context of inheritance, the `extends` keyword is active in giving the subclass the abilities of the superclass. You can think of it as a fairy Godmother: she sees your needs and helps you out with some of her fairy magic. On the other hand, in the context of generics, `extends` simply states a fact: You must be a subclass of whatever you're extending. **When used with generics (like in generic method headers), `extends` imposes a constraint rather than grants new abilities.** It's akin to a fortune teller, who just tells you something without doing much about it.

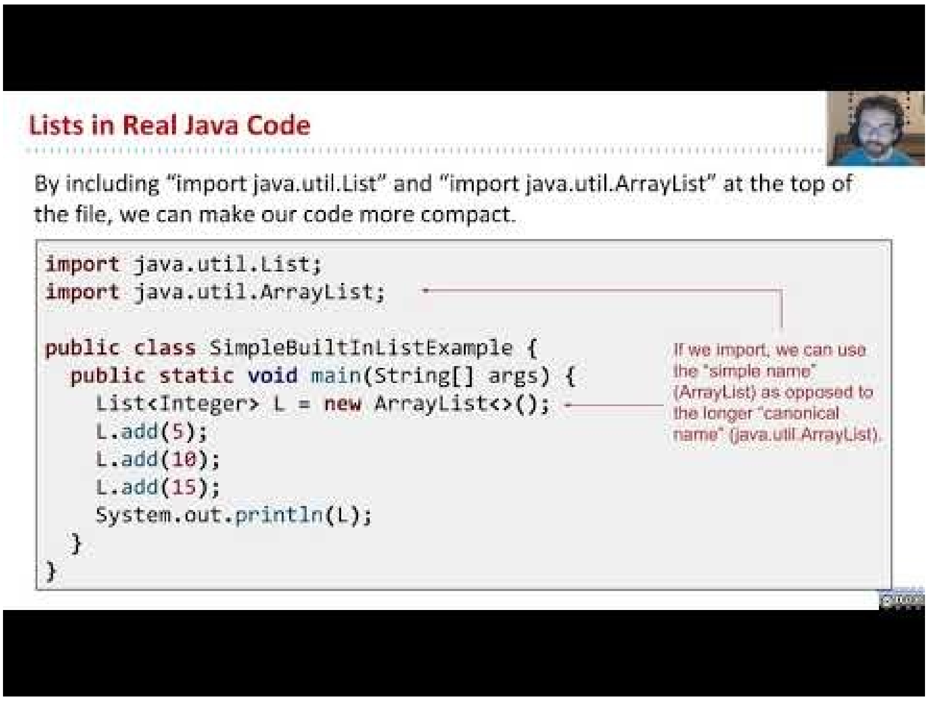
Summary

We've seen four new features of Java that make generics more powerful:

- Autoboxing and auto-unboxing of primitive wrapper types.
- Promotion/widening between primitive types.
- Specification of generic types for methods (before return type).
- Type upper bounds in generic methods (e.g. `K extends Comparable<K>`).

Lists, Sets, ArraySet

In this section we will learn about how to use Java's built in `List` and `Set` data structures as well as build our own `ArraySet`.



Video link

In this course, we've already built two kinds of lists: `AList` and `SLList`. We also built an interface `List61B` to enforce specific list methods `AList` and `SLList` had to implement. You can find the code at the following links:

- List61B
- AList
- SLList

This is how we might use `List61B` type:

```
List61B<Integer> L = new AList<>();
L.addLast(5);
L.addLast(10);
L.addLast(15);
L.print();
```

Lists in Real Java Code

We built a list from scratch, but Java provides a built-in `List` interface and several implementations, e.g. `ArrayList`. Remember, since `List` is an interface we can't instatiate it! We must instatiate one of its implementations.

To access this, we can use the full name ('canonical name') of classes, interfaces:

```
java.util.List<Integer> L = new java.util.ArrayList<>();
```

However this is a bit verbose. In a similar way to how we import `JUnit`, we can import java libraries:

```
import java.util.List;
import java.util.ArrayList;

public class Example {
    public static void main(String[] args) {
        List<Integer> L = new ArrayList<>();
        L.add(5);
        L.add(10);
        System.out.println(L);
    }
}
```

Sets

Sets are a collection of unique elements - you can only have one copy of each element. There is also no sense of order.

Java

Java has the `Set` interface along with implementations, e.g. `HashSet`. Remember to import them if you don't want to use the full name!

```
import java.util.Set;
import java.util.HashSet;
```

Example use:

```
Set<String> s = new HashSet<>();
s.add("Tokyo");
s.add("Lagos");
System.out.println(S.contains("Tokyo")); // true
```

Python

In python, we simply call `set()`. To check for `contains` we don't use a method but the keyword `in`.

```
s = set()
s.add("Tokyo")
s.add("Lagos")
print("Tokyo" in s) // True
```

ArraySet

Our goal is to make our own set, `ArraySet`, with the following methods:

- `add(value)`: add the value to the set if not already present
- `contains(value)`: check to see if ArraySet contains the key
- `size()`: return number of values

If you would like to try it yourself, find 'Do It Yourself' starter code here. In the lecture clip below, Professor Hug goes develops the solution:

Here is our code as of now:

```
import java.util.Iterator;

public class ArraySet<T> implements Iterable<T> {
    private T[] items;
    private int size; // the next item to be added will be at position size

    public ArraySet() {
        items = (T[]) new Object[100];
        size = 0;
    }

    /* Returns true if this map contains a mapping for the specified key.
     */
    public boolean contains(T x) {
        for (int i = 0; i < size; i += 1) {
            if (items[i].equals(x)) {
                return true;
            }
        }
        return false;
    }

    /* Associates the specified value with the specified key in this map. */
    public void add(T x) {
        if (contains(x)) {
            return;
        }
        items[size] = x;
        size += 1;
    }

    /* Returns the number of key-value mappings in this map. */
    public int size() {
        return size;
    }
}
```

Throwing Exceptions

Video link

Our `ArraySet` implementation from the previous section has a small error. When we add `null` to our ArraySet, we get a NullPointerException.

The probelm lies in the `contains` method where we check `items[i].equals(x)`. If the value at `items[i]` is null, then we are calling `null.equals(x)` -> NullPointerException.

Exceptions cause normal flow of control to stop. We can in fact choose to throw our own exceptions. In python you may have seen this with the `raise` keyword. In Java, Exceptions are objects and we throw exceptions using the following format:

`throw new ExceptionObject(parameter1, ...)`

Let's throw an exception when a user tries to add null to our `ArraySet`. We'll throw an `IllegalArgumentException` which takes in one parameter (a `String` message).

Our updated `add` method:

```
/* Associates the specified value with the specified key in this map.
   Throws an IllegalArgumentException if the key is null. */
public void add(T x) {
    if (x == null) {
        throw new IllegalArgumentException("can't add null");
    }
    if (contains(x)) {
        return;
    }
    items[size] = x;
    size += 1;
}
```

We get an Exception either way - why is this better?

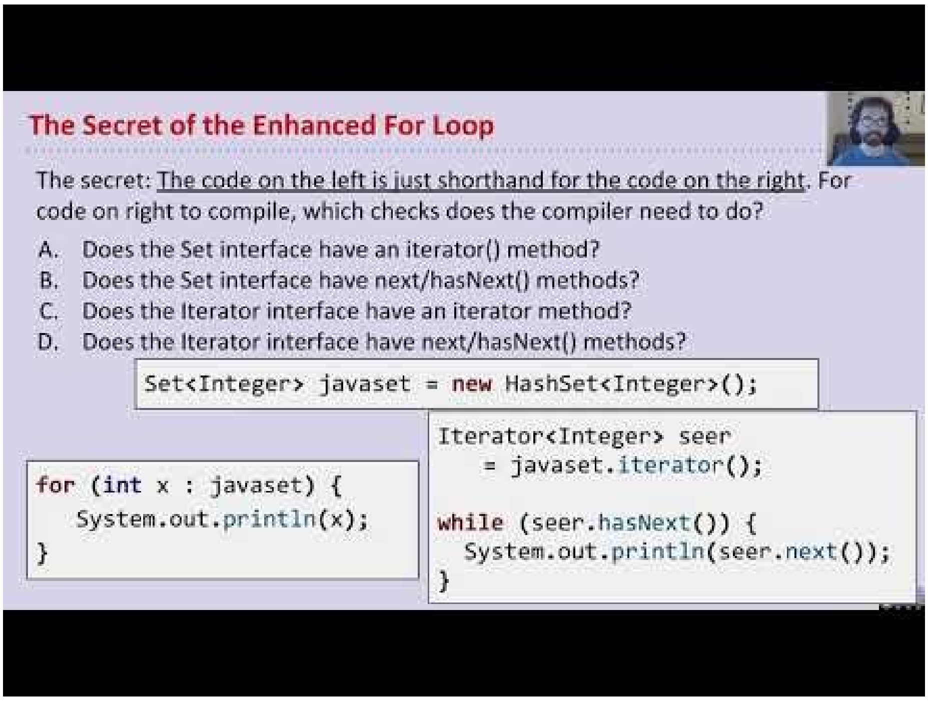
1. We have control of our code: we consciously decide at what point to stop the flow of our program
2. More useful Exception type and helpful error message for those using our code

However, it would be better if the program doesn't crash at all. There are different things we could do in this case. Here are some below:

Approach 1: Don't add `null` to the array if it is passed into `add` **Approach 2**: Change the `contains` method to account for the case if `items[i] == null`.

Whatever you decide, it is important that users know what to expect. That is why documentation (such as comments about your methods) is very important.

Iteration



Video link

We can use a clean enhanced for loop with Java's `HashSet`

```
Set<String> s = new HashSet<>();
s.add("Tokyo");
s.add("Lagos");
for (String city : s) {
    System.out.println(city);
}
```

However, if we try to do the same with our `ArraySet`, we get an error. How can we enable this functionality?

Enhanced For Loop

Let's first understand what is happening when we use an enhanced for loop. We can "translate" an enhanced for loop into an ugly, manual approach.

```
Set<String> s = new HashSet<>();
...
for (String city : s) {
    ...
}
```

The above code translates to:

```
Set<String> s = new HashSet<>();
...
Iterator<String> seer = s.iterator();
while (seer.hasNext()) {
    String city = seer.next();
    ...
}
```

Let's strip away the magic so we can build our own classes that support this.

The key here is an object called an *iterator*.

For our example, in List.java we might define an `iterator()` method that returns an iterator object.

```
public Iterator<E> iterator();
```

Now, we can use that object to loop through all the entries in our list:

```
List<Integer> friends = new ArrayList<Integer>();
...
Iterator<Integer> seer = friends.iterator();

while (seer.hasNext()) {
System.out.println(seer.next());
}
```

This code behaves identically to the foreach loop version above.

There are three key methods in our iterator approach:

First, we get a new iterator object with `Iterator<Integer> seer = friends.iterator();`

Next, we loop through the list with our while loop. We check that there are still items left with `seer.hasNext()`, which will return true if there are unseen items remaining, and false if all items have been processed.

Last, `seer.next()` does two things at once. It returns the next element of the list, and here we print it out. It also advances the iterator by one item. In this way, the iterator will only inspect each item once.

Implementing Iterators

In this section, we are going to talk about how to build a class to support iteration.

Let's start by thinking about what the compiler need to know in order to successfully compile the following iterator example:

```
List<Integer> friends = new ArrayList<Integer>();
Iterator<Integer> seer = friends.iterator();

while(seer.hasNext()) {
    System.out.println(seer.next());
}
```

We can look at the static types of each object that calls a relevant method. `friends` is a List, on which `iterator()` is called, so we must ask:

- Does the List interface have an iterator() method?

`seer` is an Iterator, on which `hasNext()` and `next()` are called, so we must ask:

- Does the Iterator interface have next/hasNext() methods?

So how do we implement these requirements?

The List interface extends the Iterable interface, inheriting the abstract iterator() method. (Actually, List extends Collection which extends Iterable, but it's easier to codethink of this way to start.)

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

```
public interface List<T> extends Iterable<T>{
    ...
}
```

Next, the compiler checks that Iterators have `hasNext()` and `next()`. The Iterator interface specifies these abstract methods explicitly:

```
public interface Iterator<T> {
    boolean hasNext();
    T next();
}
```

What if someone calls `next` when `hasNext` returns false?

```
This behavior is undefined. However, a common convention is to throw a `NoSuchElementE
xception`. See [Discussion 5](https://sp19.datastructur.es/materials/discussion/disc05
sol.pdf) for examples.
```

Will `hasNext` always be called before `next`?

```
Not necessarily. This is sometimes the case when someone using the iterator knows exac
tly how many elements are in the sequence. Thus, we can't rely on the user calling `ha
sNext` before `next`. However, you can always call `hasNext` from within your `next` f
unction.
```

Specific classes will implement their own iteration behaviors for the interface methods. Let's look at an example. (Note: if you want to build this up from the start, follow along with the live coding in the video.)

We are going to add iteration through keys to our ArrayMap class. First, we write a new class called ArraySetIterator, nested inside of ArraySet:

```
private class ArraySetIterator implements Iterator<T> {
    private int wizPos;

    public ArraySetIterator() {
        wizPos = 0;
    }

    public boolean hasNext() {
        return wizPos < size;
    }

    public T next() {
        T returnItem = items[wizPos];
        wizPos += 1;
        return returnItem;
    }
}
```

This ArraySetIterator implements a `hasNext()` method, and a `next()` method, using a `wizPos` position as an index to keep track of its position in the array. For a different data structure, we might implement these two methods differently.

Thought Excercise: How would you design `hasNext()` and `next()` for a linked list?

Now that we have the appropriate methods, we could use a ArraySetIterator to iterate through an ArrayMap:

```
ArraySet<Integer> aset = new ArraySet<>();
aset.add(5);
aset.add(23);
aset.add(42);

Iterator<Integer> iter = aset.iterator();

while(iter.hasNext()) {
    System.out.println(iter.next());
}
```

We still want to be able to support the enhanced for loop, though, to make our calls cleaner. So, we need to make ArrayMap implement the Iterable interface. The essential method of the Iterable interface is `iterator()` , which returns an Iterator object for that class. All we have to do is return an instance of our `ArraySetIterator` that we just wrote!

```
public Iterator<T> iterator() {
    return new ArraySetIterator();
}
```

Now we can use enhanced for loops with our `ArrraySet` !

```
ArraySet<Integer> aset = new ArraySet<>();
...
for (int i : aset) {
    System.out.println(i);
}
```

Here we've seen **Iterable**, the interface that makes a class able to be iterated on, and requires the method `iterator()` , which returns an Iterator object. And we've seen **Iterator**, the interface that defines the object with methods to actually do that iteration. You can think of an Iterator as a machine that you put onto an iterable that facilitates the iteration. Any iterable is the object on which the iterator is performing.

With these two components, you can make fancy for loops for your classes!

`ArraySet` code with iteration support is below:

```
import java.util.Iterator;

public class ArraySet<T> implements Iterable<T> {
    private T[] items;
    private int size; // the next item to be added will be at position size

    public ArraySet() {
        items = (T[]) new Object[100];
```

```
        size = 0;
    }

    /* Returns true if this map contains a mapping for the specified key.
     */
    public boolean contains(T x) {
        for (int i = 0; i < size; i += 1) {
            if (items[i].equals(x)) {
                return true;
            }
        }
        return false;
    }

    /* Associates the specified value with the specified key in this map.
       Throws an IllegalArgumentException if the key is null. */
    public void add(T x) {
        if (x == null) {
            throw new IllegalArgumentException("can't add null");
        }
        if (contains(x)) {
            return;
        }
        items[size] = x;
        size += 1;
    }

    /* Returns the number of key-value mappings in this map. */
    public int size() {
        return size;
    }

    /** returns an iterator (a.k.a. seer) into ME */
    public Iterator<T> iterator() {
        return new ArraySetIterator();
    }

    private class ArraySetIterator implements Iterator<T> {
        private int wizPos;

        public ArraySetIterator() {
            wizPos = 0;
        }

        public boolean hasNext() {
            return wizPos < size;
        }

        public T next() {
            T returnItem = items[wizPos];
            wizPos += 1;
            return returnItem;
        }
```

```
    }

    public static void main(String[] args) {
        ArraySet<Integer> aset = new ArraySet<>();
        aset.add(5);
        aset.add(23);
        aset.add(42);

        //iteration
        for (int i : aset) {
            System.out.println(i);
        }
    }

}
```

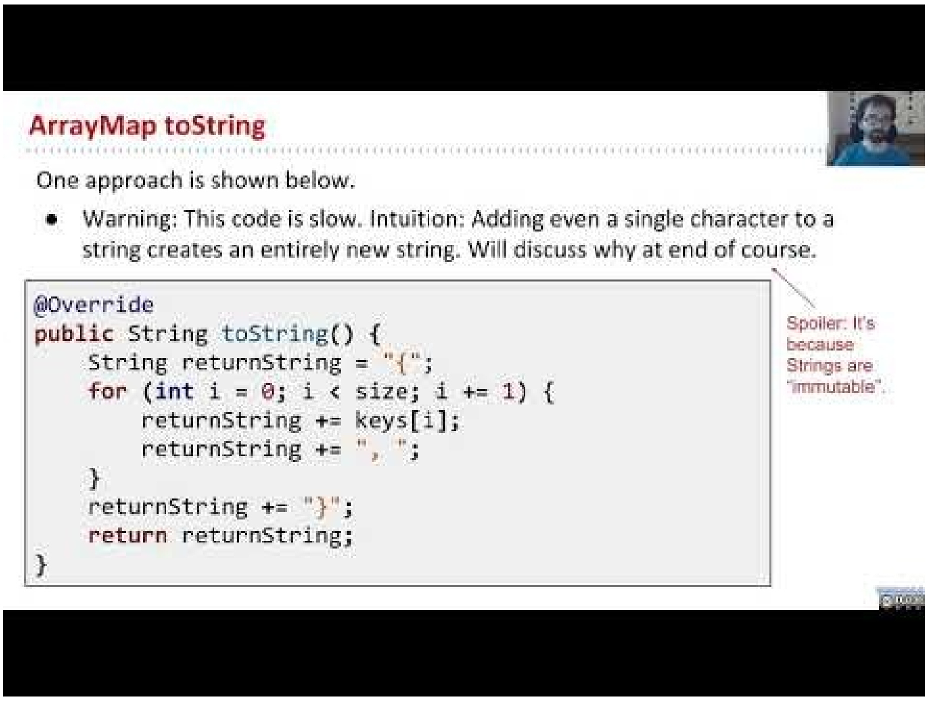
Object Methods

All classes inherit from the overarching Object class. The methods that are inherited are as follows:

- `String toString()`
- `boolean equals(Object obj)`
- `Class <?> getClass()`
- `int hashCode()`
- `protected Objectclone()`
- `protected void finalize()`
- `void notify()`
- `void notifyAll()`
- `void wait()`
- `void wait(long timeout)`
- `void wait(long timeout, int nanos)`

We are going to focus on the first two in this chapter. We will take advantage of inheritance to override these two methods in our classes to behave in the ways we want them to.

toString()



Video link

The `toString()` method provides a string representation of an object. The `System.out.println()` function implicitly calls this method on whatever object is passed to it and prints the string returned. When you run `System.out.println(dog)`, it's actually doing this:

```
String s = dog.toString()
System.out.println(s)
```

The default `Object` class' `toString()` method prints the location of the object in memory. This is a hexidecimal string. Classes like Arraylist and java arrays have their own overridden versions of the `toString()` method. This is why, when you were working with and writing tests for Arraylist, errors would always return the list in a nice format like this (1, 2, 3, 4) instead of returning the memory location.

For classes that we've written by ourselves like `ArrayDeque`, `LinkedListDeque`, etc, we need to provide our own `toString()` method if we want to be able to see the objects printed in a readable format.

Let's try to write this method for an `ArraySet` class. Read the `ArraySet` class below and make sure you understand what the various methods do. Feel free to plug the code into java visualizer to get a better understanding!

```
import java.util.Iterator;

public class ArraySet<T> implements Iterable<T> {
    private T[] items;
    private int size; // the next item to be added will be at position size

    public ArraySet() {
        items = (T[]) new Object[100];
        size = 0;
    }

    /* Returns true if this map contains a mapping for the specified key.
     */
    public boolean contains(T x) {
        for (int i = 0; i < size; i += 1) {
            if (items[i].equals(x)) {
                return true;
            }
        }
        return false;
    }

    /* Associates the specified value with the specified key in this map.
       Throws an IllegalArgumentException if the key is null. */
    public void add(T x) {
```

```
        if (x == null) {
            throw new IllegalArgumentException("can't add null");
        }
        if (contains(x)) {
            return;
        }
        items[size] = x;
        size += 1;
    }

    /* Returns the number of key-value mappings in this map. */
    public int size() {
        return size;
    }

    /** returns an iterator (a.k.a. seer) into ME */
    public Iterator<T> iterator() {
        return new ArraySetIterator();
    }

    private class ArraySetIterator implements Iterator<T> {
        private int wizPos;

        public ArraySetIterator() {
            wizPos = 0;
        }

        public boolean hasNext() {
            return wizPos < size;
        }

        public T next() {
            T returnItem = items[wizPos];
            wizPos += 1;
            return returnItem;
        }
    }

    @Override
    public String toString() {
        /* hmmm */
    }


    @Override
    public boolean equals(Object other) {
        /* hmmm */
    }

    public static void main(String[] args) {
        ArraySet<Integer> aset = new ArraySet<>();
        aset.add(5);
        aset.add(23);
```

```
        aset.add(42);

        //iteration
        for (int i : aset) {
            System.out.println(i);
        }

        //toString
        System.out.println(aset);

        //equals
        ArraySet<Integer> aset2 = new ArraySet<>();
        aset2.add(5);
        aset2.add(23);
        aset2.add(42);

        System.out.println(aset.equals(aset2));
        System.out.println(aset.equals(null));
        System.out.println(aset.equals("fish"));
        System.out.println(aset.equals(aset));
}
```

You can find the solutions here (ArraySet.java)

Exercise 6.4.1: Write the toString() method so that when we print an ArraySet, it prints the elements separated by commas inside of curly braces. i.e {1, 2, 3, 4}. Remember, the toString() method should return a string.

Solution

```
public String toString() {
    String returnString = "{";
    for (int i = 0; i < size; i += 1) {
        returnString += keys[i];
        returnString += ", ";
    }
    returnString += "}";
    return returnString;
}
```

This solution, although seemingly simple and elegant, is actually very naive. This is because when you use string concatenation in Java like so: `returnString += keys[i];` you are actually not just appending to `returnString`, you are creating an entirely new string. This is incredibly inefficient because creating a new string object takes time too! Specifically, linear in the length of the string.

Bonus Question: Let's say concatenating one character to a string takes 1 second. If we have an ArraySet of size 5: `{1, 2, 3, 4, 5}` , how long would it take to run the `toString()` method?

Answer: We set `returnString` to the left bracket which takes one second because this involves adding `{` to the empty string `""` . Adding the first element will involve creating an entirely new string, adding } and 1 which would take 2 seconds. Adding the second element takes 3 seconds because we need to add `{` , `1` , `2` . This process continues, so for the entire array set the total time is `1 + 2 + 3 + 4 + 5 + 6 + 7.`

To remedy this, Java has a special class called `StringBuilder` . It creates a string object that is mutable, so you can continue appending to the same string object instead of creating a new one each time.

Exercise 6.4.2: Rewrite the toString() method using StringBuilder.

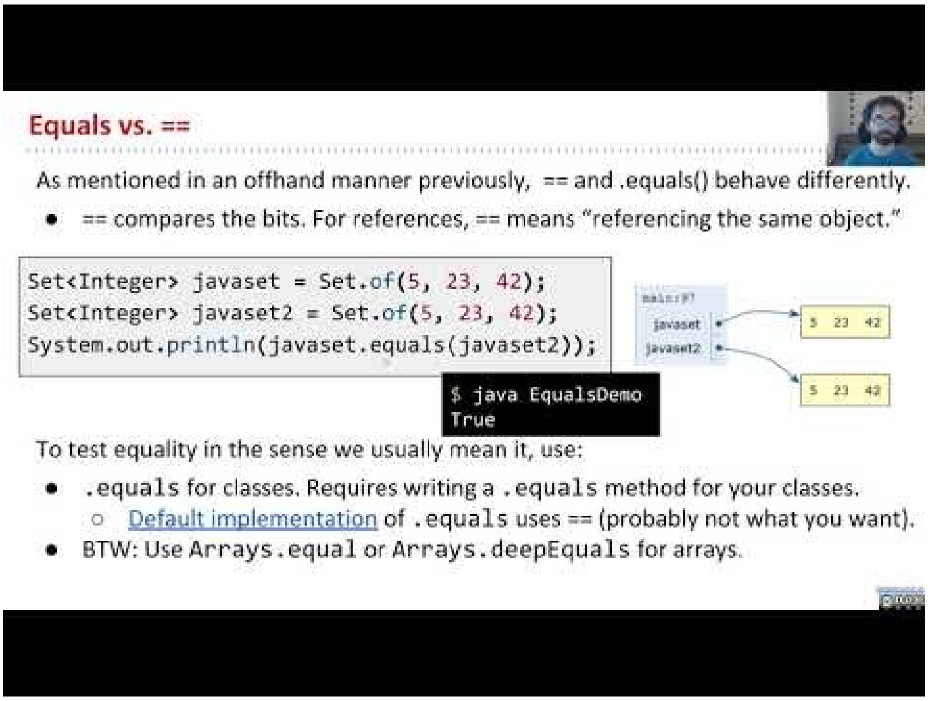
Solution

```
public String toString() {
        StringBuilder returnSB = new StringBuilder("{");
        for (int i = 0; i < size - 1; i += 1) {
            returnSB.append(items[i].toString());
            returnSB.append(", ");
        }
        returnSB.append(items[size - 1]);
        returnSB.append("}");
        return returnSB.toString();
    }
```

Now you've successfully overridden the `toString()` method! Try printing the ArraySet to see the fruits of your work.

Next we will override another important object method: `equals()`

equals()



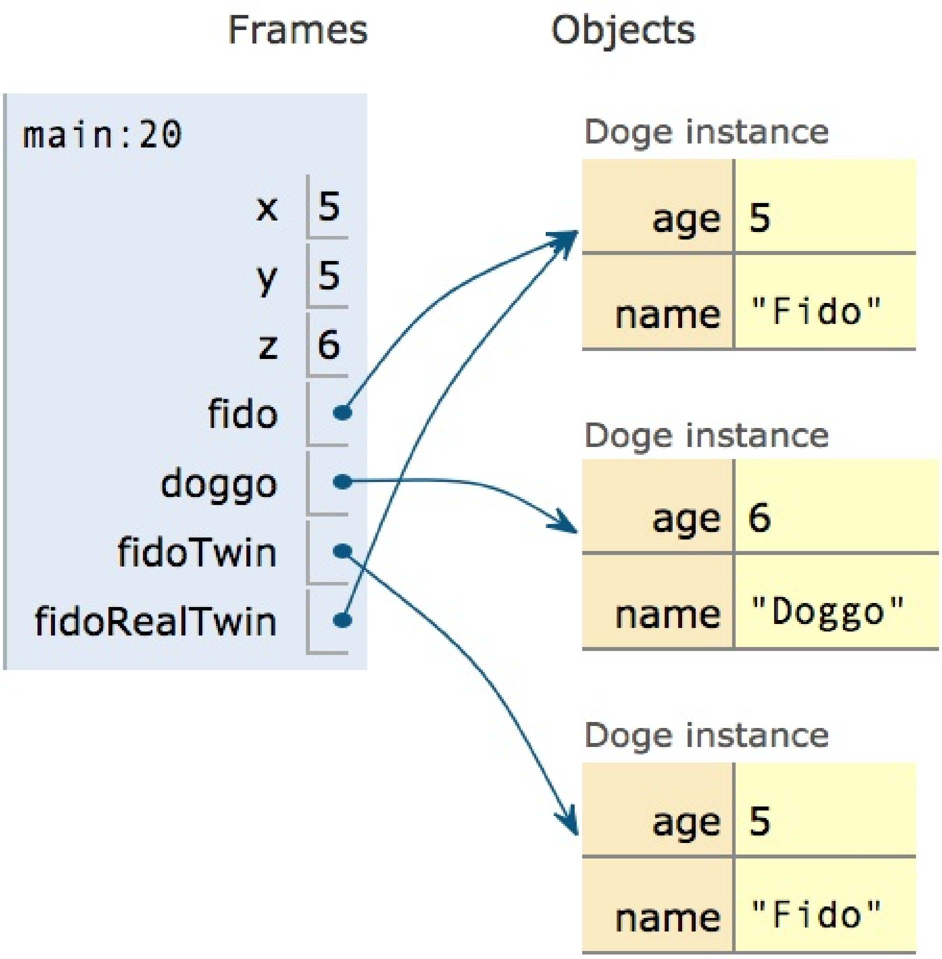
[Video link]

`equals()` and `==` have different behaviors in Java. `==` Checks if two objects are actually the same object in memory. Remember, pass-by-value! `==` checks if two boxes hold the same thing. For primitives, this means checking if the values are equal. For objects, this means checking if the address/pointer is equal.

Say we have this `Doge` class:

```java
public class Doge {

   public int age;
   public String name;

   public Doge(int age, String name){
      this.age = age;
      this.name = name;
   }
   public static void main(String[] args) {

      int x = 5;
      int y = 5;
      int z = 6;

      Doge fido = new Doge(5, "Fido");
      Doge doggo = new Doge(6, "Doggo");
      Doge fidoTwin = new Doge(5, "Fido");
      Doge fidoRealTwin = fido;
   }
}
```

If we plug this code into the java visualizer, we will see the box in pointer diagram shown below.



Exercise 6.4.2: What would java return if we ran the following?

- `x == y`
- `x == z`
- `fido == doggo`
- `fido == fidoTwin`
- `fido = fidoRealTwin`

Answers

- `True`
- `False`
- `False`
- `False`
- `True`

`fido` and `fidoTwin` are not considered `==` because they point to different objects. However, this is quite silly since all their attributes are the same. You can see how `==` can cause some problems in Java testing. When we write tests for our ArrayList and want to check if expected is the same as what is returned by our function, we create expected as a new arraylist. If we used `==` in our test, it would always return false. This is what `equals(Object o)` is for.

`equals(Object o)`

`equals(Object o)` is a method in the Object that, by default, acts like == in that it checks if the memory address of the this is the same as o. However, we can override it to define equality in whichever way we wish! For example, for two Arraylists to be considered equal, they just need to have the same elements in the same order.

Exercise 6.4.3: Let's write an equals method for the ArraySet class. Remember, a set is an unordered collection of unique elements. So, for two sets to be considered equal, you just need to check if they have the same elements.

Solution

```
public boolean equals(Object other) {
        if (this == other) {
            return true;
        }
        if (other == null) {
            return false;
        }
        if (other.getClass() != this.getClass()) {
            return false;
        }
        ArraySet<T> o = (ArraySet<T>) other;
        if (o.size() != this.size()) {
            return false;
        }
        for (T item : this) {
            if (!o.contains(item)) {
                return false;
            }
        }
        return true;
    }
```

We added a few checks in the beginning of the method to make sure our equals can handle nulls and objects of a different class. We also optimized the function by return true right away if the == methods returns true. This way, we avoid the extra work of iterating through the set.

Rules for Equals in Java: When overriding a `.equals()` method, it may sometimes be trickier than it seems. A couple of rules to adhere to while implementing your `.equals()` method are as follows:

1.) `equals` must be an equivalence relation

- **reflexive**: `x.equals(x)` is true
- **symmetric**: `x.equals(y)` if and only if `y.equals(x)`
- **transitive**: `x.equals(y)` and `y.equals(z)` implies `x.equals(z)`

2.) It must take an Object argument, in order to override the original `.equals()` method

3.) It must be consistent if `x.equals(y)`, then as long as `x` and `y` remain unchanged: `x` must continue to equal `y`

4.) It is never true for null `x.equals(null)` must be false

Bonus video

Create an even better `toString` method and `ArraySet.of`:

Video link

Link to the bonus code

Throwing Exceptions *(legacy)*

Video link

When something goes really wrong in a program, we want to break the normal flow of control. It may not make sense to continue on, or it may not be possible at all. In these cases, the program throws an exception.

Let's look at what might be a familiar case: an IndexOutOfBounds exception.

The code below inserts the value 5 into an ArrayMap under the key "hello", then tries to print out the value when getting "yolp."

```
public static void main (String[] args) {
    ArrayMap<String, Integer> am = new ArrayMap<String, Integer>();
    am.put("hello", 5);
    System.out.println(am.get("yolp"));
}
```

What happens when we run this? The program attempts to access a key which doesn't exist, and crashes! This results in the following error message:

```
$ java ExceptionDemo
Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException: -1
at ArrayMap.get(ArrayMap.java:38)
at ExceptionDemo.main(ExceptionDemo.java:6)
```

This is an *implicit exception*, an error thrown by Java itself. We can learn a little bit from this message: we see that the program crashed because of an ArrayIndexOutOfBoundsException; but it doesn't tell us very much besides that. You may have encountered similarly unhelpful error messages during your own programming endeavors. So how can we be more helpful to the user of our program?

We can throw our own exceptions, using the **throw** keyword. This lets us provide our own error messages which may be more informative to the user. We can also provide information to error-handling code within our program. This is an *explicit exception* because we purposefully threw it as the programmer.

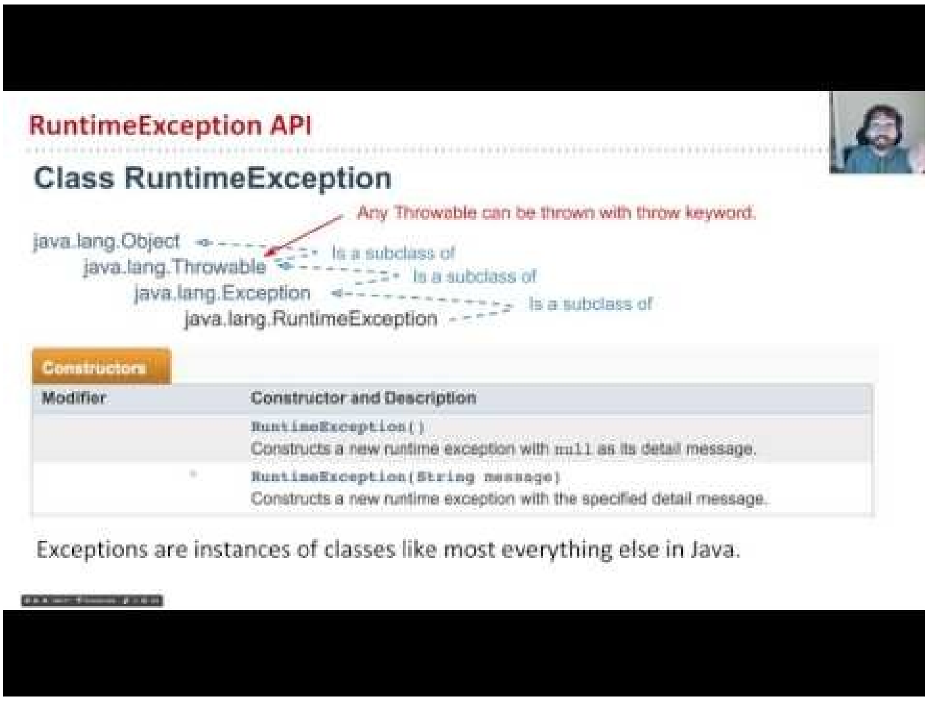
In the case above, we might implement get with a check for a missing key, that throws a more informative exception:

```
public V get(K key) {
    intlocation = findKey(key);
if(location < 0) {
    throw newIllegalArgumentException("Key " + key + " does not exist in map."\);
}
    return values[findKey(key)];
}
```

Now, instead of `java.lang.ArrayIndexOutOfBoundsException: -1`, we see:

```
$java ExceptionDemo
Exception in thread "main" java.lang.IllegalArgumentException: Key yolp does not exist
 in map.
at ArrayMap.get(ArrayMap.java:40)
at ExceptionDemo.main(ExceptionDemo.java:6)
```

Catching Exceptions



Video link

As we've seen, sometimes things go wrong while the program is running. Java handles these "exceptional events" by throwing an exception. In this section, we'll see what else we can do when such an exception is thrown.

Consider these error situations:

- You try to use 383,124 gigabytes of memory.
- You try to cast an Object as a Dog, but dynamic type is not Dog.
- You try to call a method using a reference variable that is equal to null.
- You try to access index -1 of an array.

So far, we've seen Java crash and print error messages with implicit exceptions:

```
Object o = "mulchor";
Planet x = (Planet) o;
```

resulting in:

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

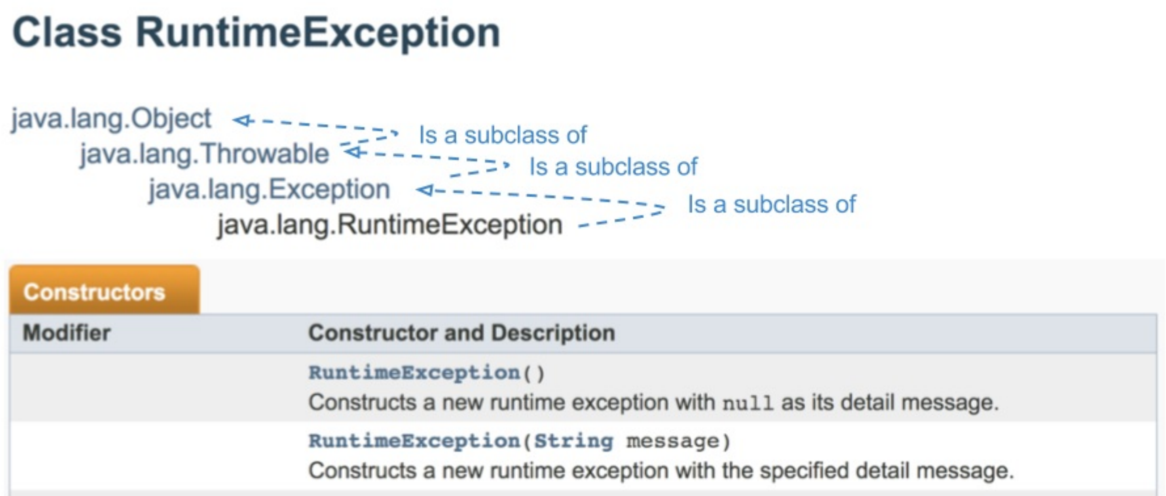
And above, we saw how to provide more informative errors by using explicit exceptions:

```java
public static void main(String[] args) {
    System.out.println("ayyy lmao");
    throw new RuntimeException("For no reason.");
}
```

which produces the error:

```
$ java Alien
ayyy lmao
Exception in thread "main" java.lang.RuntimeException: For no reason.
at Alien.main(Alien.java:4)
```

In this example, note a familiar construction: `new RuntimeException("For no reason.")`. This looks a lot like instantiating a class -- because that's exactly what it is. A RuntimeException is just a Java Object, like any other.


Class RuntimeException

java.lang.Object
java.lang.Throwable
java.lang.Exception
java.lang.RuntimeException
Is a subclass of
Is a subclass of
Is a subclass of

Constructors

Modifier	Constructor and Description
	`RuntimeException()` Constructs a new runtime exception with `null` as its detail message.
	`RuntimeException(String message)` Constructs a new runtime exception with the specified detail message.

So far, thrown exceptions cause code to crash. But we can ‘catch’ exceptions instead, preventing the program from crashing. The keywords *try* and *catch* break the normal flow of the program, protecting it from exceptions.

Consider the following example:

```
Dog d = new Dog("Lucy", "Retriever", 80);
d.becomeAngry();

try {
    d.receivePat();
} catch (Exception e) {
    System.out.println("Tried to pat: " + e);
}
System.out.println(d);
```

The output of this code might be:

```
$ java ExceptionDemo
Tried to pat: java.lang.RuntimeException: grrr... snarl snarl
Lucy is a displeased Retriever weighing 80.0 standard lb units.
```

Here we see that when we try and pat the dog when the dog is angry, it throws a RuntimeException, with the helpful error message "grrr...snarl snarl." But, it does continue on, and print out the state of the dog in the final line! This is because we caught the exception.

This might not seem particularly useful yet. But we can also use a catch statement to take corrective action.

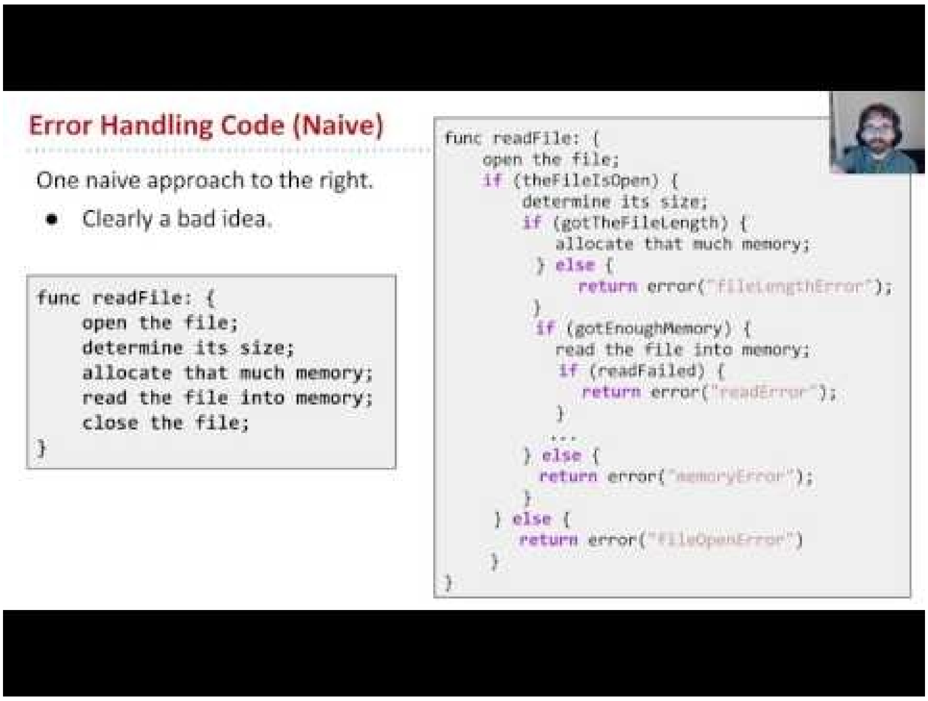
```
Dog d = new Dog("Lucy", "Retriever", 80);
d.becomeAngry();

try {
    d.receivePat();
} catch (Exception e) {
    System.out.println(
    "Tried to pat: " + e);
    d.eatTreat("banana");
}
d.receivePat();
System.out.println(d);
```

In this version of our code, we soothe the dog with a treat. Now when we try and pat it again, the method executes without failing.

```
$ java ExceptionDemo
Tried to pat: java.lang.RuntimeException: grrr... snarl snarl
Lucy munches the banana

Lucy enjoys the pat.

Lucy is a happy Retriever weighing 80.0 standard lb units.
```

In the real world, this corrective action might be extending an antenna on a robot when an exception is thrown by an operation expecting a ready antenna. Or perhaps we simply want to write the error to a log file for later analysis.

The Philosophy Of Exceptions



[Video link]

Exceptions aren't the only way to do error handling. But they do have some advantages. Most importantly, they keep error handling conceptually separate from the rest of the program.

Let's consider some psuedocode for a program that reads from a file:

```
func readFile: {
    open the file;
    determine its size;
    allocate that much memory;
    read the file into memory;
    close the file;
}
```

A lot of things might go wrong here: maybe the file doesn't exist, maybe there isn't enough memory, or the reading fails.

Without exceptions, we might handle the errors like this:

```
func readFile: {
    open the file;
    if (theFileIsOpen) {
        determine its size;
        if (gotTheFileLength) {
            allocate that much memory;
        } else {
            return error("fileLengthError");
        }
            if (gotEnoughMemory) {
                read the file into memory;
            if (readFailed) {
                return error("readError");
            }
        ...
        } else {
            return error("memoryError");
        }
    } else {
        return error("fileOpenError")
    }
}
```

But this super messy! And deeply frustrating to read.

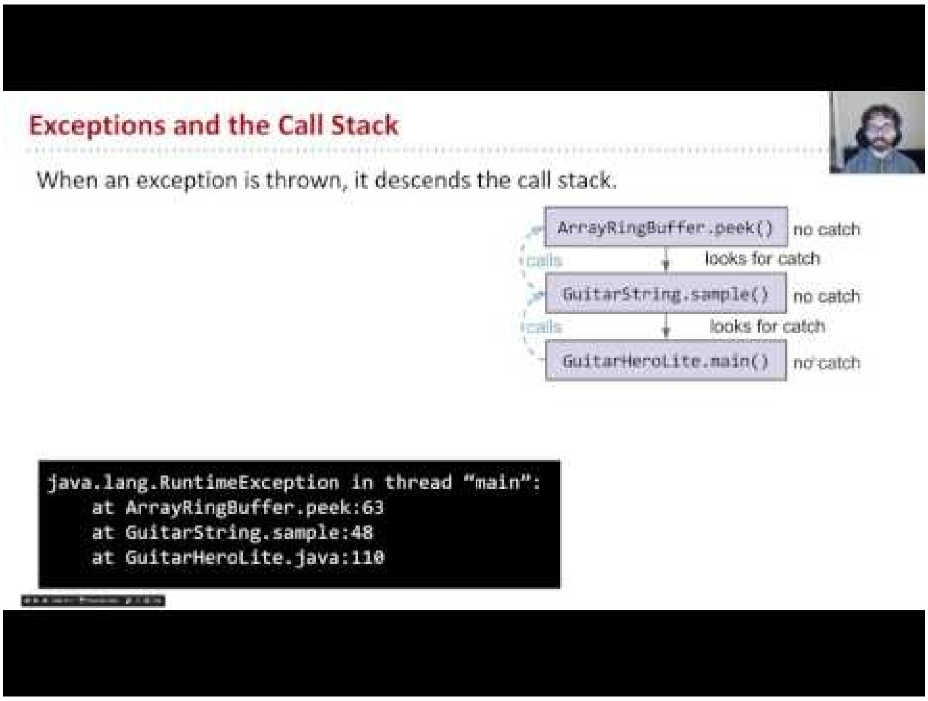
With exceptions, we might rewrite this as:

```
func readFile: {
    try {
        open the file;
        determine its size;
        allocate that much memory;
        read the file into memory;
        close the file;
    } catch (fileOpenFailed) {
        doSomething;
    } catch (sizeDeterminationFailed) {
        doSomething;
    } catch (memoryAllocationFailed) {
        doSomething;
    } catch (readFailed) {
        doSomething;
    } catch (fileCloseFailed) {
        doSomething;
    }
}
```

Here, we first do all the things associated with reading our file, and wrap them in a try statement. Then, if an error happens anywhere in that sequence of operations, it will get caught by the appropriate catch statement. We can provide distinct behaviors for each type of exception.

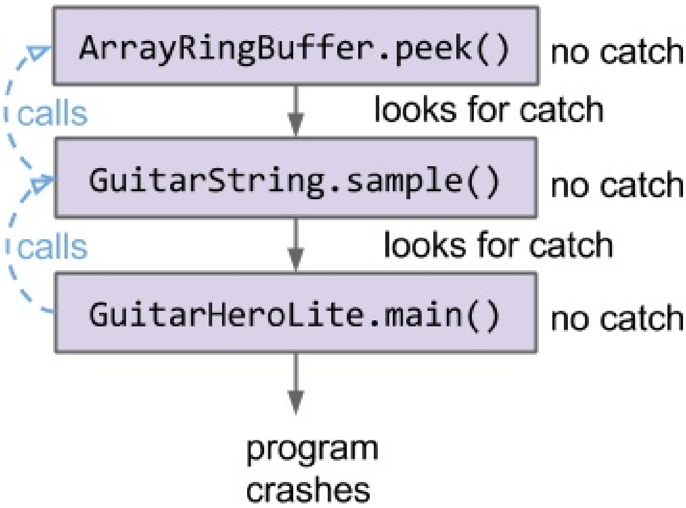
The key benefit of the exceptions version, in contrast to the naive version above, is that the code flows in a clean narrative. First, try to do the desired operations. Then, catch any errors. Good code feels like a story; it has a certain beauty to its construction. That clarity makes it easier to both write and maintain over time.

Uncaught Exceptions



Video link

When an exception is thrown, it descends the call stack.



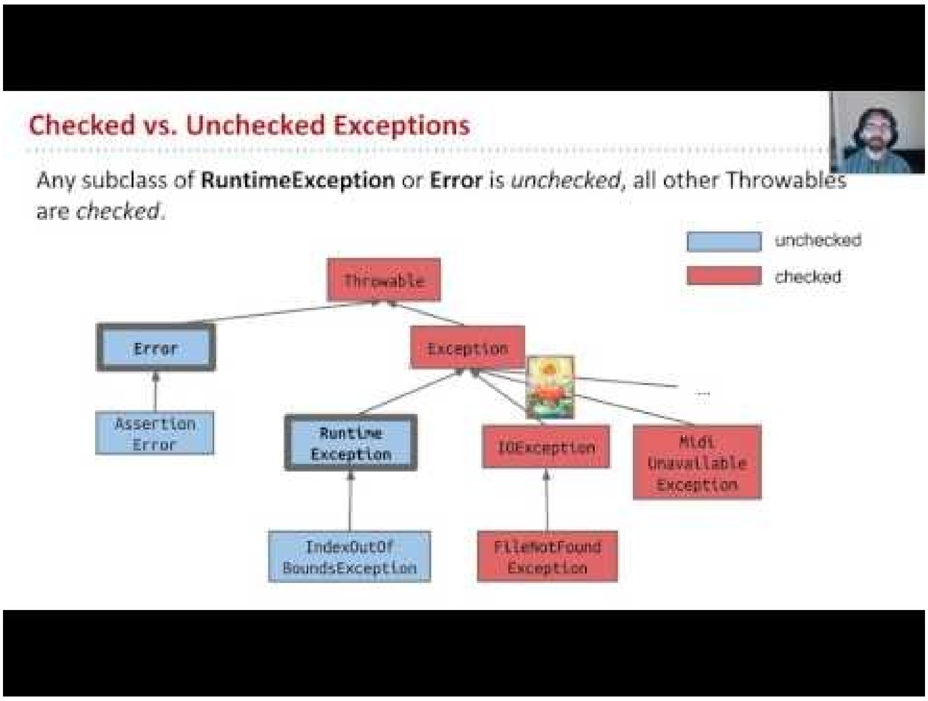
If the `peek()` method does not explicitly catch the exception, the exception will propagate to the calling function, `sample()`. We can think of this as popping the current method off the stack, and moving to the next method below it. If `sample()` also fails to catch the exception, it moves to `main()`.

If the exception reaches the bottom of the stack without being caught, the program crashes and Java provides a message for the user, printing out the *stack trace*. Ideally the user is a programmer with the power to do something about it.

```
java.lang.RuntimeException in thread “main”:
at ArrayRingBuffer.peek:63
at GuitarString.sample:48
at GuitarHeroLite.java:110
```

We can see by looking at the stack trace where the error occurred: on line 63 of `ArrayRingBuffer.peek()`, after being called by line 48 of `GuitarString.sample()`, after being called by the main method of `GuitarHeroLite.java` on line 110. But this isn't super helpful unless the user also happens to be a programmer with the power to do something about the error.

Checked vs Unchecked Exceptions *(legacy)*



[Video link](#)

The exceptions we've seen above have all occurred at runtime. Occasionally, you'll find that your code won't even compile, for the mysterious reason that an exception "must be caught or declared to be thrown".

What's going on in that case? The basic idea is that some exceptions are considered so disgusting by the compiler that you MUST handle them somehow.

We call these "checked" exceptions. (You might think of that as shorthand for "must be checked" exceptions.)

Let's consider this example:

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

It looks reasonable enough. But when we attempt to compile, we receive this error:

```
$ javac What.java
What.java:2: error: unreported exception IOException; must be caught or declared to be
 thrown
Eagle.gulgate();
^
```

We can't compile, because of an "unreported IOException." Let's look a little deeper into the Eagle class:

```
public class Eagle {
    public static void gulgate() {
        if (today == “Thursday”) {
            throw new IOException("hi"); }
        }
    }
```

On Thursdays, the `gulgate()` method is programmed to throw an IOException. If we try and compile Eagle.java, we receive a similar error to the one we saw when compiling the calling class above:

```
$ javac Eagle
Eagle.java:4: error: unreported exception IOException; must be caught or declared to b
e thrown
throw new IOException("hi"); }
^
```

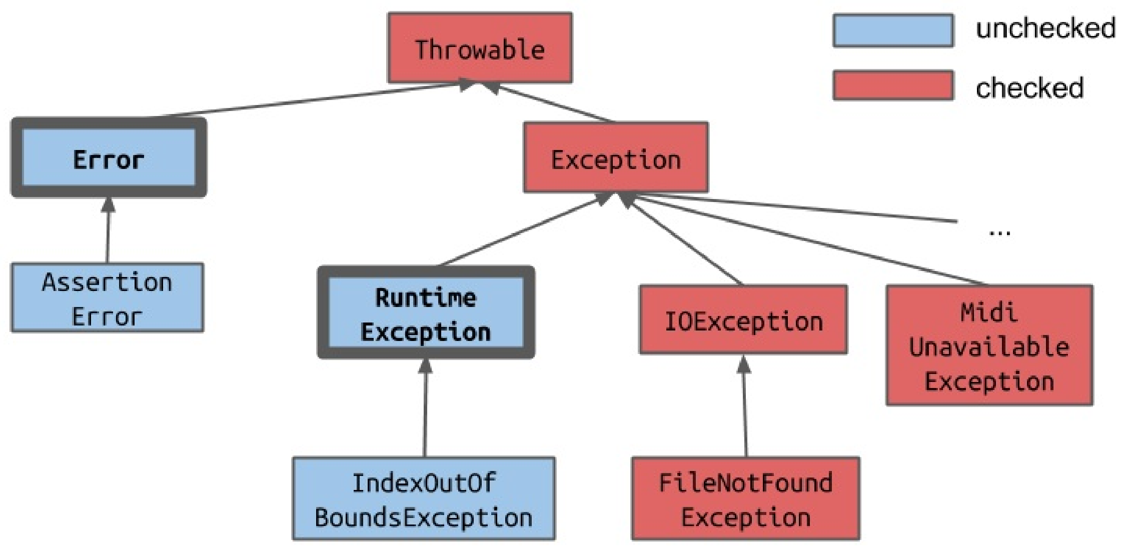
It's clear that Java isn't happy about this IOException. This is because IOExceptions are "checked' exceptions and must be handled accordingly. We will go over how this handling occurs a bit later in the chapter. But what if we threw a RuntimeException instead, like we did in previous sections?

```
public class UncheckedExceptionDemo {
    public static void main(String[] args) {
        if (today == “Thursday”) {
            throw new RuntimeException("as a joke");
        }
    }
}
```

RuntimeExceptions are considered "unchecked" exceptions, and do not have the same requirements as the checked exceptions. The code above will compile just fine -- though it will crash at runtime on Thursdays:

```
$ javac UncheckedExceptionDemo.java
$ java UncheckedExceptionDemo
Exception in thread "main" java.lang.RuntimeException: as a joke.
at UncheckedExceptionDemo.main(UncheckedExceptionDemo.java:3)
```

How do we know which types of exceptions are checked, and which are unchecked?



Errors and Runtime Exceptions, and all their children, are unchecked. These are errors that cannot be known until runtime. They also tend to be ones that can't be recovered from -- what can you do to fix it if the code tries to get the -1 element from an array? Not much.

Everything else is a checked exception. Most of these have productive fixes. For instance, if we run into a FileNotFound Exception, perhaps we can ask the user to re-specify the file they want -- they might have mistyped it.

Since Java is on your side, and wants to do its best to make sure that every program runs without crashing, it will not let a program with a possible fixable error compile unless it is indeed handled in some way.

There are two ways to handle a checked error:

1) Catch 2) Specify

Using a **catch** block is what we have seen above. In our `gulgate()` method, it might look like this:

```
public static void gulgate() {
    try {
        if (today == “Thursday”) {
            throw new IOException("hi");
        }
    } catch (Exception e) {
        System.out.println("psych!");
    }
}
```

If we don't want to handle the exception in the `gulgate()` method, we can instead defer the responsibility to somewhere else. We mark, or **specify** the method as dangerous by modifying the method definition as follows:

```
public static void gulgate() throws IOException {
... throw new IOException("hi"); ...
}
```

But specifying the exception does not yet handle it. When we call `gulgate()` from somewhere else, that new method now becomes dangerous as well!

Since `gulgate()` might throw an uncaught exception, now `main()` can also throw that exception, and the following code won't compile:

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

We can solve this in one of two ways: catch, or specify in the calling method.

Catch:

```
public static void main(String[] args) {
    try {
        gulgate();
    } catch(IOException e) {
        System.out.println("Averted!");
    }
}
```

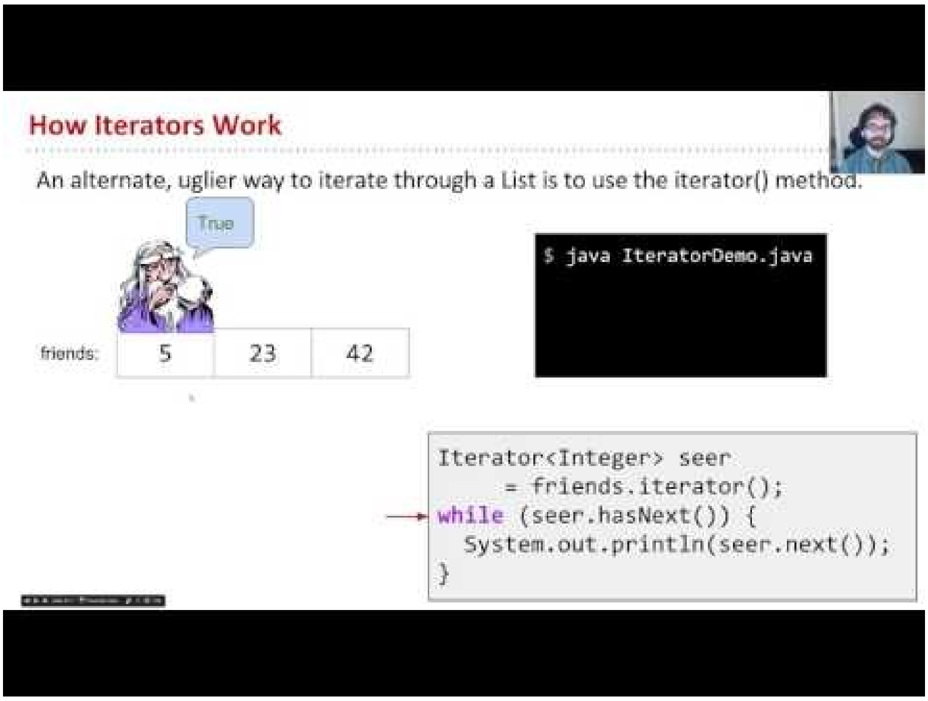
Specify:

```
public static void main(String[] args) throws IOException {
    gulgate();
}
```

Catch the error when you can handle the problem there. Keep it from escaping!

Specify the error when someone else should handle the error. Make sure the caller knows the method is dangerous!

Iteration *(legacy)*



[Video link]

We saw that Java allows us to iterate through Lists using a convenient shorthand syntax sometimes called the “foreach” or “enhanced for” loop.

For example,

```
List<Integer> friends =
new ArrayList<Integer>();
friends.add(5);
friends.add(23);
friends.add(42);
for (int x : friends) {
    System.out.println(x);
}
```

Let’s strip away the magic so we can build our own classes that support this.

The key here is an object called an *iterator*.

For our example, in List.java we might define an `iterator()` method that returns an iterator object.

```
public Iterator<E> iterator();
```

Now, we can use that object to loop through all the entries in our list:

```
List<Integer> friends = new ArrayList<Integer>();
...
Iterator<Integer> seer = friends.iterator();

while (seer.hasNext()) {
    System.out.println(seer.next());
}
```

This code behaves identically to the foreach loop version above.

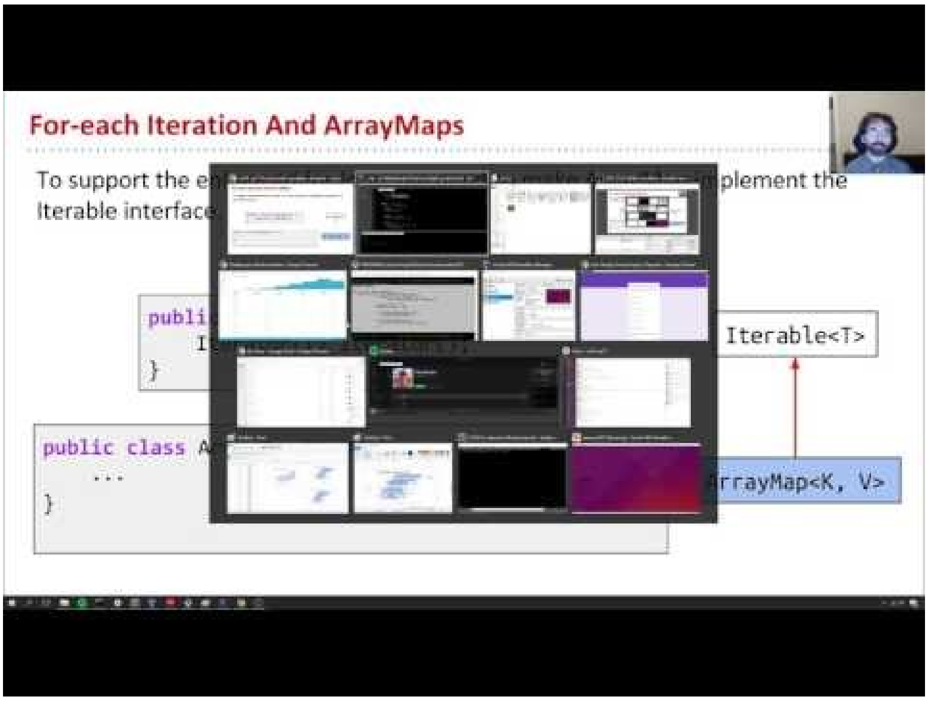
There are three key methods in our iterator approach:

First, we get a new iterator object with `Iterator<Integer> seer = friends.iterator();`

Next, we loop through the list with our while loop. We check that there are still items left with `seer.hasNext()` , which will return true if there are unseen items remaining, and false if all items have been processed.

Last, `seer.next()` does two things at once. It returns the next element of the list, and here we print it out. It also advances the iterator by one item. In this way, the iterator will only inspect each item once.

Implementing Iterators



Video link

In this section, we are going to talk about how to build a class to support iteration.

Let's start by thinking about what the compiler need to know in order to successfully compile the following iterator example:

```
List<Integer> friends = new ArrayList<Integer>();
Iterator<Integer> seer = friends.iterator();

while(seer.hasNext()) {
    System.out.println(seer.next());
}
```

We can look at the static types of each object that calls a relevant method. `friends` is a List, on which `iterator()` is called, so we must ask:

- Does the List interface have an iterator() method?

`seer` is an Iterator, on which `hasNext()` and `next()` are called, so we must ask:

- Does the Iterator interface have next/hasNext() methods?

So how do we implement these requirements?

The List interface extends the Iterable interface, inheriting the abstract iterator() method. (Actually, List extends Collection which extends Iterable, but it's easier to codethink of this way to start.)

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

```
public interface List<T> extends Iterable<T>{
...
}
```

Next, the compiler checks that Iterators have `hasNext()` and `next()`. The Iterator interface specifies these abstract methods explicitly:

```
package java.util;
public interface Iterator<T> {
    boolean hasNext();
    T next();
}
```

Specific classes will implement their own iteration behaviors for the interface methods. Let's look at an example. (Note: if you want to build this up from the start, follow along with the live coding in the video.)

We are going to add iteration through keys to our ArrayMap class. First, we write a new class called KeyIterator, nested inside of ArrayMap:

```
public class KeyIterator {
    private int ptr;
    public KeyIterator() {
        ptr = 0;
    }

    public boolean hasNext() {
        return (ptr != size);
    }

    public K next() {
        K returnItem = keys[ptr];
        ptr = ptr + 1;
        return returnItem;
    }
}
```

This KeyIterator implements a `hasNext()` method, and a `next()` method, using a `ptr` variable to keep track of its position in the array of keys. For a different data structure, we might implement these two methods differently.

Thought Excercise: How would you design `hasNext()` and `next()` for a linked list?

Now that we have the appropriate methods, we can use a KeyIterator to iterate through an ArrayMap:

```
ArrayMap<String, Integer> am = new ArrayMap<String, Integer>();
am.put("hello", 5);
am.put("syrups", 10);
ArrayMap.KeyIterator ami = am.new KeyIterator();

while (ami.hasNext()) {
    System.out.println(ami.next());
}
```

There's an interesting line in the code above: `am.new KeyIterator();`. This construction allows us to instantiate a non-static nested class, meaning a class that needs to be associated with a particular instance of the enclosing class. It wouldn't make sense to have a KeyIterator not associated with a particular ArrayMap -- what would it iterate through? So, we must use dot notation with a specific ArrayMap instance to create a new KeyIterator associated with that instance.

Now we have a KeyIterator, and it can loop through an ArrayMap. We still want to be able to support the enhanced for loop, though, to make our calls cleaner. So, we need to make ArrayMap implement the Iterable interface. The essential method of the Iterable interface is `iterator()` , which returns an Iterator object for that class:

```
public class ArrayMap<K, V> implements Iterable<K>
    @Override
    public Iterator<T> iterator() {
        return new KeyIterator();
    }
}
```

We override that `iterator()` method to return the KeyIterator that we just wrote.

There's one more step before the code will compile: we have to tell Java that KeyIterator is an Iterator. To make that happen, KeyIterator must implement Iterator. This way, we can return a KeyIterator in our `iterator()` method above, and successfully implement the Iterator methods:

```
public class KeyIterator implements Iterator<K> {
    private int ptr;
    public KeyIterator() { ptr = 0; }
    public boolean hasNext() { return (ptr != size); }
    public K next() { ... }
}
```

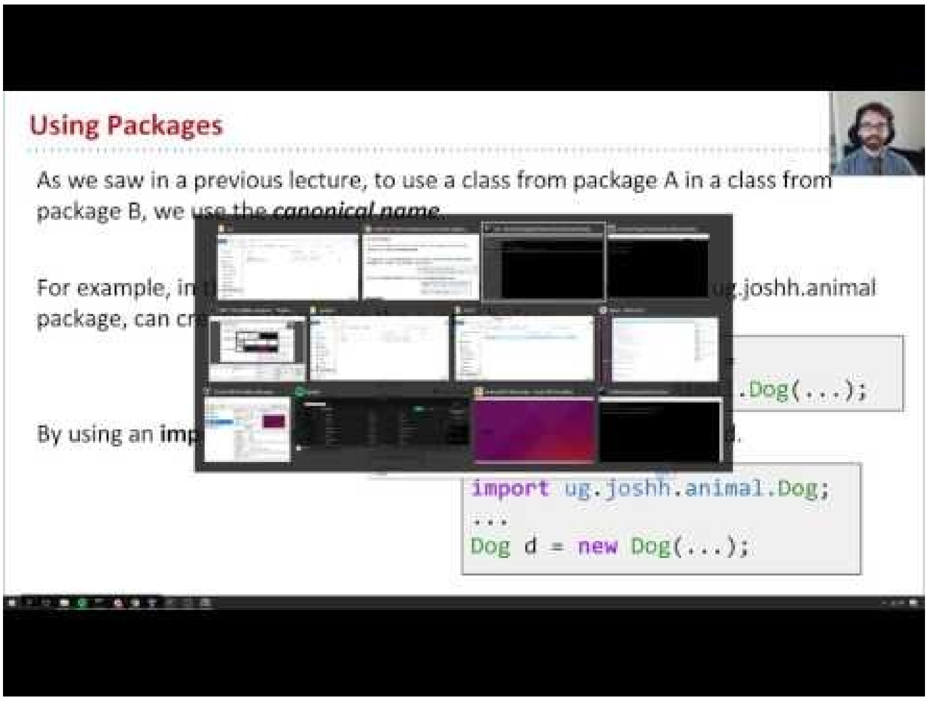
Now we can use our enhanced for loop construction with an ArrayMap:

```
ArrayMap<String, Integer> am = new ArrayMap<String, Integer>();
for (String s : am) {
    System.out.println(s);
}
```

Here we've seen **Iterable**, the interface that makes a class able to be iterated on, and requires the method `iterator()` , which returns an Iterator object. And we've seen **Iterator**, the interface that defines the object with methods to actually do that iteration. You can think of an Iterator as a machine that you put onto an iterable that facilitates the iteration. Any iterable is the object on which the iterator is performing.

With these two components, you can make fancy for loops for your classes!

Packages and JAR files



Video link

It is very possible that with all the code in this world, you would create classes that share names with those from a different project. How can you then organize these classes, such that there is less ambiguity when you're trying to access or use them? How will your program know that you mean to use your Dog.class, versus Josh Hug's Dog.class?

Herein enters the **package** — a namespace that organizes classes and interfaces. In general, when creating packages you should follow the following naming convention: package name starts with the website address, backwards.

For example, if Josh Hug were trying to distribute his Animal package, which contains various different types of animal classes, he would name his package as following:

```
ug.joshh.animal; // note: his website is joshh.ug
```

However, in CS61B you do not have to follow this convention, as your code isn't intended for distribution.

Using Packages

If you're accessing the class from within the same package, you can just use its simple name:

```
Dog d = new Dog(...);
```

If you're accessing the classes from outside the package, then use its entire canonical name:

```
ug.joshh.animal.Dog d = new ug.joshh.animal.Dog(...);
```

To make things easier, you can import the package, and use the simple name instead!

```
import ug.joshh.animal.Dog;
...
Dog d = new Dog(...);
```

Creating a Package

Creating a package takes the following two steps:

1.) Put the package name at the top of every file in this package

```
package ug.joshh.animal;

public class Dog {
    private String name;
    private String breed;
    …
}
```

2.) Store the file in a folder that has the appropriate folder name. The folder should have a name that matches your package:

i.e. `ug.joshh.animal` package is in ug/joshh/animal folder

Creating a Package, in IntelliJ

1.) File → New Package

1.) Choose package name (i.e. "ug.joshh.animal")

Adding (new) Java Files to a Package, in IntelliJ

1.) Right-click package name

2.) Select New → Java Class

3.) Name your class, and IntelliJ will automatically put it in the correct folder + add the "package ug.joshh.animal" declaration for you.

Adding (old) Java Files to a Package, in IntelliJ

1.) Add "package [packagename]" to the top of the file.

2.) Move the .java file into the corresponding folder.

Default packages

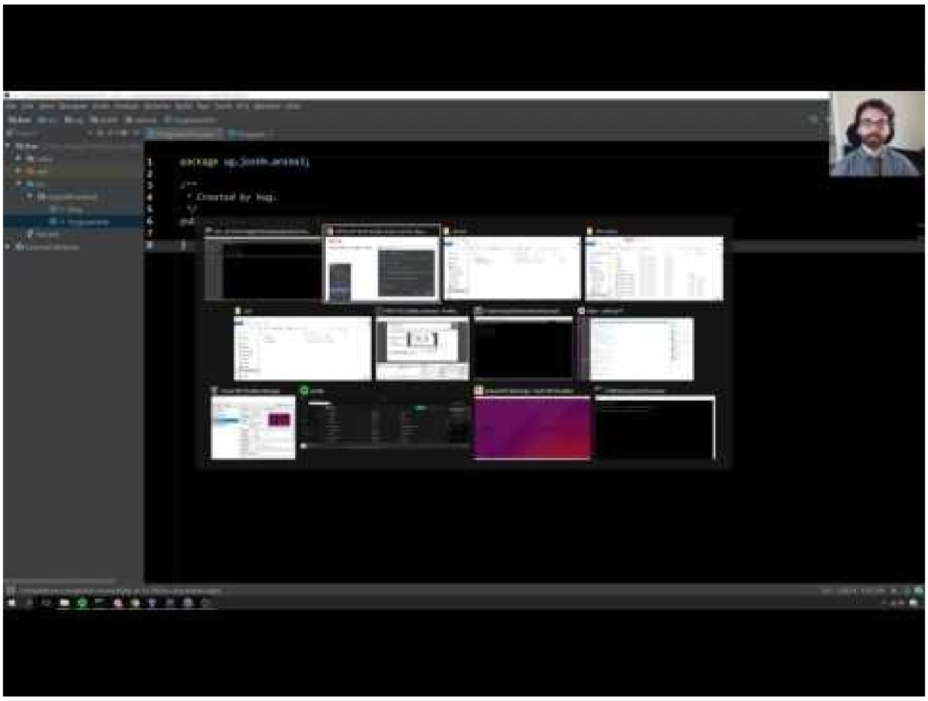
Any Java class without an explicit package name at the top of the file is automatically considered to be part of the "default" package. However, when writing real programs, you should avoid leaving your files in the default package (unless it's a very small example program). This is because code from the default package cannot be imported, and it is possible to accidentally create classes with the same name under the default package.

For example, if I were to create a "DogLauncher.java" class in the default package, I would be unable to access this DogLauncher class anywhere else outside of the default package.

```
DogLauncher.launch(); // won't work
default.DogLauncher.launch(); // doesn't exist
```

Therefore, your Java files should generally start with an explicit package declaration.

JAR Files



Video link

Oftentimes, programs will contain multiple .class files. If you wanted to share this program, rather than sharing all the .class files in special directories, you can “zip” all the files together by creating a JAR file. This single .jar file will contain all your .class files, along with some other additional information.

It is important to note that JAR files are just like zip files. It is entirely possible to unzip and transform the files back into .java files. JAR files do not keep your code safe, and thus you should not share your .jar files of your projects with other students.

Creating a JAR File (IntelliJ)

1.) Go to File → Project Structure → Artifacts → JAR → “From modules with dependencies”

2.) Click OK a couple of times

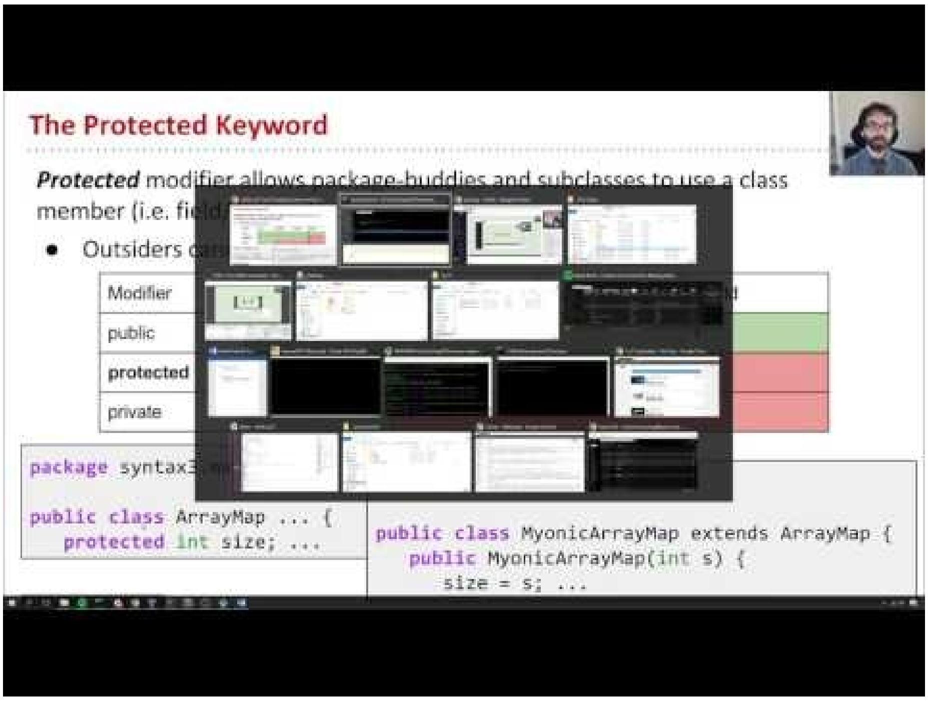
3.) Click Build → Build Artifacts (this will create a JAR file in a folder called “Artifacts”)

4.) Distribute this JAR file to other Java programmers, who can now import it into IntelliJ (or otherwise)

Build Systems

Rather than importing a list of libraries or whatnot each time we wanted to create a project, we can simply put the files into the appropriate place, and use “Build Systems” to automate the process of setting up your project. The advantages of Build Systems are especially seen in bigger teams and projects, where it’s largely beneficial to automate the process of setting up the project structure. Though the advantages of Build Systems are rather minimal in 61B, we did use Maven in Project 3 (BearMaps, Spring 2017), which is one of many popular build systems (including Ant and Gradle).

Access Control



Video link

We now run into the question of how public and private members behave in packages and subclasses. Think to yourself right now: when inheriting from a parent class, can we access the private members in that parent class? Or, can two classes in the same package access the other's private members?

If you don't know the answers right away, you can read on to find out!

Private Only code from the given class can access **private** members. It is truly *private* from everything else, as subclasses, packages, and other external classes cannot access private members. *TL;DR: only the class needs this piece of code*

Package Private This is the default access given to Java members if there is no explicit modifier written. Package private entails that classes that belong in the same package can access, but not subclasses! Why is this useful? Usually, packages are handled and modified by the same (group of) people. It is also common for people to extend classes that they didn't initially write. The original owners of the class that's being extended may not want certain features or members to be tampered with, if people choose to extend it — hence, package-private allows those who are familiar with the inner workings of the program to access and modify certain members, whereas it blocks those who are subclassing from doing the same. *TL;DR: only classes that live in the same package can access*

Protected Protected members are protected from the "outside" world, so classes within the same package and subclasses can access these members, but the rest of the world (e.g. classes external to the package or non-subclasses) cannot! *TL;DR: subtypes might need it, but subtype clients will not*

Public This keyword opens up the access to everyone! This is generally what clients of the package can rely on to use, and once deployed, the public members' signatures should not change. It's like a promise and contract to people using this public code that it will always be accessible to them. Usually if developers want to "get rid of" something that's public, rather than removing it, they would call it "deprecated" instead.

TL;DR: open and promised to the world

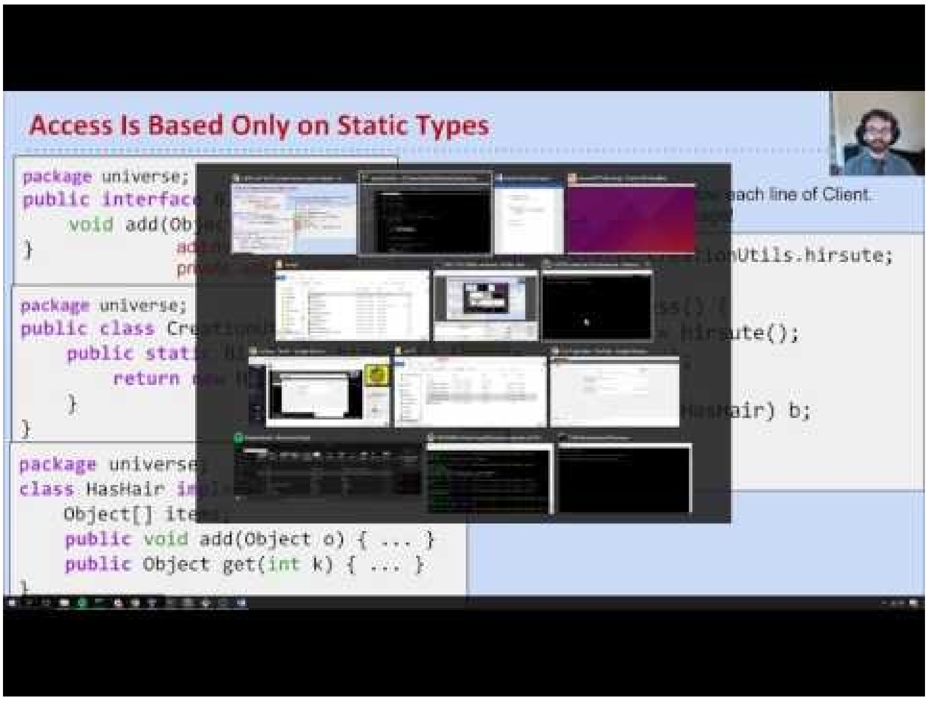
Exercise 7.1.1 See if you can draw the access table yourself, from memory.

Have the following be the column titles: Modifier, Class, Package, Subclass, World, with the following as the Rows: public, protected, package-private, private.

Indicate whether or not each row/access type has access to that particular column's "type".

Modifier	Class	Package	Subclass	World
public	Y	Y	Y	Y
protected	Y	Y	Y	N
[redacted]	Y	Y	N	N
private	Y	N	N	N

Access Control Subtleties



Video link

Default Package Code that does not have a package declaration is automatically part of the default package. If these classes have members that don't have access modifiers (i.e. are package-private), then because everything is part of the same (unnamed) default package, these members are still accessible between these "default"-package classes.

Access is Based Only on Static Types It is important to note that for interfaces, the default access for its methods is actually public, and not package-private. Additionally, like this subtitle indicates, the access depends only on the static types.

Exercise 7.1.2

Given the following code, which lines in the demoAccess method, if any, will error during compile time?

```java
package universe;
public interface BlackHole {
    void add(Object x); // this method is public, not package-private!
}

package universe;
public class CreationUtils {
    public static BlackHole hirsute() {
         return new HasHair();
    }
}

package universe;
class HasHair implements BlackHole {
    Object[] items;
    public void add(Object o) { ... }
    public Object get(int k) { ... }
}

import static CreationUtils.hirsute;
class Client {
   void demoAccess() {
      BlackHole b = hirsute();
      b.add("horse");
      b.get(0);
      HasHair hb = (HasHair) b;
   }
}
```

ANSWER

- b.get(0); This line errors because `b` is of static type `BlackHole`, but the `BlackHole` interface does not define a `get` method! Even though you and I both know that `b` is dynamically a `HasHair`, and thus has the `get` method, the compiler bases its checks off the static type.
- HasHair hb = (HasHair) b; This one is tricky, but notice that the `HasHair` class is not a public class - it's package-private. This means that `Client`, a class outside of the `universe` package, can't see that the `HasHair` class exists.

Efficient Programming

“An engineer will do for a dime what any fool will do for a dollar” -- Paul Hilfinger

Efficiency comes in two flavors:

1.) Programming cost.

- How long does it take to develop your programs?
- How easy is it to read, modify, and maintain your code?

2.) Execution cost (starting next week).

- How much time does your program take to execute?
- How much memory does your program require?

Today, we will be focusing on how to reduce programming cost. Of course, want to keep programming costs low, both so we can write code faster and so we can have less frustrated people which will also help us write code faster (people don't code very fast when they are frustrated).

Some helpful Java features discussed in 61B:

- Packages.
  - Good: Organizing, making things package private
  - Bad: Specific
- Static type checking.
  - Good: Checks for errors early , reads more like a story
  - Bad: Not too flexible, (casting)
- Inheritance.
  - Good: Reuse of code
  - Bad: “Is a”, the path of debugging gets annoying, can’t instantiate, implement every method of an interface

We will explore some new ways in this chapter!

Encapsulation

We will first define a few terms:

- **Module:** A set of methods that work together as a whole to perform some task or set of related tasks.
- **Encapsulated:** A module is said to be encapsulated if its implementation is completely

hidden, and it can be accessed only through a documented interface.

API's

An API(Application Programming Interface) of an ADT is the list of constructors and methods and a short description of each.

API consists of syntactic and semantic specification.

- Compiler verifies that **syntax** is met.
    - AKA, everything specified in the API is present.
- Tests help verify that **semantics** are correct.
    - AKA everything actually works the way it should.
    - Semantic specification usually written out in English (possibly including usage examples). Mathematically precise formal specifications are somewhat possible but not widespread.

ADT's

ADT's (Abstract Data Structures) are high-level types that are defined by their **behaviors**, not their implementations.

i.e.) Deque in Proj1 was an ADT that had certain behaviors (addFirst, addLast, etc.). But, the data structures we actually used to implement it was ArrayDeque and LinkedListDeque

Some ADT's are actually special cases of other ADT's. For example, Stacks and Queues are just lists that have even more specific behavior.

Exercise 8.1.1
Write a Stack class using a Linked List as its underlying data structure. You only need to implement a single function: push(Item x). Make sure to make the class generic with "Item" being the generic type!

You may have written it a few different ways. Let's look at three popular solutions:

```
public class ExtensionStack<Item> extends LinkedList<Item> {
    public void push(Item x) {
        add(x);
    }
}
```

This solution uses *extension*. it simply borrow the methods from `LinkedList<Item>` and uses them as its own.

```
public class DelegationStack<Item> {
    private LinkedList<Item> L = new LinkedList<Item>();
    public void push(Item x) {
        L.add(x);
    }
}
```

This approach uses Delegation. It creates a Linked List object and calls its methods to accomplish its goal.

```
public class StackAdapter<Item> {
    private List L;
    public StackAdapter(List<Item> worker) {
        L = worker;
    }

    public void push(Item x) {
        L.add(x);
    }
}
```

This approach is similar to the previous one, except it can use any class that implements the **List** interface (Linked List, ArrayList, etc).

Warning: be mindful of the difference between "is-a" and "has-a" relationships.

- A cat has-a claw
- A cat is-a feline

Earlier in the section define that delegation is accomplished by passing in a class while extension is defined as inheriting (just because it may be hard to notice at first glance).

Delegation vs Extension: Right now it may seem that Delegation and Extension are pretty much interchangeable; however, there are some important differences that must be remembered when using them.

Extension tends to be used when you know what is going on in the parent class. In other words, you know how the methods are implemented. Additionally, with extension, you are basically saying that the class you are extending from acts similarly to the one that is doing the extending. On the other hand, Delegation is when you do not want to consider your current class to be a version of the class that you are pulling the method from.

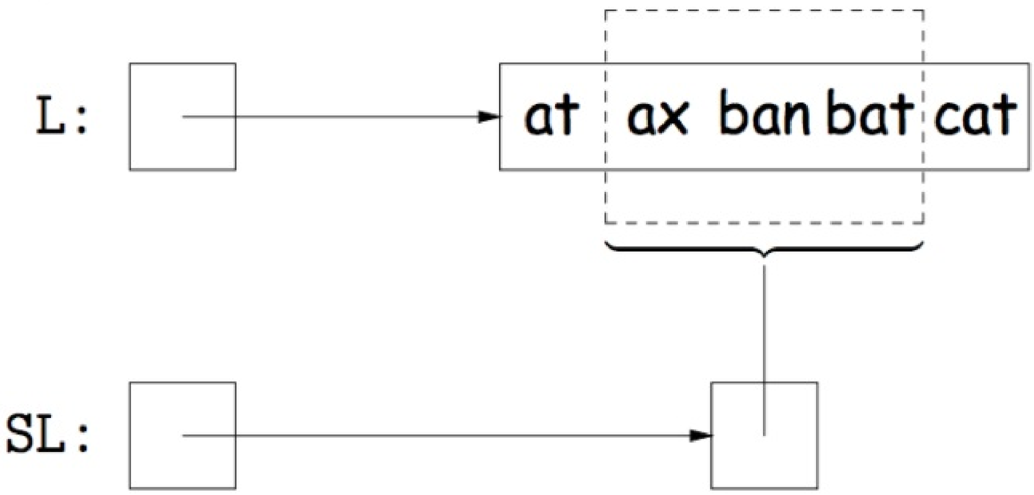
Views: Views are an alternative representation of an existed object. Views essentially limit the access that the user has to the underlying object. However, changes done through the views will affect the actual object.

```
/** Create an ArrayList. */
List<String> L = new ArrayList<>();
/** Add some items. */
L.add(“at”); L.add(“ax”); …
```

Say you only want a list from index 1 and 4. Then you can use a method called sublist do this by the following and you will

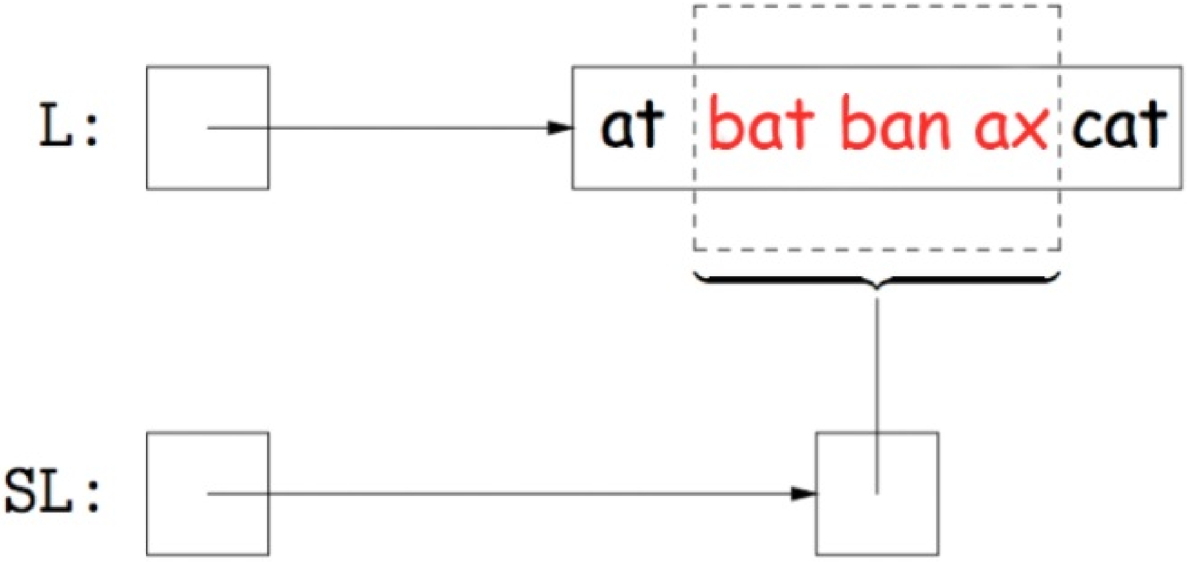
```
/** subList me up fam. */
List<String> SL = l.subList(1, 4);
/** Mutate that thing. */
SL.set(0, “jug”);
```

Now why is this useful? Well say we want to reverse only part of the list. For example in the below image, we would want to reverse ax ban bat in the above picture.



The most intuitive way is to create a method that takes in a list object and the indices which should be reversed. However, this can be a bit painful because we add some extraneous logic.

To get around doing this, we can just create a general reverse function that takes in a list and reverses that list. Because views mutates the underlying object that it represents, we can create a sublist like earlier and reverse the sublist. The end result would actually mutate the actual list and not the copy.



This is all fine and dandy. However, it lends itself to an issue. You are claiming that you can give a list object that when manipulated, can affect the original list object- that's a bit weird. Just thinking "How do you return an actual List but still have it affect another List?" is a bit confusing. Well the answer is access methods.

The first thing to notice is that the sublist method returns a list type. Additionally, there is a defined class called Sublist which extends AbstractList. Since Abstract List it implements the List interface it and Sublist are List types.

```
List<Item> sublist(int start, int end){
    Return new this.Sublist(start,end);
}
```

This first thing to notice from the above code is that subList returns a List type.

```
Private class Sublist extends AbstractList<Item>{
    Private int start end;
    Sublist(inst start, int end){...}
}
```

Now the reason the sublist function returns a List is because the class SubList extends AbstractList. Since AbstractList implements the List interface both it and Sublist are List Types.

```
public Item get(int k){return AbstractList.this.get(start+k);}
public void add(int l, Item x){AbstractList.this.add(start+k, x); end+=1}
```

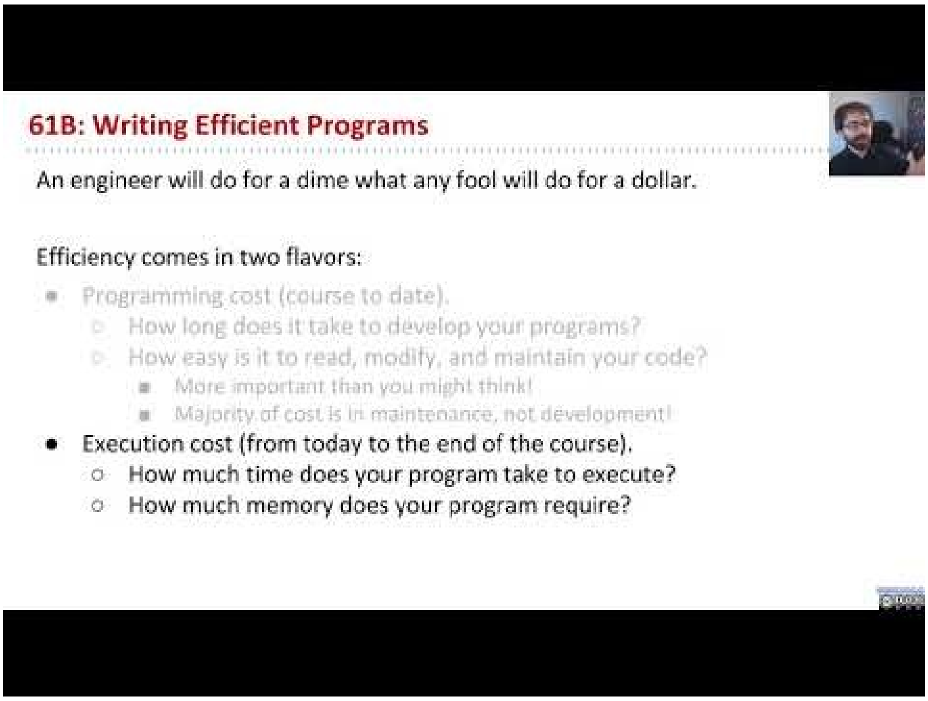
An observation that should be made is that getting the kth item from our sublist is the same as getting the the kth item from our original list with an offset equal to our start index. Because we are using a get method of our outer class (the most parent one) we change our original list.

Similarly, adding an element to our sublist is the same as adding an element to our original list with an offset equal to the start index of the sublist.

The Takeaway:

- APIs are pretty hard to design; however, having a coherent design philosophy can make your code much cleaner and easier to deal with.
- Inheritance is tempting to use frequently, but it has problems and should be use sparingly, only when you are certain about attributes of your classes (both those being extended and doing the extending).

Asymptotics I: An Introduction to Asymptotic Analysis



Video link

We can consider the process of writing efficient programs from two different perspectives:

1. Programming Cost *(everything in the course up to this date)*
   i. How long does it take for you to develop your programs?
   ii. How easy is it to read or modify your code?
   iii. How maintainable is your code? (very important — much of the cost comes from maintenance and scalability, not development!)
2. Execution Cost *(everything in the course from this point on)*
   i. **Time complexity**: How much time does it take for your program to execute?
   ii. **Space complexity**: How much memory does your program require?

Example of Algorithm Cost

Objective: Determine if a *sorted* array contains any duplicates.

Silly Algorithm: Consider ***every*** pair, returning true if any match!

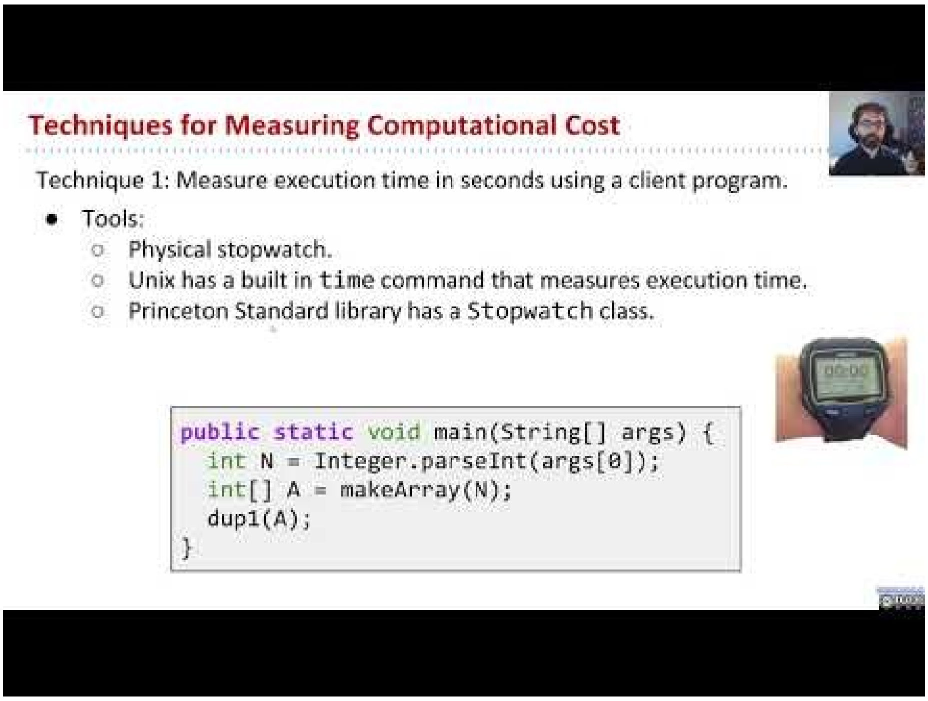
Better Algorithm: Take advantage of the ***sorted*** nature of our array.

- We know that if there are duplicates, they must be next to each other.
- Compare neighbors: return true first time you see a match! If no more items, return

false.

We can see that the Silly algorithm seems like it's doing a lot more unnecessary, redundant work than the Better algorithm. But how much more work? How do we actually quantify or determine how efficient a program is? This chapter will provide you the formal techniques and tools to compare the efficiency of various algorithms!

Runtime Characterization



Video link

To investigate these techniques, we will be characterizing the runtimes of the following two functions, dup1 and dup2. These are the two different ways of finding duplicates we discussed above.

Things to keep in mind about our characterizations:

- They should be simple and mathematically rigorous.
- They should also clearly demonstrate the superiority of dup2 over dup1.

```java
//Silly Duplicate: compare everything
public static boolean dup1(int[] A) {
  for (int i = 0; i < A.length; i += 1) {
    for (int j = i + 1; j < A.length; j += 1) {
      if (A[i] == A[j]) {
         return true;
      }
    }
  }
  return false;
}

//Better Duplicate: compare only neighbors
public static boolean dup2(int[] A) {
  for (int i = 0; i < A.length - 1; i += 1) {
    if (A[i] == A[i + 1]) {
      return true;
    }
  }
  return false;
}
```

Techniques for Measuring Computational Cost

Technique 1: Measure execution time in seconds using a client program (i.e. actually seeing how quick our program runs in physical seconds)

Procedure

- Use a physical stopwatch
- Or, Unix has a built in `time` command that measures execution time.
- Or, Princeton Standard library has a `stopwatch` class

Observations

- As our input size increases, we can see that `dup1` takes a longer time to complete, whereas `dup2` completes at relatively around the same rate.

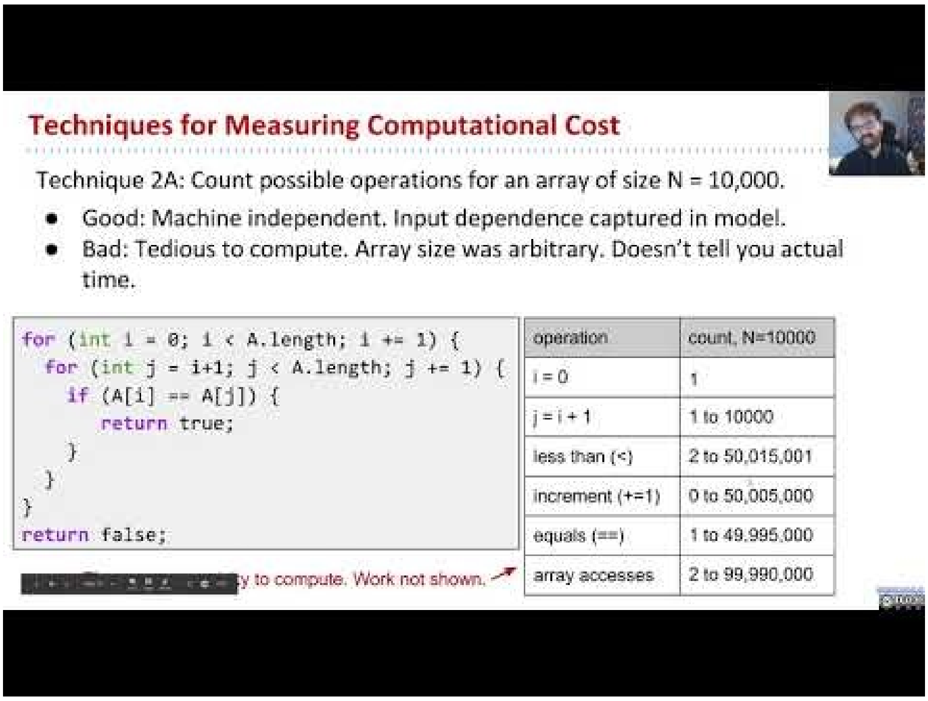
Pros vs. Cons

- Pros: Very easy to measure (just run a stopwatch). Meaning is clear (look at the actual length of time it takes to complete).
- Cons: May take a lot of time to test. Results may also differ based on what kind of machine, compiler, input data, etc. you’re running your program with.

So how does this method match our goals? It's simple, so that's good, but not mathematically rigorous. Moreover, the differences based on machine, compiler, input, etc. mean that the results may not clearly demonstrate the relationship between dup1 and dup2.

How about another method?

Technique 2



[Video link]

Technique 2A: Count possible operations for an array of size N = 10,000.

```
for (int i = 0; i < A.length; i += 1) {
  for (int j = i+1; j < A.length; j += 1) {
    if (A[i] == A[j]) {
       return true;
    }
  }
}
return false;
```

Procedure

- Look at your code and the various operations that it uses (i.e. assignments, incrementations, etc.)
- Count the number of times each operation is performed.

Observations

- Some counts get tricky to count.
- How did we get some of these numbers? It can be complicated and tedious.

Pros vs. Cons

- Pros: Machine independent (for the most part). Input dependence captured in model.
- Cons: Tedious to compute. Array size was arbitrary (we counted for N = 10,000 — but what about for larger N? For a smaller N? How many counts for those?). Number of operations doesn't tell you the actual time it takes for a certain operation to execute (some might be quicker to execute than others).

So maybe this one has solved some of our cons from the timing simulation above, but it has problems of its own.

Technique 2B: Count possible operations in terms of input array size N (symbolic counts)

Pros vs. Cons

- Pros: Still machine independent (just counting the number of operations still). Input dependence still captured in model. But now, it tells us how our algorithm scales as a function of the size of our input.
- Cons: Even more tedious to compute. Still doesn't tell us the actual time it takes!

Checkpoint: Applying techniques 2A and B to `dup2`

- Come up with counts for each operation, for the following code, with respect to N.
- Predict the ***rough*** magnitudes of each one!

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

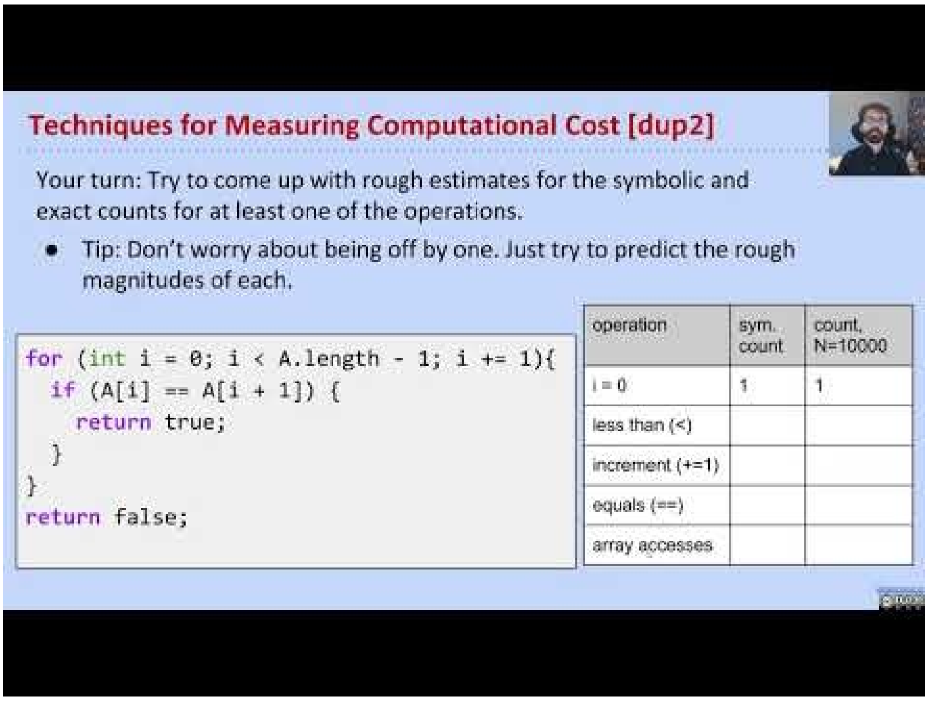
operation	symbolic count	count, N=10000
i = 0	1	1
less than (<)		
increment (+=1)		
equals (==)		
array accesses		

Answer:

operation	symbolic count	count, N=10000
i = 0	1	1
less than (<)	0 to N	0 to 10000
increment (+=1)	0 to N - 1	0 to 9999
equals (==)	1 to N - 1	1 to 9999
array accesses	2 to 2N - 2	2 to 19998

Note: It's okay if you were slightly off — as mentioned earlier, you want ***rough*** estimates.

Checkpoint



[Video link]

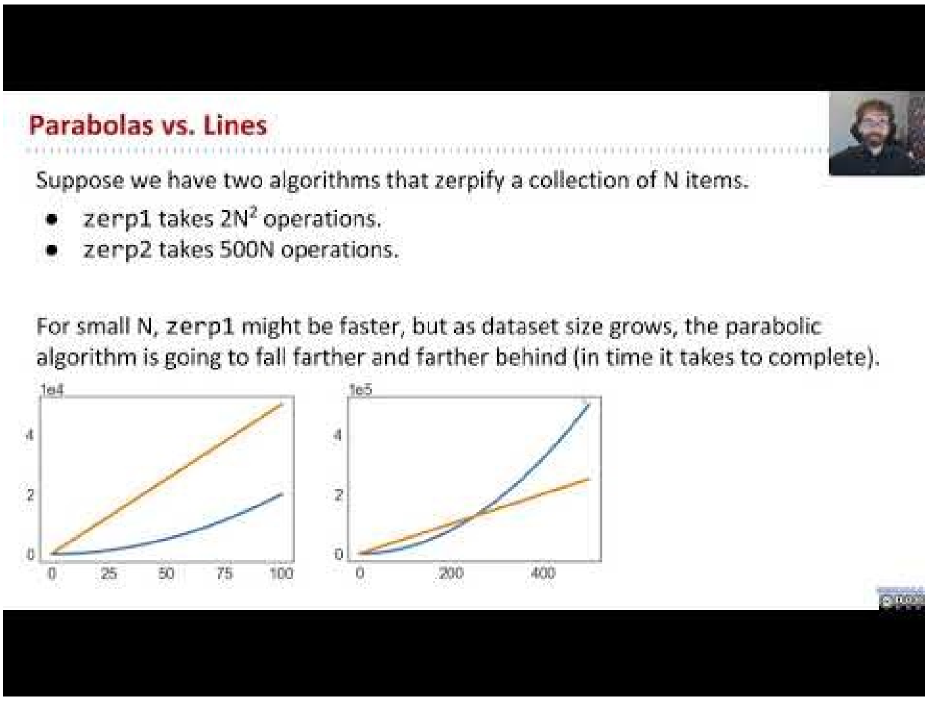
Checkpoint: Now, considering the following two filled out tables, which algorithm seems better to you and why? `dup1`

operation	symbolic count	count, N=10000
i = 0	1	1
j = i + 1	1 to N	1 to 10000
less than (<)	2 to $(N^2 + 3N + 2)/2$	2 to 50,015,001
increment (+=1)	0 to $(N^2 + N)/2$	0 to 50,005,000
equals (==)	1 to $(N^2 - N)/2$	1 to 49,995,000
array accesses	2 to $N^2 - N$	2 to 99,990,000

`dup2`

operation	symbolic count	count, N=10000
i = 0	1	1
less than (<)	0 to N	0 to 10000
increment (+=1)	0 to N - 1	0 to 9999
equals (==)	1 to N - 1	1 to 9999
array accesses	2 to 2N - 2	2 to 19998

Answer (and Why Scaling Matters)



[Video link]

`dup2` is better! But why?

- An answer: It takes fewer operations to accomplish the same goal.
- Better answer: Algorithm scales better in the worst case $(N^2 + 3N + 2)/2$ vs. N
- Even better answer: Parabolas N^2 grow faster than lines N
    - Note: This is the same idea as our "better" answer, but it provides a more general geometric intuition.

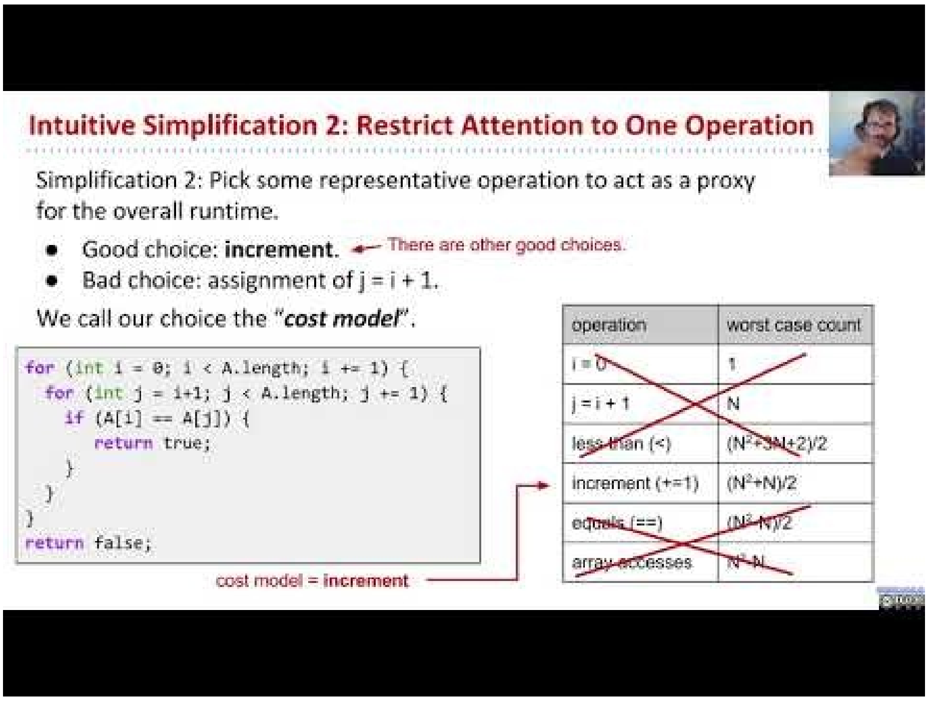
Asymptotic Behavior

In most cases, we only care about what happens for very large N (asymptotic behavior). We want to consider what types of algorithms would best handle big amounts of data, such as in the examples listed below:

- Simulation of billions of interacting particles
- Social network with billions of users
- Encoding billions of bytes of video data

Algorithms that scale well (i.e. look like lines) have better asymptotic runtime behavior than algorithms that scale relatively poorly (i.e. looks like parabolas).

Parabolas vs. Lines



Video link

What about constants? If we had functions that took $2N^2$ operations vs. $500N$ operations, wouldn't the one that only takes $2N^2$ operations be faster in certain cases, like if N = 4 (32 vs. 20,000 operations).

- Yes! For some small N, $2N^2$ may be smaller than $500N$.
- However, as N grows, the $2N^2$ will dominate.
- i.e. N = 10,000 → 2*100000000 vs. 5 * 1000000

The important thing is the "shape" of our graph (i.e. parabolic vs. linear) Let us (for now) informally refer to the shape of our graph as the "orders of growth".

Returning to Duplicate Finding

Returning to our original goals of characterizing the runtimes of `dup1` vs. `dup2`

- They should be **simple** and **mathematically rigorous.**
- They should also clearly demonstrate the **superiority of dup2 over dup1.**

We've accomplished the second task! We were able to clearly see that `dup2` performed better than `dup1`. However, we didn't do it in a very simple or mathematically rigorous way.

We did however talk about how `dup1` performed "like" a parabola, and `dup2` performed "like" a line. Now, we'll be more formal about what we meant by those statements by applying the four simplifications.

Intuitive Simplification 1: Consider only the Worst Case

When comparing algorithms, we often only care about the worst case (though we'll see some exceptions later in this course).

Checkpoint: Order of Growth Identification

Consider the counts for the algorithm below. What do you expect will be the order of growth of the runtime for the algorithm?

- N [linear]
- N^2 [quadratic]
- N^3 [cubic]
- N^6 [sextic]

operation	count
less than (<)	$100N^2 + 3N$
greater than (>)	$N^3 + 1$
and (&&)	$5,000$

Answer: It's cubic (N^3)!

- Why? Here's an argument:
- Suppose the $<$ operator takes α nanoseconds, the $>$ operator takes β nanoseconds, and && takes γ nanoseconds.
- Total time is $\alpha(100N^2 + 3N) + \beta(2N^3 + 1) + 5000\gamma$ nanoseconds.
- For very large N, the $2\beta N^3$ term is much larger than the others.
  - You can think of it in terms of calculus if it helps.
  - What happens as N approaches infinity? When it becomes super large? Which term ends up dominating?
  - Very **important** point/observation to understand why this term is much larger!

Intuitive Simplification 2: Restrict Attention to One Operation

Pick some representative operation to act as a proxy for overall runtime.

- Good choice: `increment`, or **less than** or **equals** or **array accesses**
- Bad choice: **assignment of** `j = i + 1`, or `i = 0`

The operation we choose can be called the "**cost model**."

Intuitive Simplification 3: Eliminate Low Order Terms

Ignore lower order terms!

Sanity check: Why does this make sense? (Related to the checkpoint above!)

Intuitive Simplification 4: Eliminate Multiplicative Constants

Ignore multiplicative constants.

- Why? No real meaning!

- Remember that by choosing a single representative operation, we already "threw away" some information
- Some operations had counts of $3N^2$, $N^2/2$, etc. In general, they are all in the family/shape of N^2!

This step is also related to the example earlier of $500N$ vs. $2N^2$.

Simplification Summary

- Only consider the worst case.
- Pick a representative operation (aka: cost model)
- Ignore lower order terms
- Ignore multiplicative constants.

Checkpoint: Apply these four steps to `dup2`, given the following tables.

operation	count
i = 0	1
less than (<)	0 to N
increment (+=1)	0 to N - 1
equals (==)	1 to N - 1
array accesses	2 to 2N - 2

operation	worst case orders of growth

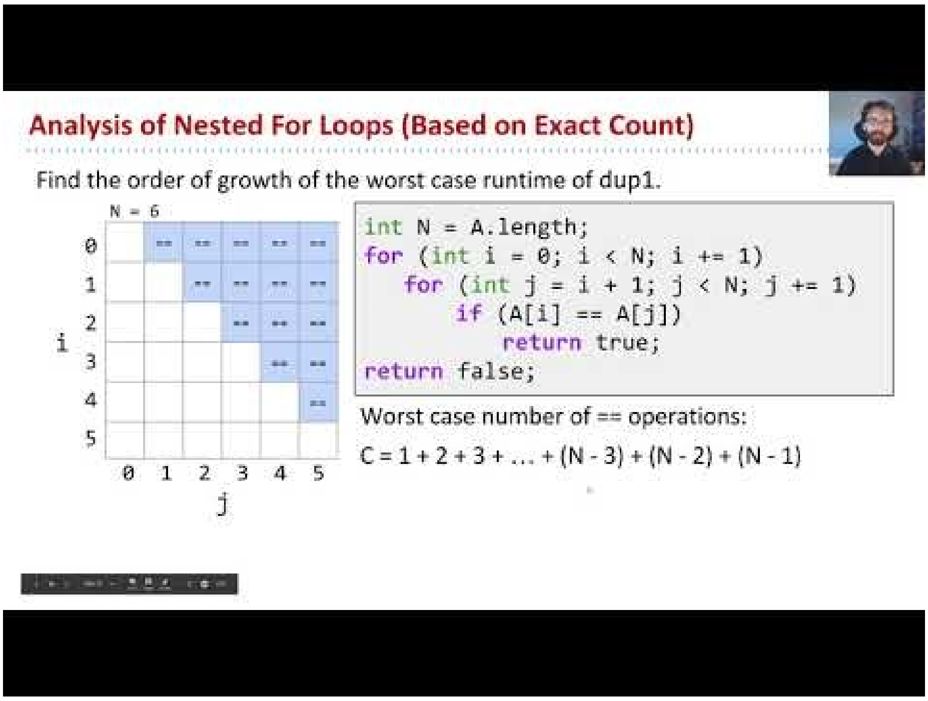
Sample Answer: `Array accesses` | `N`, or `less than/increment/equals` | `N`

Summary of our (Painful) Analysis Process

- Construct a table of exact counts of all possible operations (takes lots of effort!)
- Convert table into worst case order of growth using 4 simplifications.

But, what if we just avoided building the table from the get-go, by using our simplifications from the very start?

Simplified Analysis Process



[Video link]

Rather than building the entire table, we can instead:

- Choose our cost model (representative operation we want to count).
- Figure out the order of growth for the count of our representative operation by either:
    - Making an exact count, and discarding unnecessary pieces
    - Or, using intuition/inspection to determine orders of growth (comes with practice!)

We'll now re-analyze `dup1` using this process.

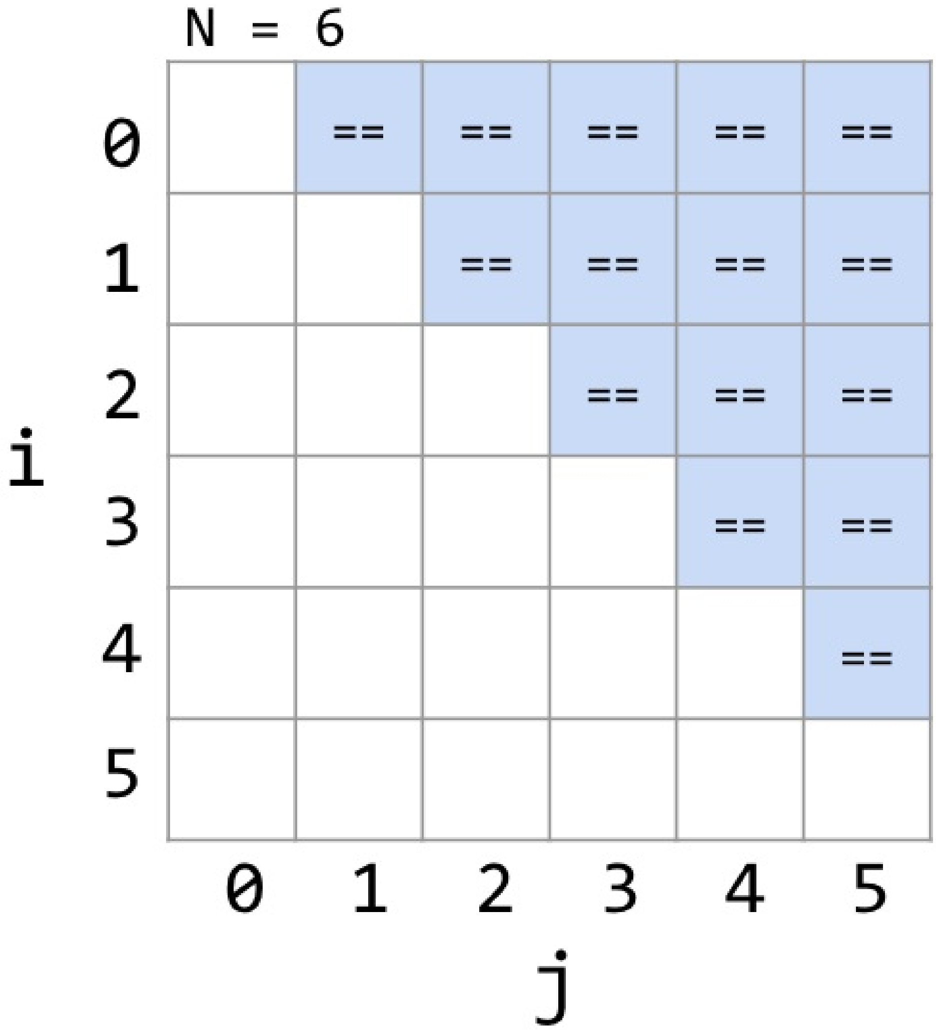
Analysis of Nested For Loops: Exact Count

Find order of growth of worst case runtime of `dup1`.

```
int N = A.length;
for (int i = 0; i < N; i += 1)
   for (int j = i + 1; j < N; j += 1)
      if (A[i] == A[j])
         return true;
return false;
```

Cost model: number of == operations

Given the following chart, how can we determine how many == occurs? The y axis represents each increment of i, and the x access represents each increment of j.



- Worst case number of == operations:
  - Cost = 1 + 2 + 3 + … + (N-2) + (N-1)
- How do we sum up this cost?
- Well, we know that it can also be written as:
  - Cost = (N-1) + (N-2) + … + 3 + 2 + 1
- Let's sum up these two Cost equations:
  - 2*Cost = N + N + N + … N
- How many N terms are there?
  - N-1! (the pairs that summed up to N, through adding the two Cost equations together)
- Therefore: 2*Cost = N(N-1)
- Therefore: Cost = N(N-1)/2
- If we do our simplification (throwing away lower order terms, getting rid of multiplicative constants), we get worst case orders of growth = N^2

Analysis of Nested For Loops: Geometric Argument

- We can see that the number of equals can be given by the area of a right triangle, which has a side length of N - 1
- Therefore, the order of growth of area is N^2
- Takes time and practice to be able to do this!

https://www.youtube.com/watch?v=sMlmXdKb9fA&list=PL8FaHk7qbOD69aH2dhqcY64VMmX7frTUO&index=8

Formalizing Order of Growth

Given some function, Q(N), we can apply our last two simplifications to get the order of growth of Q(N).

- Reminder: last two simplifications are dropping lower order terms and multiplicative constants.
- Example: $Q(N) = 3N^3 + N2$
- After applying the simplifications for order of growth, we get: N^3

Now, we'll use the formal notation of “Big-Theta" to represent how we've been analyzing our code.

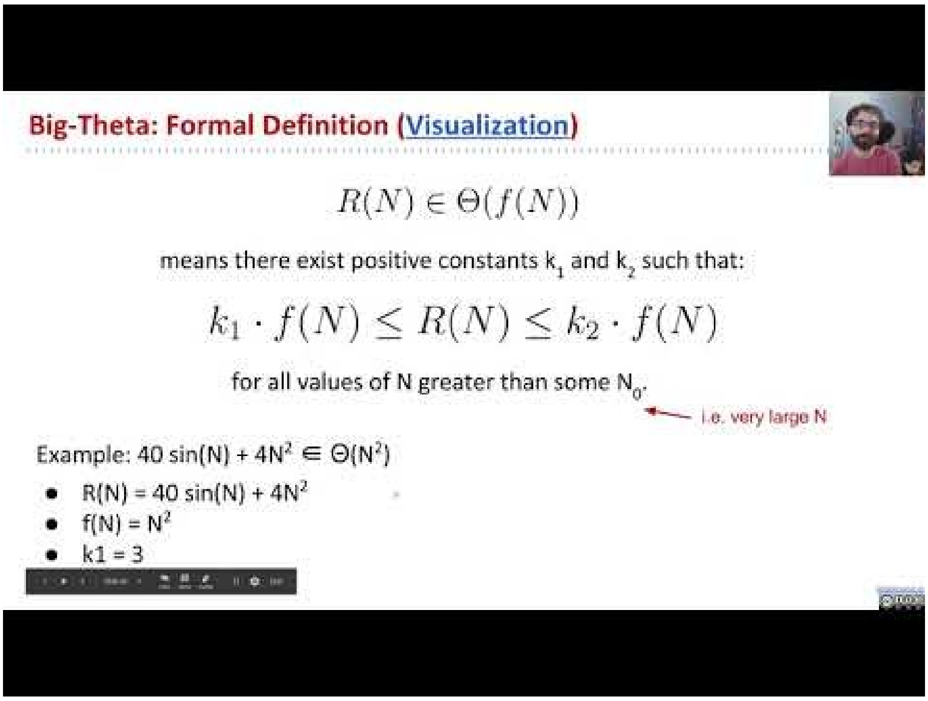
Checkpoint: What's the shape/orders of growth for the following 5 functions?

function	order of growth
$N^3 + 3N^4$	
$1/N + N^3$	
$1/N + 5$	
$Ne^N + N$	
$40sin(N) + 4N^2$	

Answer:

order of growth
N^4
N^3
1
Ne^N
N^2

Big-Theta



Video link

Suppose we have a function R(N) with order of growth f(N). In “Big-Theta” notation we write this as R(N) \in \Theta(f(N)). This notation is the formal way of representing the "families" we've been finding above.

Examples (from the checkpoint above):

- $N^3 + 3N^4 \in \Theta(N^4)$
- $1/N + N^3 \in \Theta(N^3)$
- $1/N + 5 \in \Theta(1)$

- $Ne^N + N \in \Theta(Ne^N)$
- $40sin(N) + 4N^2 \in \Theta(N^2)$

Formal Definition

$R(N) \in \Theta(f(N))$ means that there exists positive constants k_1, k_2 such that:

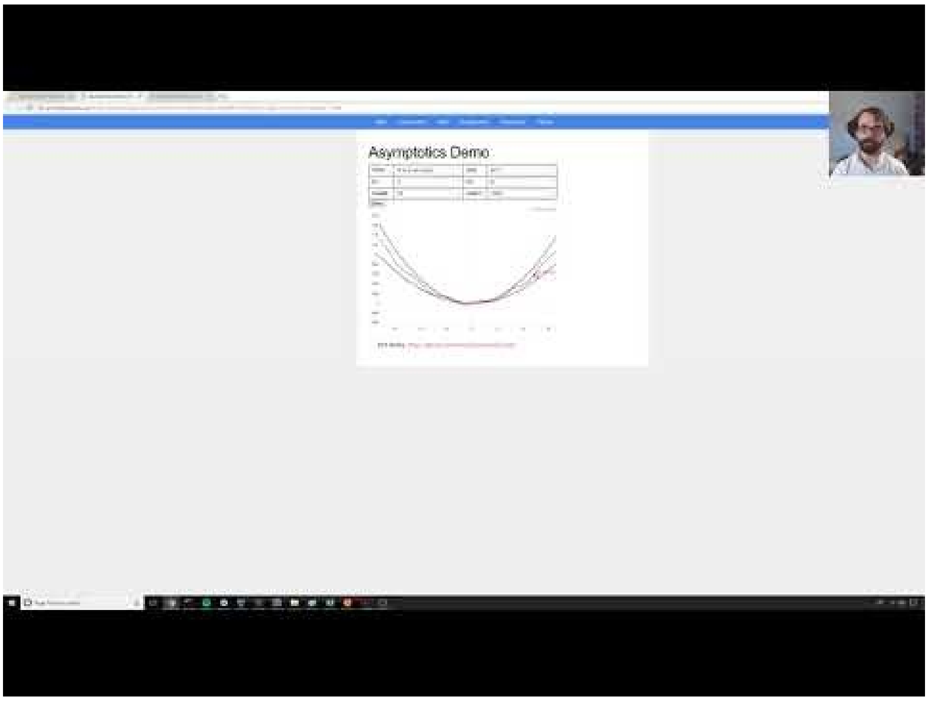
$k_1 \cdot f(N) \leq R(N) \leq k_2 \cdot f(N)$, for all values of N greater than some N_0 (a very large N).

Big-Theta and Runtime Analysis

Using this notation doesn't change anything about how we analyze runtime (no need to find the constants k_1, k_2)!

The only difference is that we use the Θ symbol in the place of "order of growth" (i.e. worst case runtime: $\Theta(N^2)$)

Big O



Video link

Earlier, we used Big Theta to describe the order of growth of functions as well as code pieces. Recall that if $R(N) \in \Theta(f(N))$, then $R(N)$ is both upper and lower bounded by $\Theta(f(N))$. Describing runtime with both an upper and lower bound can informally be though of as runtime "equality".

Some examples:

function $R(N)$	**Order of growth**
$N^3 + 3N^4$	$\Theta(N^4)$
$N^3 + 1/N$	$\Theta(N^3)$
$5 + 1/N$	$\Theta(1)$
$Ne^N + N$	$\Theta(Ne^N)$
$40\sin(N) + 4N^2$	$\Theta(N^2)$

For example, $N^3 + 3N^4 \in \Theta(N^4)$. It is both upper and lower bounded by N^4.

On the other hand, Big O can be though of as a runtime inequality, namely, as "less than or equal". For example, all of the following are true: $N^3 + 3N^4 \in O(N^4)$ $N^3 + 3N^4 \in O(N^6)$ $N^3 + 3N^4 \in O(N!)$ $N^3 + 3N^4 \in O(N^{N!})$

In other words, if a function, like the one above, is upper bounded by N^4, then it is also upper bounded by functions that themselves upper bound N^4. $N^3 + 3N^4$ is "less than or equal to" all of these functions in the asymptotic sense.

Recall the formal definition of Big Theta: $R(N) \in \Theta(f(N))$ means that there exists positive constants k_1, k_2 such that:

$k_1 \cdot f(N) \leq R(N) \leq k_2 \cdot f(N)$ for all values of N greater than some N_0 (a very large N).

Formal Definition

Similarly, here's the formal definition of Big O: $R(N) \in O(f(N))$ means that there exists positive constants k_2 such that:

$R(N) \leq k_2 \cdot f(N)$ for all values of N greater than some N_0 (a very large N).

Observe that this is a looser condition than Big Theta since Big O does not care about the lower bound.

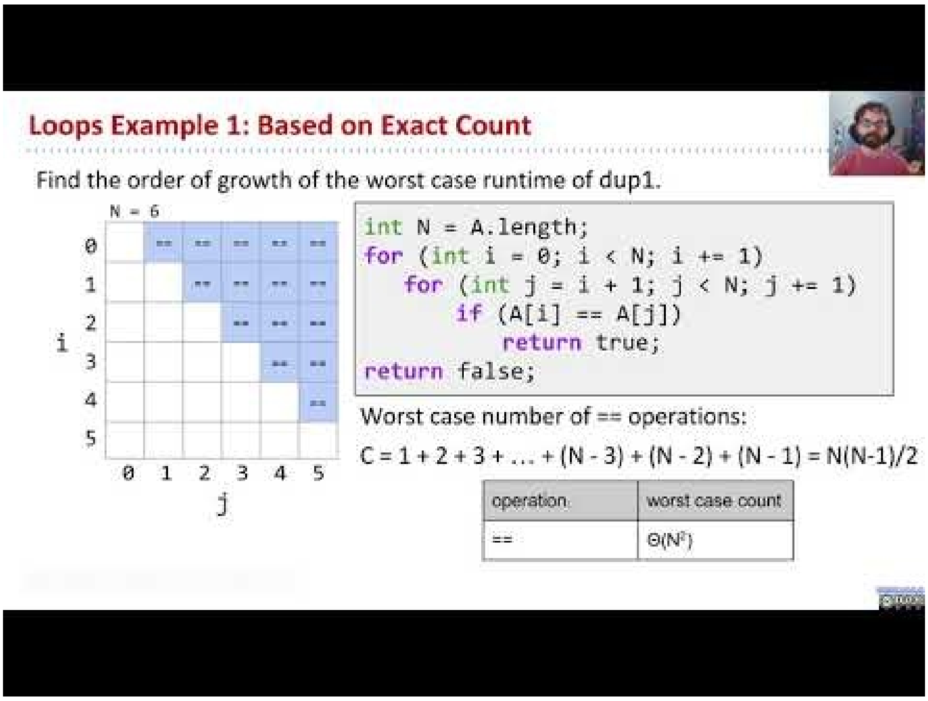
Summary

- Given a piece of code, we can express its runtime as a function R(N)
  - N is some **property** of the input of the function
  - i.e. oftentimes, N represents the **size** of the input
- Rather than finding R(N) exactly, we instead usually only care about the **order of growth** of R(N).
- One approach (not universal):
  - Choose a representative operation
  - Let C(N) = count of how many times that operation occurs, as a function of N.
  - Determine order of growth $f(N)$ for $C(N)$, i.e. $C(N) \in \Theta(f(N))$
  - Often (but not always) we consider the worst case count.
  - If operation takes constant time, then $R(N) \in \Theta(f(N))$

Asymptotics II

A first example with Loops

Now that we've seen some runtime analysis, let's work through some more difficult examples. Our goal is to get some practice with the patterns and methods involved in runtime analysis. This can be a tricky idea to get a handle on, so the more practice the better.



[Video link]

Last time, we saw the function dup1, that checks for the first time any entry is duplicated in a list:

```
int N = A.length;
for (int i = 0; i < N; i += 1)
   for (int j = i + 1; j < N; j += 1)
      if (A[i] == A[j])
         return true;
return false;
```

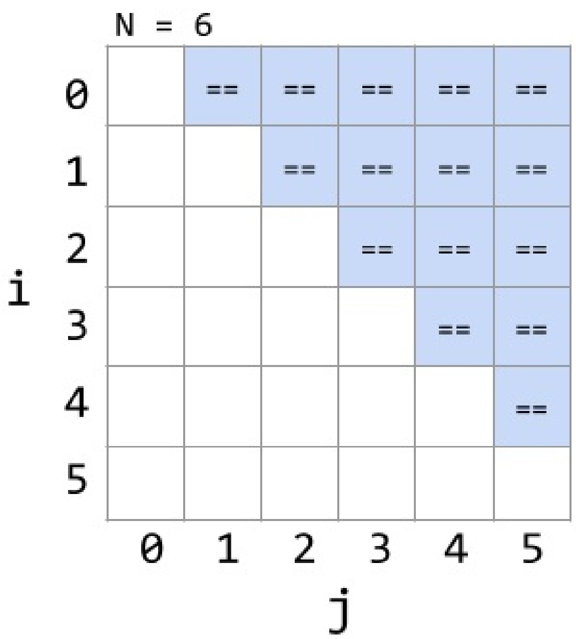
We have two ways of approaching our runtime analysis: first, by counting the number of operations; second, a geometric argument.

First method: Since the main repeating operation is the comparator, we will count the number of == operations that must occur. The first time through the outer loop, the inner loop will run N-1 times. The second time, it will run N-2 times. Then N-3... In the worst case, we have to go through every entry (the outer loop runs N times).

In the end, we see that the number of comparisons is:
$C = 1 + 2 + 3 + ... + (N-3) + (N-2) + (N-1) = N(N-1)/2$

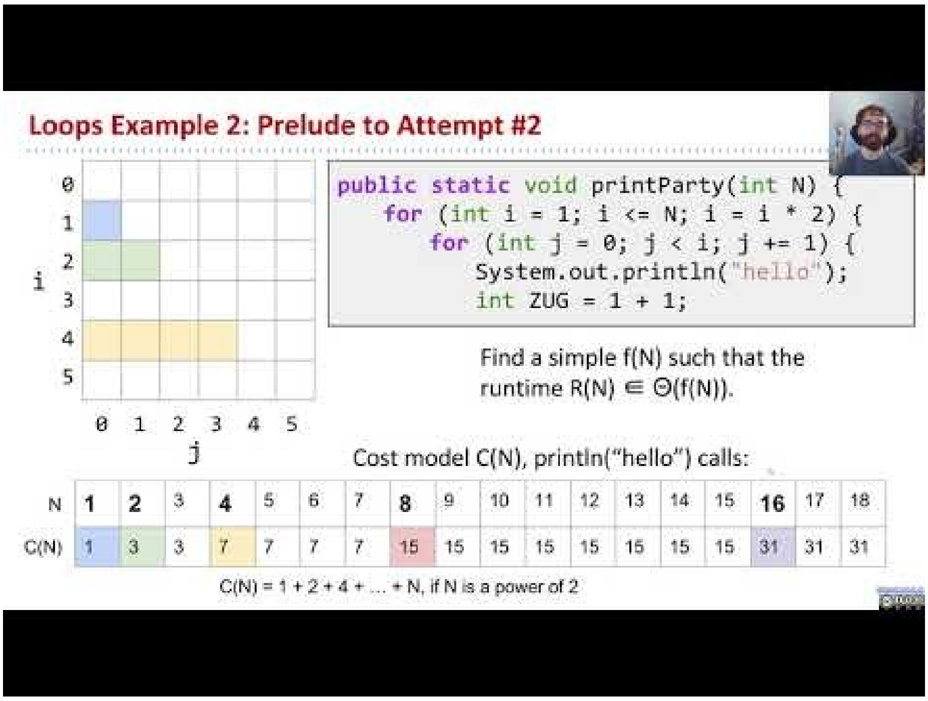
$N(N-1)/2$ is of the family N^2. Since == is a constant time operation, the overall runtime in the worst case is $\theta(N^2)$.

Second method: We can also approach this from a geometric view. Let's draw out when we use == operations in the grid of i,j combinations:



We see that the number of == operations is the same as the area of a right triangle with a side length of $N-1$. Since area is in the N^2 family, we see again that the overall runtime is $\theta(N^2)$.

Loop Example 2



[Video link]

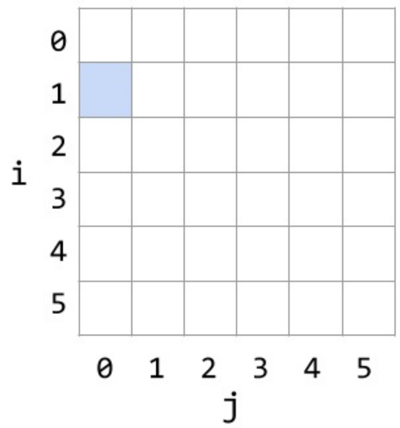
Let's look at a more involved example next. Consider the following function, with similar nested for loops:

```
public static void printParty(int N) {
   for (int i = 1; i <= N; i = i * 2) {
      for (int j = 0; j < i; j += 1) {
         System.out.println("hello");
         int ZUG = 1 + 1;
      }
   }
}
```

The first loop advances by *multiplying* `i` by 2 each time. The inner loop runs from 0 to the current value of `i`. The two operations inside the loop are both constant time, so let's approach this by asking "how many times does this print out "hello" for a given value of N?"

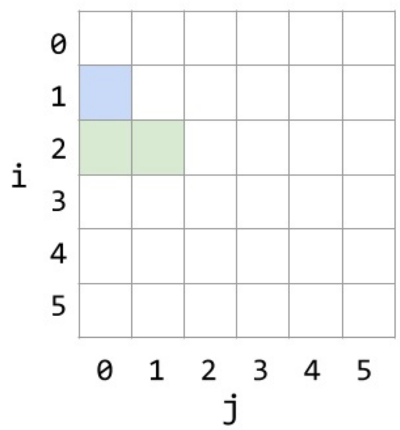
Our visualization tool from above helped us see dup1's runtime, so let's use a similar approach here. We'll lay out the grid for the nested for loops, and then track the total number of print statements needed for a given N below.

If N is 1, then `i` only reaches 1, and `j` is only 0, since 0 < 1. So there is only one print statement:



N	**1**	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
C(N)	1																	

If N is 2, the next time through the loop `i` will be $1*2=2$, and `j` can reach 1. The total number of print statements will be 3: 1 for the first loop plus 2 for the second time through.



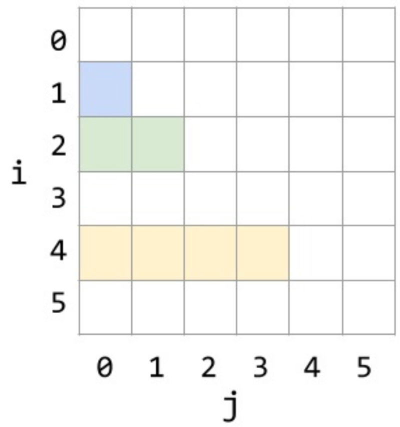
N	**1**	**2**	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
C(N)	1	3																

What happens when N is 3? Does `i` go through another loop?

...

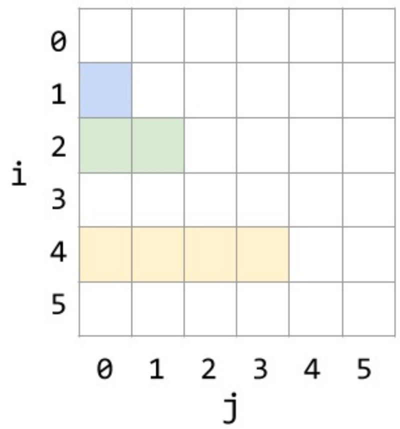
Well, after the second loop, $i=2*2=4$, which is greater than N, so the outer loop does not continue, and ends after `i = 2`, just like N = 2 did. N = 3 will have the same number of print statements as N = 2.

The next change is at N=4, where there will be 4 prints when i = 4, 3 prints when i = 2, and 1 print when i = 1 (remember `i` never equals 3). So a total of 7.



N	**1**	**2**	3	**4**	5	6	7	8	9	10	11	12	13	14	15	16	17	18
C(N)	1	3	3	7														

We can keep filling out our diagram to get a fuller picture. Here it is up to N = 18:

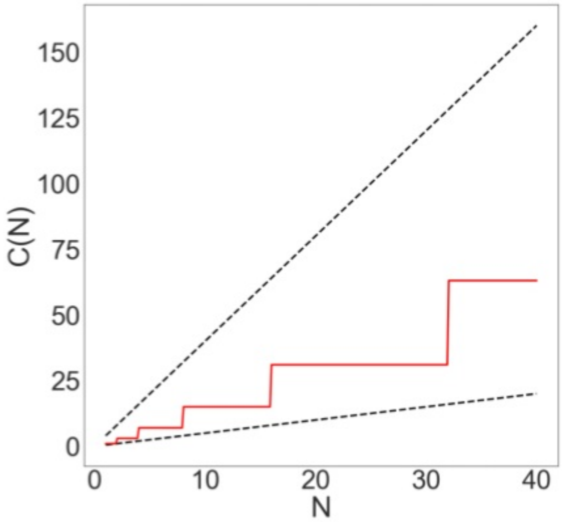


N	**1**	**2**	3	**4**	5	6	7	**8**	9	10	11	12	13	14	15	**16**	17	18
C(N)	1	3	3	7	7	7	7	15	15	15	15	15	15	15	15	31	31	31

What we see, if we add up all the counts at each stage of the loops, is that the number of print statements is: C(N) = 1 + 2 + 4 + ... + N (if N is a power of 2).

But what does this mean for the runtime?

Again, we can think of this in a couple ways. Since we're already on a graphical roll, let's start there. If we graph the trajectory of 0.5 N (lower dashed line), and 4N (upper dashed line), and C(N) itself (the red staircase line) we see that C(N) is fully bounded between those two dashed lines.



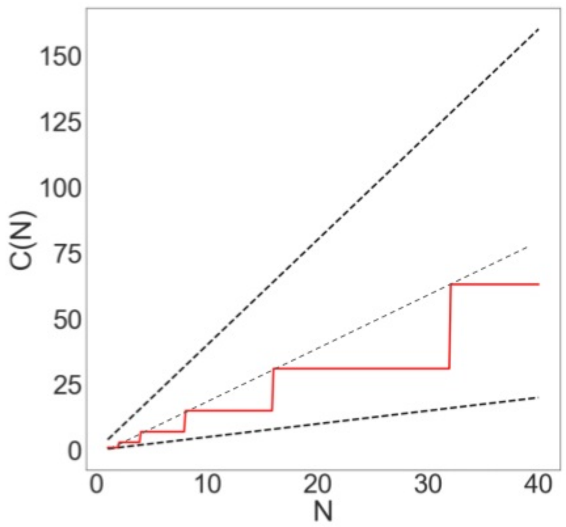
Therefore, the runtime (by definition) must also be linear!

Let's look at this another way as well:

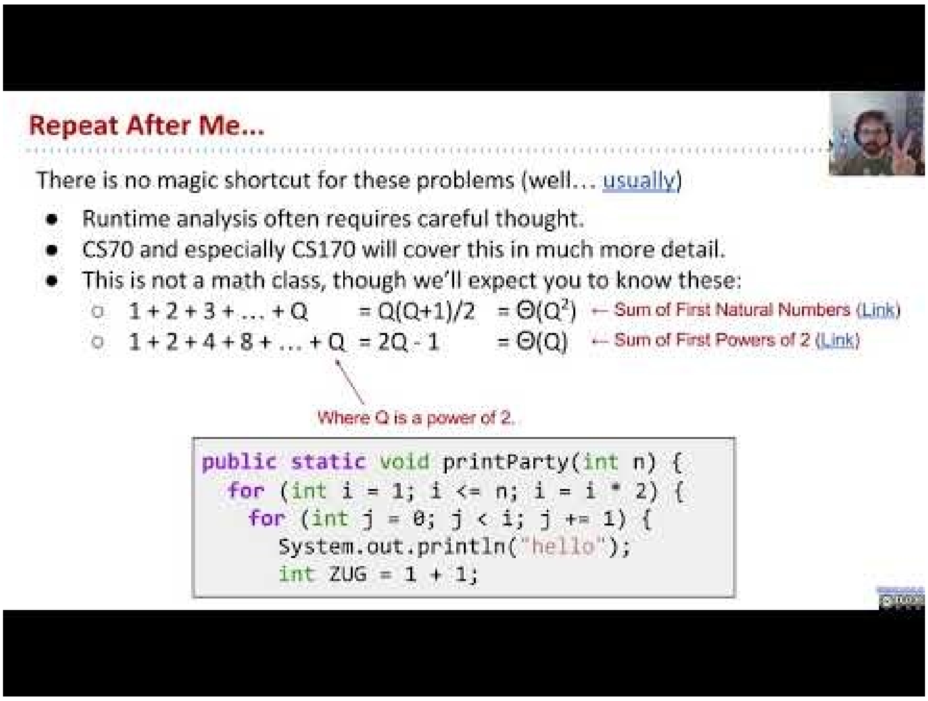
We can solve our equation from above with the power of mathematics, and find that: $C(N) = 1 + 2 + 4 + ... + N = 2N - 1$ (again if N is a power of 2). For example, If N = 8 $1 + 2 + 4 + 8 = 15 = 2 * 8 - 1$

And by removing lesser terms and multiplicative constants, we know that 2N - 1 is in the linear family.

We can also see this on our graph by plotting 2N:



There is no magic shortcut :(



Video link

It would be really nice if there were some magic way to look at an algorithm and just *know* its runtime. It would be super convenient if all nested for loops were N^2. They're not. And we know this because we just did two nested for loop examples above, each with different runtimes.

In the end, there is no shortcut to doing runtime analysis. It requires careful thought. But there are a few useful techniques and things to know.

Techniques:

- *Find exact sum*
- *Write out examples*
- *Draw pictures*

We used each of these in the examples above.

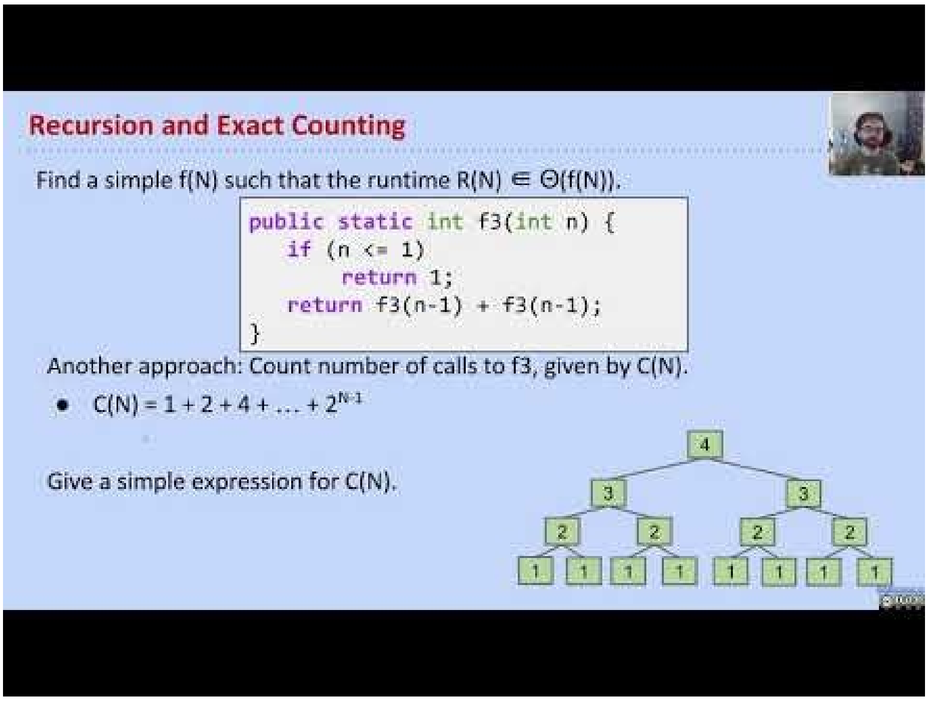
Sum Things to Know Here are two important sums you'll see quite often, and should remember:

$1 + 2 + 3 + ... + Q = Q(Q + 1)/2 = \Theta(Q^2)$ (Sum of First Natural Numbers)

$1 + 2 + 4 + 8 + ... + Q = 2Q - 1 = \Theta(Q)$ (Sum of First Powers of 2)

You saw both of these above, and they'll return again and again in runtime analysis.

Recursion



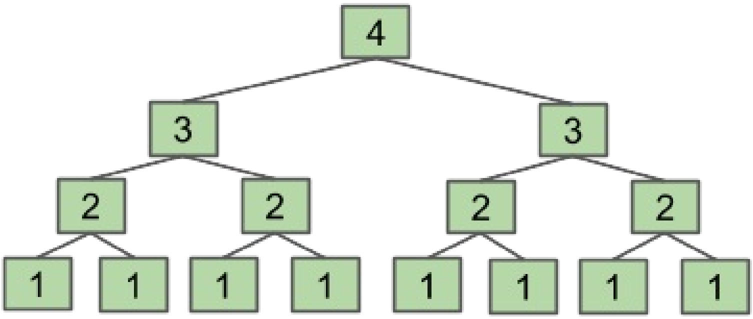
[Video link]

Now that we've done a couple of nested for loops, let's take a look at a recursive example. Consider the function `f3` :

```
public static int f3(int n) {
   if (n <= 1)
      return 1;
   return f3(n-1) + f3(n-1);
}
```

What does this function do?

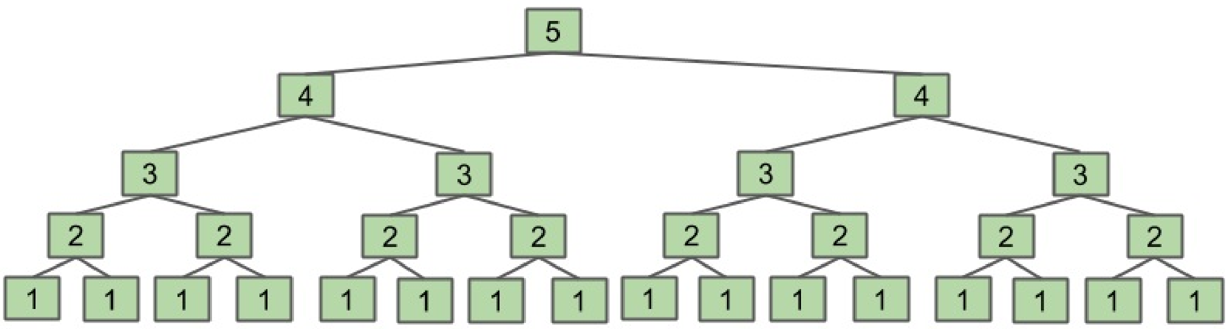
Let's think of an example. If we call `f3(4)`, it will return `f3(4-1) + f3(4-1)` which are each `f3(3-1) + f3(3-1)`, which are each `f3(2-1) + f3(2-1)`, which each return 1. So we see that in the end we have `f3(2-1)` summed 8 times, which equals 8. We can visualize this as a tree, where each level is the argument to the function:



You can do a couple more examples, and see that this function returns 2^{N-1}. This is useful for getting a sense of what the function is doing.

The Intuitive Method

Now, let's think about the runtime. We can notice that every time we add one to N, we double the amount of work that has to be done:



This intuitive argument shows that the runtime is 2^N.

This is a pretty good argument, but let's work this example a couple more ways.

The Algebraic Method

The second way to approach this problem is to count the number of calls to `f3` involved. For instance:

$C(1) = 1$ $C(2) = 1 + 2$ $C(3) = 1 + 2 + 4$ $C(N) = 1 + 2 + 4 + ... +$???

How do we generalize this last case? A useful approach is to do another couple examples. What is C(4)?

Well, we can look at our helpful diagram for `f3(4)` above, and see that the final row has 8 boxes, so:

$C(4) = 1 + 2 + 4 + 8$ and $C(5) = 1 + 2 + 4 + 8 + 16$

The final term in each of these is equal to 2^{N-1}, for example: $16 = 2^{5-1}$, $8 = 2^{4-1}$ ...

Our general form then is: $C(N) = 1 + 2 + 4 + ... + 2^{N-1}$

And this should start to look a bit familiar. Above we saw the **sum of the first powers of 2**: $1 + 2 + 4 + 8 + ... + Q = 2Q - 1$

In this case, $Q = 2^{N-1}$.

So, $C(N) = 2Q - 1 = 2(2^{N-1}) - 1 = 2^N - 1$

The work during each call is constant (not including recursive work), so this is $\theta(2^N)$.

Recurrence Relations

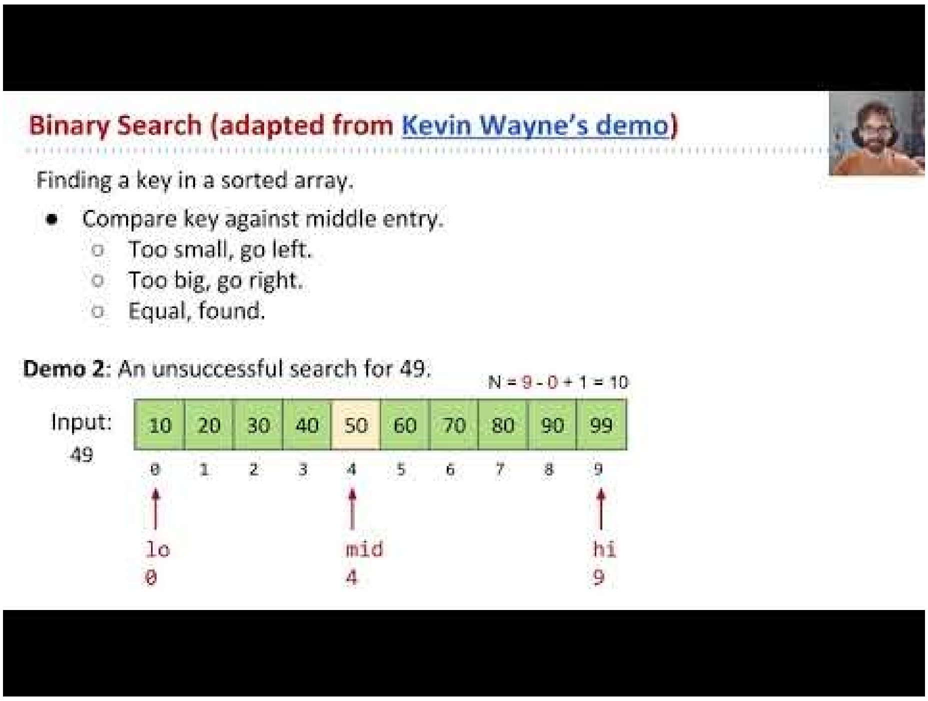
This method is not required reading and is outside of the course scope, but worth mentioning for interest's sake.

We can use a "recurrence relation" to count the number of calls, instead of an algebraic approach. This looks like:

$C(1) = 1$ $C(N) = 2C(N-1) + 1$

Expanding this out with a method we will not go over but you can read about in the slides or online, we reach a similar sum to the one above, and can again reduce it to $2^N - 1$, reaching the same result of $\theta(2^N)$.

Binary Search



Video link

Binary search is a nice way of searching a list for a particular item. It requires the list to be in sorted order, and uses that fact to find an element quickly.

To do a binary search, we start in the middle of the list, and check if that's our desired element. If not, we ask: is this element bigger or smaller than our element?

If it's bigger, then we know we only have to look at the half of the list with smaller elements. If it's too small, then we only look at the half with bigger elements. In this way, we can cut in half the number of options we have left at each step, until we find it.

What's the worst possible case? When the element we want isn't in the list at all. Then we will make comparisons until we've eliminated all regions of the list, and there's no more bigger or smaller halves left.

For an animation of binary search, see these slides.

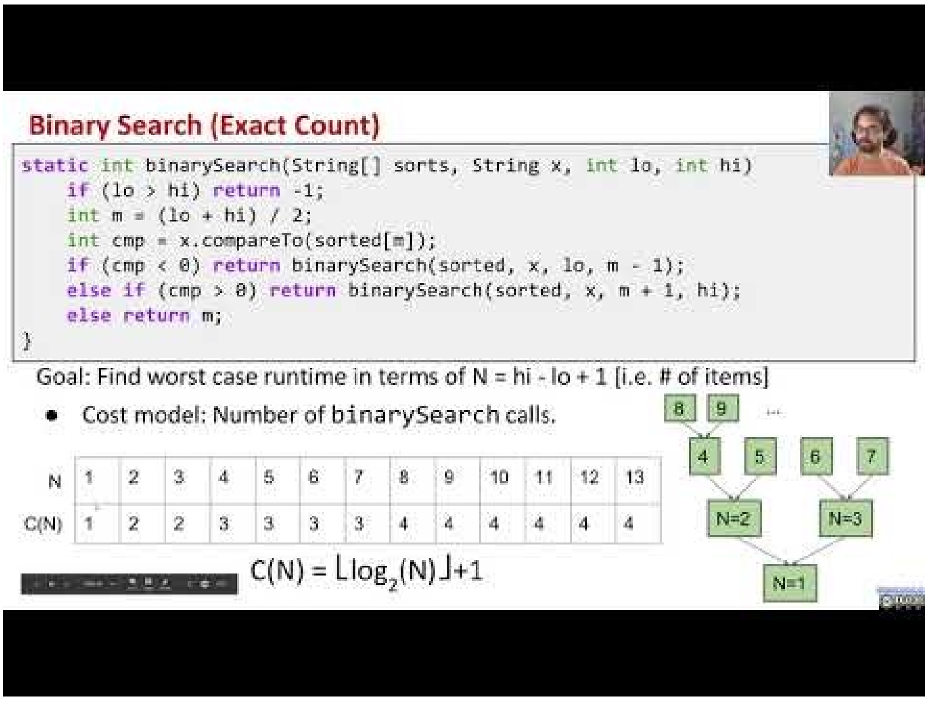
What's the intuitive runtime of binary search? Take a minute and use the tools you know to consider this.

...

We start with n options, then n/2, then n/4 ... until we have just 1. Each time, we cut the array in half, so in the end we must perform a total of $\log_2(n)$ operations. Each of the $\log_2(n)$ operations, eg. finding the middle element and comparing with it, takes constant time. So the overall runtime then is order $\log_2(n)$.

It's important to note, however that each step doesn't cut it *exactly* in half. If the array is of even length, and there is no 'middle', we have to take either a smaller or a larger portion. But this is a good intuitive approach.

We'll do a precise way next.



[Video link](#)

To precisely calculate the runtime of binary search, we'll count the number of operations, just as we've done previously.

First, we define our cost model: let's use the number of recursive binary search calls. Since the number of operations inside each call is constant, the number of calls will be the only thing varying based on the size of the input, so it's a good cost model.

Like we've seen before, let's do some example counts for specific N. As an exercise, try to fill this table in before continuing:

N	1	2	3	4	5	6	7	8	9	10	11	12	13
Count													

Alright, here's the result:

N	1	2	3	4	5	6	7	8	9	10	11	12	13
Count	1	2	2	3	3	3	3	4	4	4	4	4	4

These seems to support our intuition above of $\log_2(n)$. We can see that the count seems to increase by one only when N hits a power of 2.

...but we can be even more precise: $C(N) = \lfloor log_2(N) \rfloor + 1$ (These L-shaped bars are the "floor" function, which is the result of the expression rounded down to the nearest integer.)

A couple properties worth knowing (see below for proofs): $\lfloor f(N) \rfloor = \Theta(f(N))$

$\lceil f(N) \rceil = \Theta(f(N))$ $\log_p(N) = \Theta(\log_q(N))$

The last one essentially states that for logarithmic runtimes, the base of the logarithm doesn't matter at all, because they are all equivalent in terms of Big-O (this can be seen by applying the logarithm change of base). Applying these simplifications, we see that

$\Theta(\lfloor \log_2(N) \rfloor) = \Theta(\log N)$

just as we expected from our intuition.

Example Proof: Prove $\lfloor f(N) \rfloor = \Theta(f(N))$ **Solution:**
$f(N) - 1/2 < f(N) \leq \lfloor f(N) + 1/2 \rfloor \leq f(N) + 1/2$ Simplifying $f(N) + 1/2$ and $f(N) - 1/2$ according to our big theta rules by dropping the constants, we see that they are of order $f(N)$. Therefore $\lfloor f(N) + 1/2 \rfloor$ is bounded by two expressions of order $f(N)$, and is therefore also $\Theta(f(N))$

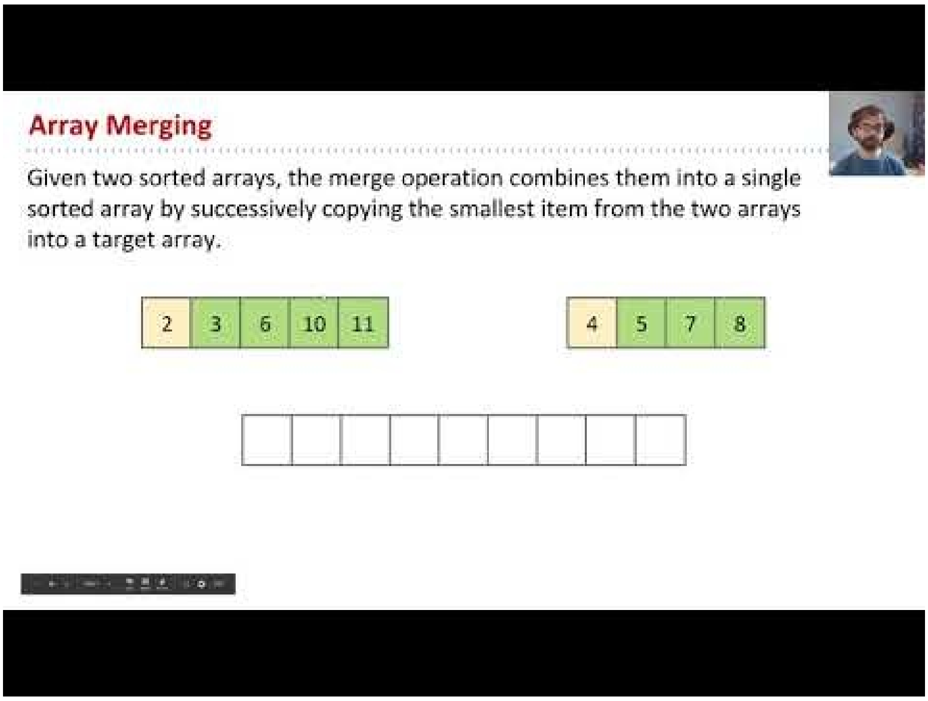
Exercise: Prove $\lceil f(N) \rceil = \Theta(f(N))$ **Exercise:** Prove $\log_p(N) = \Theta(\log_q(N))$

One cool fact to wrap up with: Log time is super good! It's almost as fast as constant time, and way better than linear time. This is why we like binary search, rather than stepping one by one through our list and looking for the right thing.

To show this concretely:

N	$\log_2 N$	Typical runtime (nanoseconds)
100	6.6	1
100,000	16.6	2.5
100,000,000	26.5	4
100,000,000,000	36.5	5.5
100,000,000,000,000	46.5	7

Merge Sort



Video link

In our last example, we'll analyze merge sort, another cool sorting algorithm.

First, let's remind ourselves of selection sort, which we will initially use as a building block for merge sort.

Selection sort works off two basic steps:

- Find the smallest item among the unsorted items, move it to the front, and ‘fix’ it in place.
- Sort the remaining unsorted/unfixed items using selection sort.

If we analyze selection sort, we see that it's $\Theta(N^2)$.

Exercise: To convince yourself that selection sort has $\Theta(N^2)$ runtime, work through the geometric approach (try drawing out the state of the list at every sort call), or count the operations.

Let's introduce one other idea here: **arbitrary units of time**. While the exact time something will take will depend on the machine, on the particular operations, etc., we can get a general sense of time through our arbitrary units (AU).

If we run an N=6 selection sort, and the runtime is order N^2, it will take ~36 AU to run. If N=64, it'll take ~2048 AU to run. Now we don't know if that's 2048 nanoseconds, or seconds, or years, but we can get a relative sense of the time needed for each size of N.

Hold onto this thought for later analysis.

Now that we have selection sort, let's talk about **merging.**

Say we have two **sorted** arrays that we want to combine into a single big sorted array. We could append one to the other, and then re-sort it, but that doesn't make use of the fact that each individual array is already sorted. How can we use this to our advantage?

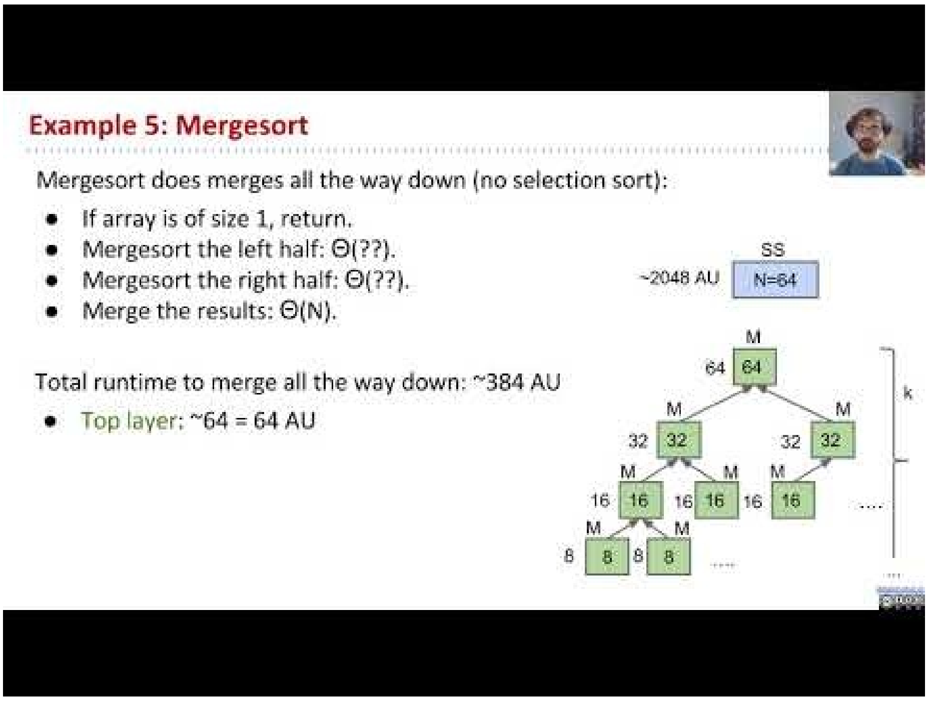
It turns out, we can merge them more quickly using the sorted property. The smallest element must be at the start of one of the two lists. So let's compare those, and put the smallest element at the start of our new list.

Now, the next smallest element has to be at the new start of one of the two lists. We can continue comparing the first two elements and moving the smallest into place until one list is empty, then copy the rest of the other list over into the end of the new list.

To see an animation of this idea, go here.

What is the runtime of merge? We can use the number of "write" operations to the new list as our cost model, and count the operations. Since we have to write each element of each list only once, the runtime is $\Theta(N)$.

Selection sort is slow, and merging is fast. How do we combine these to make sorting faster?



Video link

We noticed earlier that doing selection sort on an N=64 list will take ~2048 AU. But if we sort a list half that big, N=32, it only takes ~512 AU. That's more than twice as fast! So making the arrays we sort smaller has big time savings.

Having two sorted arrays is a good step, but we need to put them together. Luckily, we have merge. Merge, being of linear runtime, only takes ~64 AU. So in total, splitting it in half, sorting, then merging, only takes 512 + 512 + 64 = 1088 AU. Faster than selection sorting the whole array. But how much faster?

Now, AUs aren't real units, but they're sometimes easier and more intuitive than looking at the runtime. The runtime for our split-in-half-then-merge-them sort is $N + 2(N/2)^2$, which is about half of N^2 for selection sort. However, they are still both $\Theta(N^2)$.

What if we halved the arrays again? Will it get better? Yes! If we do two layers of merges, starting with lists of size N/4, the total time will be ~640 AU.

Exercise: Show why the time is ~640AU by calculating the time to sort each sub-list and then merge them into one array.

What if we halved it again? And again? And again?

Eventually we'll reach lists of size 1. At that point, we don't even have to use selection sort, because a list with one element is already sorted.

This is the essence of **merge sort:**

- If the list is size 1, return. Otherwise:
- Mergesort the left half
- Mergesort the right half
- Merge the results

So what's the running time of **merge sort**?

We know merge itself is order N, so we can start by looking at each layer of merging:

- To get the top layer: merge ~64 elements = 64 AU
- Second layer: merge ~32 elements, twice = 64 AU
- Third layer: ~16*4 = 64 AU
- ...

Overall runtime in AU is ~64*k, where k is the number of layers. Here, $k = \log_2(64) = 6$, so the overall cost of mergesort is ~384 AU.

Now, we saw earlier that splitting up more layers was faster, but still order N^2. Is merge sort faster than N^2?

...

Yes!
Mergesort has worst case runtime = $\Theta(N \log N)$.

- The top level takes ~N AU.
- Next level takes ~N/2 + ~N/2 = ~N.
- One more level down: ~N/4 + ~N/4 + ~N/4 + ~N/4 = ~N.

Thus, total runtime is ~Nk, where k is the number of levels.

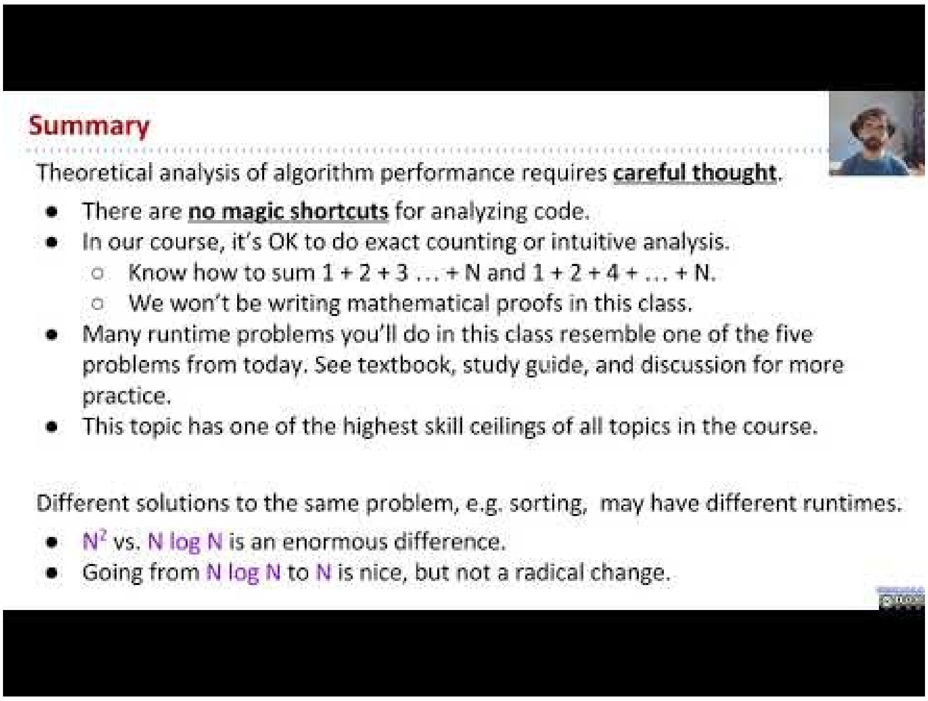
How many levels are there? We split the array until it is length 1, so $k = log_2(N)$. Thus the overall runtime is $\Theta(NlogN)$.

Exercise: Use exact counts to argue for $\Theta(NlogN)$. Account for cases where we cannot divide the list perfectly in half.

So is $\Theta(NlogN)$ actually better than $\Theta(N^2)$? Yes! It turns out $\Theta(NlogN)$ is not much slower than linear time.

	n	$n \log_2 n$	n^2	n^3	1.5^n	2^n	$n!$
$n = 10$	< 1 sec	< 1 sec	< 1 sec	< 1 sec	< 1 sec	< 1 sec	4 sec
$n = 30$	< 1 sec	< 1 sec	< 1 sec	< 1 sec	< 1 sec	18 min	10^{25} years
$n = 50$	< 1 sec	< 1 sec	< 1 sec	< 1 sec	11 min	36 years	very long
$n = 100$	< 1 sec	< 1 sec	< 1 sec	1 sec	12,892 years	10^{17} years	very long
$n = 1{,}000$	< 1 sec	< 1 sec	1 sec	18 min	very long	very long	very long
$n = 10{,}000$	< 1 sec	< 1 sec	2 min	12 days	very long	very long	very long
$n = 100{,}000$	< 1 sec	2 sec	3 hours	32 years	very long	very long	very long
$n = 1{,}000{,}000$	1 sec	20 sec	12 days	31,710 years	very long	very long	very long

Wrapup



Video link

Takeaways

- There are no magic shortcuts for analyzing code runtime.
- In our course, it's OK to do exact counting or intuitive analysis.
- Know how to sum 1 + 2 + 3 + ... + N and 1 + 2 + 4 + ... + N.
- We won't be writing mathematical proofs in this class.
- Many runtime problems you'll do in this class resemble one of the five problems from today. See textbook, study guide, and discussion for more practice.
- This topic has one of the highest skill ceilings of all topics in the course. All the tools are here, but **practice** is your friend!

- Different solutions to the same problem, e.g. sorting, may have different runtimes (with big enough differences for the runtime to go from impractical to practical!).

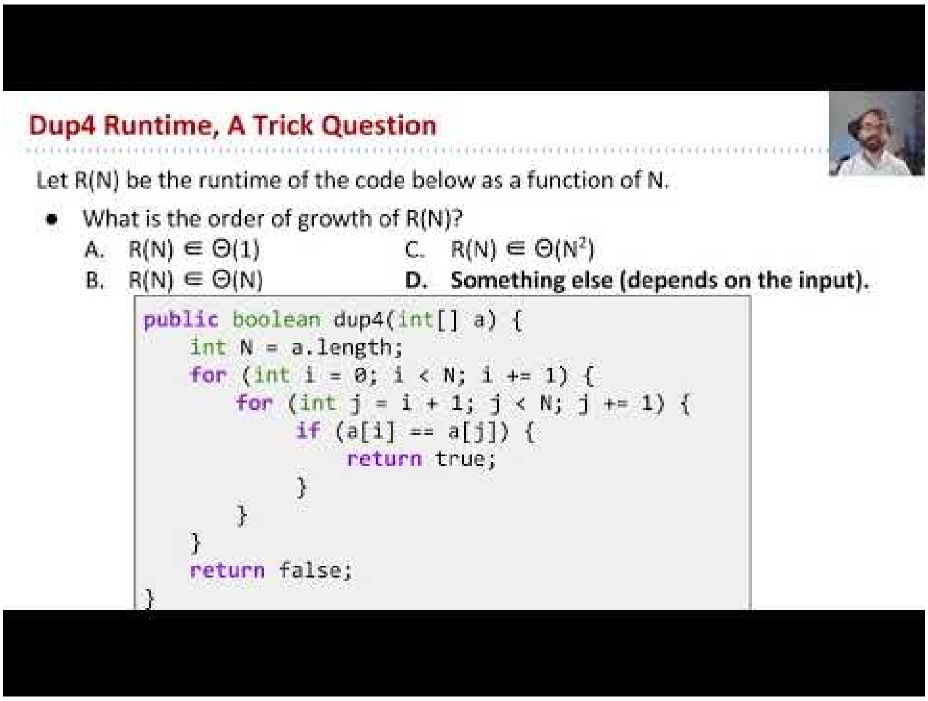
- N^2 vs. $NlogN$ is an enormous difference.
- Going from $NlogN$ to N is nice, but not a radical change.

Hopefully this set of examples has provided some good practice with the techniques and patterns of runtime analysis. Remember, there are no magic shortcuts, but you have to tools to approach the problems. Go forth and analyze!!

Omega and Amortized Analysis

In this section, we'll wrap up our discussion of asymptotics. **Much of this material won't be expanded on until later in the course.** This section expands on the concept of Big O and introduces Omega. We'll also explore the idea of amortized runtimes and their analysis. Finally, we'll end on empirical analysis of runtimes and a sneak preview of complexity theory.

Runtime Analysis Subtleties



Video link

To demonstrate why it is useful to use Big O, let's go back to our duplicate-finding functions consider the following exercises.

Exercise: Let $R(N)$ be the runtime of dup3 as a function of N, the length of the array. What is the order of growth of $R(N)$?

```
public boolean dup3(int[] a) {
    int N = a.length;
    for (int i = 0; i < N; i += 1) {
        for (int j = 0; j < N; j += 1) {
            if (a[i] == a[j]) {
                return true;
            }
        }
    }
    return false;
}
```

Answer: $R(N) \in \Theta(1)$, it's constant time! That's because there's a bug in dup3: it always compares the first element with itself. In the very first iteration, i and j are both 0, so the function always immediately returns. Bummer!

Let's fix up the bug in dup4 and try it again.

Exercise: Let $R(N)$ be the runtime of dup4 as a function of N, the length of the array. What is the order of growth of $R(N)$?

```
public boolean dup4(int[] a) {
    int N = a.length;
    for (int i = 0; i < N; i += 1) {
        for (int j = i + 1; j < N; j += 1) {
            if (a[i] == a[j]) {
                return true;
            }
        }
    }
    return false;
}
```

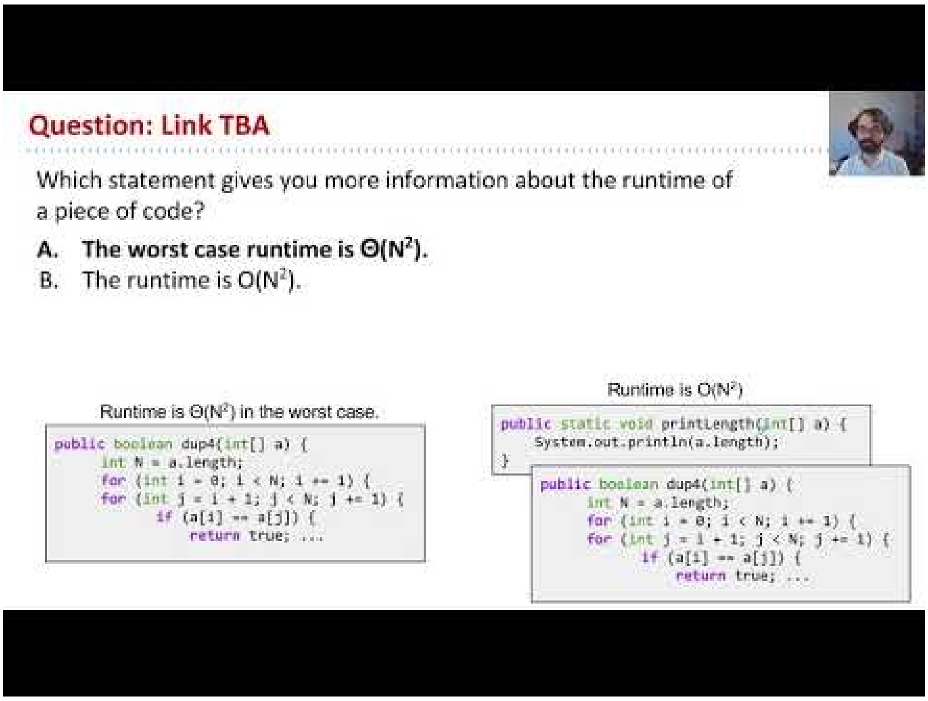
Answer: This time, the runtime depends on not only the length of the input, but also the array's contents. In the best case, $R(N) \in \Theta(1)$. If the input array contains all of the same element, then no matter how long it is, dup4 will return on the first iteration.

On the other hand, in the worst case, $R(N) \in \Theta(N^2)$. If the array has no duplicates, then dup4 will never return early, and the nested for loop will result in quadratic runtime.

This exercise highlights one limitation of Big Theta. Big Theta expresses the exact order of as a function of the input size. However, if the runtime depends on more than just the *size* of the input, then we must qualify our statements into different cases before using Big Theta.

Big O does away with this annoyance. Rather than having to describe both the best and worse case, for the example above, we can simply say that the runtime of dup4 is $O(N^2)$. Sometimes dup4 is faster, but it's at worst quadratic.

Big O Abuse



[Video link]

Consider the following statements:

1. The most expensive room in the hotel is $639 per night.
2. Every room in the hotel is less than or equal to $639 per night.

Which statement gives you more information about a hotel?

The first one. The second statement provides only an upper bound on room prices, ie. an inequality. The first statement tells you not only the upper bound of room prices, but also that this upper bound is reached. For example, consider a cheap hotel whose most expensive room is $89/night and an expensive one whose most expensive room is $639/night. Both hotels fulfill the second statement, but the first statement narrows it down to just the latter hotel.

Exercise: Which statement gives you more information about the runtime of a piece of code?

1. The worst case runtime is Θ(N^2).
2. The runtime is O(N^2). **Answer:** Similar to the hotel problem, the first statement provides more information. Consider the following method:

```java
public static void printLength(int[] a) {
    System.out.println(a.length);
}
```

Both this simple method and dup4 have runtime $O(N^2)$, so knowing statement 2 would not be able to distinguish between these. But statement 1 is more precise, and is only true for dup4.

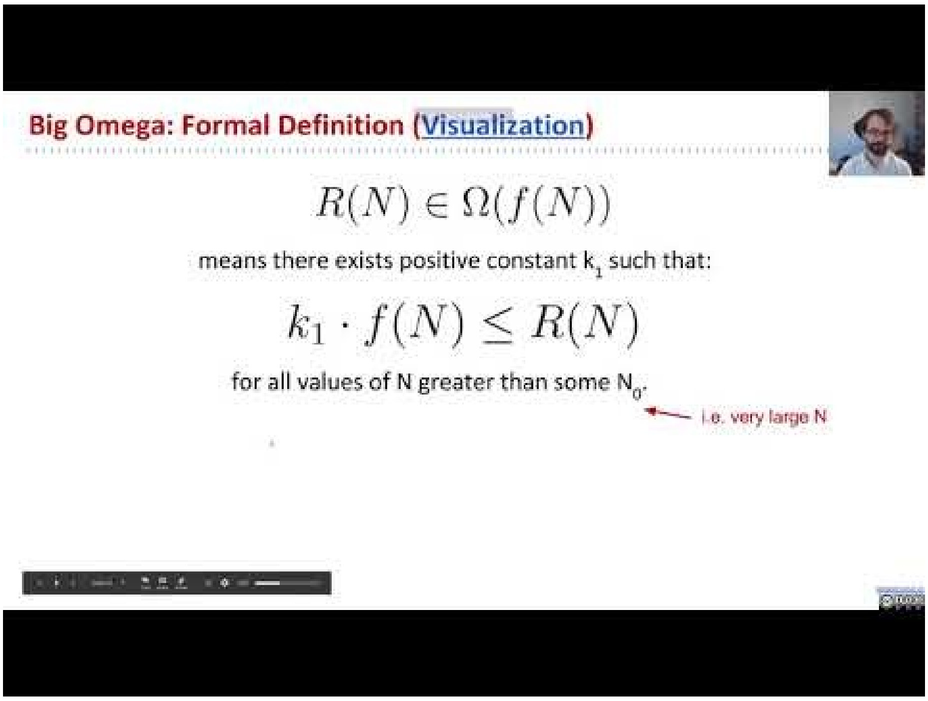
In the real world, and oftentimes in conversation, Big O is often used where in places where Big Theta would be more informative. We saw one good reason for this -- it frees us from needing to use qualifying statements. However, while the looser statement is true, it's not as useful as a Big Theta bound.

Note: Big O is NOT the same as "worst case". But it is often used as such.

To summarize the usefulness of Big O:

- It allows us to make simple statements without case qualifications, in cases where the runtime is different for different inputs.
- Sometimes, for particularly tricky problems, we (the computer science community) don't know the exact runtime, so we may only state an upper bound.
- It's a lot easier to write proofs for Big O than Big Theta, like we saw in finding the runtime of mergesort in the previous chapter. This is beyond the scope of this course.

Big Omega



[Video link]

To round out our understanding of runtimes, let's also define the complement of Big O, to describe lower bounds.

While Big Theta can be informally thought of as runtime equality and Big O represents "less than or equal", Big Omega can be thought of as the "greater than or equal". For example, in addition to knowing that $N^3 + 3N^4 \in \Theta(N^4)$, all of the following statements are true:

$N^3 + 3N^4 \in \Omega(N^4)$ $N^3 + 3N^4 \in \Omega(N^3)$ $N^3 + 3N^4 \in \Omega(\log N)$ $N^3 + 3N^4 \in \Omega(1)$

If $N^3 + 3N^4 \in \Theta(N^4)$, then the function $N^3 + 3N^4$ is also "greater than or equal to" N^4. The function must also grow faster than any function slower asymptotically than N^4, eg. 1 and N^3.

There's two common uses for Big Omega:

1. It's used to prove Big Theta runtime. If $R(N) = O(f(N))$ and $R(N) = \Omega(f(N))$, then $R(N) = \Theta(f(N))$. Sometimes, it's easier to prove O and Ω separately. This is outside the scope of this course.
2. It's used to prove the difficulty of a problem. For example, ANY duplicate-finding algorithm must be $\Omega(N)$, because the algorithm must at least look at each element.

Here's a table to summarize the three Big letters:

	Informal Meaning	**Example Family**	**Example Family Members**
Big Theta $\Theta(f(N))$	Order of growth is f(N)	$\Theta(N^2)$	$N^2/2, 2N^2, N^2 + 38N + 1/N$
Big O $O(f(N))$	Order of growth is less than or equal to f(N)	$O(N^2)$	$N^2/2, 2N^2, \log N$
Big Omega $\Omega(f(N))$	Order of growth is greater than or equal to f(N)	$\Omega(N^2)$	$N^2/2, 2N^2, 5^N$

Amortized Analysis (Intuitive Explanation)

Video link

Grigometh's Urn

Grigometh is a demon dog that looks a bit spooky. He offers you the ability to appear to horses in their dreams, just like how he sometimes appears in your dreams and midterms. However, in return for this ability, you must periodically offer him urnfuls of hay, as tribute. He gives you two payment options:

- Choice 1: Every day, Grigometh eats 3 bushels of hay from your urn.
- Choice 2: Grigometh eats exponentially more hay over time, but comes exponentially less frequently. Specifically:
  - On day 1, he eats 1 bushel of hay (total 1)
  - On day 2, he eats 2 additional bushels of hay (total 3)
  - On day 4, he eats 4 additional bushels of hay (total 7)
  - On day 8, he eats 8 additional bushels of hay (total 15)

You want to develop a routine and put a fixed amount of hay in your urn each day. How much hay must you put in the earn each day for each payment scheme? Which is cheaper?

For the first choice, you'd have to put exactly 3 bushels of hay in the urn each day. However, the second choice actually turns out to be a a bit cheaper! You can get away with just putting in 2 bushels of hay in the urn each day. (Try to convince yourself why this is true -- one way to do this is the write out the total amount of hay you contribute after each day. You'll notice that whenever Grigometh comes for his hay snack, you'll always have one extra bushel in the urn after he takes his fill. Neat.)

Bushels aside, notice here that Grigometh's hay consumption per day is effectively constant, and in choice 2, we can describe this situation as **amortized constant** hay consumption.

AList Resizing and Amortization

It turns out, Grigometh's hay dilemma is very similar to AList resizing from early in the course. Recall that in our implementation of array-based lists, when we call the add method on an AList whose underlying array is full, we need to resize the array. In other words, the add method, when full, must create a new array of larger size, copy over the old elements, and then finally add the new element. How much bigger should our new array be? Recall the following two implementations:

Implementation 1

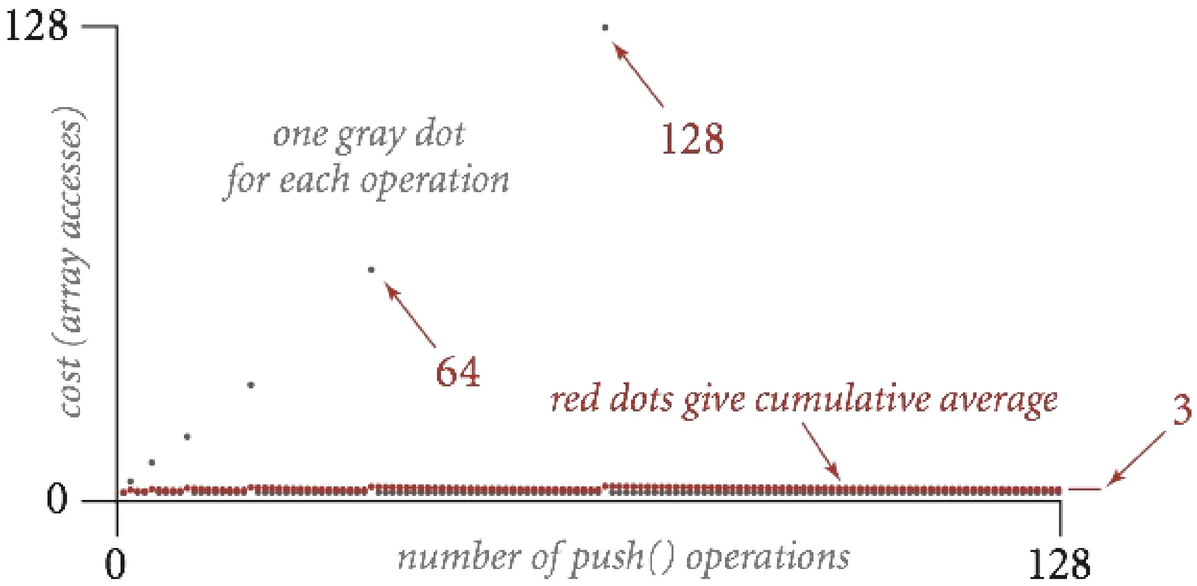
```
public void addLast(int x) {
  if (size == items.length) {
    resize(size + RFACTOR);
  }
  items[size] = x;
  size += 1;
}
```

Implementation 2

```
public void addLast(int x) {
  if (size == items.length) {
    resize(size * RFACTOR);
  }
  items[size] = x;
  size += 1;
}
```

The first implementation turned out to be unusably bad. When our array fills up, every time a new element is added, the entire array must be copied to a new one. The second implementation, which we called geometric resizing, on the other hand, worked nicely. In fact, this is how Python lists are implemented.

Let's look into the runtime of implementation 2 in more detail. Let RFACTOR be 2. When the array is full, resize doubles its size. Most add operations take $\Theta(1)$ time, but some are very expensive, and linear to the current size. However, if we average out the cost of expensive adds with resize over all the adds that are cheap, and given that expensive adds with happen half as frequently every time it happens, it turns out that **on average**, the runtime of add is $\Theta(1)$. We'll prove this in the next section. In the meantime, here's a graph to illustrate this:



Amortized Analysis (Rigorous Explanation)

A more rigorous examination of amortized analysis is done here, in three steps:

1. Pick a cost model (like in regular runtime analysis)
2. Compute the average cost of the i'th operation
3. Show that this average (amortized) cost is bounded by a constant.

Suppose that initially, our ArrayList contains a length-1 array. Let's apply these three steps to ArrayList resizing:

1. For our cost model, we'll only consider array reads and writes. (You could also include other operations into your cost model, such as array creation and the cost of filling in default array values. But it turns out that these will all yield the same result.)
2. Let's compute the cost of a sequence of array adds. Suppose we had the following code and accompanying diagram:

TODO: image

- x.add(0) performs 1 write operation. No resizing. Total: 1 operation
- x.add(1) resizes and copies the existing array (1 read, 1 write), and then writes the new element. Total: 3 operations
- x.add(2) resizes and copies the existing array (2 reads, 2 writes), and then writes the new element. Total: 5 operations
- x.add(3) does not resize, and only writes the new element. Total: 1 operations
- x.add(4) resizes and copies the existing array (4 reads, 4 writes), and then writes the new element. Total: 9 operations

It's easier to keep track of this in a table:

Insert #	0	1	2	3	4	5	6	7	8	9	10	11	12	13
a[i] write cost	1	1	1	1	1									
Resize/copy cost	0	2	4	0	8									
Total cost for #	1	3	5	1	9									
Cumulative cost	1	4	9	10	19									

Exercise: Fill out the rest of this table. Total cost is the total cost for one particular insert. Cumulative cost is the cost for all inserts so far.

Answer:

Insert #	0	1	2	3	4	5	6	7	8	9	10	11	12
a[i] write cost	1	1	1	1	1	1	1	1	1	1	1	1	1
Resize/copy cost	0	2	4	0	8	0	0	0	16	0	0	0	0
Total cost for #	1	3	5	1	9	1	1	1	17	1	1	1	1
Cumulative cost	1	4	9	10	19	20	21	22	39	40	41	42	43

So what is the average cost of the a sequence of adds? For 13 adds, this average cost is $44/13 = 3.14$ per add. But for the first 8 adds, the average cost is $39/8 = 4.875$ per add.

1. Is the average (amortized) cost bounded by a constant? It seems like it might be bounded by 5. But just by looking at the first 13 adds, we cannot be completely sure.

We'll now introduce the idea of "potential" to aid us in solving this amortization mystery. For each operation i, eg. each add or Grigometh visit, let c_i be the true cost of the operation, while a_i be some arbitrary amortized cost of the operation. a_i, a constant, must be the same for all i.

Let Φ_i be the potential at operation i, which is the cumulative difference between amortized and true cost: $\Phi_i = \Phi_{i-1} + a_i - c_i$

a_i is an arbitrary constant, meaning we can chose it. **If we chose a_i such that Φ_i is never negative and a_i is constant for all i, then the amortized cost is an upper bound on the true cost.** And if the true cost is upper bounded by a constant, then we've shown that it is on average constant time!

Let's try this for Grigometh's hay tribute. For each day i, the actual cost is how much hay Grigometh eats on that day. The amortized cost is how much hay we put in the urn on that day -- let's assume that's 3. Let the initial potential be $\Phi_0 = 0$:

Day (i)	1	2	3	4	5	6	7	8	9	10	11	12	13
Actual cost c_i	1	2	0	4	0	0	0	0	8	0	0	0	0
Amortized cost a_i	3	3	3	3	3	3	3	3	3	3	3	3	3
Change in potential	2	1	3	-1	3	3	3	3	-5	3	3	3	3
Potential Φ_i	2	3	6	5	8	11	14	17	12	15	18	21	24

If we let $a_i = 3$, the potential never falls negative -- in fact, it looks like after many days, we will keep on having a surplus of hay for Grigometh, if we add 3 bushels per day. We'll save the rigorous proof of this for another course, but hopefully the trend looks convincing enough. So Grigometh's hay tribute is on average constant.

Exercise: We'd like to conserve as much hay as possible. Show that we can satisfy Grigometh's hunger and never have negative potential even if we set $a_i = 2$.

Now back to ArrayList resizing.

Exercise: What is the value of c_i for ArrayList add operations? If we let the amortized cost $a_i = 5$, will the potential ever become negative? Is there a smaller amortized cost that works? Fill out a table like the one for Grigometh to help with this.

Answer: c_i is the total cost for array resizing and adding the new element, where $c_i = 2i + 1$ if i is a power of 2, and $c_i = 1$ otherwise.

Insert #	0	1	2	3	4	5	6	7	8	9	10	11	12
Actual cost c_i	1	3	5	1	9	1	1	1	17	1	1	1	1
Amortized cost a_i	5	5	5	5	5	5	5	5	5	5	5	5	5
Change in potential	4	2	0	4	-4	4	4	4	-12	4	4	4	4
Potential Φ_i	4	6	6	10	6	10	14	18	6	10	14	18	22

By looking at the trend, the potential should never be negative (proof for this is omitted). Intuitively, for high-cost operations, we use the previous low-cost operations to store up potential.

Finally, we've shown that ArrayList add operations are indeed amortized constant time. Geometric resizing (multiplying by the RFACTOR) leads to good list performance.

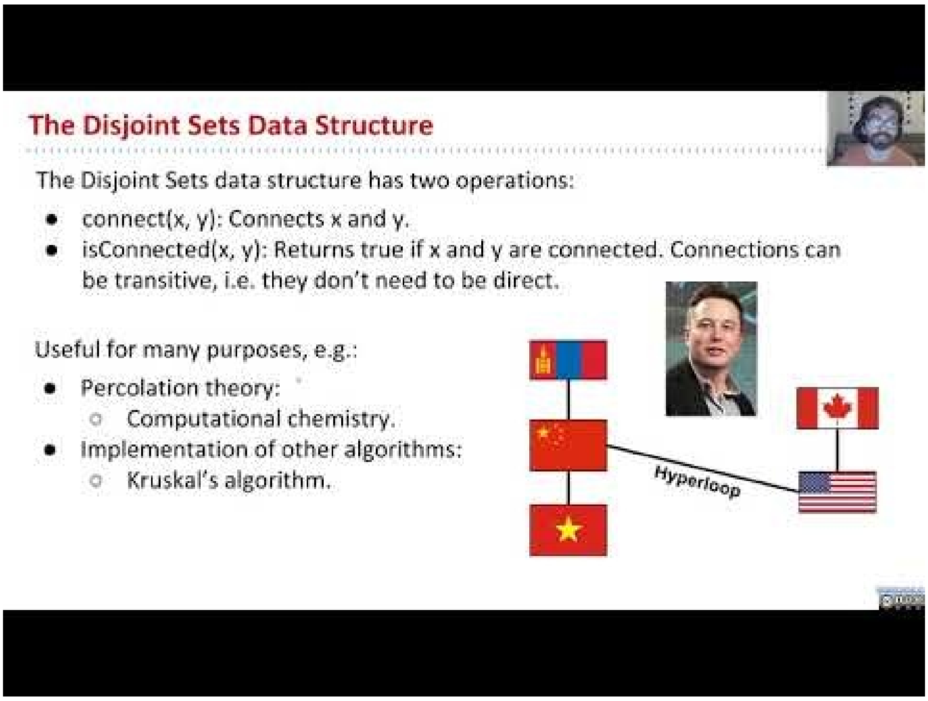
Summary

- Big O is an upper bound ("less than or equals")
- Big Omega is a lower bound ("greater than or equals")
- Big Theta is both an upper and lower bound ("equals")
- Big O does NOT mean "worst case". We can still describe worst cases using Big Theta
- Big Omega does NOT mean "best case". We can still describe best cases using Big Theta
- Big O is sometimes colloquially used in cases where Big Theta would provide a more precise statement
- Amortized analysis provides a way to prove the average cost of operations.
- If we chose a_i such that Φ_i is never negative and a_i is constant for all i, then the amortized cost is an upper bound on the true cost.

Introduction to Disjoint Sets

Two sets are named *disjoint sets* if they have no elements in common. A Disjoint-Sets (or Union-Find) data structure keeps track of a fixed number of elements partitioned into a number of *disjoint sets*. The data structure has two operations:

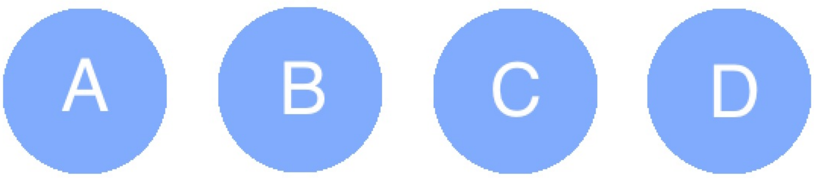
1. `connect(x, y)` : connect `x` and `y` . Also known as `union`
2. `isConnected(x, y)` : returns true if `x` and `y` are connected (i.e. part of the same set).



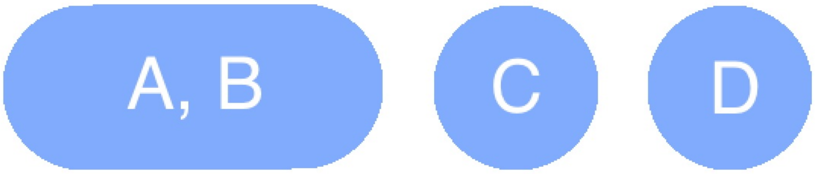
[Video link](#)

A Disjoint Sets data structure has a fixed number of elements that each start out in their own subset. By calling `connect(x, y)` for some elements `x` and `y` , we merge subsets together.

For example, say we have four elements which we'll call A, B, C, D. To start off, each element is in its own set:



After calling `connect(A, B)`:



Note that the subsets A and B were merged. Let's check the output some `isConnected` calls:

`isConnected(A, B) -> true`

`isConnected(A, C) -> false`

After calling `connect(A, D)`:

We find the set A is part of and merge it with the set D is part of, creating one big A, B, D set. C is left alone.

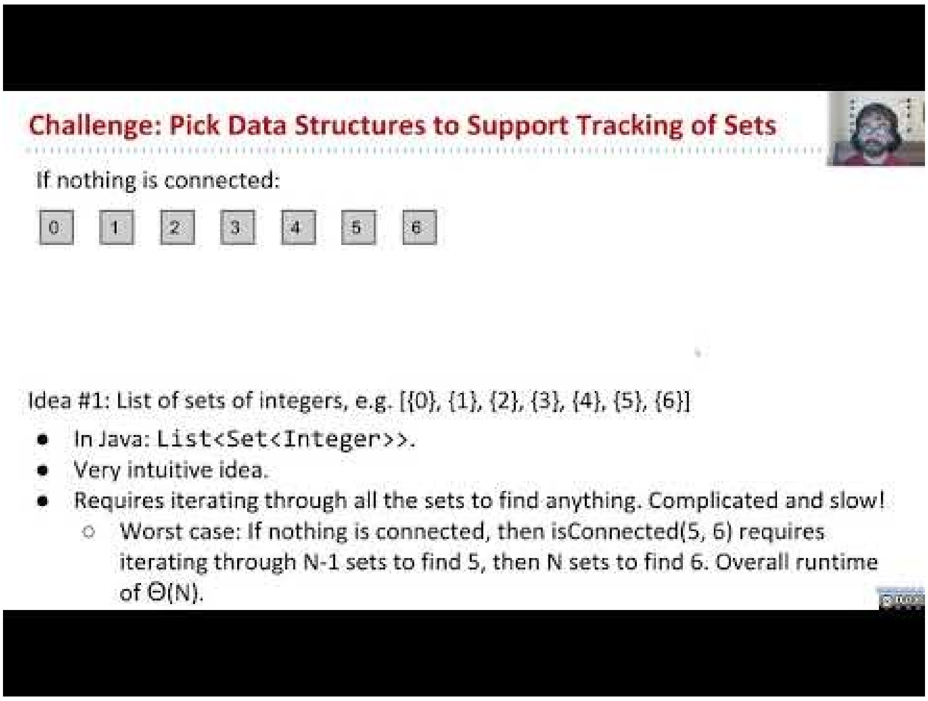
`isConnected(A, D) -> true`
`isConnected(A, C) -> false`

With this intuition in mind, let's formally define what our DisjointSets interface looks like. As a reminder, an **interface** determines *what* behaviors a data structure should have (but not *how* to accomplish it). For now, we'll only deal with sets of non-negative integers. This is not a limitation because in production we can assign integer values to anything we would like to represent.

```java
public interface DisjointSets {
    /** connects two items P and Q */
    void connect(int p, int q);

    /** checks to see if two items are connected */
    boolean isConnected(int p, int q);
}
```

In addition to learning about how to implement a fascinating data structure, this chapter will be a chance to see how an implementation of a data structure evolves. We will discuss four iterations of a Disjoint Sets design before being satisfied: *Quick Find* → *Quick Union* → *Weighted Quick Union (WQU)* → *WQU with Path Compression*. **We will see how design decisions greatly affect asymptotic runtime and code complexity.**

Quick Find



[Video link]

Now we tackle *how* to achieve the desired behavior of our `DisjointSets` interface. Our challenge is to keep track of set membership.

ListOfSets

Intuitively, we might first consider representing Disjoint Sets as a list of sets, e.g, `List<Set<Integer>>`.

For instance, if we have N=6 elements and nothing has been connected yet, our list of sets looks like: `[{0}, {1}, {2}, {3}, {4}, {5}, {6}]`. Looks good. However, consider how to complete an operation like `connect(5, 6)`. We'd have to iterate through up to `N` sets to find 5 and `N` sets to find 6. Our runtime becomes `O(N)`. And, if you were to try and implement this, the code would be quite complex.

> The lesson to take away is that **initial design decisions determine our code complexity and runtime.**

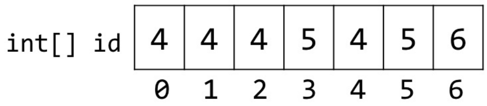
Quick Find

Let's consider another approach using a *single array of integers*.

- The **indices of the array** represent the elements of our set.

- The **value at an index** is the set number it belongs to.

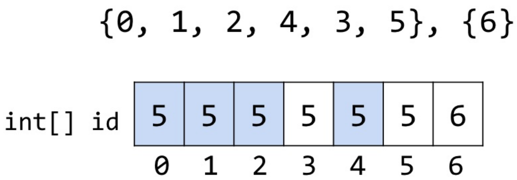
For example, we represent `{0, 1, 2, 4}, {3, 5}, {6}` as:



The array indices (0...6) are the elements. The value at `id[i]` is the set it belongs to. *The specific set number doesn't matter as long as all elements in the same set share the same id.*

`connect(x, y)`

Let's see how the connect operation would work. Right now, `id[2] = 4` and `id[3] = 5`. After calling `connect(2, 3)`, all the elements with id 4 and 5 should have the same id. Let's assign them all the value 5 for now:



`isConnected(x, y)`

To check `isConnected(x, y)`, we simply check if `id[x] == id[y]`. Note this is a constant time operation!

We call this implementation "Quick Find" because finding if elements are connected takes constant time.

Summary and Code

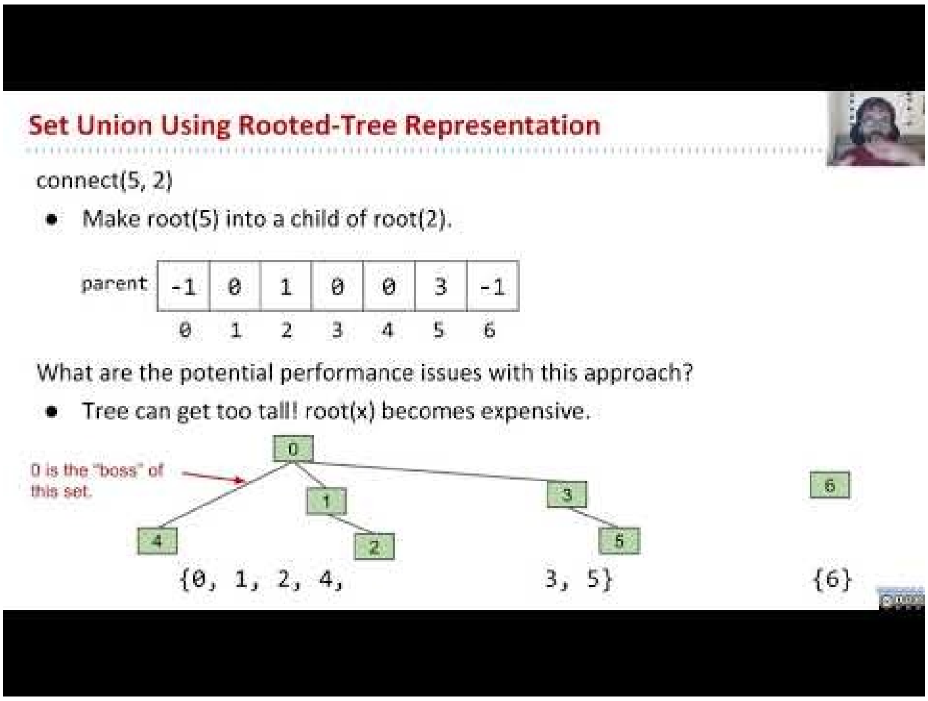
Implementation	Constructor	`connect`	`isConnected`
ListOfSets	Θ(N)[1]	O(N)	O(N)
QuickFind	Θ(N)	Θ(N)	Θ(1)

N = number of elements in our DisjointSets data structure

```java
public class QuickFindDS implements DisjointSets {

    private int[] id;

    /* Θ(N) */
    public QuickFindDS(int N){
        id = new int[N];
        for (int i = 0; i < N; i++){
            id[i] = i;
        }
    }

    /* need to iterate through the array => Θ(N) */
    public void connect(int p, int q){
        int pid = id[p];
        int qid = id[q];
        for (int i = 0; i < id.length; i++){
            if (id[i] == pid){
                id[i] = qid;
            }
        }
    }

    /* Θ(1) */
    public boolean isConnected(int p, int q){
        return (id[p] == id[q]);
    }
}
```

> 1. We didn't discuss this but you can reason that having to create N distinct sets initially is Θ(N) ↩

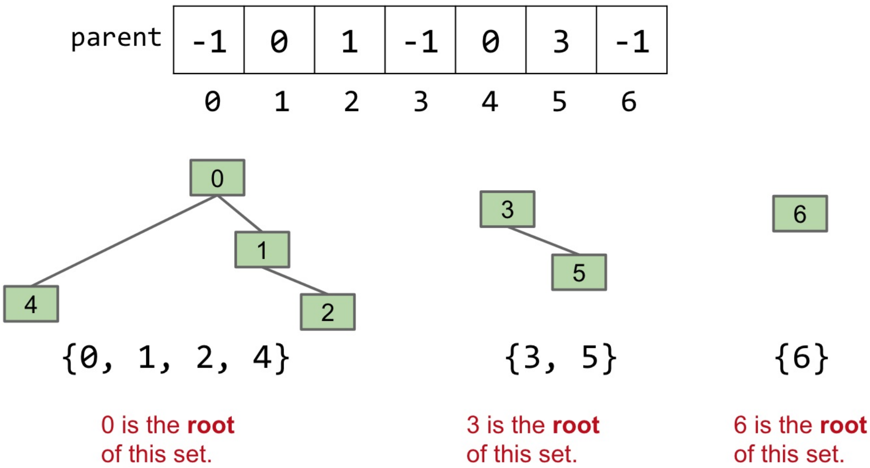
Quick Union



[Video link](#)

Suppose we prioritize making the `connect` operation fast. We will still represent our sets with an array. Instead of an id, we assign each item the index of its parent. If an item has no parent, then it is a 'root' and we assign it a negative value.

This approach allows us to imagine each of our sets as a tree. For example, we represent `{0, 1, 2, 4}, {3, 5}, {6}` as:



Note that we represent the sets using **only an array**. We visualize it ourselves as trees.

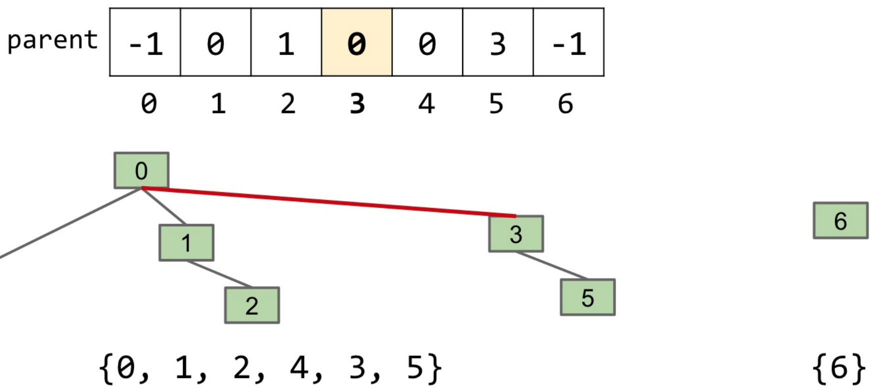
For QuickUnion we define a helper function `find(int item)` which returns the root of the tree `item` is in. For example, for the sets above, `find(4) == 0`, `find(1) == 0`, `find(5) == 3`, etc. Each element has a unique root.

`connect(x, y)`

To connect two items, we find the set that each item belongs to (the roots of their respective trees), and make one the child of the other. Example:

`connect(5, 2)`:

1. `find(5)` -> 3
2. `find(2)` -> 0
3. Set `find(5)`'s value to `find(2)` aka `parent[3] = 0`



Note how element 3 now points to element 0, combining the two trees/sets into one.

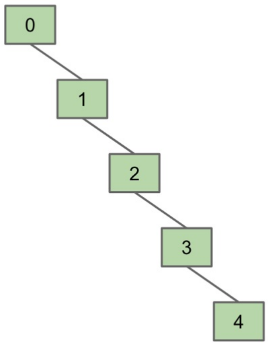
In the best case, if `x` and `y` are both roots of their trees, then `connect(x, y)` just makes `x` point to `y`, a Θ(1) operation! (Hence the name QuickUnion)

`isConnected(x, y)`

If two elements are part of the same set, then they will be in the same tree. Thus, they will have the same root. So for `isConnected(x, y)` we simply check if `find(x) == find(y)`.

Performance

There is a potential performance issue with QuickUnion: the tree can become very long. In this case, finding the root of an item (`find(item)`) becomes very expensive. Consider the tree below:



In the worst case, we have to traverse all the items to get to the root, which is a Θ(N) runtime. Since we have to call `find` for both `connect` and `isConnected`, the runtime for both is upper bounded by O(N).

Summary and Code

Implementation	Constructor	`connect`	`isConnected`
QuickUnion	Θ(N)	O(N)	O(N)
QuickFind	Θ(N)	Θ(N)	Θ(1)
QuickUnion	Θ(N)	O(N)	O(N)

N = number of elements in our DisjointSets data structure

From the runtime chart, QuickUnion seems worse than QuickFind! Note however that O(N) as an **upper bound**. When our trees are balanced, both `connect` and `isConnected` perform reasonably well. In the next section we'll see how to *guarantee* they perform well.

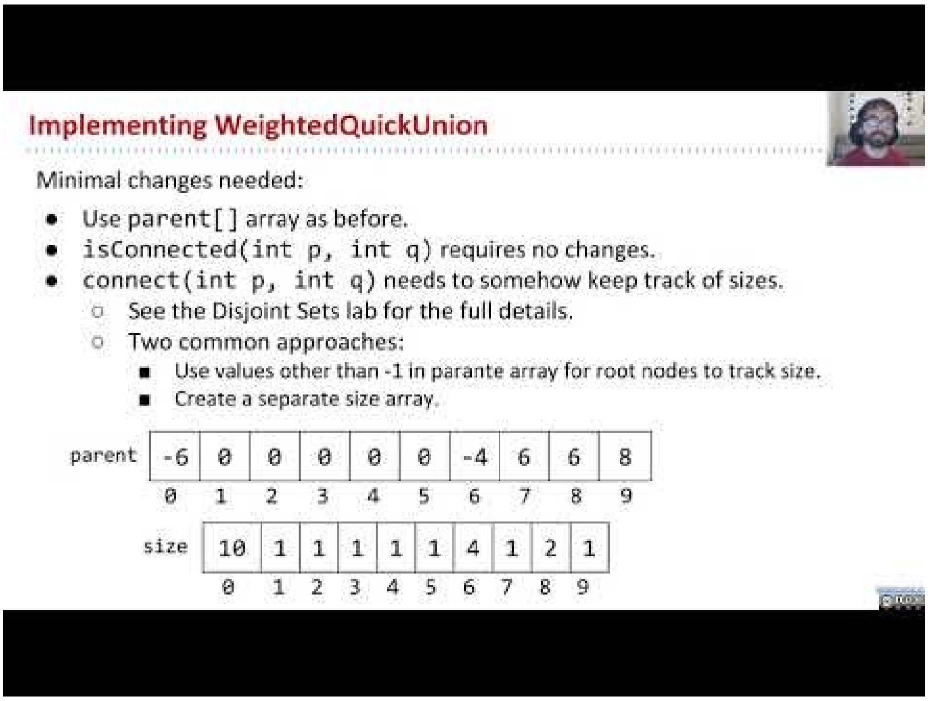
```
public class QuickUnionDS implements DisjointSets {
    private int[] parent;

    public QuickUnionDS(int num) {
        parent = new int[num];
        for (int i = 0; i < num; i++) {
            parent[i] = i;
        }
    }

    private int find(int p) {
        while (parent[p] >= 0) {
            p = parent[p];
        }
        return p;
    }

    @Override
    public void connect(int p, int q) {
        int i = find(p);
        int j= find(q);
        parent[i] = j;
    }

    @Override
    public boolean isConnected(int p, int q) {
        return find(p) == find(q);
    }
}
```

Weighted Quick Union (WQU)

Improving on Quick Union relies on a key insight: whenever we call `find`, we have to climb to the root of a tree. Thus, the shorter the tree the faster it takes!

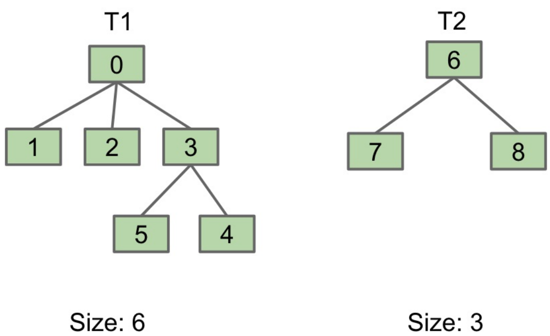
New rule: whenever we call `connect`, we always link the root of the **smaller** tree **to the larger** tree.

Following this rule will give your trees a maximum height of $\log N$, where N is the number of elements in our Disjoint Sets. How does this affect the runtime of `connect` and `isConnected`?

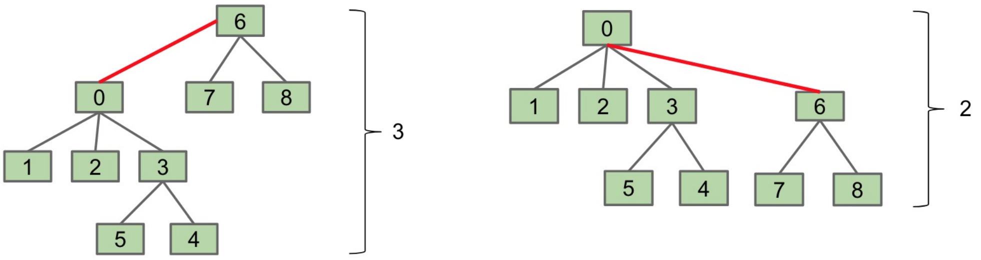


[Video link]

Let's illustrate the benefit of this with an example. Consider connecting the two sets T1 and T2 below:



We have two options for connecting them:



Option 1 Option 2

The first option we link T1 to T2. In the second, we link T2 to T1.

The **second option is preferable** as it only has a height of 2, rather than 3. By our new rule, we would choose the second option as well because T2 is smaller than T1 (size of 3 compared to 6).

We determine smaller / larger by the number of items in a tree. Thus, when connecting two trees we need to know their size (or weight). We can store this information in the root of the tree by replacing the `-1` 's with `-(size of tree)` . You will implement this in Lab 6.

Maximum height: Log N

Following the above rule ensures that the *maximum* height of any tree is Θ(log N). N is the number of elements in our Disjoint Sets. **By extension, the runtimes of `connect` and `isConnected` are bounded by O(log N).**

Why log N? The video above presents a more visual explanation. Here's an optional mathematical explanation why the maximum height is $\log_2 N$. Imagine any element x in tree $T1$. The depth of x increases by 1 only when $T1$ is placed below another tree $T2$. When that happens, the size of the resulting tree will be at least double the size of $T1$ because $size(T2) \geq size(T1)$. The tree with x can double at most $\log_2 N$ times until we've reached a total of N items ($2^{\log_2 N} = N$). So we can double up to $\log_2 N$ times and each time, our tree adds a level $\rightarrow$ maximum $\log_2 N$ levels.

You may be wondering why we don't link trees based off of height instead of weight. It turns out this is more complicated to implement and gives us the same Θ(log N) height limit.

Summary and Code

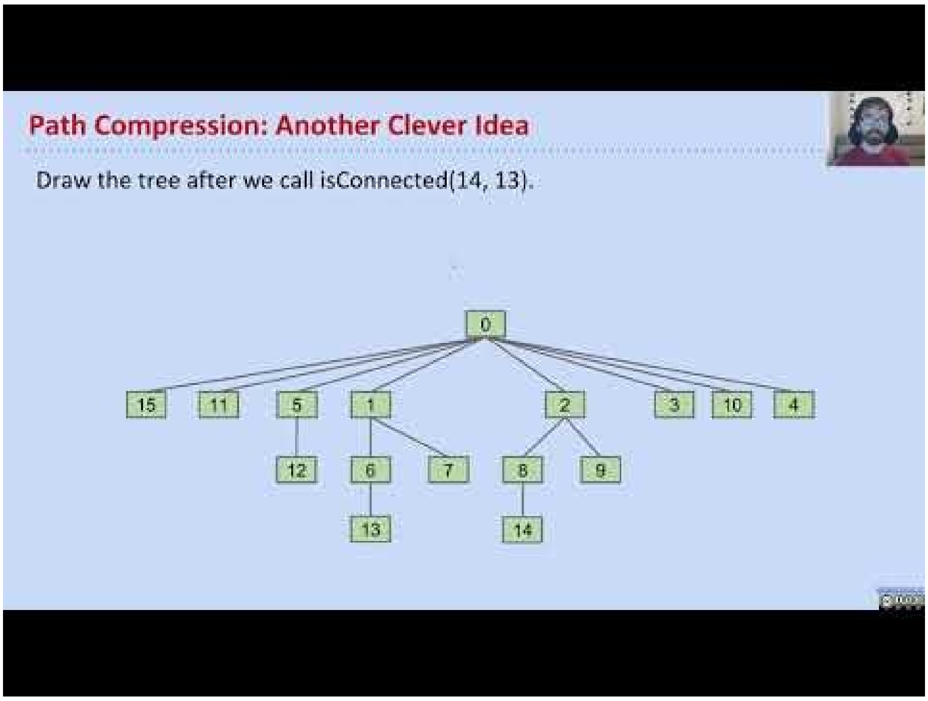
Implementation	Constructor	`connect`	`isConnected`
QuickUnion	Θ(N)	O(N)	O(N)
QuickFind	Θ(N)	Θ(N)	Θ(1)
QuickUnion	Θ(N)	O(N)	O(N)
Weighted Quick Union	Θ(N)	O(log N)	O(log N)

N = number of elements in our DisjointSets data structure

Code? That's your Lab 6!

Weighted Quick Union with Path Compression

Weighted Quick Union is pretty good, but we can do even better!



[Video link]

The clever insight is realizing that whenever we call `find(x)` we have to traverse the path from `x` to root. So, along the way we can connect all the items we visit to their root at no extra asymptotic cost.

Connecting all the items along the way to the root will help make our tree shorter with each call to `find`.

Recall that **both `connect(x, y)` and `isConnected(x, y)` always call `find(x)` and `find(y)`.** Thus, after calling `connect` or `isConnected` enough, essentially all elements will point directly to their root.

By extension, the average runtime of `connect` and `isConnected` becomes **almost constant** in the long term! This is called the *amortized runtime* (from amortized analysis, ch. 8.4).

More specifically, for M operations on N elements, WQU with Path Compression is in O(N + M (lg* N)). lg* is the iterated logarithm which is less than 5 for any real-world input. [1]

Summary

N: number of elements in Disjoint Set

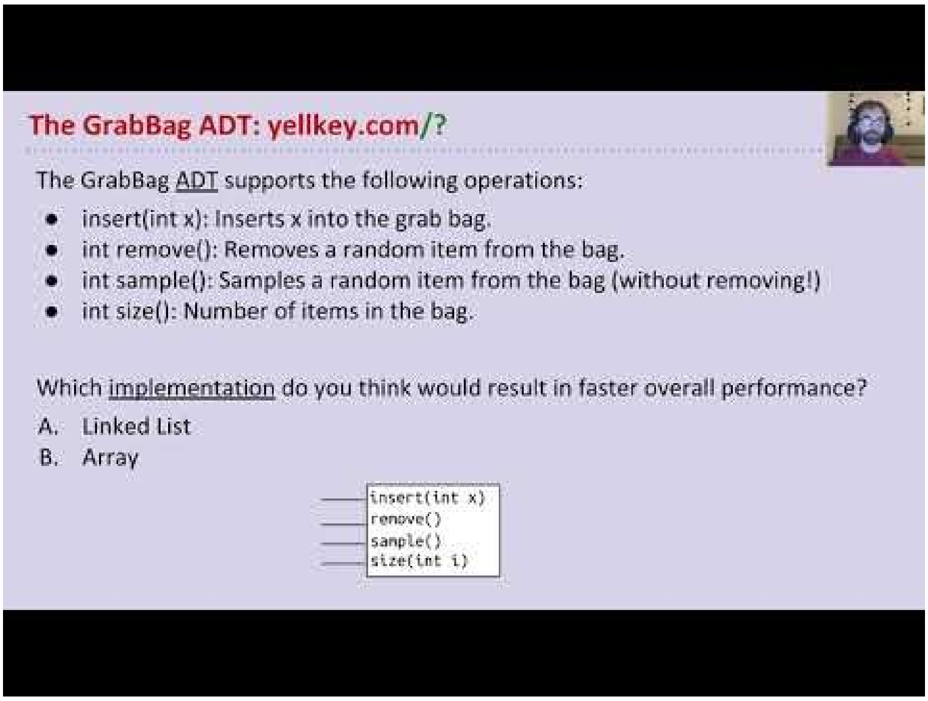
Implementation	`isConnected`	`connect`
Quick Find	Θ(N)	Θ(1)
Quick Union	O(N)	O(N)
Weighted Quick Union (WQU)	O(log N)	O(log N)
WQU with Path Compression	O(α(N))*	O(α(N))*

*behaves as constant in long term.

Code? This is your lab 6!

> 1. Students interested in understanding where the iterated logarithm comes from can read this proof or page 9 from these 170 notes. Path compression is actually even better than iterated log - it's bounded by the inverse Ackermann function α which is comically out of scope. ↩

ADTs



Video link

An Abstract Data Type (ADT) is defined only by its operations, not by its implementation. For example in proj1a, we developed an ArrayDeque and a LinkedListDeque that had the same methods, but how those methods were written was very different. In this case, we say that ArrayDeque and LinkedListDeque are *implementations* of the Deque ADT. From this description, we see that ADT's and interfaces are somewhat related. Conceptually, Deque is an interface for which ArrayDeque and LinkedListDeque are its implementations. In code, in order to express this relationship, we have the ArrayDeque and LinkedListDeque classes inherit from the Deque interface.

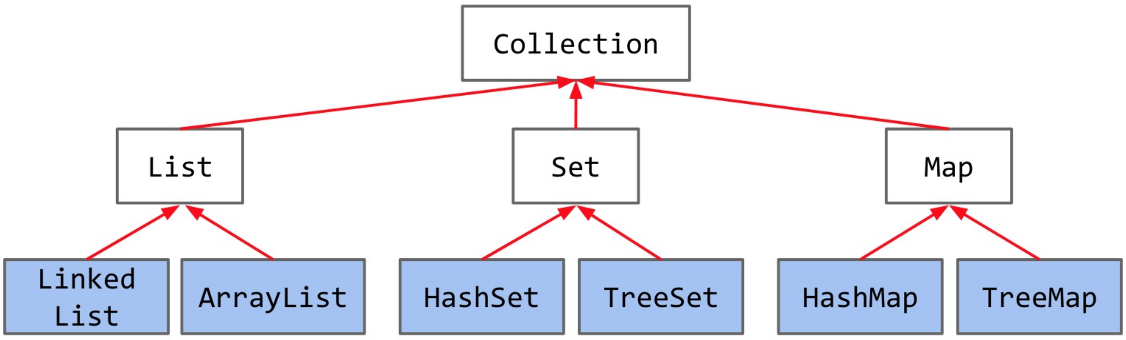
Some commonly used ADT's are:

- Stacks: Structures that support last-in first-out retrieval of elements
  - `push(int x)` : puts x on the top of the stack
  - `int pop()` : takes the element on the top of the stack
- **Lists**: an ordered set of elements
  - `add(int i)` : adds an element
  - `int get(int i)` : gets element at index i
- **Sets**: an unordered set of unique elements (no repeats)
  - `add(int i)` : adds an element
  - `contains(int i)` : returns a boolean for whether or not the set contains the value
- **Maps**: set of key/value pairs

- `put(K key, V value)` : puts a key value pair into the map
- `V get(K key)` : gets the value corresponding to the key

**The bolded ADT's are a subinterfaces of a bigger overarching interface called `Collections`

Below we show the relationships between the interfaces and classes. Interfaces are in white, classes are in blue.

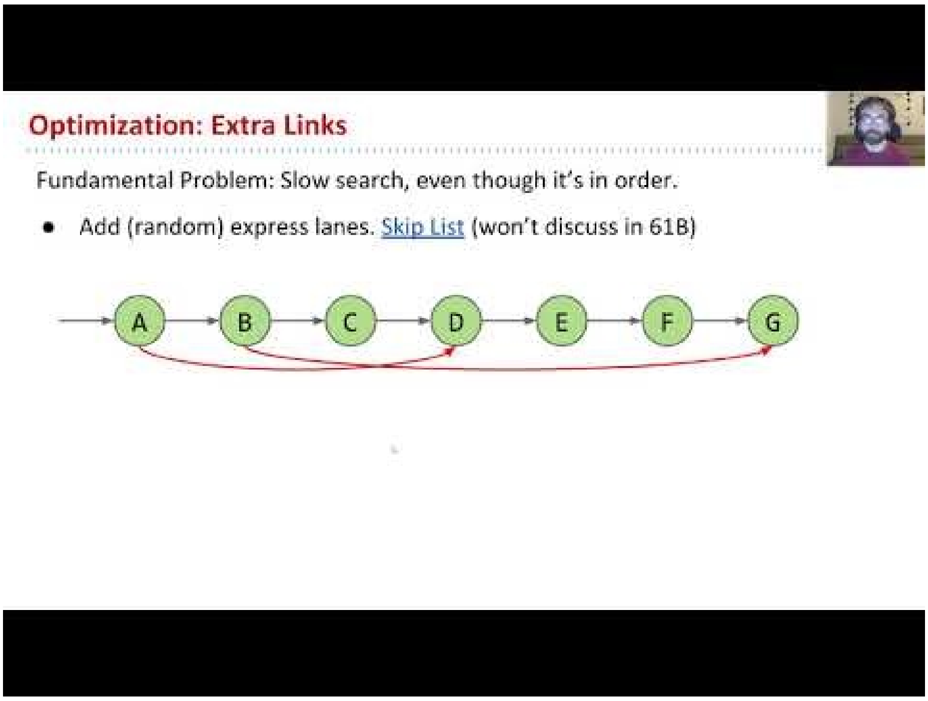


ADT's allow us to make use of object oriented programming in an efficient and elegant way. You saw in proj1b how we could swap `OffByOne` and `OffByN` comparators because they both implemented the same interface! In the same way, you can use an ArrayDeque or a LinkedListArrayDeque interchangeably because they are both part of the Deque ADT.

In the following chapters, we will work on defining some more ADT's and enumerating their different implementations.

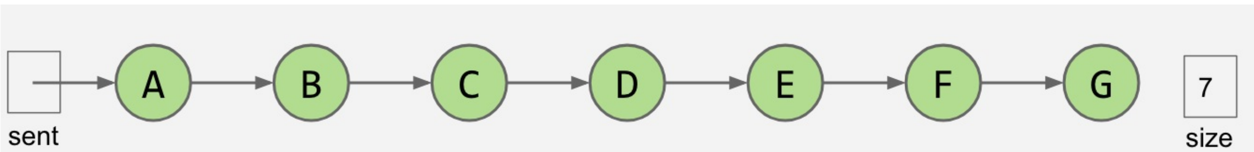
Binary Search Trees

Now we are going to learn about perhaps the most important data structure ever.



Video link

Linked Lists are great, but it takes a long time to search for an item, even if the list is sorted! What if the item is at the end of the list? That would take linear time! Take a look at the linked list below and convince yourself that this is true.

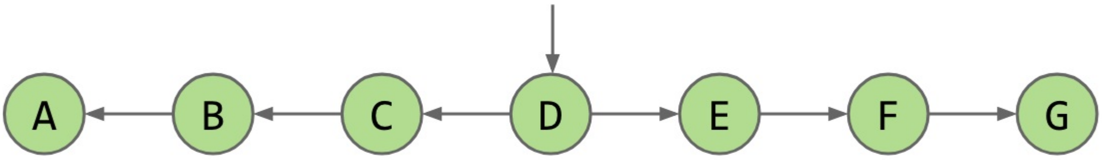


We know that for an array, we can use binary search to find an element faster. Specifically, in $\log(n)$ time. For a short explanation of binary search, check out this link.

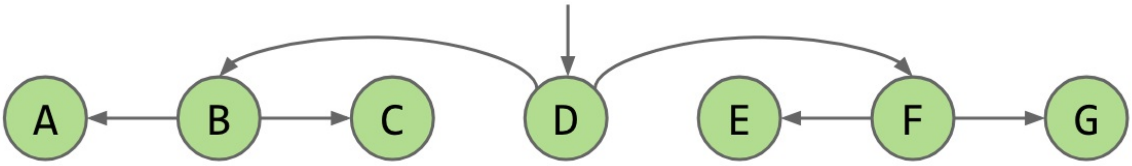
TL;DR: In binary search, we know the list is sorted, so we can use this information to narrow our search. First, we look at the middle element. If it is bigger than the element we are searching for, we look to the left of it. If it is smaller than the element we are searching for, we look to the right. We then look at the middle element of the respective halves and repeat the process until we find the element we are looking for (or don't find it because the list doesn't contain it).

But how do we run binary search for a linked list? We would have to traverse all the way to the middle in order to check the element there, which takes linear time just on its own!

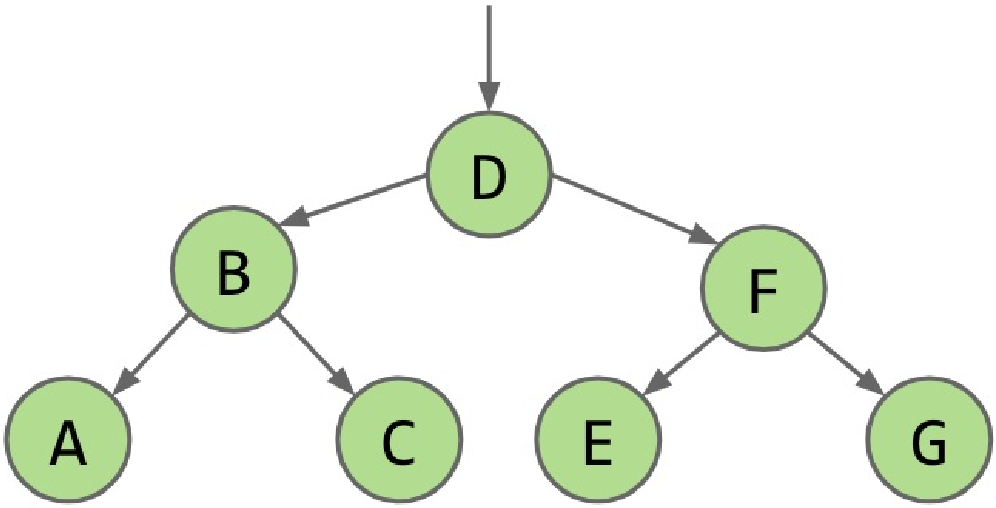
One optimization we can implement is to have a reference to the middle node. This way, we can get to the middle in constant time. Then, if we flip the nodes' pointers, which allows us to traverse to both the left and right halves, we can decrease our runtime by half!



But, we can do better than that. We can further optimize by adding pointers to the middle of each recursive half like so.



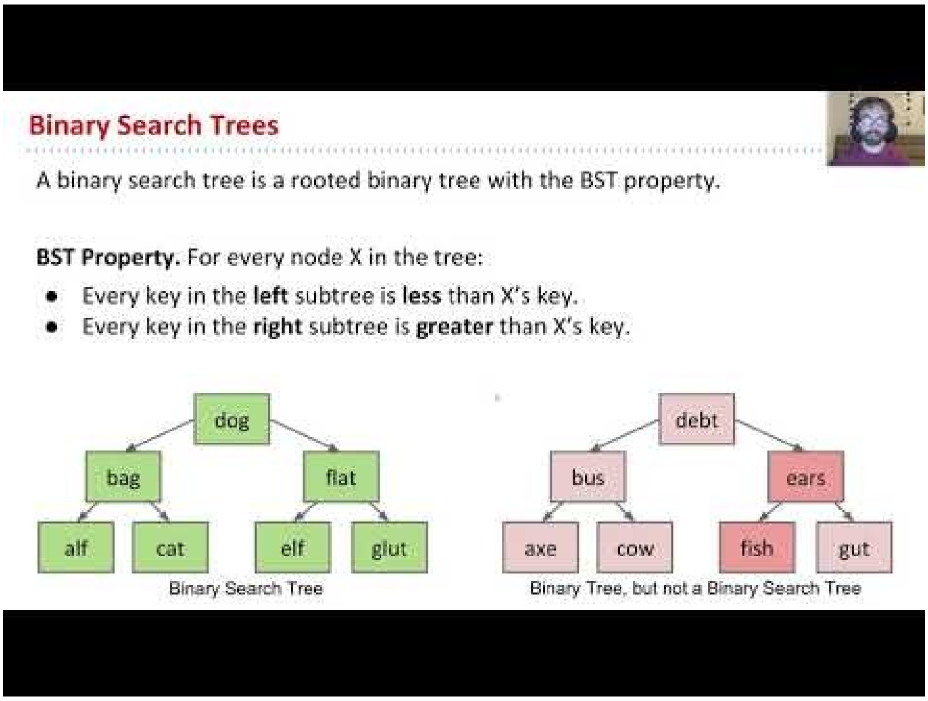
Now, if you stretch this structure vertically, you will see a tree!



This specific tree is called a **binary tree** because each juncture splits in 2.

Properties of trees

Let's formalize the tree data structure a bit more.



Video link

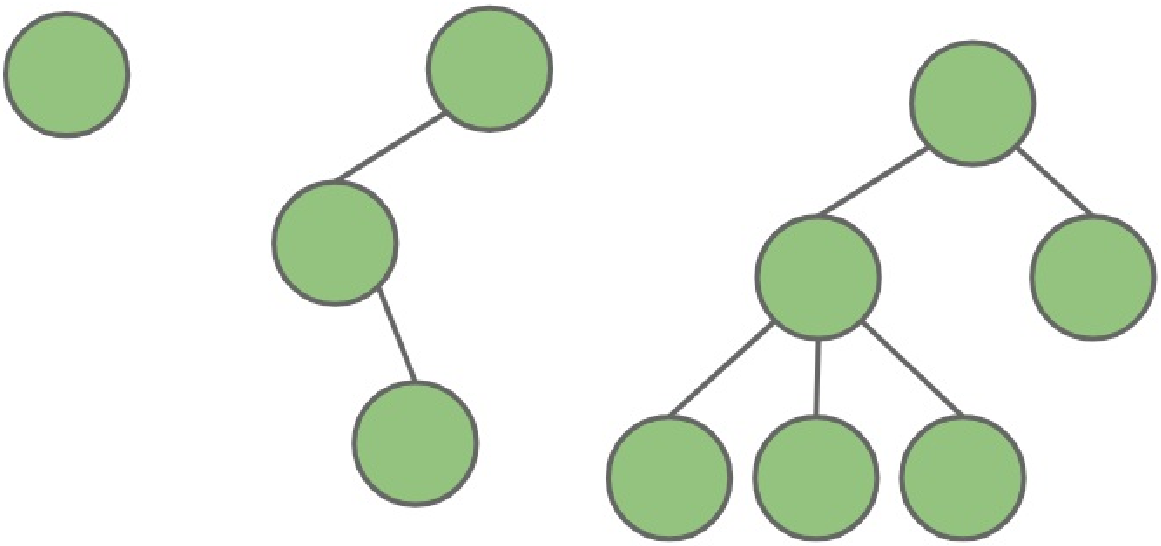
Trees are composed of:

- nodes
- edges that connect those nodes.
    - **Constraint**: there is only one path between any two nodes.

In some trees, we select a **root** node which is a node that has no parents.

A tree also has **leaves**, which are nodes with no children.

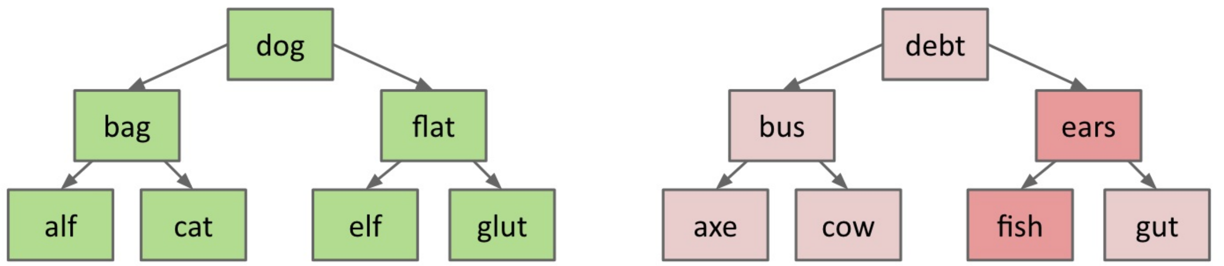
The below structures are valid trees:



Exercise 10.2.1: Can you come up with an example of a non-valid tree?

Relating this to the original tree structure we came up with earlier, we can now introduce new constraints to the already existing constraints. This creates more specific types of trees, two examples being Binary Trees and Binary Search Trees.

- **Binary Trees**: in addition to the above requirements, also hold the binary property constraint. That is, each node has either 0, 1, or 2 children.
- **Binary Search Trees**: in addition to all of the above requirements, also hold the property that For every node X in the tree:
  - Every key in the left subtree is less than X's key.
  - Every key in the right subtree is greater than X's key. **Remember this property!! We will reference it a lot throughout the duration of this module and 61B.



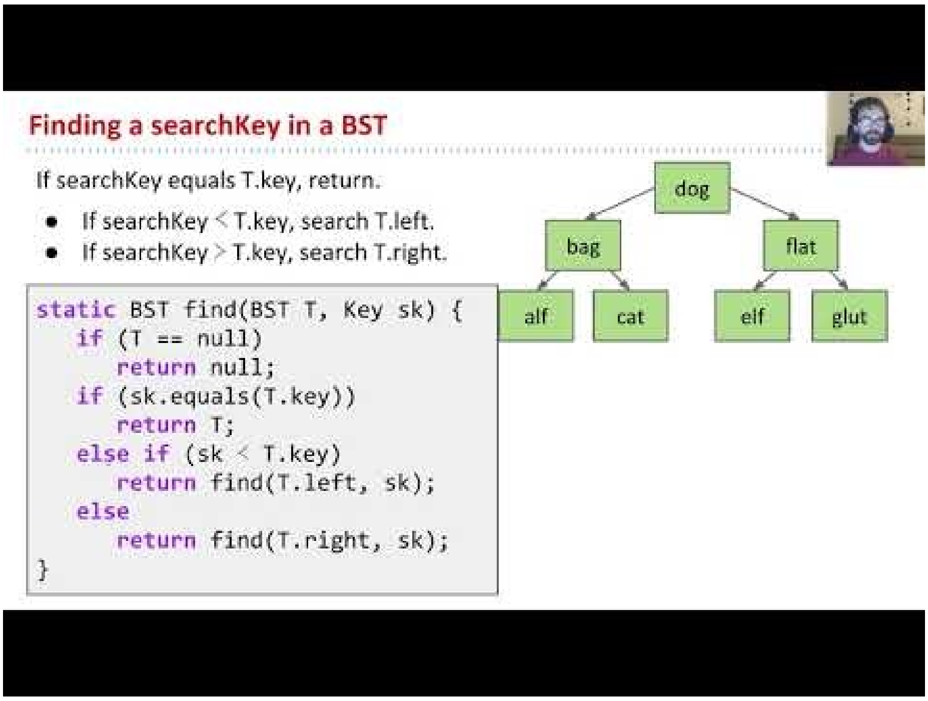
Binary Search Tree

Binary Tree, but not a Binary Search Tree

Here is the BST class we will be using in this module:

```
private class BST<Key> {
    private Key key;
    private BST left;
    private BST right;

    public BST(Key key, BST left, BST Right) {
        this.key = key;
        this.left = left;
        this.right = right;
    }

    public BST(Key key) {
        this.key = key;
    }
}
```

Binary Search Tree Operations



Video link

Search

To search for something, we employ binary search which is made easy due to the BST property described in the previous section!

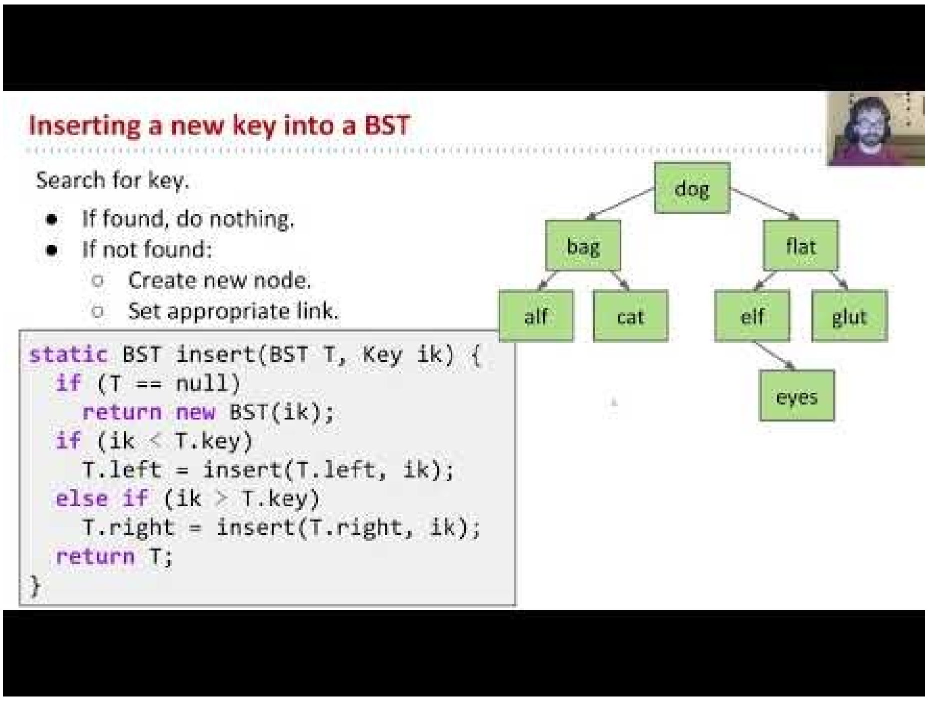
We know that the BST is structured such that all elements to the right of a node are greater and all elements to the left are smaller. Knowing this, we can start at the root node and compare it with the element, X, that we are looking for. If X is greater to the root, we move on to the root's right child. If its smaller, we move on to the root's left child. We repeat this process recursively until we either find the item or we get to a leaf in which case the tree does not contain the item.

Exercise 10.2.2: Try to write this method by yourself. Here is the method header: `static BST find(BST T, Key key)`. It should return the BST rooted at the node whose key matched the key parameter.

```
static BST find(BST T, Key sk) {
   if (T == null)
      return null;
   if (sk.equals(T.key))
      return T;
   else if (sk < T.key)
      return find(T.left, sk);
   else
      return find(T.right, sk);
}
```

If our tree is relatively "bushy", the find operation will run in $\log(n)$ time because the height of the tree is logn, that's pretty fast!

Insert



[Video link]

We **always** insert at a leaf node!

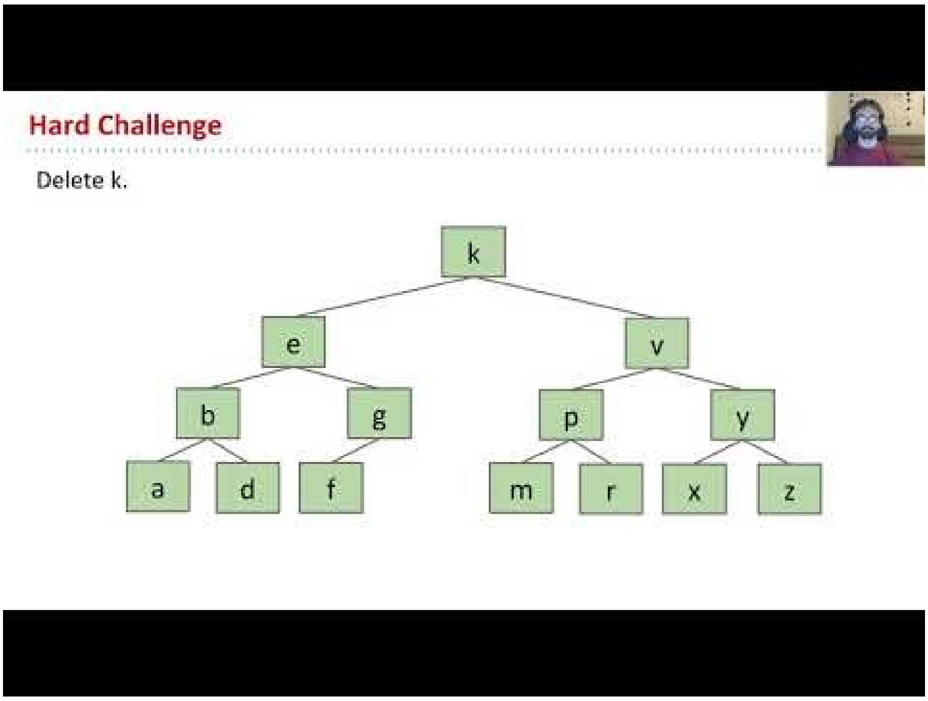
First, we search in the tree for the node. If we find it, then we don't do anything. If we don't find it, we will be at a leaf node already. At this point, we can just add the new element to either the left or right of the leaf, preserving the BST property.

Exercise 10.2.3: Try to write this method by yourself. Here is the method header: `static BST insert(BST T, Key ik)`. It should return the full BST with the new node inserted in the correct position.

```
static BST insert(BST T, Key ik) {
  if (T == null)
    return new BST(ik);
  if (ik < T.key)
    T.left = insert(T.left, ik);
  else if (ik > T.key)
    T.right = insert(T.right, ik);
  return T;
}
```

Exercise 10.2.4: Think of an order of insertions that would result in differing heights of trees. Try to find the two extreme cases for the height of a tree. Hint: Your first insertion will determine much of the behavior for the following insertions.

Delete



Video link

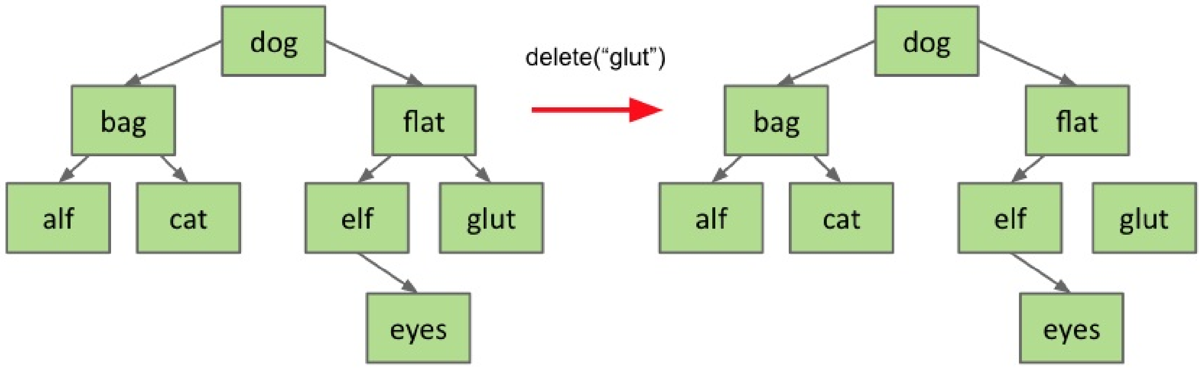
Deleting from a binary tree is a little bit more complicated because whenever we delete, we need to make sure we reconstruct the tree and still maintain its BST property.

Let's break this problem down into three categories:

- the node we are trying to delete has no children
- has 1 child
- has 2 children

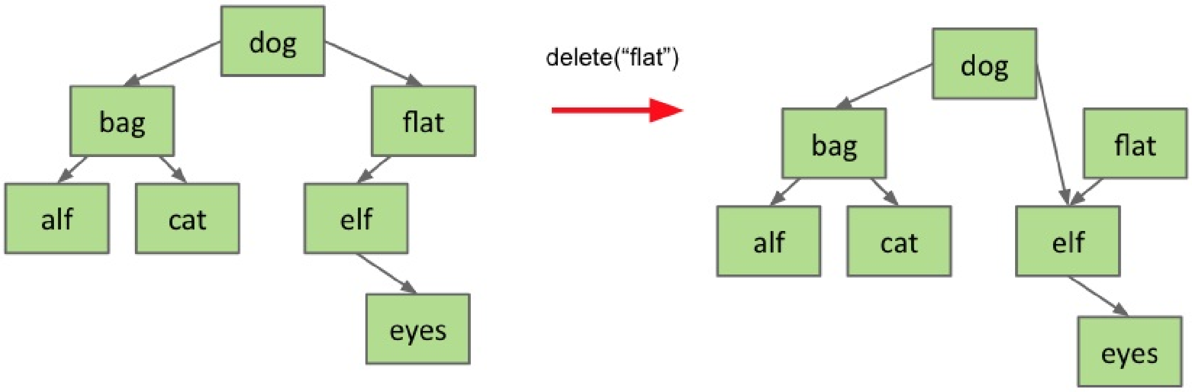
No children

If the node has no children, it is a leaf, and we can just delete its parent pointer and the node will eventually be swept away by the garbage collector.



One child

If the node only has one child, we know that the child maintains the BST property with the parent of the node because the property is recursive to the right and left subtrees. Therefore, we can just reassign the parent's child pointer to the node's child and the node will eventually be garbage collected.



Two children

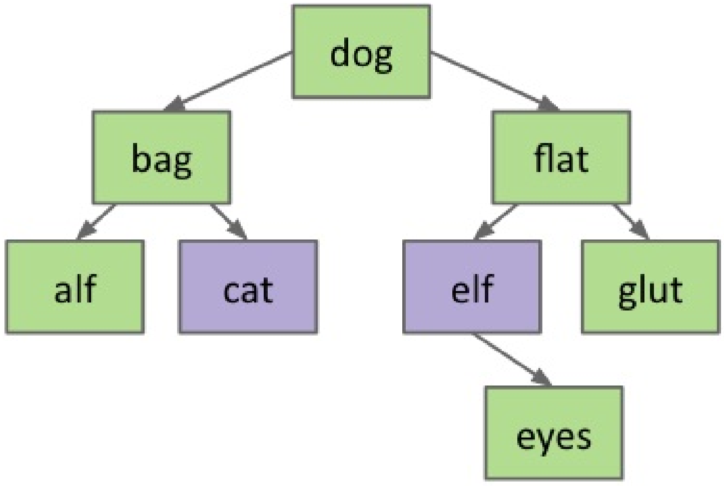
If the node has two children, the process becomes a little more complicated because we can't just assign one of the children to be the new root. This might break the BST property.

Instead, we choose a new node to replace the deleted one.

We know that the new node must:

- be > than everything in left subtree.
- be < than everything right subtree.

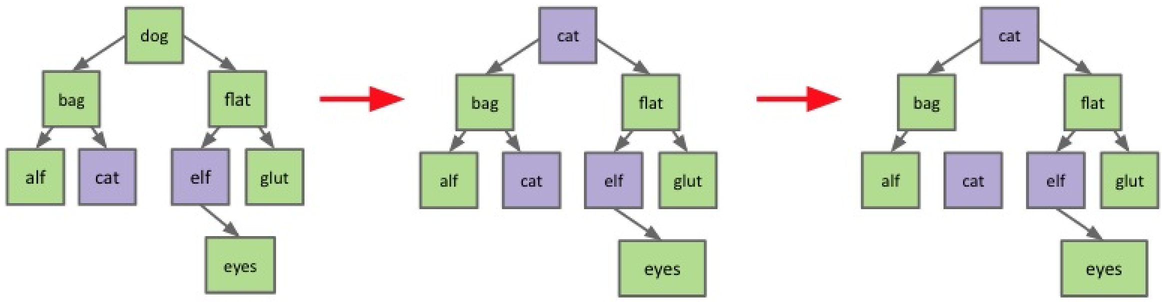
In the below tree, we show which nodes would satisfy these requirements given that we are trying to delete the `dog` node.



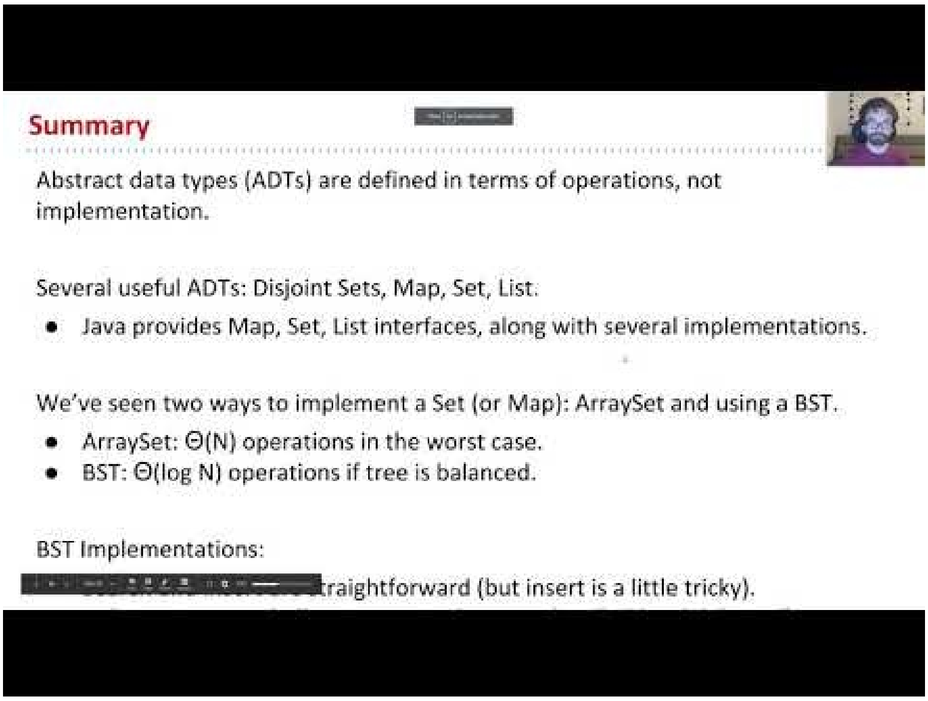
To find these nodes, you can just take the right-most node in the left subtree or the left-most node in the right subtree.

Then, we replace the `dog` node with either `cat` or `elf` and then remove the old `cat` or `elf` node.

This is called **Hibbard deletion**, and it gloriously maintains the BST property amidst a deletion.



BSTs as Sets and Maps



[Video link]

We can use a BST to implement the `Set` ADT! But its even better because in an ArraySet, we have worst-case $O(n)$ runtime to run `contains` because we need to search the entire set. However, if we use a BST, we can decrease this runtime to $\log(n)$ because of the BST property which enables us to use binary search!

We can also make a binary tree into a map by having each BST node hold `(key,value)` pairs instead of singular values. We will compare each element's key in order to determine where to place it within our tree.

Summary

Abstract data types (ADTs) are defined in terms of operations, not implementation.

Several useful ADTs:

- Disjoint Sets, Map, Set, List.
- Java provides Map, Set, List interfaces, along with several implementations.

We've seen two ways to implement a Set (or Map):

- ArraySet: $\Theta(N)$ operations in the worst case.
- BST: $\Theta(\log N)$ operations if tree is balanced.

BST Implementations:

- Search and insert are straightforward (but insert is a little tricky).

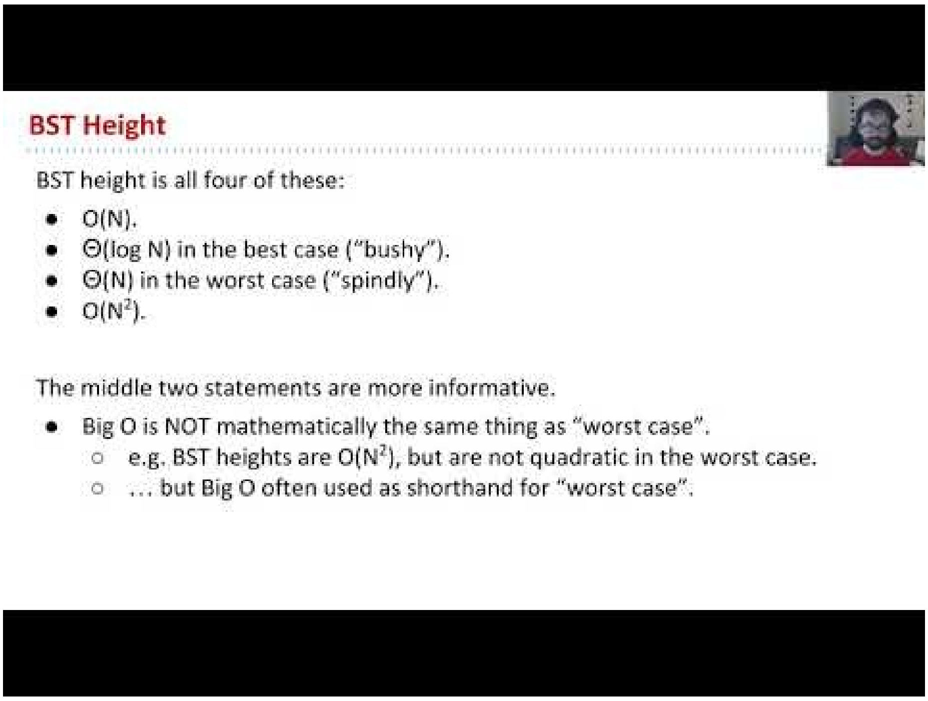
- Deletion is more challenging. Typical approach is “Hibbard deletion”.

What Next

- Lab 7
- Discussion 7

Intro to Balanced Search Trees

Binary Tree Height



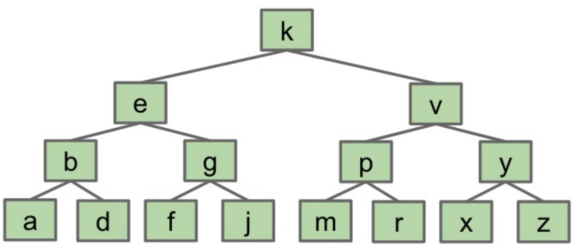
[Video link]

The difference in runtime between a worst-case tree and best-case tree is very dramatic.

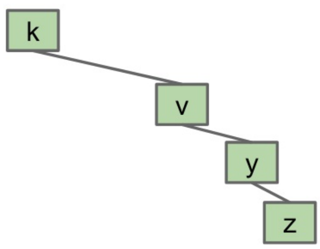
Worst case: $\Theta(N)$

Best-case: $\Theta(\log N)$ (where N is number of nodes in the tree)

The runtimes are dependent on the structure of the tree. If the tree is really spindly, then its basically a linked list and the runtime is linear. If the tree is bushy, then the height of the tree is $\log N$ and therefore the runtime grows in $\log N$ time.



Best case



Worst case

A short detour into BigO and worst case

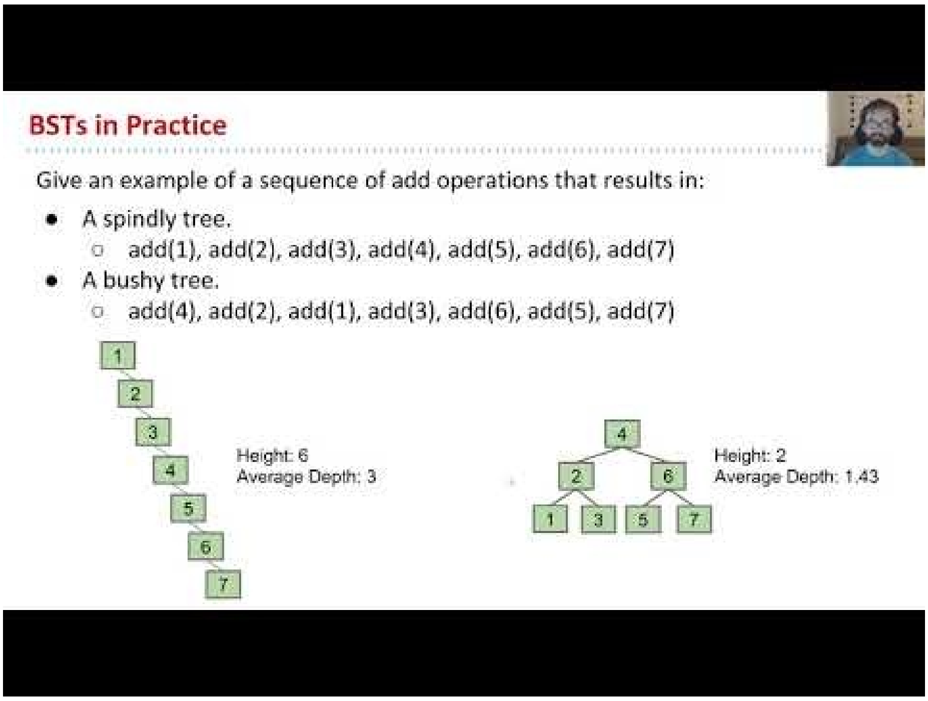
BigO is **not** equivalent to worst case! Remember, BigO is an upper bound. As long as a function falls within that bound, it is considered to be inside the BigO of that function. Worst-case is more restrictive than BigO.

As an example, think about searching for hotels. If you are searching for a hotel with worst-case $500 rooms, then only a few hotels would fit that requirement, perhaps only the ritz carlton. On the other hand, if you were searching for hotels that are in O(500) or less than $500 rooms, then many hotels would fall under that category from the Motel 6 to the Rodeway Inn to the Ritz Carlton.

Thus, even though we said the worst-case runtime of a BST is $\Theta(N)$, it also falls under $O(N^2)$.

Many people use BigO as shorthand for worst-case, but this is technically not synonymous. We want you guys to be cream-of-the-crop computer scientists, so we are making this distinction!

BST Performance



[Video link]

Some terminology for BST performance:

- **depth**: the number of links between a node and the root.
- **height**: the lowest depth of a tree.

- **average depth**: average of the total depths in the tree. You calculate this by taking $\frac{\sum_{i=0}^{D} d_i n_i}{N}$ where d_i is depth and n_i is number of nodes at that depth.

The **height** of the tree determines the worst-case runtime, because in the worst case the node we are looking for is at the bottom of the tree.

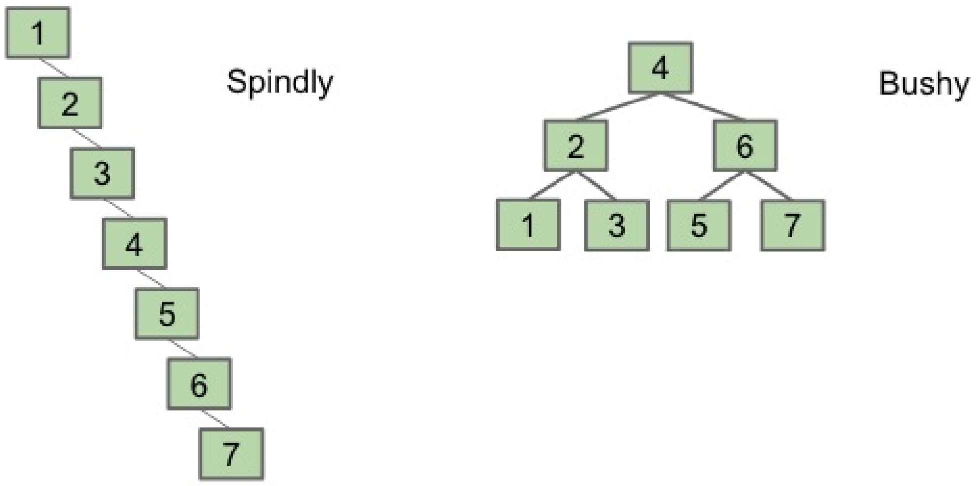
The **average depth** determines the average-case runtime.

BST insertion order

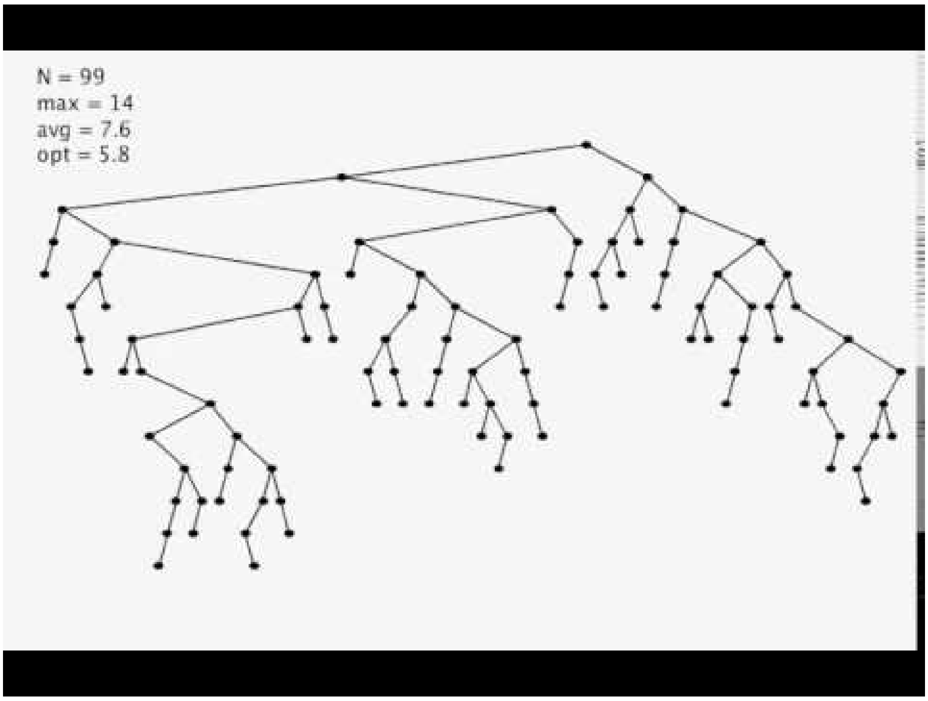
The order you insert nodes into a BST determines its height.

Exercise 11.1.1 Given integers 1-7, what order of inserting would result in the worst-case height? What order would result in the best case?

Answer To get the worst-case height you can just insert in order from 1-7. To get the best case height insert 4 first, then 2 and 6, then 1, 3, 5, 7. Convince yourself of this by actually going through the insertion process.



Yes, having a specific insertion order can help us get a balanced tree. However, we don't even need to be that intentional. If we just insert the nodes in **random** order, it will actually end up being relatively bushy!



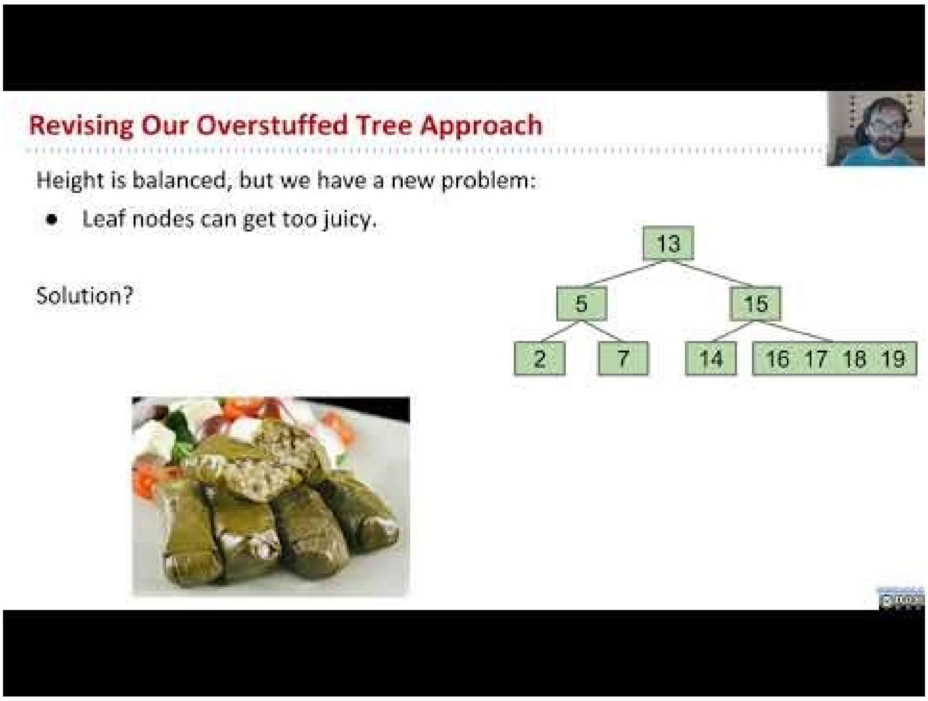
Video link

You don't have to know the proof of this, but when we insert randomly into a BST the **average depth** and **height** are expected to be $\Theta(logN)$.

However, we won't always be able to insert into a BST in random order. What if our data comes in real-time? Then, we will be forced to insert in the order that data comes to us.

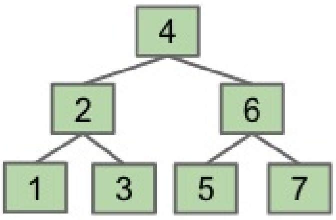
In the next chapter we will learn about a tree that always maintains its balance!

B-Trees



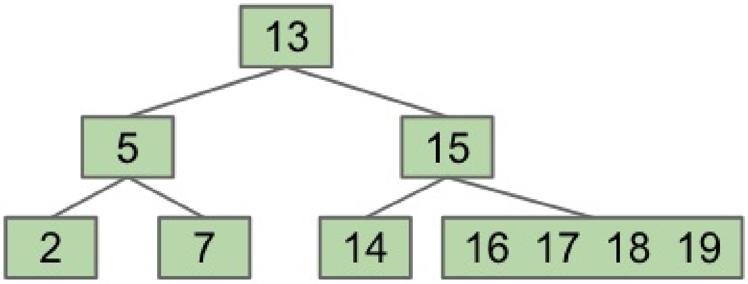
[Video link](#)

The problem with BST's is that we always insert at a leaf node. This is what causes the height to increase. Take this nice balanced tree below:



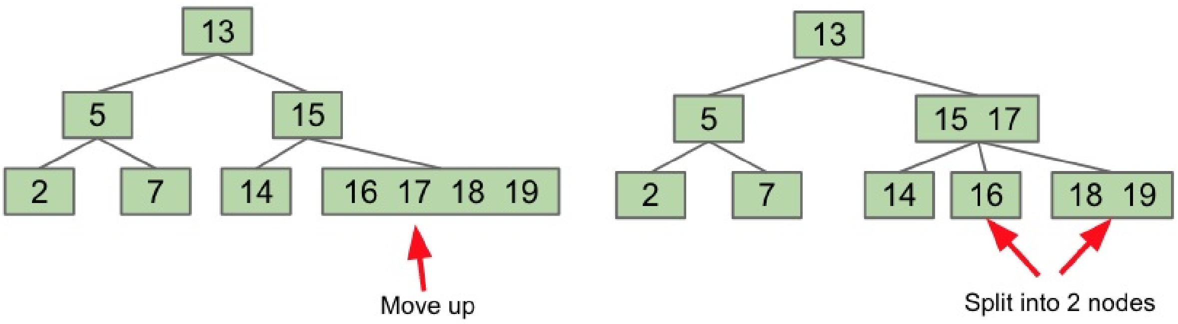
When we start inserting nodes, we could potentially break the balanced structure. So, let's come up with a way to keep the tree balanced when we add new nodes!

Crazy Idea: let's just never add a leaf node! When we insert, let's just add to a current leaf node. This way, the height will never increase.



However, could you see a potential problem with this insertion scheme? If we search for 19, then we will traverse down to the node that contains it, and we still have to look through that node as if we are looking through an array in order to get to the 19 element. This will lead to a runtime of N again!

Solution: Set a limit on the number of elements in a single node. Let's say 4. If we need to add a new element to a node when it already has 4 elements, we will split the node in half. by bumping up the middle left node.



Notice that now the 15-17 node has 3 children now, and each child is either less than, between, or greater than 15 and 17. By keeping the children sorted, we can continue to use binary search.

By splitting nodes in the middle, we maintain perfect balance! These trees are called **B-trees** or **2-3-4/2-3 Trees**. 2-3-4 and 2-3 refer to the number of children each node can have. So, a 2-3-4 tree can have 2, 3, or 4 children while a 2-3 tree can have 2 or 3. This means that 2-3-4 trees split nodes when they have 3 nodes and one more needs to be added. 2-3 trees split after they have 2 nodes and one more needs to be added.

Insertion Process

The process of adding a node to a 2-3-4 tree is:

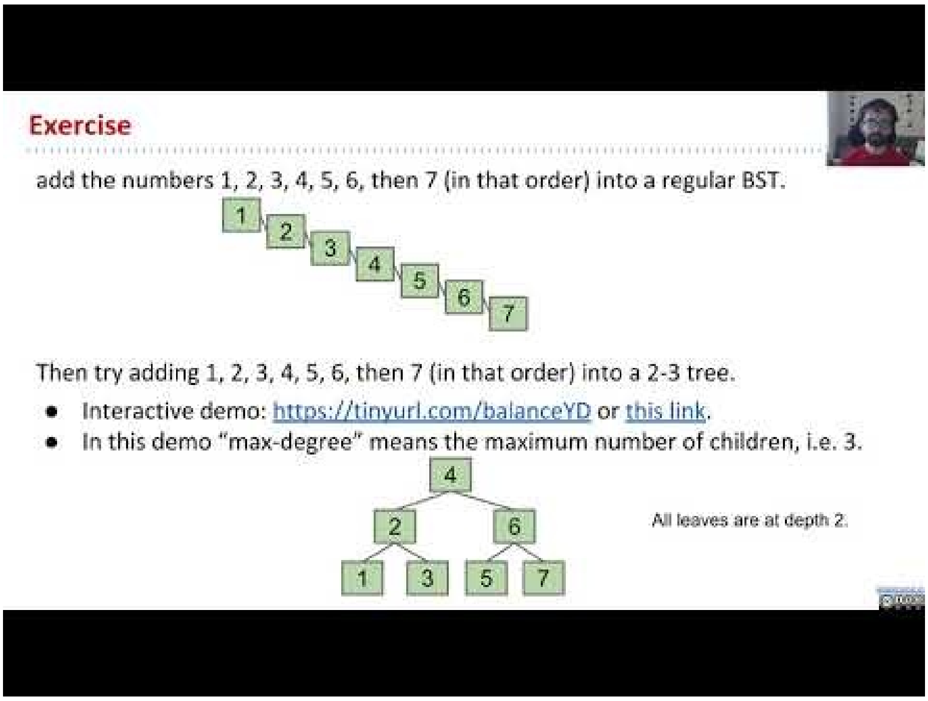
1. We still always inserting into a leaf node, so take the node you want to insert and traverse down the tree with it, going left and right according to whether or not the node to be inserted is greater than or smaller than the items in each node.

2. After adding the node to the leaf node, if the new node has 4 nodes, then pop up the middle left node and re-arrange the children accordingly.
3. If this results in the parent node having 4 nodes, then pop up the middle left node again, rearranging the children accordingly.
4. Repeat this process until the parent node can accommodate or you get to the root.

For a 2-3 tree, go through the same process except push up the middle node in a 3-element node.

If you want some more practice, go through the exercises in these slides

B-Tree invariants



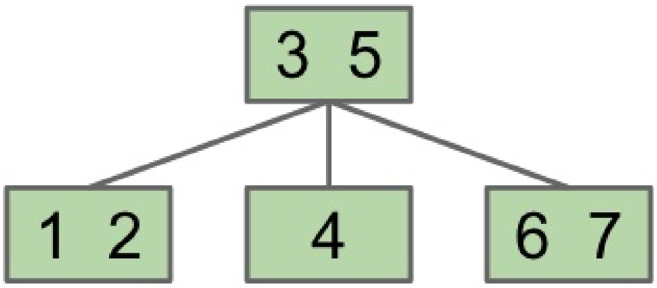
[Video link](#)

We mentioned in chapter 11.1 that order matters when inserting into a BST.

Question: Is this also true for a B-Tree?

Exercise 11.3.1: Insert 1-7 into a B-tree in that order. What is the height of the tree? Can we change the order of the insertions so that we can decrease the height? Here is a [cool B-tree visualizer](https://www.cs.usfca.edu/~galles/visualization/BTree.html\) that might help!

Solution: Yes, we can get a tree of height 1 by inserting in this order: 2, 3, 4, 5, 6, 1, 7.



Yes, depending on the order you insert nodes the height of a B-tree may change. However, the tree will always be **bushy.**

A B-tree has the following helpful invariants:

- All leaves must be the same distance from the source.

- A non-leaf node with k items must have exactly $k+1$ children.

In tandem, these invariants cause the tree to always be bushy.

B-Tree runtime analysis

Video link

The worst-case runtime situation for search in a B-tree would be if each node had the maximum number of elements in it and we had to traverse all the way to the bottom. We will use L to denote the number of elements in each node. This means would would need to explore $\log N$ nodes (since the max height is $\log N$ due to the bushiness invariant) and at each node we would need to explore L elements. In total, we would need to run $L \log N$ operations. However, we know L is a constant, so our total runtime is $O(\log N)$.

B-Tree deletion (Extra)

See these extra slides if you're curious. We won't discuss them here.

Summary

BSTs have best case height $\Theta(\log N)$, and worst case height $\Theta(N)$.

- Big O is not the same thing as worst case!

B-Trees are a modification of the binary search tree that avoids $\Theta(N)$ worst case.

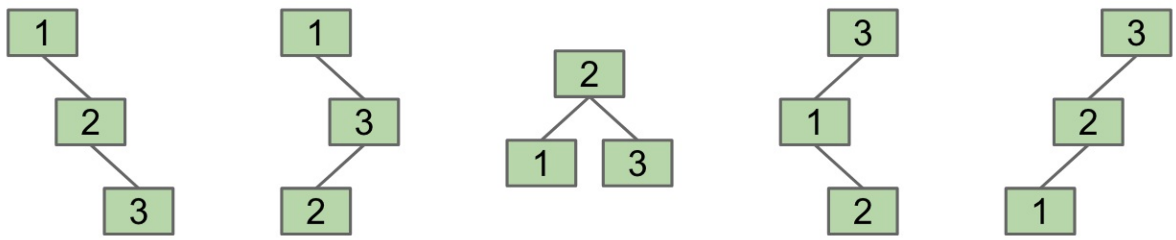
- Nodes may contain between 1 and L items.
- contains works almost exactly like a normal BST.
- add works by adding items to existing leaf nodes.
  - If nodes are too full, they split.
- Resulting tree has perfect balance. Runtime for operations is $O(\log N)$.
- Have not discussed deletion. See extra slides if you're curious.
- Have not discussed how splitting works if $L > 3$ (see some other class).
- B-trees are more complex, but they can efficiently handle ANY insertion order.

Rotating Trees

Wonderfully balanced as they are, B-Trees are really difficult to implement. We need to keep track of the different nodes and the splitting process is pretty complicated. As computer scientists who appreciate good code and a good challenge, let's find another way to create a balanced tree.

BST structure

For any BST, there are multiple ways to structure it so that you maintain the BST invariants. In chapter 11.1 we talked about how **inserting** elements in different orders will result in a different BST. The BST's below all consist of the elements 1, 2, and 3, yet all have different structures.



Exercise 11.4.1: For each tree shown above, provide an order of insertion that yields the structure.

However, insertion is not the only way to yield different structures for the same BST. One thing we can do is change the tree with the nodes already in place through a process called **rotating**.

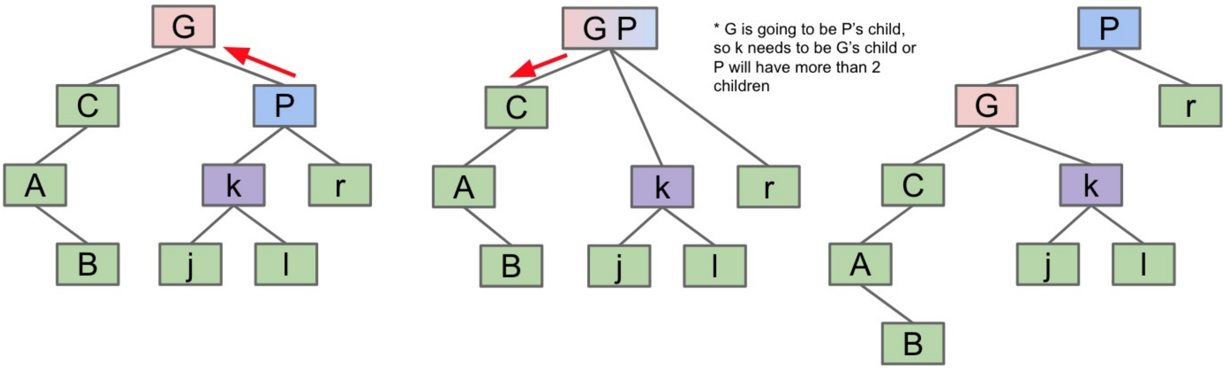
Tree Rotation

The formal definition of rotation is:

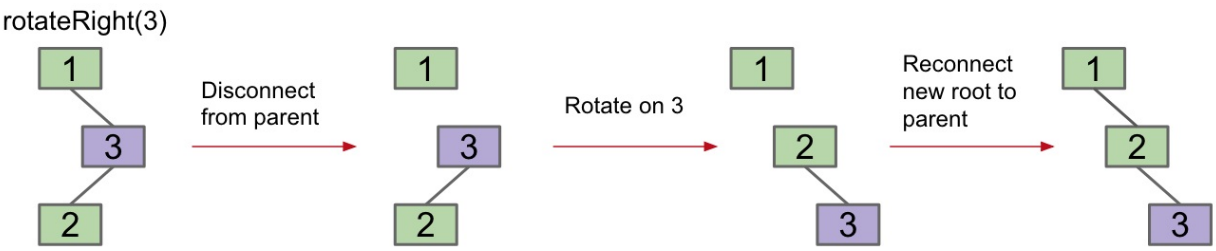
`rotateLeft(G): Let x be the right child of G. Make G the new left child of x.`

`rotateRight(G): Let x be the left child of G. Make G the new right child of x.`

We will slowly demystify this process in the next few paragraphs. Below is a graphical description of what happens in a left rotation on the node G.



G's right child, P, merges with G, bringing its children along. P then passes its left child to G and G goes down to the left to become P's left child. You can see that the structure of the tree changes as well as the number of levels. We can also rotate on a non-root node. We just disconnect the node from the parent temporarily, rotate the subtree at the node, then reconnect the new root.



Here are the implementations of `rotateRight` and `rotateLeft` courtesy of the [princeton docs](https://algs4.cs.princeton.edu/33balanced/RedBlackBST.java.html\) with some lines omitted for simplicity.

```
private Node rotateRight(Node h) {
    // assert (h != null) && isRed(h.left);
    Node x = h.left;
    h.left = x.right;
    x.right = h;
    return x;
}

// make a right-leaning link lean to the left
private Node rotateLeft(Node h) {
    // assert (h != null) && isRed(h.right);
    Node x = h.right;
    h.right = x.left;
    x.left = h;
    return x;
}
```

Let's practice this on a few more examples.

Exercise 11.4.2: Give a sequence of rotations (`rotateRight(G)` , `rotateLeft(G)`) that will convert the tree on the left to the tree on the right.

Solution: `rotateRight(3)` , `rotateLeft(1)`

With rotations, we can actually completely balance a tree. See the demo in these slides: https://docs.google.com/presentation/d/1pfkQENflBwiThGGFVO5xvlVp7XAUONI2BwBqYxib0A4/edit#slide=id.g465b5392c_00\

In the next chapter, we will learn about a specific tree data structure that remains balanced through using rotations.

Red-Black Trees

We said in the previous section that we really like 2-3 trees because they always remain balanced, but we also don't like them because they are hard to implement. But why not both? Why not create a tree that is implemented using a BST, but is structurally identical to a 2-3 tree and thus stays balanced? (Note that in this chapter we will be honing in on 2-3 Trees specifically, not 2-3-4 trees)

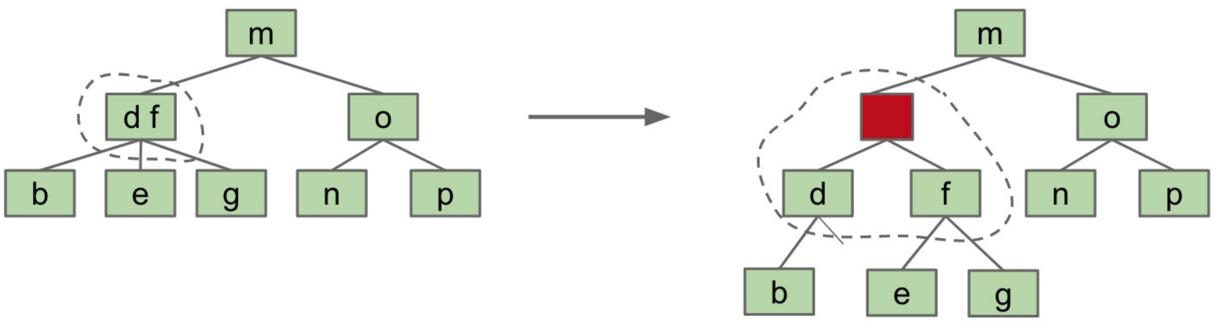
Enter the Red-Black Tree

We are going to create this tree by looking at a 2-3 tree and asking ourselves what kind of modifications we can make in order to convert it into a BST.

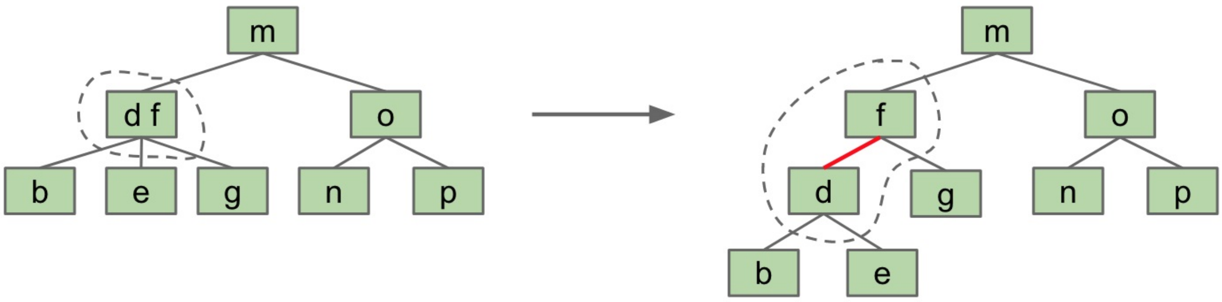
For a 2-3 tree that only has 2-nodes (nodes with 2 children), we already have a BST, so we don't need to make any modifications!

However, what happens when we get a 3-node?

One thing we could do is create a "glue" node that doesn't hold any information and only serves to show that its 2 children are actually a part of one node.



However, this is a very inelegant solution because we are taking up more space and the code will be ugly. So, instead of using glue nodes we will use glue links instead!



We choose arbitrarily to make the left element a child of the right one. This results in a **left-leaning** tree. We show that a link is a glue link by making it red. Normal links are black.

Because of this, we call these structures **left-leaning red-black trees (LLRB)**. We will be using left-leaning trees in 61B.

Left-Leaning Red-Black trees have a 1-1 correspondence with 2-3 trees. Every 2-3 tree has a unique LLRB red-black tree associated with it. As for 2-3-4 trees, they maintain correspondence with standard Red-Black trees.

Properties of LLRB's

Here are the properties of LLRB's:

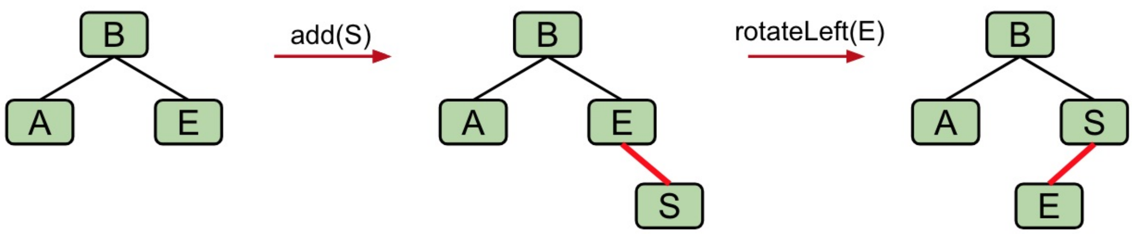
- 1-1 correspondence with 2-3 trees.
- No node has 2 red links.
- There are no red right-links.
- Every path from root to leaf has same number of black links (because 2-3 trees have same number of links to every leaf).
- Height is no more than 2x height of corresponding 2-3 tree.

Inserting into LLRB

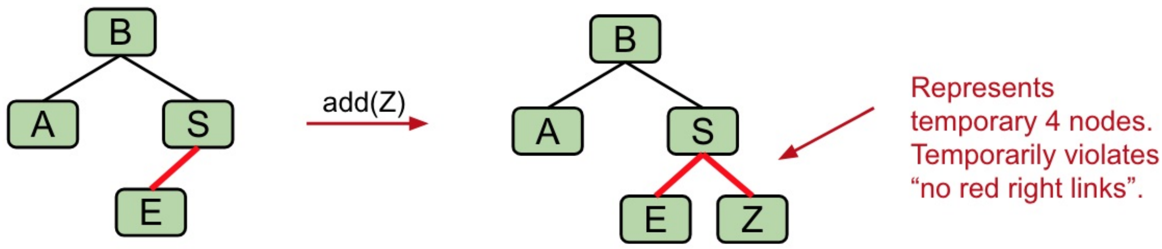
We can always insert into a LLRB tree by inserting into a 2-3 tree and converting it using the scheme from above. However, this would be contrary to our original purpose of creating a LLRB, which was to avoid the complicate code of a 2-3 tree! Instead, we insert into the LLRB as we would with a normal BST. However, this could break its 1-1 mapping to a 2-3 tree, so we will use rotations to massage the tree back into a proper structure.

We will go over the different tasks we need to address when inserting into a LLRB below.

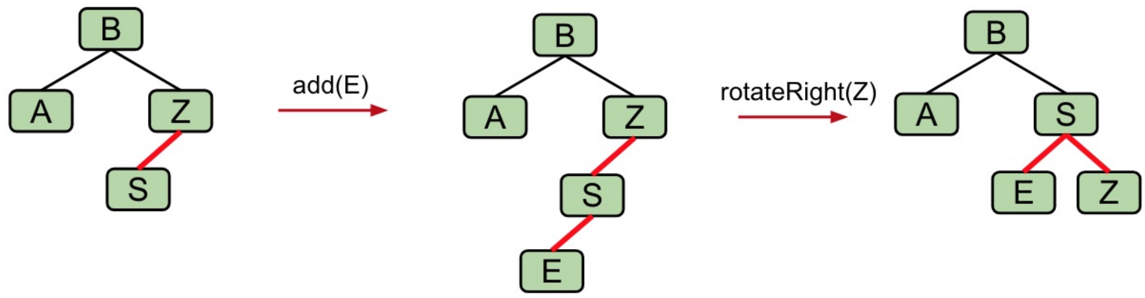
1. **Task 1: insertion color**: because in a 2-3 tree, we are always inserting by adding to a leaf node, the color of the link we add should always be red.
2. **Task 2: insertion on the right**: recall, we are using *left-leaning* red black trees, which means we can never have a right red link. If we insert on the right, we will need to use a rotation in order to maintain the LLRB invariant.



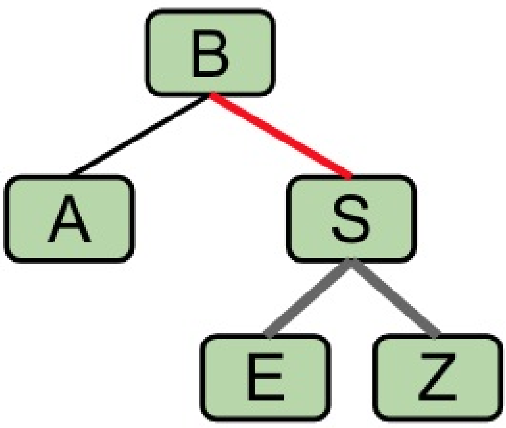
However, if we were to insert on the right with a red link and the left child is *also* a red link, then we will temporarily allow it for purposes that will become clearer in task 3.



3. **Task 3: double insertion on the left:** If there are 2 left red links, then we have a 4-node which is illegal. First, we will rotate to create the same tree seen in task 2 above.



   Then, in both situations, we will flip the colors of all edges touching S. This is equivalent to pushing up the middle node in a 2-3 tree.



You may need to go through a series of rotations in order to complete the transformation. The process is: while the LLRB tree does not satisfy the 1-1 correspondence with a 2-3 tree or breaks the LLRB invariants, perform task 1, 2, or 3 depending on the condition of the tree until you get a legal LLRB.

Here is a summary of all the operations:

- When inserting: Use a red link.
- If there is aright leaning “3-node”, we have a Left Leaning Violation
  - Rotate left the appropriate node to fix.
- If there are two consecutive left links, we have an incorrect 4 Node Violation!

- Rotate right the appropriate node to fix.
- If there are any nodes with two red children, we have a temporary 4 Node.
    - Color flip the node to emulate the split operation.

Runtime

Because a left-leaning red-black tree has a 1-1 correspondence with a 2-3 tree and will always remain within 2x the height of its 2-3 tree, the runtimes of the operations will take $\log N$ time.

Here's the abstracted code for insertion into a LLRB:

```
private Node put(Node h, Key key, Value val) {
    if (h == null) { return new Node(key, val, RED); }

    int cmp = key.compareTo(h.key);
    if (cmp < 0)      { h.left  = put(h.left,  key, val); }
    else if (cmp > 0) { h.right = put(h.right, key, val); }
    else              { h.val   = val;                    }

    if (isRed(h.right) && !isRed(h.left))      { h = rotateLeft(h);  }
    if (isRed(h.left)  &&  isRed(h.left.left)) { h = rotateRight(h); }
    if (isRed(h.left)  &&  isRed(h.right))     { flipColors(h);      }

    return h;
}
```

Look how short and sweet!

Summary

- Binary search trees are simple, but they are subject to imbalance which leads to crappy runtime.
- 2-3 Trees (B Trees) are balanced, but painful to implement and relatively slow.
- LLRBs insertion is simple to implement (but deletion is hard).
    - Works by maintaining mathematical bijection with a 2-3 trees.
- Java's TreeMap is a red-black tree (but not left leaning).
- LLRBs maintain correspondence with 2-3 tree, Standard Red-Black trees maintain correspondence with 2-3-4 trees.

- Allows glue links on either side (see Red-Black Tree).
- More complex implementation, but significantly faster.

Hashing

Issues with what we've seen so far

So far, we've looked at a few data structures for efficiently searching for the existence of items within the data structure. We looked at Binary Search Trees, then made them balanced using 2-3 Trees.

However, there are some limitations that these structures impose (yes, even 2-3 trees!)

1. They require that items be comparable. How do you decide where a new item goes in a BST? You have to answer the question "are you smaller than or bigger than the root"? For some objects, this question may make no sense.
2. They give a complexity of $\Theta(\log N)$. Is this good? Absolutely. But maybe we can do better.

A first attempt: `DataIndexedIntegerSet`

Let us begin by considering the following approach.

For now, we're only going to try to improve issue #2 above (improve complexity from $\Theta(\log N)$ to $\Theta(1)$. We're going to not worry about issue #1 (comparability). In fact, we're going to only consider storing and searching for `int` s.

Here's an idea: let's create an ArrayList of type `boolean` and size 2 billion. Let everything be false by default.

- The `add(int x)` method simply sets the `x` position in our ArrayList to true. This takes $\Theta(1)$ time.
- The `contains(int x)` method simply returns whether the `x` position in our ArrayList is `true` or `false`. This also takes $\Theta(1)$ time!

```
public class DataIndexedIntegerSet {
    private boolean[] present;

    public DataIndexedIntegerSet() {
        present = new boolean[2000000000];
    }

    public void add(int x) {
        present[i] = true;
    }

    public boolean contains(int x) {
        return present[i];
    }
```

There, we have it. That's all folks.

Well, not really. What are some potential **issues** with this approach?

- Extremely wasteful. If we assume that a `boolean` takes 1 byte to store, the above needs `2GB` of space per `new DataIndexedIntegerSet()`. Moreover, the user may only insert a handful of items...
- What do we do if someone wants to insert a `String`?
    - Let's look at this next. Of course, we may want to insert other things, like `Dog` s. That'll come soon!

Solving the word-insertion problem

Our `DataIndexedIntegerSet` only allowed for integers, but now we want to insert the `String` `"cat"` into it. We'll call our data structure that can insert strings `DataIntexedEnglishWordSet` Here's a crazy idea: let's give every string a number. Maybe "cat" can be `1`, "dog" can be `2` and "turtle" can be `3`.

(The way this would work is — if someone wanted to add a "cat" to our data structure, we would 'figure out' that the number for "cat" is 1, and then set `present[1]` to be `true`. If someone wanted to ask us if "cat" is in our data structure, we would 'figure out' that "cat" is 1, and check if `present[1]` is true.)

But then if someone tries to insert the word "potatocactus", we'll not know what to do. We need to develop a general strategy so that given a string, we can figure out a number representation for it.

Strategy 1: Use the first letter.

A simple idea is to just use the first character of any given string to convert it to its number representation. So "cat" -> "c" -> 3. "Dog" -> "d" -> 4. But also, "drum" -> "d" -> 4.

What if someone wanted to insert "dog" and "drum" into our `DataIntexedEnglishWordSet`? All bets are off, and we don't know how to do it.

Note that when two different inputs ("dog" and "drum") map to the same integer, we call that a **collision**. We don't know how to deal with collisions yet, so let's figure out a way to avoid them. (What we do with most of our problems lol...)

Strategy 2: Avoiding collisions

To motivate this part, let's understand how our number system works.

A four digit number, say 5149, can be written as

$$5 \cdot 10^3 + 1 \cdot 10^2 + 4 \cdot 10^1 + 9 \cdot 10^0.$$

Actually, **any** 4 digit number can be written **uniquely** in this form. What that means is given 4 digits, a, b, c, d, we can write $a \cdot 10^3 + b \cdot 10^2 + c \cdot 10^1 + d \cdot 10^0$ and that gives us a unique 4 digit number: $abcd$.

Notice that the 10 is important here. If we chose a bad number, say 2, the same is not true. Let's make sure we see what happens if we chose 2 as the multiplier.

$a, b, c, d = 1, 1, 1, 1$ gives $1 \cdot 2^3 + 1 \cdot 2^2 + 1 \cdot 2^1 + 1 \cdot 2^0 = 15$.

$a, b, c, d = 0, 3, 1, 1$ gives $0 \cdot 2^3 + 3 \cdot 2^2 + 1 \cdot 2^1 + 1 \cdot 2^0 = 15$.

A collision on inputs (1, 1, 1, 1) and (0, 3, 1, 1)!

So why is 10 important? It's because there are 10 unique digits in our decimal system: $0, 1, 2, 3, 4, 5, 6, 7, 8, 9$.

Similarly, there are 26 unique characters in the english lowercase alphabet. Why not give each one a number: $a = 1, b = 2, \ldots, z = 26$. Now, we can write any unique lowercase string in **base 26**. (Note that **base 26** simply means that we will use **26** as the multiplier, much like we used **10** and **2** as examples above.)

- "cat" = "c" 26^2 + *'a'* 26^1 + 't' 26^0_ = $3_26^2 + 1_26^1 + 20_26^0 = 2074$.

Quick check

- How do you represent "dog"?

This representation gives a unique integer to every english word containing lowercase letters, much like using base 10 gives a unique representation to every number. We are guaranteed to not have collisions.

Our Data Structure `DataIndexedEnglishWordSet`

```
public class DataIndexedEnglishWordSet {
    private boolean[] present;

    public DataIndexedEnglishWordSet() {
        present = new boolean[2000000000];
    }

    public void add(String s) {
        present[englishToInt(s)] = true;
    }

    public boolean contains(int i) {
        resent present[englishToInt(s)];
    }
}
```

Uses a helper method

```
public static int letterNum(String s, int i) {
    /** Converts ith character of String to a letter number.
    * e.g. 'a' -> 1, 'b' -> 2, 'z' -> 26 */
    int ithChar = s.charAt(i)
    if ((ithChar < 'a') || (ithChar > 'z')) {
        throw new IllegalArgumentException();
    }

    return ithChar - 'a' + 1;
}

public static int englishToInt(String s) {
    int intRep = 0;
    for (int i = 0l i < s.length(); i += 1) {
        intRep = intRep * 26;
        intRep += letterNum(s, i);
    }

    return intRep;
}
```

Where are we?

Recall, we started with wanting to

(a) Be better than $\Theta(\log N)$. We've now done this for integers and for single english words.

(b) Allow for non-comparable items. We haven't touched this yet, although we are getting there. So far, we've only learnt how to add integers and english words, both of which *are* comparable, **but**, have we ever **used** the fact that they are comparable? I.e., have we ever tried to compare them (like we did in BSTs)? No. So we're getting there, but we haven't actually inserted anything non-comparable yet.

(c) We have data structures that insert integers and english words. Let's make a quick visit to inserting arbitrary `String` objects, with spaces and all that. And maybe even insert other languages and emojis!

(d) Further recall that our approach is still very wasteful of memory. We haven't solved that issue yet!

Inserting `String` s beyond single english words

33	!	49	1	65	A	81	Q	97	a	113	q
34	"	50	2	66	B	82	R	98	b	114	r
35	#	51	3	67	C	83	S	99	c	115	s
36	$	52	4	68	D	84	T	100	d	116	t
37	%	53	5	69	E	85	U	101	e	117	u
38	&	54	6	70	F	86	V	102	f	118	v
39	'	55	7	71	G	87	W	103	g	119	w
40	(	56	8	72	H	88	X	104	h	120	x
41	)	57	9	73	I	89	Y	105	i	121	y
42	*	58	:	74	J	90	Z	106	j	122	z
43	+	59	;	75	K	91	[	107	k	123	{
44	,	60	<	76	L	92	\	108	l	124	|
45	-	61	=	77	M	93	]	109	m	125	}
46	.	62	>	78	N	94	^	110	n	126	~
47	/	63	?	79	O	95	_	111	o		
48	0	64	@	80	P	96	`	112	p		

biggest value is 126

There is a character format called **ASCII**, which has an integer per character. Here, we see that the largest value (i.e., the base/multiplier we need to use) is 126. Let's just do that. The same thing as `DataIndexedEnglishWordSet` , but just with base `126` .

```
public static int asciiToInt(String s) {
    int intRep = 0;
    for (int i = 0; i < s.length(); i += 1) {
        intRep = intRep * 126;
        intRep = intRep + s.charAt(i);
    }
    return intRep;
}
```

What about adding support for Chinese? The largest possible representation is 40959, so we need to use that as the base. Here's an example:

Example:

- $\text{守门员}_{40959} = (23432 \times 40959^{2}) + (38376 \times 40959^{1}) + (21592 \times 40959^{0}) = 39{,}312{,}024{,}869{,}368$

So... to store a 3 character Chinese word, we need an array of size larger than **39 trillion** (with a T)!. This is getting out of hand... so let's explore what we can do.

Handling Integer Overflow and Hash Codes

Overflow issues

The largest possible value for integers in Java is 2,147,483,647. The smallest value is -2,147,483,64**8**.

If you try to take the largest value and add 1, you get the smallest value!

```
int x = 2147483647;
System.out.println(x);
System.out.println(x + 1);
```

```
jug ~/Dropbox/61b/lec/hashing
$ javac BiggestPlusOne.java
$ java BiggestPlusOne
2147483647
-2147483648
```

So, we will run into problems, even with just ASCII characters (which are in base 126, remember).

$omens_{126} = 28,196,917,171.$ `asciiToInt(omens)` returns `-1,867,853,901` instead.

`melt banana` and `subterresetrial anticosmetic` actually have the same representation according to `asciiToInt` because of overflow. So if we added `melt banana` and then tried to ask `contains(subterrestrial anticosmetic)`, we would get `true`.

The inevitable truth.

From the smallest to the largest possible integers, there are a total of 4,294,967,296 integers in Java. Yet, there are more than that many total objects that could be created in Java, and so collision is inevitable. Resistance is futile. We **must** figure out how to deal with collision head-on, instead of trying to work around it.

(If you don't believe that there are more than 4 billions objects one could create in Java, just consider: "one", "two", ..., "five trillion" — each of which is a unique string.)

We must handle collisions.

A subtle point

Note that our problem is not inherently the fact that overflow *exists*. All we wanted was for a way to convert a `String` to a number. Even if overflow *exists*, we do manage to convert a `String` to a number. The inherent problem is caused by the fact that *overflow causes collisions*, which we don't know how to deal with.

Overflow *is* often bad in other contexts, for instance, it has some unexpected results if you don't know that overflow happens. But here, overflow's existence doesn't ruin the fact that we wanted to convert a `String` to an `int`. So, we have that going for us.

Hash Codes

In computer science, taking an object and converting it into some integer is called "computing the **hash code** of the object". For instance, the hashcode of "melt banana" is 839099497.

We looked at how to compute this hashcode for Strings. For other Objects, there are one of two things we do:

- Every Object in Java has a default `.hashcode()` method, which we can use. Java computes this by figuring out where the `Object` sits in memory (every section of the memory in your computer has an address!), and uses that memory's address to do something similar to what we did with `String`s. This methods gives a *unique* hashcode for every single Java object.
- Sometimes, we write our own `hashcode` method. For example, given a `Dog`, we may use a combination of its `name`, `age` and `breed` to generate a `hashcode`.

Properties of HashCodes

Hash codes have three necessary properties, which means a hash code must have these properties in order to be **valid**:

1. It must be an Integer
2. If we run `.hashCode()` on an object twice, it should return the **same** number
3. Two objects that are considered `.equal()` must have the same hash code.

Not all hash codes are created equal, however. If you want your hash code to be considered a **good** hash code, it should:

1. Distribute items evenly

Note that at this point, we know how to add arbitrary objects to our data structure, not only strings.

Pending issues

- Space: we still haven't figured out how to use less space.
- Handling Collisions: we have determined that we need to handle collisions, but we haven't actually handled them yet.

Everything else has been solved!

Handling Collisions

Time to address the elephant in the room. The big idea is to change our array ever so slightly to not contain just items, but instead contain a LinkedList (or any other List) of items. So...

Everything in the array is originally empty.
If we get a new item, and its hashcode is h:

- If there is nothing at index h at the moment, we'll create a new `LinkedList` for index h, place it there, and then add the new item to the newly created `LinkedList`.
- If there is already something at index h, then there is already a `LinkedList` there. We simply add our new item to that `LinkedList`. **Note: Our data structure is not allowed to have any duplicate items / keys. Therefore, we must first check whether the item we are trying to insert is already in this LinkedList. If it is, we do nothing! This also means that we will insert to the END of the linked list, since we need to check all of the elements anyways.**

Concrete workflow

- `add` item
    - Get hashcode (i.e., index) of item.
    - If index has no item, create new List, and place item there.
    - If index has a List already, check the List to see if item is already in there. If *not*, add item to List.
- `contains` item
    - Get hashcode (i.e., index) of item.
    - If index is empty, return `false`.
    - Otherwise, check all items in the List at that index, and if the item exists, return `true`.

Runtime Complexity

Worst case time	contains(x)	add(x)
Bushy BSTs	Θ(log N)	Θ(log N)
DataIndexedArray	Θ(1)	Θ(1)
Separate Chaining Data Indexed Array	Θ(Q)	Θ(Q)

Q: Length of longest list

Why Q and not 1?

Why is contains $\Theta(Q)$?
Because we need to look at all the items in the LinkedList at the hashcode (i.e., index).

Why is add $\Theta(Q)$?
Can't we just add to the beginning of the LinkedList, which takes $\Theta(1)$ time? No! Because **we have to check to make sure the item isn't already in the linked list.**

You gain some, you lose some.

- Space: Still unsolved.
- Handling collisions: done.
- Runtime complexity? We've lost some. In the worst case, all of our items' `hashcode` could be the same, and so they all go to the same index. If we have N items, it's possible that they **all** go to the same index, creating a linked list of length N, providing a runtime of $\Theta(N)$.

Solving space

Why keep an ArrayList of size 4 billion around? Recall that we did that to avoid collisions, because we wanted to be able to add every integer / word / `String` to our data structure. But now that we allow for collisions *anyway*, we can relax this a bit.

An idea: modulo. Let's just create an ArrayList of size, say, 100. Let's not change how the `hashcode` functions behaves (let it return a crazy large integer.) But after we get the `hashcode`, we'll take its modulo 100 to get an index within the $0 \ldots 99$ range that we want. And if collisions happen? Doesn't matter, we know how to deal with it!

Do note that our LinkedLists within the array will now be longer, because we're taking all the items spread across the 4 billions indices and compressing them into a 100 indices.

Where we are

- Space: Has been solved.
- Handling collisions: Done!
- Runtime complexity? We lost some earlier at $\Theta(Q)$ for `add` and `contains`, and then in the `Solving space` section, we realized that we lost some more because our LinkedLists will potentially be larger (so `Q` will be larger.)

Our Final Data Structure: `HashTable`

What we've created now is called a `HashTable`.

- Inputs are converted by a hash function (`hashcode`) into an integer. Then, they're converted to a valid index using the modulus operator. Then, they're added at that index (dealing with collisions using LinkedLists).
- `contains` works in a similar fashion by figuring out the valid index, and looking in the corresponding LinkedList for the item.

Dealing with Runtime

The only issue left to solve is the issue of runtime. If we have 100 items, and our ArrayList is of size 5, then

- In the best case, all items get sent to the different indices evenly. That is, we have 5 linkedLists, and each one contains 20 of the items.
- In the worst case, all items get sent to the same index! That is, we have just 1 LinkedList, and it has all 100 items.

There are two ways to try to fix this:

- Dynamically growing our hashtable.
- Improving our Hashcodes

Dynamically growing the hash table

Suppose we have M buckets (indices) and N items. We say that our **load factor** is N/M.

(Note that the **load factor** is equivalent to our **best** case runtime from above.)

So... we have incentive to keep our load factor low (after all, it is the best runtime we could possible achieve!).

And note that if we keep M (the number of buckets) fixed, and N keeps increasing, the load factor consistently keeps increasing.

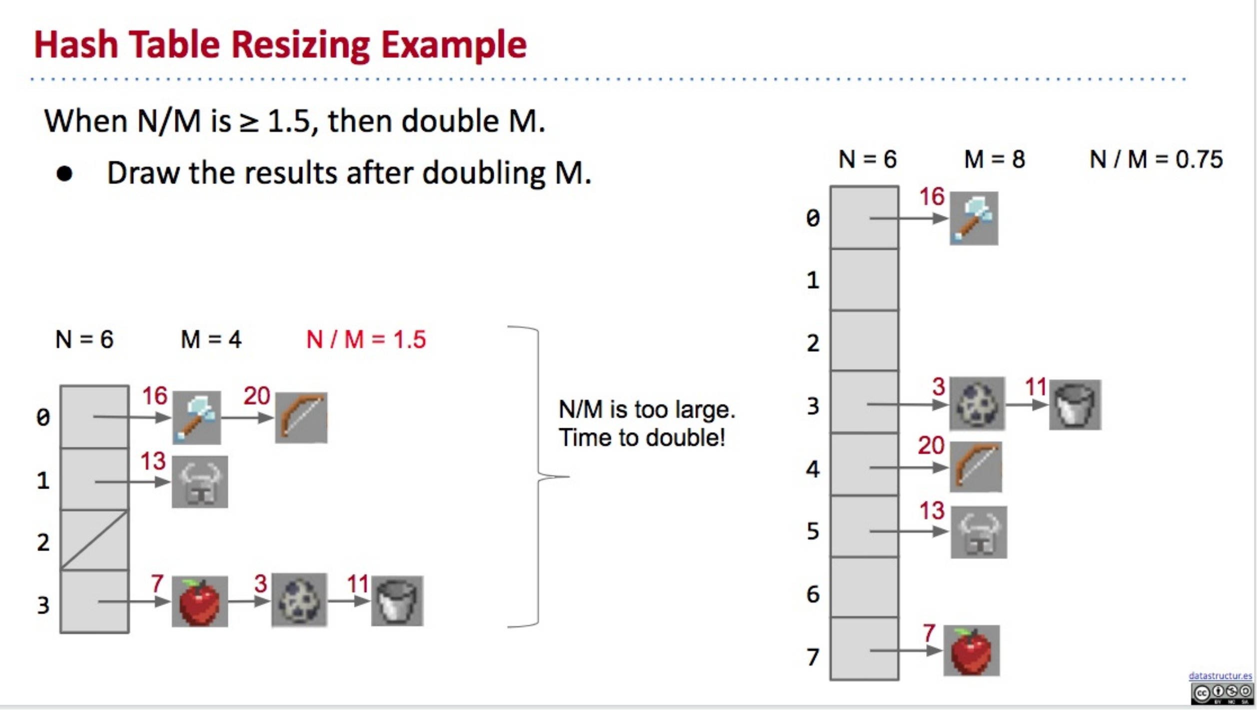
Strategy? Every so often, just double M. The way we do this is as follows:

- Create a new HashTable with 2M buckets.
- Iterate through all the items in the old HashTable, and one by one, add them into this new HashTable.

- We need to add elements one by one again because since the size of the array changes, the modulus also changes, therefore the item probably belongs in a different bucket in the new hashtable than in the old one.

We do this by setting a **load factor threshold**. As soon as the load factor becomes bigger than this threshold, we resize.

Take a look at the example below. The hashcode for the "helmet" is 13. In the first hashtable, it gets sent to bucket $13\%4 = 1$. In the second hash table, it gets sent to bucket $13\%8 = 5$. **Note that resizing the hash table also helps with shuffling the items in the hashtable**!



At this point, *assuming items are evenly distributed*, all the lists will be approximately N/M items long, resulting in $\Theta(N/M)$ runtime. Remember that N/M is only allowed to be under a constant **load factor threshold**, and so, $\Theta(N/M) = \Theta(1)$.

Note also that resizing takes $\Theta(N)$ time. Why? Because we need to add N items to the hashtable, and each add takes $\Theta(1)$ time.

A small point: when doing the resize, we don't actually need to check if the items are already present in the LinkedList or not (because we know there are no duplicates), so we can just add each item in $\Theta(1)$ time for sure by adding it to the front of the linked list. (Recall that usually we have to search the LinkedList to make sure the item isn't there... but we can skip that step when resizing.)

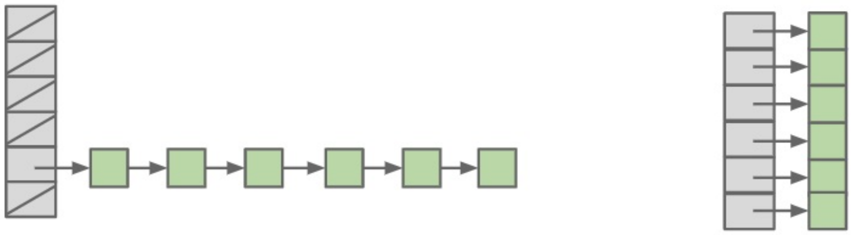
Of course, we need to revisit our assumption of assuming items are evenly distributed. If items are not evenly distributed, our runtime will be $\Theta(N)$ because there could be a single linked list of size N.

Assuming that items are evenly distributed?

Items will distribute evenly if we have good hash codes (i.e. hashcodes which give fairly random values for different items.) Doing this in general is.. well... hard.

Goal: We want hash tables that look like the table on the right.

- Want a hashCode that spreads things out nicely on real data.
  - Example #1: return 0 is a bad hashCode function.
  - Example #2: just returning the first character of a word, e.g. “cat” → 3 was also a bad hash function.
  - Example #3: Adding chars together is bad. “ab” collides with “ba”.
  - Example #4: returning string treated as a base B number can be good.
- Writing a good hashCode() method **can be tricky**.



Some general good rules of thumb:

- Use a 'base' strategy similar to the one we developed earlier.
- Use a 'base' that's a small prime.
  - Base 126 isn't actually very good, because using base 126 means that any string that ends in the same last 32 characters has the same hashcode.
  - This happens because of overflow.
  - Using prime numbers helps avoid overflow issues (i.e., collisions due to overflow).
  - Why a small prime? Because it's easier to compute.

Some examples

Lists are a lot like strings: Collection of items each with its own hashCode:

```
@Override
public int hashCode() {
    int hashCode = 1;
    for (Object o : this) {
        hashCode = hashCode * 31;
        hashCode = hashCode + o.hashCode();
    }
    return hashCode;
}
```

elevate/smear the current hash code

add new item's hash code

For example, binary tree hashCode (assuming sentinel leaves):

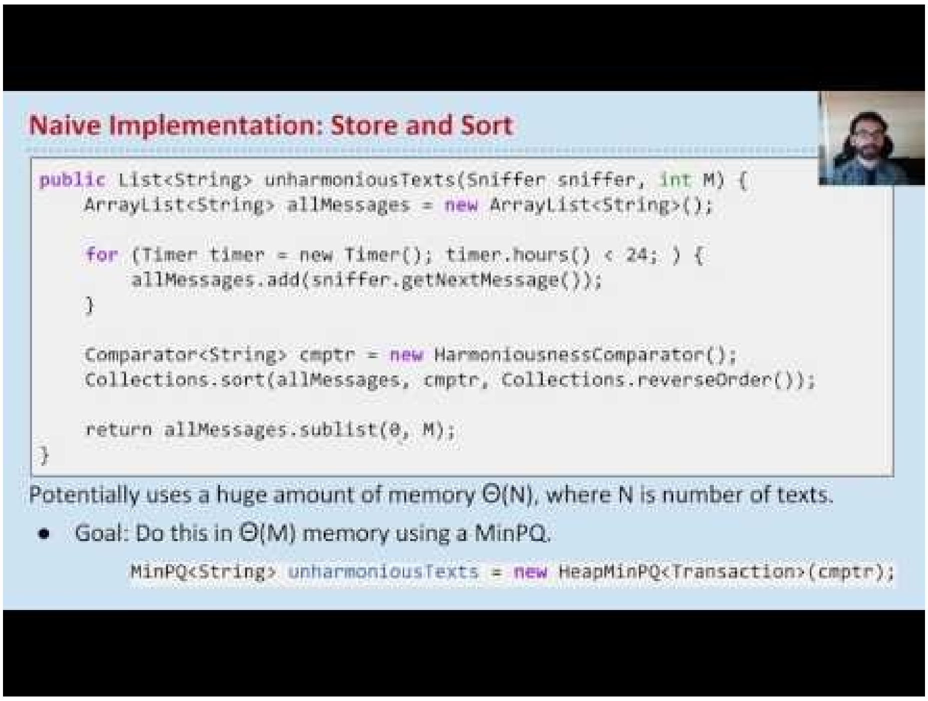
```
@Override
public int hashCode() {
    if (this.value == null) {
        return 0;
    }
    return  this.value.hashCode() +
     31 * this.left.hashCode() +
     31 * 31 * this.right.hashCode();
}
```

Next Steps

Wow, we've just gone through the creation of a data structure from scratch! Proud of you guys. To apply your knowledge finish HW3:
https://sp19.datastructur.es/materials/hw/hw3/hw3\

The Priority Queue Interface

The last **ADT** we learned about were Binary Search Trees, which allowed us efficient search only taking $\log N$ time. This was because we could eliminate half of the elements at every step of our search. What if we cared more about quickly finding the *smallest* or *largest* element instead of quickly searching?



Video link

Now we come to the Abstract Data Type of a *Priority Queue*. To understand this ADT, consider a bag of items. You can add items to this bag, you can remove items from this bag, etc. The one caveat is that you can only interact with the smallest items of this bag.

```
/** (Min) Priority Queue: Allowing tracking and removal of
  * the smallest item in a priority queue. */
public interface MinPQ<Item> {
    /** Adds the item to the priority queue. */
    public void add(Item x);
    /** Returns the smallest item in the priority queue. */
    public Item getSmallest();
    /** Removes the smallest item from the priority queue. */
    public Item removeSmallest();
    /** Returns the size of the priority queue. */
    public int size();
}
```

Using Priority

Where would we actually use this structure or need it?

- Consider the scenario where we are monitoring text messages between citizens and want to keep track of unharmonious conversations.
- Each day, you prepare a report of M messages that are the most unharmonious using a `HarmoniousnessComparator`.

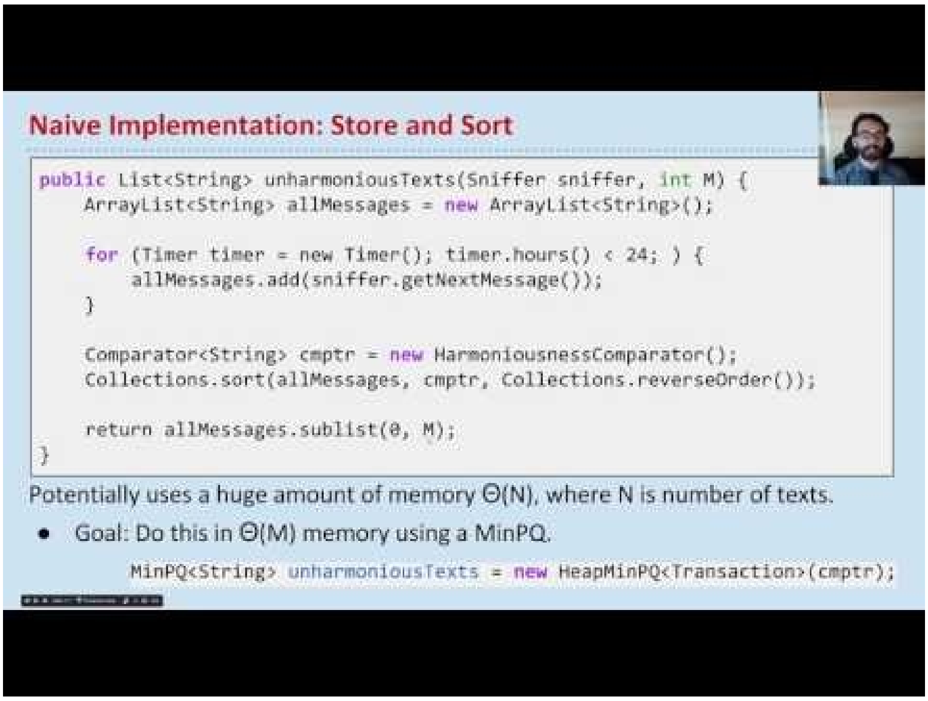
Let's take this approach: Collect all of the messages that we receive throughout the day into a single list. Sort this list and return the first M messages.

```
public List<String> unharmoniousTexts(Sniffer sniffer, int M) {
    ArrayList<String>allMessages = new ArrayList<String>();
    for (Timer timer = new Timer(); timer.hours() < 24; ) {
        allMessages.add(sniffer.getNextMessage());
    }

    Comparator<String> cmptr = new HarmoniousnessComparator();
    Collections.sort(allMessages, cmptr, Collections.reverseOrder());

    return allMessages.sublist(0, M);
```

Potential downsides? This approach will use $\Theta(N)$ space when actuality we only really need to use $\Theta(M)$ space.

Exercise 13.1.1. Complete the method listed above, `unharmoniousTexts` with the same functionality as described using only $\Theta(M)$ space.

Priority Queue Implementation



Video link

We solved the same problem using the Priority Queue ADT, making memory more efficient. We can observe that the code is slightly more complicated, but this is not always the case.

Remember that ADT are similar interfaces and the implementation is still to be defined. Let's consider possible implementations using the data structure implementations we already know, analyzing the **worst case** runtimes of our desired operations:

- Ordered Array
    - `add` : $\Theta(N)$
    - `getSmallest` : $\Theta(1)$
    - `removeSmallest` : $\Theta(N)$
- Bushy BST
    - `add` : $\Theta(\log N)$
    - `getSmallest` : $\Theta(\log N)$
    - `removeSmallest` : $\Theta(\log N)$
- HashTable
    - `add` : $\Theta(1)$
    - `getSmallest` : $\Theta(N)$
    - `removeSmallest` : $\Theta(N)$

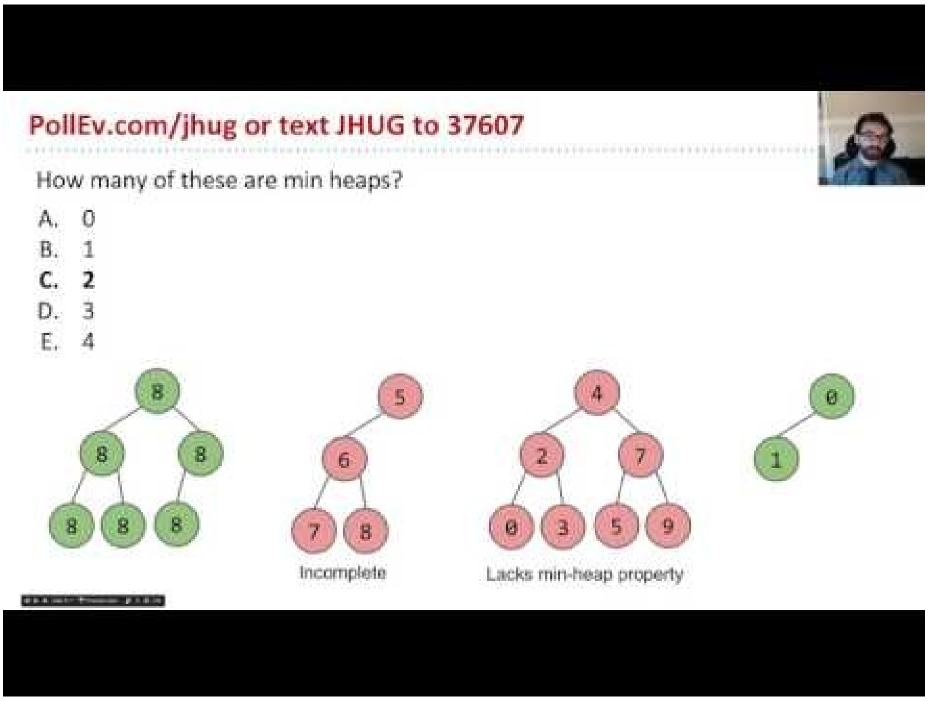
Exercise 13.1.2. Explain each of the runtimes above. What are the downfalls of each data structure? Describe a way of modifying one of these data structures to improve performance.

Can we do better than these suggested data structures?

Summary

- Priority Queue is an Abstract Data Type that optimizes for handling minimum or maximum elements.
- There can be space/memory benefits to using this specialized data structure.
- Implementations for ADTs that we currently know don't give us efficient runtimes for PQ operations.
    - A binary search tree among the other structures is the most efficient

Heap Structure

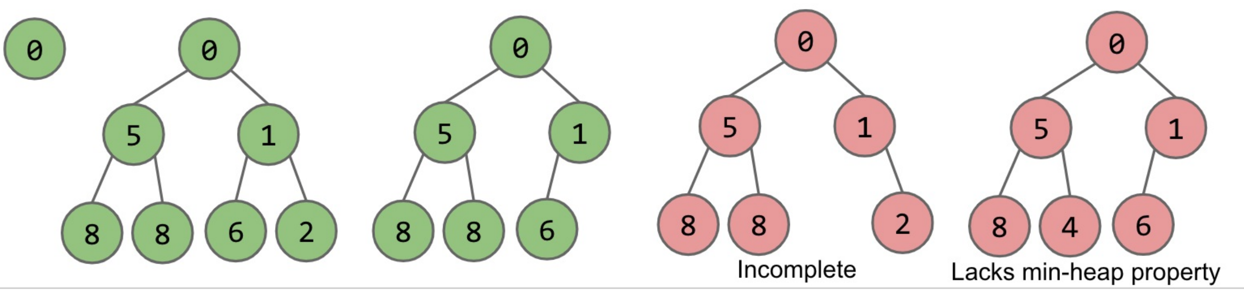


[Video link](#)

We previously saw that our known data structures with the best runtime for our PQ operations was the *binary search tree*. Modifying its structure and the constraints, we can further improve the runtime and efficiency of these operations.

We will define our binary min-heap as being **complete** and obeying **min-heap** property:

- Min-heap: Every node is less than or equal to both of its children
- Complete: Missing items only at the bottom level (if any), all nodes are as far left as possible.

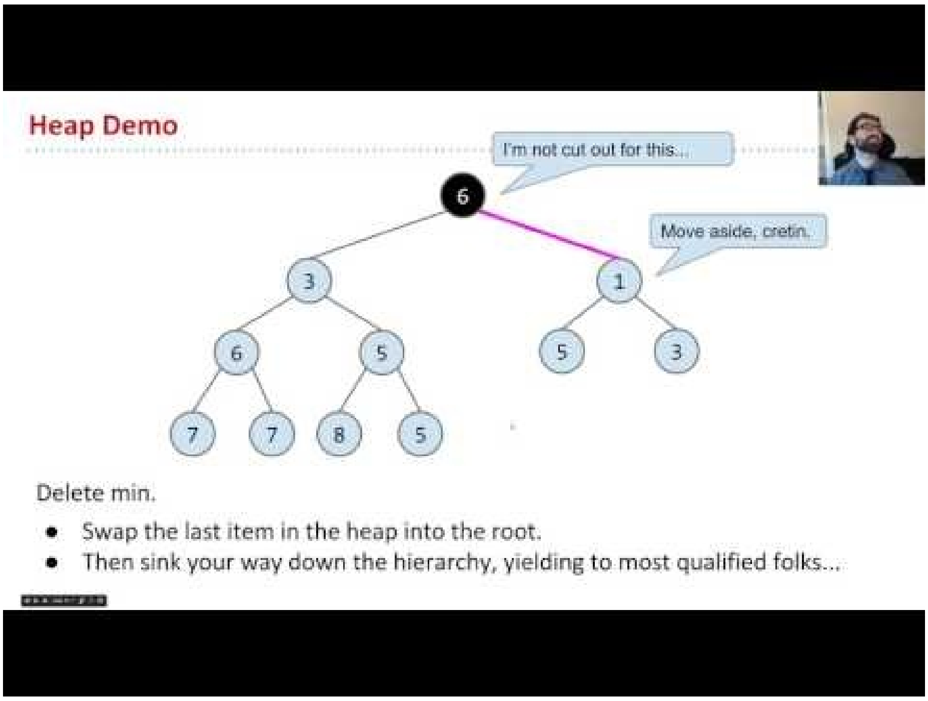


As we can see in the figures above, the green colored heaps are valid and the red ones aren't. The last two aren't because they violate at least one of the properties that we defined above.

Now let's consider how this structure lends itself to the abstract data type we described in the previous chapter. We will do this through analyzing our desired operations.

Exercise 13.2.1. Determine how each method of our Priority Queue interface will be implemented given this heap structure. Don't write actual code, just pseudocode!

Heap Operations



Video link

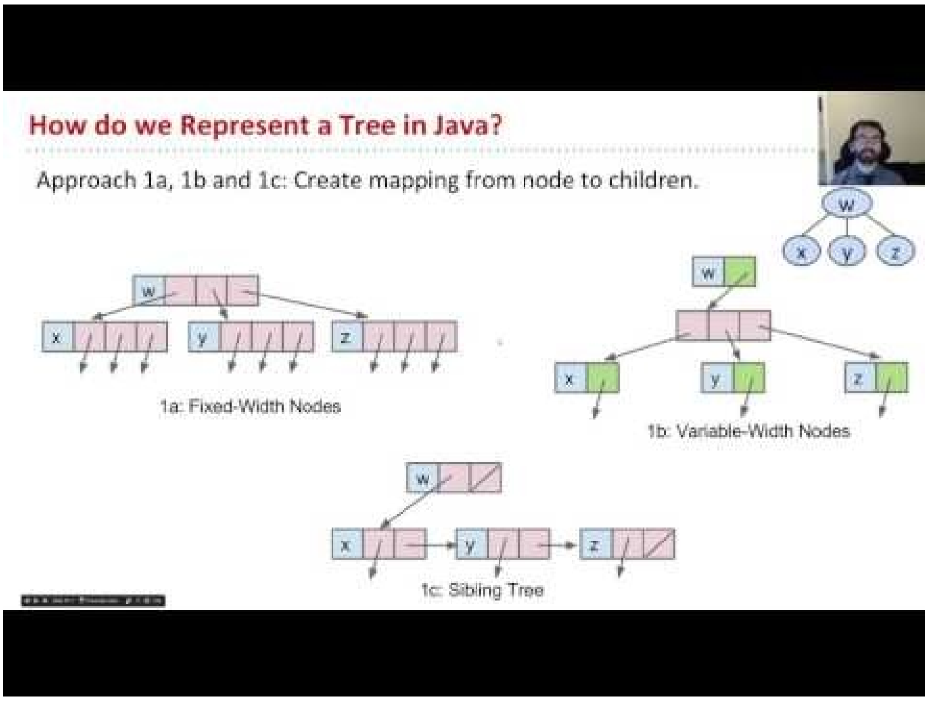
The three methods we care about for the PriorityQueue ADT are `add`, `getSmallest`, and `removeSmallest`. We will start by conceptually describing how these methods can be implemented given our given schema of a heap.

- `add`: Add to the end of heap temporarily. Swim up the hierarchy to the proper place.
    - Swimming involves swapping nodes if child < parent
- `getSmallest`: Return the root of the heap (This is guaranteed to be the minimum by our *min-heap* property
- `removeSmallest`: Swap the last item in the heap into the root. Sink down the hierarchy to the proper place.
    - Sinking involves swapping nodes if parent > child. Swap with the smallest child to preserve *min-heap* property.

Great! We have determined how we will approach the operations specified by the PriorityQueue interface in an efficient way. But how do we actually code this?

Exercise 13.2.2. Give the runtime for each of the methods specified above. Worst cases and best cases.

Tree Representation



Video link

There are many approaches we can take to representing trees.

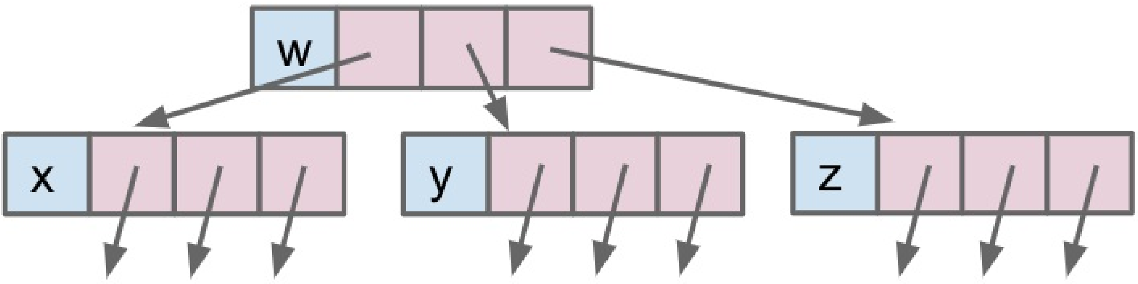
Approach 1a, 1b, and 1c

Let us consider the most intuitive and previously used representation for trees. We will create mappings between nodes and their children. There are several ways to do this which we will explore right now.

- In approach **Tree1A**, we consider creating pointers to our children and storing the value inside of the node object. These are hardwired links that give us fixed-width nodes. We can observe the code:

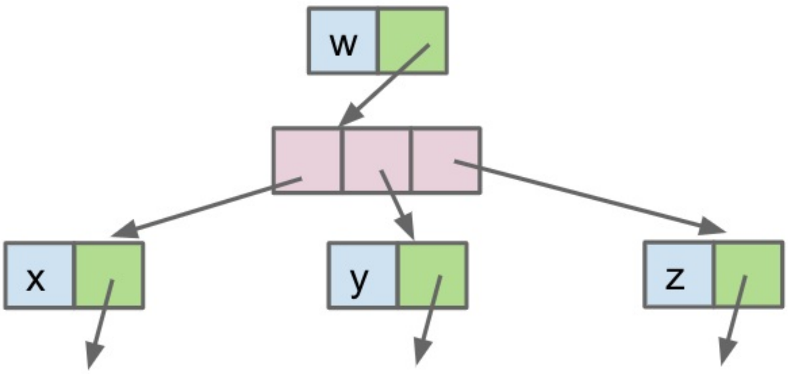
```
public class Tree1A<Key> {
  Key k;
  Tree1A left;
  Tree1A middle;
  Tree1A right;
  ...
}
```

The visualization of this type of structure is shown below.



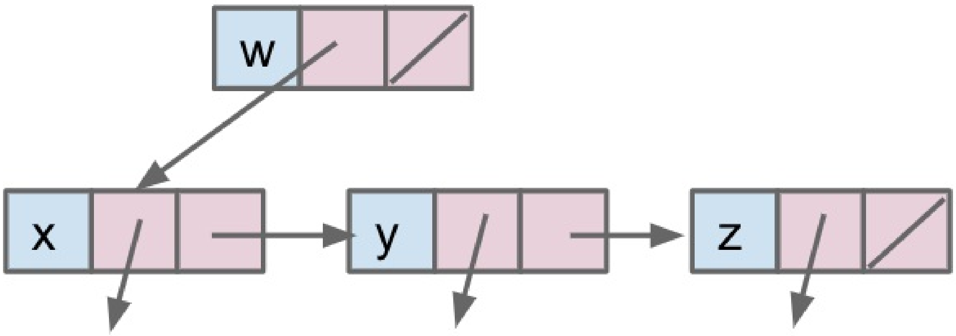
- Alternatively, in **Tree1B**, we explore the use of arrays as representing the mapping between children and nodes. This would give us variable-width nodes, but also awkward traversals and performance will be worse.

```
public class Tree1B<Key> {
  Key k;
  Tree1B[] children;
  ...
}
```



- Lastly, we can use the approach for **Tree1C**. This will be slightly different from the usual approaches that we've seen. Instead of only representing a node's children, we say that nodes can also maintain a reference to their siblings.

```
public class Tree1C<Key> {
  Key k;
  Tree1C favoredChild;
  Tree1C sibling;
  ...
}
```

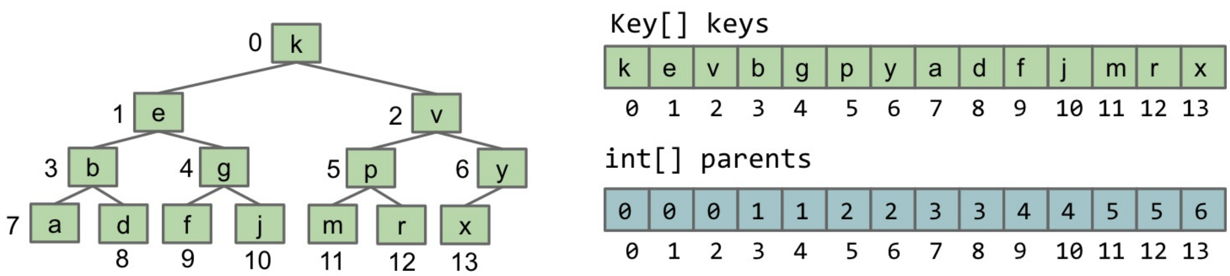


In all of these approaches, we store explicit references to who is below us. These explicit references take the form of pointers to the actual Tree objects that are our children. Let's think of more exotic approaches that don't store explicit references to children.

Approach 2

Recall the Disjoint Sets ADT. The way that we represented this Weighted Quick Union structure was through arrays. For representing a tree, we can store the *keys* array as well as a *parents* array. The keys array represent which index maps to which key, and the parents array represents which key is a child of another key.

```
public class Tree2<Key> {
  Key[] keys;
  int[] parents;
  ...
}
```



Take some time to ensure that the tree on the left corresponds to the representation in the arrays on the right.

It's time to make a very important observation! Based on the structure of the tree and the relationship between the array representations and the diagram of the tree, we can see:

1. The tree is **complete**. This is a property we have defined earlier.
2. The parents array has a sort of redundant pattern where elements are just doubled.
3. Reading the level-order of the tree, we see that it matches exactly the order of the keys in the *keys* array.

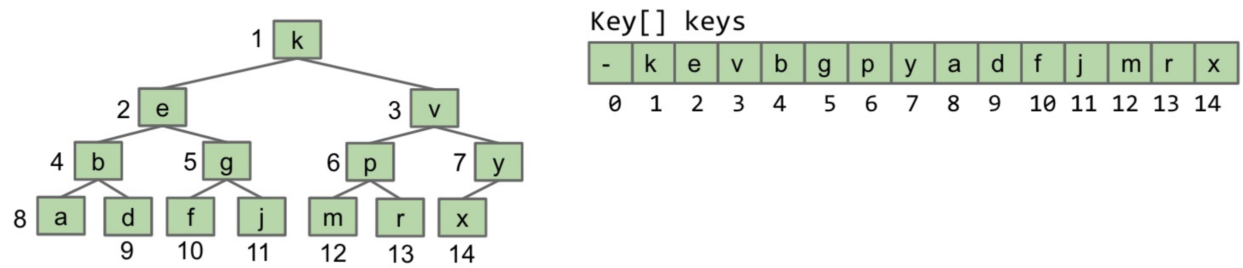
What does this all mean? We know the parents array is redundant so we can ignore it and we know that a tree can be represented by level order in an array.

Approach 3

In this approach, we assume that our tree is complete. This is to ensure that there are no "gaps" inside of our array representation. Thus, we will take this complex 2D structure of the tree and flatten it into an array.

```
public class TreeC<Key> {
  Key[] keys;
  ...
}
```

With the continued diagram from above:



This will be our similar to the book's scheme for implementing the heap, which is the underlying implementation for the Priority Queue ADT.

Swim

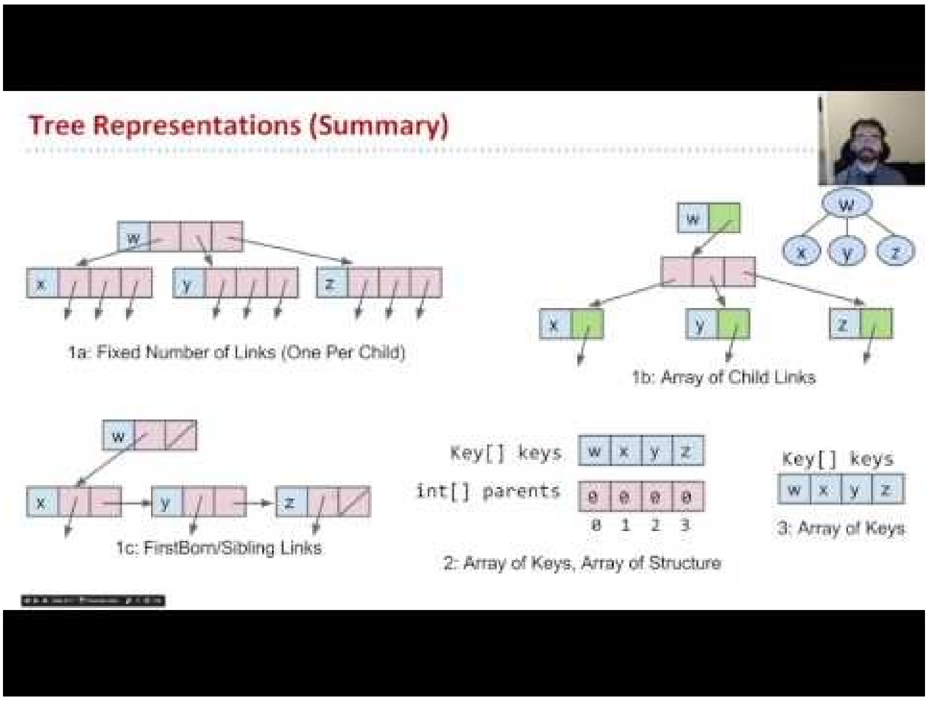
Given this implementation, we define the following code for the "swim" described in the Heap Operations section.

```
public void swim(int k) {
    if (keys[parent(k)] > keys[k]) {
       swap(k, parent(k));
       swim(parent(k));
    }
}
```

What does the parent method do? It returns the parent of the given k using the representation in Approach 3.

Exercise 13.2.3. Write the parent method. For extra challenge, try to write the methods for finding the left child and right child of a given item.

The Implementation



Video link

The actual implementation, which we will use and the book uses, is quite similar to the representation we discussed at the end of the previous chapter. The one difference is that we will leave one empty spot at the beginning of the array to simplify computation.

- `leftChild(k)` $= k * 2$
- `rightChild(k)` $= k * 2 + 1$
- `parent(k)` $= k/2$

Comparing to alternative implementations

Methods	Ordered Array	Bushy BST	Hash Table	Heap
`add`	$\Theta(N)$	$\Theta(\log N)$	$\Theta(1)$	$\Theta(\log N)$
`getSmallest`	$\Theta(1)$	$\Theta(\log N)$	$\Theta(N)$	$\Theta(1)$
`removeSmallest`	$\Theta(N)$	$\Theta(\log N)$	$\Theta(N)$	$\Theta(\log N)$

Awesome! We can see that we have improved our runtime and we have also solved the problem of duplicate elements. Couple notes:

- Heap operations are **amortized** analysis, since the array will have to resize (not a big

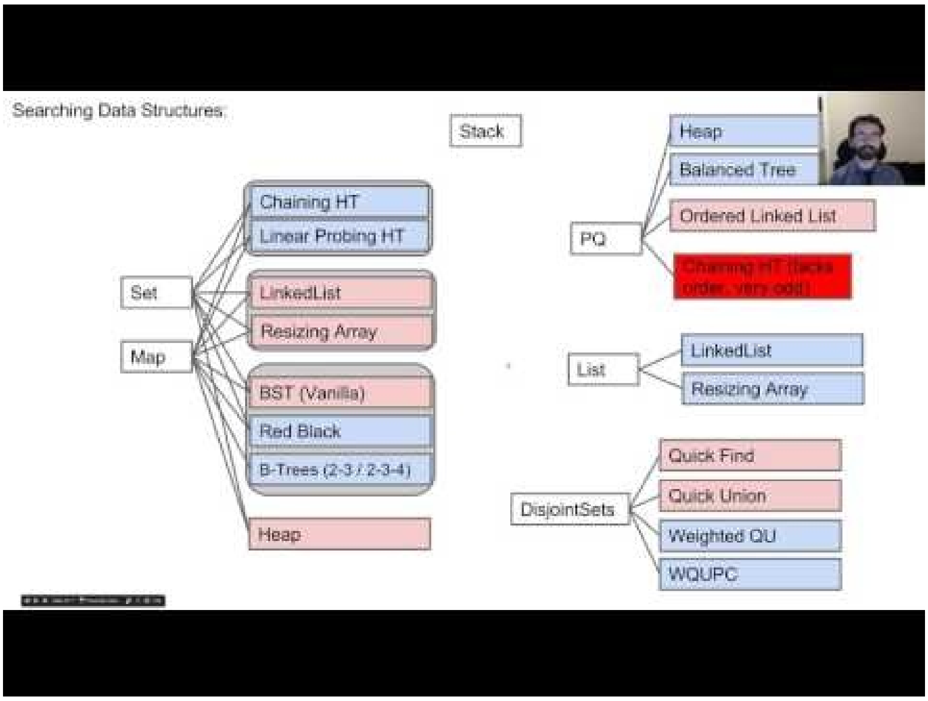
deal)
- BST's can have constant time `getSmallest` if pointer is stored to smallest element
- Array-based heaps take around 1/3rd the memory it takes to represent a heap using approach 1A (direct pointers to children)

Some lingering implementation questions.

1. How will the PQ know how to order the items? Say we had a PQ of dogs, would it order by weight or breed?
2. Is there a way to allow for a flexibility of orderings?
3. What could we do to make a MinPQ into a MaxPQ?

Exercise 13.3.1. Answer the questions above.

Data Structures Summary



Video link

The Search Problem

The problem we are presented: Given a stream of data, retrieve information of interest.

What are some examples of this?

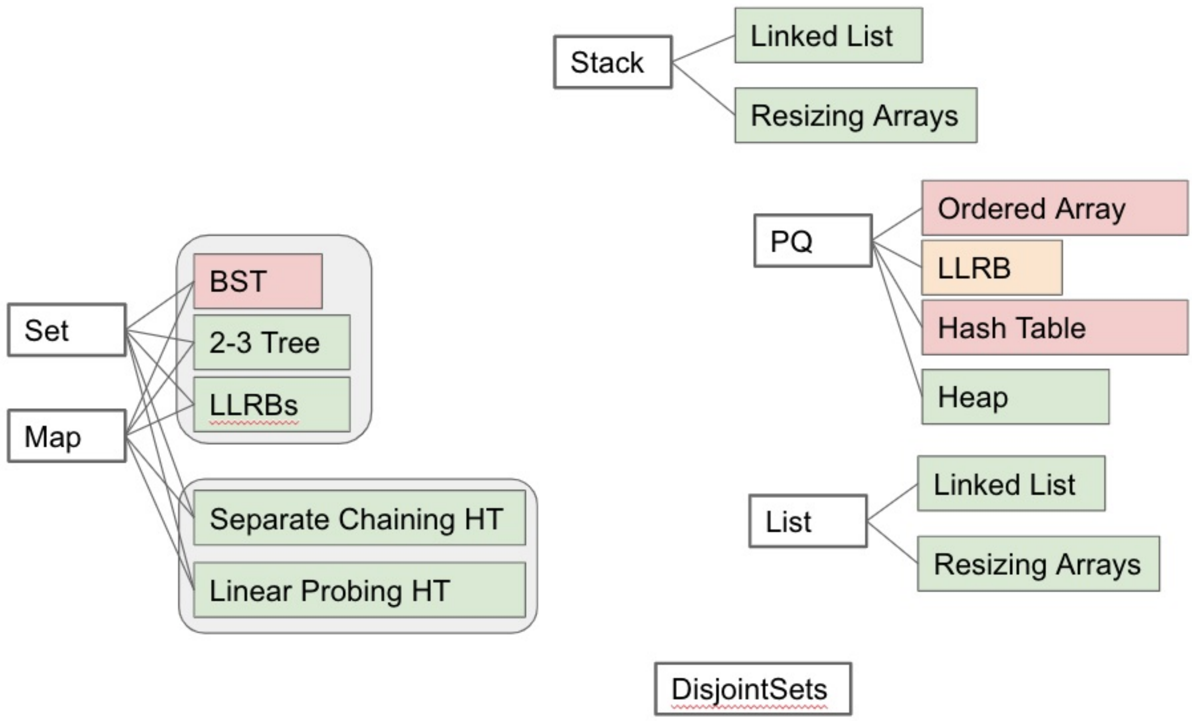
- Website users post to personal page. Serve content only to friends.
- Given logs for thousands of weather stations, display weather map for specified date and time.
- Dog owners request the best pet store, choosing to define their best store either in price, quality, or atmosphere.

All of the data structures we have discussed so far have been to *solve* the search problem. How you might ask? Each of the data structures we've learned are used for storing information in schemes that make searching efficient in specific scenarios.

Search Data Structures

Name	Store Operation(s)	Primary Retrieval Operation	Retrieve By
List	`add(key)` , `insert(key, index)`	`get(index)`	index
Map	`put(key, value)`	`get(key)`	key identity
Set	`add(key)`	`containsKey(key)`	key identity
PQ	`add(key)`	`getSmallest()`	key order (aka key size)
Disjoint Sets	`connect(int1, int2)`	`isConnected(int1, int2)`	two integer values

Remember that these are **Abstract** Data Types. This means that we define the behavior, not the implementation. We've defined many of the possible implementations in previous chapters. Let's think about how these implementations and ADTs interact:



What does this diagram tell us? The first thing you'll notice is that many implementations that we have devised in earlier chapters can be used in implementing many different ADTs. You'll also notice the red colored implementations (meaning poor performance), telling us that not all implementations are optimal for the behavior we are trying to achieve.

Exercise 14.1.1. Consider how each of these implementations can be modified in order to accommodate to the behavior attempting to be defined. For example, how would we use a Hash Table to function as a Priority Queue?

Abstraction

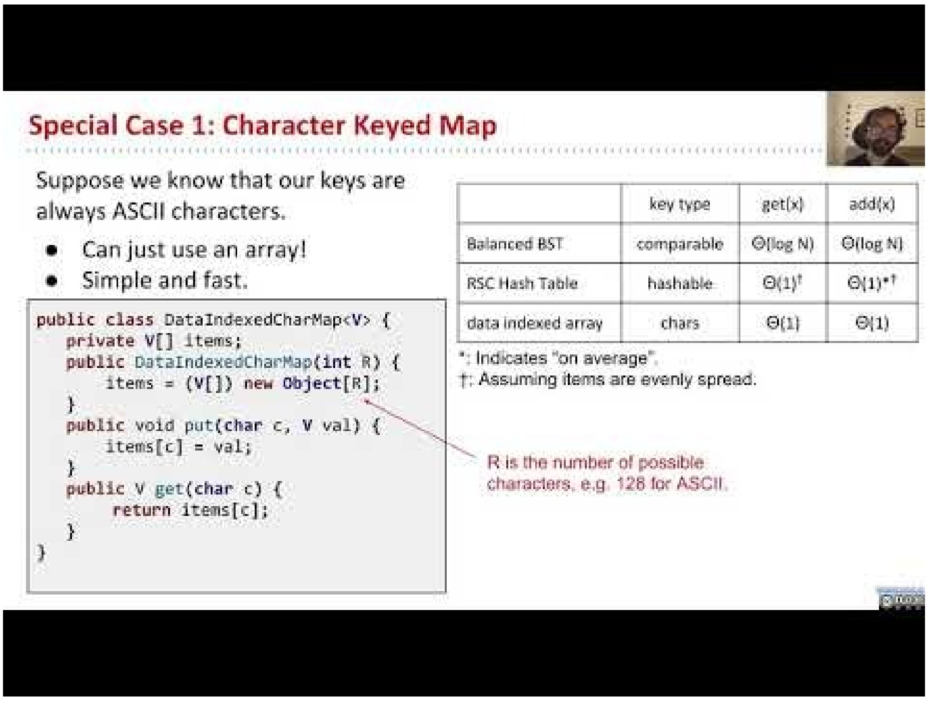
Abstraction often happens in layers. Abstract Data Types can often contain two abstract ideas boiling down to one implementation. Let's consider some examples:

- If we remembered the Priority Queue ADT, we were attempting to find an implementation that would be efficient for PQ operations. We decided that our Priority Queue would be implemented using a Heap Ordered Tree, but as we saw we had several approaches (1A, 1B, 1C, 2, 3) of representing a tree for heaps.
- A similar idea is an External Chaining Hash Table. This data structure is implemented using an array of buckets, but these buckets can be done using either an ArrayList, Resizing Array, Linked List, or BST.

These two examples tell us that we can often think of an ADT by the use of another ADT. And that Abstract Data Types have layers of abstraction, each defining a behavior that is more specific than the idea that came before it.

Tries

We are now going to learn about a new data structure called *Tries*. These will serve as a new implementation (from what we have previously learned) of a *Set* and a *Map* that has some special functionality for certain types of data and information.



Video link

Reflection

Since we are thinking about the ADTs *Set* and *Map*, let us review the implementations we've learned so far. In the past we've mainly learned how to implement these ADTs using Binary Search Trees or Hash Tables. Let's recall the runtimes for both of these:

- Balanced Search Tree:
  - `contains(x)` : $\Theta(\log N)$
  - `add(x)` : $\Theta(\log N)$
- Resizing Separate Chaining Hash Table:
  - `contains(x)` : $\Theta(1)$ (assuming even spread)
  - `add(x)` : $\Theta(1)$ (assuming even spread and amortized)

These runtimes are fantastic! We can see that the implementations for operations associated with Sets and Maps are extremely fast.

Question: Can we do even better than this? That could depend on the nature of our problem. What if we knew a special characteristic of the keys?

Answer: Yes, for example if we always know our keys are ASCII characters then instead of using a general-purpose HashMap, simply use an array where each character is a different index in our array:

```
public class DataIndexedCharMap<V> {
    private V[] items;
    public DataIndexedCharMap(int R) {
        items = (V[]) new Object[R];
    }
    public void put(char c, V val) {
        items[c] = val;
    }
    public V get(char c) {
        return items[c];
    }
}
```

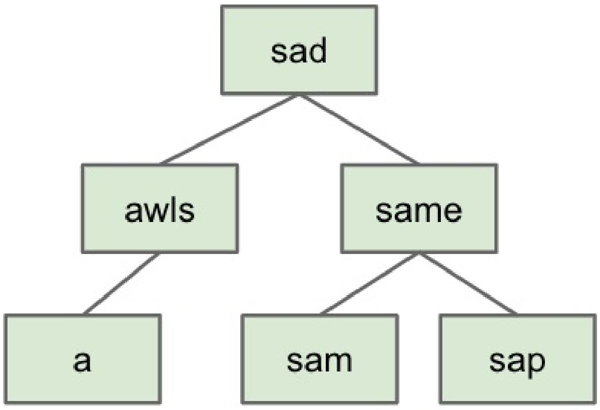
Here we have created an implementation for a map that takes in character keys. The value R represents the number of possible characters (e.g. 128 for ASCII). Nice! By knowing the range of our keys as well as what types of values they will be we have a simple and efficient data structure. Now we have created something for characters, what about strings?

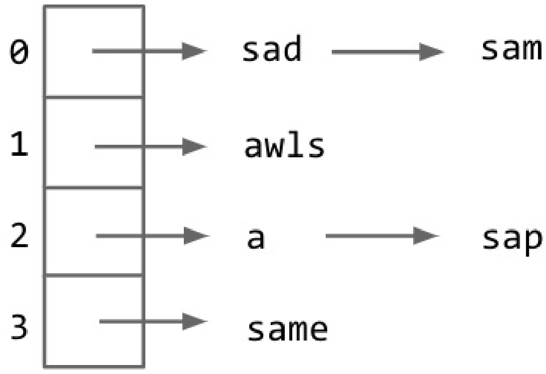
Exercise 15.1.1. Come up with an implementation for a Map that contains only integer keys from 0 to 100. Ensure that this performs better than a regular HashMap.

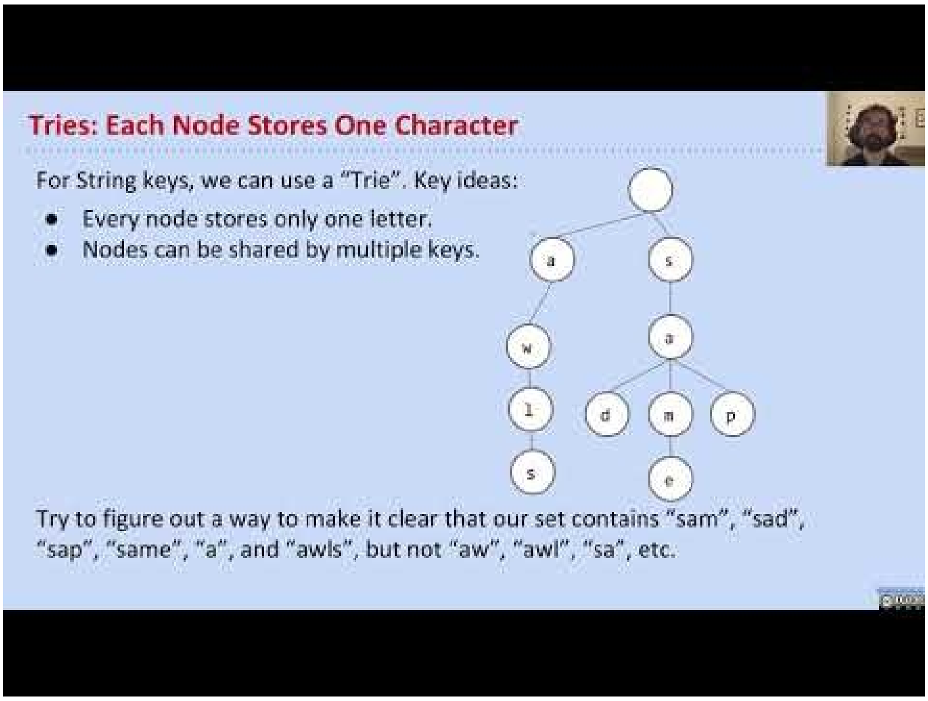
Inventing the Trie

Tries are a very useful data structure used in cases where keys can be broken into "characters" and share prefixes with other keys (e.g. strings).

Suppose we had a set containing "sam", "sad", "sap", "same", "a", and "awls. With the existing Set implementations, we have the following visual structures. How might we improve upon this using other possible data structures we know? How might we take advantage of the structure strings?







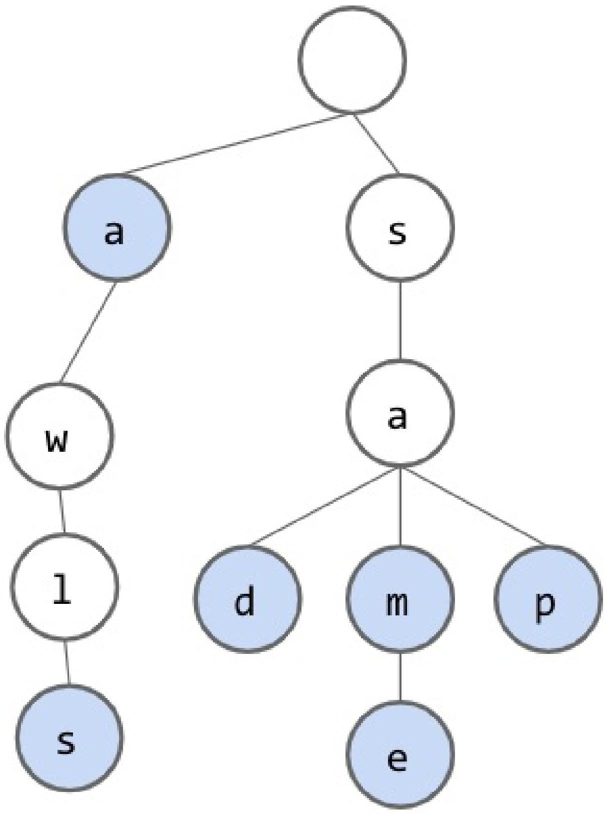
Video link

Here are some key ideas that we will use:

- Every node stores only one letter
- Nodes can be shared by multiple keys

Therefore, we can insert "sam", "sad", "sap", "same", "a", and "awls" into a tree structure that contains single character nodes. An important observation to make is that most of these words share the same *prefixes*, therefore we can utilize these similarly structured strings for our structure. In other words we don't store the same prefixes (e.g. "sa-") multiple times.

Take a look at the graphic below to see how a trie would look like:



Tries work by storing each 'character' of our keys as a node. Keys that share common prefixes share the same nodes. To check if the trie contains a key, walk down the tree from the root along the correct nodes.

Since we are going to share nodes, we must figure out some way to represent which strings belong in our set and which don't. We will solve this problem through marking the color of the last character of each string to be blue. Observe our final strategy below.

Suppose we have inserted strings into our set and we end up with the trie above, we must figure out how searching will work in our current scheme. To search, we will traverse our trie and compare to each character of the string as we go down. Thus, there are only two cases when we wouldn't be able to find a string; either the final node is white or we fall off the tree.

- `contains("sam")` : true, blue node
- `contains("sa")` : false, white node
- `contains("a")` : true, blue node
- `contains("saq")` : false, fell off tree

Exercise 15.1.2. Add the strings "ants", "zebra", "potato" and "sadness" to the trie above. Draw out your resulting trie structure.

Exercise 15.1.3. Think about the difference between a trie being used as a map versus a trie being used as a set. What about (if any) the implementation would be different?

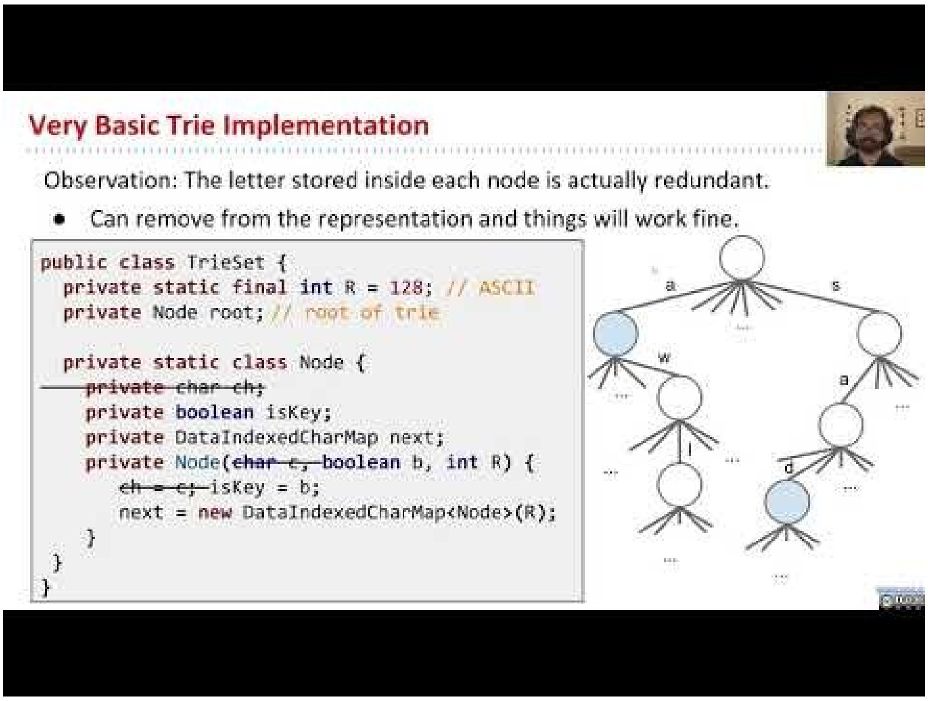
See an animated demo of creation of a trie map here.

Summary

A key takeaway is that we can often improve a general-purpose data structure when we add specificity to our problem, often by adding additional constraints. For example, we improved our implementation of HashMap when we restricted the keys to only be ASCII character, creating extremely efficient data structure.

- There is a distinction between ADTs and specific implementations.As an example, Disjoint Sets is an ADT: any Disjoint Sets has the methods `connect(x, y)` and `isConnected(x, y)`. There are four different ways to *implement* those methods: Quick Find, Quick Union, Weighted QU, and WQUPC.
- The Trie is a specific implementation for Sets and Maps that is specialized for strings.
  - We give each node a single character and each node can be a part of several keys inside of the trie.
  - Searching will only fail if we hit an unmarked node or we fall off the tree
  - Short for Re**trie**val tree, almost everyone pronounces it as "try" but Edward Fredkin suggested it be pronounced as "tree"

Implementation



[Video link]

Let's actually try building a Trie. We'll take a first approach with the idea that each node stores a letter, its children, and a color. Since we know each node key is a character, we can use our `DataIndexedCharMap` class we defined earlier to map to all of a nodes' children. Remember that each node can have at most the number of possible characters as its number of children.

```
public class TrieSet {
   private static final int R = 128; // ASCII
   private Node root;    // root of trie

   private static class Node {
      private char ch;
      private boolean isKey;
      private DataIndexedCharMap next;

      private Node(char c, boolean blue, int R) {
         ch = c;
         isKey = blue;
         next = new DataIndexedCharMap<Node>(R);
      }
   }
}
```

Zooming in on a single node with one child we can observe that its `next` variable, the DataIndexedCharMap object, will have mostly null links if nodes in our tree have relatively few children. We will have 128 links with 127 equal to null and 1 being used. This means that we are wasting a lot of excess space! We will explore alternative representations further on.

But we can make an important observation: each link corresponds to a character if and only if that character **exists**. Therefore, we can remove the Node's character variable and instead base the value of the character from its position in the parent `DataIndexedCharMap`.

```
public class TrieSet {
   private static final int R = 128; // ASCII
   private Node root;    // root of trie

   private static class Node {
      private boolean isKey;
      private DataIndexedCharMap next;

      private Node(boolean blue, int R) {
         isKey = blue;
         next = new DataIndexedCharMap<Node>(R);
      }
   }
}
```

Exercise 15.2.1. Come up with a solution to the excess use of space. *Hint*: Try using some of the implementations we have discussed before.

Performance

Given a Trie with N keys the runtime for our Map/Set operations are as follows:

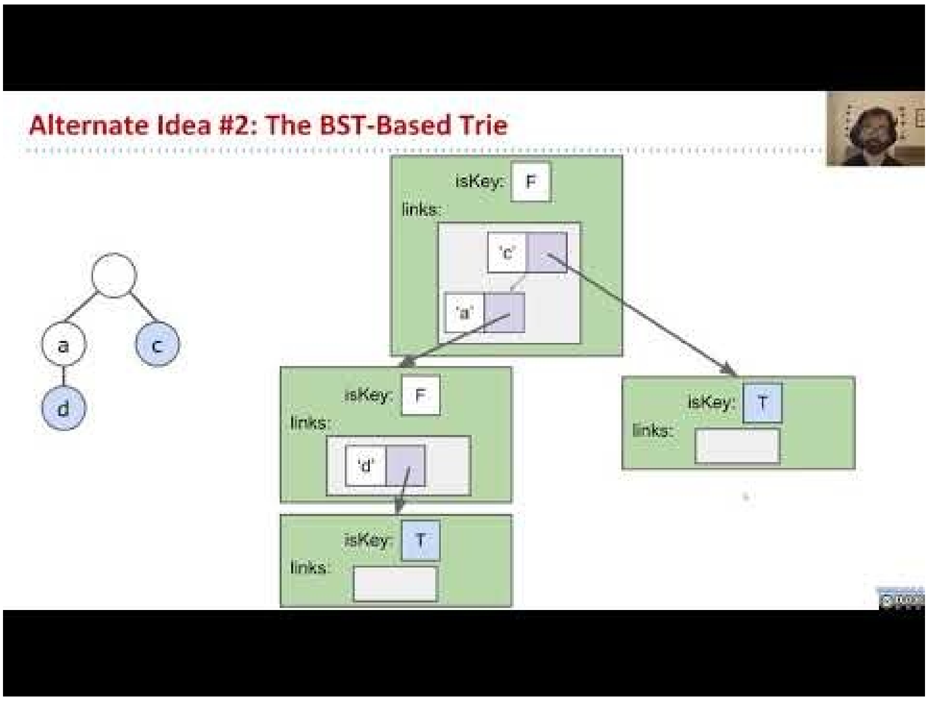
- `add` : $\Theta(1)$
- `contains` : $\Theta(1)$

Why is this the case? It doesn't matter how many items we have in our trie, the runtime will always be independent of the number of keys. This is because we only traverse the length of one key in the worst case ever, which is never related to the number of keys in the trie. Therefore, let's look at the runtime through a measurement that can be measured; in terms of L, the length of the key:

- `add` : $\Theta(L)$
- `contains` : $O(L)$

We have achieved constant runtime without having to worry about amortized resizing times or an even spreading of keys, but as we mentioned above our current design is extremely wasteful since each node contains an array for every single character even if that character doesn't exist.

Child Tracking



Video link

The problem we were running into was waste of space from our implementation of a DataIndexedCharMap object to track each node's children. The problem with this approach was that we would have initialized many **null** spots that don't contain any children.

- *Alternate Idea #1*: Hash-Table based Trie. This won't create an array of 128 spots, but instead initialize the default value and resize the array only when necessary with the load factor.
- *Alternate Idea #2*: BST based Trie. Again this will only create children pointers when necessary, and we will store the children in the BST. Obviously, we still have the worry of the runtime for searching in this BST, but this is not a bad approach.

When we implement a Trie, we have to pick a map to our children. A Map is an ADT, so we must also choose the underlying implementation for the map. What does this reiterate to us? There is an **abstraction** barrier between the implementations and the ADT that we are trying to create. This abstraction barrier allows us to take advantage of what each implementation has to offer when we try to meet the ADT behavior. Let's consider each advantage:

- DataIndexedCharMap
  - Space: 128 links per node
  - Runtime: $\Theta(1)$
- BST
  - Space: C links per node, where C is the number of children
  - Runtime: $O(\log R)$, where R is the size of the alphabet
- Hash Table
  - Space: C links per node, where C is the number of children
  - Runtime: $O(R)$, where R is the size of the alphabet

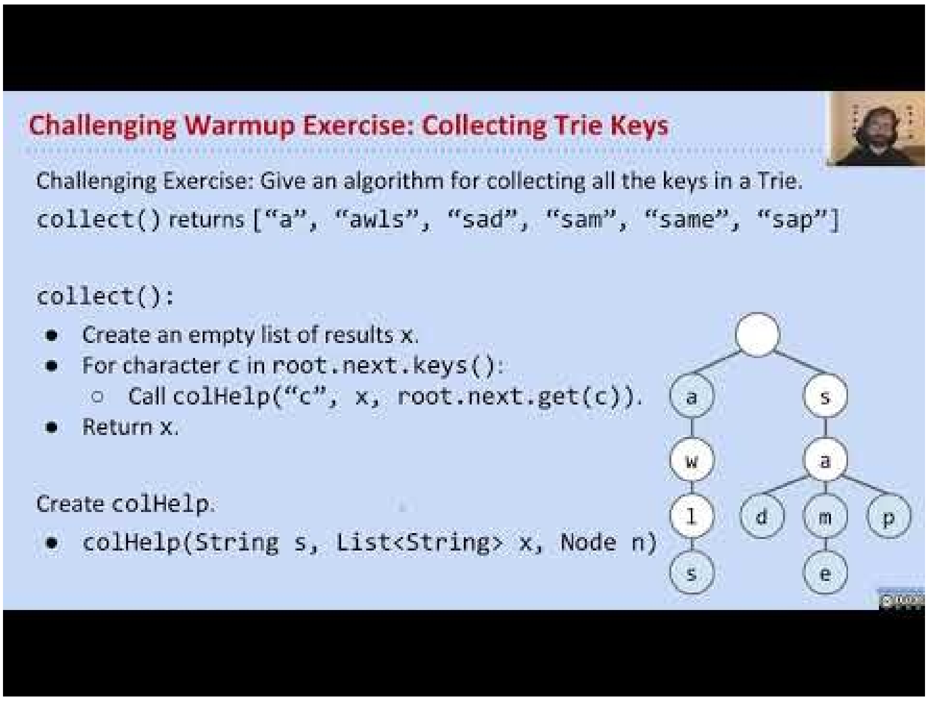
Note: Cost per link is higher in BST and Hash Tables; R is a fixed number (this means we can think of the runtimes as constant)

We can takeaway a couple of things. There is a slight memory and efficiency trade off (with BST/Hash Tables vs. DataIndexedCharMap). The runtimes for Trie operations are still constant without any caveats. Tries will especially thrive with some special operations.

Trie String Operations

Recall all of the comparisons that we've made between Tries and other data structures. We can see that Tries offer us constant time lookup and insertion, but do they actually perform better than BSTs or Hash Tables? Possibly not. For every string we have to traverse through every character, whereas in BSTs we have access to the entire string immediately. So what are Tries good for then?

Prefix Matching



[Video link]

The main appeal of tries is the ability to efficiently support specific string operations like *prefix matching*. You can imagine why tries make this extremely efficient! Say we were trying to find the longestPrefixOf. Just take the word you're looking for, compare each character with characters in your trie until you can go no longer. Similarly, if we wanted keyWithPrefix, we can traverse to the end of the prefix and return all remaining keys in the trie.

Let's attempt to define a method `collect` which returns all of the keys in a Trie. The pseudocode will be as follows:

```
collect():
    Create an empty list of results x
    For character c in root.next.keys():
        Call colHelp(c, x, root.next.get(c))
    Return x

colHelp(String s, List<String> x, Node n):
    if n.isKey:
        x.add(s)
    For character c in n.next.keys():
        Call colHelp(s + c, x, n.next.get(c))
```

We first initialize our values inside of the parent function, and then create a recursive helper function to hold more parameters throughout the recursive calls. We only add the current string if it is a key, otherwise we concatenate the character to the string/path we are currently traversing and call the helper on the next child.

Now we can try writing the method `keysWithPrefix` which returns all keys that contain the prefix passed in as an argument. We will borrow heavily from the collect method above.

```
keysWithPrefix(String s):
    Find the end of the prefix, alpha
    Create an empty list x
    For character in alpha.next.keys():
        Call colHelp("sa" + c, x, alpha.next.get(c))
    Return x
```

Exercise 15.3.1. Write pseudocode for `longestPrefixOf`. This will be an exercise you can do in lab. Make sure to watch the video on how to implement the other methods.

Autocomplete

When you type into any search browser, for example Google, there are always suggestions of what you are about to type. This is extremely helpful and convenient. Say we were searching "How are you doing", if we just type in "how are" into google, we will see that it suggests this exact query.

One way to achieve this is using a Trie! We will build a map from strings to values.

- Values will represent how important Google thinks that string is (Probably frequency)
- Store billions of strings efficiently since they share nodes, less wasteful duplicates
- When a user types a query, we can call the method `keysWithPrefix(x)` and return the 10 strings with the highest value

One major flaw with this system is if the user types in short length strings. You can imagine that the number of keys with the prefix of the input is in the millions when in reality we only want 10. A possible solution to this issue is to store the best value of a substring in each node. We can then consider children in the order of the best value.

Another optimization is to merge nodes that are redundant. This would give us a "radix trie", which holds characters as well as strings in each node. We won't discuss this in depth.

Exercise 15.3.2. Consider what adding the best value of a substring to a node will add. How can you use a Priority Queue to create an algorithm?

Summary

Knowing the types of data that you are storing can give you great power in creating efficient data structures. Specifically for implementing Maps and Sets, if we know that all keys will be Strings, we can use a Trie:

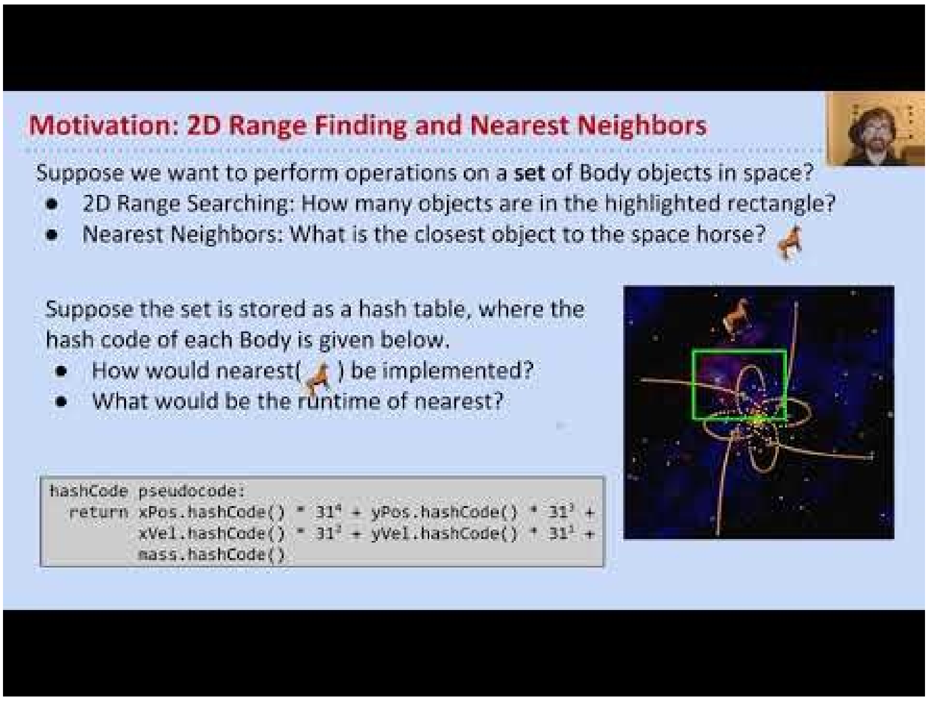
- Tries theoretically have better performances for searching and insertion than hash tables or balanced search trees
- There are more implementations for how to store the children of every node of the trie, specifically three. These three are all fine, but hash table is the most natural
  - *DataIndexedCharMap* (Con: excessive use of space, Pro: speed efficient)
  - *Bushy BST* (Con: slower child search, Pro: space efficient)
  - *Hash Table* (Con: higher cost per link, Pro: space efficient)
- Tries may not actually be faster in practice, but they support special string operations that other implementations don't
  - `longestPrefixOf` and `keysWithPrefix` are easily implemented since the trie is stored character by character
  - `keysWithPrefix` allows for algorithms like autocomplete to exist, which can be optimized through use of a priority queue.=

	key type	`get(x)`	`add(x)`
Balanced BST	comparable	$\Theta(\log N)$	$\Theta(\log N)$
RSC Hash Table	hashable	$\Theta(1)^{\dagger}$	$\Theta(1)^{*\dagger}$
Data Indexed Array	chars	$\Theta(1)$	$\Theta(1)$
Tries (BST, HT, DICM)	Strings	$\Theta(1)$	$\Theta(1)$

*: Indicates "on average"; †: Indicates items are evenly spread.

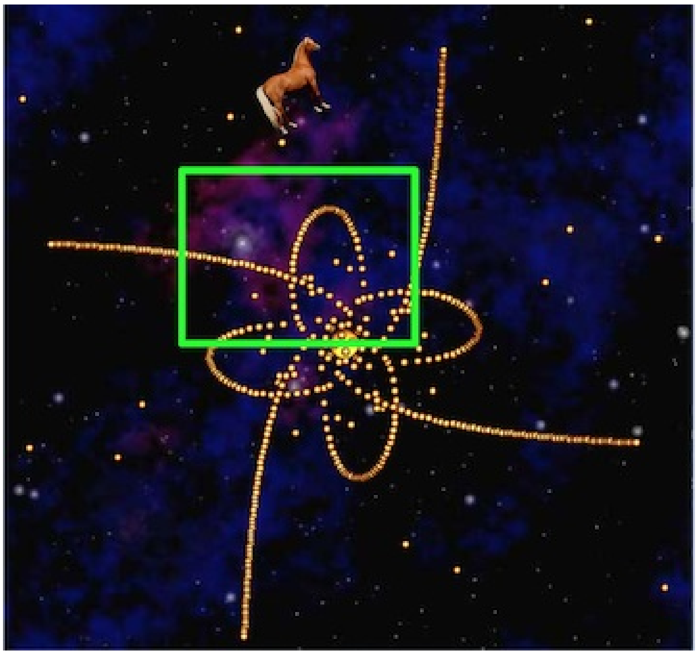
Uniform Partitioning

Motivation



Video link

Suppose we want to perform operations on a set of Body objects in space. For example, perhaps we wanted to ask questions about the Sun bodies (shown as yellow dots below) in our two-dimension image space.



First Question: 2D Range Finding

One question we might ask is: **How many objects are in a region**, such as in the highlighted green rectangle above?

Second Question: Nearest Neighbors

Another question we might ask is: **What is the closest object to another object**, such as which sun is closest to our space horse? (The desired answer as found by visual inspection is the sun closest to its back hoof.)

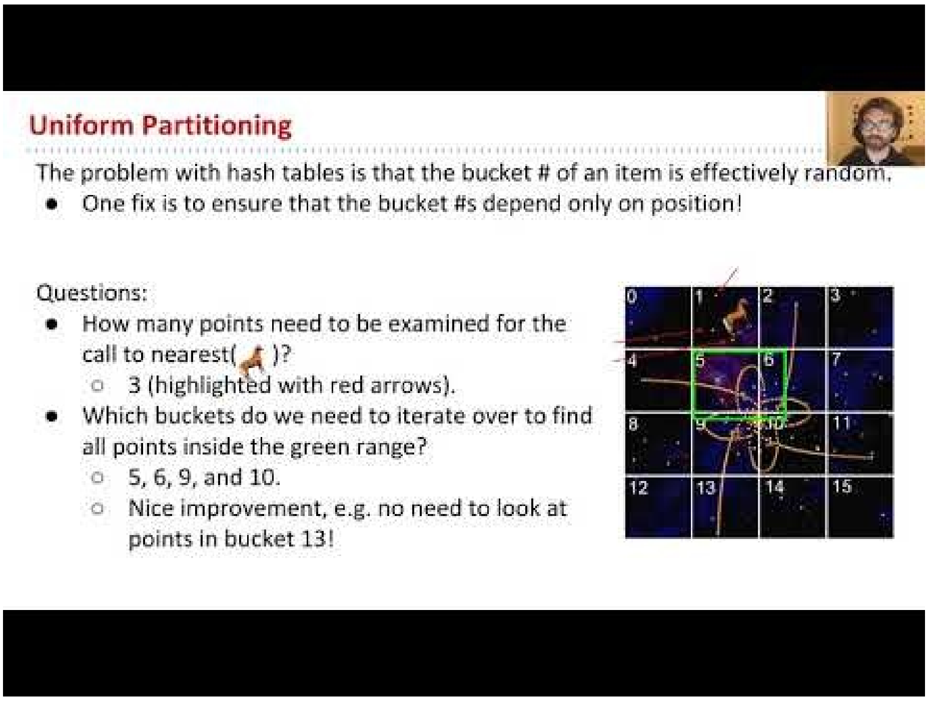
Initial Attempt: HashTable

Question: If our set of suns were stored in a HashTable, what is the runtime for finding the answer to our Nearest Neighbors question?

Solution: The bucket that each object resides in is effectively random, and so we would have to iterate over all N items to check if each sun could possibly be the closest to the horse. $\Theta(N)$.

Let's try to improve so that we don't have to look at every single sun in our set to find our answer.

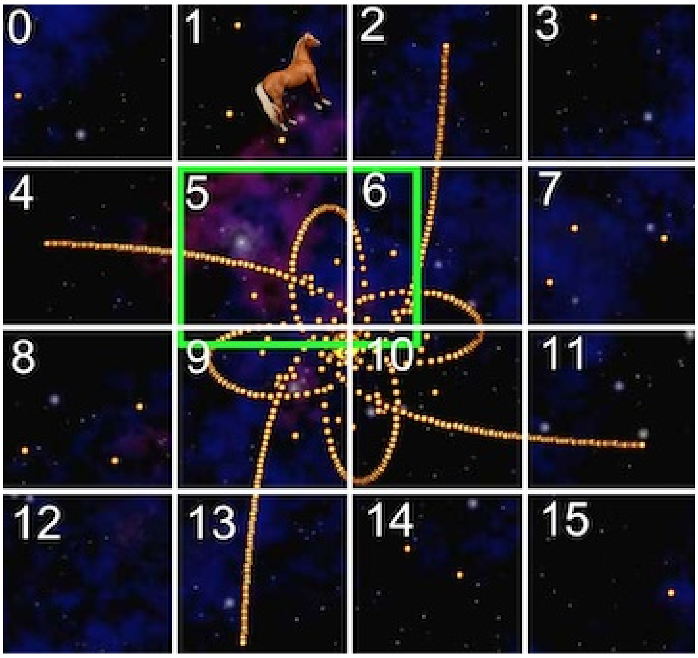
Second Attempt: Uniform Partitioning



Video link

The problem with hash tables is that the bucket number of an item is effectively random. Hash tables are, by definitely, unordered collections. One fix is to ensure that the bucket numbers depend only on position!

If we uniformly partition our image space by throwing a 4x4 grid over it, we get nice organized buckets that look something like this (this is also sometimes called "spatial hashing"):



This can be implemented by not using the object's `hashCode()` function, and instead having each object provide a `getX()` and `getY()` function so that it can compute its own bucket number.

Now, we know which grid cells our searches can be confined to, and we only have to look at suns in those particular cells rather than looking at all the suns in our entire image space as we had to before.

How many objects are in a region?: We jsut need to look in buckets 5, 6, 9, and 10.

Which sun is closest to the horse?: First, we start in the cell that the horse resides in: 1. Then, we can move outwards to 0, 4, 5, 6, and 2. etc.

Question: Using uniform partitioning, what is the runtime for finding the answer to our Nearest Neighbors question, assuming the suns are evenly spread out?

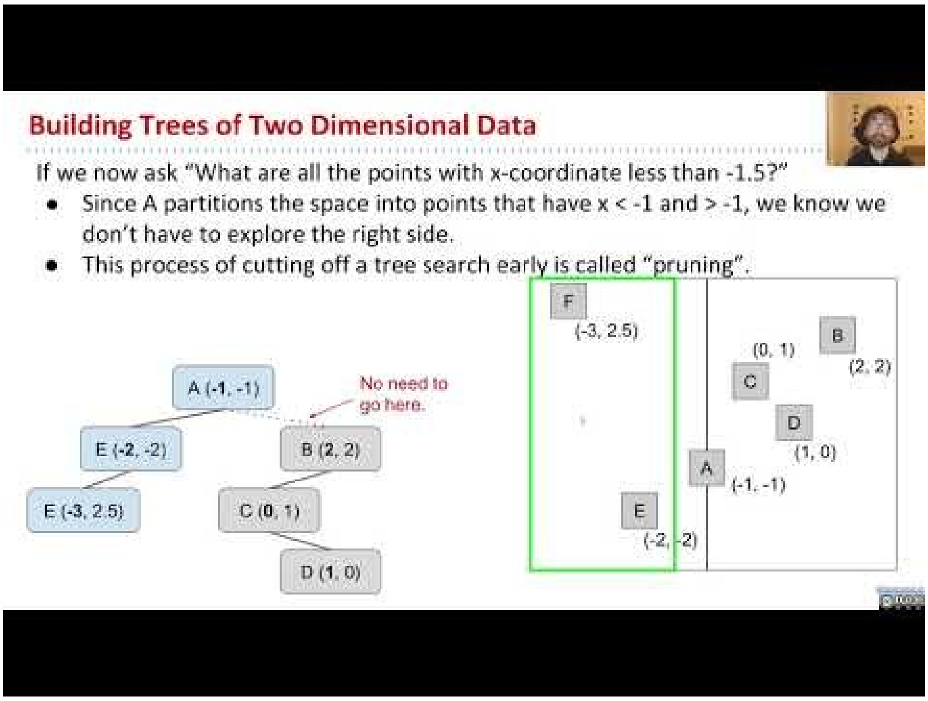
Solution: On average, the runtime will be 16 times faster than without spatial partitioning, but unfortunately $\frac{N}{16}$ is still $\Theta(N)$. BUT, this does indeed work better in practice.

Still, let's see if there's an even better way.

QuadTrees

Third Attempt: QuadTrees

X-Based Tree or Y-Based Tree

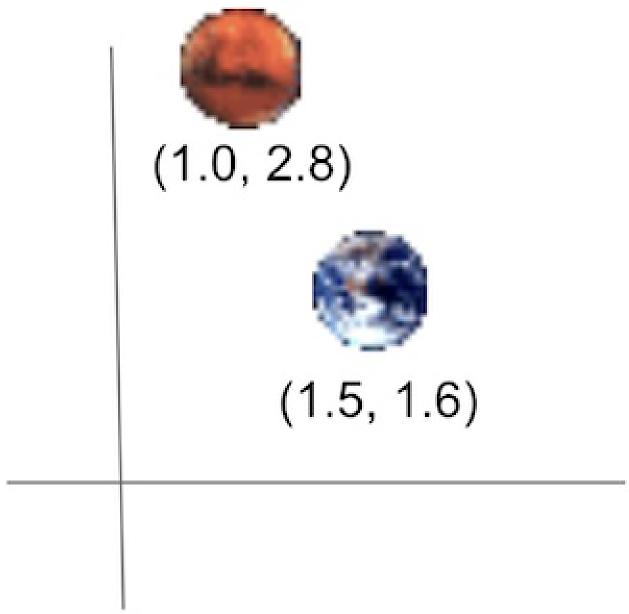


Video link

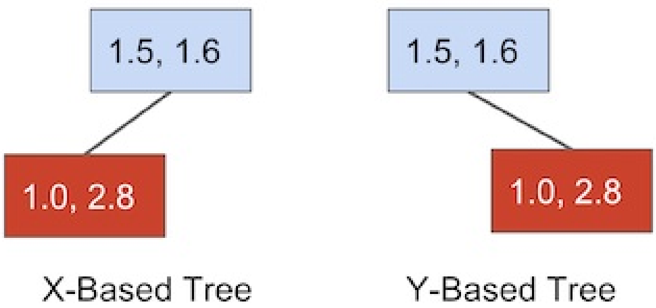
One key advantage of Search Trees over Hash Tables is that trees explicitly track the order of items. For example, finding the minimum item in a BST is $\Theta(\log N)$ time, but $\Theta(N)$ in a hash table. Let's try to leverage that to our advantage here to give us better performance for our motivating goals.

This isn't trivial though...in order to build a Binary Search Tree, we need to be able to compare objects. However, in two (or more) dimensional space, one object might be "less than" another in one dimension, but "greater than" the other in the other dimension. So which should be the "lesser" and "greater" for the purposes of our search tree?

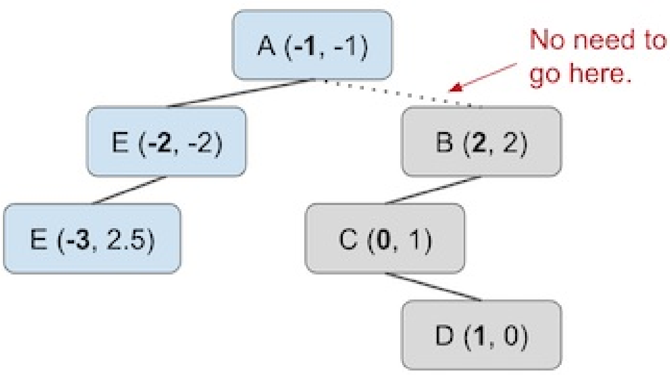
For example, below Mars is "less than" Earth in the x-dimension, but "greater than" Earth in the y-dimension.

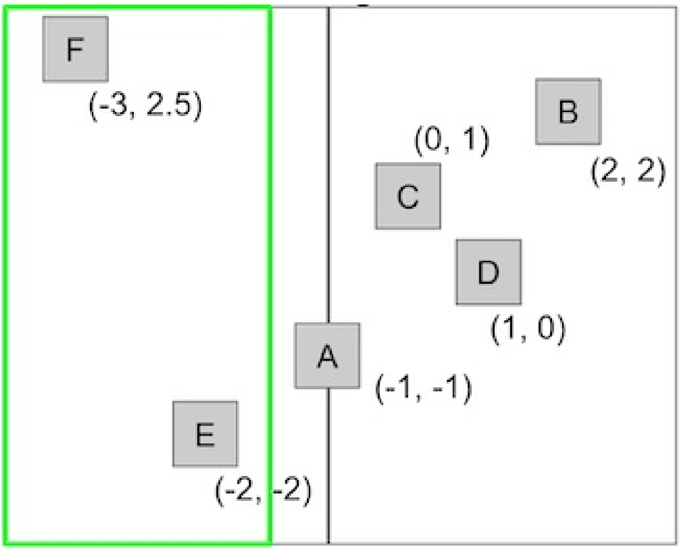


So which of these two representations shown below should we use? Remember that we don't want to tie-break arbitrarily because we need to be able to know for certain that a node is going to be down one particular path or not in the tree at all. Otherwise, we may lose our $\log N$ runtime.



Say we use the X-based Tree--that is--we construct a BST looking only at the x-coordinate. (We ignore the y-coordinate when organizing it.) For a larger example, we might get something like this:



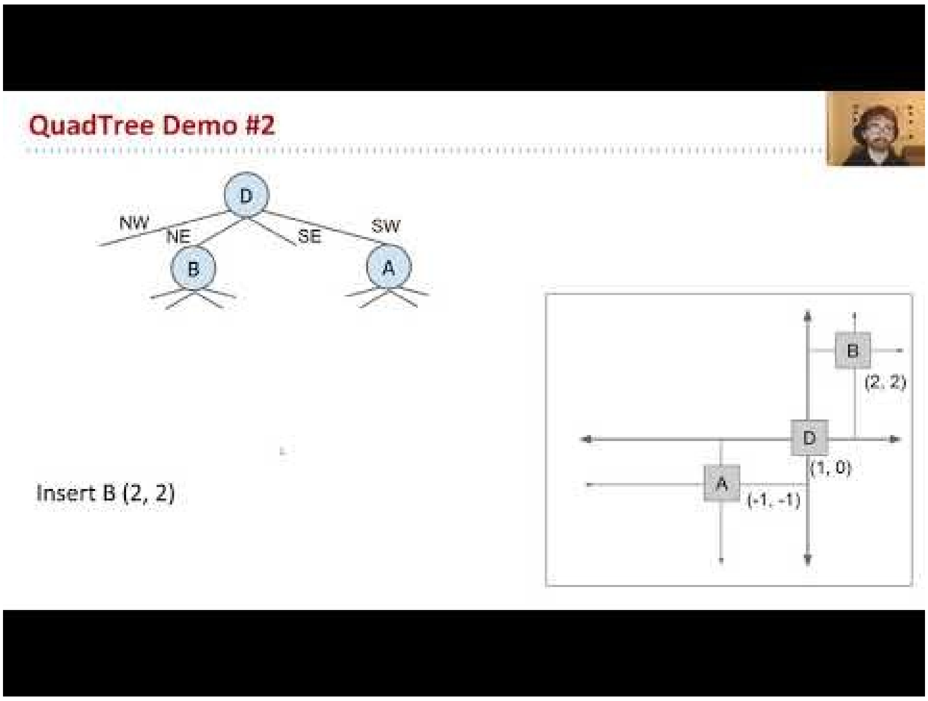


Notice that if we are performing a search on this tree, and we're looking for a point that has an x-coordinate less than -1, from the root when we choose to take the left path, we immediately get to discard everything in the right subtree. And this is analogous to saying that we have been able to restrict our search space from the entire image space, to just the green rectangle. The ability to skip searching through parts of your search tree is called "pruning".

However on the flip side, if we were looking for a point that has a particular y-coordinate, our X-based tree is not optimal for that kind of search and we'd have to perform a linear search on all the nodes.

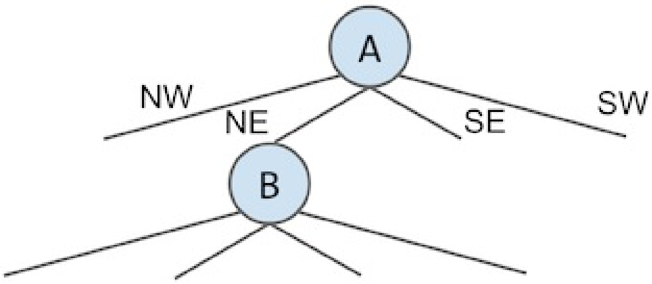
No matter whether we choose the X-based tree representation or the Y-based tree representation, we will always have suboptimal pruning; search in the optimized dimension will be $\log N$, but search in the non-optimized dimension will be N in runtime.

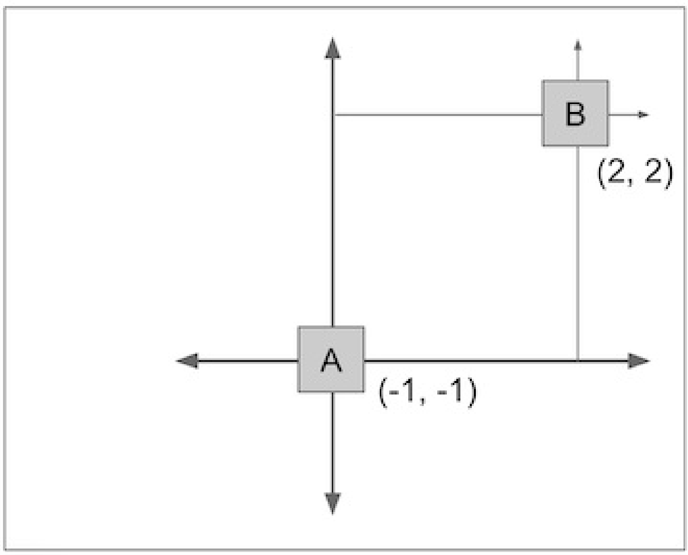
QuadTree



[Video link]

We can solve this problem by splitting in both directions simultaneously. This is the QuadTree.





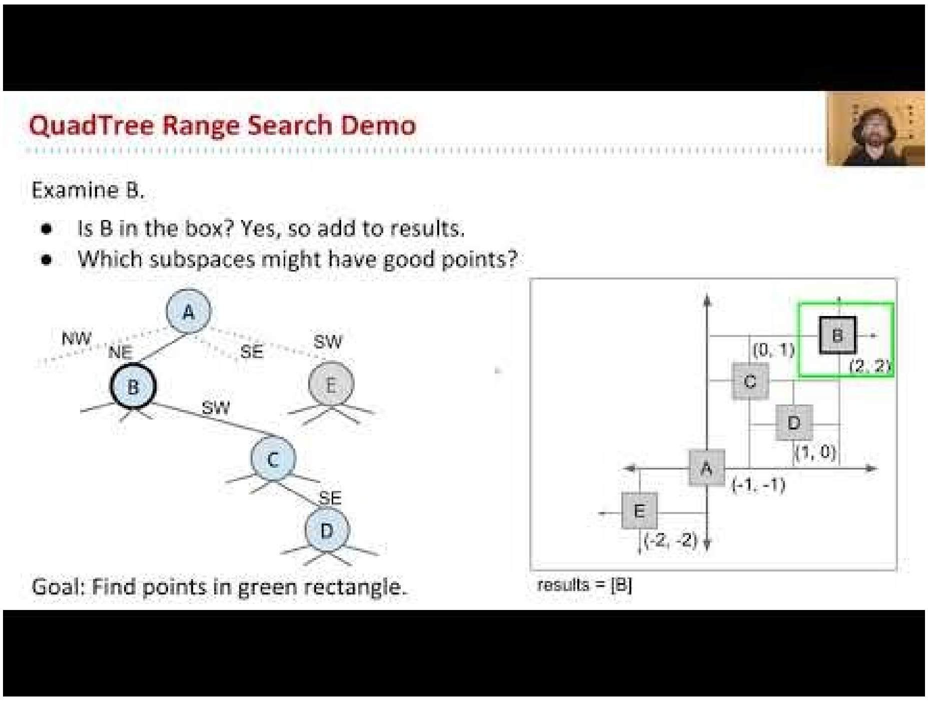
Here, we see that node A splits its surrounding area into a northwest, northeast, southeast, and southwest region. Since B resides in the northeast quadrant of A, when we insert B, we can put it as a child of A as its NE child.

Note that just like in a BST, the order in which we insert nodes determines the topology of the QuadTree.

Also note that QuadTrees are a form of spatial partitioning in disguise. Similar to how uniform partitioning created a perfect grid before, QuadTrees hierarchically partition by having each node "own" 4 subspaces.

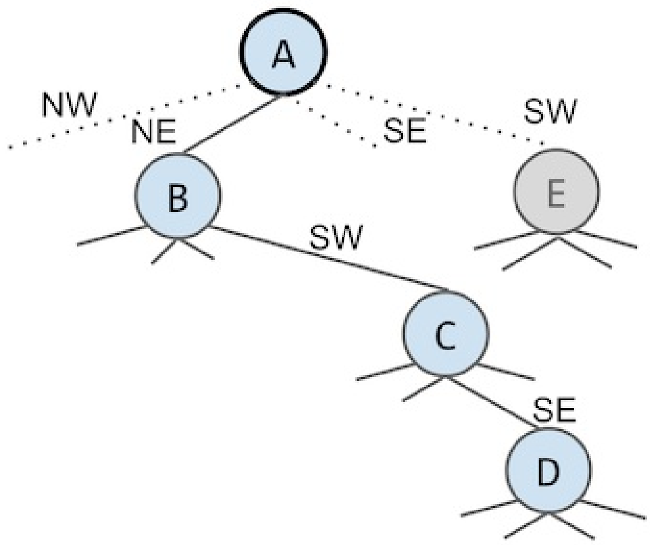
Effectively, spaces where there are many points are broken down into more finely divided regions, and in many cases this gives better performances.

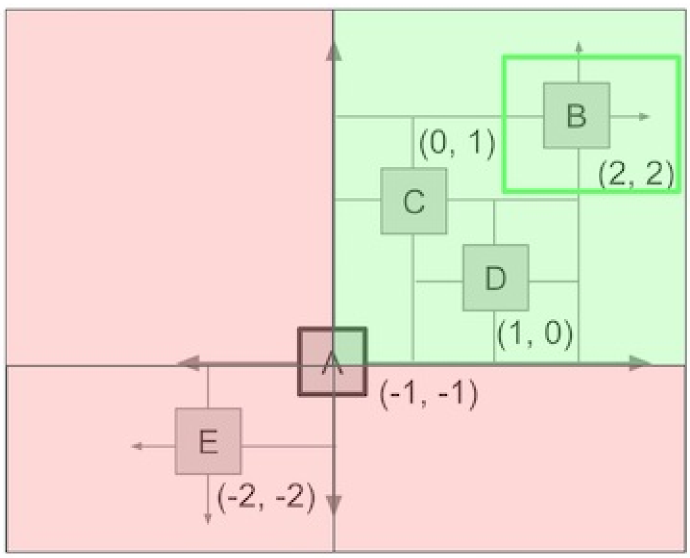
Range Search using a QuadTree



Video link

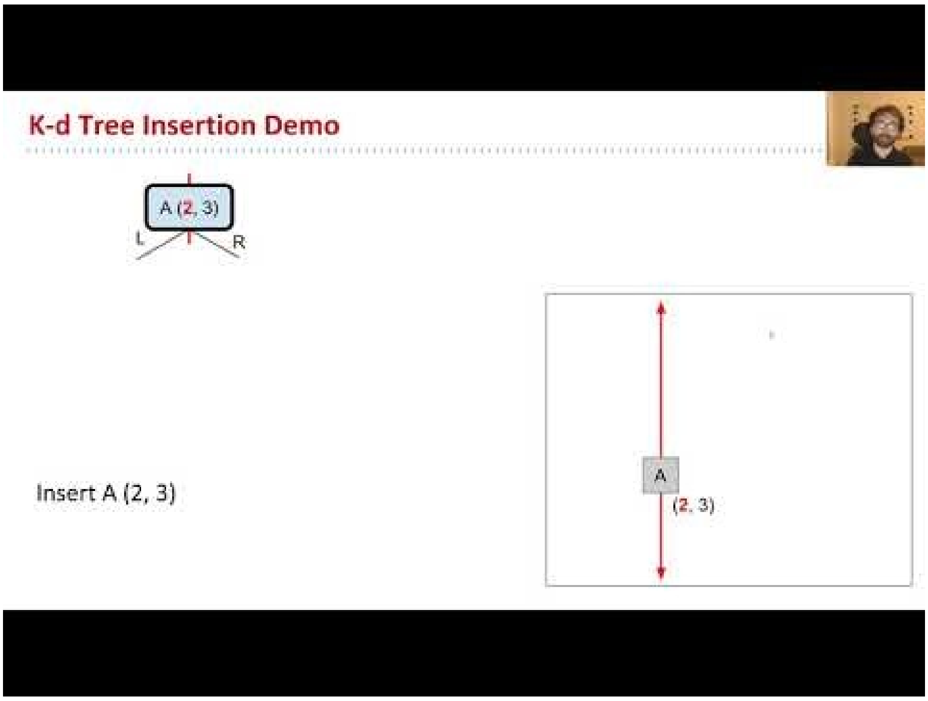
Notice that with the 4-way division imposed by each node of the QuadTree, we still have the pruning effect that was so advantageous in our X-Based Tree and Y-Based Tree! If we are looking for points inside a green rectangle as shown below, from any node we can decide whether the green rectangle lies within one or more quadrants, and only explore the branches/subtrees corresponding to those quadrants. All other quadrants can be safely ignored and pruned away. Below, we see that the green rectangle lies only in the northeast quadrant, and so the NW, SE, and SW quadrants can all be pruned away and left unexplored. We can proceed recursively.





Quad-Trees are great for 2-D spaces, because there are only 4 quadrants. However, what do we do if we want to move into higher dimension space? We'll explore another data structure in the next chapter that is equipped to tackle this question.

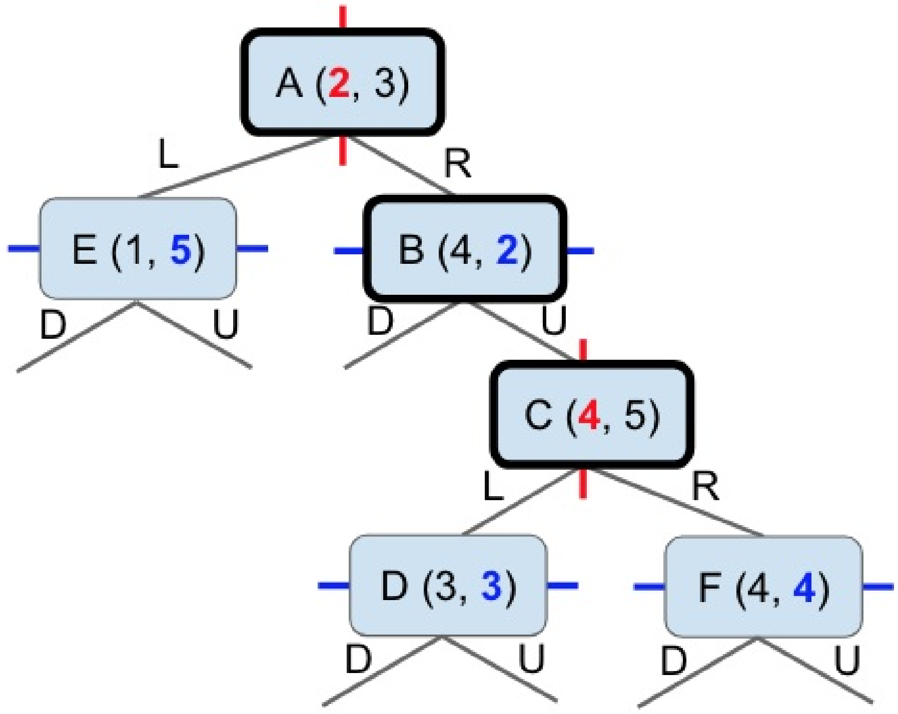
K-D Trees

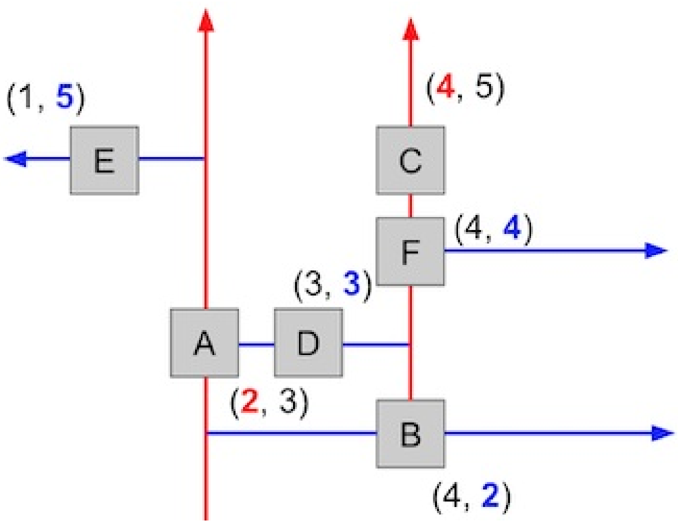


[Video link]

One way we can extend the hierarchical partitioning idea to dimensions greater than two is by using a K-D Tree. It works by rotating through all the dimensions level by level.

So for the 2-D case, it partitions like an X-based Tree on the first level, then like a Y-based Tree on the next, then as an X-based Tree on third level, a Y-based Tree on the fourth, etc.





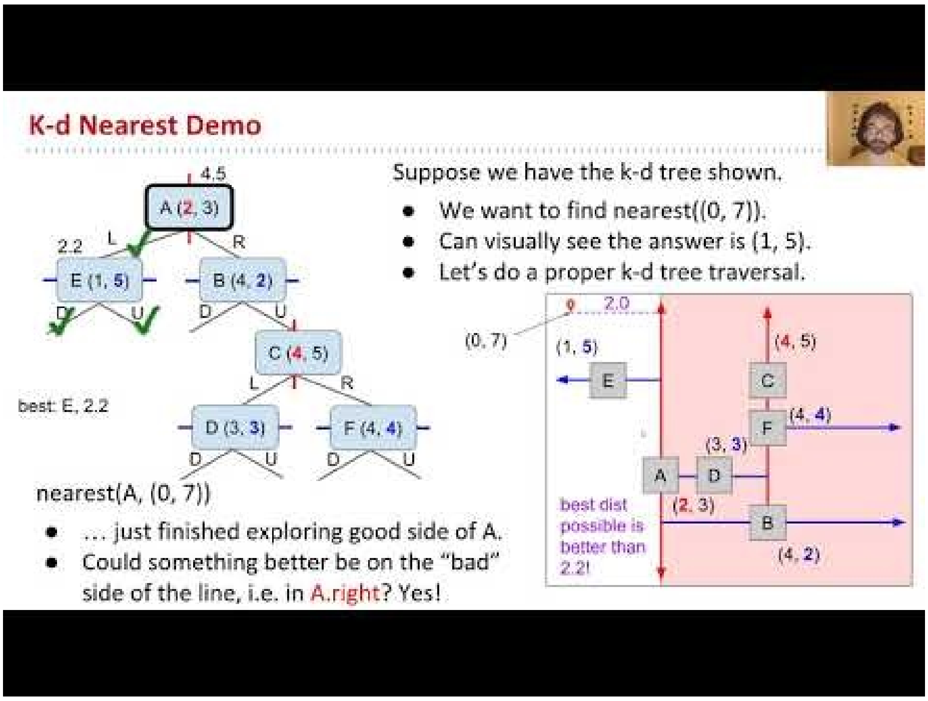
In the first graphic, you can see how each level is partitioned. In the one below, you can see the nodes of the tree in relation to one another in the 2-D space. **Note** when you are running operations on a K-D Tree or quadtree, you should be thinking about the tree structure (1st image) and not the 2-D space (2nd picture) because only the tree holds information about the levels.

For the 3-D case, it rotates between each of the three dimensions every three levels, and so on and so forth for even higher dimensions. Here you can see the advantages in a K-D Tree in how it is more easily generalized to higher dimensions. But, no matter how high the dimensions get, a K-D tree will always be a **binary** tree, since each level is partitioned into "greater" and "less than".

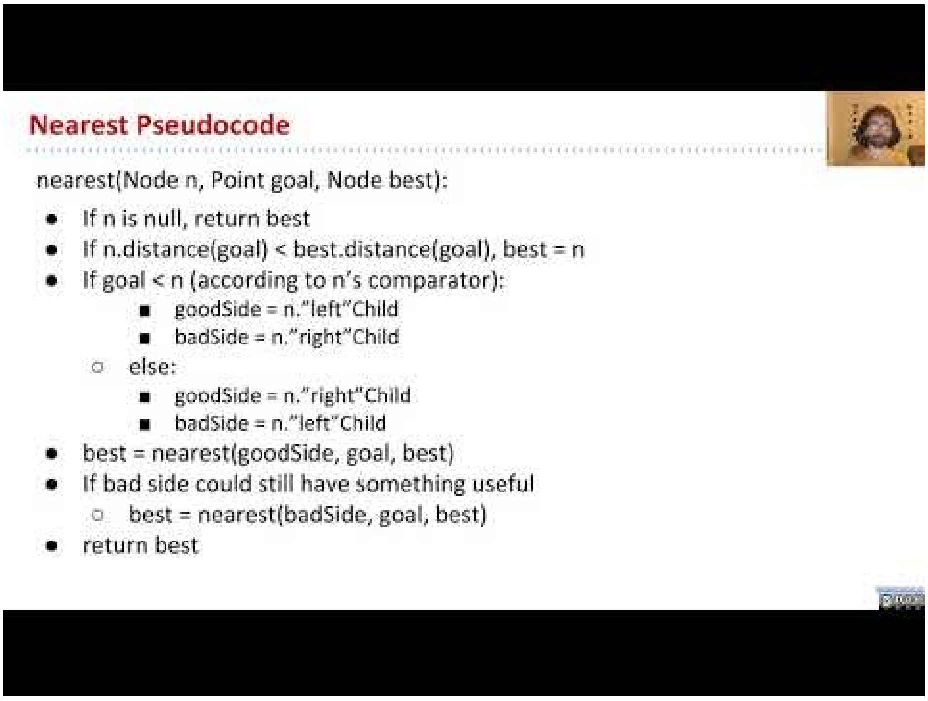
For a demo on K-D tree insertion, check out these slides

We can break ties by saying that items equal in one dimension should always fall the same way across the border. (For example, an item equal in the x-dimension to its parent node will always fall to the right of the partition.)

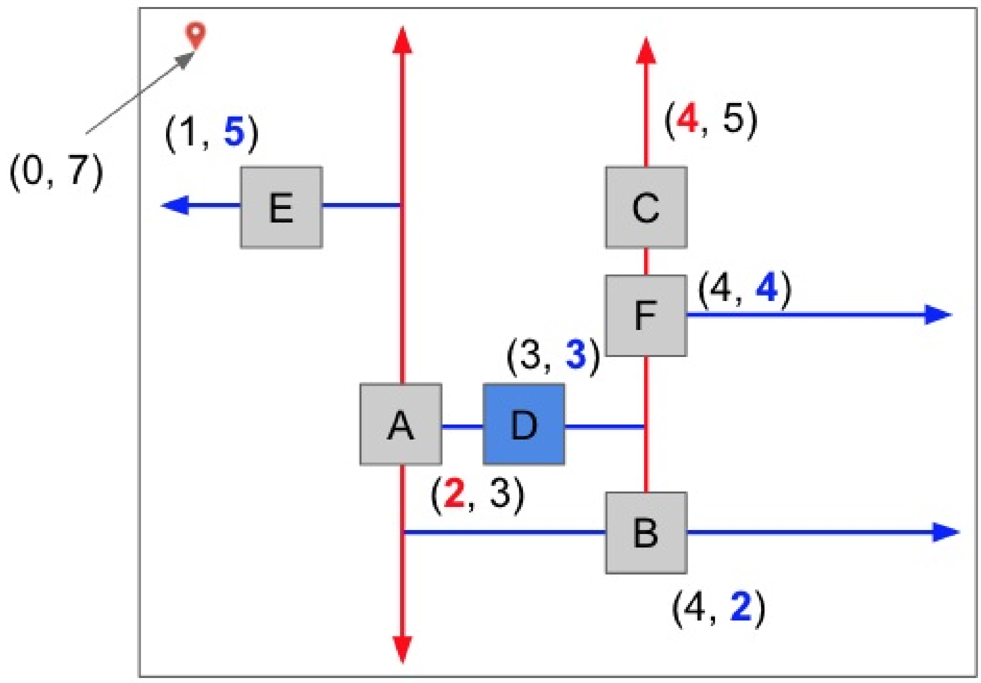
Nearest Neighbor using a K-D Tree



Video link

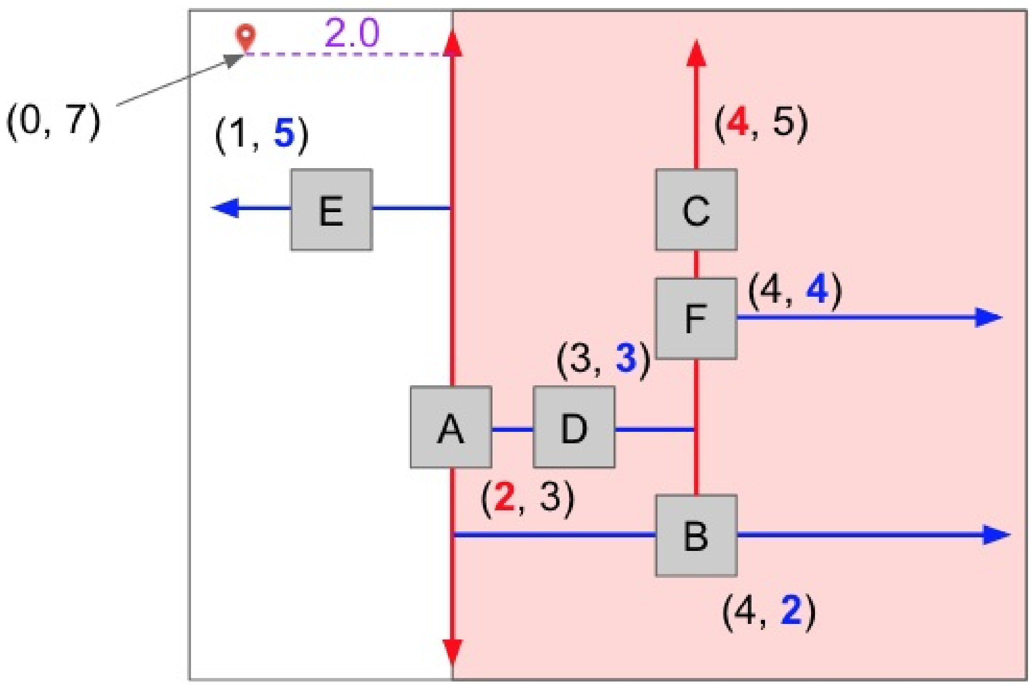


Video link



To find the point that is the nearest neighbor to a query point, we follow this procedure in our K-D Tree:

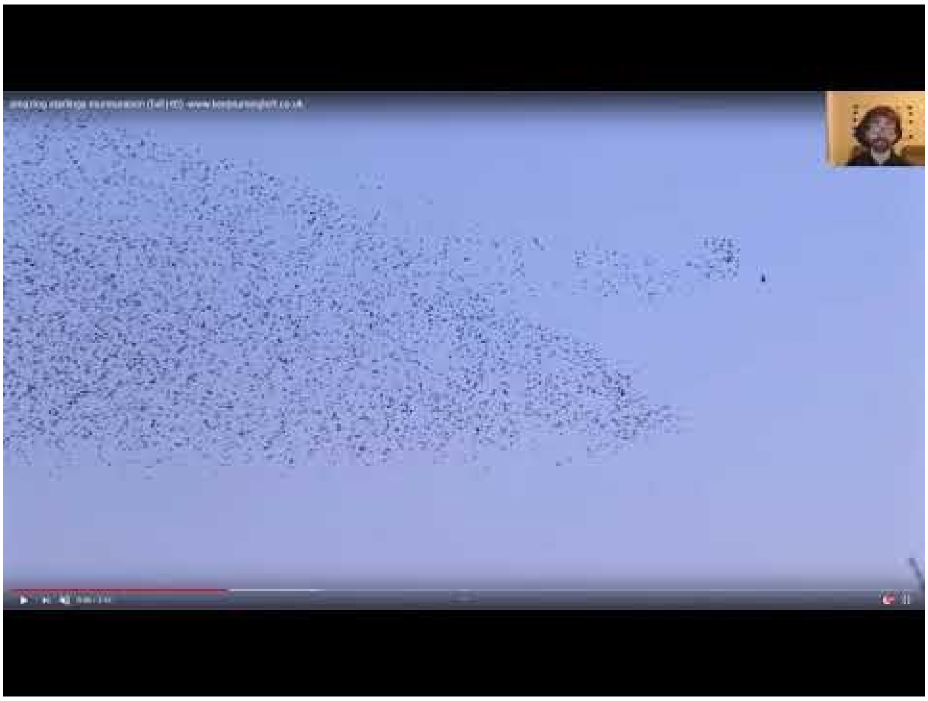
- Start at the root and store that point as the "best so far". Compute its distance to the query point, and save that as the "score to beat". In the image above, we start at A whose distance to the flagged point is 4.5.
- This node partitions the space around it into two subspaces. For each subspace, ask: "Can a better point be possibly found inside of this space?" This question can be answered by computing the shortest distance between the query point and the edge of our subspace (see dotted purple line below).



- Continue recursively for each subspace identified as containing a possibly better point.
- In the end, our "best so far" is the best point; the point closest to the query point.

For a step by step walkthrough, see these slides

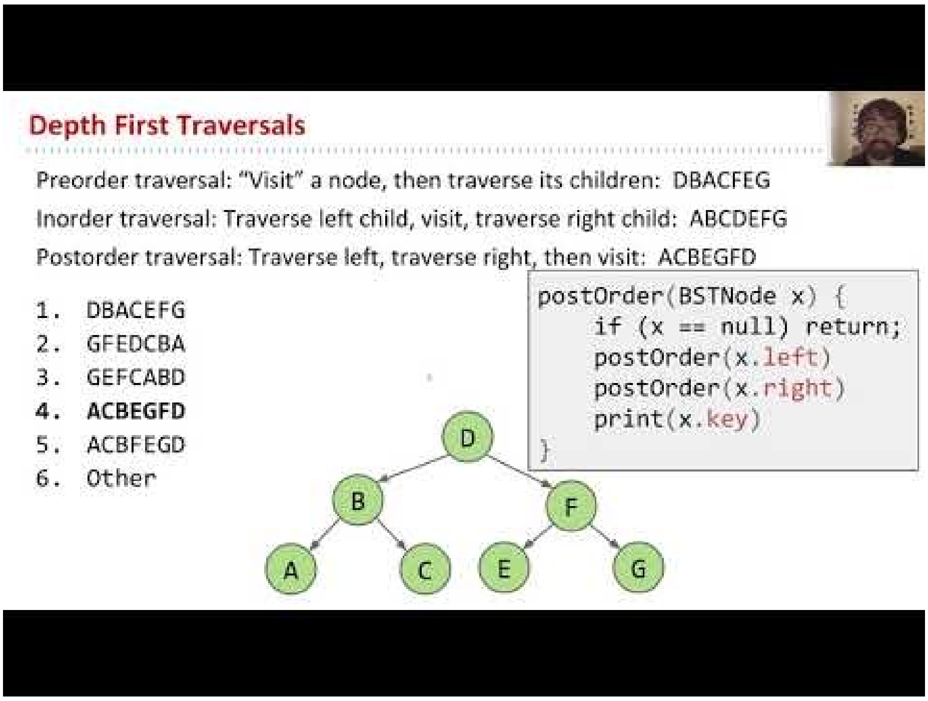
Summary and Applications



Video link

See the video above for a summary of this chapter, and some interesting applications of the presented data structures.

Trees and Traversals

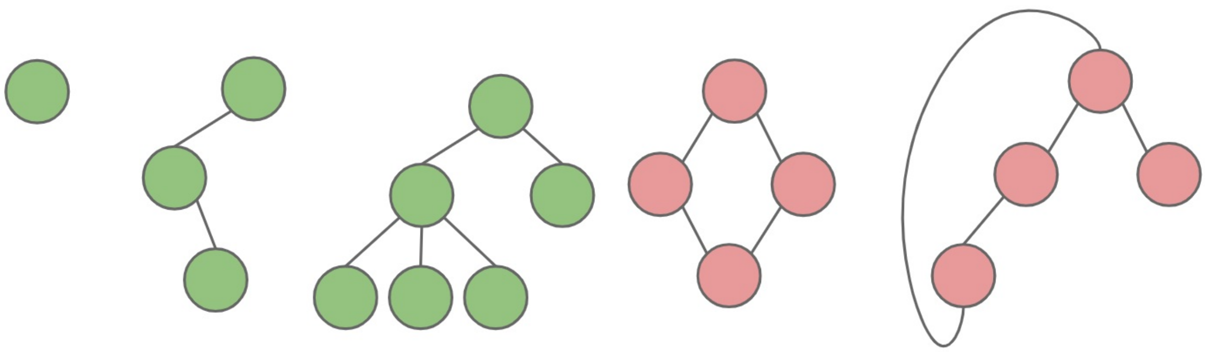


Video link

What is a tree?

Recall that a tree consists of:

- A set of nodes (or vertices). We use both terms interchangeably.
- A set of edges that connect those nodes.
  - **Constraint:** There is exactly one path between any two nodes.



The left-most structure is a tree. It has a node. It has no edges. That's *OK*!

The second and third structures are trees.

The fourth is not a tree. Why? There are two paths from the top node to the bottom node, and so this does not obey our constraint.

Exercise 17.1.1. Determine the reason why the fifth structure is not a tree. Also, modify the invalid trees above so that they are valid.

What is a rooted tree?

Recall that a rooted tree is a tree with a designated root (typically drawn as the top most node.)

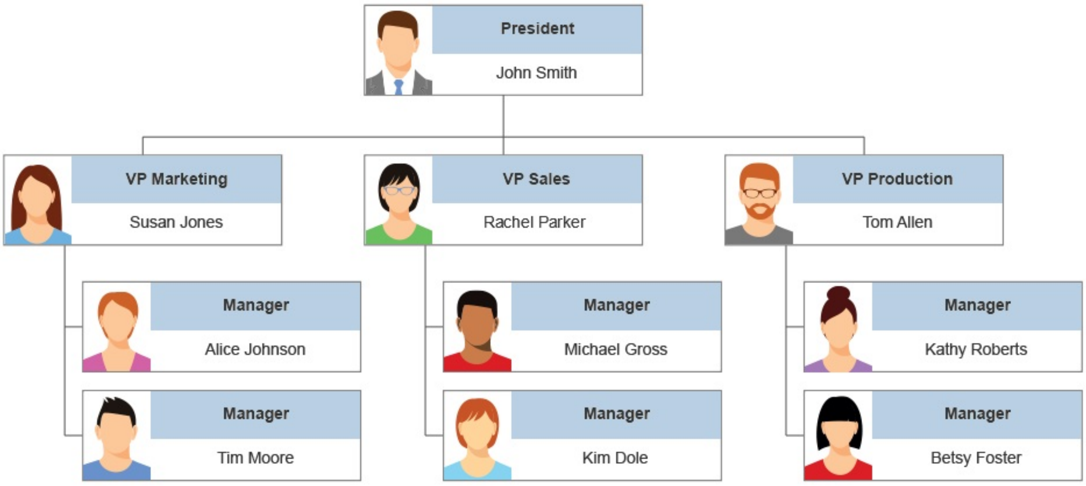
This gives us the notion of two more definitions

- A parent. Every node except the root has exactly one parent.
  - What if a node had 2 parents? Would it be a tree? (Hint: No.)
- A child. A node can have 0 or more children.
  - What if a node has 0 children? It's called a leaf.

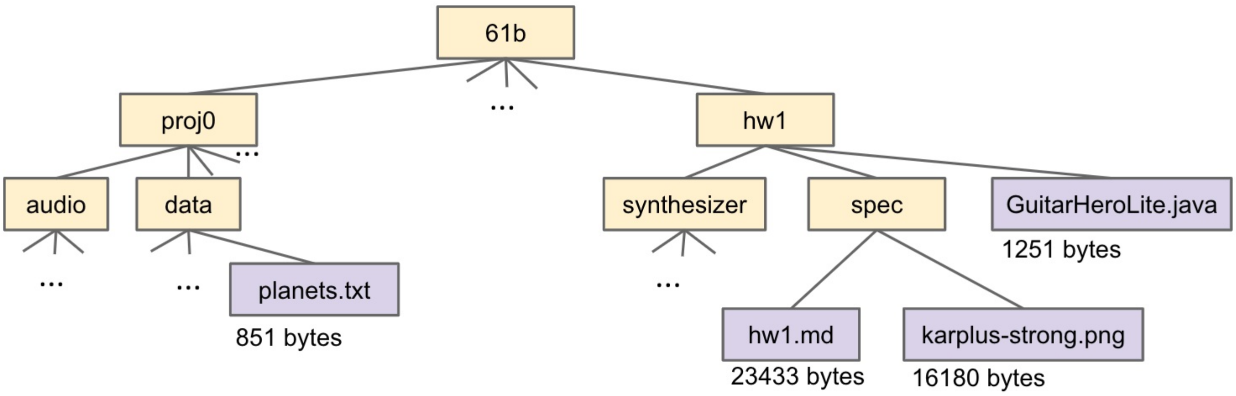
What are trees useful for?

So far, we've looked at Search Trees, Tries, Heaps, Disjoint Sets, etc. These were extremely useful in our journey to create efficient algorithms: speeding up searching for items, allowing prefixing, checking connectedness, and so on.

But the fact of the matter is that they are even more ubiquitous than we realize. Consider an organization chart. Here, the President is the 'root'. The 'VP's are children of the root, and so on.



Another tree structure is the `61b/` directory on your Desktop (it is on your Desktop, isn't it?). As we can see, when you traverse to a subfolder it goes to subsequent subfolders and so on. This is exactly tree-like!



Exercise 17.1.2. Think of other common uses of trees that weren't mentioned above. Try and determine possible implementations or designs of these trees.

Tree ~~Iteration~~ Traversal

Remember how we learned to iterate through lists? There was a way to iterate through lists that felt natural. Just start at the beginning... and keep going.

Or maybe we did some things that were a little strange, like iterate through the reverse of the list. Recall we also wrote iterators in discussion to skip over students who didn't write descriptions on the Office Hours queue.

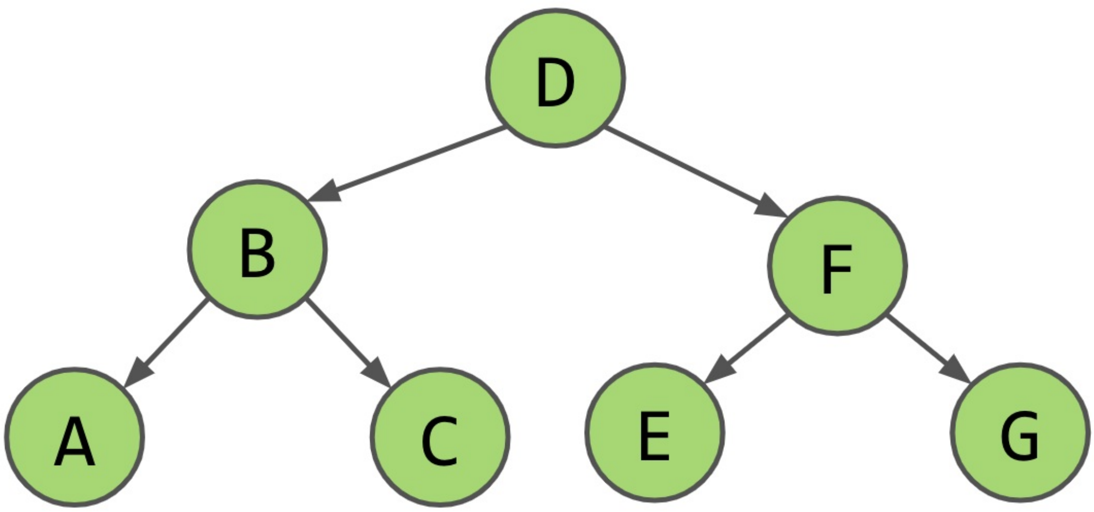
Now how do you iterate over a tree? What's the correct 'order'?

Before we answer that question, we must not use the word iteration. Instead, we'll call it 'traversing through a tree' or a 'tree traversal'. Why? No **real** reason, except that everyone calls iteration through trees 'traversals'. Maybe it's because the world likes alliterations.

So what are some natural ways to 'traverse' through a tree? As it turns out, there are a few — unlike a list which basically has one natural way to iterate through it:

1. Level order traversal.
2. Depth-First traversals — of which there are three: pre-order, in-order and post-order.

Let's test out these traversals mentioned above on the tree below.



Level Order Traversal

We'll iterate by levels, left to right. Level 0? D. Level 1? B and then F. Level 2? A, C, E, and G.

This gives us `D B F A C E G`.

Imagine each level was a sentence in english, and we just read it off line by line.

Exercise 17.2.1. Write the code for performing a level order traversal (warning, this is more difficult than the writing the other traversals). *Hint*: You will want to keep track of what level you are at.

Pre-order Traversal

Here's the idea behind pre-order traversal. Start at the root. **Visit the root** (aka, do the **action** you want to do.) The action here is "print".

So, we'll print the root. D. There we go.

Now, go left, and recurse. Then, go right and recurse.

So now we've gone left. We're at the B node. Print it. B. We'll go left after printing. (Remember, after we're done with B's left branch, we'll come back up and visit B's right.)

Keep following this logic, and you get `D B A C F E G`.

```
preOrder(BSTNode x) {
    if (x == null) return;
    print(x.key)
    preOrder(x.left)
    preOrder(x.right)
}
```

In-order Traversal

Slightly different, but same big-picture idea. Here, instead of **visiting** (aka **printing**) first, we'll first visit the left branch. Then we'll print. Then we'll visit the right branch.

So we start at D. We don't print. We go left.

We start at B. We don't print. We go left.

We start at A. We don't print. We go left. We find nothing. We come back up, and print A.

Then go to A's right. Find nothing. Go back up. Now we're at B. Remember, we print after visiting left and before visiting right, so now we'll print B, then we'll visit right.

Keep following this and you get `A B C D E F G`.

An alternative way to think about this is as follows:

First, we're at D. We know we'll print out the items from left, then D, then items from right.

[items from left] D [items from right].

Now what's [items from left] equal to? We'll start at B, print out left, then print B, then print stuff from right of B.

[items from left] = [items from B's left] B [items from B's right] = A B C.

A B C D [stuff from right] = A B C D E F G.

```
inOrder(BSTNode x) {
    if (x == null) return;
    inOrder(x.left)
    print(x.key)
    inOrder(x.right)
}
```

Post-order Traversal

Again, same big-picture idea, but now we'll print left branch, then right branch, then ourselves.

Using the method we devised earlier, the result looks like:

[items from left] [items from right] D.

What's [items from left]? It's the output from the B subtree.

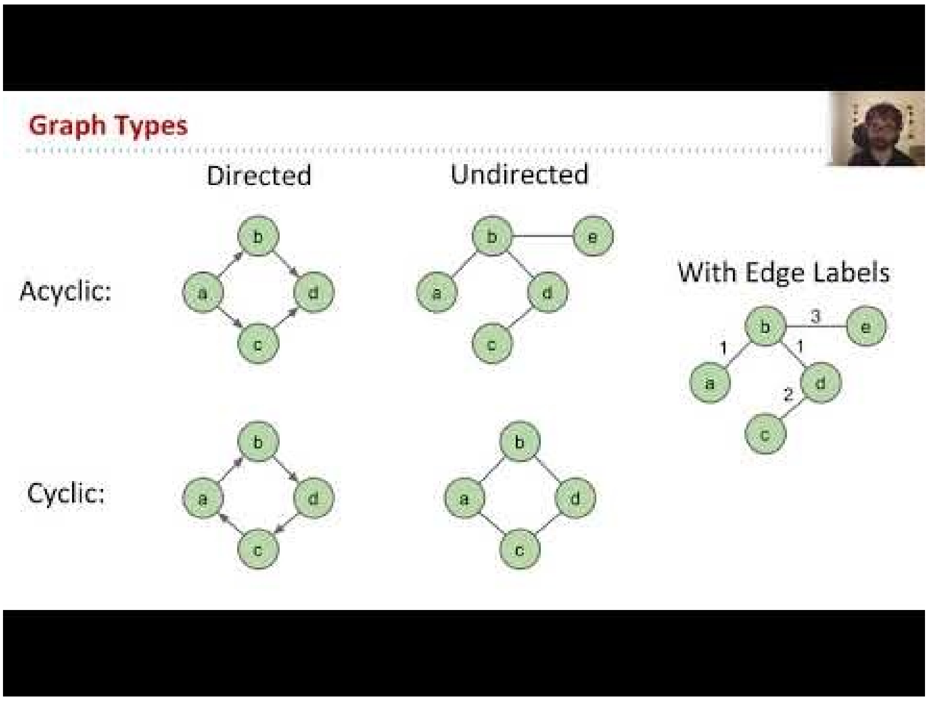
If we're at B, we'd get [items from left of B] [items from right of B] B, which is equal to A C B.

Following this through, we get: `A C B E G F D`

```
postOrder(BSTNode x) {
    if (x == null) return;
    postOrder(x.left)
    postOrder(x.right)
    print(x.key)
}
```

Graphs

Trees are great, aren't they? But as we saw, we could draw some things using nodes and edges that weren't trees. Specifically, our restriction that there can only be one path between any two nodes didn't fit every situation. Let's see what happens when we get rid of that restriction.



[Video link]

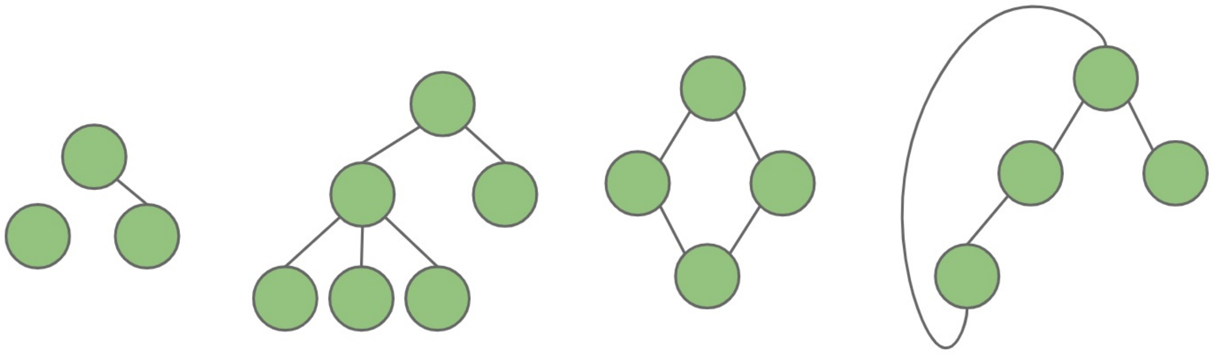
What is a graph?

A graph consists of:

- A set of nodes (or vertices)
- A set of zero of more edges, each of which connects two nodes.

That's it! No other restrictions.

All of the structures below in green? Everything is a valid graph! The second one is also a tree, but none of the others are.

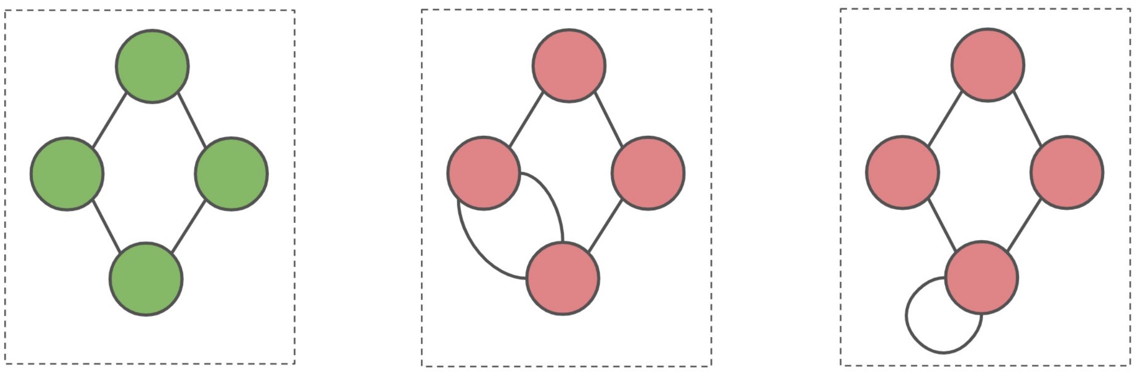


In general, note that **all trees are also graphs, but not all graphs are trees.**

Simple Graphs only

Graphs can be divided into two categories: *simple* graphs and *multigraphs* (or complicated graphs, a term I invented, because that's how I like to think of them.) Fortunately, in this course (and almost all applications and research) focuses only on simple graphs. So when we say "graph" in this course, you should always think of a "simple graph" (unless we say otherwise.)

Well, it's time to address the elephant in the room. What's a simple graph?



Look at the graphs in red. The graph in the middle has 2 distinct edges going from/to bottom-next to/from bottom-left node. In other words, there are multiple edges between two nodes. This is **not** a simple graph, and we ignore their existence unless specified otherwise. Graphs like these are called multigraphs.

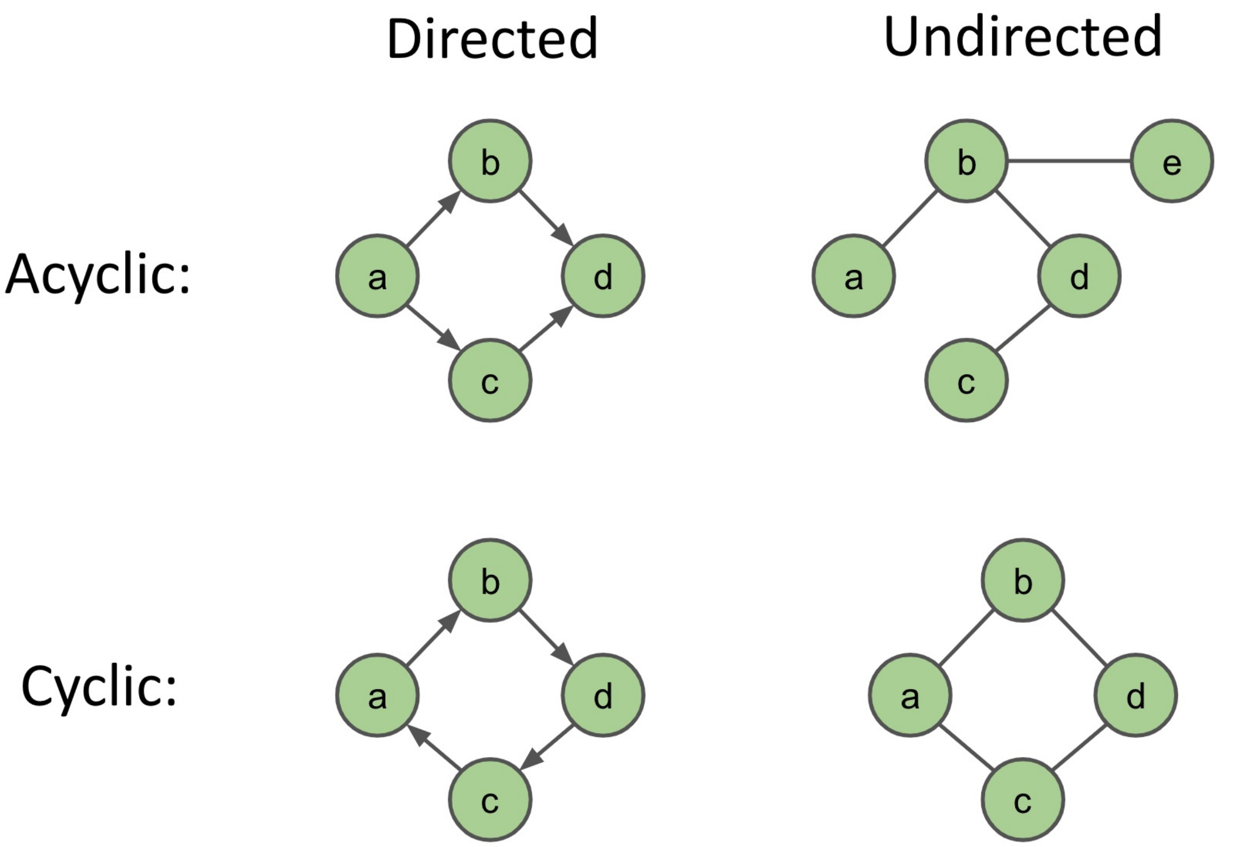
Look at the third graph. It has a loop! An edge from a node to itself! We don't allow this either. Graphs like these are sometimes categorized as multigraphs, and sometimes, even multigraphs explicitly ban self-loops.

More categorizations.

Graphs are simple in the following text, and in this course, unless specified otherwise. But there are more categorizations.

There are undirected graphs, where an edge `(u, v)` can mean that the edge goes from the nodes u to v and from the nodes v to u too. There are directed graphs, where the edge `(u, v)` means that the edge starts at u, and goes to v (and the vice versa is not true, unless the edge `(v, u)` also exists.) Edges in directed graphs have an arrow.

There are also acyclic graphs. These are graphs that don't have any cycles. And then there are cyclic graphs, i.e., there exists a way to start at a node, follow some **unique** edges, and return back to the same node you started from.



In the above picture, we can clearly see the difference between how we draw directed and undirected edges.

Take a look at the cyclic graphs. If you start at `a`, you can run back around using only distinct edges and get back to `a`. Thus, the graph is cyclic.

Take a look at the top-left graph. Is there any node `n`, such that if you start at `n`, you can follow some distinct edges, and get back to `n`? Nope! (Remember than for directed edges, you must follow the directions. You can go from `a` to `b` but not `b` to `a`.)

More definitions

Graph Terminology

- Graph:
    - Set of ***vertices***, a.k.a. ***nodes***.
    - Set of ***edges***: Pairs of vertices.
    - Vertices with an edge between are ***adjacent***.
    - Optional: Vertices or edges may have ***labels*** (or ***weights***).
- A ***path*** is a sequence of vertices connected by edges.
- A ***cycle*** is a path whose first and last vertices are the same.
    - A graph with a cycle is 'cyclic'.
- Two vertices are ***connected*** if there is a path between them. If all vertices are connected, we say the graph is connected.

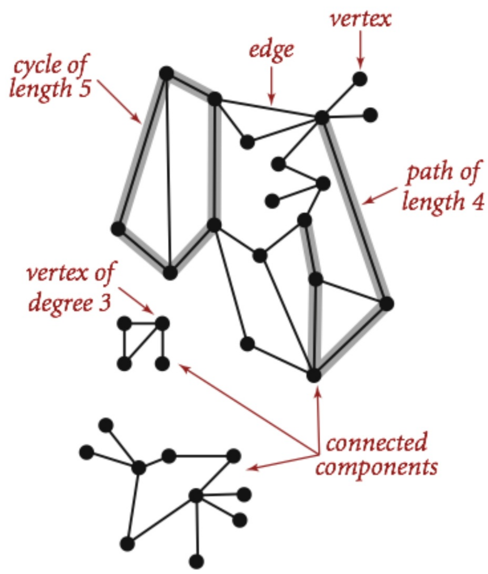
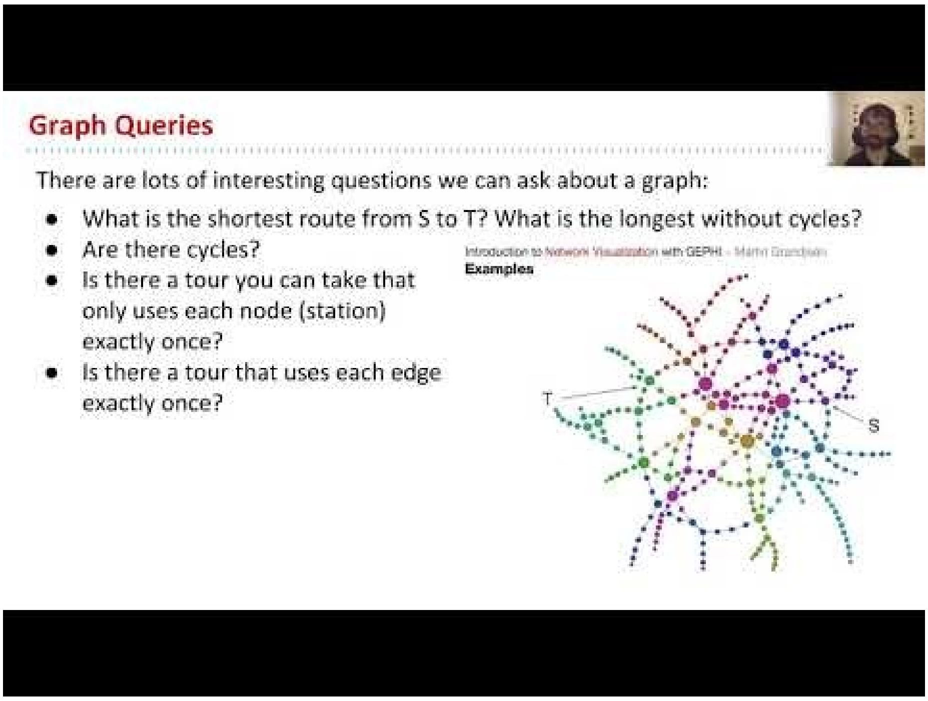


Figure from Algorithms 4th Edition

Graph Problems



[Video link]

There are many questions we can ask about a graph.

For example,

- **s-t Path**: Is there a path between vertices s and t?
- **Connectivity**: Is the graph connected, i.e. is there a path between all vertices?
- **Biconnectivity**: Is there a vertex whose removal disconnects the graph?
- **Shortest s-t Path**: What is the shortest path between vertices s and t?
- **Cycle Detection**: Does the graph contain any cycles?
- **Euler Tour**: Is there a cycle that uses every edge exactly once?
- **Hamilton Tour**: Is there a cycle that uses every vertex exactly once?
- **Planarity**: Can you draw the graph on paper with no crossing edges?
- **Isomorphism**: Are two graphs isomorphic (the same graph in disguise)?

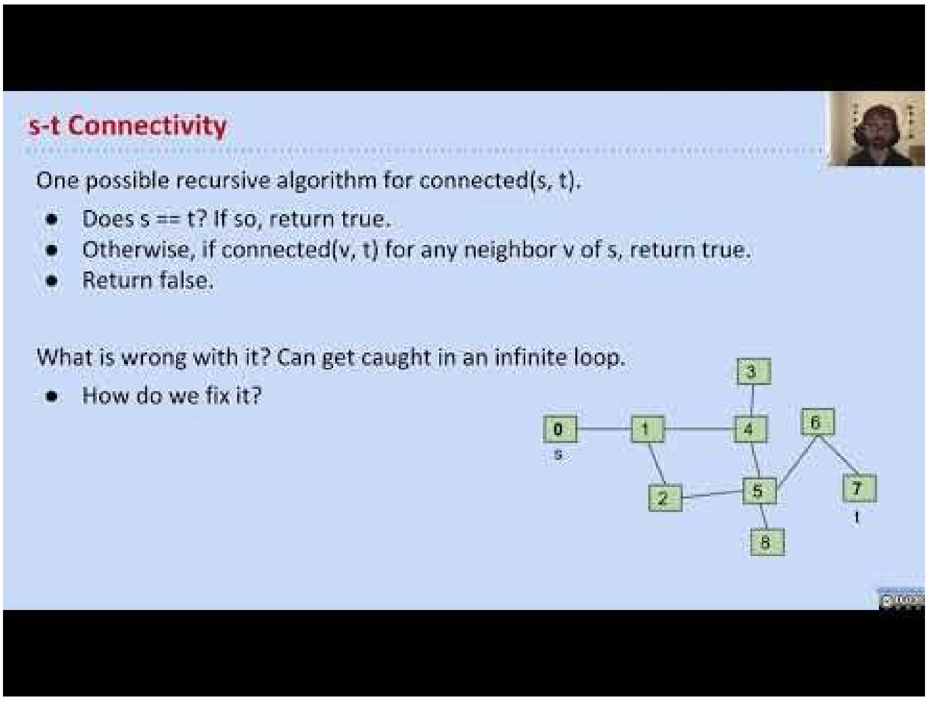
What's cool and also weird about graph problems is that it's very hard to *tell* which problems are very hard, and which ones aren't all that hard.

For example, consider the Euler Tour and the Hamilton Tour problems. The former... is a solved problem. It was solved as early as 1873. The solution runs in $O(E)$ where E is the number of edges in the graph.

The latter? If you were to solve it efficiently today, you would win every Math award there was, become one of the most famous computer scientists, win a million dollars, etc. No one has been able to solve this **efficiently**. The best known algorithms run in exponential times. People have been working on it for many decades!

One step at a time!

Alright, well, before we solve the million dollar problem, let's solve the first one on the list. Given a source vertex s and a target vertex t, is there a path between s and t?



Video link

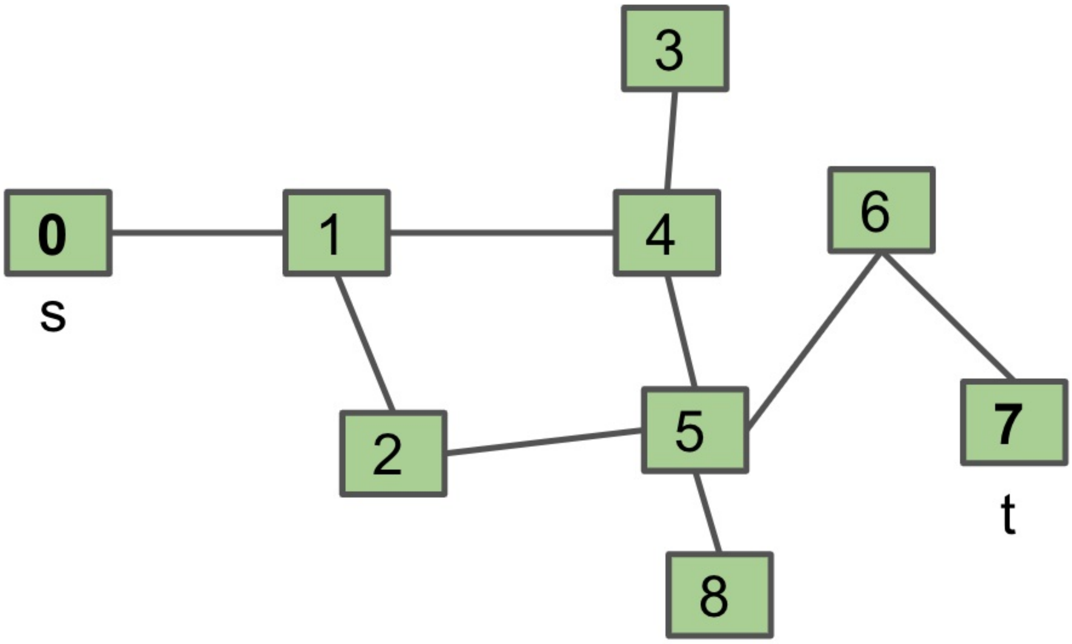
In other words, write a function `connected(s, t)` that takes in two vertices and returns whether there exists a path between the two.

To begin, let's guess that we have the following code:

```
if (s == t):
    return true;

for child in neighbors(s):
    if isconnected(child, t):
        return true;

return false;
```

Exercise 17.4.1. Before you move on, please read the code above, and spend time thinking about it. Does it work? Is it efficient? Run through a couple scenarios.

Alright, so, let's try it out.



We start with `connected(0, 7)`? That recursively calls `connected(1, 7)`, which then recursively calls `connected(0, 7)`. Uh-oh. Infinite looping has occurred.

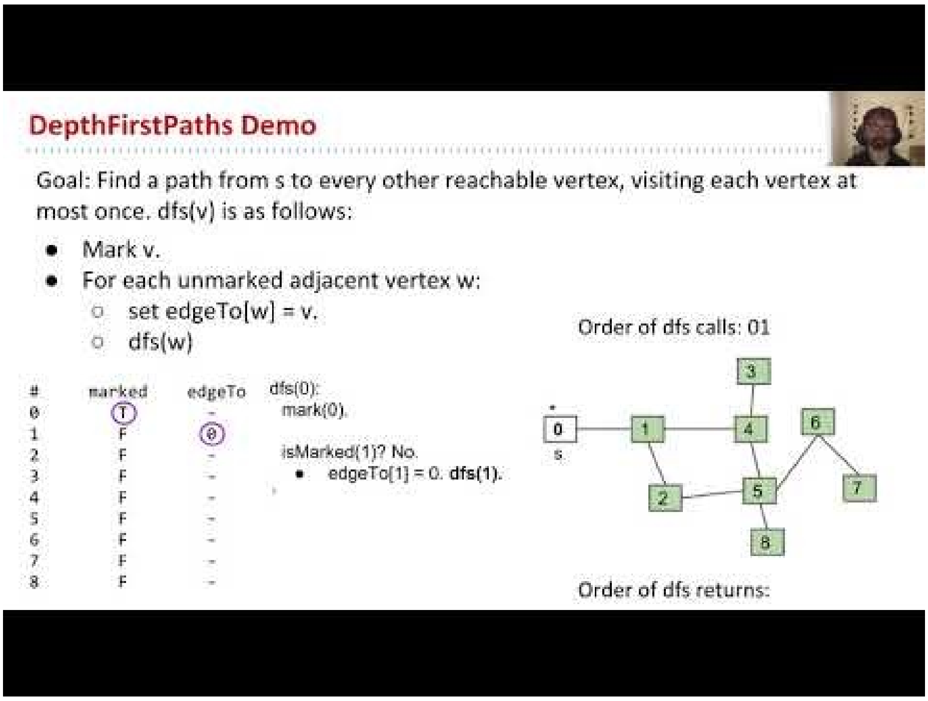
Exercise 17.4.2. How could we fix this? Once again, thinking about this. What was the problem? We visited `s` again... but did we need to?

Alright, let's try a "remember what we visited" approach.

```
mark s  // i.e., remember that you visited s already
if (s == t):
    return true;

for child in unmarked_neighbors(s): // if a neighbor is marked, ignore!
    if isconnected(child, t):
        return true;

return false;
```

As it turns out, this does work! Follow the example in these slides to see how.

Woah, what did we just develop?



Video link

You may not have realized it, but we just developed a **depth-first traversal** (like pre-order, post-order, in-order) but for graphs. What did we do? Well, we marked ourself. Then we visited our first child. Then our first child marked itself, and visited its children. Then our first child's first child marked itself, and visited its children.

Intuitively, we're going deep (i.e., down our family tree to our first child, our first child's first child aka our first grandchild, our first grandchild's first child, and so on... visiting this entire lineage), before we even touch our second child.

Up next, we'll see the opposite notion, where first we visit all our children, then our grandchildren, and so on.

BFS

In Chapter 17.4, we developed DFS (Depth First Search) Traversal for graphs. In DFS, we visit down the entire lineage of our first child before we even begin to look at our second child - we literally search ***depth first***.

Here, we will talk about BFS (Breadth First Search) (also known as Level Order Traversal). In BFS, we visit all of our immediate children before continuing on to any of our grandchildren. In other words, we visit all nodes 1 edges from our source. Then, all nodes 2 edges from our source, etc.

The pseudocode for BFS is as follows:

```
Initialize the fringe (a queue with the starting vertex) and mark that vertex.
Repeat until fringe is empty:
Remove vertex v from the fringe.
For each unmarked neighbor n of v:
Mark n.
Add n to fringe.
Set edgeTo[n] = v.
Set distTo[n] = distTo[v] + 1.
```

A *fringe* is just a term we use for the data structure we are using to store the nodes on the frontier of our traversal's discovery process (the next nodes it is waiting to look at). For BFS, we use a queue for our fringe.

`edgeTo[...]` is a map that helps us track how we got to node `n`; we got to it by following the edge from `v` to to `n`.

`distTo[...]` is a map that helps us track how far `n` is from the starting vertex. Assuming that each edge is worth a distance of `1`, then the distance to `n` is just one more than the distance to get to `v`. Why? We can use the way we know how to get to `v`, then pay one more to arrive at `n` via the edge that necessarily exists between `v` and `n` (it must exist since in the `for` loop header, `n` is defined as a neighbor of `v`).

This slide deck illustrates how this pseudocode can be carried out on an example graph.

DFS vs BFS

Question 18.1: What graph traversal algorithm uses a stack rather than a queue for its fringe?

Answer 18.1: DFS traversal.

Note however that DFS and BFS differ in more than just their fringe data structure. They differ in the order of marking nodes. For DFS we mark nodes only once we visit a node - aka pop it from the fringe. As a result, it's possible to have multiple instances of the same node on the stack at a time if that node has been queued but not visited yet. With BFS we mark nodes as soon as we add them to the fringe so this is not possible.

Recursive DFS implements this naturally via the recursive stack frames; iterative DFS implements it manually:

```
Initialize the fringe, an empty stack
    push the starting vertex on the fringe
    while fringe is not empty:
        pop a vertex off the fringe
        if vertex is not marked:
            mark the vertex
            visit vertex
            for each neighbor of vertex:
                if neighbor not marked:
                    push neighbor to fringe
```

Representing Graphs

Let's spend some time now talking about how to implement these graphs and graph algorithms in code.

We will discuss our choice of **API**, and also the **underlying data structures** used to represent the graph. Our decisions can have profound implications on our *runtime*, *memory usage*, and *difficulty of implementing various graph algorithms*.

Graph API

An API (Application Programming Interface) is a list of methods available to a user of our class, including the method signatures (what arguments/parameters each function accepts) and information regarding their behaviors. You have already seen APIs from the Java developers for the classes they provide, such as the Deque.

For our Graph API, let's use the common convention of assigning each unique node to an integer number. This can be done by maintaining a map which can tell us the integer assigned to each original node label. Doing so allows us to define our API to work with integers specifically, rather than introducing the need for generic types.

We can then define our API to look something like this perhaps:

```
public class Graph {
  public Graph(int V):               // Create empty graph with v vertices
  public void addEdge(int v, int w): // add an edge v-w
  Iterable<Integer> adj(int v):      // vertices adjacent to v
  int V():                           // number of vertices
  int E():                           // number of edges
...
```

Clients (people who wish to use our Graph data structure), can then use any of the functions we provide to implement their own algorithms. The methods we provide can have a significant impact on how easy/difficult it may be for our clients to implement particular algorithms.

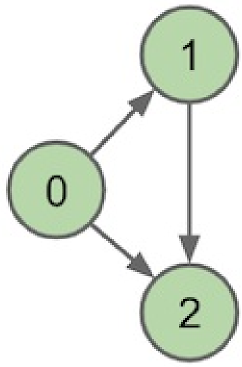
Graph Representations

Next, we'll talk about the underlying data structures that can be used to represent our graph.

Adjacency Matrix

One way we can do this is by using a 2D array. There is an edge connecting vertex `s` to `t` iff that corresponding cell is `1` (which represents `true`). Notice that if the graph is undirected, the adjacency matrix will be symmetric across its diagonal (from the top left to the bottom right corners).

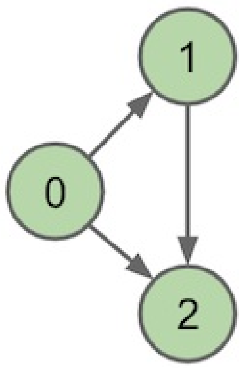
s \ t	0	1	2
0	0	1	1
1	0	0	1
2	0	0	0



Edge Sets

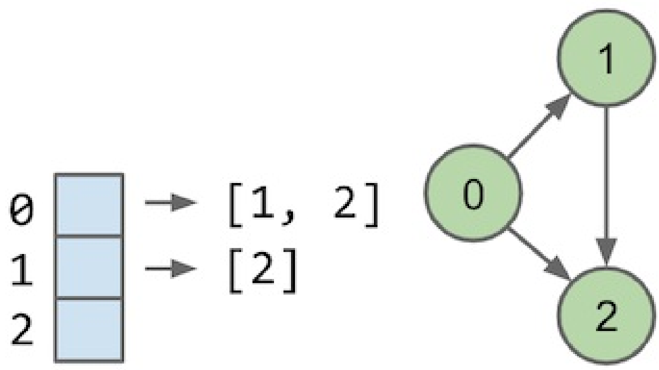
Another way is to store a single set of all the edges.

```
{(0, 1), (0, 2), (1, 2)}
```



Adjacency Lists

A third way is to maintain an array of lists, indexed by vertex number. Iff there is an edge from `s` to `t` , the list at array index `s` will contain `t` .



In practice, adjacency lists are most common since graphs tend to be sparse (there are not many edges in each bucket).

Efficiency

Your choice of underlying data structure can impact the runtime and memory usage of your graph. This table from the slides summarizes the efficiencies of each representation for various operations. Do not copy this on to your cheatsheet without taking the time to first understand where these bounds come from. The lecture contained walkthroughs explaining the rationale behind several of these cells.

idea	addEdge(s, t)	for(w : adj(v))	print()	hasEdge(s, t)	space used
adjacency matrix	Θ(1)	Θ(V)	Θ(V^2)	Θ(1)	Θ(V^2)
list of edges	Θ(1)	Θ(E)	Θ(E)	Θ(E)	Θ(E)
adjacency list	Θ(1)	Θ(1) to Θ(V)	Θ(V+E)	Θ(degree(v))	Θ(E+V)

Further, DFS/BFS on a graph backed by adjacency lists runs in $O(V+E)$, while on a graph backed by an adjacency matrix runs in $O(V^2)$. See the slides for help in understanding why.

Shortest Paths

Recalls

So far, we have methods to do the following

- find a path from a given vertex, s, to every reachable vertex in the graph.
- find a **shortest** path from a given vertex, s to every reachable vertex in the graph. (...or do we?)

Before we answer the mysterious question posed above, let's further recall the two types of searches we could use to do the above two things: BFS or DFS.

Are both going to always be correct? Yes. Does one give better results? BFS finds you the **shortest** paths whereas DFS does not. Is one more efficient than the other, runtime-wise? No. Is one more efficient than the other, space-wise?

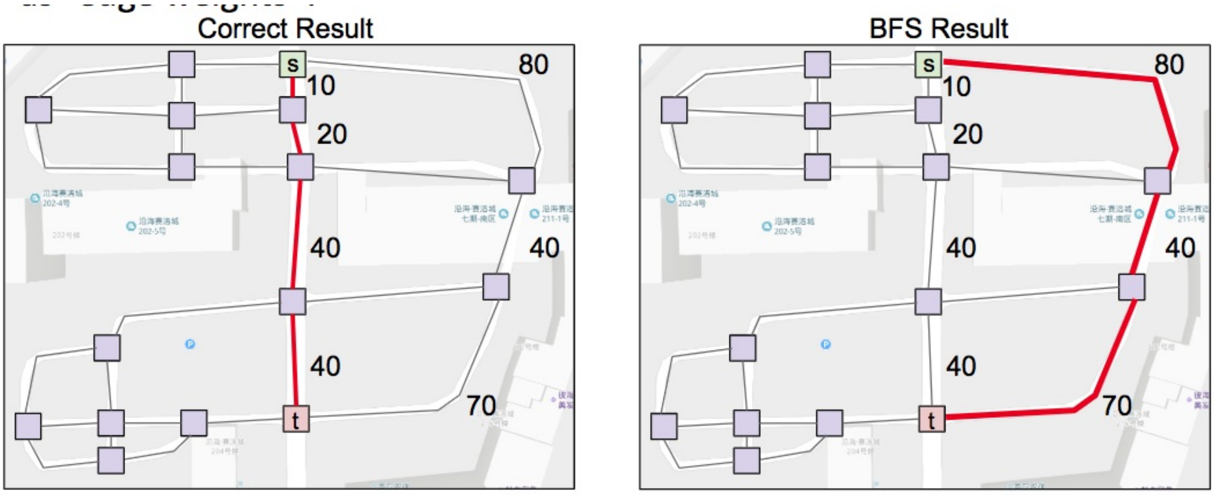
- DFS is worse for spindly graphs. Imagine a graph with 10000 nodes all spindly. We'll end up making 10000 recursive calls, which is bad for space.
- BFS is worse for "bushy" graphs, because our queue gets used a lot.

Answering the mysterious question

Did we develop an algorithm to find the **shortest** path from a given vertex to every other reachable vertex? Well, kind of. We developed an algorithm that works well on graphs with no edge labels. Here's what we did: we developed an algorithm that finds us the shortest (**where shortest means the fewest number of edges**) paths from a given source vertex.

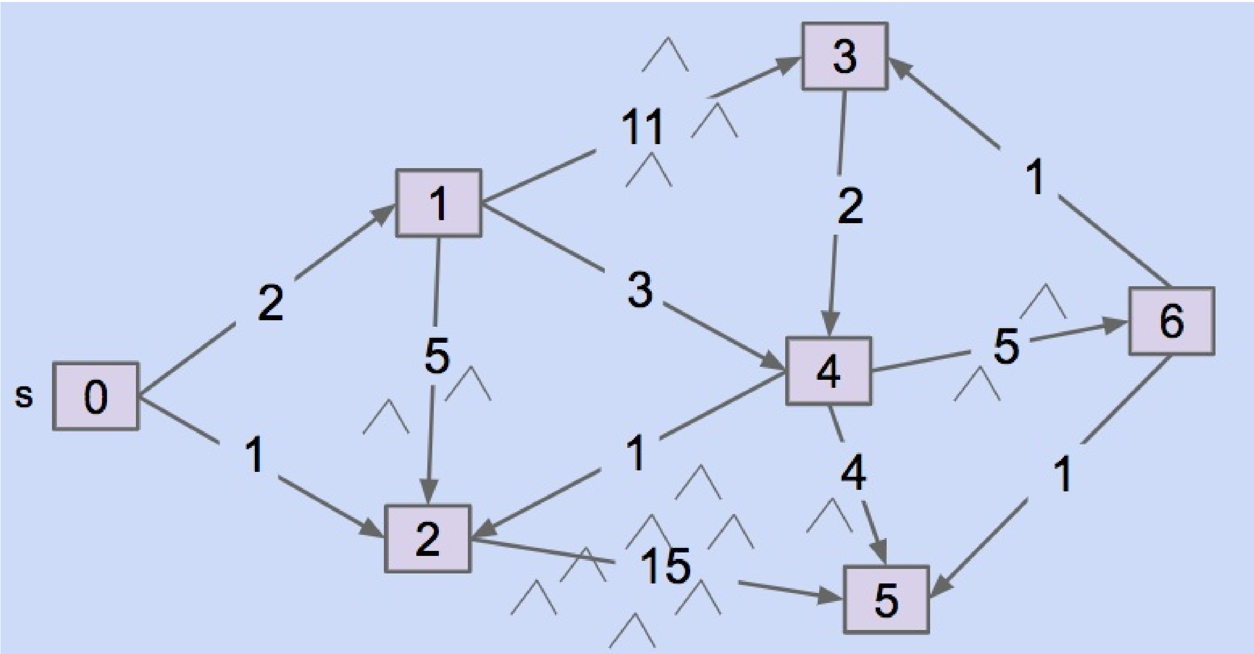
But that's not always the correct definition of shortest. Sometimes, our graph edges might have 'weights', and A-B is considered farther than A-C if the A-B edge has weight, say, 5 and the A-C edge only has weight, say, 3.

Consider the following image to see the issue.


Correct Result
BFS Result
s
t
10
20
40
40
80
40
70

Dijkstra's Algorithm

Do it by hand



Do the following two things.

1. Find a path from the vertex labeled 0 to the vertex labeled 5.
2. Find a shortest-paths tree from the vertex labeled 0. (i.e., find the shortest path from 0 to every single vertex in the graph.)
3. Try to come up with an algorithm to do this.

(The solutions to 1 and 2 are at the end of this section.)

Observations

Note that the shortest path (for a graph whose edges have weights) can have many, many edges. What we care to minimize is the sum of the weights of the edges on the selected path.

Secondly, note the fact that the shortest paths tree from a source s can be created in the following way:

- For every vertex v (which is not s) in the graph, find the shortest path from s to v.
- "Combine"/"Union" all the edges that you found above. Tada!

Thirdly, note that the "Shortest Path Tree" will **always be a tree**. Why? Well, let's think about our original solution, where we maintained an `edgeTo` array. For every node, there was exactly one "parent" in the `edgeTo` array. (Why does this imply that the "Shortest Path Tree" will be a tree? Hint: A tree has $V - 1$ edges, where V is the number of nodes in the tree.)

Dijkstra's Algorithm [[/ˈdaɪkstrə/]]

Dijkstra's algorithm takes in an input vertex s, and outputs the shortest path tree from s. How does it work?

1. Create a priority queue.
2. Add s to the priority queue with priority 0. Add all other vertices to the priority queue with priority ∞.
3. While the priority queue is not empty: pop a vertex out of the priority queue, and **relax** all of the edges going out from the vertex.

What does it mean to relax?

Suppose the vertex we just popped from the priority queue was v. We'll look at all of v's edges. Say, we're looking at edge (v, w) (the edge that goes from v to w). We're going to try and relax this edge.

What that means is: Look at your current best distance to w from the source, call it $\texttt{curBestDistToW}$. Now, look at your $\texttt{curBestDistTo}\mathbf{V} + \texttt{weight}(v, w)$ (let's call it $\texttt{potentialDistToWUsingV}$.

Is $\texttt{potentialDistToWUsingV}$ **better, i.e., smaller** than $\texttt{curBestDistToW}$? In that case, set $\texttt{curBestDistToW} = \texttt{potentialDistToWUsingV}$, and update the $\texttt{edgeTo}[w]$ to be v.

Important note: we never relax edges that point to already visited vertices.

This whole process of calculating the potential distance, checking if it's better, and potentially updating is called relaxing.

Alternate definition is captured by the following image. 

Pseudocode

```
def dijkstras(source):
    PQ.add(source, 0)
    For all other vertices, v, PQ.add(v, infinity)
    while PQ is not empty:
        p = PQ.removeSmallest()
        relax(all edges from p)
```

```
def relax(edge p,q):
   if q is visited (i.e., q is not in PQ):
       return

   if distTo[p] + weight(edge) < distTo[q]:
       distTo[q] = distTo[p] + w
       edgeTo[q] = p
       PQ.changePriority(q, distTo[q])
```

Guarantees

As long as the edges are all non-negative, Dijkstra's is guaranteed to be optimal.

Proofs and Intuitions

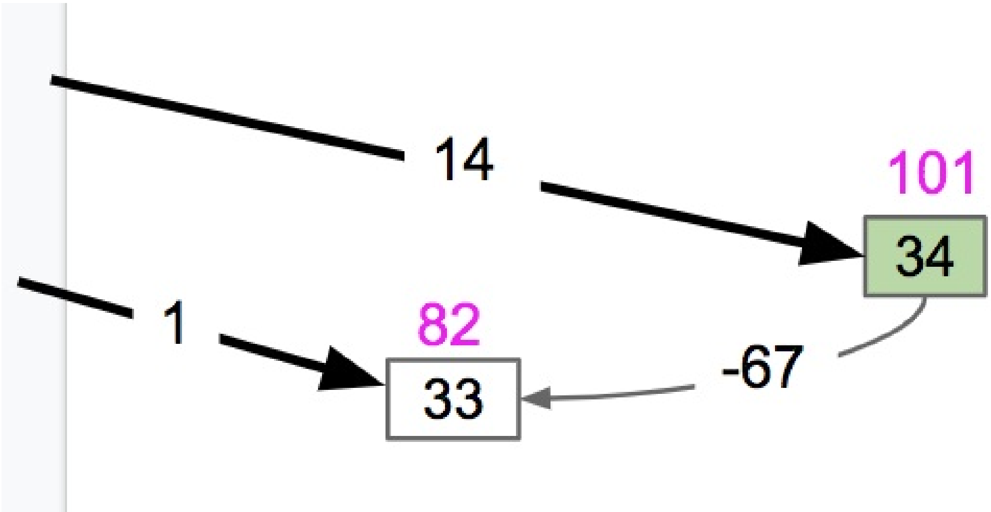
Assume all edges are non-negative.

- At start, distTo[source] = 0. This is optimal.
- After relaxing all edges from source, let vertex v_1 be the vertex with the minimum weight (i.e., the one that's closest to the source.) **Claim: distTo[v_1] is optimal, i.e., whatever the value of distTo[v_1] is at this point is the shortest distance from s to v_1**. Why?
  - Let's try to see why this **MUST** be the case.
  - Suppose that it isn't the case. Then that means that there is some other path from s to v_1 which is shorter than the direct path (s, v_1). Ok, so let's consider this hypothetical cool shorter path... it would have to look like $(s, v_a, v_b, \ldots, v_1)$. But... (s, v_a) is **already** bigger than (s, v_1). (Note that this is true because v_1 is the vertex that is closest to s from above.) So how can such a path exist which is actually shorter? It can't!
- Now, the next vertex to be popped will be v_1. (Why? Note that it currently has the lowest priority in the PQ!)
- So now, we can make this same argument for v_1 and all the relaxation it does. (This is

called "proof by induction". It's kind of like recursion for proofs.) And that's it; we're done.

Negative Edges?

Things can go pretty badly when negative edges come into the picture. Consider the following image.



Suppose you're at that vertex labeled 34. Now you're going to try to relax all your edges. You have only one outgoing edge from yourself to 33 with weight -67. Ah, but note: vertex 33 is already visited (it's marked with white.) So... we don't relax it. (Recall the pseudocode for the relax method.)

Now we go home thinking that the shortest distance to 33 is 82 (marked in pink.) But really, we should have taken the path **through** 34 because that would have given us a distance of $101 - 67 = 34$. Oops.

Dijkstra's algorithm is not guaranteed to be correct for negative edges. It might work... but it isn't guaranteed to work.

Try this out: suppose that your graph has negative edges, but all the negative edges only go out of the source vertex s that you were passed in. Does Dijkstra's work? Why / Why not?

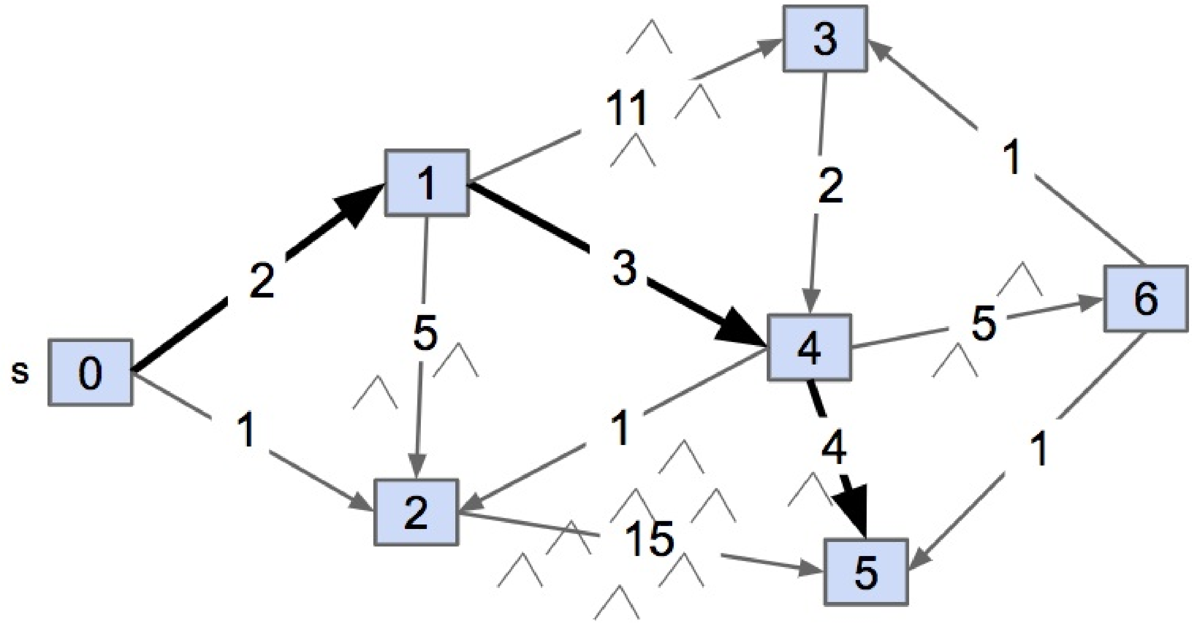
A noteworthy invariant

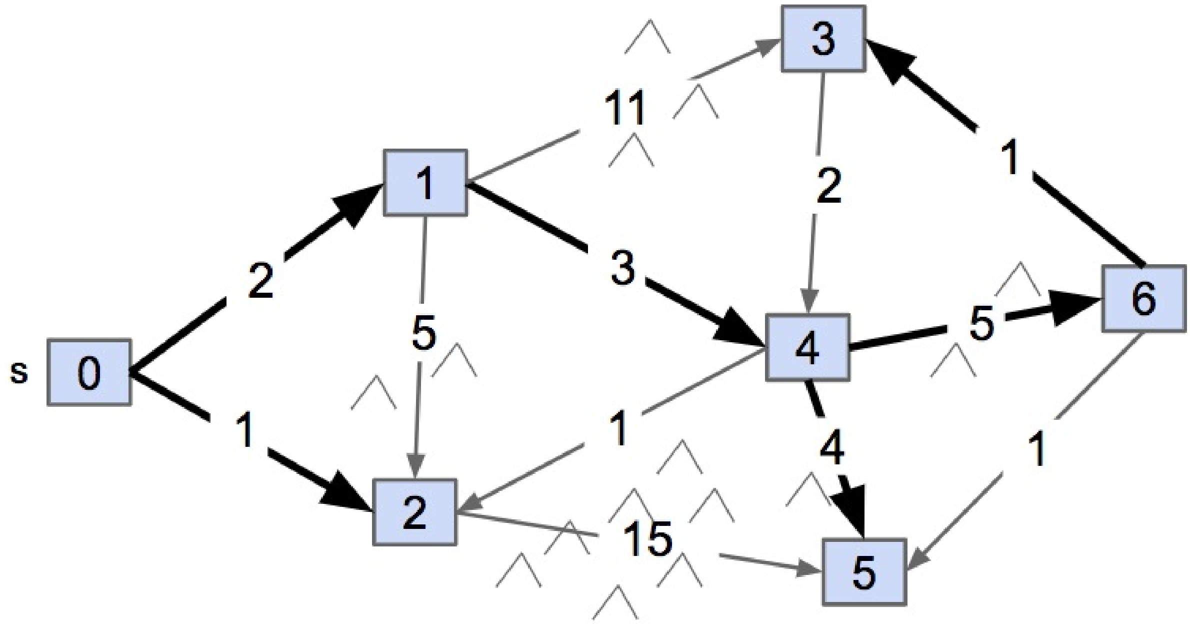
Observe that once a vertex is popped off the priority queue, it is never re-added. Its distance is never re-updated. So, in other words, once a vertex is popped from the priority queue, we **know** the true shortest distance to that vertex from the source.

One nice consequence of this fact is "short-circuiting". Suppose... that I didn't care about the shortest-paths tree, but just wanted to find the shortest path from some source to some other target. Suppose that you wanted to take, like, the cities of the world on a graph, and find the shortest path from Berkeley to Oakland. Running `dijkstra(Berkeley)` will mean that you can't actually stop this powerful beast of an algorithm... you have to let it run... till it finds the shortest path to LA, and Houston, and New York City, and everywhere possible!

Well. Once `Oakland` is popped off the priority queue in the algorithm, we can just stop. We can just return the distance and the path we have at that point, and it will be correct. So **sometimes** `dijkstra` takes in not only a source, but also a target. This is for the purposes of short-circuiting.

The promised solutions





A*

We ended the section on Dijkstra's by discussing a possible way to make Dijkstra's short-circuit and just stop once it hits a given target. Is this good enough?

To answer the above question, we need to sit down and think about how dijkstra's really works. Pictorially, Dijkstra's starts at the source node (imagine the source node being the center of a circle.) And Dijkstra's algorithm now makes concentric circles around this point, in increasing radii, and 'sweeps' these circles, capturing points.

So... the first node Dijkstra's visits is the city closest to the source, then the city next-closest, then the city next-closest, and so on. This sounds like a good idea. What Dijkstra's is doing is first visiting all the cities that are 1-unit distance away, then 2 unit-distance away, and so on. In concentric circles.

Now imagine the following: on a map of the US, start somewhere in the center, say, Denver. Now I want you to find me a path to New York using Dijkstra's. You'll end up traversing nodes in 'closest concentric circle' order.

You'll make a small circle first, just around Denver, visiting all the cities in that circle. Eventually, your circles will get bigger, and you'll make a circle that passes through Las Vegas (and would have visited, by now, all the other cities that fall within the circle.) Then, your circle will be big enough to engulf Los Angeles and Dallas... but you're nowhere close to New York yet. All this effort, all these circles, but still... so far from the target. Short-circuiting helps, but only if you actually hit the target node fast.

If only there existed a way to use your prior knowledge: the fact that new-york was eastwards, so you could "hint" your algorithm to prefer nodes that are on the east instead of those that are on the west.

Introducing: A Star

No, not the sun. It's an algorithm called A*.

Observe the following: Dijkstra's is a "true" (i.e., not an estimate) measure of the distance **to** a node from the source. So, say, you visit a city in Illinois and your source was Denver, then by that time, you have a true measure of the distance **to** Denver. What we're missing is: some janky, rough estimate of the distance from a node **to** the target node, New York. That would complete the picture. Because then, if you sum these two things up (the measure from the source to the node + the estimate from the node to the target), you get (an estimate from the source to the target.) Of course, the better your original estimate from the node to the target, the better your estimate from the source to the target, the better your A* algorithm runs.

So, let's modify our Dijkstra's algorithm slightly. In Dijkstra's, we used $\texttt{bestKnownDistToV}$ as the priority in our algorithm. This time, we'll use $\texttt{bestKnownDistToV} + \texttt{estimateFromVToGoal}$ as our heuristic.

Here is a demo!

Chicken And Egg

We have a problem. How do we know what the estimate is? I mean, the estimate itself is a **distance**, and we're using A* to **find** the distance from some node to some other node.

It seems like we're in an instance of the classic chicken and egg problem. "What came first? The chicken or the egg?" Aside, FYI, one reddit user had an idea about this.

Well, it's called an estimate because it's exactly that. We use A* to get the **true** shortest path from a source to a target, but the estimate is something we approximate. Coming up with good estimates is hard sometimes.

But to give you an example in our Denver - New York case. What we might do is just look up the GPS Coordinates of these cities, and calculate the straight line distance between those somehow. Of course, this wouldn't be correct because there's probably no straight line that one could take from Denver to NYC, but it's a fairly good estimate!

Bad Heuristics

Suppose that the shortest path from Denver to New York goes through some city C. Suppose that my GPS is broken, and so I think that this city C is infinity far away from everything, and I set the estimated distance to C from every other node in the graph to ∞.

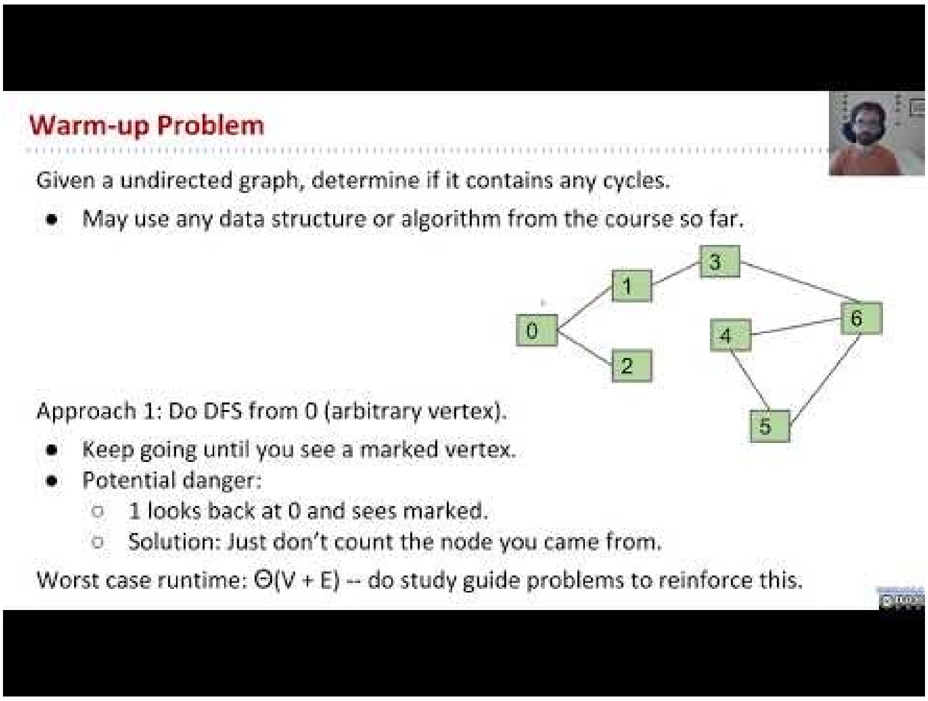
What will happen? Well A* will basically never want to visit this city. (Remember what our priorities are in the priority queue; for this city, the priority will always be ∞, even if I visit the immediate neighbors of this city. The estimated distances from the immediate neighbors of this city to this city were set to ∞ after all.)

So... now what? We lose. A* breaks. We get the wrong answer back. Oops.

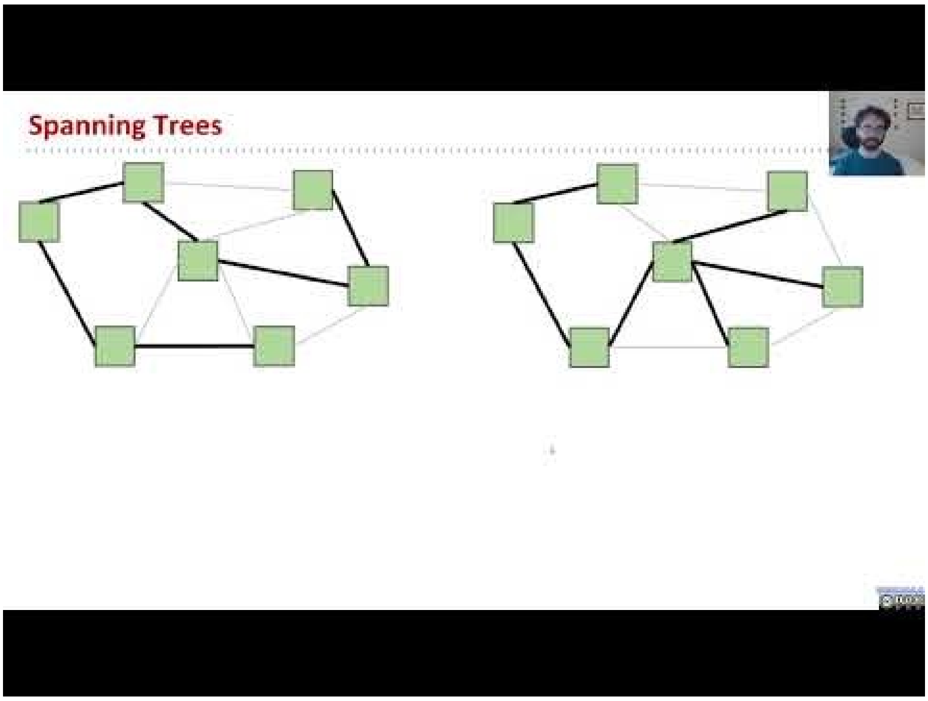
The takeaway here is that heuristics need to be good. There are two definitions required for goodness.

1. Admissibility. heuristic(v, target) $\leq$ trueDistance(v, target). (Think about the problem above. The true distance from the neighbor of C to C wasn't infinity, it was much, much smaller. But our heuristic said it was ∞, so we broke this rule.)
2. Consistency. For each neighbor v of w:
    - heuristic(v, target) $\leq$ dist(v, w) + heuristic(w, target)
    - where dist(v, w) is the weight of the edge from v to w.

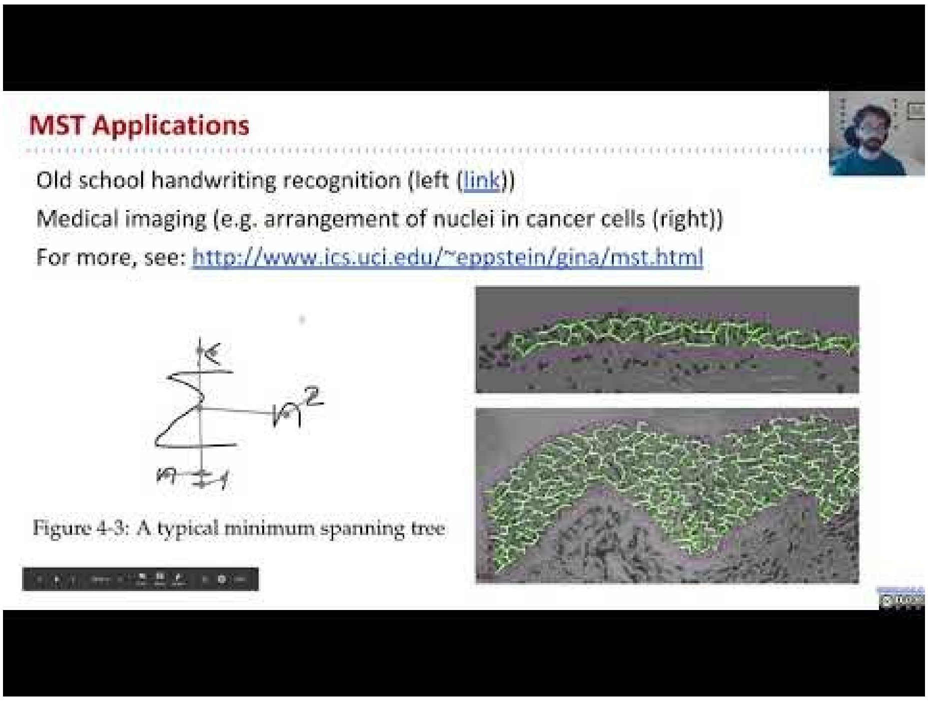
Minimum Spanning Trees



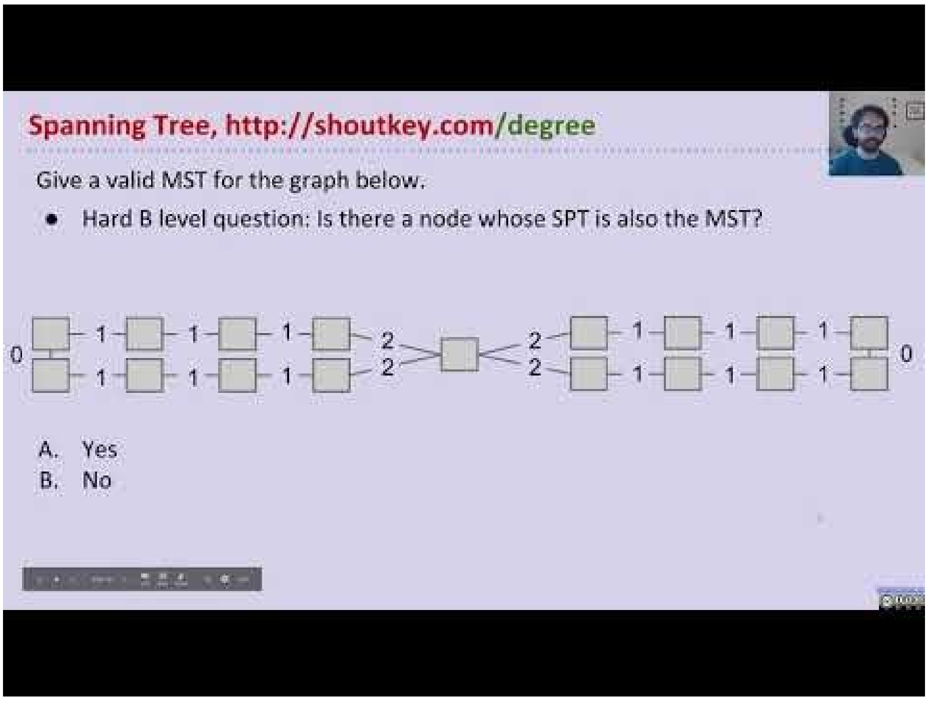
[Video link](#)



[Video link](#)



Video link



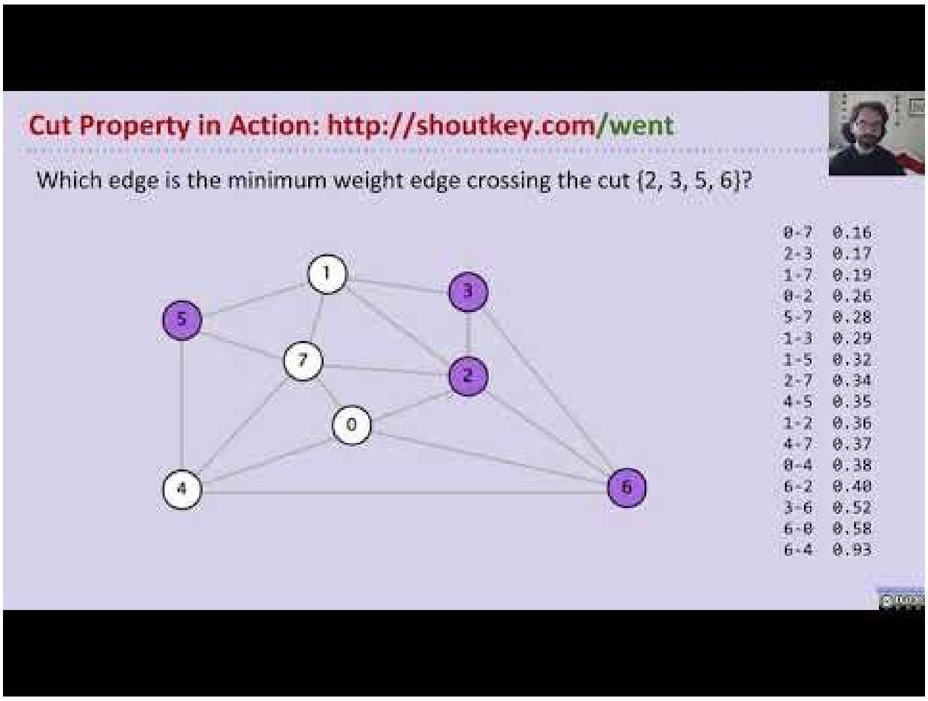
Video link

A minimum spanning tree (MST) is the lightest set of edges in a graph possible such that all the vertices are connected. Because it is a tree, it must be connected and acyclic. And it is called "spanning" since all vertices are included.

In this chapter, we will look at two algorithms that will help us find a MST from a graph.

Before we do that, let's introduce ourselves to the Cut Property, which is a tool that is useful for finding MSTs.

Cut Property

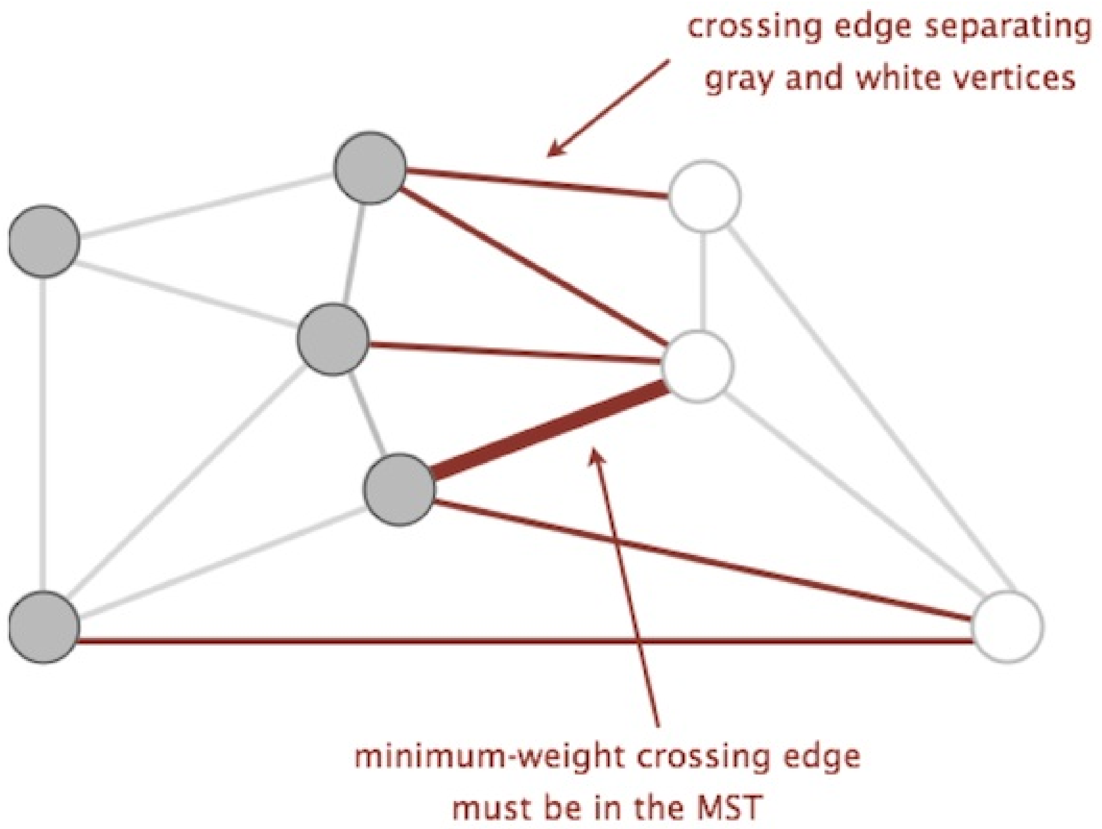


Video link

We can define a **cut** as an assignment of a graph's nodes to two non-empty sets (i.e. we assign every node to either set number one or set number two).

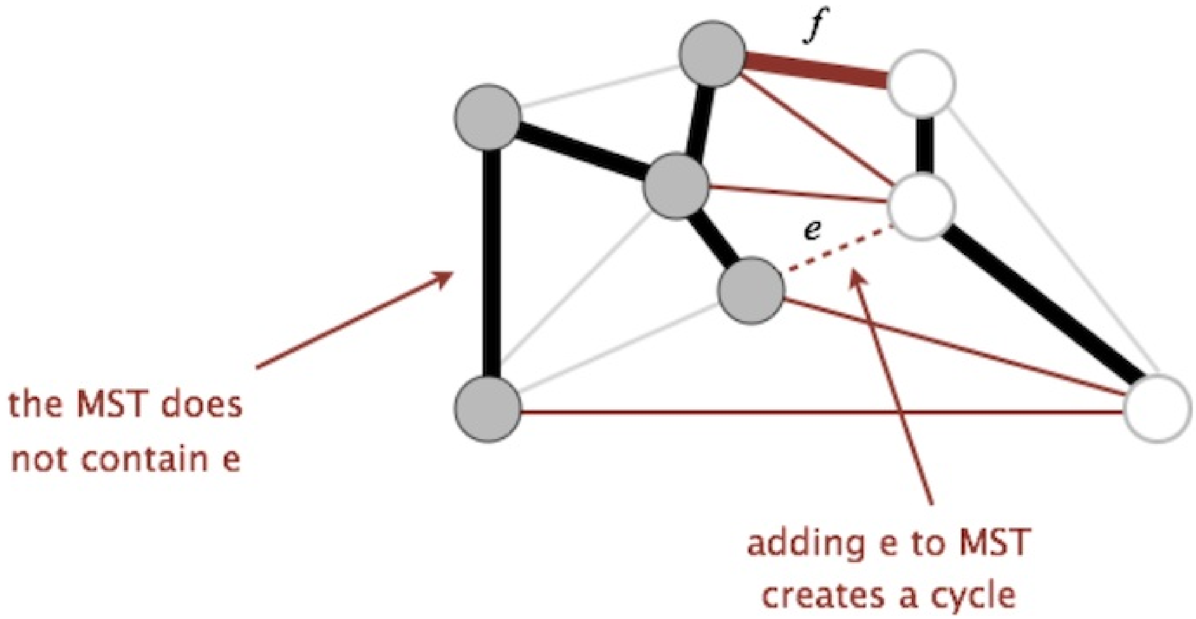
We can define a **crossing edge** as an edge which connects a node from one set to a node from the other set.

With these two definitions, we can understand the **Cut Property**; given any cut, the minimum weight crossing edge is in the MST.

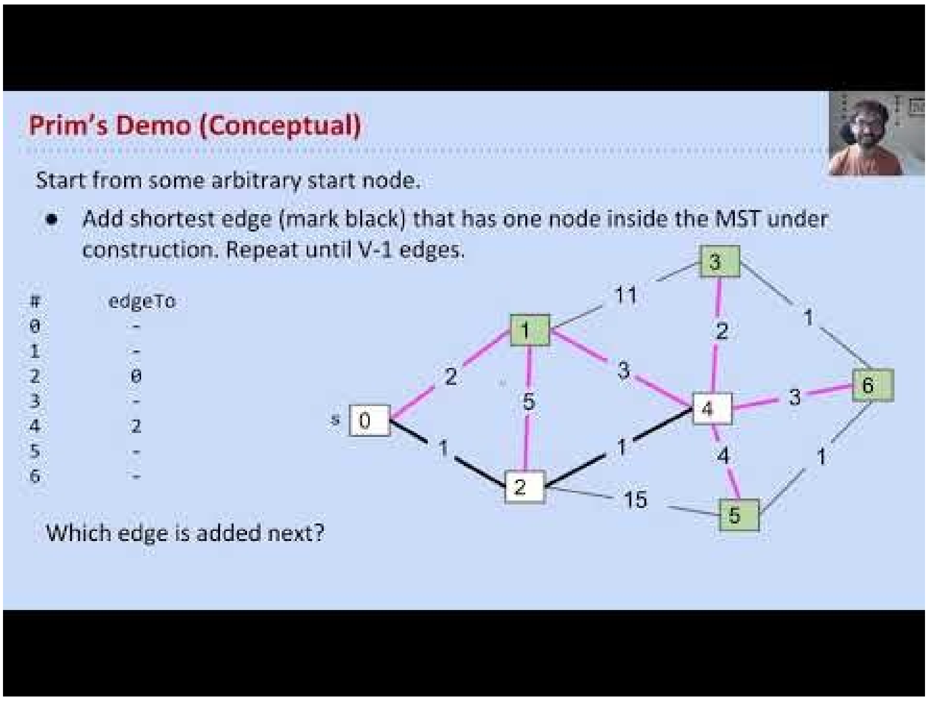


The proof for the cut property is as follows: Suppose (for the sake of contradiction) that the minimum crossing edge *e* were not in the MST. Since it is not a part of the MST, if we add that edge, a cycle will be created. Because there is a cycle, this implies that some other edge f must also be a crossing edge (for a cycle, if *e* crosses from one set to another, there must be another edge that crosses back over to the first set). Thus, we can remove *f* and keep *e*, and this will give us a lower weight spanning tree. But this is a contradiction because we supposedly started with a MST, but now we have a collection of edges which is a spanning tree but that weighs less, thus the original MST was not actually minimal. As a result, the cut property must hold.

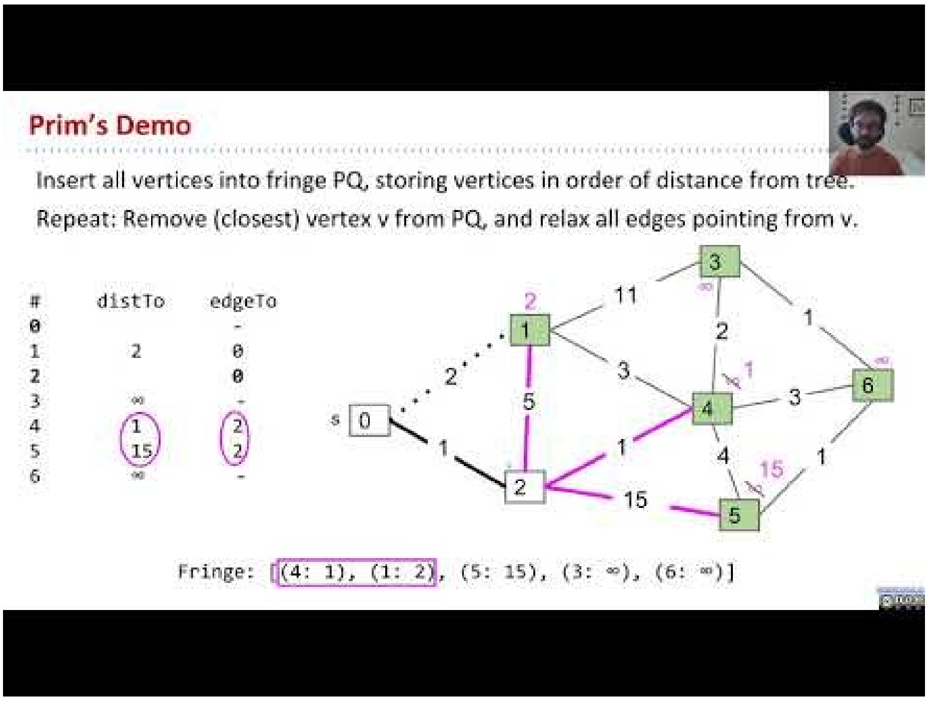
Here is a diagram illustrating some of the arguments of the above proof:



Prim's Algorithm



Video link

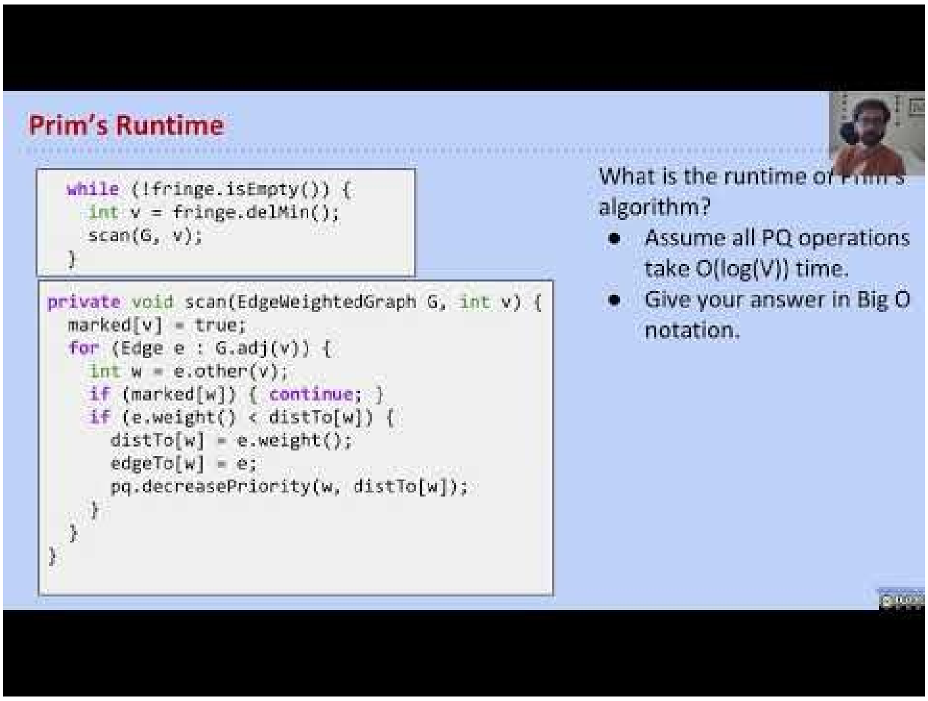


Video link

This is one algorithm to find a MST from a graph. It is as follows:

```
1. Start from some arbitrary start node.
2. Repeatedly add the shortest edge that has one node inside the MST under constructio
n.
3. Repeat until there are V-1 edges.
```

Prim's algorithm works because at all stages of the algorithm, if we take all the nodes that are part of our MST under construction as one set, and all other nodes as a second set, then this algorithm always adds the lightest edge that crosses this cut, which is necessarily part of the final MST by the Cut Property.

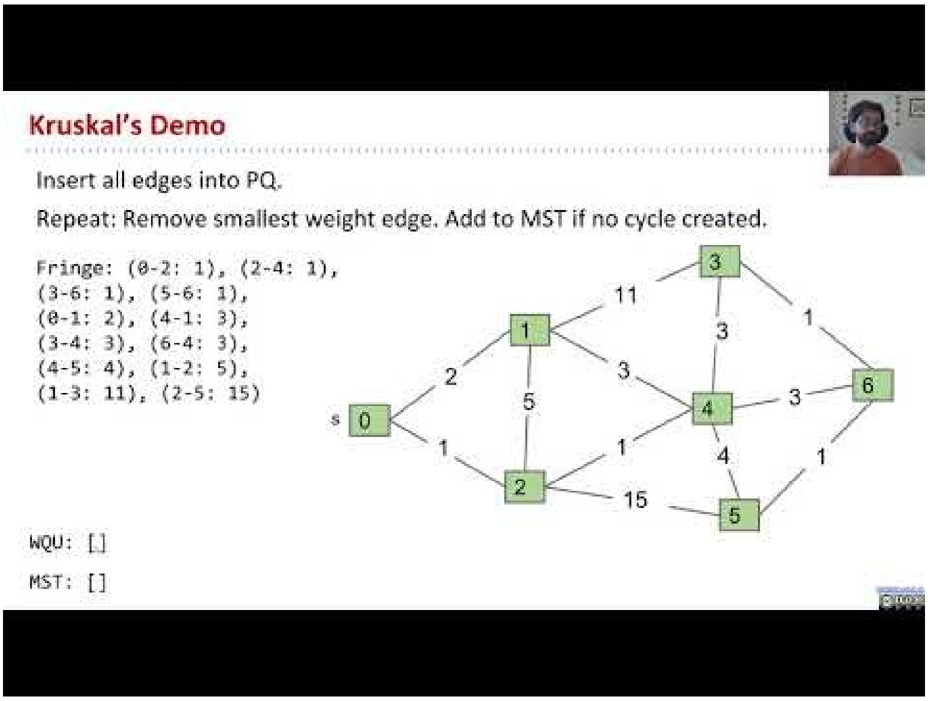


[Video link](#)

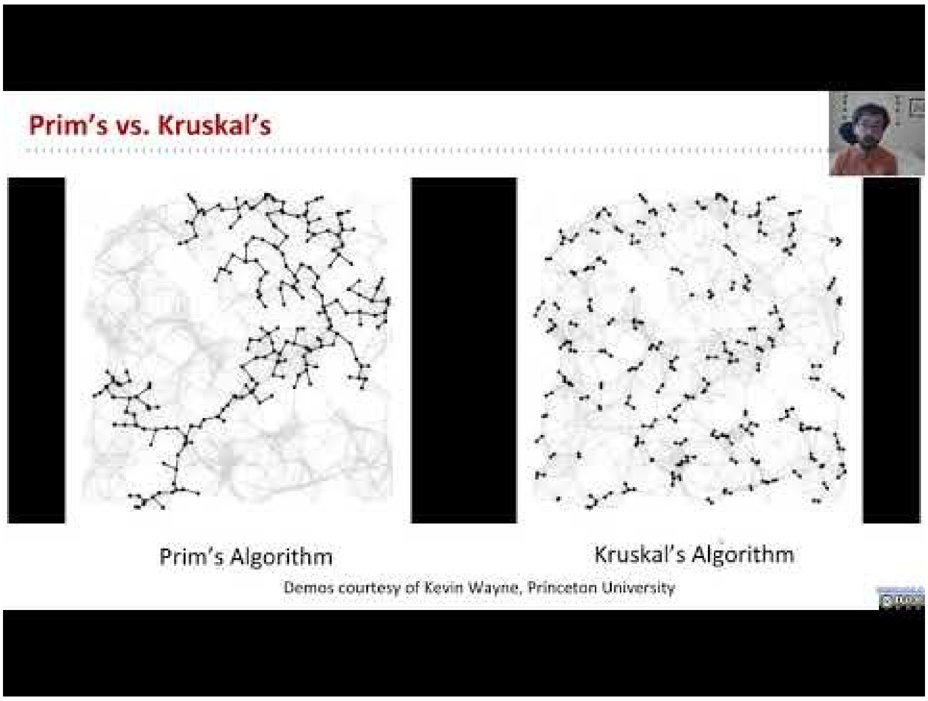
Essentially, this algorithm runs via the same mechanism as Dijkstra's algorithm, but while Dijkstra's considers candidate nodes by their distance from the source node, Prim's looks at each candidate node's distance from the MST under construction.

Thus, the runtime of Prim's if done using the same mechanism as Dijkstra's, would be the same as Dijkstra's, which is $O((|V| + |E|) \log |V|)$. Remember, this is because we need to add to a priority queue fringe once for every edge we have, and we need to dequeue from it once for every vertex we have.

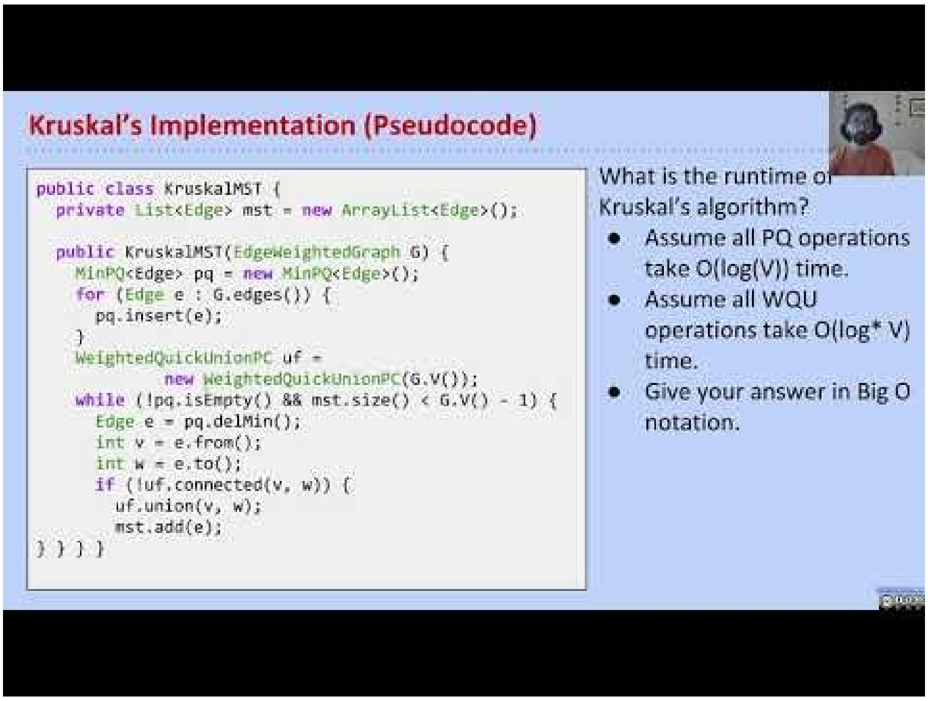
Kruskal's Algorithm



[Video link]



[Video link]



[Video link]

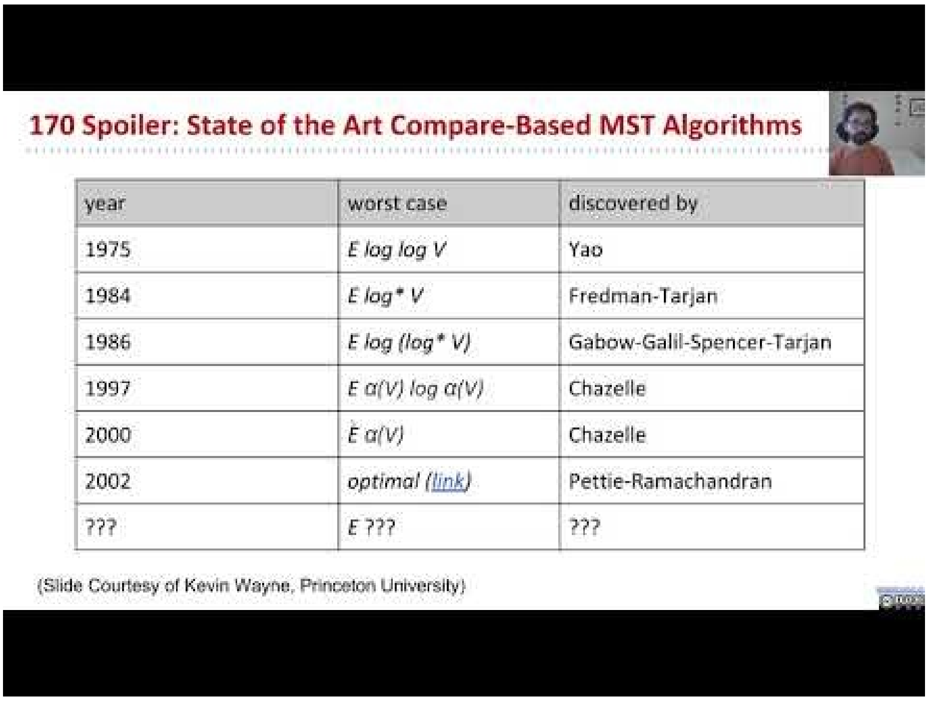
This is another algorithm that can be used to find a MST from a graph. The MST returned by Kruskal's might not be the same one returned by Prim's, but both algorithms will always return a MST; since both are minimal (optimal), they will both give valid optimal answers (they are tied as equally minimal / same total weight, and this is as low as it can be).

The algorithm is as follows:

```
1. Sort all the edges from lightest to heaviest.
2. Taking one edge at a time (in sorted order), add it to our MST under construction i
f doing so does not introduce a cycle.
3. Repeat until there are {% math %}V-1{% endmath %} edges.
```

Kruskal's algorithm works because any edge we add will be connecting one node, which we can say is part of one set, and a second node, which we can say is part of a second set. This edge we add is not part of a cycle, because we are only adding an edge if it does not introduce a cycle. Further, we are looking at edge candidates in order from lightest to heaviest. Therefore, this edge we are adding must be the lightest edge across this cut (if there was a lighter edge that would be across this cut, it would have been added before this, and adding this one would cause a cycle to appear). Therefore, this algorithm works by the Cut Property as well.

Kruskal's runs in $O(|E| \log |E|)$ time because the bottleneck of the algorithm is sorting all of the edges to start (for example, we can use heap sort, in which we insert all of the edges into a heap and remove the min one at a time). If we are given pre-sorted edges and don't have to pay for that, then the runtime is $O(|E| \log^* |V|)$. This is because with every edge we propose to add, we need to check whether it will introduce a cycle or not. One way we know how to do this is by using Weighted Quick Union with Path Compression; this will efficiently tell us whether two nodes are connected (unioned) together already or not. This will cost $|E|$ calls on isConnected, which costs $O(\log^* |V|)$ each, where $\log^*$ is the Ackermann function.



Video link

Topological Sort

We have covered a tremendous amount of material so far. Programming practices, using an IDE, designing data structures, asymptotic analysis, implementing a ton of different abstract data types (e.g. using a BST, Trie, or HashTable to implement a map, heaps to implement a Priority Queue), and finally algorithms on graphs.

> Why is this knowledge useful?

You may have heard people say that CS 61B teaches much of what you need to solve standard interview questions at tech companies - but why do companies seek candidates with this specific knowledge?

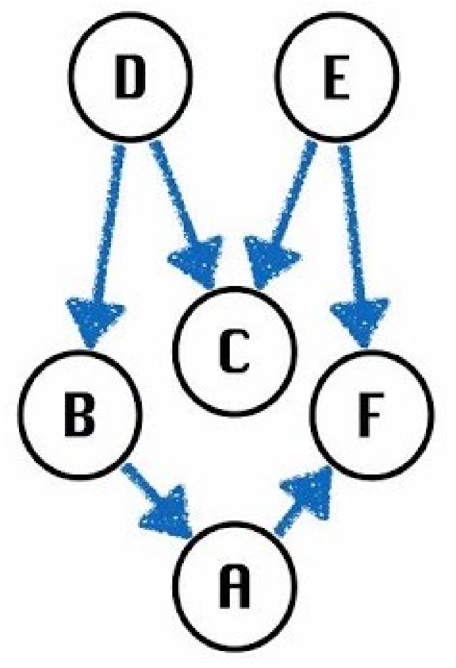
One major reason is that many real world problems can be formulated in such a way that they're solvable with the data structures and algorithms we've learned. This chapter is about working through some tricky problems using the tools we have already learned.

Topological Sorting

Suppose we have a collection of different tasks or activities, some of which must happen before another. How do we find sort the tasks such that for each task v, the tasks that happen before v come earlier in our sort?

We can first view our collection of tasks as a graph in which each node represents a task. An edge $v \rightarrow w$ indicates that v must happen before w. Now our original problem is reduced to finding a **topological sort**.

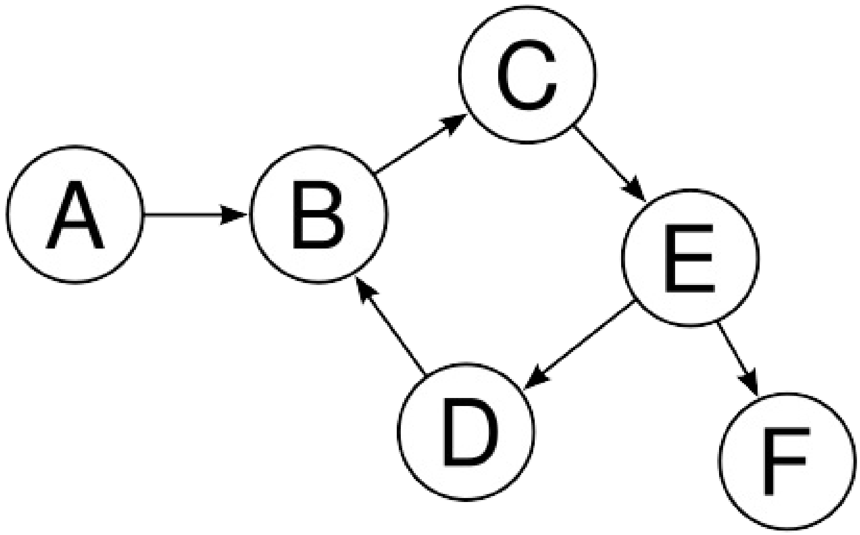
> **Topological Sort:** an ordering of a graph's vertices such that for every directed edge $u \rightarrow v$, u comes before v in the ordering.



Question 1.1: What are some valid topological orderings of the above graph?

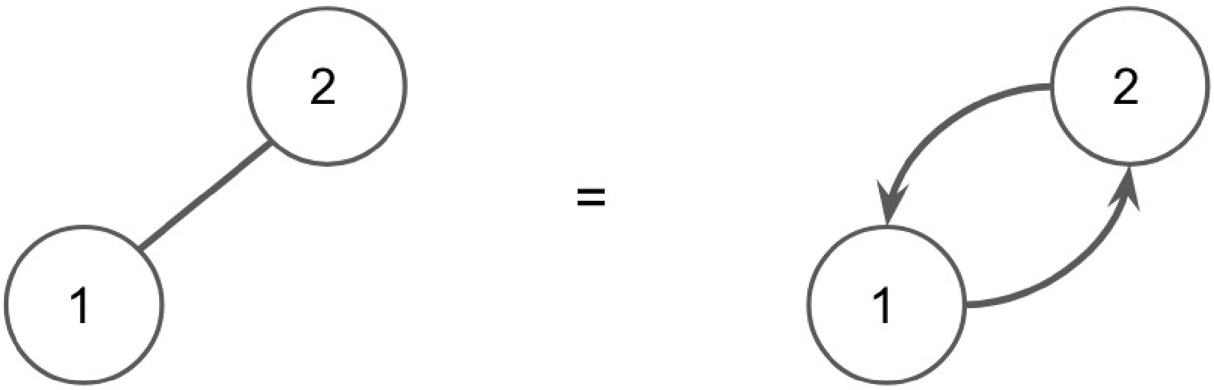
Answer: Valid orderings include: $[D, B, A, E, C, F]$, $[E, D, C, B, A, F]$.

An important note is that it only makes sense to topological sort certain types of graphs. To see this, consider the following graph:



What is a valid topological sorting of this?

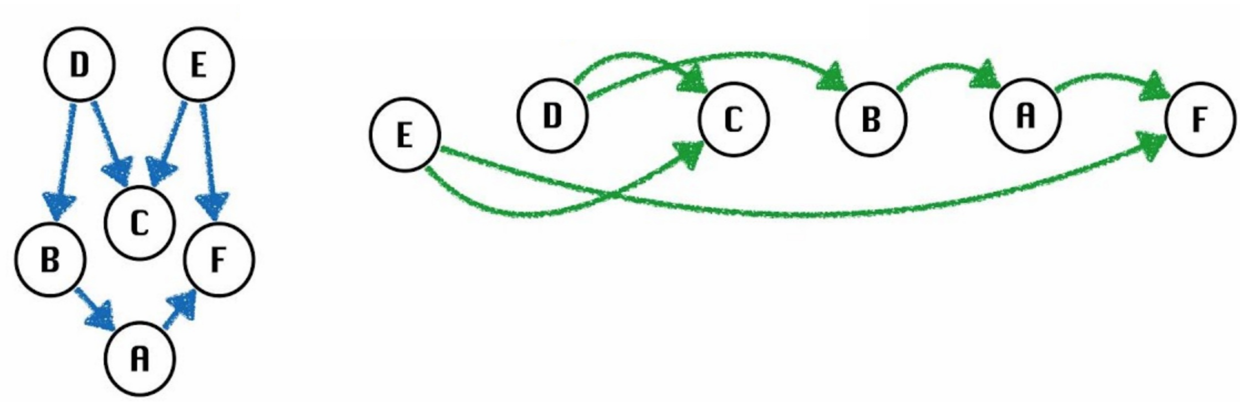
There isn't one! D comes before B but B comes before C, E, D. Since we have a cycle, topological sort is not defined. We also can't topologically sort an undirected graph since each edge in an undirected graph creates a cycle.



So topological sorts only apply to **directed, acyclic (no cycles) graphs** - or **DAG**s.

> **Topological Sort:** an ordering of a **DAG**'s vertices such that for every directed edge $u \to v$, u comes before v in the ordering.

For any topological ordering, you can redraw the graph so that the vertices are all in one line. Thus, topological sort is sometimes called a **linearization** of the graph. For example, here's the earlier example linearized for one of the topological orderings.



Notice that the topological sort for the above DAG has to start with either D or E and must end with F or C. For this reason, D and E are called *sources*, and F and C are called *sinks*.

Topological Sort Algorithm

How can we find a topological sort? Take a moment to think of existing graph algorithms you already know could be helpful in solving this problem.

Topological Sort Algorithm:

- Perform a DFS traversal from every vertex in the graph, **not** clearing markings in between traversals.
- Record DFS postorder along the way.
- Topological ordering is the reverse of the postorder.

Why it works: Each vertex v gets added to the end of the postorder list only after considering **all** descendants of v. Thus, when any v is added to the postorder list, all its descendants are already on the list. Thus reversing this list gives a topological ordering.

Since we're simply using DFS, the runtime of this is $O(V + E)$ where V and E are the number of nodes and edges in the graph respectively.

Pseudocode

```
topological(DAG):
    initialize marked array
    initialize postOrder list
    for all vertices in DAG:
        if vertex is not marked:
            dfs(vertex, marked, postOrder)
    return postOrder reversed

dfs(vertex, marked, postOrder):
    marked[vertex] = true
    for neighbor of vertex:
        dfs(neighbor, marked, postOrder)
    postOrder.add(vertex)
```

(Out of scope) Extra question: How could we implement topological sort using BFS? *Hint 1: We'd definitely need to store some extra information. Hint 2: Think about keeping track of the in-degrees of each vertex.*

Solution:

1. Calculate in-degree of all vertices.
2. Pick any vertex v which has in-degree of 0.
3. Add v to our topological sort list. Remove the vertex v and all edges coming out of it. Decrement in-degrees of all neighbors of vertex v by 1.
4. Repeat steps 2 and 3 until all vertices are removed.

How can we accomplish Step 2 efficiently? We can use a min Priority Queue of vertices with priority equal to the in-degrees.

Review

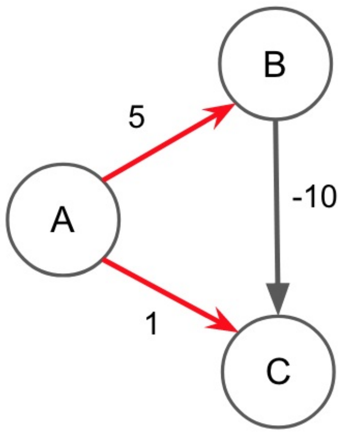
- Topological sorts are a way of linearizing **Directed, Acyclic Graphs (DAGs)**.
- We can find a topological sort of any DAG in $O(V + E)$ time using DFS (or BFS).

Shortest Paths on DAGs

Recall from the previous section that **DAGs** are **directed, acyclic graphs**. If we wanted to find the shortest path on DAGs we could use Dijkstra's. However, with DAGs there's a simple shortest path algorithm which also handles negative edge weights!

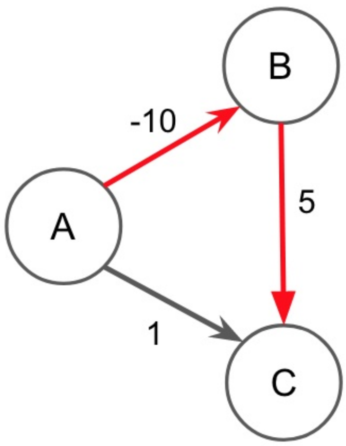
Dijkstra's Negative Edge Weight Failure

Recall that Dijkstra's can fail if negative edges exist because it relies on the assumption that once we visit an edge, we've found the shortest path to that edge. But if negative edge weights can exist ahead of where we can see, then this assumption fails. Consider the following example:



Starting from A, Dijkstra's will visit C first, then B (never even considering the edge $B \rightarrow C$).[1]

Of course, negative edge weights do not mean Dijkstra's is guaranteed to fail. Dijkstra's succeeds with the following example:



> 1. This technically depends on your implementation of Dijkstra's. If we ensure that the relaxation step only considers neighbors that are still in the queue (haven't been visited yet), then it is true that $B \to C$ will never be considered. If you don't have that check, then technically when we pop the last node (B) from the queue, we'd consider B's neighbors and update C which gives us the right answer for this specific example. However, in that case one could argue that the graph breaks the Dijkstra invariant and thus Dijkstra has 'failed'. Note, the Dijkstra invariant: *once a node is deleted from the queue (visited) then you've found the shortest path to that node.* ↩

Shortest Path Algorithm for DAGs

Visit vertices in topological order:

- On each visit, relax all outgoing edges

Recall the definition for relaxing an edge $u \to v$ with weight w:

```
if distTo[u] + w < distTo[v]:
    distTo[v] = distTo[u] + w
    edgeTo[v] = u
```

Since we visit vertices in topological order, a vertex is visited only when all possible info about it has been considered. This means that if negative edge weights exist along a path to v, then those have been taken into account by the time we get to v!

Finding a topological sort takes $O(V+E)$ time while relaxation from each vertex also takes $O(V+E)$ time in total. Thus, the overall runtime is $O(V+E)$. Recall that Dijkstra's takes $O((V+E)\log V)$ time because of our min-heap operations.

What if we want to solve the shortest path problem on graphs that aren't DAGs and also may have negative edges? An extension of Dijkstra's called Bellman Ford can suit your needs, though it is out of scope for this course.

Longest Paths

In General

Consider the problem of finding the longest path from a start vertex to every other vertex. The path must be simple (contain no cycles).

It turns out that best known algorithm is exponential (impractically inefficient).

Negating all the edge weights and finding the shortest path leaves us in a tricky situation because then we could have negative cycles and we could go around and around them indefinitely.

Longest Paths on DAGs

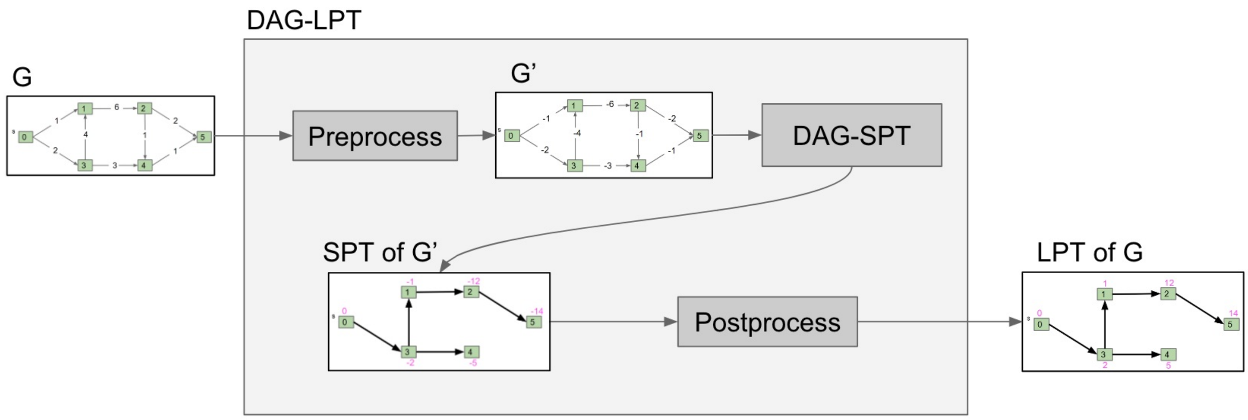
But what if we are dealing with DAGs? In that case, we have no cycles so we can do as suggested above:

1. Form a new copy of the graph, called G', with all edge weights negated (signs flipped).
2. Run DAG shortest paths on G' yielding result X
3. Flip the signs of all values in X.distTo. X.edgeTo is already correct.

Alternatively, we could modify the DAG shortest path algorithm from the previous section to choose the larger distTo when relaxing an edge. While this would make more sense in practice, the benefit of thinking of the first approach is that wree a able to use an existing algorithm as a "black box" to solve a new problem. We'll learn more about this kind of problem solving in the next section: Reductions.

Reductions

Recall in previous section that to solve one problem (longest paths), we created a new graph G' and fed it into a different algorithm and then interpreted the result.



This process is known as **reduction**. Since DAG-SPT can be used to solve DAG-LPT, we say that "DAG-LPT reduces to DAG-SPT."

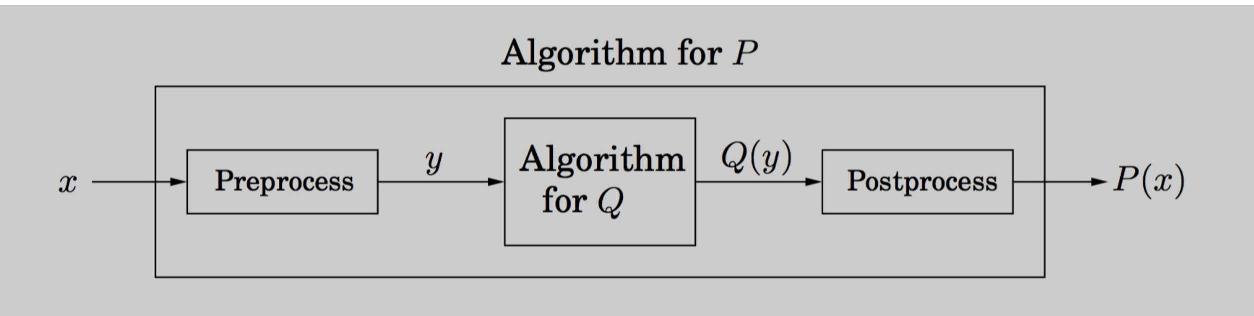
In other words, the problem of DAG-LPT can be reduced to the problem of DAG-SPT.

A problem like DAG-LPT can potentially be reduced to multiple other problems. As a real-world analogy, consider climbing a hill. There are many ways we can solve the problem of "climbing a hill."

- "Climbing a hill" reduces to "riding a ski lift"
- "Climbing a hill" reduces to "being shot out of a cannon"
- "Climbing a hill" reduces to "riding a bike up the hill"

Formally, **if any subroutine for task Q can be used to solve P, we say P reduces to Q.**

This definition is visualized below:



Note that this is simply a generalization of the first graphic on this page. P reduces to Q since Q is used to solve P. This works by preprocessing the input x into y, running the algorithm Q on y, and postprocessing the output into a solution for P. This is what we did for reducing DAG-LPT to DAG-SPT.

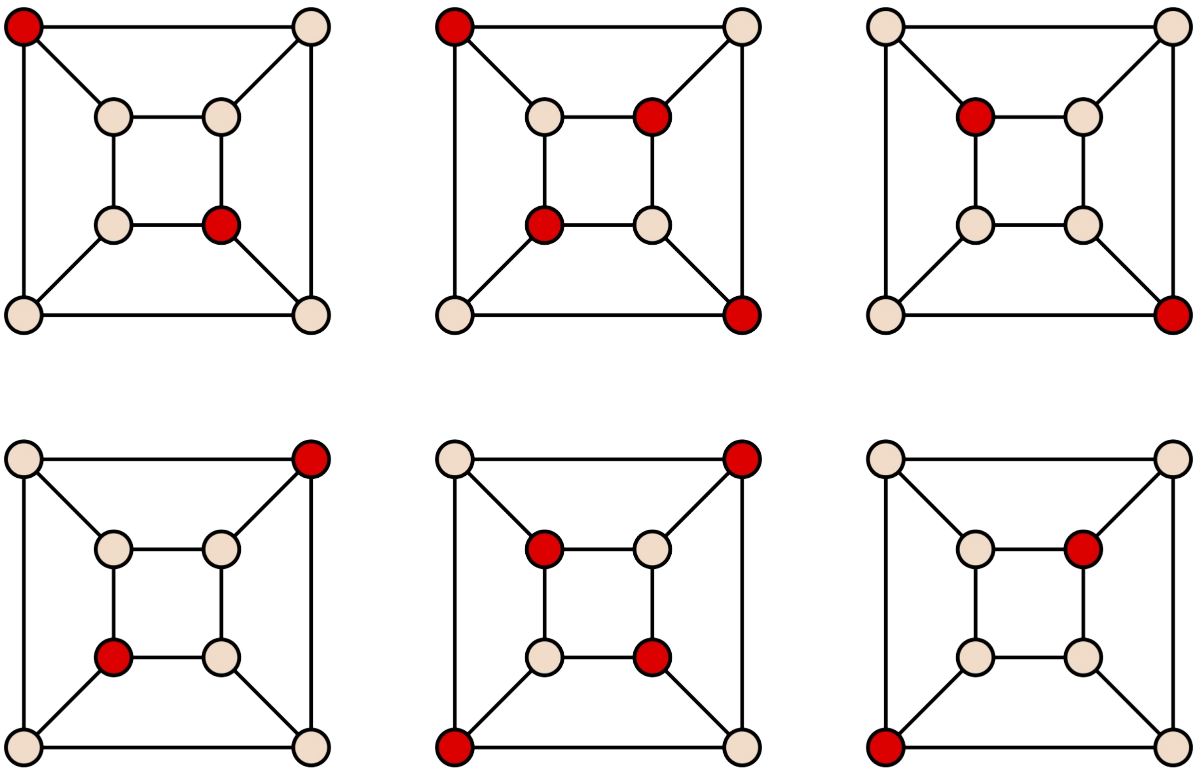
Example

Here we'll show how one problem can reduce to a seemingly unrelated different problem. First, the two problems:

Independent Set Problem

An independent set is a set of vertices in which no two vertices are adjacent.

The Independent Set Problem: Does there exist an independent set of size k? In other words, can we color k vertices red, such that none touch?



Example of independent sets solutions for k=2 and k=4

3SAT Problem

What values of `x1`, `x2`, `x3`, `x4` satisfy the following boolean formula: `(x1 || x2 || !x3) && (x1 || !x1 || x1) && (x2 || x3 || x4)`?

The 3SAT Problem: Given a boolean formula, does there exist a truth value for boolean variables that obeys a set of 3-variable disjunctive constraints?

Terminology clarification:

- Constraints are True/False values.
- **Disjunctive** means separated by OR. 3SAT has a set of "clauses," each made up of 3 literals with each literal separated by an OR. For example, the first clause above is `(x1 || x2 || !x3)`.
- In the 3SAT problem we must satisfy the entire set of clauses (combine each clause with AND).

e.g.: `(x1 || x2 || !x3) && (x1 || !x1 || x1) && (x2 || x3 || x4)` Yes, a solution for x1, x2, x3, x4 exists Solution: x1 = true, x2 = true, x3 = true, x4 = false

Reduction

CLAIM: 3SAT reduces to Independent Set

- Recall this means we claim we can solve 3SAT by using the Independent Set algorithm!

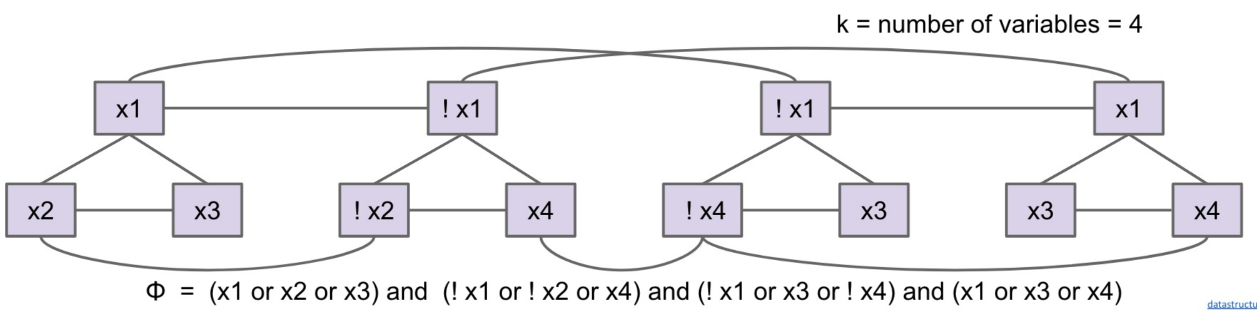
PROOF: To prove the reduction, we need to argue that we can:

1. Preprocess a given 3SAT problem
2. Solve it with Independent Set
3. Postprocess the output of part 2 into a solution to the original 3SAT problem.

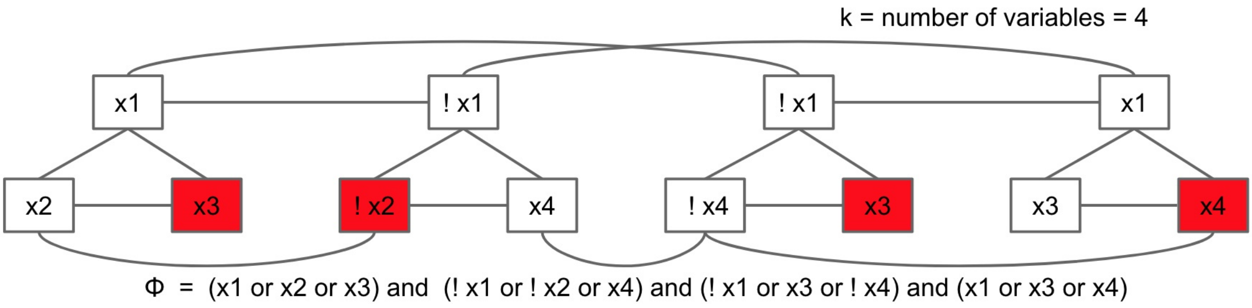
Let's do it!

Preprocess a given 3SAT problem Given an instance X of 3SAT, preprocess it into a graph G:

1. For each clause in X, create 3 vertices in a triangle
2. Add an edge between each literal and its negation



Solve with Independent Sets On graph G, find an independent set of *size = number of clauses in 3SAT*.



Postprocess the output Elements in the independent set are considered "True", while elements outside are considered "False." If you can find an independent set of size = number of clauses in 3SAT, then you've successfully solved 3SAT (using independent sets whoo!).

In the above example, since `x3` , `!x2` , `x3` , `x4` were picked for the independent set, we consider each of those literals to be True and values for the rest don't matter. Therefore, `x3 = True, x2 = False, x4 = True, x1 = doesn't matter.`

Why this works: We'll reference the below example when going through the proof.

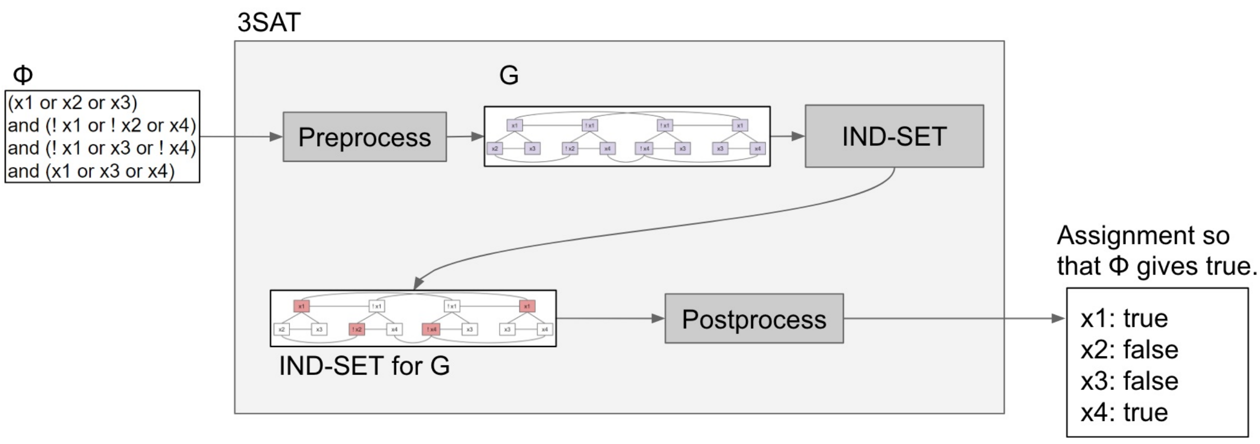
```
(x1 || x2 || !x3) && (x1 || !x1 || x1) && (x2 || x3 || x4)
```

The above 3SAT problem has 3 clauses. To form a satisfying truth assignment we must pick one literal from each clause and give it the value True. Of course, we must be consistent. If we choose `x1` to be True in the first clause, we can't choose `!x1` to be True in the third clause (x1 can't both be True and False!).

Representing a clause by a triangle forces us to pick only literal in a clause for the independent set. Repeat this for every clause and and finding an independent set of *size = number of clauses* means exactly one literal will be picked from each clause (we'll consider a picked node to be True).

We also make sure to add an edge from each literal to its negation to prevent us from choosing opposite literals (e.g. both `x1` and `!x1`) in different clauses. This may also have the effect of finding an independent set impossible - in this case, 3SAT is also not solvable.

Here's a visualization of the above reduction:



Note that reductions are a general concept and apply to many different types of problems (they don't always involve creating graphs!)

Reflection

One can argue that we have been doing reduction all throughout the course.

- Abstract Lists reduce to arrays or linked lists
- Percolation reduces to Disjoint Sets
- Maze generation reduces to [your solution here ;)]

However these aren't exactly reductions because you aren't using a single other algorithm to solve your problem. Notably, in the earlier reduction example we used the Independent Sets algorithm as a 'black box' to solve 3SAT.

Perhaps a better term for what we've been accomplishing earlier in the course is *decomposition* - breaking a complex task into smaller parts. Using abstraction to make problem solving easier. This is the heart of computer science.