Table of Contents

Introduction 1.1
1. Introduction to Java 1.2
    1.1 Essentials 1.2.1
    1.2 Objects 1.2.2
2. Lists 1.3
    2.1 Mystery of the Walrus 1.3.1
    2.2 The SLList 1.3.2
    2.3 The DLList 1.3.3
    2.4 Arrays 1.3.4
    2.5 The AList 1.3.5
3. Testing 1.4
    3.1 A New Way 1.4.1
4. Inheritance, Implements 1.5
    4.1 Intro and interfaces 1.5.1

Hug61B

This book is the companion to Josh Hug's version of CS61B, UC Berkeley's Data Structures course. The current version of the course can be found at datastructur.es/sp18.

<!--This course is about training you to be an efficient programmer [add more].

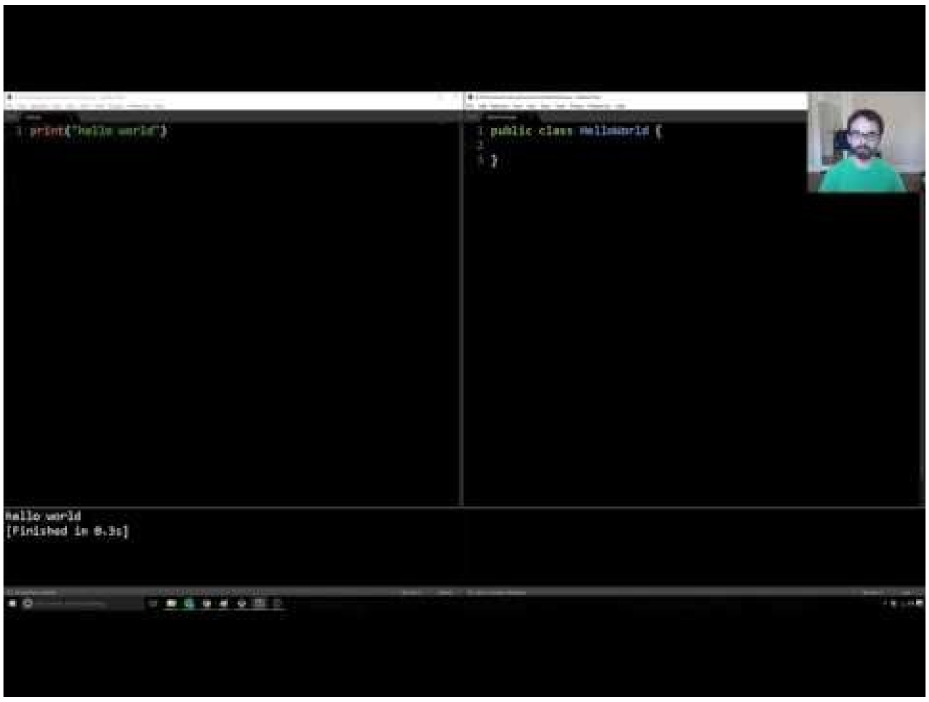
[VIDEO]-->

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 CS61A.

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:

Hello World



```
print("hello world")
```

```
public class HelloWorld {

}
```

```
hello world
[Finished in 0.3s]
```

Video link

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

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

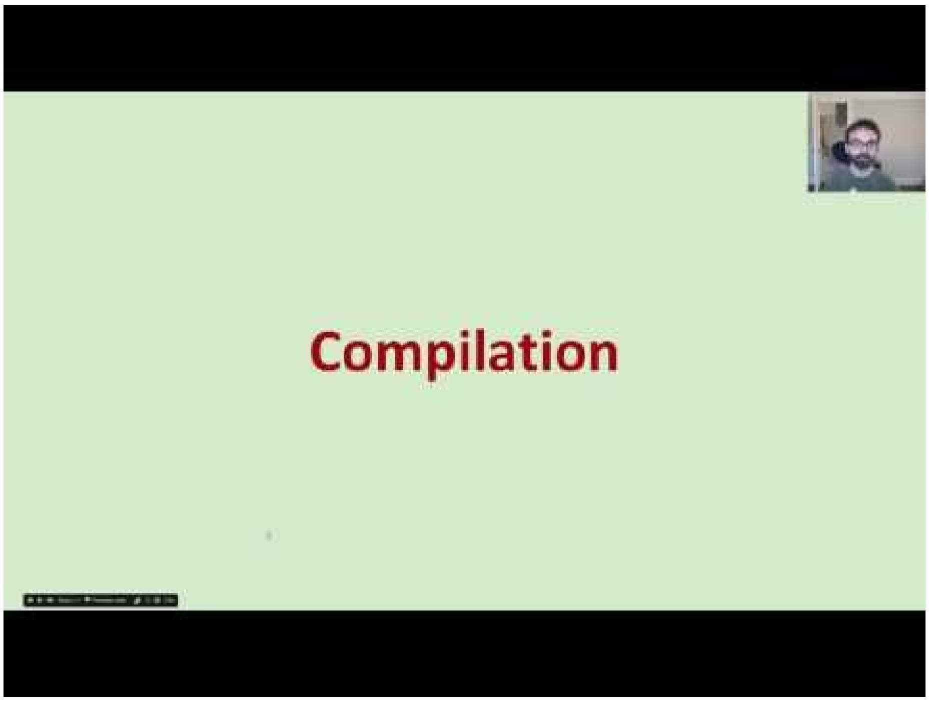
For those of you coming from a language like Python, this probably seems needlessly verbose. However, it's all for good reason, which we'll come to understand over the next couple of weeks. Some key syntactic features to notice:

- 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.

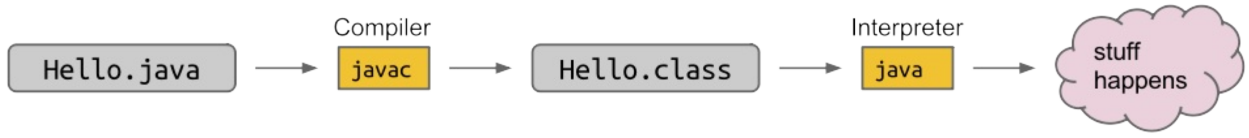
For fun, see Hello world! in other languages.

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).

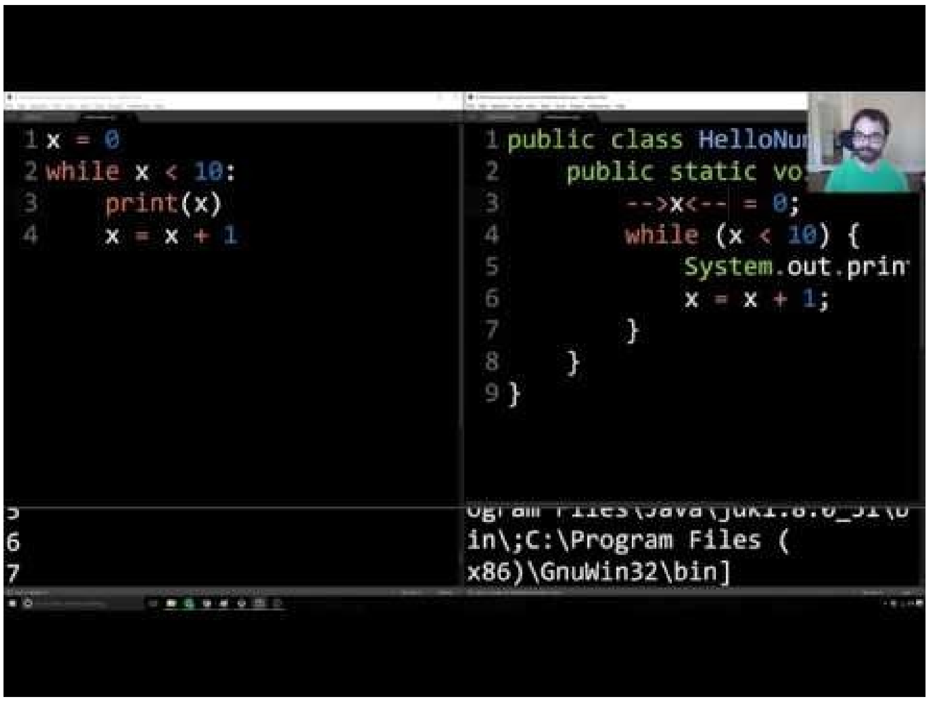
Exercise 1.1.1. Create a file on your computer called HelloWorld.java and copy and paste the exact program from above. Try out the `javac HelloWorld.java` command. It'll look like nothing happened.

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 in a moment.

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



```
x = 0
while x < 10:
    print(x)
    x = x + 1
```

```
public class HelloNu
    public static vo
        -->x<-- = 0;
        while (x < 10) {
            System.out.prin
            x = x + 1;
        }
    }
}
```

```
6
7
```

```
in\;C:\Program Files (
x86)\GnuWin32\bin]
```

Video link

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

```
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.
- 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. For more on gradescope and how to submit your work, see the gradescope guide (link coming later).

If you'd like, you can try submitting `HelloNumbers.java` under the `Lab 1` assignment.

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 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.
- 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")`, as we saw above. Why is that so?

Consider these two Java statements:

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

and

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

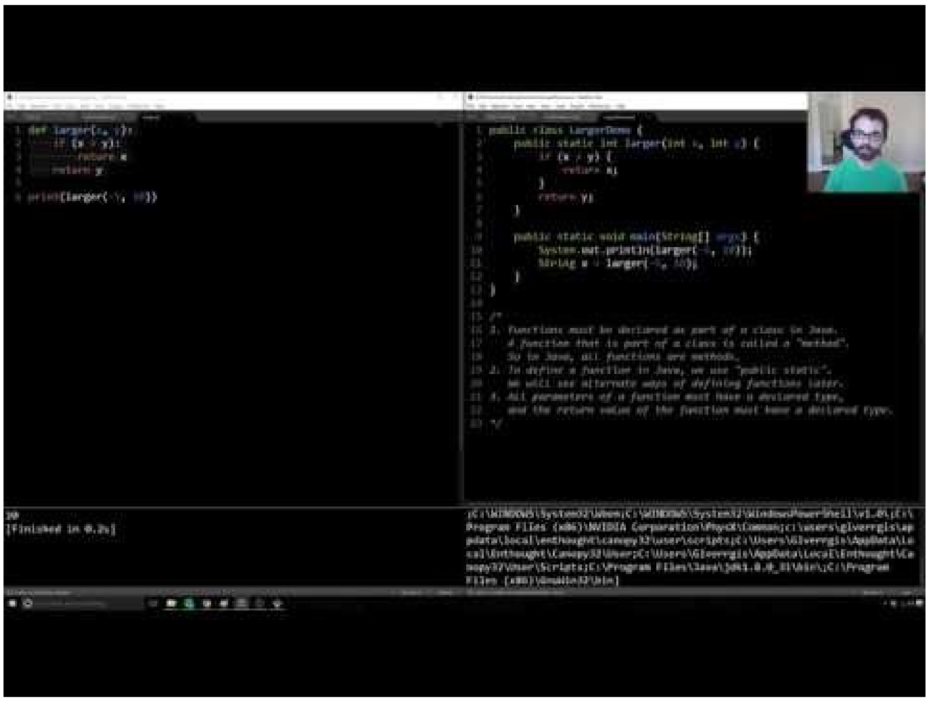
The first one of these will succeed; the second will give a compiler error. Since Java is strongly typed, if you tell it `h` is a string, it can concatenate the elements and give you a string. But when `h` is an `int`, it can't concatenate a number and a string and give you a number.

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



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

print(larger(-5, 10))
```

```
10
[Finished in 0.2s]
```

```
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(-5, 10));
        String x = larger(-5, 10);
    }
}

/*
1. Functions must be declared as part of a class in Java.
   A function that is part of a class is called a "method".
   So in Java, all functions are methods.
2. To define a function in Java, we use "public static".
   We will see alternate ways of defining functions later.
4. All parameters of a function must have a declared type,
   and the return value of the function must have a declared type.
*/
```

```
;C:\WINDOWS\System32\Wbem;C:\WINDOWS\System32\WindowsPowerShell\v1.0\;C:\
Program Files (x86)\NVIDIA Corporation\PhysX\Common;c:\users\glverrgis\ap
pdata\local\enthought\canopy32\user\scripts;C:\Users\Glverrgis\AppData\Lo
cal\Enthought\Canopy32\User;C:\Users\Glverrgis\AppData\Local\Enthought\Ca
nopy32\User\Scripts;C:\Program Files\Java\jdk1.8.0_31\bin\;C:\Program
Files (x86)\GnuWin32\bin]
```

[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.

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:

```java
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.

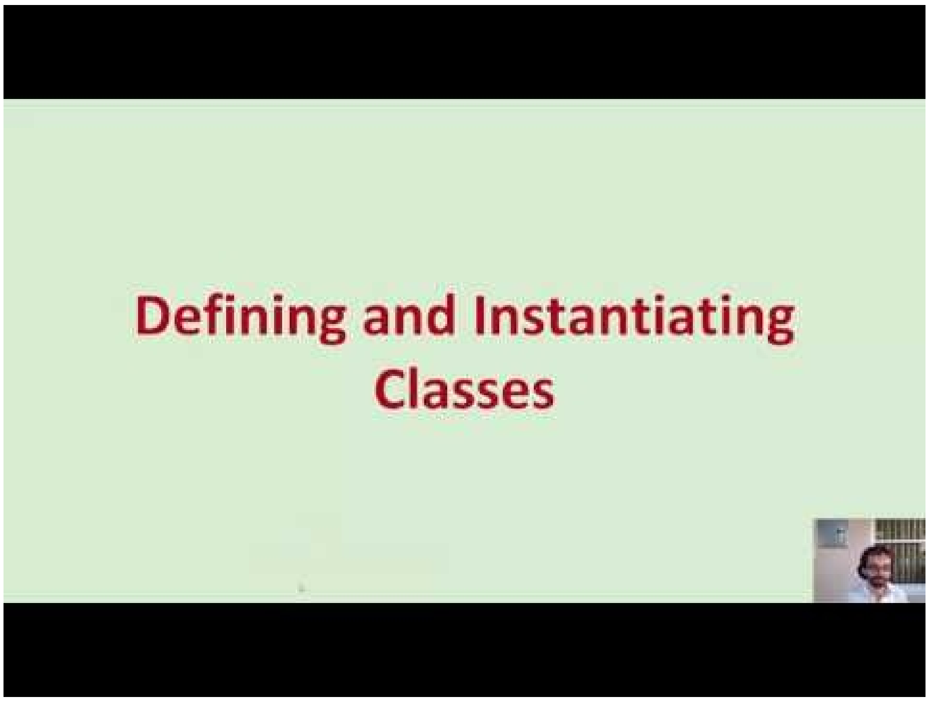
- Homework 0
- Lab 1b
- Lab 1
- Discussion 1

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)
```

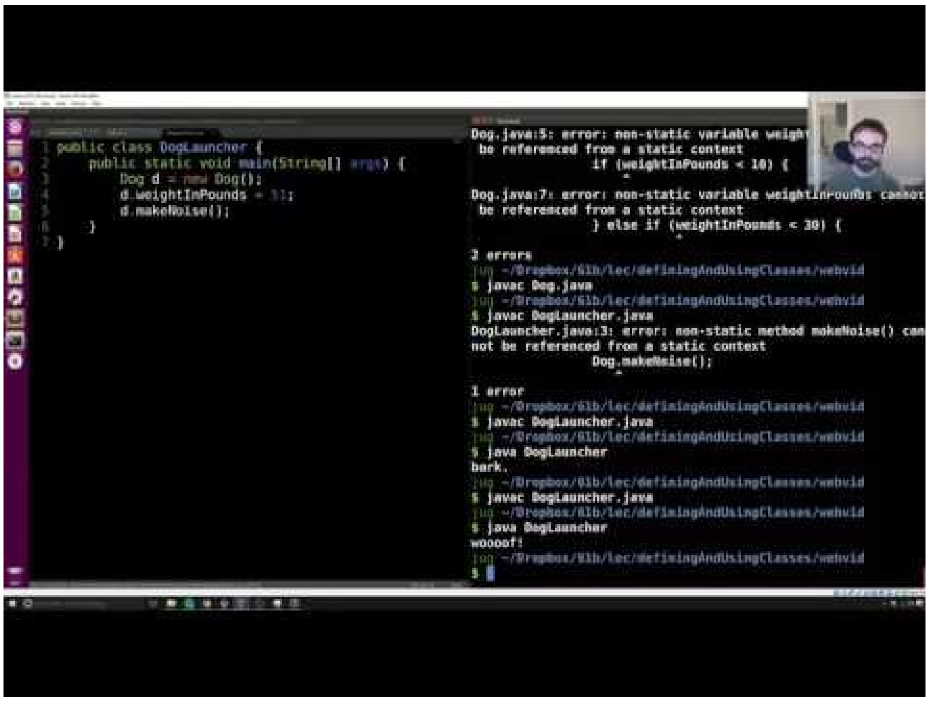
The `Dog` class we've defined doesn't do anything. We've simply defined something that `Dog` can do, namely make noise. To actually run the class, we'd either need to add a main method to the `Dog` class, as we saw in chapter 1.1. Or we could create a separate `DogLauncher` class that runs methods from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



```
public class DogLauncher {
    public static void main(String[] args) {
        Dog d = new Dog();
        d.weightInPounds = 51;
        d.makeNoise();
    }
}
```

```
Dog.java:5: error: non-static variable weigh
 be referenced from a static context
                if (weightInPounds < 10) {
                    ^
Dog.java:7: error: non-static variable weightInPounds cannot
 be referenced from a static context
                } else if (weightInPounds < 30) {
                           ^
2 errors
jug ~/Dropbox/61b/lec/definingAndUsingClasses/webvid
$ javac Dog.java
jug ~/Dropbox/61b/lec/definingAndUsingClasses/webvid
$ javac DogLauncher.java
DogLauncher.java:3: error: non-static method makeNoise() can
not be referenced from a static context
                Dog.makeNoise();
                   ^
1 error
jug ~/Dropbox/61b/lec/definingAndUsingClasses/webvid
$ javac DogLauncher.java
jug ~/Dropbox/61b/lec/definingAndUsingClasses/webvid
$ java DogLauncher
bark.
jug ~/Dropbox/61b/lec/definingAndUsingClasses/webvid
$ javac DogLauncher.java
jug ~/Dropbox/61b/lec/definingAndUsingClasses/webvid
$ java DogLauncher
wooof!
jug ~/Dropbox/61b/lec/definingAndUsingClasses/webvid
$
```

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:

```
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:

```
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*:

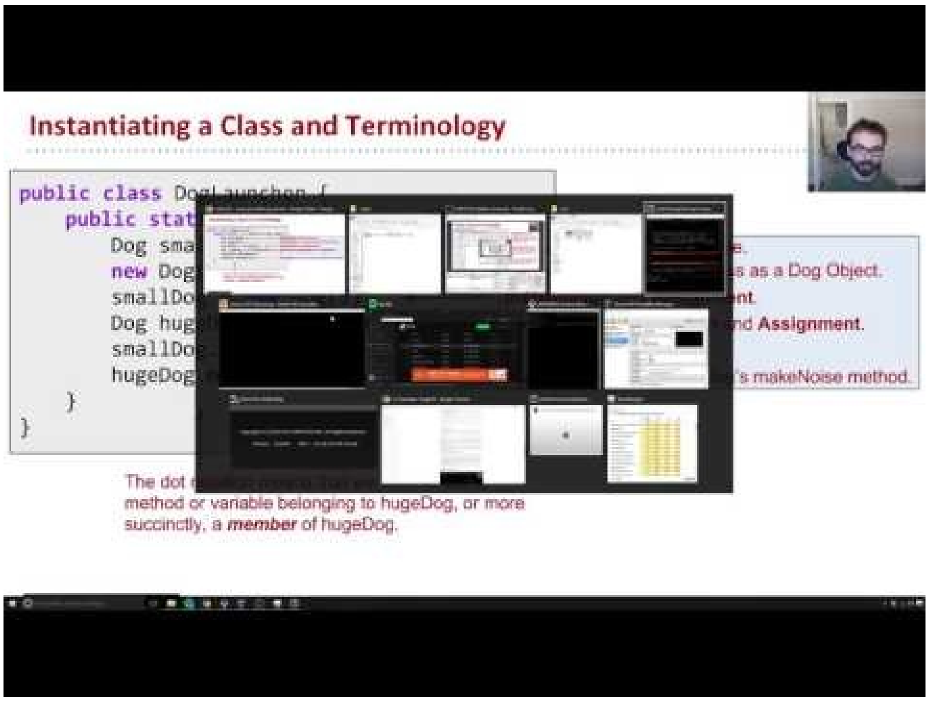
```
public class DogLauncher {
    public static void main(String[] args) {
        Dog d = new Dog(20);
        d.makeNoise();
    }
}
```

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



Instantiating a Class and Terminology

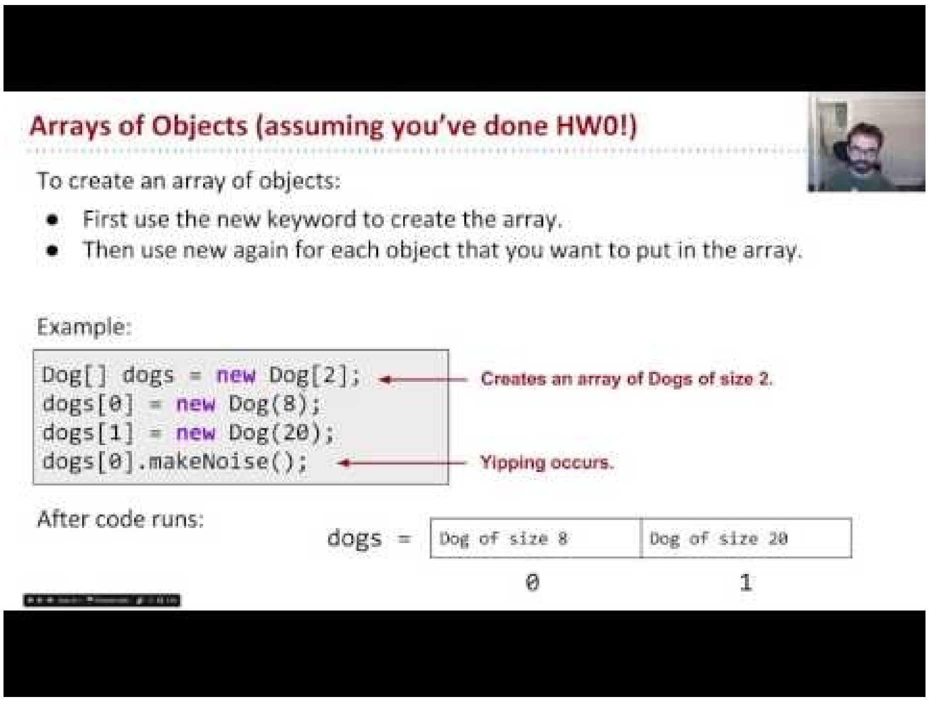
```
public class DogLauncher {
    public stat
        Dog sma
        new Dog
        smallDo
        Dog hug
        smallDo
        hugeDog
    }
}
```

e.
s as a Dog Object.
nt.
nd Assignment.
's makeNoise method.

The dot
method or variable belonging to hugeDog, or more succinctly, a *member* of hugeDog.

[Video link](#)

Array Instantiation, Arrays of Objects



Arrays of Objects (assuming you've done HW0!)

To create an array of objects:

- First use the new keyword to create the array.
- Then use new again for each object that you want to put in the array.

Example:

```
Dog[] dogs = new Dog[2];
dogs[0] = new Dog(8);
dogs[1] = new Dog(20);
dogs[0].makeNoise();
```

Creates an array of Dogs of size 2.

Yipping occurs.

After code runs:

	Dog of size 8	Dog of size 20
dogs =	0	1

[Video link](#)

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

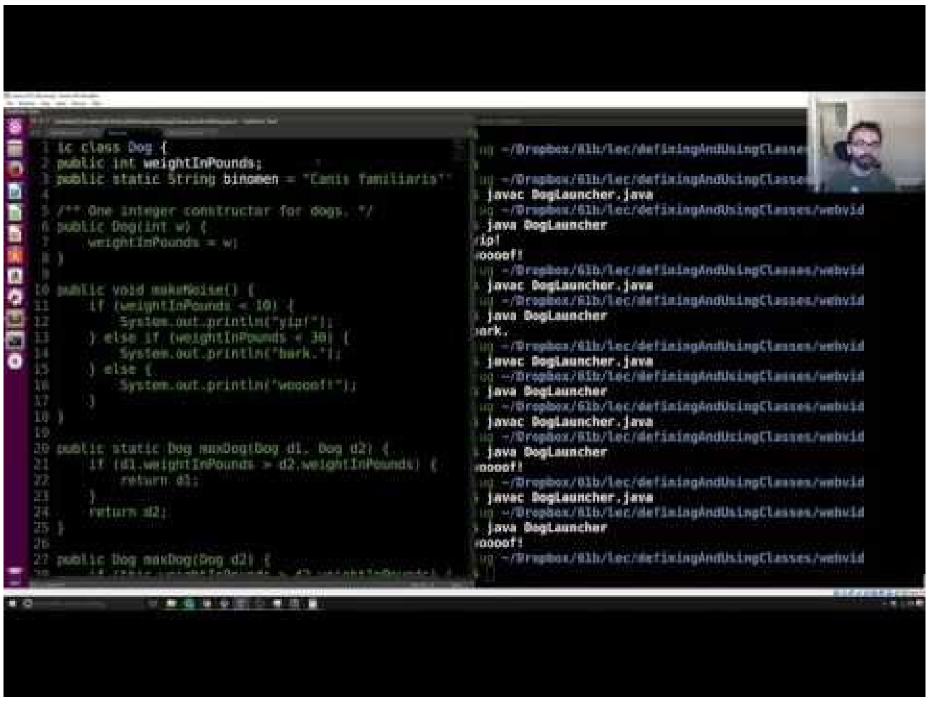
```
public class ArrayDemo {
    public static void main(String[] args) {
        /* Create an array of five integers. */
        int[] someArray = new int[5];
        someArray[0] = 3;
        someArray[1] = 4;
    }
}
```

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

```
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



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

/** One integer constructor for dogs. */
public Dog(int w) {
    weightInPounds = w;
}

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

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

public Dog maxDog(Dog d2) {
```

```
ug ~/Dropbox/61b/lec/definingAndUsingClasse
ug ~/Dropbox/61b/lec/definingAndUsingClasse
 javac DogLauncher.java
ug ~/Dropbox/61b/lec/definingAndUsingClasses/webvid
 java DogLauncher
ip!
oooof!
ug ~/Dropbox/61b/lec/definingAndUsingClasses/webvid
 javac DogLauncher.java
ug ~/Dropbox/61b/lec/definingAndUsingClasses/webvid
 java DogLauncher
ark.
ug ~/Dropbox/61b/lec/definingAndUsingClasses/webvid
 javac DogLauncher.java
ug ~/Dropbox/61b/lec/definingAndUsingClasses/webvid
 java DogLauncher
ug ~/Dropbox/61b/lec/definingAndUsingClasses/webvid
 javac DogLauncher.java
ug ~/Dropbox/61b/lec/definingAndUsingClasses/webvid
 java DogLauncher
oooof!
ug ~/Dropbox/61b/lec/definingAndUsingClasses/webvid
 javac DogLauncher.java
ug ~/Dropbox/61b/lec/definingAndUsingClasses/webvid
 java DogLauncher
oooof!
ug ~/Dropbox/61b/lec/definingAndUsingClasses/webvid
```

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";
    ...
}
```

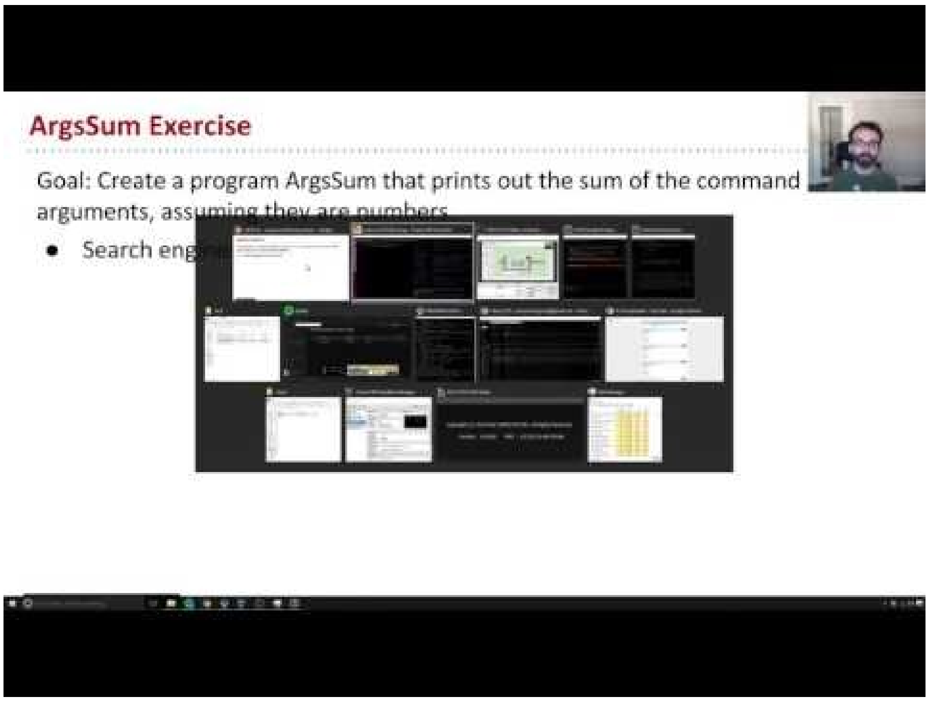
Static variables should be accessed using the name of the class rather than a specific instance, e.g. you should use `Dog.binomen`, not `d.binomen`.

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)


ArgsSum Exercise

Goal: Create a program ArgsSum that prints out the sum of the command arguments, assuming they are numbers

- Search eng

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:

```
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.

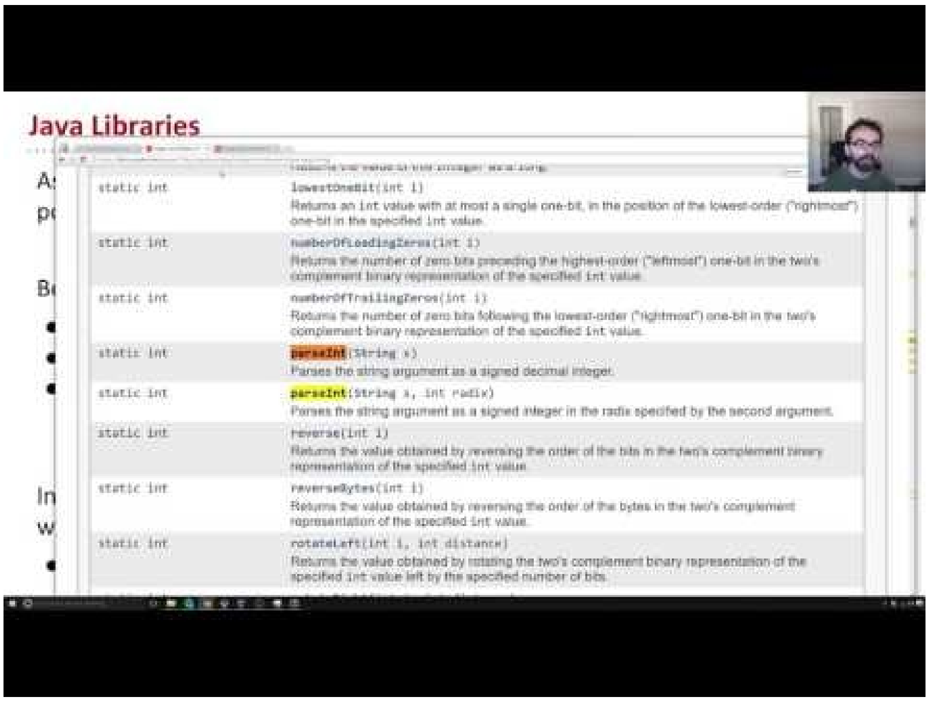
```
$ java ArgsDemo these are command line arguments
these
```

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


Java Libraries

static int	lowestOneBit(int i) Returns an int value with at most a single one-bit, in the position of the lowest-order ("rightmost") one-bit in the specified int value.
static int	numberOfLeadingZeros(int i) Returns the number of zero bits preceding the highest-order ("leftmost") one-bit in the two's complement binary representation of the specified int value.
static int	numberOfTrailingZeros(int i) Returns the number of zero bits following the lowest-order ("rightmost") one-bit in the two's complement binary representation of the specified int value.
static int	parseInt(String s) Parses the string argument as a signed decimal integer.
static int	parseInt(String s, int radix) Parses the string argument as a signed integer in the radix specified by the second argument.
static int	reverse(int i) Returns the value obtained by reversing the order of the bits in the two's complement binary representation of the specified int value.
static int	reverseBytes(int i) Returns the value obtained by reversing the order of the bytes in the two's complement representation of the specified int value.
static int	rotateLeft(int i, int distance) Returns the value obtained by rotating the two's complement binary representation of the specified int value left by the specified number of bits.

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

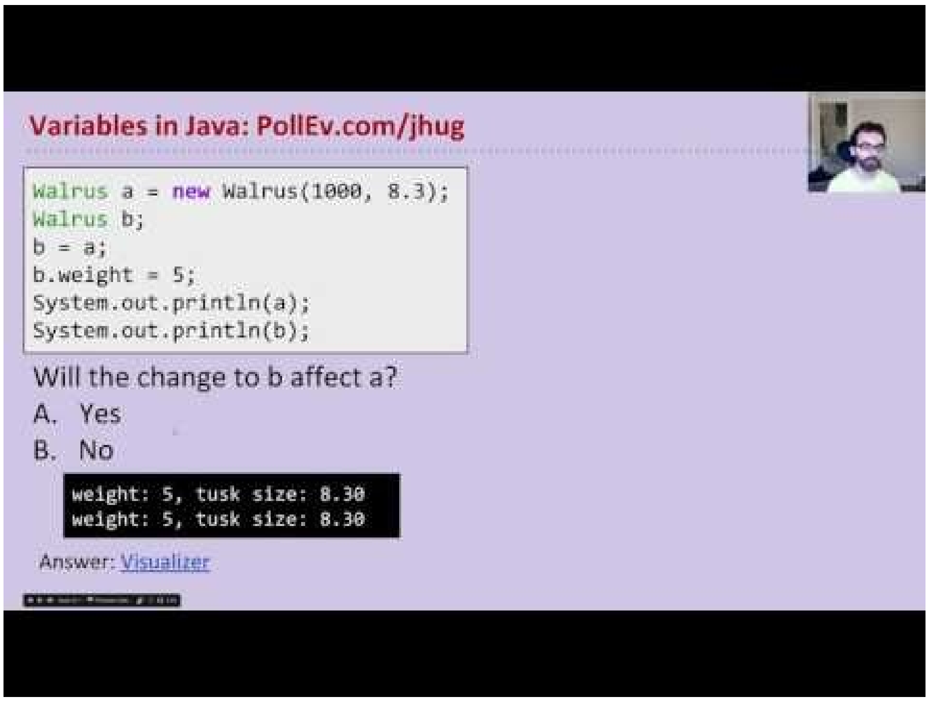
In Project 0, we use arrays to track the positions of N objects in space. One thing we would not have been able to easily do is change the number of objects after the simulation had begun. This is because arrays have a fixed size in Java that can never change.

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



Variables in Java: PollEv.com/jhug

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

Will the change to b affect a?

A. Yes

B. No

```
weight: 5, tusk size: 8.30
weight: 5, tusk size: 8.30
```

Answer: Visualizer

[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);
```

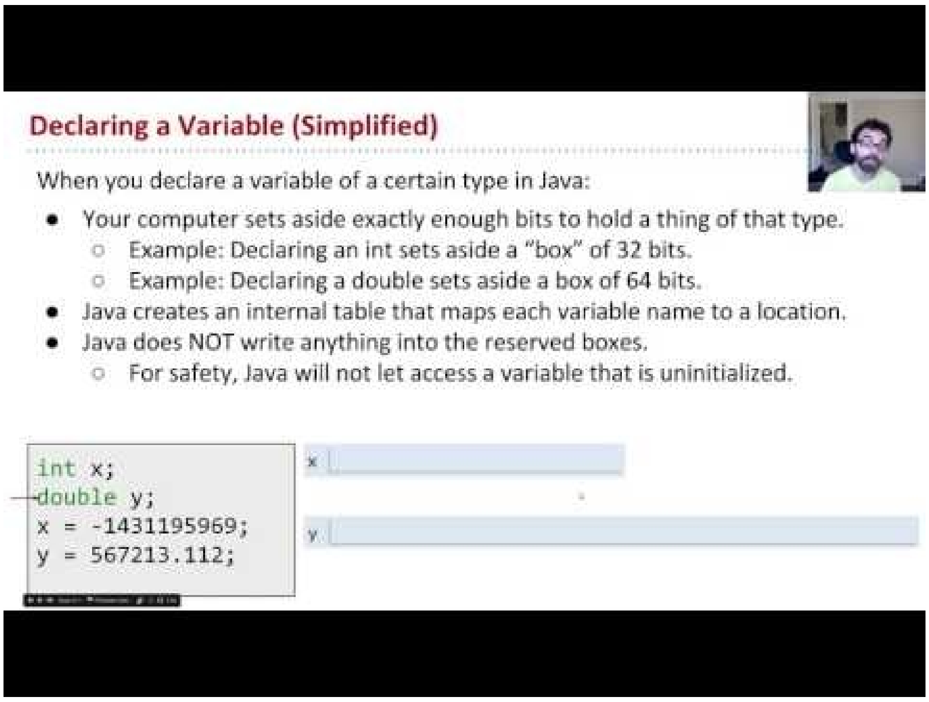
Now try to predict what happens when we run the code below. Does the change to x affect y?

```
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



Declaring a Variable (Simplified)

When you declare a variable of a certain type in Java:

- Your computer sets aside exactly enough bits to hold a thing of that type.
  - Example: Declaring an int sets aside a "box" of 32 bits.
  - Example: Declaring a double sets aside a box of 64 bits.
- Java creates an internal table that maps each variable name to a location.
- Java does NOT write anything into the reserved boxes.
  - For safety, Java will not let access a variable that is uninitialized.

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

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 of 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.

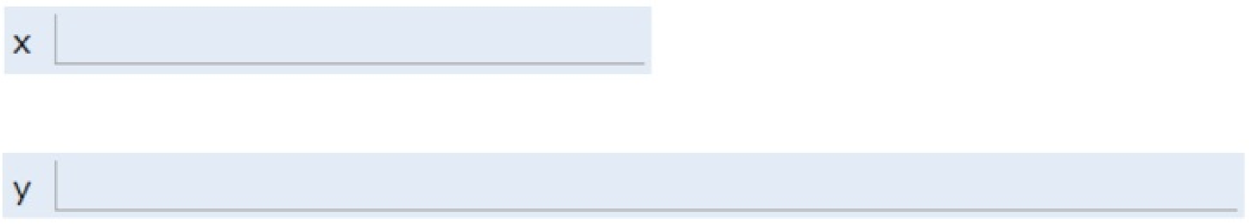
For the sake of having a convenient metaphor, we'll call one of these blocks a "box" of bits.

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:

```java
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".

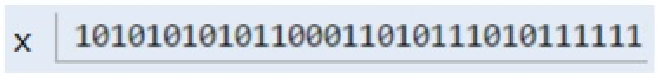
As an analogy, you do not have direct control over your heartbeat. While this restricts your ability to optimize for certain situations, it also avoids the possibility of making stupid errors like accidentally turning it off.

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.

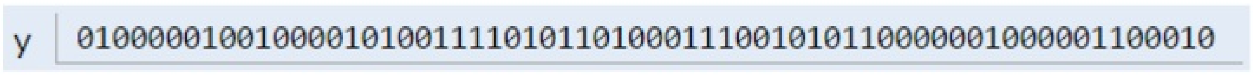
When you assign values to a memory box, it is filled with the bits you specify. For example, if we execute the lines:

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

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



```
x  10101010101100011010111010111111
```



```
y  0100000100100001010011110101101000111001010110000001000001100010
```

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.

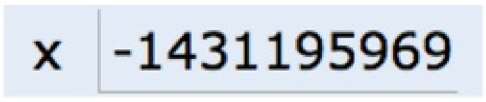
Simplified Box Notation

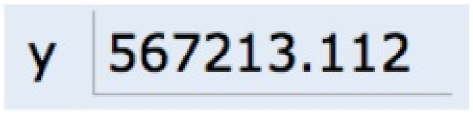
While the box notation we used in the previous section is great for understanding approximately what's going on under the hood, it's not useful for practical purposes since we don't know how to interpret the binary bits.

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.

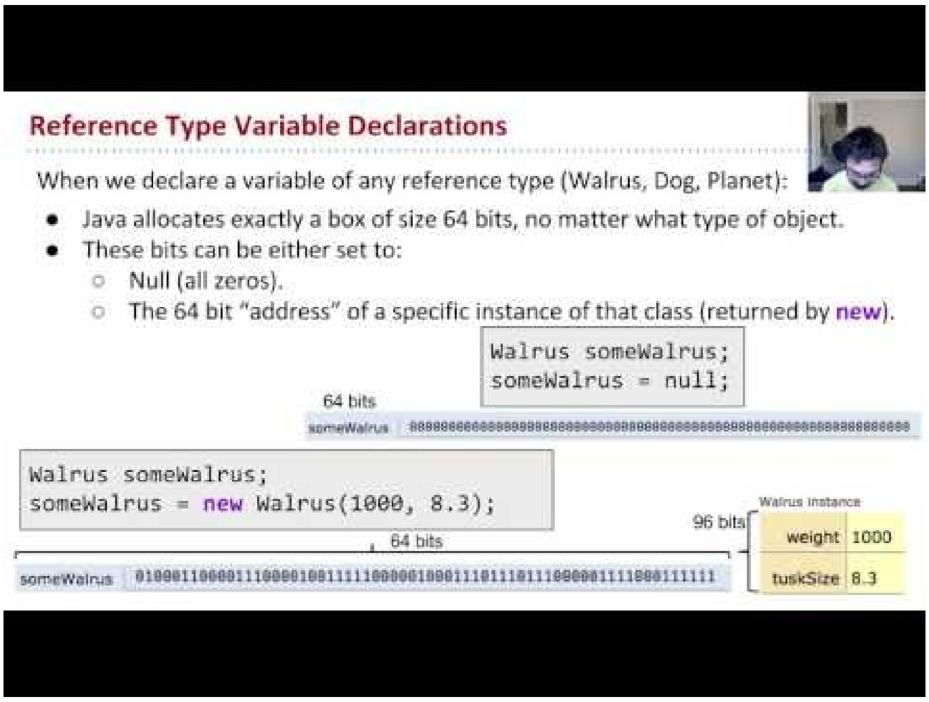
```
int x = 5;
int y;
y = x;
x = 2;
System.out.println("x is: " + x);
System.out.println("y is: " + y);
```

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



Reference Type Variable Declarations

When we declare a variable of any reference type (Walrus, Dog, Planet):

- Java allocates exactly a box of size 64 bits, no matter what type of object.
- These bits can be either set to:
  - Null (all zeros).
  - The 64 bit "address" of a specific instance of that class (returned by new).

```
Walrus someWalrus;
someWalrus = null;
```

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

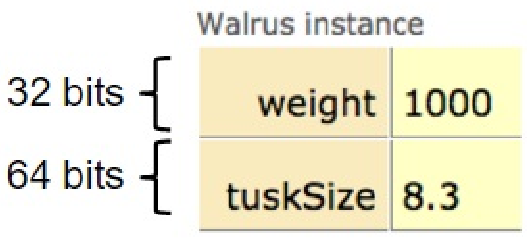
Video link

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.

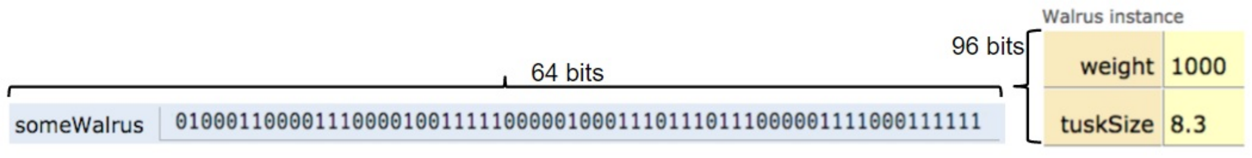
However, this problem is easily resolved with the following piece of information: the 64 bit box contains not the data about the walrus, but instead the address of the Walrus in memory.

As an example, suppose we call:

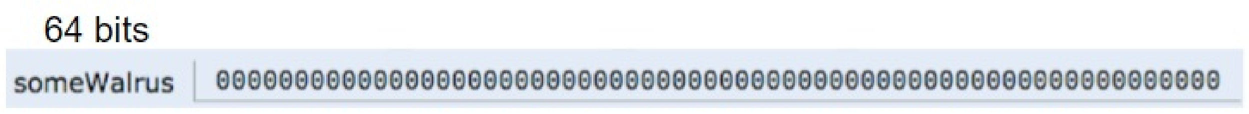
```
Walrus someWalrus;
someWalrus = new Walrus(1000, 8.3);
```

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 `0100011000011100001001111110000010001110111011100000111000111111`, giving us in box notation:



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



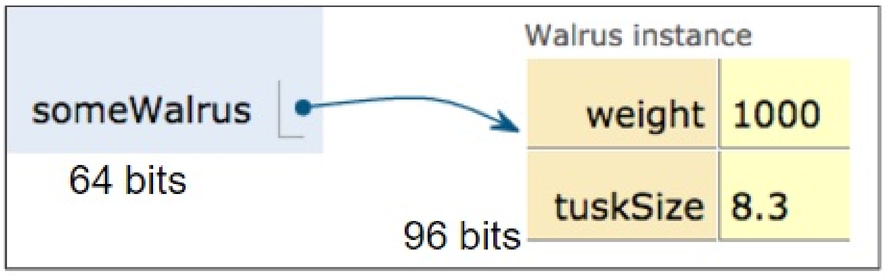
Box and Pointer Notation

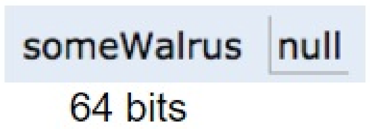
Just as before, it's hard to interpret a bunch of bits inside a reference variable, so we'll create a simplified box notation for reference variable as follows:

- 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:



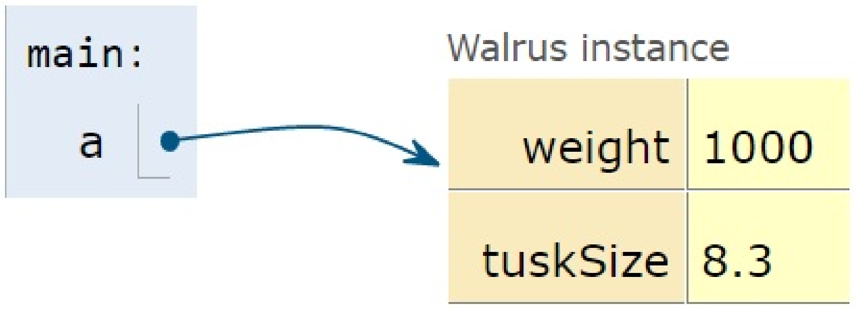


Resolving the Mystery of the Walrus

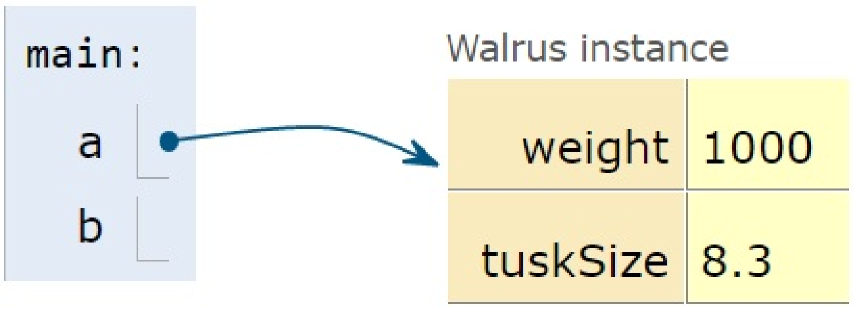
We're now finally ready to resolve, fully and completely, the Mystery of the Walrus.

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

After the first line is executed, we have:

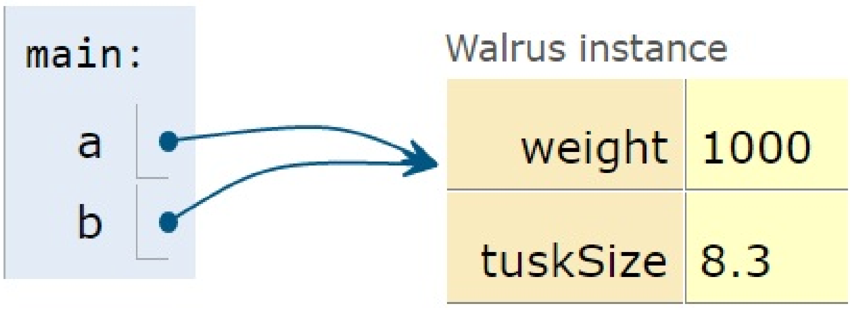


After the second line is executed, we have:



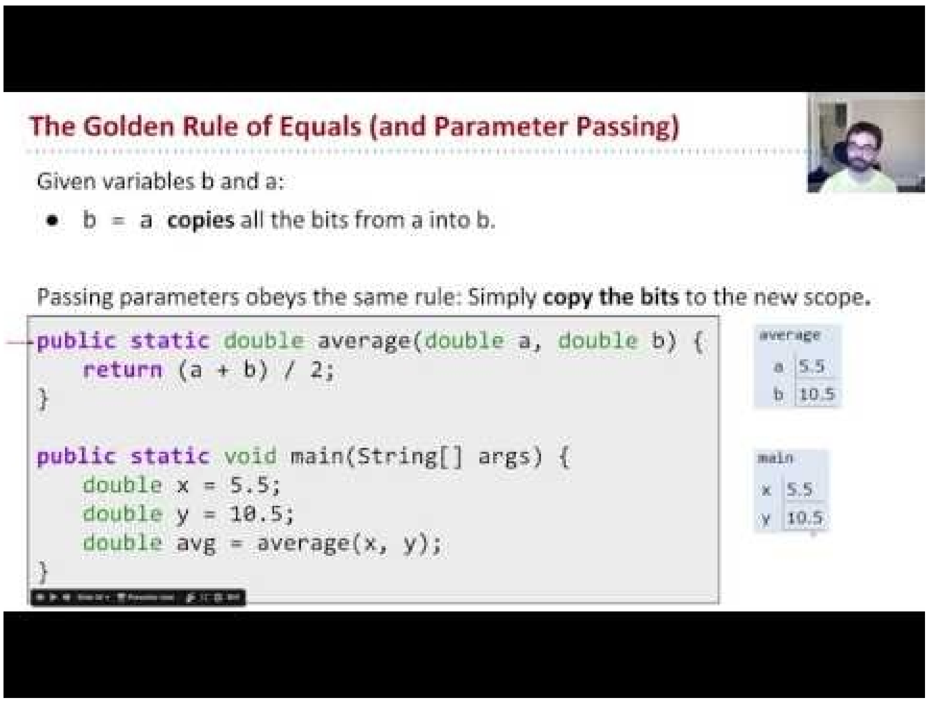
Note that above, b is undefined, not null.

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



The Golden Rule of Equals (and Parameter Passing)

Given variables b and a:

- `b = a` **copies** all the bits from a into b.

Passing parameters obeys the same rule: Simply **copy the bits** to the new scope.

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

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

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.

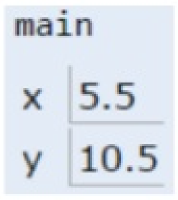
For example, consider the function below:

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

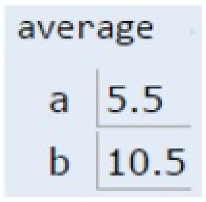
Suppose we invoke this function as shown below:

```
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:

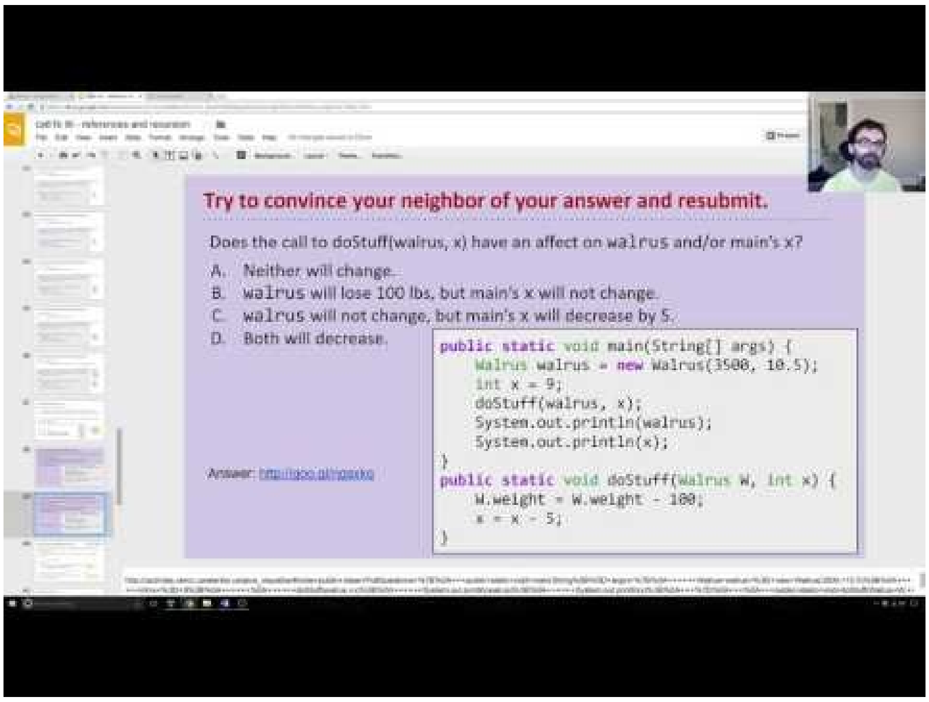


When the function is invoked, the `average` function has its own scope with two new boxes labeled as `a` and `b`, and the bits are simply copied in.



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


Try to convince your neighbor of your answer and resubmit.

Does the call to doStuff(walrus, x) have an affect on walrus and/or main's x?

A. Neither will change.
B. walrus will lose 100 lbs, but main's x will not change.
C. walrus will not change, but main's x will decrease by 5.
D. Both will decrease.

Answer: http://goo.gl/ngsxkq

```
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;
}
```

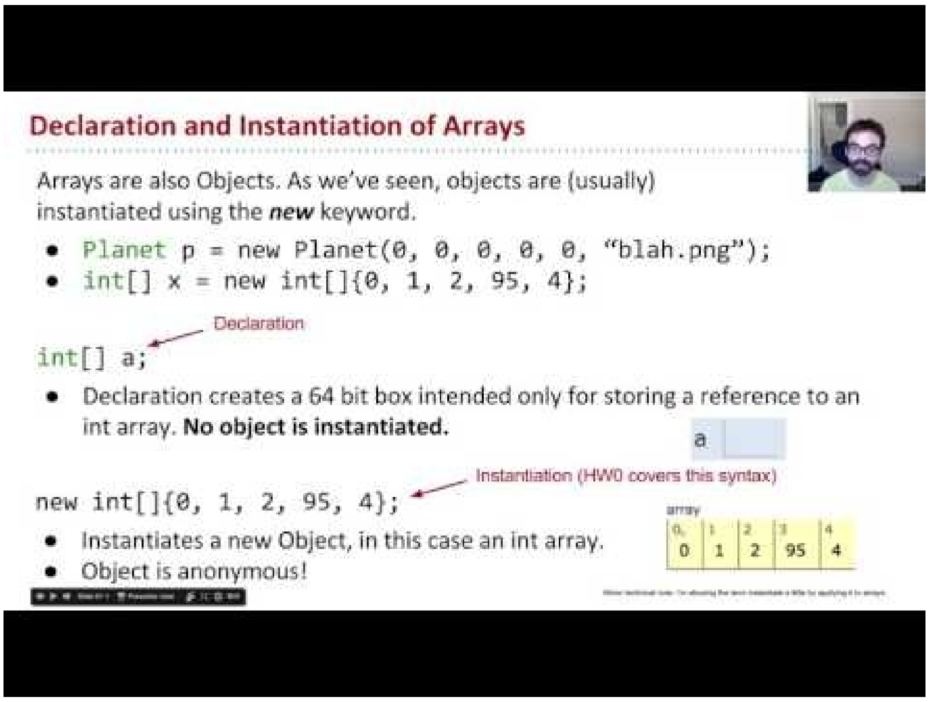
Video link

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



Declaration and Instantiation of Arrays

Arrays are also Objects. As we've seen, objects are (usually) instantiated using the ***new*** keyword.

- `Planet p = new Planet(0, 0, 0, 0, 0, “blah.png”);`
- `int[] x = new int[]{0, 1, 2, 95, 4};`

```
int[] a;
```

- Declaration creates a 64 bit box intended only for storing a reference to an int array. **No object is instantiated.**

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

- Instantiates a new Object, in this case an int array.
- Object is anonymous!

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.

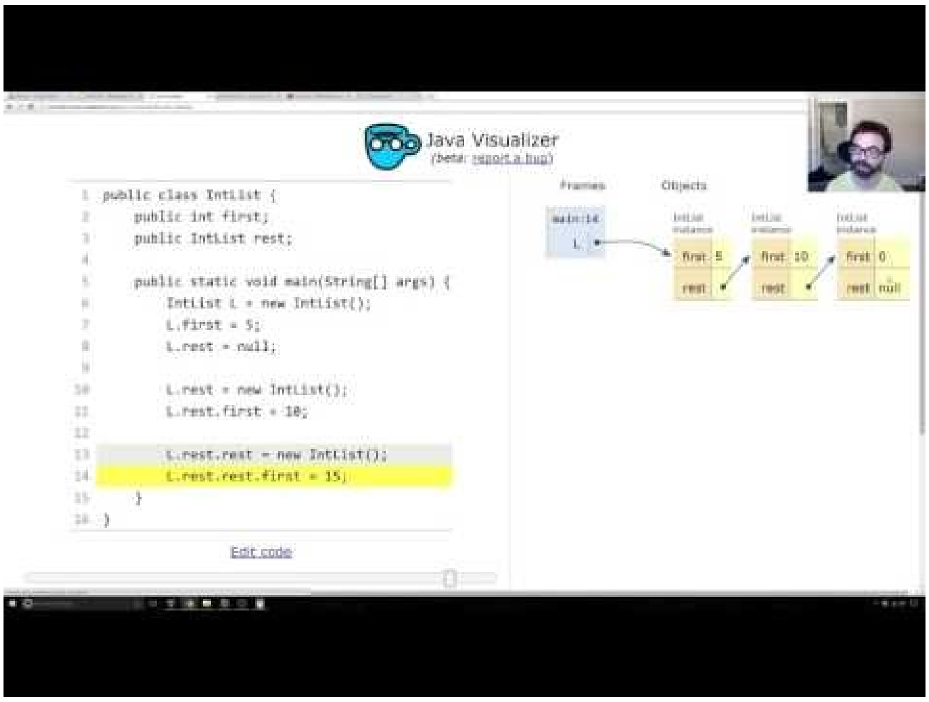
Objects can be lost if you lose the bits corresponding to the address. For example if the only copy of the address of a particular Walrus is stored in `x` , then `x = null` will cause you to permanently lose this Walrus. This isn't necessarily a bad thing, since you'll often decide you're done with an 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



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

    public static void main(String[] args) {
        IntList L = new IntList();
        L.first = 5;
        L.rest = null;

        L.rest = new IntList();
        L.rest.first = 10;

        L.rest.rest = new IntList();
        L.rest.rest.first = 15;
    }
}
```

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:

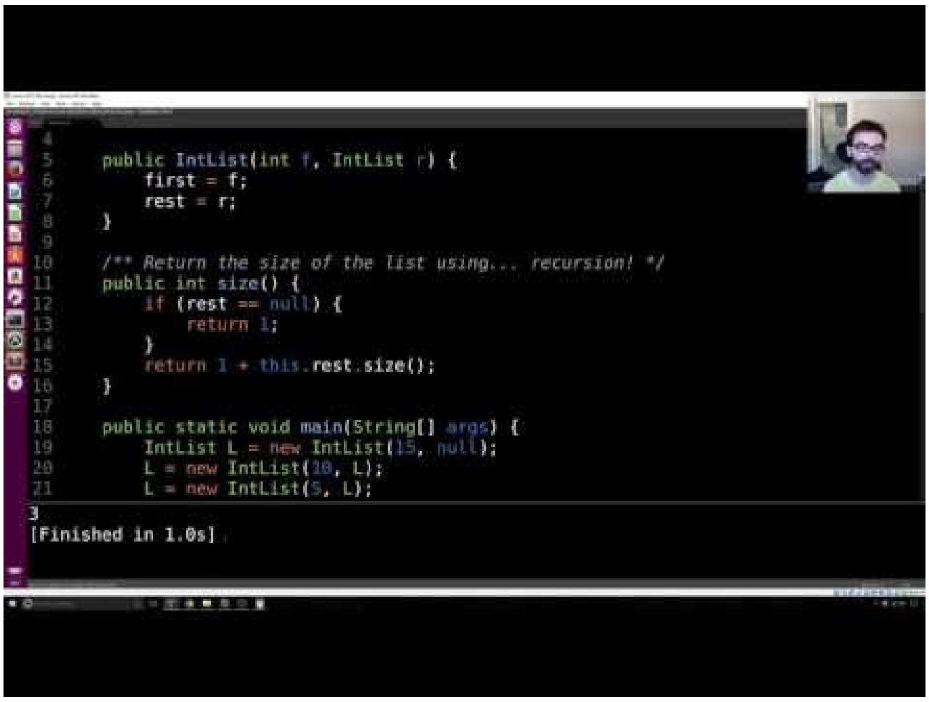
```
IntList L = new IntList(15, null);
L = new IntList(10, L);
L = new IntList(5, L);
```

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`.

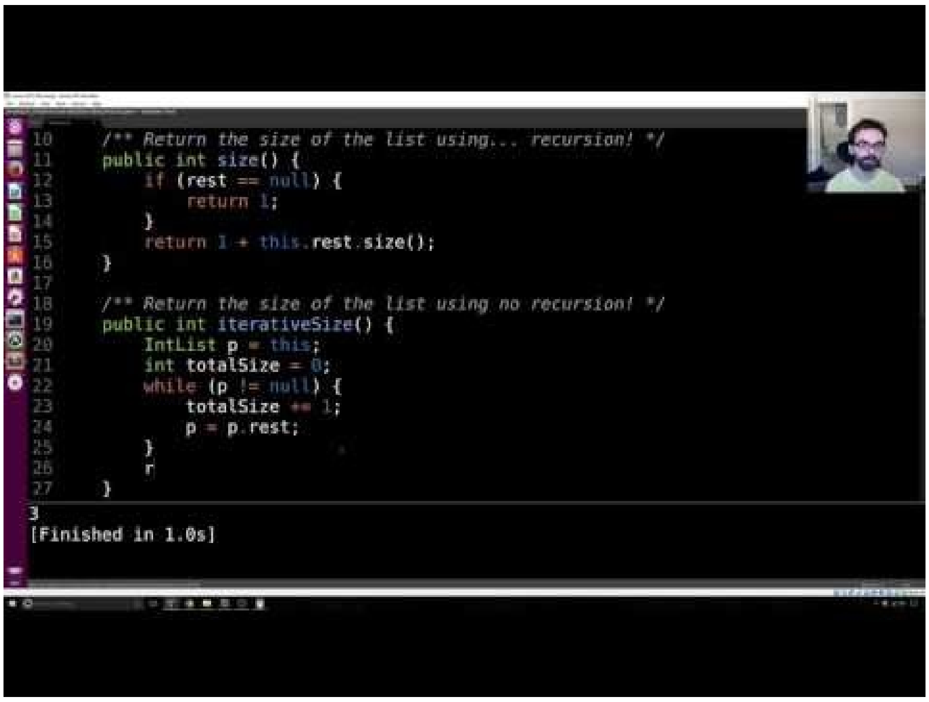
Consider writing a `size` and `iterativeSize` method before reading the rest of this chapter. `size` should use recursion, and `iterativeSize` should not. You'll probably learn more by trying on your own before seeing how I do it. The two videos provide a live demonstration of how one might implement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



```java

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

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

    public static void main(String[] args) {
        IntList L = new IntList(15, null);
        L = new IntList(10, L);
        L = new IntList(5, L);
```

```
3
[Finished in 1.0s]
```

Video link



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

    /** 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;
        }
        r
    }
```

```
3
[Finished in 1.0s]
```

Video link

My `size` method is as shown below:

```
/** 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.

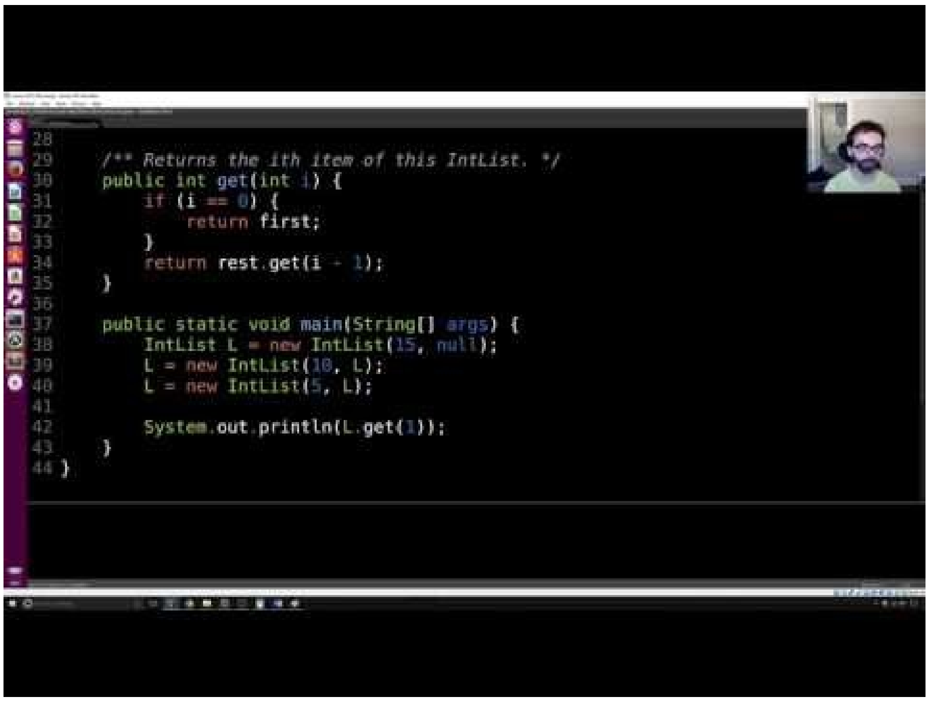
Exercise: You might wonder why we don't do something like `if (this == null) return 0;`. Why wouldn't this work?

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.

```
/** 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



```
    /** Returns the ith item of this IntList. */
    public int get(int i) {
        if (i == 0) {
            return first;
        }
        return rest.get(i - 1);
    }

    public static void main(String[] args) {
        IntList L = new IntList(15, null);
        L = new IntList(10, L);
        L = new IntList(5, L);

        System.out.println(L.get(1));
    }
}
```

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

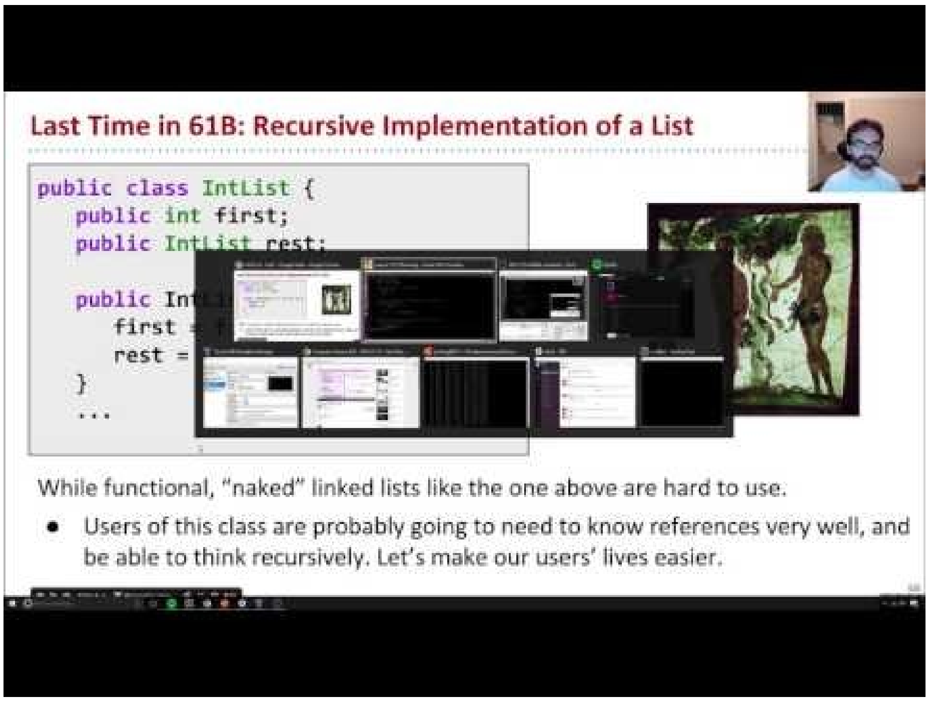
SLLists

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



Last Time in 61B: Recursive Implementation of a List

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

   public Int
      first =
      rest =
   }
   ...
```

While functional, "naked" linked lists like the one above are hard to use.

- Users of this class are probably going to need to know references very well, and be able to think recursively. Let's make our users' lives easier.

Video link

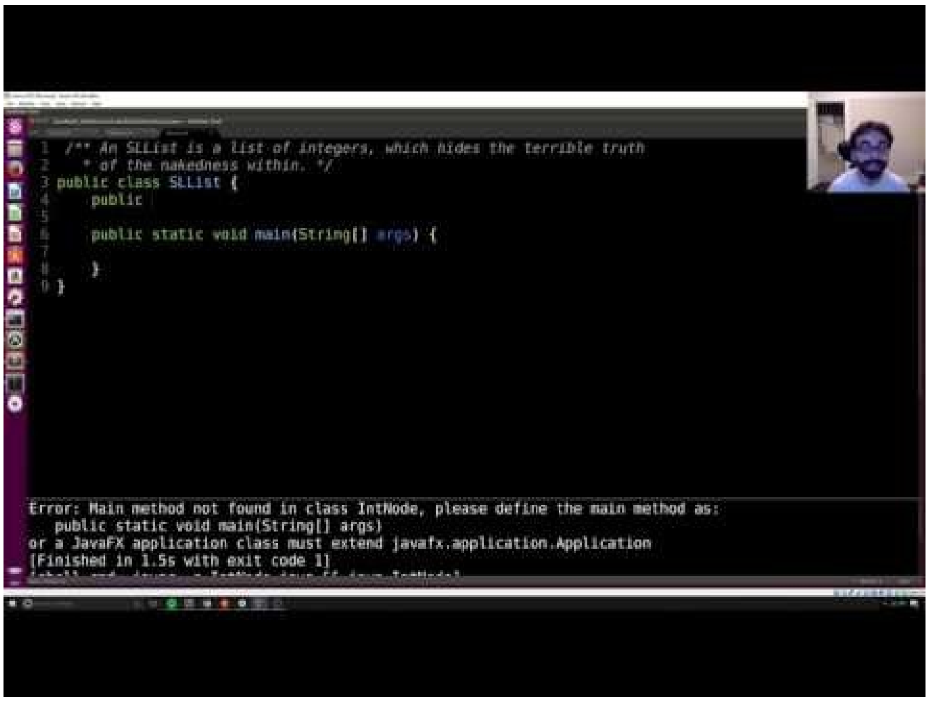
Our `IntList` class from last time was as follows, with helper methods omitted:

```
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.

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

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

Improvement #2: Bureaucracy



```
/** An SLList is a list of integers, which hides the terrible truth
  * of the nakedness within. */
public class SLList {
    public

    public static void main(String[] args) {

    }
}
```

```
Error: Main method not found in class IntNode, please define the main method as:
   public static void main(String[] args)
or a JavaFX application class must extend javafx.application.Application
[Finished in 1.5s with exit code 1]
```

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:

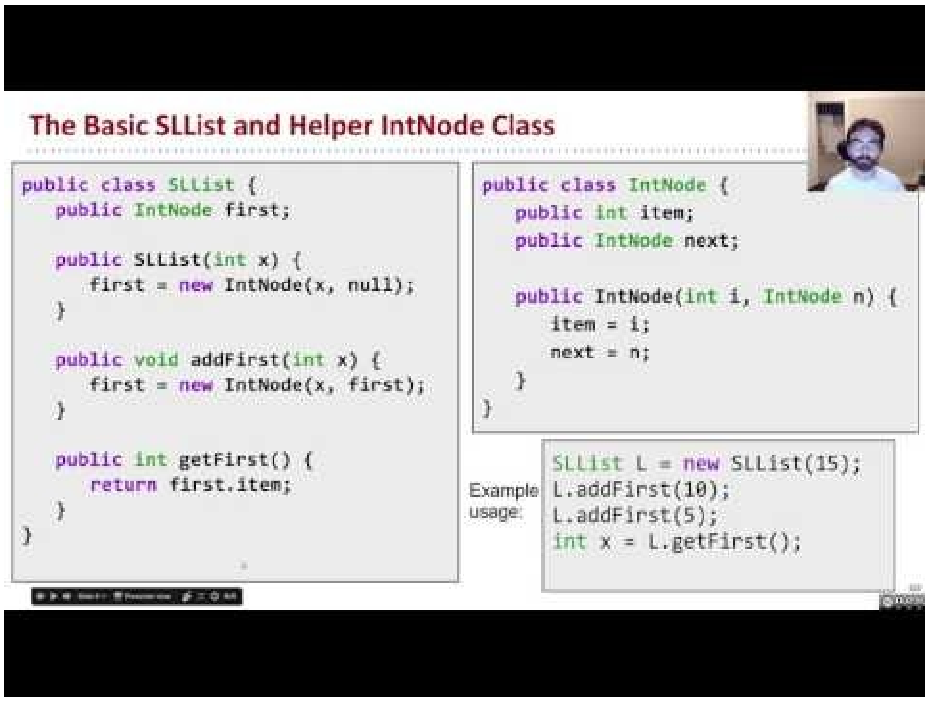
```
public class SLList {
    public IntNode first;

    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



The Basic SLList and Helper IntNode Class

```
public class SLList {
   public IntNode first;

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

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

   public int getFirst() {
      return first.item;
   }
}
```

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

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

Example usage:

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

```
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` :

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

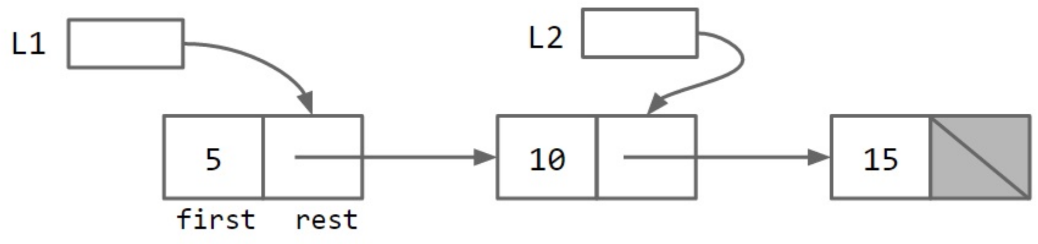
The resulting `SLList` class is much easier to use. Compare:

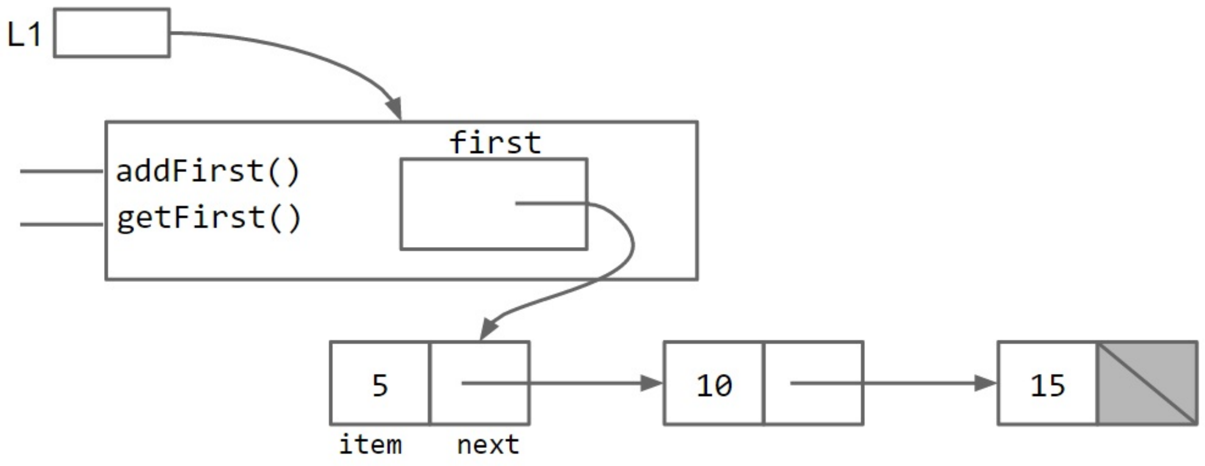
```
SLList L = new SLList(15);
L.addFirst(10);
L.addFirst(5);
int x = L.getFirst();
```

to the `IntList` equivalent:

```
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:

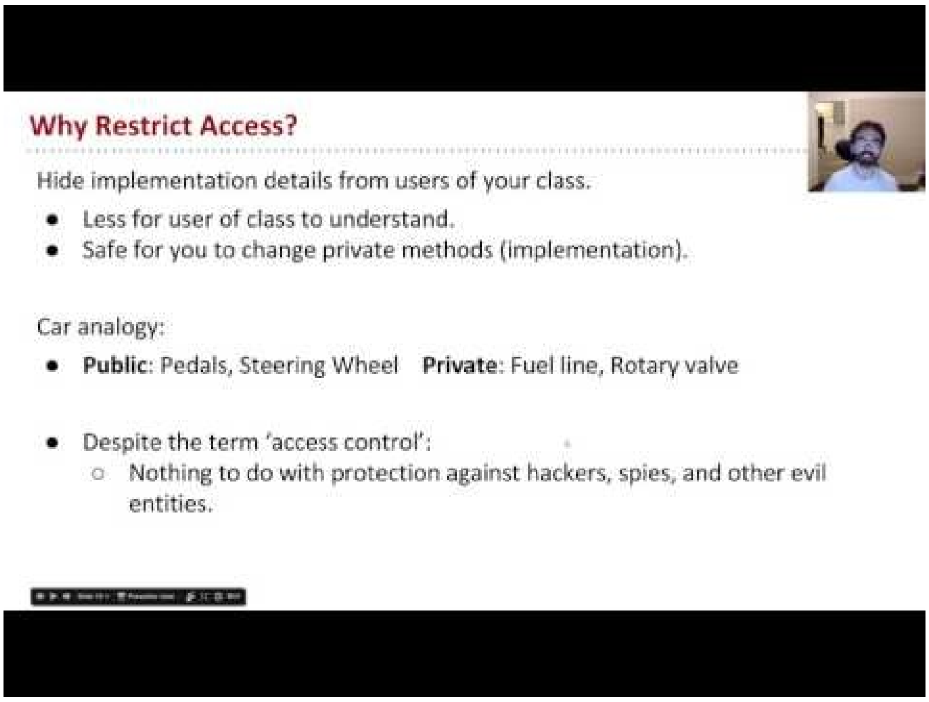




Essentially, the `SLList` class acts as a middleman between the list user and the naked recursive data structure. As Ovid said: Mortals cannot look upon a god without dying, so perhaps it is best that the `SLList` is there to act as our intermediary.

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



Why Restrict Access?

Hide implementation details from users of your class.

- Less for user of class to understand.
- Safe for you to change private methods (implementation).

Car analogy:

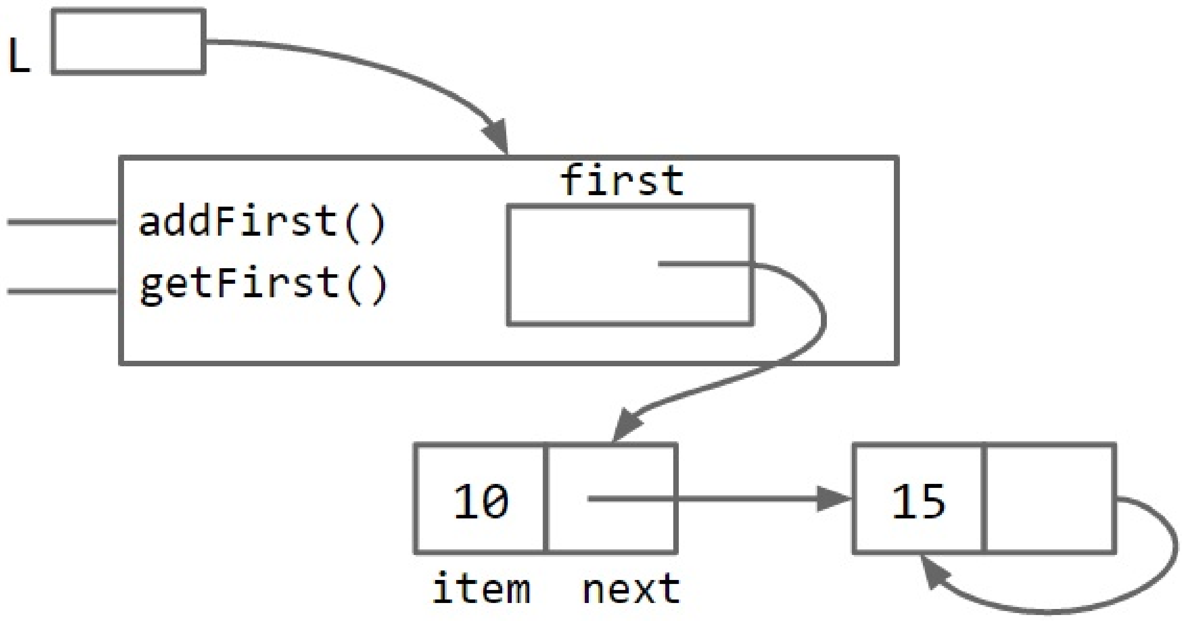
- **Public**: Pedals, Steering Wheel **Private**: Fuel line, Rotary valve

- Despite the term 'access control':
    - Nothing to do with protection against hackers, spies, and other evil entities.

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;
       ...
```

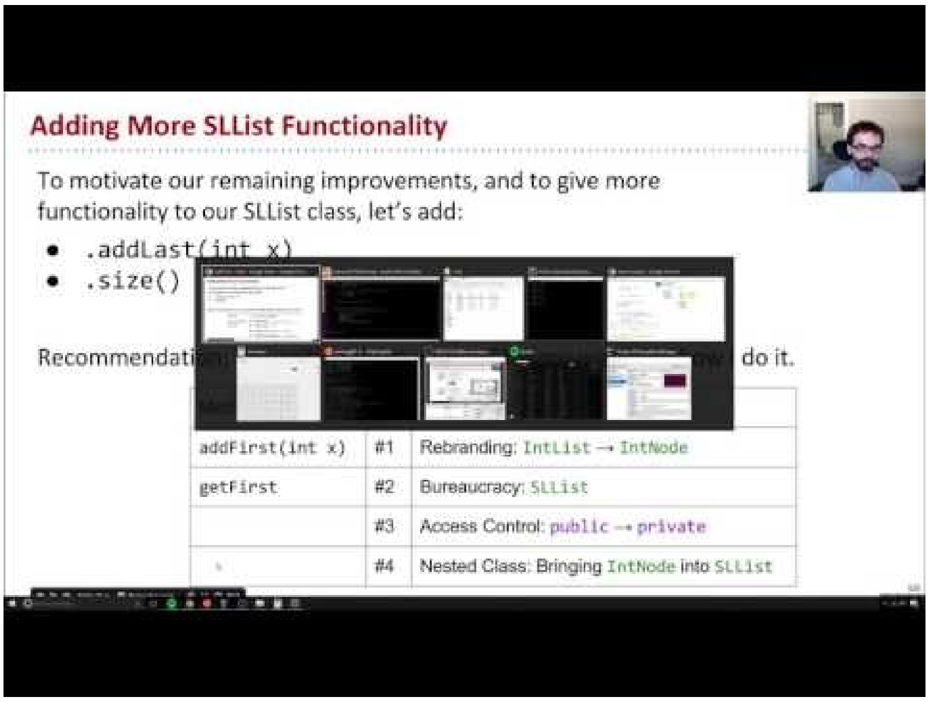
This saves a bit of memory, because each `IntNode` no longer needs to keep track of how to access its enclosing `SLList`.

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. Make sure to read the comments at the top before doing the exercise.

addLast() and size()


Adding More SLList Functionality

To motivate our remaining improvements, and to give more functionality to our SLList class, let's add:

- .addLast(int x)
- .size()

Recommendati do it.

addFirst(int x)	#1	Rebranding: IntList → IntNode
getFirst	#2	Bureaucracy: SLList
	#3	Access Control: public → private
	#4	Nested Class: Bringing IntNode into SLList

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.

```
/** 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:

```
/** 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);
}
```

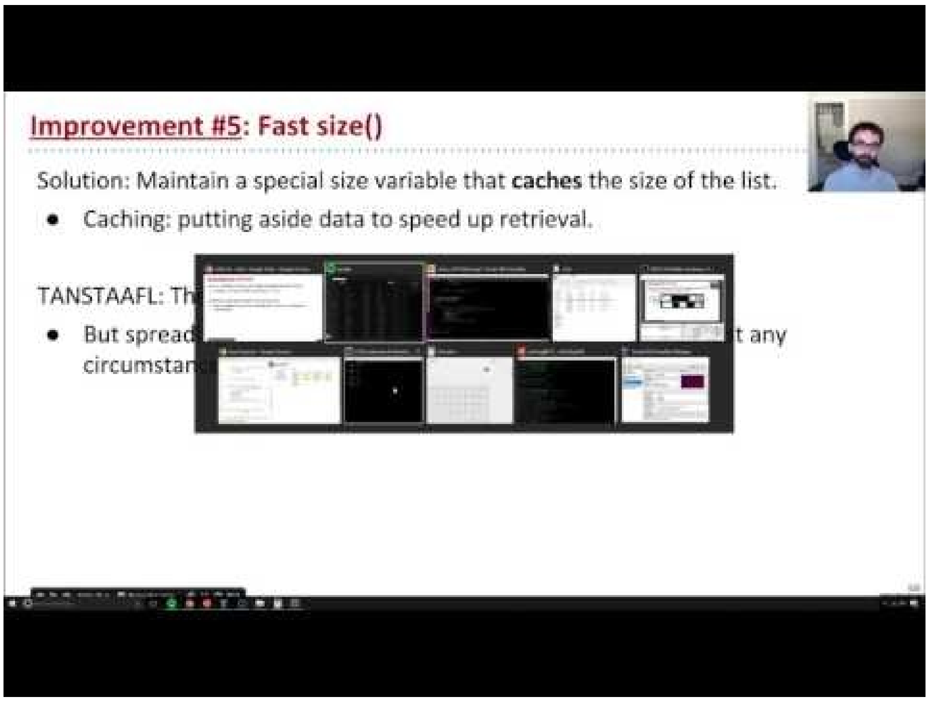
Using this method, we can easily compute the size of the entire list:

```
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


Improvement #5: Fast size()

Solution: Maintain a special size variable that **caches** the size of the list.

- Caching: putting aside data to speed up retrieval.

TANSTAAFL: Th

- But spread t any circumstan

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.

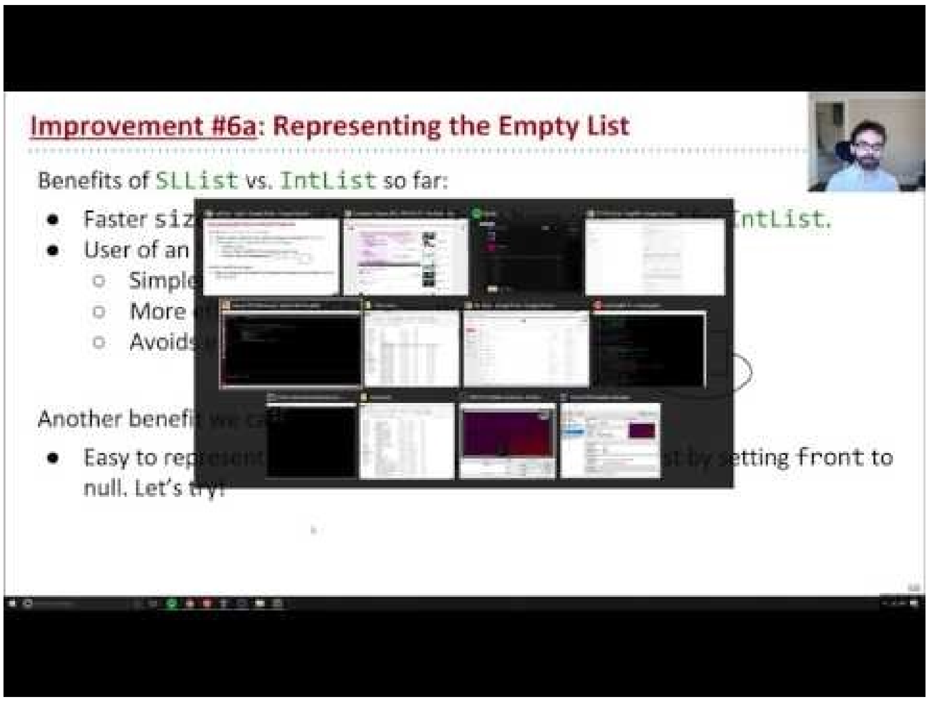
It is possible to rewrite `size` so that it takes the same amount of time, no matter how large the list.

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**.

```java
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



Improvement #6a: Representing the Empty List

Benefits of SLList vs. IntList so far:

- Faster siz IntList.
- User of an
  - Simple
  - More
  - Avoids

Another benefi we ca

- Easy to represent t by setting front to null. Let's try!

[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:

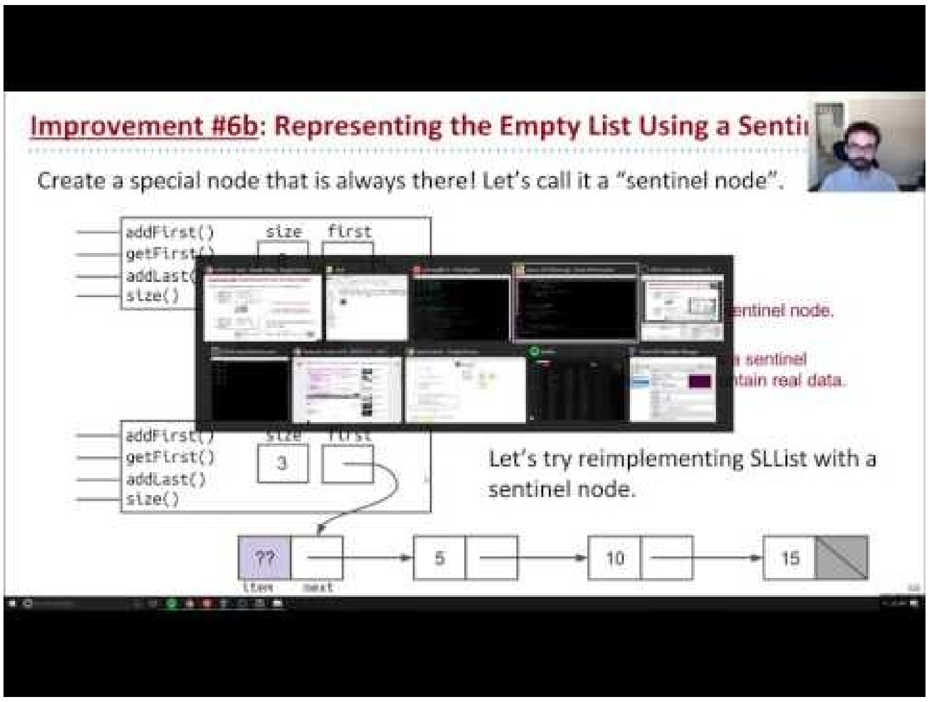
```
public SLList() {
    first = null;
    size = 0;
}
```

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 2.2.2: Fix the `addLast` method. Starter code here.

Improvement #6b: Sentinel Nodes


Improvement #6b: Representing the Empty List Using a Senti

Create a special node that is always there! Let's call it a "sentinel node".

Let's try reimplementing SLList with a sentinel node.

Video link

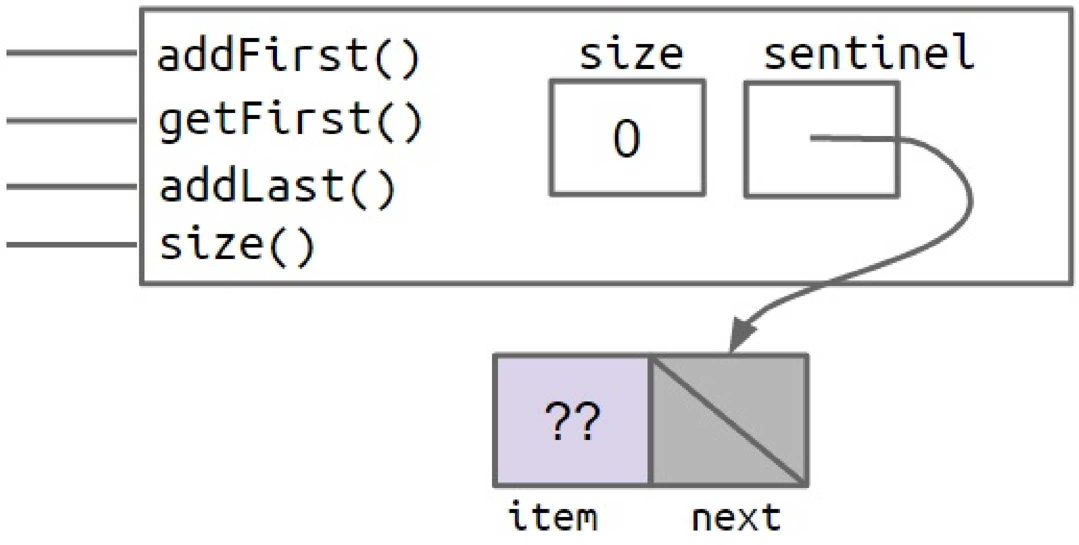
One solution to fix `addLast` is to create a special case for the empty list, as shown below:

```
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);
}
```

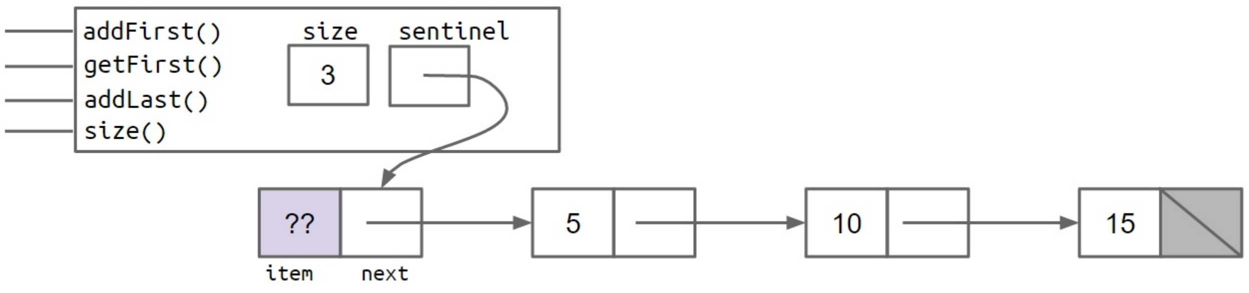
This solution works, but special case code like that shown above should be avoided when necessary. Human beings only have so much working memory, and thus we want to keep complexity under control wherever possible. For a simple data structure like the `SLList`, the number of special cases is small. More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



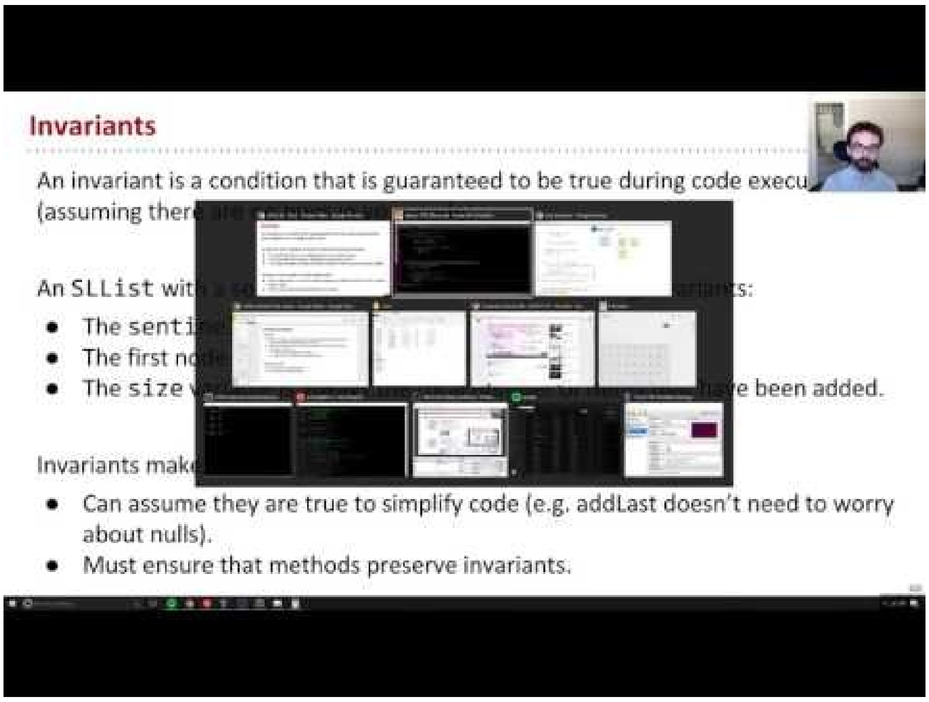
In the figures above, the lavender ?? value indicates that we don't care what value is there. Since Java does not allow us to fill in an integer with question marks, we just pick some abitrary value like -518273 or 63 or anything else.

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



Invariants

An invariant is a condition that is guaranteed to be true during code execu

(assuming ther

An SLList wit

- The senti
- The first n
- The size

e been added.

Invariants mak

- Can assume they are true to simplify code (e.g. addLast doesn't need to worry about nulls).
- Must ensure that methods preserve 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.

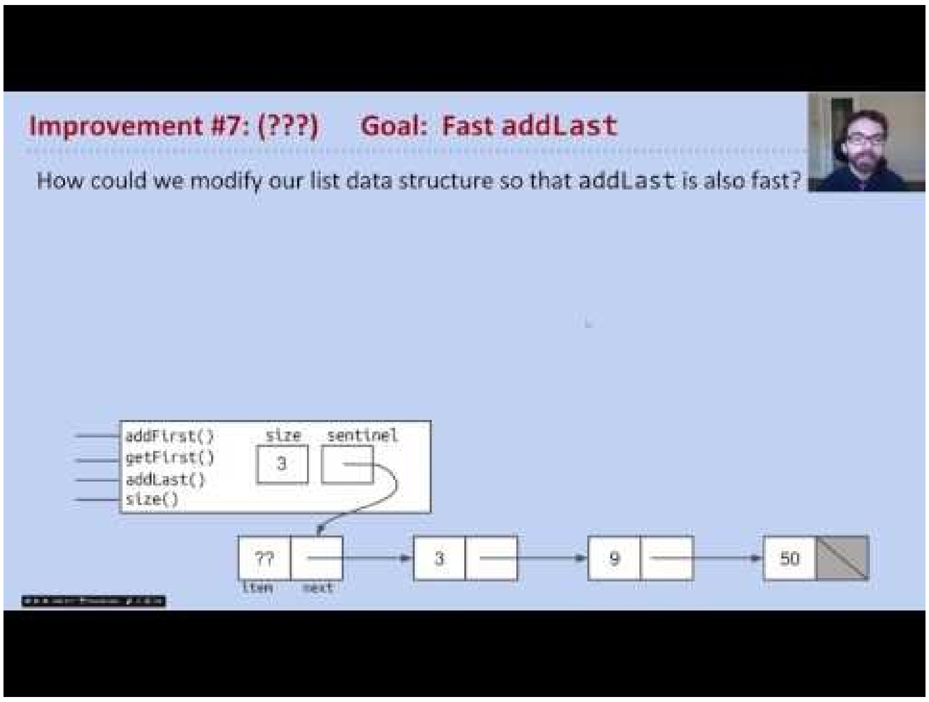
What Next

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



Improvement #7: (???) Goal: Fast addLast

How could we modify our list data structure so that addLast is also fast?

[Video link]

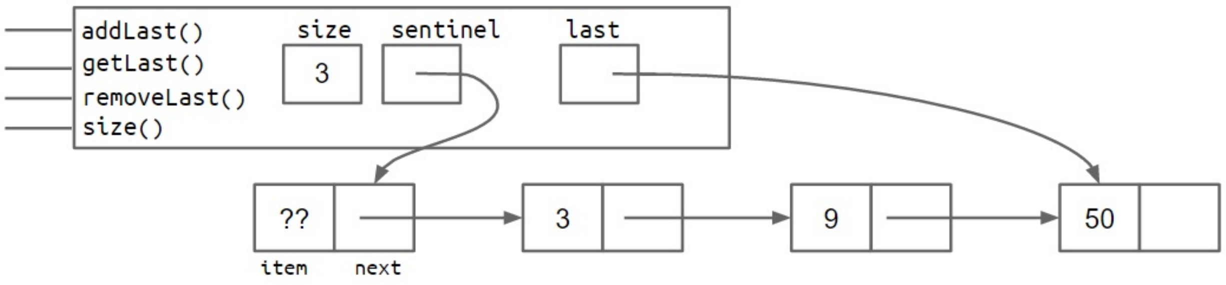
Consider the `addLast(int x)` method from the previous chapter.

```
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:

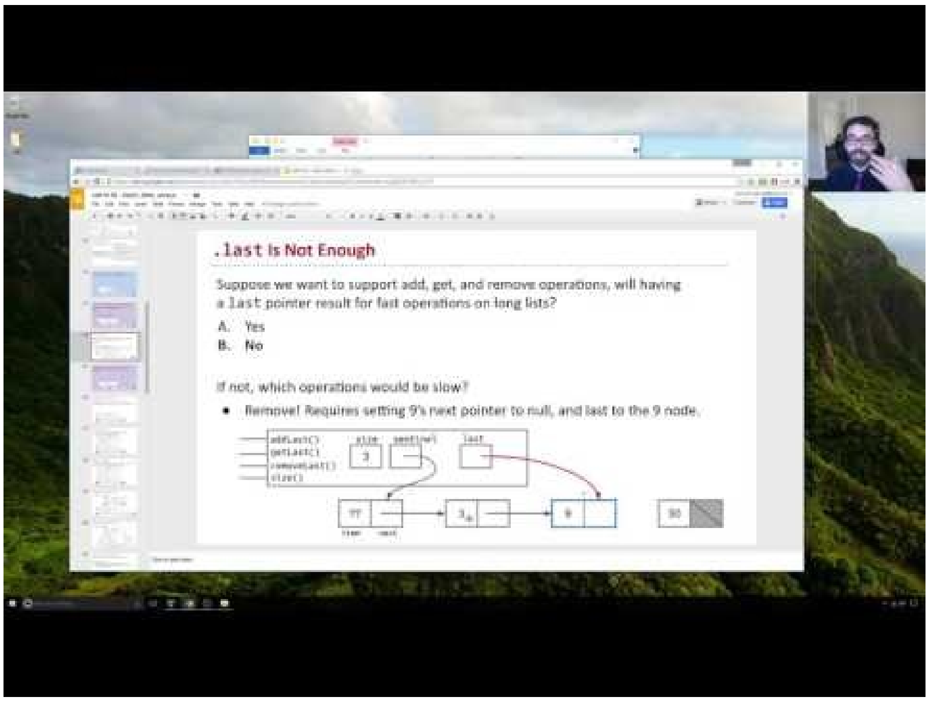
```
public class SLList {
    private IntNode sentinel;
    private IntNode last;
    private int size;

    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



.last Is Not Enough

Suppose we want to support add, get, and remove operations, will having a last pointer result for fast operations on long lists?

A. Yes
B. No

If not, which operations would be slow?

- Remove! Requires setting 9's next pointer to null, and last to the 9 node.

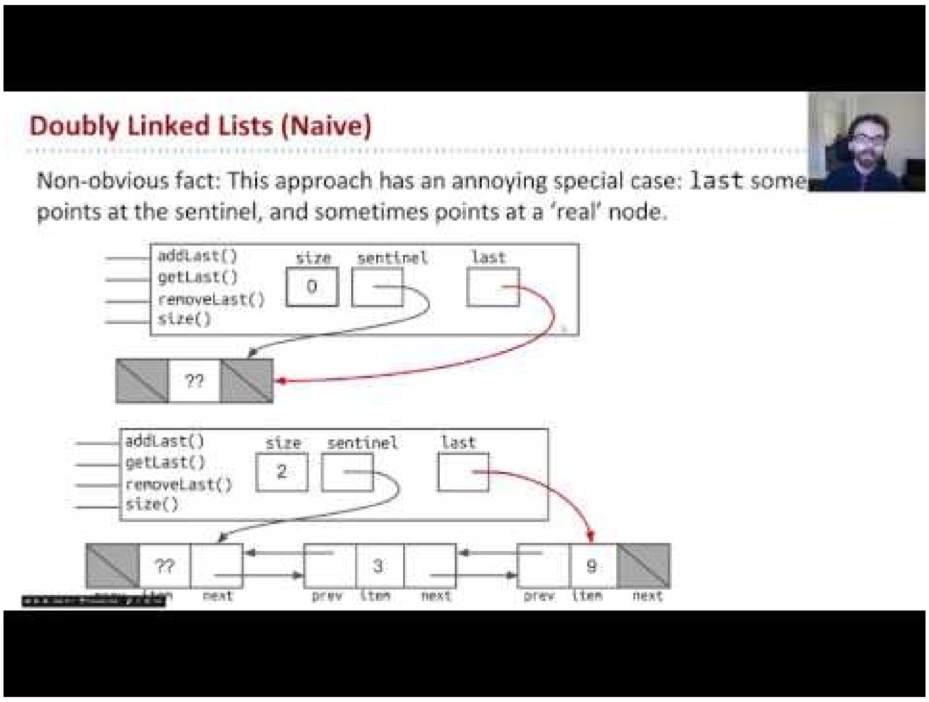
Video link

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


Doubly Linked Lists (Naive)

Non-obvious fact: This approach has an annoying special case: last some
points at the sentinel, and sometimes points at a 'real' node.

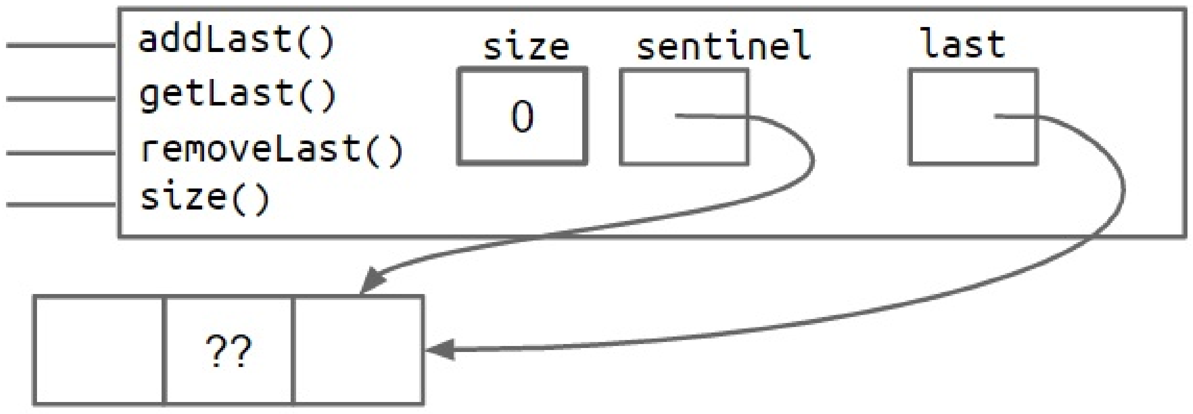
Video link

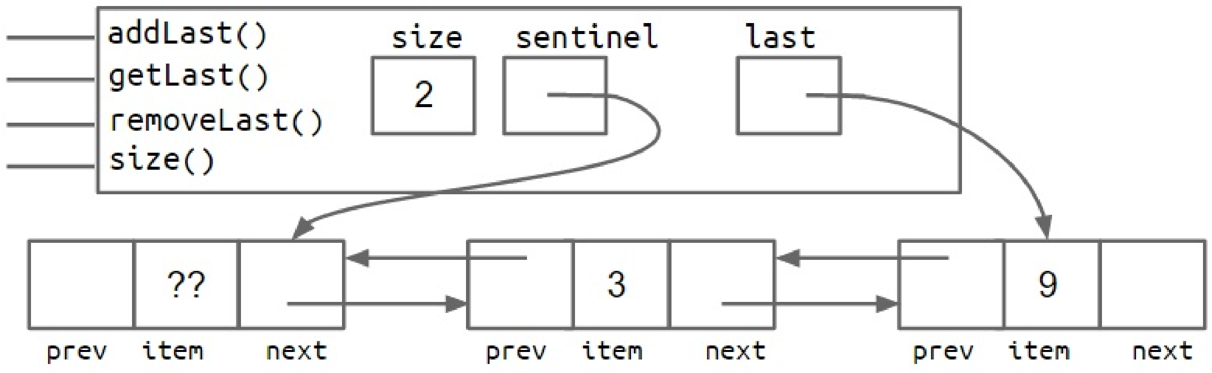
The most natural way to tackle this issue is to add a previous pointer to each `IntNode`, i.e.

```java
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 the chaper 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.

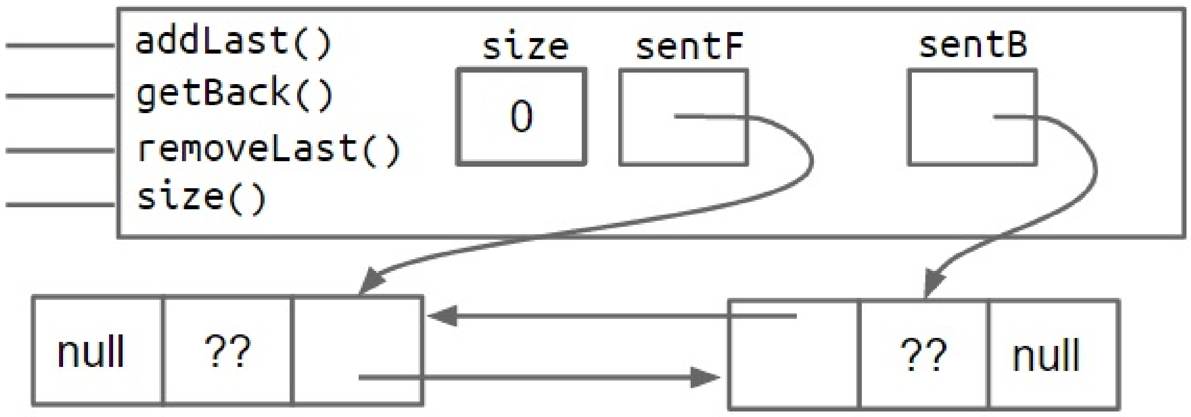


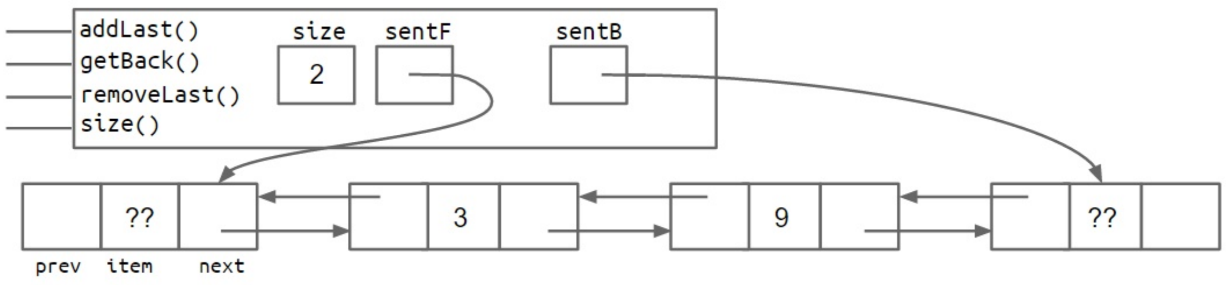


Improvement #8: Sentinel Upgrade

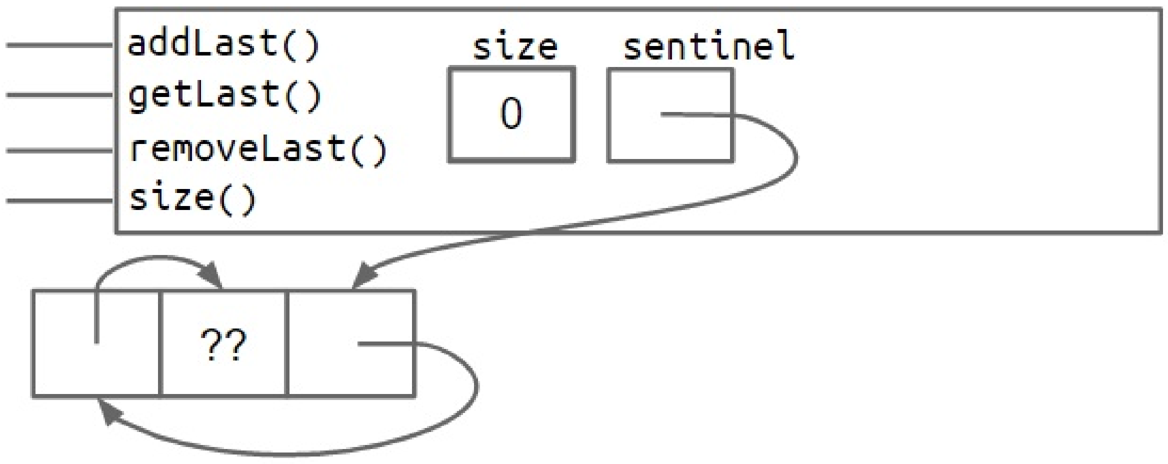
Back pointers allow a list to support adding, getting, and removing the front and back of a list in constant time. There is a subtle issue with this design where the `last` pointer sometimes points at the sentinel node, and sometimes at a real node. Just like the non-sentinel version of the `SLList`, 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?)

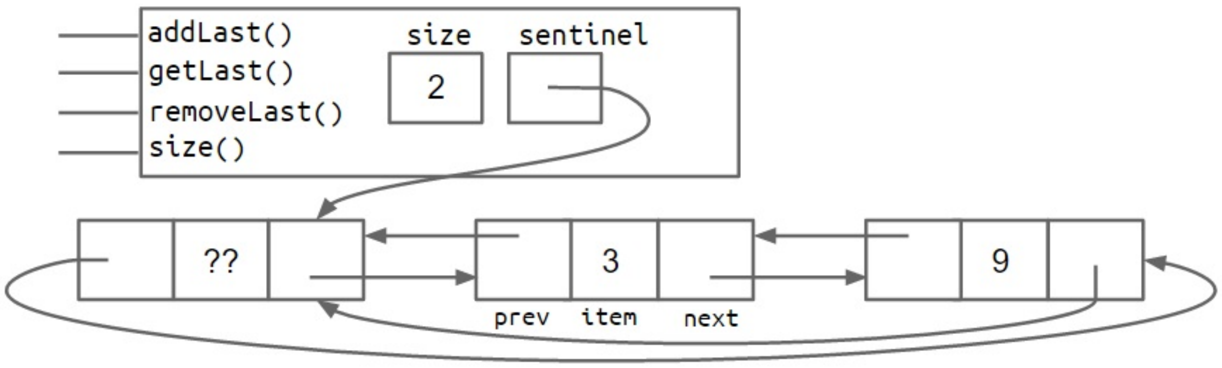
One fix is to add a second sentinel node to the back of the list. This results in the topology shown below as a box and pointer diagram.





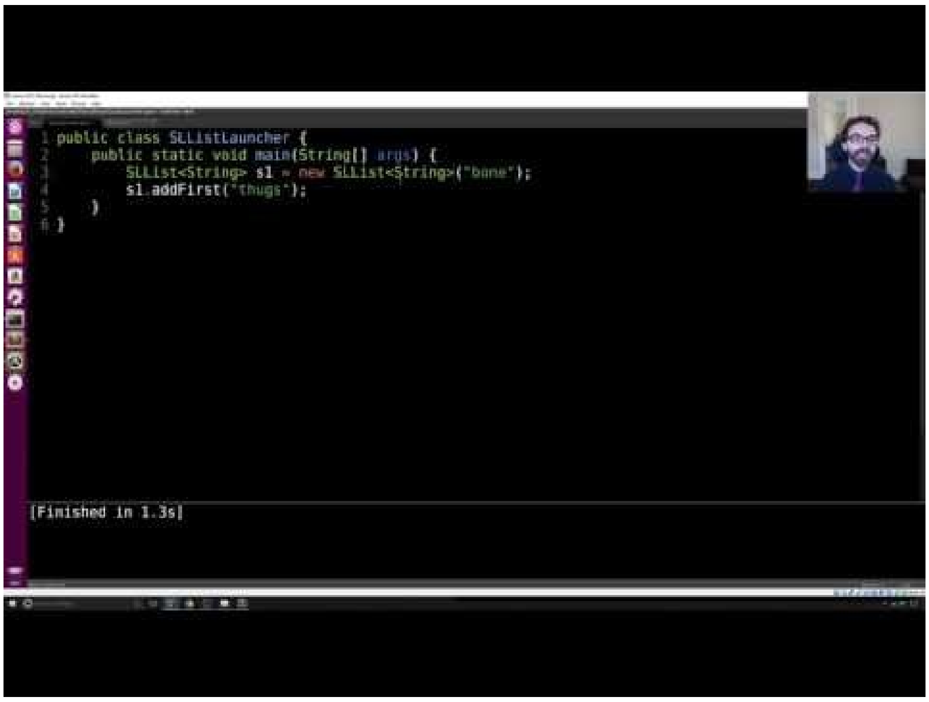
An alternate approach is to implement the list so that it is circular, with the front and back pointers sharing the same sentinel node.





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



```
public class SLListLauncher {
    public static void main(String[] args) {
        SLList<String> s1 = new SLList<String>("bone");
        s1.addFirst("thugs");
    }
}
```

```
[Finished in 1.3s]
```

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.

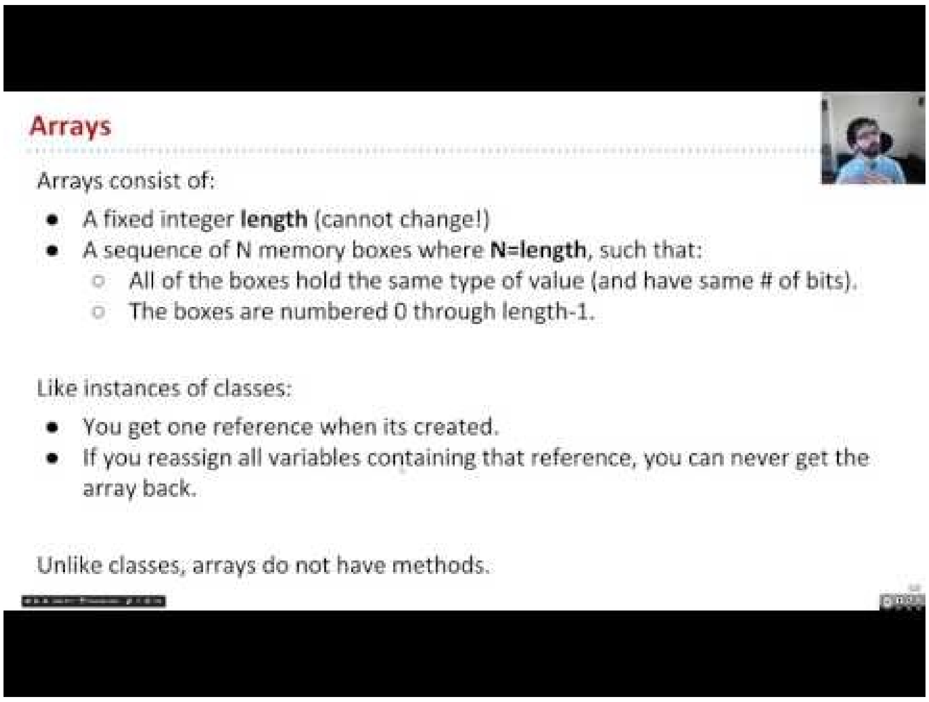
```
DLList<Integer> d1 = new DLList<Integer>(5);
```

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



Arrays

Arrays consist of:

- A fixed integer **length** (cannot change!)
- A sequence of N memory boxes where **N=length**, such that:
  - All of the boxes hold the same type of value (and have same # of bits).
  - The boxes are numbered 0 through length-1.

Like instances of classes:

- You get one reference when its created.
- If you reassign all variables containing that reference, you can never get the array back.

Unlike classes, arrays do not have methods.

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.

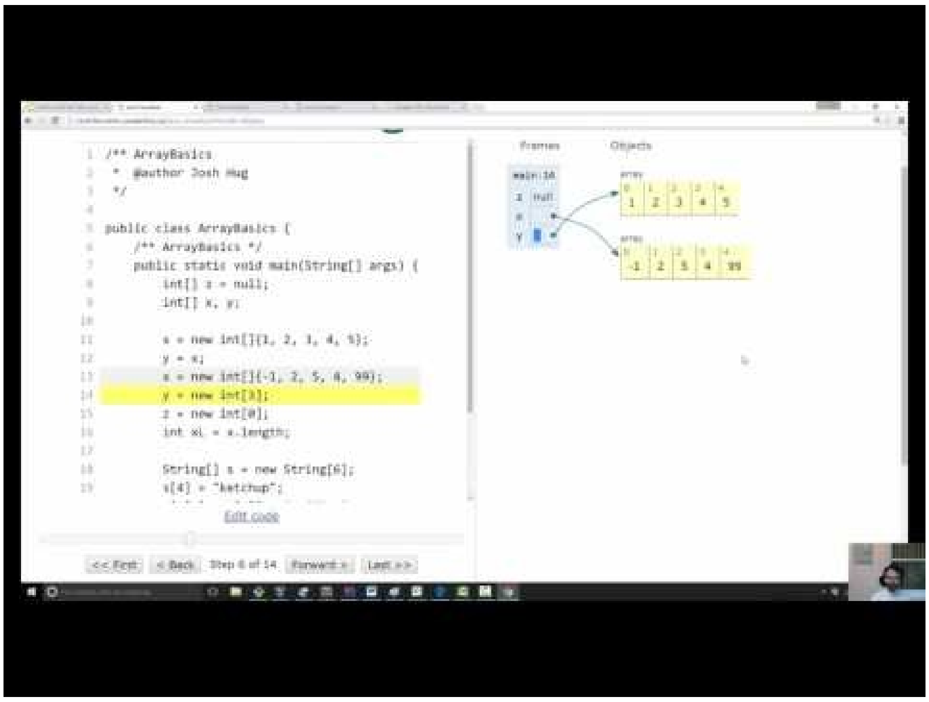
Arrays are a special type of object that consists of a numbered sequence of memory boxes. This is unlike class instances, which have named memory boxes. To get the ith item of an array, we use bracket notation as we saw in HW0 and Project 0, e.g. `A[i]` to get the ith element of A.

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



```
/** ArrayBasics
 *  @author Josh Hug
 */

public class ArrayBasics {
    /** ArrayBasics */
    public static void main(String[] args) {
        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";
```

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 accomodate 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

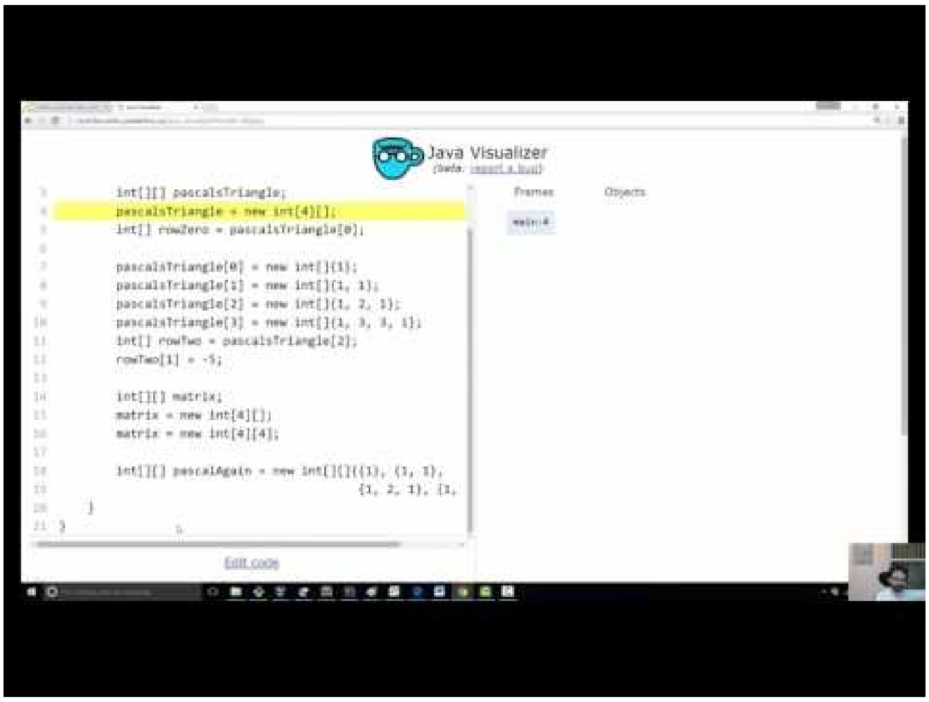
For Python veterans, `System.arraycopy(b, 0,x, 3, 2)` is the equivalent of `x[3:5] = b[0:2]` in Python.

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.

```java
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



```java
        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,
    }
}
```

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).

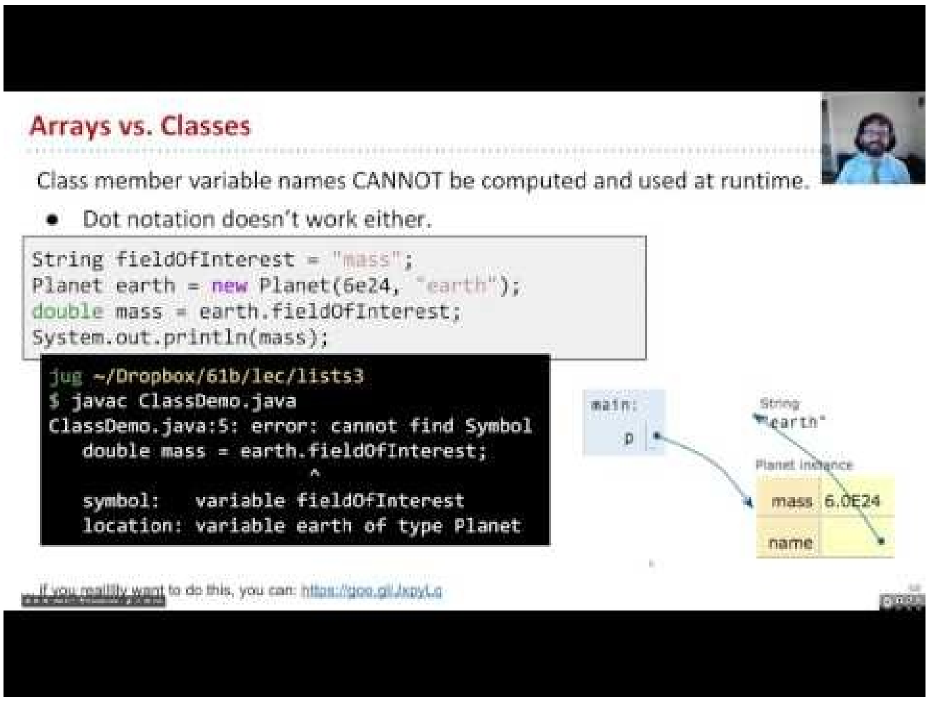
Try running the code below line-by-lines, and see if the results match your intuition. For an interactive visualization, click here.

```
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



Arrays vs. Classes

Class member variable names CANNOT be computed and used at runtime.

- Dot notation doesn't work either.

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

```
jug ~/Dropbox/61b/lec/lists3
$ javac ClassDemo.java
ClassDemo.java:5: error: cannot find Symbol
   double mass = earth.fieldOfInterest;
                      ^
   symbol:   variable fieldOfInterest
   location: variable earth of type Planet
```

If you reallllly want to do this, you can: https://goo.gl/JxpyLq

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

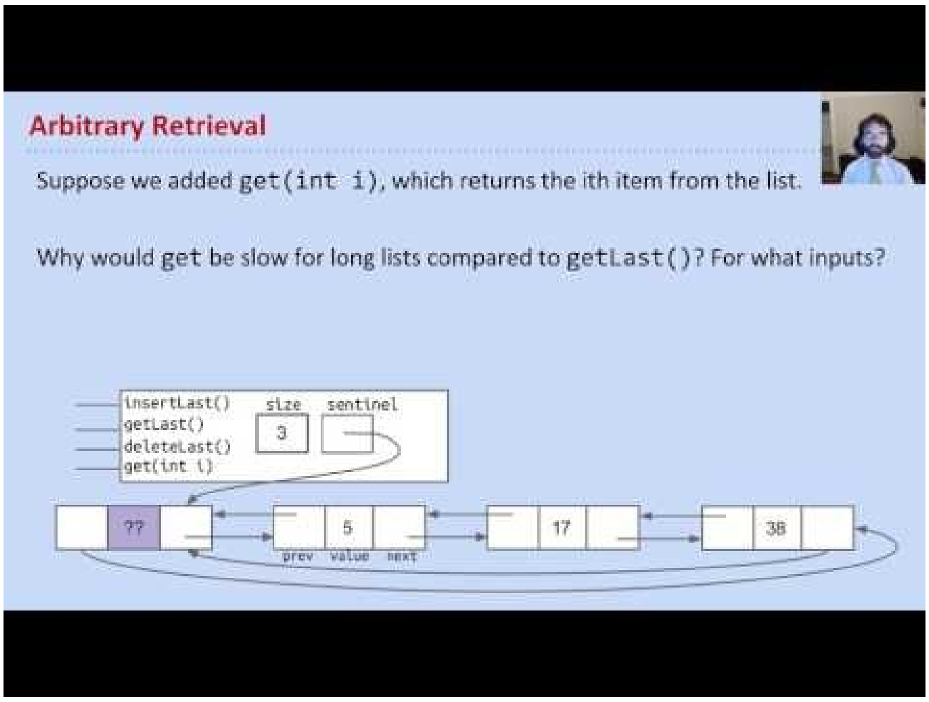
Compared to arrays in other languages, Java arrays:

- 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



Arbitrary Retrieval

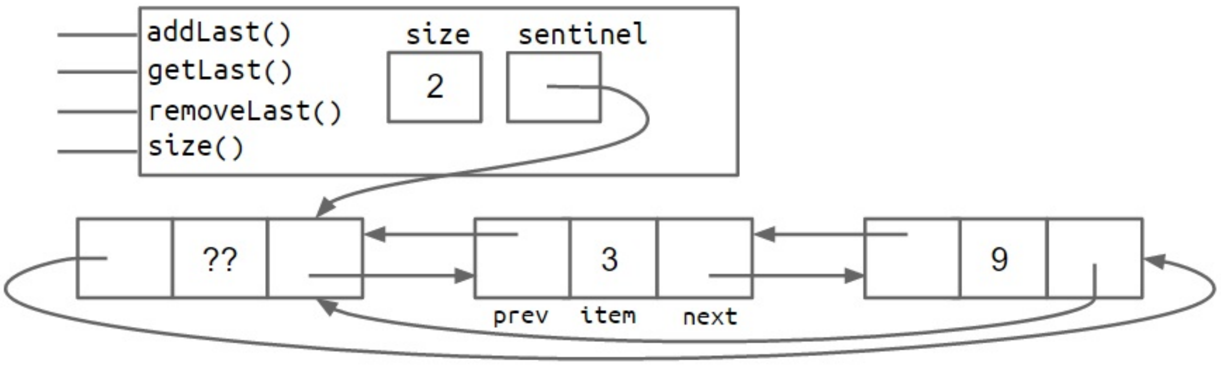
Suppose we added `get(int i)`, which returns the ith item from the list.

Why would `get` be slow for long lists compared to `getLast()`? For what inputs?

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 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.

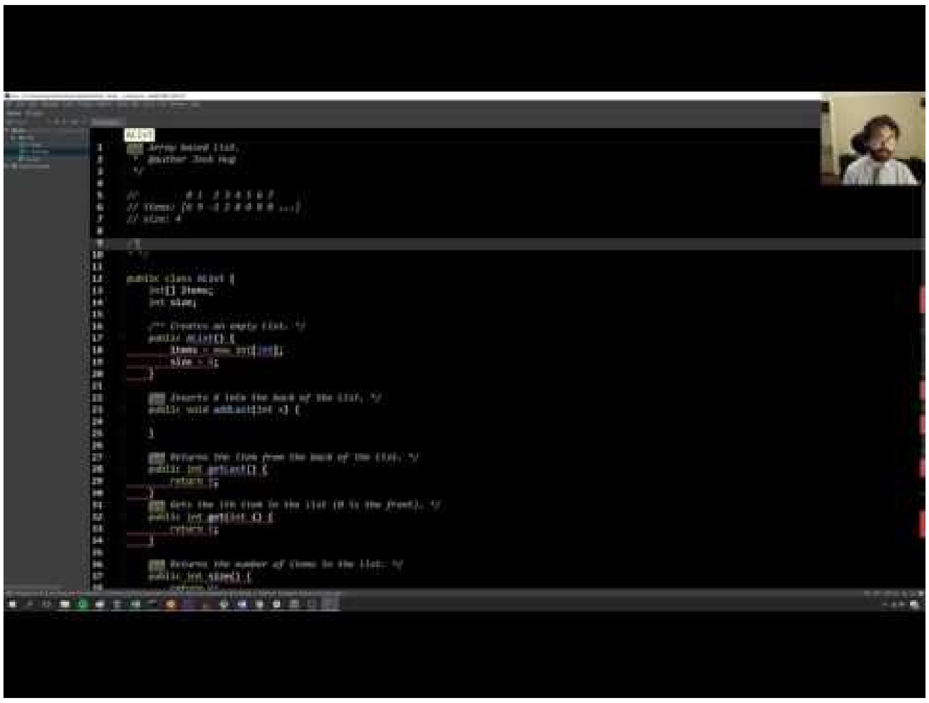
In the very worst case, the item is in the very middle and we'll need to walk through a number of items proportional to the length of the list (specifically, the number of items divided by two). In other words, our worst case execution time for `get` is linear in the size of the entire list. This in 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 ith 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.



```
/** Array based list.
 *  @author Josh Hug
 */

//          [illegible]
// items: [illegible]
// size: 4

/*
 */

public class AList {
    int[] items;
    int size;

    /** Creates an empty list. */
    public AList() {
        items = new int[100];
        size = 0;
    }

    /** Inserts X into the back of the list. */
    public void addLast(int x) {

    }

    /** Returns the item from the back of the list. */
    public int getLast() {
        return 0;
    }
    /** Gets the ith item in the list (0 is the front). */
    public int get(int i) {
        return 0;
    }

    /** Returns the number of items in the list. */
    public int size() {
```

Video link

My solution has the following handy invariants.

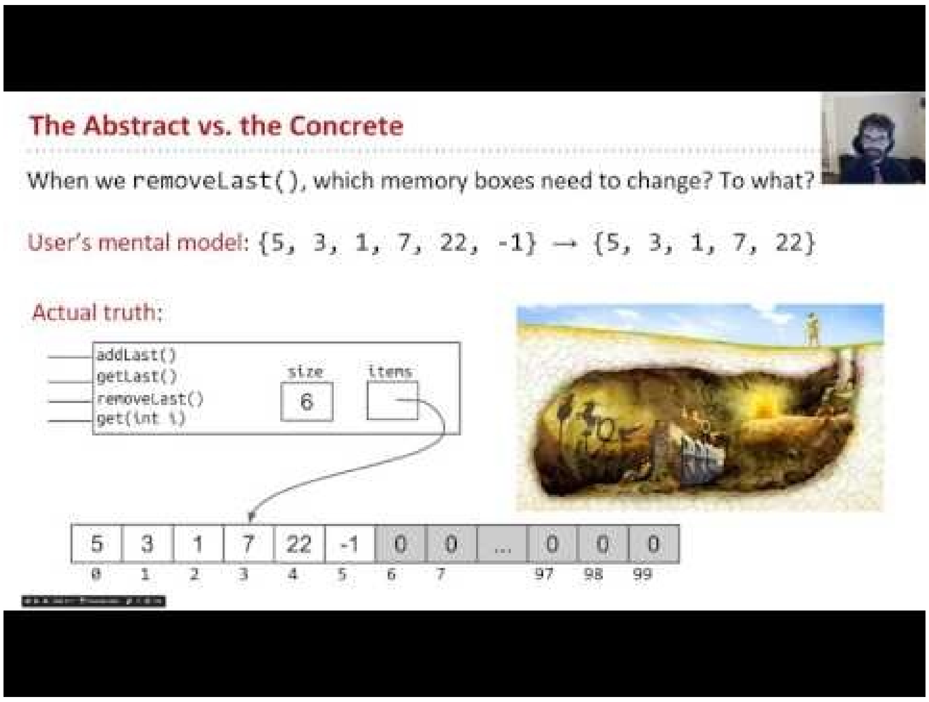
- The position of the next item to be inserted (using `addLast`) is always `size`.
- The number of items in the AList is always `size`.
- The position of the last item in the list is always `size - 1`.

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.



The Abstract vs. the Concrete

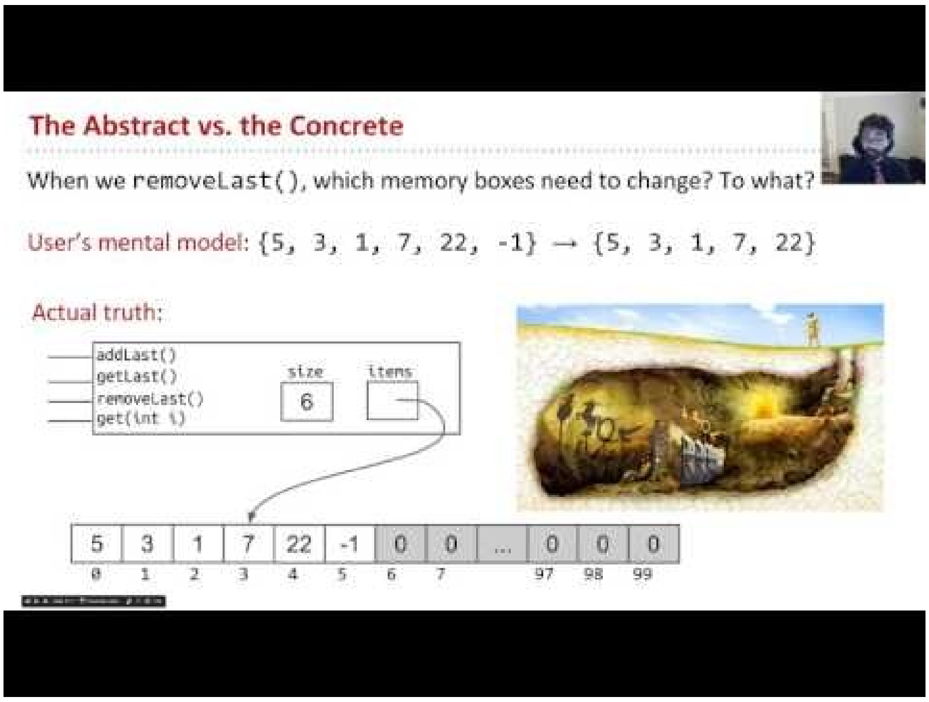
When we removeLast(), which memory boxes need to change? To what?

User's mental model: {5, 3, 1, 7, 22, -1} → {5, 3, 1, 7, 22}

Actual truth:

[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.



The Abstract vs. the Concrete

When we removeLast(), which memory boxes need to change? To what?

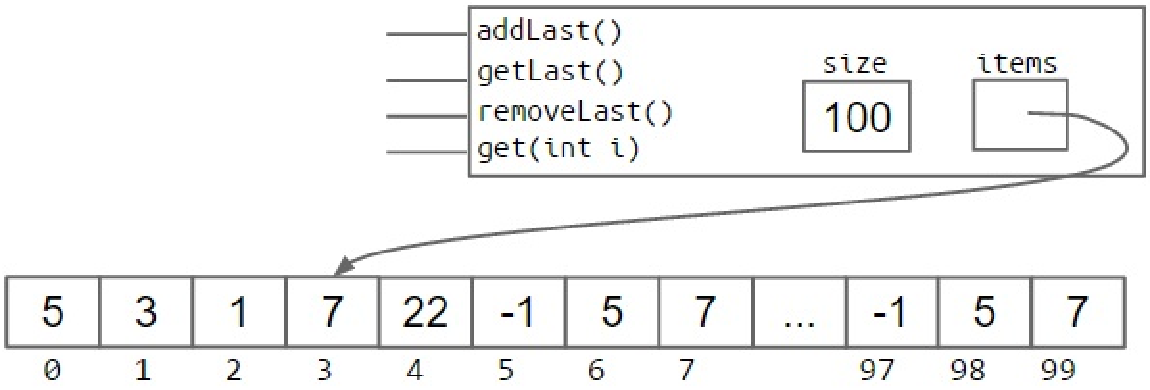
User's mental model: {5, 3, 1, 7, 22, -1} → {5, 3, 1, 7, 22}

Actual truth:

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?



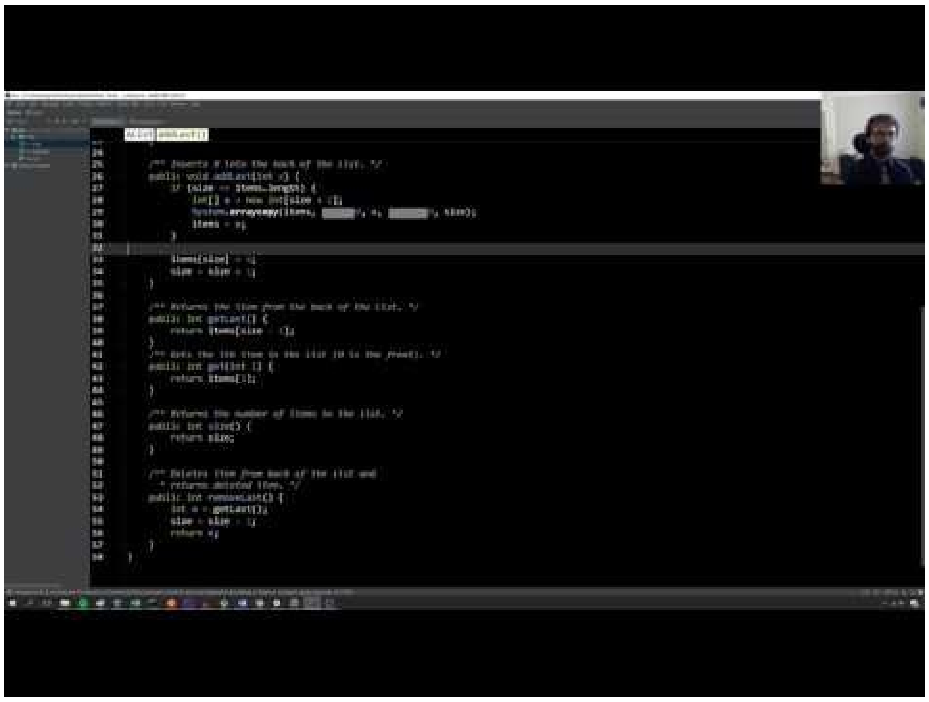
Video link

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.

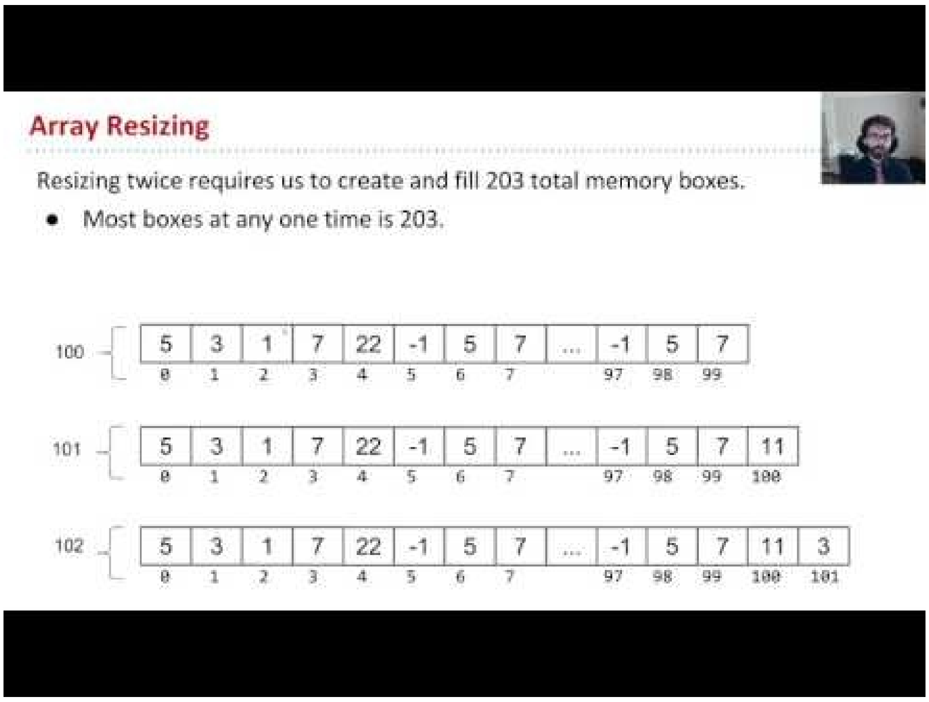
Exercise 2.5.4: Try to implement the `addLast(int i)` method to work with resizing arrays.



```
    /** Inserts X into the back of the list. */
    public void addLast(int x) {
        if (size == items.length) {
            int[] a = new int[size + 1];
            System.arraycopy(items, [illegible], a, [illegible], size);
            items = a;
        }

        items[size] = x;
        size = size + 1;
    }

    /** Returns the item from the back of the list. */
    public int getLast() {
        return items[size - 1];
    }
    /** Gets the ith item in the list (0 is the front). */
    public int get(int i) {
        return items[i];
    }

    /** Returns the number of items in the list. */
    public int size() {
        return size;
    }

    /** Deletes item from back of the list and
      * returns deleted item. */
    public int removeLast() {
        int x = getLast();
        size = size - 1;
        return x;
    }
}
```

Video link

Analyzing the Naive Resizing Array



Array Resizing

Resizing twice requires us to create and fill 203 total memory boxes.

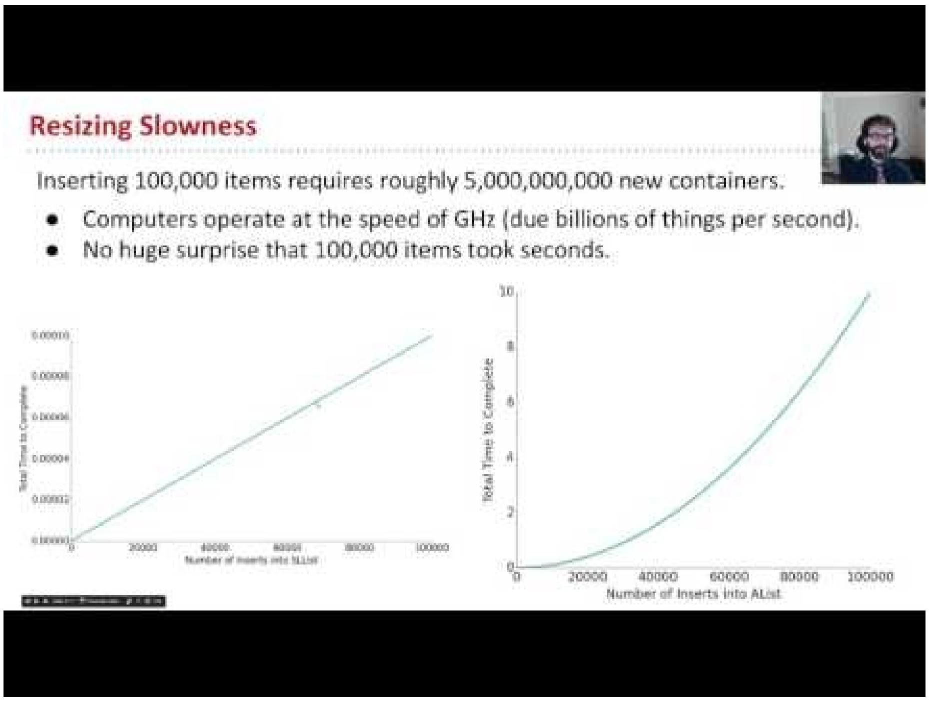
- Most boxes at any one time is 203.

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?


Resizing Slowness

Inserting 100,000 items requires roughly 5,000,000,000 new containers.

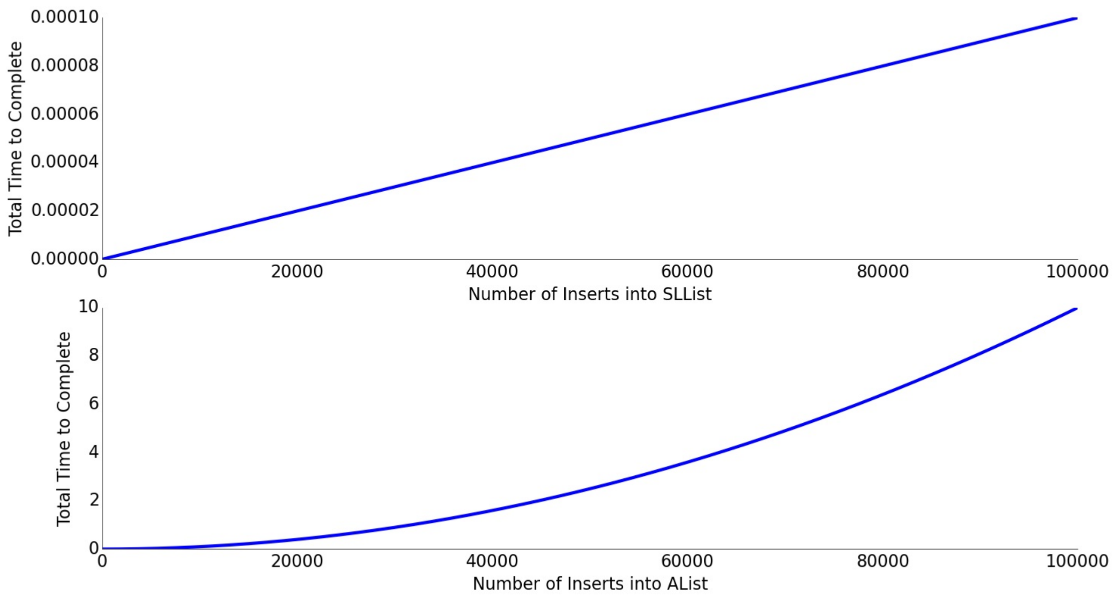
- Computers operate at the speed of GHz (due billions of things per second).
- No huge surprise that 100,000 items took seconds.

[Video link](#)

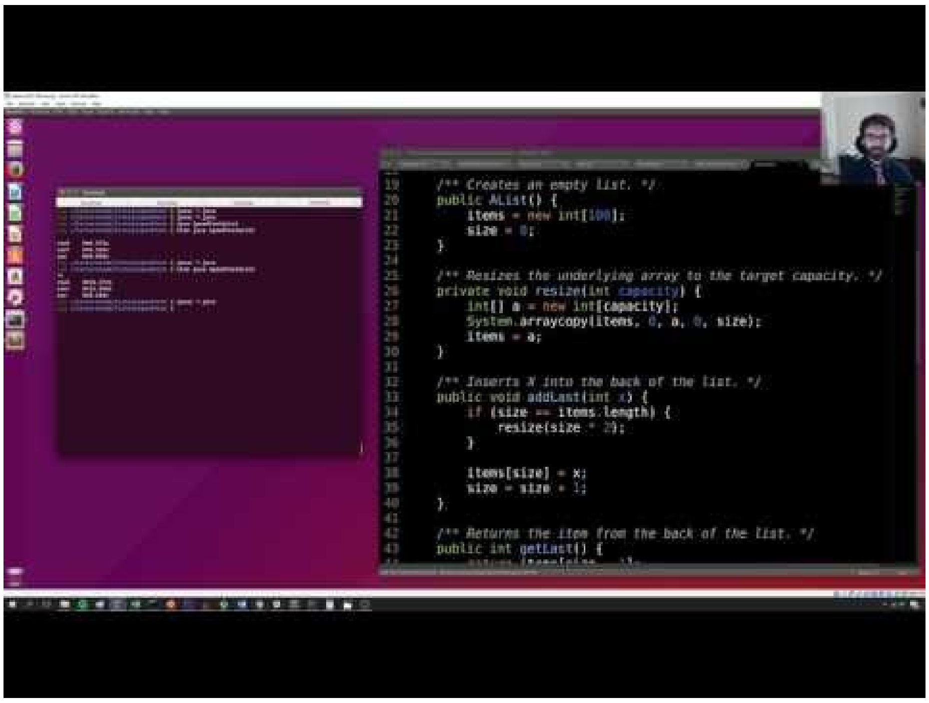
Exercise 2.5.6: Starting from an array of size 100, approximately how many memory boxes get created and filled if we call `addLast` 1,000 times?

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



```
    /** Creates an empty list. */
    public AList() {
        items = new int[100];
        size = 0;
    }

    /** Resizes the underlying array to the target capacity. */
    private void resize(int capacity) {
        int[] a = new int[capacity];
        System.arraycopy(items, 0, a, 0, size);
        items = a;
    }

    /** Inserts X into the back of the list. */
    public void addLast(int x) {
        if (size == items.length) {
            resize(size * 2);
        }

        items[size] = x;
        size = size + 1;
    }

    /** Returns the item from the back of the list. */
    public int getLast() {
```

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` .

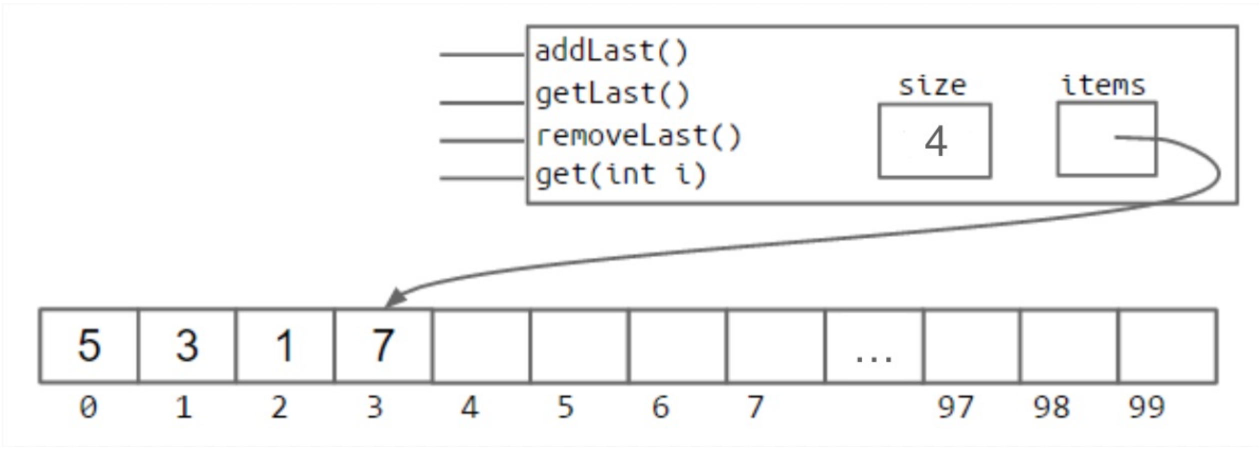
```
public void insertBack(int x) {
    if (size == items.length) {
           resize(size * RFACTOR);
    }
    items[size] = x;
    size += 1;
}
```

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 TBA
- Lab2

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

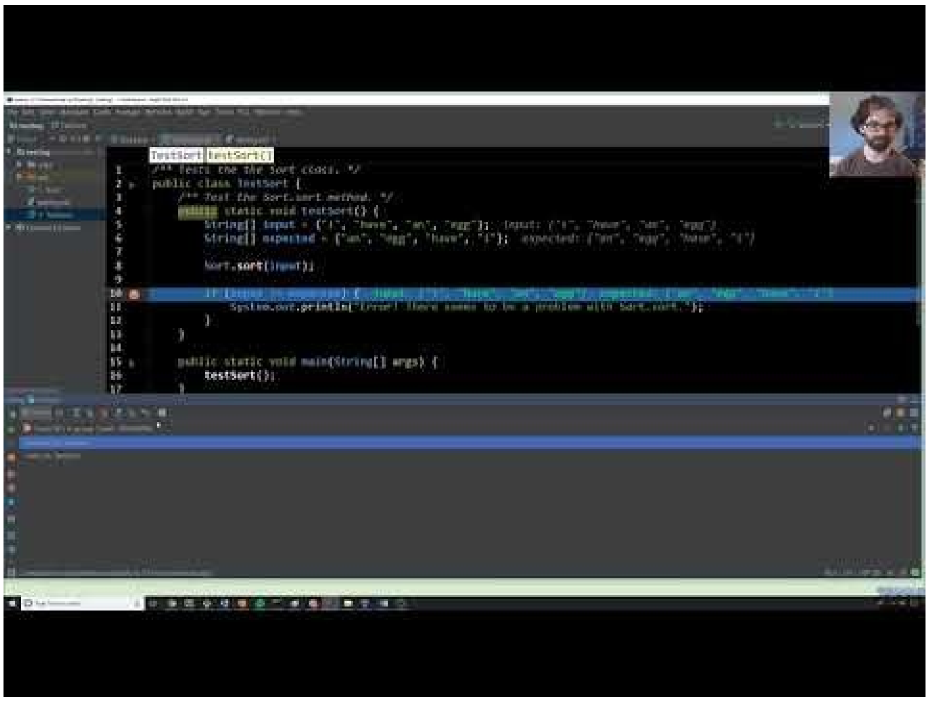
When you write a program, it may have errors. In a classroom setting, you gain confidence in your code's correctness through some combination of user interaction, code analysis, and autograder testing, with this last item being of the greatest importance in many cases, particularly as it is how you earn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



```
/** Tests the the Sort class. */
public class TestSort {
    /** Test the Sort.sort method. */
    public static void testSort() {
        String[] input = {"i", "have", "an", "egg"};
        String[] expected = {"an", "egg", "have", "i"};

        Sort.sort(input);

        if ([illegible]) {
            System.out.println("Error! There seems to be a problem with Sort.sort.");
        }
    }

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

[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:

```
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:

```
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.
```

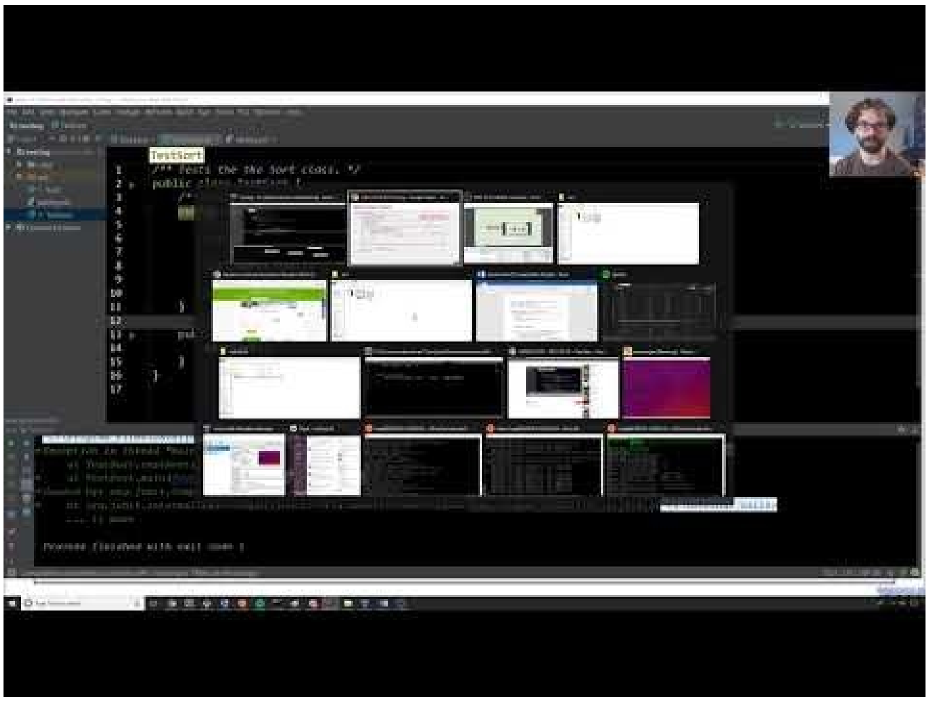
The fact that we're getting an error message is a good thing! This means our test is working. What's very interesting about this is that we've now created a little game for ourselves to play, where the goal is to modify the code for `Sort.sort` so that this error message no longer occurs. It's a bit 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:

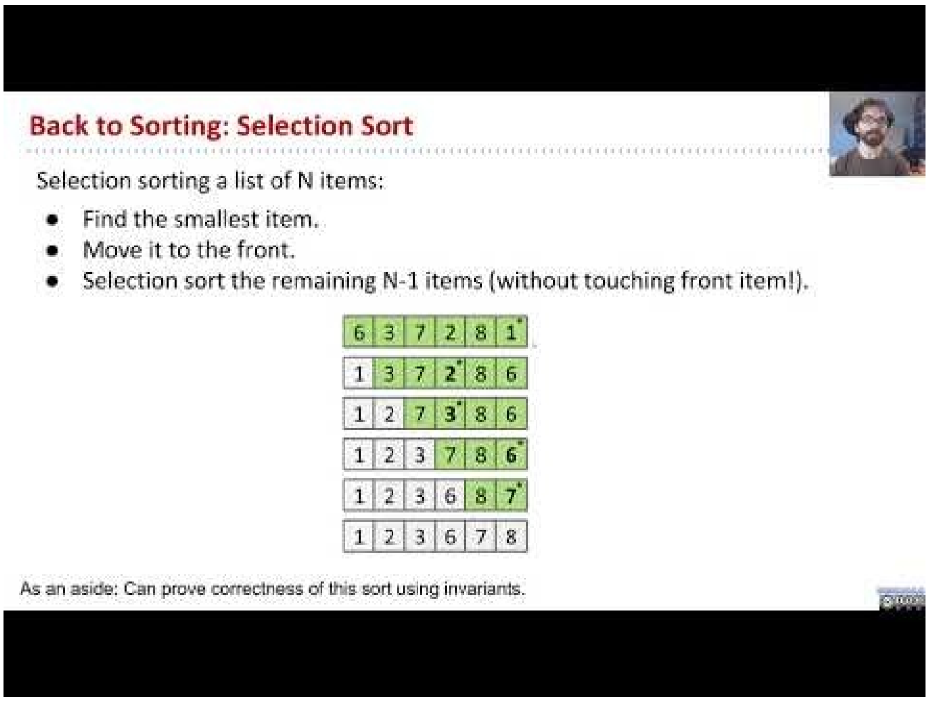
```java
public static void testSort() {
    String[] input = {"i", "have", "an", "egg"};
    String[] expected = {"an", "egg", "have", "i"};
    Sort.sort(input);
    org.junit.Assert.assertArrayEquals(expected, input);
}
```

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



Back to Sorting: Selection Sort

Selection sorting a list of N items:

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

```
6 3 7 2 8 1*
1 3 7 2* 8 6
1 2 7 3* 8 6
1 2 3 7 8 6*
1 2 3 6 8 7*
1 2 3 6 7 8
```

As an aside: Can prove correctness of this sort using invariants.

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}`.

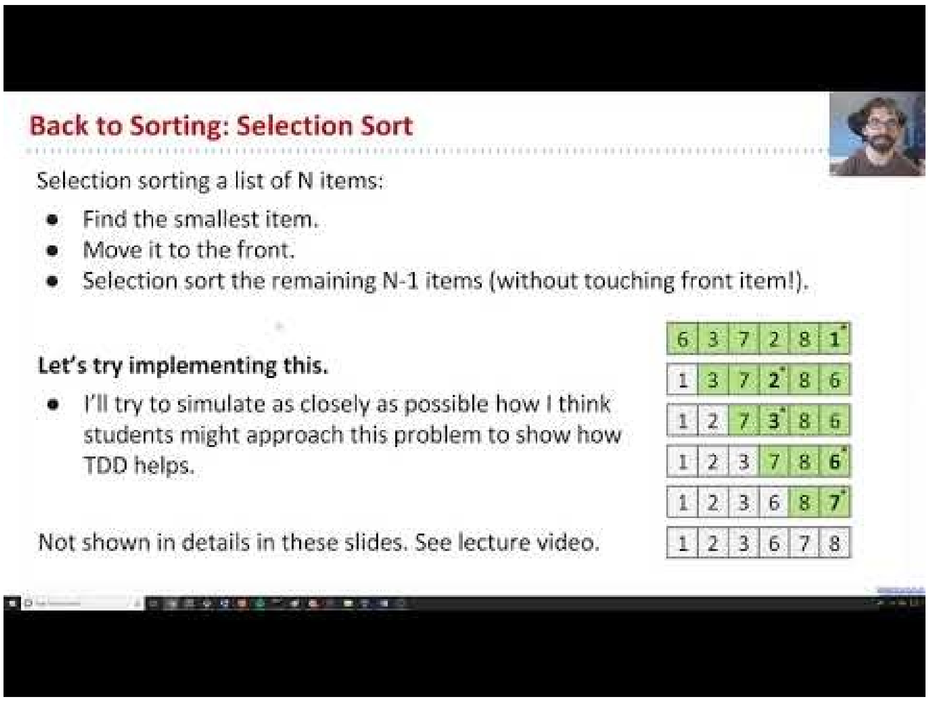
We could mathematically prove the correctness of this sorting algorithm on any arrays by using the concept of invariants that was originally introduced in chapter 2.4, though we will not do so in this textbook. Before proceeding, try writing out your own short array of numbers and perform selection sort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



Back to Sorting: Selection Sort

Selection sorting a list of N items:

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

Let's try implementing this.

- I'll try to simulate as closely as possible how I think students might approach this problem to show how TDD helps.

Not shown in details in these slides. See lecture video.

6	3	7	2	8	1*
1	3	7	2*	8	6
1	2	7	3*	8	6
1	2	3	7	8	6*
1	2	3	6	8	7*
1	2	3	6	7	8

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:

```java
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.

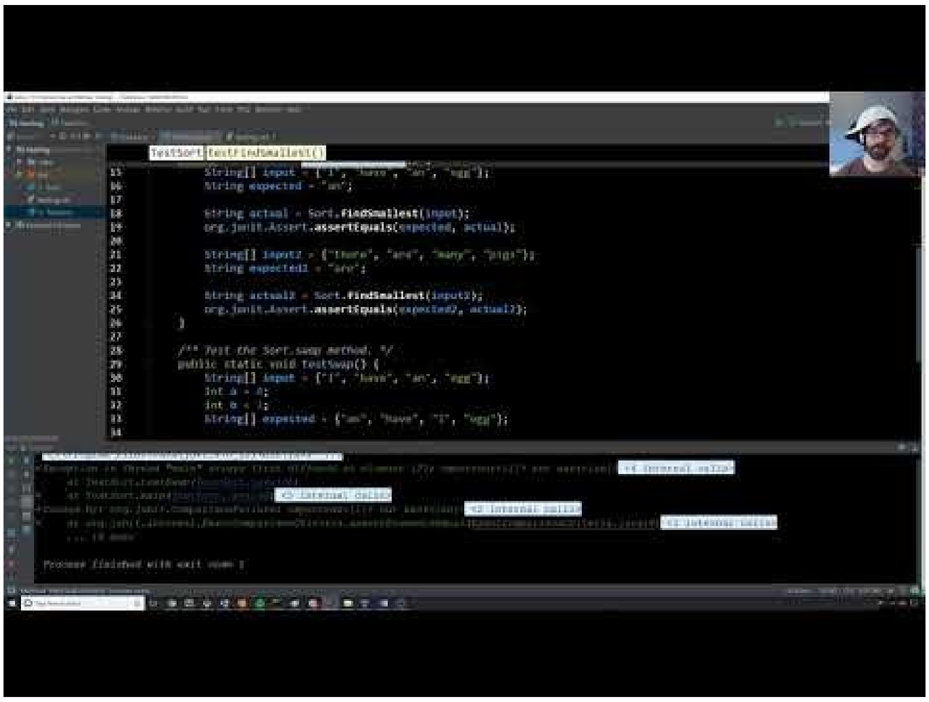
Since we are using syntax features that are totally new to us, we might lack confidence in the correctness of our `findSmallest` method. Luckily, we just wrote that test a little while ago. If we try running it, we'll see that nothing gets printed, which means our code is probably correct.

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:

```java
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



```java
        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);
    }

    /** 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"};
```

```
Exception in thread "main" [illegible] <4 internal calls>
    at TestSort.testSwap([illegible])
    at TestSort.main([illegible]) <5 internal calls>
Caused by: org.junit.ComparisonFailure: [illegible] <2 internal calls>
    at [illegible] <2 internal calls>
    ... [illegible] more

Process finished with exit code 1
```

[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`.

```
/** 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:

```
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:

```
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`.

```
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();
}
```

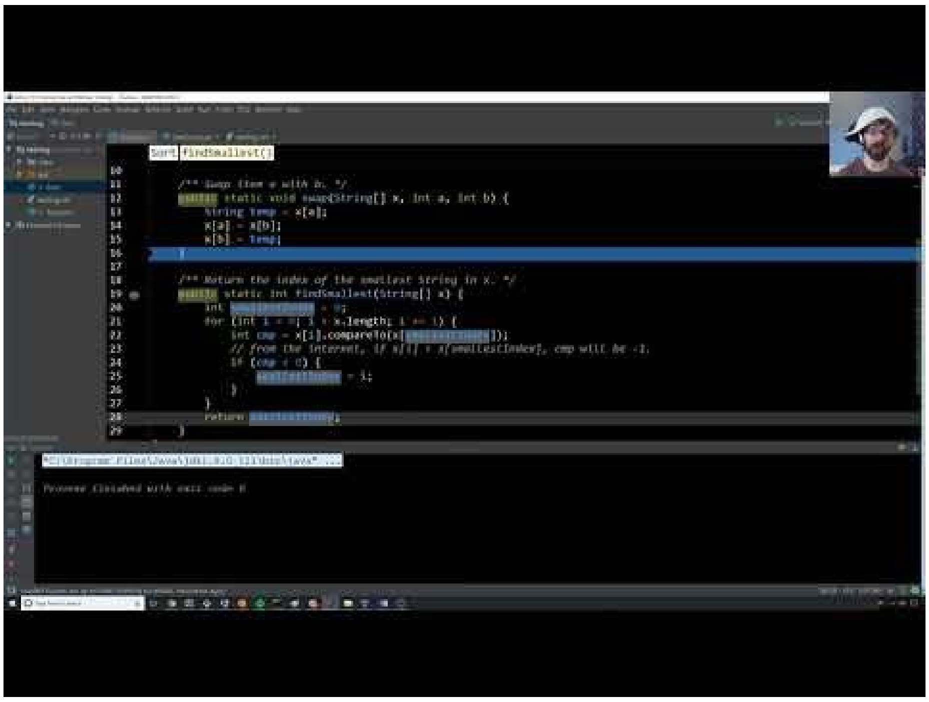
We will learn a more elegant way to deal with multiple tests at the end of this chapter that will avoid the need to manually specify which tests to run.

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 Lab2. 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



```
/** Swap item a with b. */
public static void swap(String[] x, int a, int b) {
    String temp = x[a];
    x[a] = x[b];
    x[b] = temp;
}

/** Return the index of the smallest String in x. */
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]);
        // from the internet, if x[i] < x[smallestIndex], cmp will be -1.
        if (cmp < 0) {
            smallestIndex = i;
        }
    }
    return smallestIndex;
}
```

```
Process finished with exit code 0
```

[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);
}
```

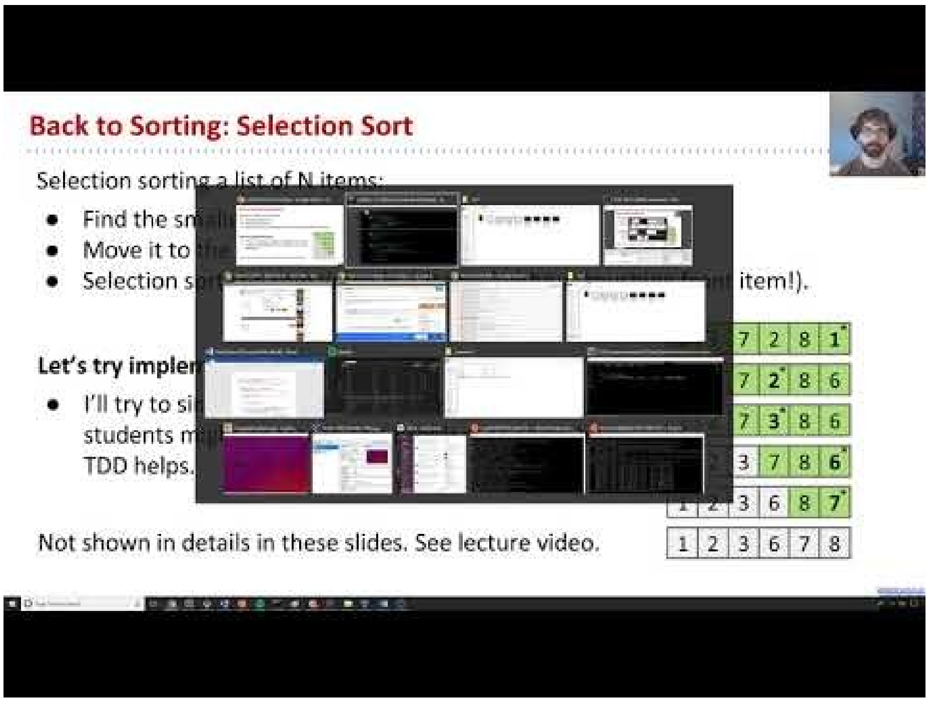
After modifying `TestSort` so that this test is run, and running `TestSort.main` , we see that our code passes the tests. Now, revising sort, we can fill in the first two steps of our sorting algorithm.

```
/** 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. In the meantime, in the figure below, we have detailed the evolution of our design. 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.

FIGURECOMINGSOON

Recursive Helper Methods



Back to Sorting: Selection Sort

Selection sorting a list of N items:

- Find the sm
- Move it to the
- Selection s item!).

Let's try implem

- I'll try to si students m TDD helps.

Not shown in details in these slides. See lecture video.

Video link

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

```
/** 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.

```
/** 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` .

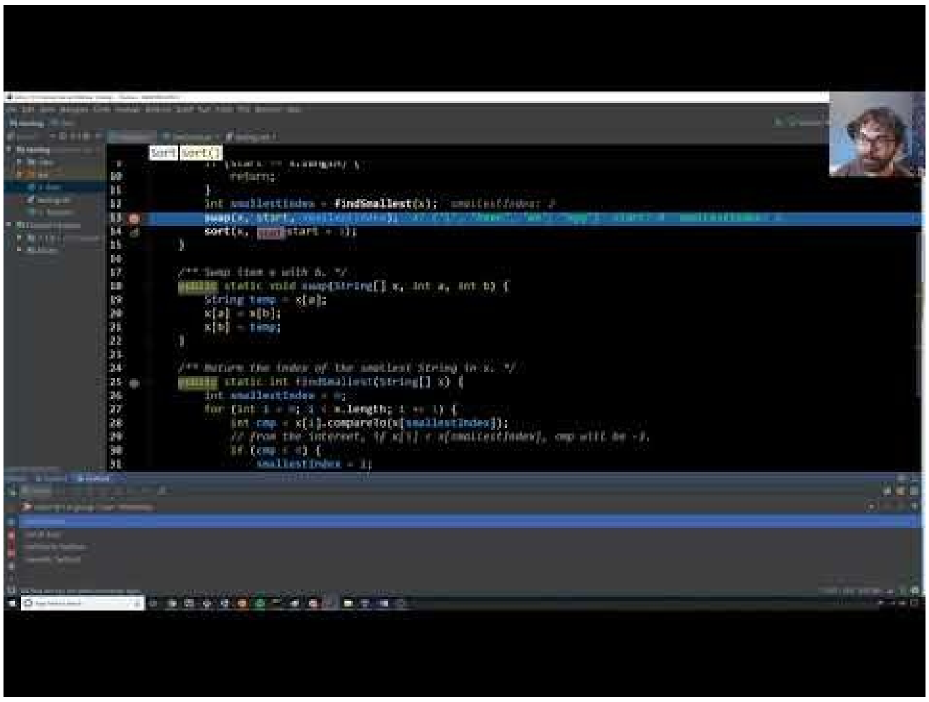
```
/** Sorts strings destructively starting from item start. */
private static void sort(String[] x, int start) {
    // TODO
}
```

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.

```
/** Sorts strings destructively starting from item start. */
private static void sort(String[] x, int start) {
   int smallestIndex = findSmallest(x);
   swap(x, 0, smallestIndex);
   sort(x, start + 1);
}
```

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

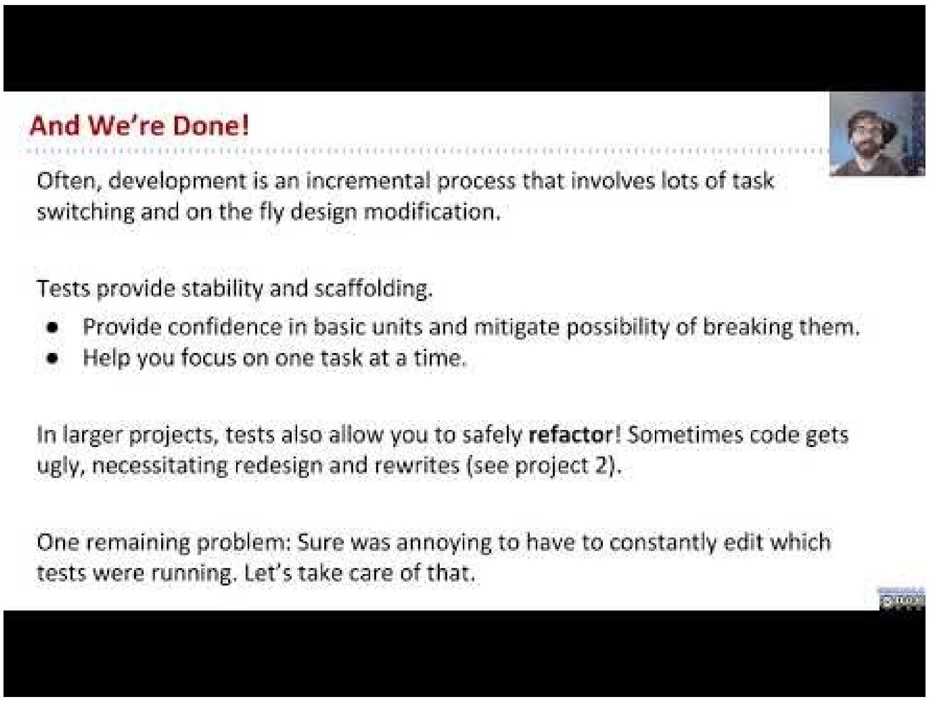
Debugging and Completing Sort



```
        if (start == x.length) {
            return;
        }
        int smallestIndex = findSmallest(x);   smallestIndex: 2
        swap(x, start, smallestIndex);   [illegible]
        sort(x, start + 1);
    }

    /** Swap item a with b. */
    public static void swap(String[] x, int a, int b) {
        String temp = x[a];
        x[a] = x[b];
        x[b] = temp;
    }

    /** Return the index of the smallest String in x. */
    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]);
            // From the internet, if x[i] < x[smallestIndex], cmp will be -1.
            if (cmp < 0) {
                smallestIndex = i;
```

Video link

Reflections on the Development Process



And We're Done!

Often, development is an incremental process that involves lots of task switching and on the fly design modification.

Tests provide stability and scaffolding.

- Provide confidence in basic units and mitigate possibility of breaking them.
- Help you focus on one task at a time.

In larger projects, tests also allow you to safely **refactor**! Sometimes code gets ugly, necessitating redesign and rewrites (see project 2).

One remaining problem: Sure was annoying to have to constantly edit which tests were running. Let's take care of that.

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.

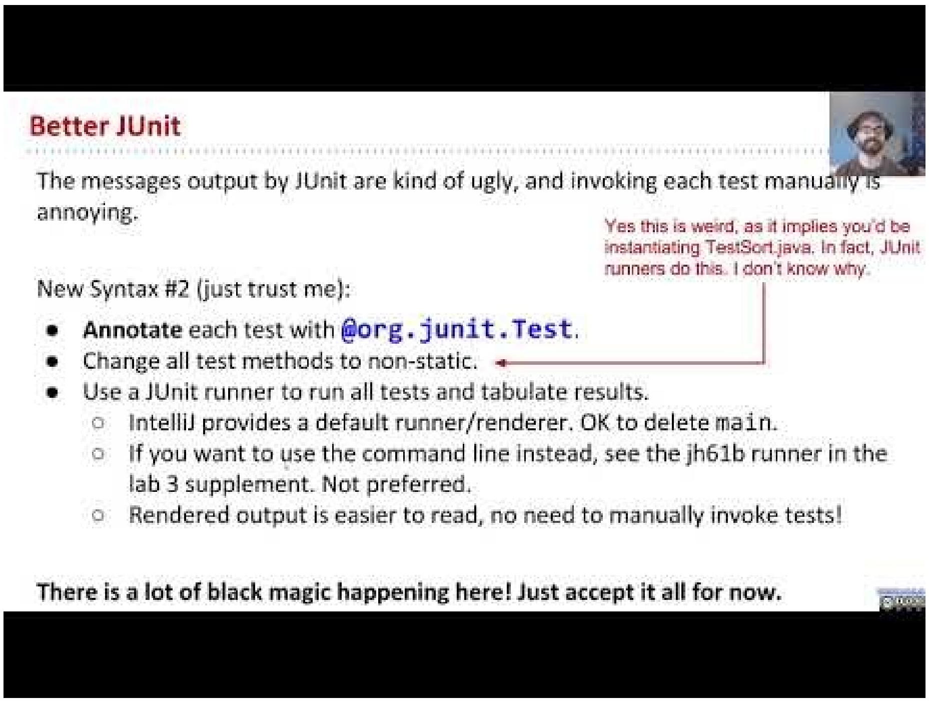
Having a set of automated tests helps reduce this cognitive load. For example, we were in the middle of writing `sort` when we realized there was a bug in `findSmallest`. We were able to switch contexts to consider `findSmallest` and establish that it was correct using our `testFindSmallest` method, 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



Better JUnit

The messages output by JUnit are kind of ugly, and invoking each test manually is annoying.

Yes this is weird, as it implies you'd be instantiating TestSort.java. In fact, JUnit runners do this. I don't know why.

New Syntax #2 (just trust me):

- Annotate each test with `@org.junit.Test`.
- Change all test methods to non-static.
- Use a JUnit runner to run all tests and tabulate results.
  - IntelliJ provides a default runner/renderer. OK to delete `main`.
  - If you want to use the command line instead, see the jh61b runner in the lab 3 supplement. Not preferred.
  - Rendered output is easier to read, no need to manually invoke tests!

There is a lot of black magic happening here! Just accept it all for now.

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 it is 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);`

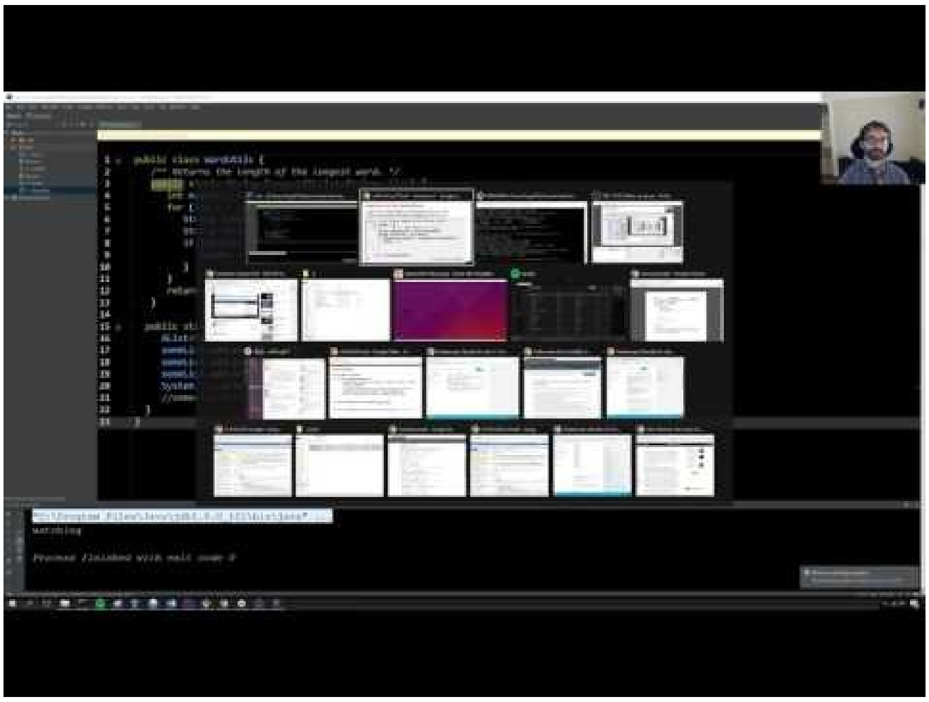
We will explain exactly why import statements are in a later lecture. For now, just use and enjoy.

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

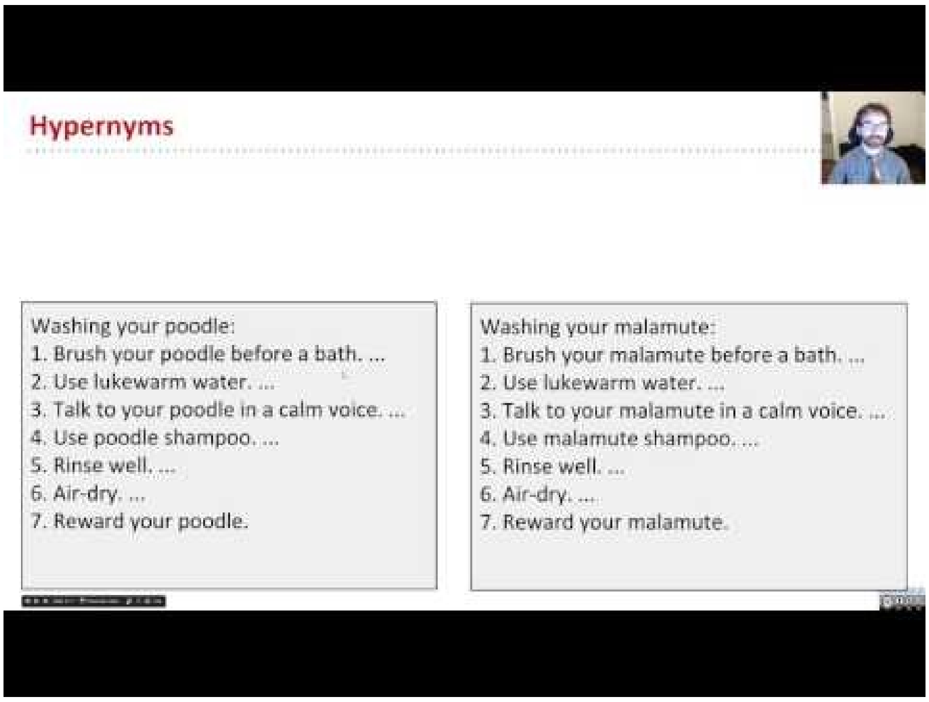
```
public static String longest(AList<String> list)
```

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 and Hyponyms


Hypernyms

Washing your poodle:
1. Brush your poodle before a bath. ...
2. Use lukewarm water. ...
3. Talk to your poodle in a calm voice. ...
4. Use poodle shampoo. ...
5. Rinse well. ...
6. Air-dry. ...
7. Reward your poodle.

Washing your malamute:
1. Brush your malamute before a bath. ...
2. Use lukewarm water. ...
3. Talk to your malamute in a calm voice. ...
4. Use malamute shampoo. ...
5. Rinse well. ...
6. Air-dry. ...
7. Reward your malamute.

Video link

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 selects the method to run based on the type of the argument you pass in. If you supply it with an AList, it will call the AList method. The same procedure would apply if we passed in an SLList.

It's nice that Java is smart enough to know how to deal with methods that share the same name, but have different parameter 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.

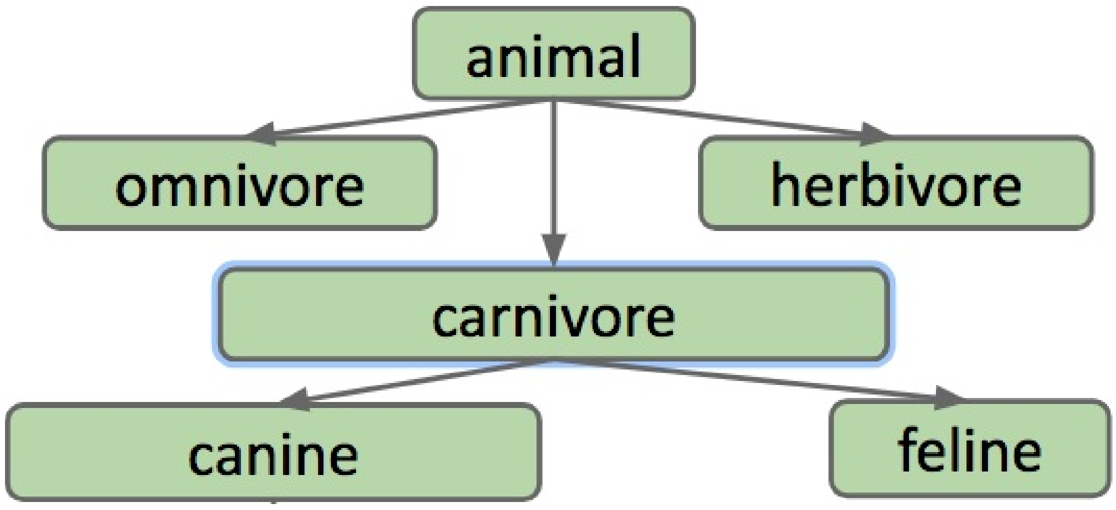
Hypernyms, Hyponyms, and Interface Inheritance

In the English language and life in general, there exists 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



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.

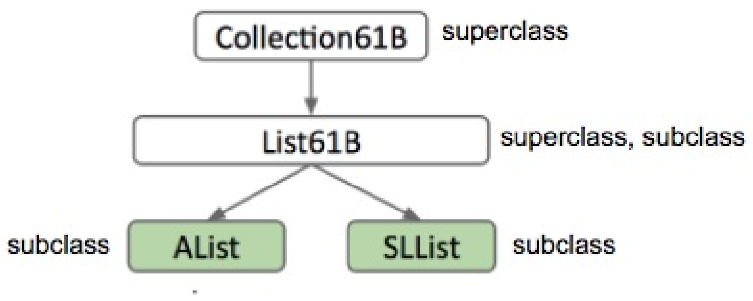


Figure 4.1.1

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> {...}`

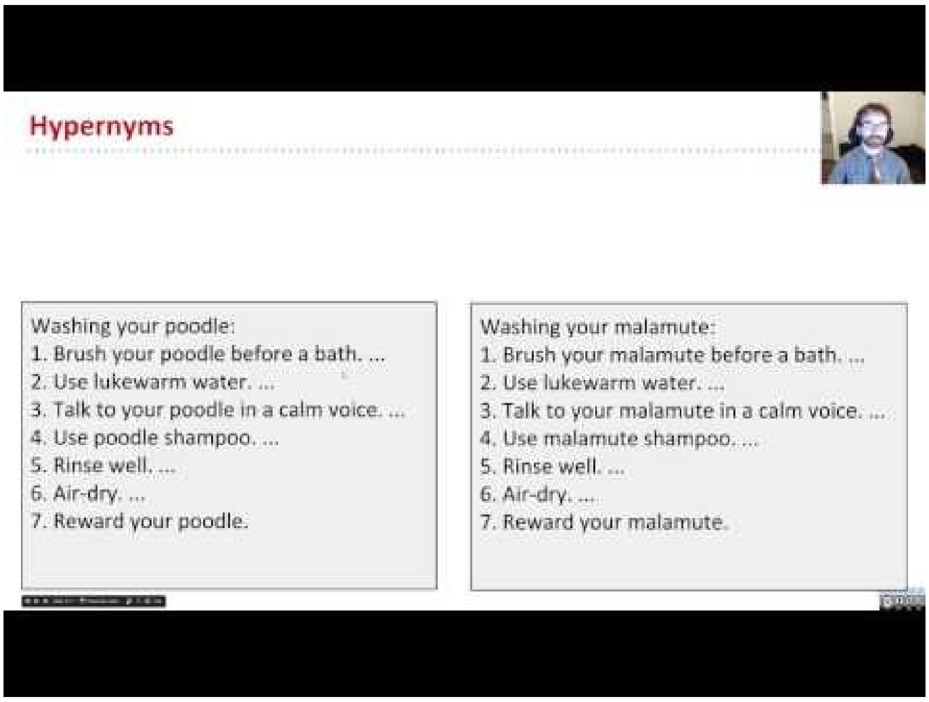
a relationship-defining word: implements.

`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



Hypernyms

Washing your poodle:
1. Brush your poodle before a bath. ...
2. Use lukewarm water. ...
3. Talk to your poodle in a calm voice. ...
4. Use poodle shampoo. ...
5. Rinse well. ...
6. Air-dry. ...
7. Reward your poodle.

Washing your malamute:
1. Brush your malamute before a bath. ...
2. Use lukewarm water. ...
3. Talk to your malamute in a calm voice. ...
4. Use malamute shampoo. ...
5. Rinse well. ...
6. Air-dry. ...
7. Reward your malamute.

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.

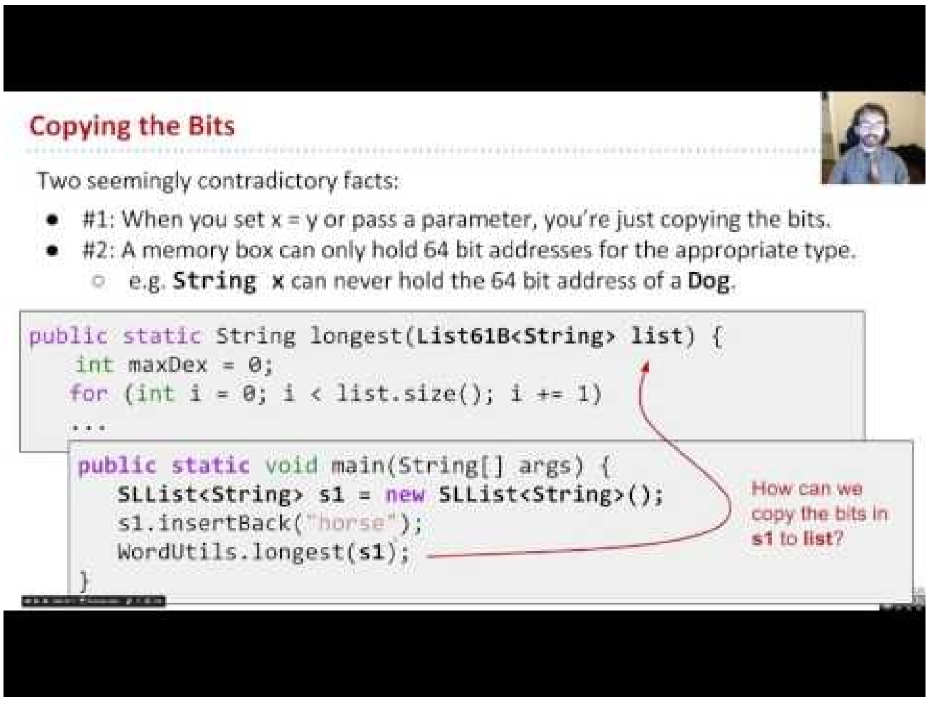
When implementing the required functions in the subclass, it's useful (and actually required in 61B) to include the `@Override` tag right on top of the method signature. Here, we have done that for just one method.

```
@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



Copying the Bits

Two seemingly contradictory facts:

- #1: When you set x = y or pass a parameter, you're just copying the bits.
- #2: A memory box can only hold 64 bit addresses for the appropriate type.
  - e.g. **String x** can never hold the 64 bit address of a **Dog**.

```
public static String longest(List61B<String> list) {
   int maxDex = 0;
   for (int i = 0; i < list.size(); i += 1)
   ...
```

```
public static void main(String[] args) {
   SLList<String> s1 = new SLList<String>();
   s1.insertBack("horse");
   WordUtils.longest(s1);
}
```

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!

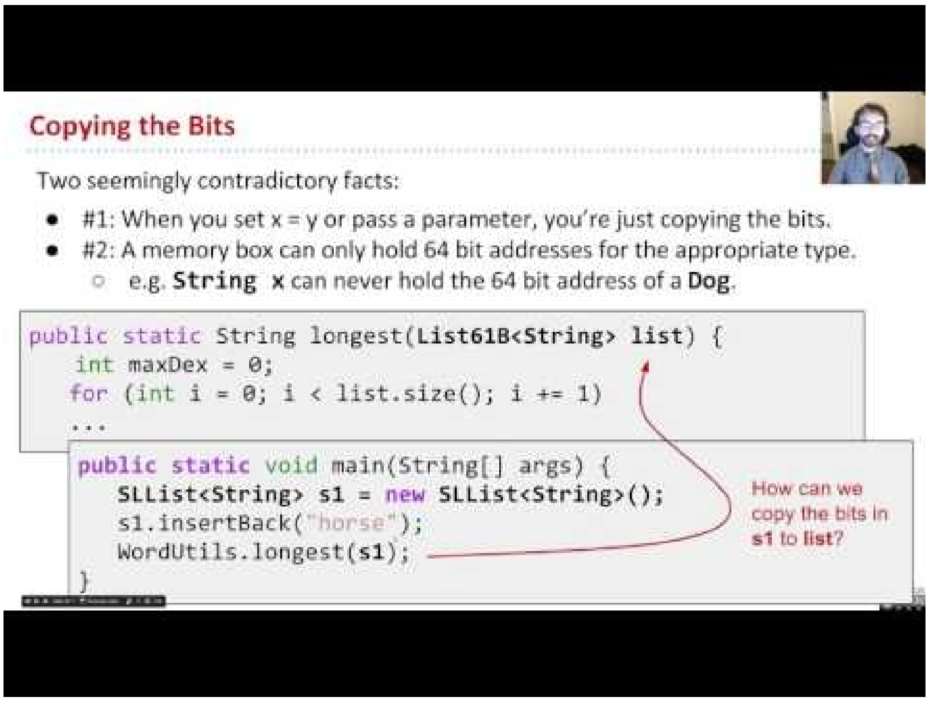
Exercise 4.1.2 Do you think the code below will compile? If so, what happens when it runs?

```
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, and 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



Copying the Bits

Two seemingly contradictory facts:

- #1: When you set x = y or pass a parameter, you're just copying the bits.
- #2: A memory box can only hold 64 bit addresses for the appropriate type.
  - e.g. String x can never hold the 64 bit address of a Dog.

```
public static String longest(List61B<String> list) {
   int maxDex = 0;
   for (int i = 0; i < list.size(); i += 1)
   ...
```

```
public static void main(String[] args) {
   SLList<String> s1 = new SLList<String>();
   s1.insertBack("horse");
   WordUtils.longest(s1);
}
```

How can we copy the bits in s1 to list?

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.

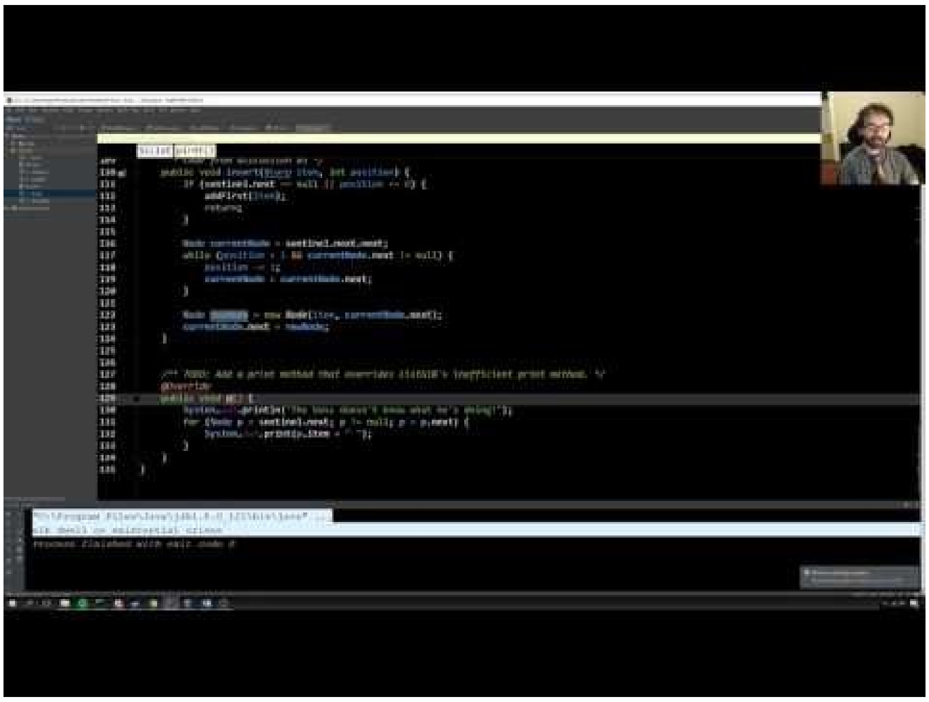
In order to do this, you must include the `default` keyword in the method signature.

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?



```
public void insert(Blorp item, int position) {
    if (sentinel.next == null || position == 0) {
        addFirst(item);
        return;
    }

    Node currentNode = sentinel.next.next;
    while (position > 1 && currentNode.next != null) {
        position -= 1;
        currentNode = currentNode.next;
    }

    Node newNode = new Node(item, currentNode.next);
    currentNode.next = newNode;
}


/** TODO: Add a print method that overrides List61B's inefficient print method. */
@Override
public void p() {
    System.out.println("The boss doesn't know what he's doing!");
    for (Node p = sentinel.next; p != null; p = p.next) {
        System.out.print(p.item + " ");
    }
}
}
```

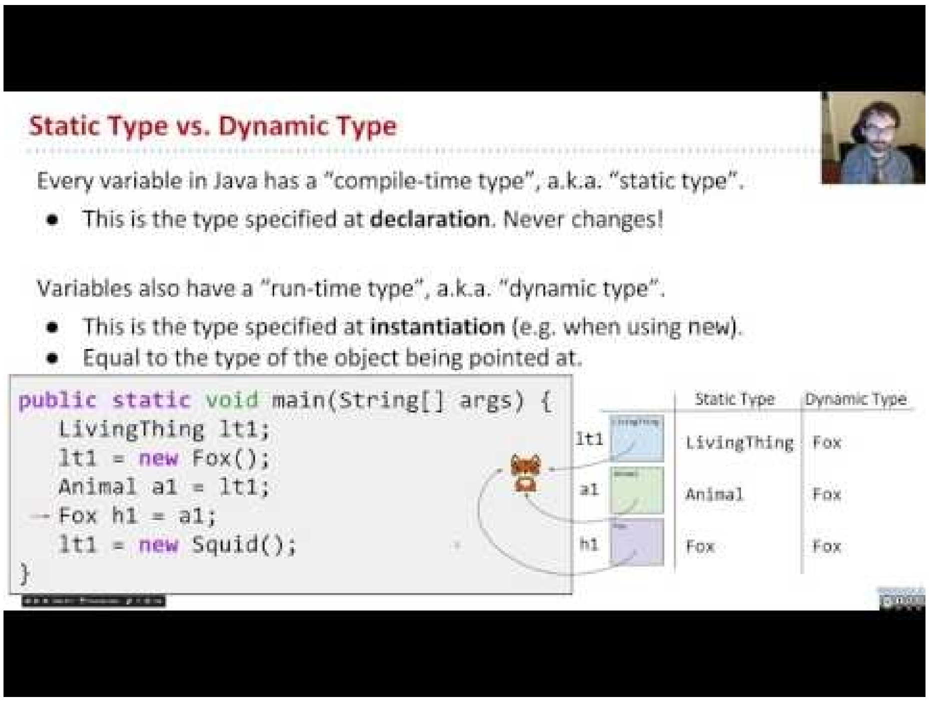
Video link

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.



Static Type vs. Dynamic Type

Every variable in Java has a "compile-time type", a.k.a. "static type".

- This is the type specified at **declaration**. Never changes!

Variables also have a "run-time type", a.k.a. "dynamic type".

- This is the type specified at **instantiation** (e.g. when using new).
- Equal to the type of the object being pointed at.

```
public static void main(String[] args) {
   LivingThing lt1;
   lt1 = new Fox();
   Animal a1 = lt1;
   Fox h1 = a1;
   lt1 = new Squid();
}
```

	Static Type	Dynamic Type
lt1	LivingThing	Fox
a1	Animal	Fox
h1	Fox	Fox

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>();
```

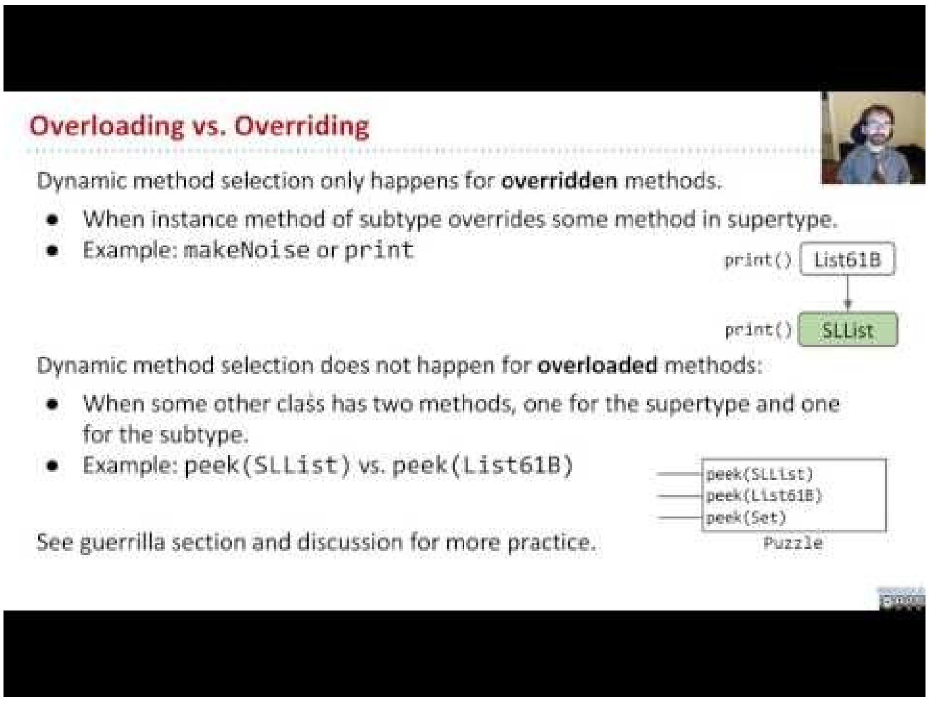
In the above declaration and instantiation, lst is of type "List61B". This is called the "static type"

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 l be reassigned to point to an object of another type, say a AList object, l’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!



Overloading vs. Overriding

Dynamic method selection only happens for **overridden** methods.

- When instance method of subtype overrides some method in supertype.
- Example: makeNoise or print

Dynamic method selection does not happen for **overloaded** methods:

- When some other class has two methods, one for the supertype and one for the subtype.
- Example: peek(SLList) vs. peek(List61B)

See guerrilla section and discussion for more practice.

Video link

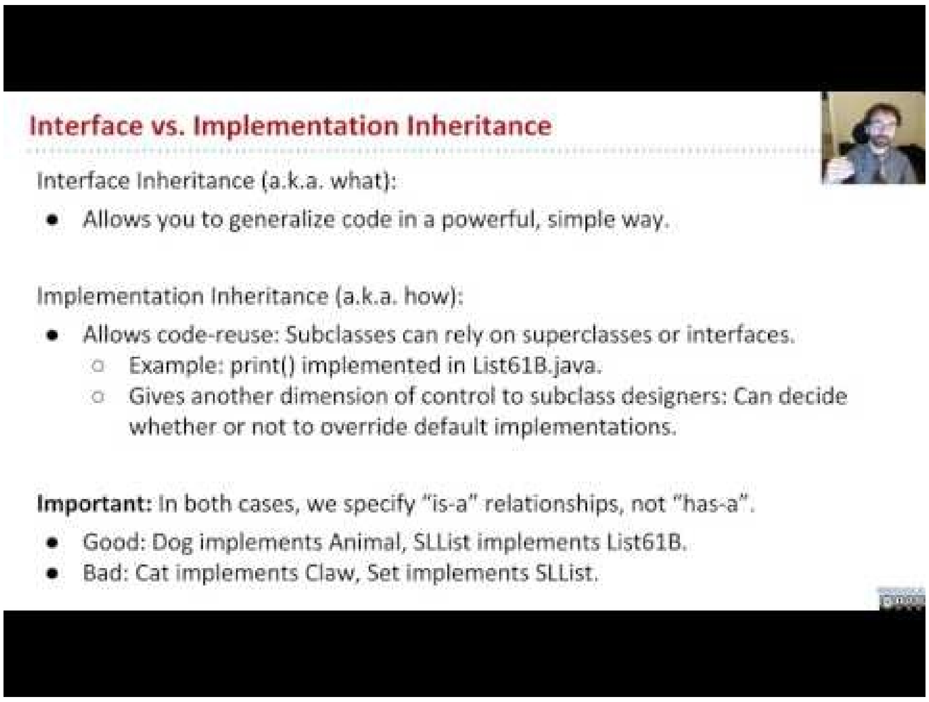
Say there are two methods in the same class

```
SLList<String> SP = new SLList<String>(List61B<String> list) {
    System.out.println(list.getLast());
}

SLList<String> SP = new SLList<String>(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.



Interface vs. Implementation Inheritance

Interface Inheritance (a.k.a. what):

- Allows you to generalize code in a powerful, simple way.

Implementation Inheritance (a.k.a. how):

- Allows code-reuse: Subclasses can rely on superclasses or interfaces.
  - Example: print() implemented in List61B.java.
  - Gives another dimension of control to subclass designers: Can decide whether or not to override default implementations.

Important: In both cases, we specify "is-a" relationships, not "has-a".

- Good: Dog implements Animal, SLList implements List61B.
- Bad: Cat implements Claw, Set implements SLList.

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