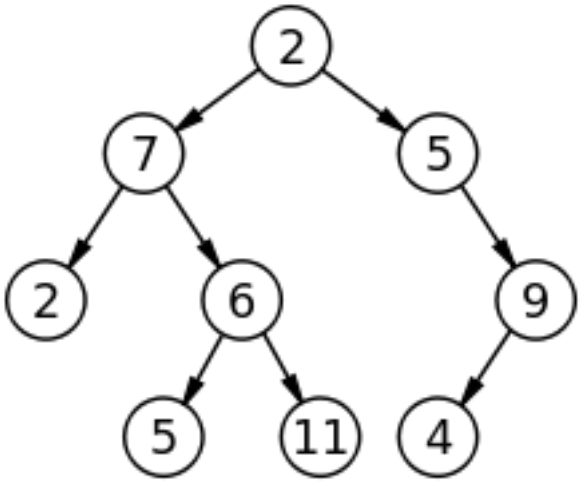
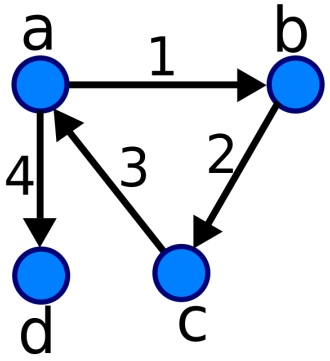
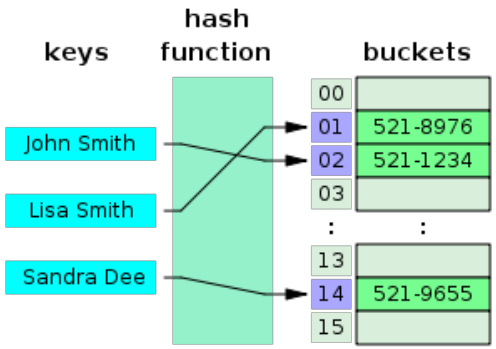
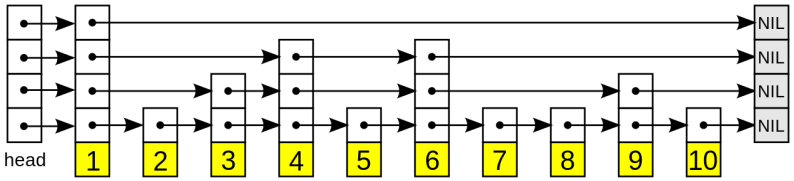
More reasons to understand linked lists!

If you understand how linked lists and arrays work, you can combine the ideas to create many other data structures!







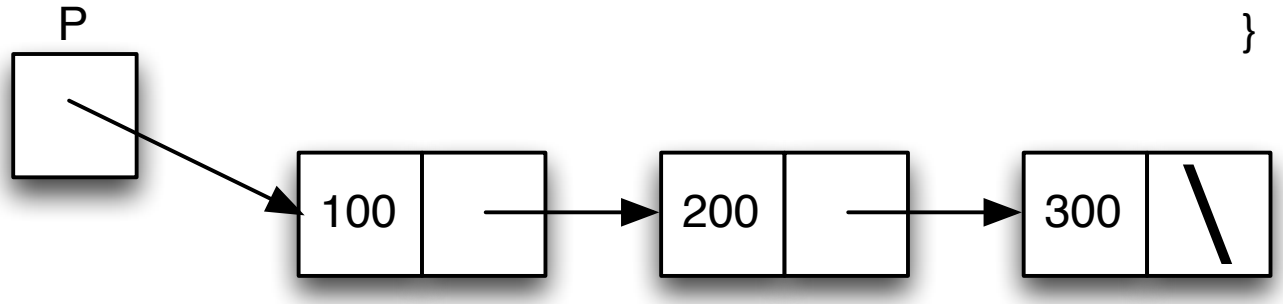


https://en.wikipedia.org/wiki/File:Binary_tree.svg

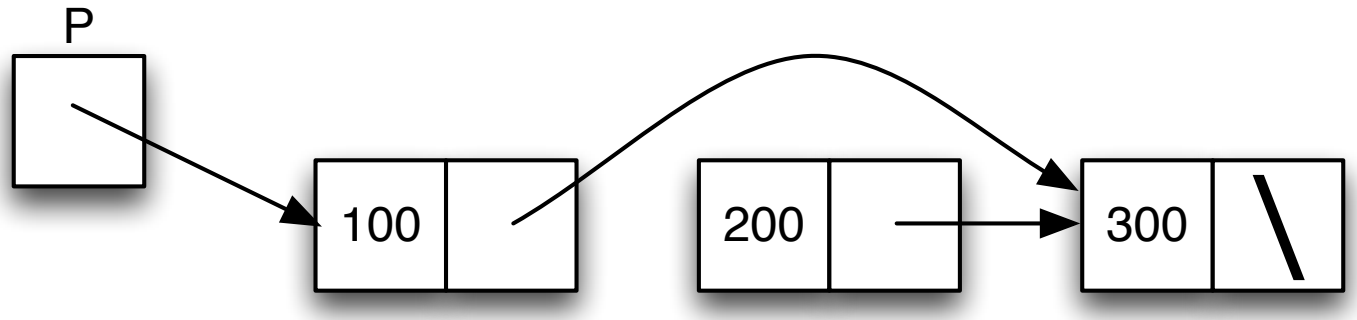
Quiz

```
class ListNode {
    int data;
    ListNode next;
}
```

BEFORE

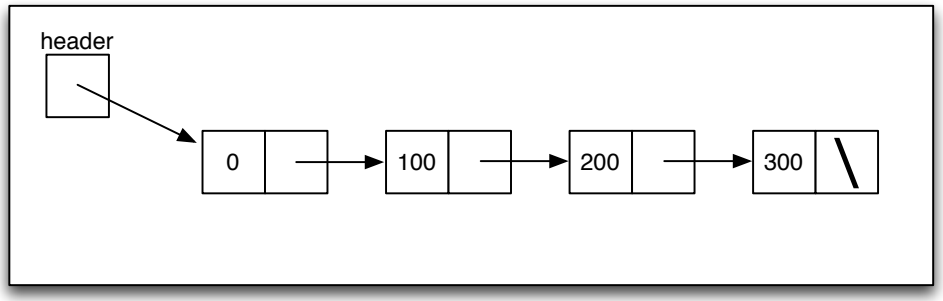


AFTER

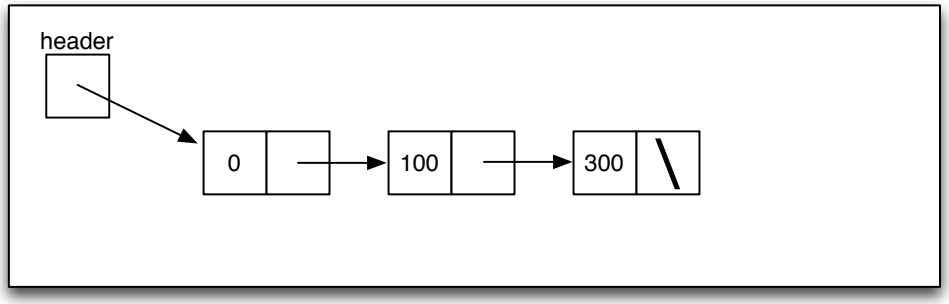


Write *one line* of Java code that turns BEFORE into AFTER.

```
public class SLinkedList {
  private ListNode header;


public SLinkedList () {
  // putting data=0 but doesn't
matter (we won't ever look)
  header = new ListNode(0);
}
```



remove(1)




```
public void remove(int index) {
  int curIndex = 0;
  ListNode current = header;
  while (_____A_____) {
    current = current.next;
    curIndex++;
  }
  _____B_____;
}
}
```

CS 2230
CS II: Data structures

Meeting 8: Testing

Brandon Myers

University of Iowa

Today's big ideas

- An **abstract data type (ADT)** is a model of data from the perspective of the user
- A Java **interface** defines only the signatures of methods. Interfaces are useful for defining ADTs
- **List** is an ADT that models a mutable list of elements; LinkedList and ArrayList are concrete implementations of List
- Interfaces can be used to write code once that works for objects of different classes (called ***reuse***)

new Abstract Data Type: List

Example behavior of a List

Method call	Return value	list after this call (abstract representation)
isEmpty()	true	[]
append(5)		[5]
append(7)		[5,7]
isEmpty()	false	[5,7]
get(0)	5	[5,7]
get(1)	7	[5,7]
removeFirst()	5	[7]
get(0)	7	[7]

List is an ADT, so we only know its methods and how they behave.

Trick question: How are the elements stored?
Answer: An ADT does not say anything about implementation!

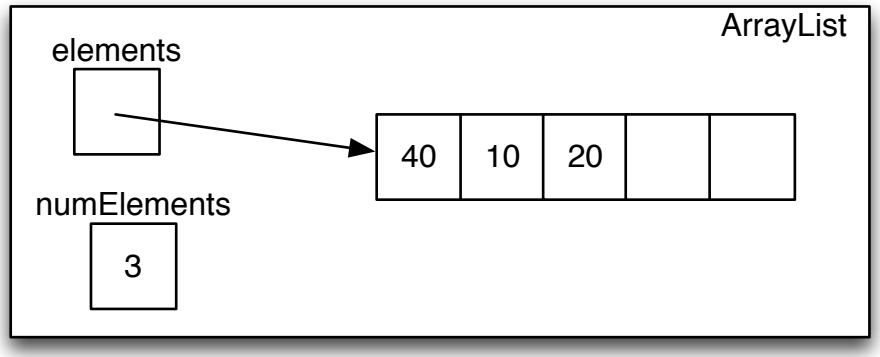
Defining List using a Java ***interface***

```
/*
Interface for a List ADT, an ordered collection of elements
*/
public interface List {
    // add the element to the end of the list
    public void append(Object element);


    // check if the element exists in the list
    public boolean contains(Object element);


    // remove first element from the list and return it
    public Object removeFirst();


    // return the element at index i
    public Object get(int i);


    // return true if the list is empty
    public boolean isEmpty();
}
```

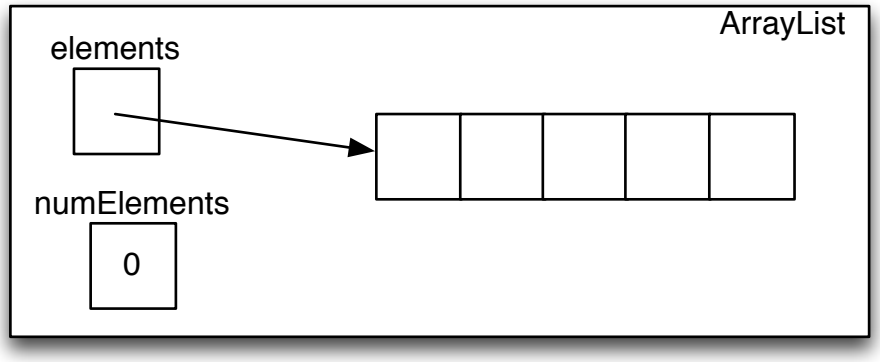
Implementation of List: ArrayList

```
/*
A List that is implemented using an array
*/
public class ArrayList implements List {
      private Object[] elements;
      private int numElements;

      public ArrayList() {
          elements = new Object[5];
          numElements = 0;
      }

      @Override
      public void append(Object ele) {
        put implementation here
      }

      @Override
      public boolean contains(Object element) {
       put implementation here
      }

      @Override
      public Object removeFirst() {
       put implementation here
      }

      @Override
      public Object get(int i) {
       put implementation here
      }

      @Override
      public boolean isEmpty() {
       put implementation here
      }
}
```

Example ArrayList



Initial state:



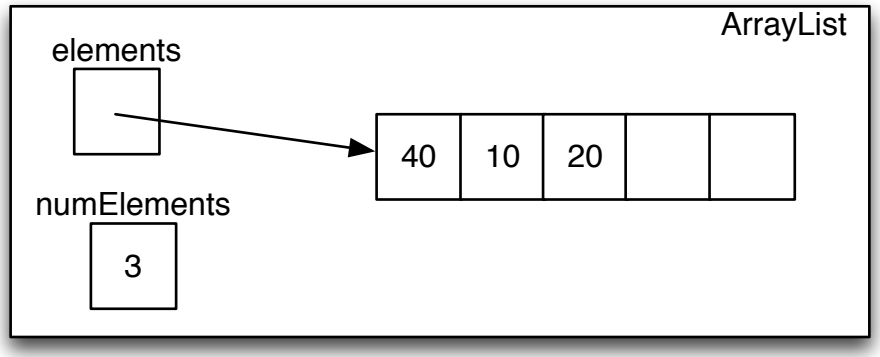
Method call	Return value	list after this call (abstract representation)
isEmpty()	true	[]
append(5)		[5]
append(7)		[5,7]
isEmpty()	false	[5,7]
get(0)	5	[5,7]
get(1)	7	[5,7]
removeFirst()	5	[7]
get(0)	7	[7]

a) Draw the ArrayList after this line

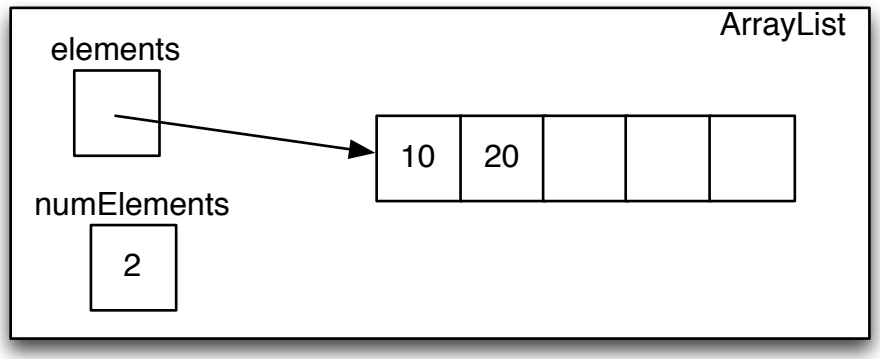
b) Draw the ArrayList after this line

```
public class ArrayList implements List {
     private Object[] elements;
     private int numElements;


     @Override
     public Object removeFirst() {









     }
}
```

Implement one of the methods in ArrayList



BEFORE

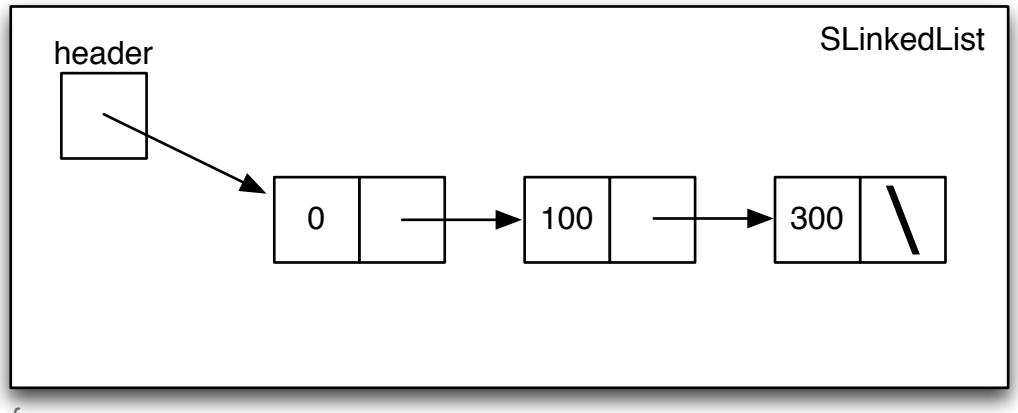


AFTER

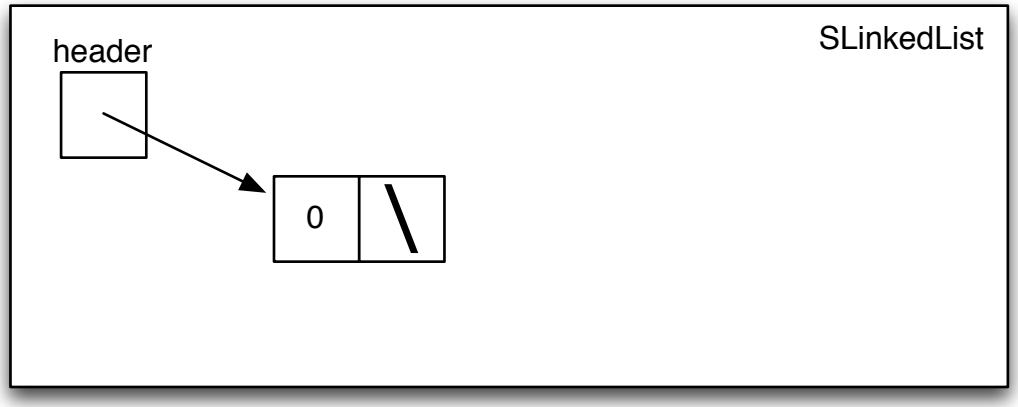
Implementation of List: SLinkedList

```
/*
A List that is implemented using a linked list
*/
public class SLinkedList implements List {
        private ListNode header;

        public SLinkedList() {
            header = new ListNode(0);
        }


    @Override
    public void append(Object element) {
     put implementation here
    }

    @Override
    public boolean contains(Object element) {
     put implementation here
    }

    @Override
    public Object removeFirst() {
     put implementation here
    }

    @Override
    public Object get(int i) {
     put implementation here
    }


    @Override
    public boolean isEmpty() {
     put implementation here
    }
}
```

Example SLinkedList



Initial state:



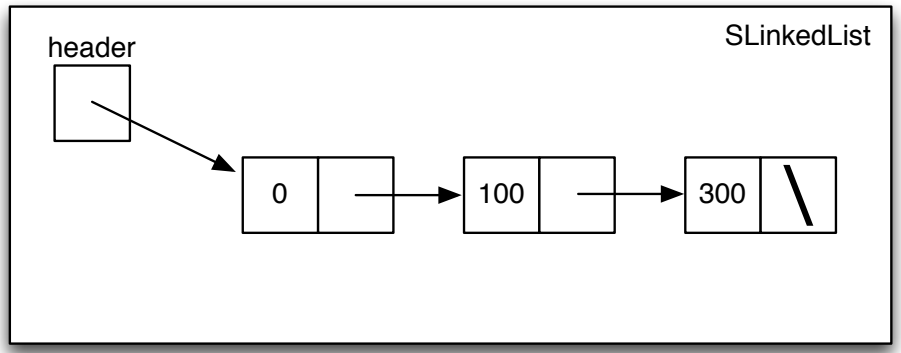
Method call	Return value	list after this call (abstract representation)
isEmpty()	true	[]
append(5)		[5]
append(7)		[5,7]
isEmpty()	false	[5,7]
get(0)	5	[5,7]
get(1)	7	[5,7]
removeFirst()	5	[7]
get(0)	7	[7]

a) Draw the SLinkedList after this line

b) Draw the SLinkedList after this line

```
public class SLinkedList implements List {
        private ListNode header;

        @Override
        public Object removeFirst() {









        }
}
```

Implement one of the methods in SLinkedList



https://b.socrative.com/login/student/
room CS2230X ids 1000-2999
room CS2230Y ids 3000+

```java
public interface I {
    int foo(int i, int j);
    int bar(String s);
}
```

```java
public class A implements I {
    @Override
    int foo(int x, int y) {
        return x;
    }
    @Override
    int bar(int s) {
        return s+1;
    }
}
```

```java
public class B implements I {
    @Override
    void foo(int i, int j) {
        int z = i + j;
    }
}
```

List all of the problems you can find in this program (has three files `I.java`, `A.java`, and `B.java`)

Another interface: `Comparable`

Comparing things is certainly useful, e.g. for sorting

“Show me cats ordered by cuteness"



“Show shapes ordered by number of sides"

An example interface: `Comparable`

- Since sorting things is so useful, we might want to write code that knows how to sort “anything”
  - (or, at least...anything *Comparable*)

```
public static void insertSorted(Comparable[] sorted, int
currentSize, Comparable newElement)
```

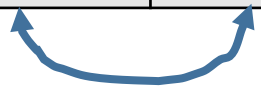
recall the algorithm for inserting into a sorted array:
insert new element at the the end, then swap until it is in the right place

Harry	Ron	Snape			

`registerNewPatient("Hermione")`

Harry	Ron	Snape	Hermione		

Harry	Ron	Hermione	Snape		



Harry	Hermione	Ron	Snape		

```
public static void insertSorted(Comparable[] sorted, int
currentSize, Comparable newElement) {

    sorted[currentSize] = newElement;
    int i = currentSize;
    while (i > 0 && sorted[i-1].compareTo(sorted[i]) > 0) {
    // swap
    ...
    }
}
```

initially `myarray`: [1, 4, 7]

What is the sequence of compareTo return values when we call `insertSorted(myarray, 3, 5)`

a) 1 1 -1

b) 1 0

c) 1 0 -1

d) -1 -1 1

e) 1 -1

https://b.socrative.com/login/student/
room CS2230X ids 1000-2999
room CS2230Y ids 3000+

Today's big ideas

- An **abstract data type (ADT)** is a model of data from the perspective of the user
- A Java **interface** defines only the signatures of methods. Interfaces are useful for defining ADTs
- **List** is an ADT that models a mutable list of elements; LinkedList and ArrayList are concrete implementations of List
- Interfaces can be used to write code once that works for objects of different classes (called ***reuse***)

Today's big ideas 2

- Testing
  - you should write tests
  - JUnit helps you organize and run your tests
  - a look at test driven development

- Different kinds of equality in Java

How do you know your program works?

What evidence did you have that your programs worked in HW1 and HW2?

https://b.socrative.com/login/student/
room CS2230X ids 1000-2999
room CS2230Y ids 3000+

What do we mean by “your program works” ?

Usually, we mean that the *implementation* obeys the *specification*

```
// add the integer d to the end of the list
// (also...please don't crash!)
public void append(int d) { }
```

the specification for LinkedList.append()

your implementation of LinkedList.append() is correct if it obeys the above specification

Many ways that people gather evidence that their program works

- **testing:** run it, automatically check the output
- **static analysis:** automatically check properties without actually running your program
- **handwritten proof:** write a mathematical proof that the program works
- **machine-checked proof:** automatically verify your proof that the program works is correct
- **model checking**: with help, the computer comes up with a proof for you, but only for “finite” number of cases

Evidence in CS2230

you will mostly rely on tests that you write

- **testing:** run it, automatically check the output

compiler will check for simple errors for you (e.g., types and syntax)

- **static analysis:** automatically check properties without actually running your program

before you write code for an algorithm you want at least an informal proof that it is correct

- **handwritten proof:** write a mathematical proof that the program works

Approach in CS 2230

- you will learn to **write and use tests**
- you will learn to **debug** programs **systematically**
- you will learn to write “assertions” that **check the invariants** of your data structures

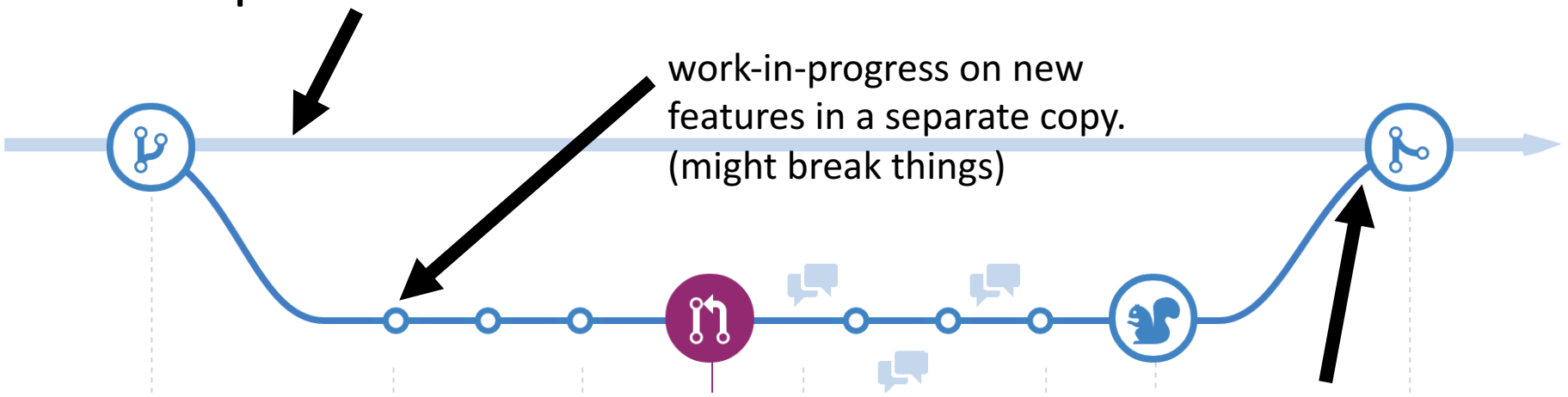
What should you test?

(be brief but specific)

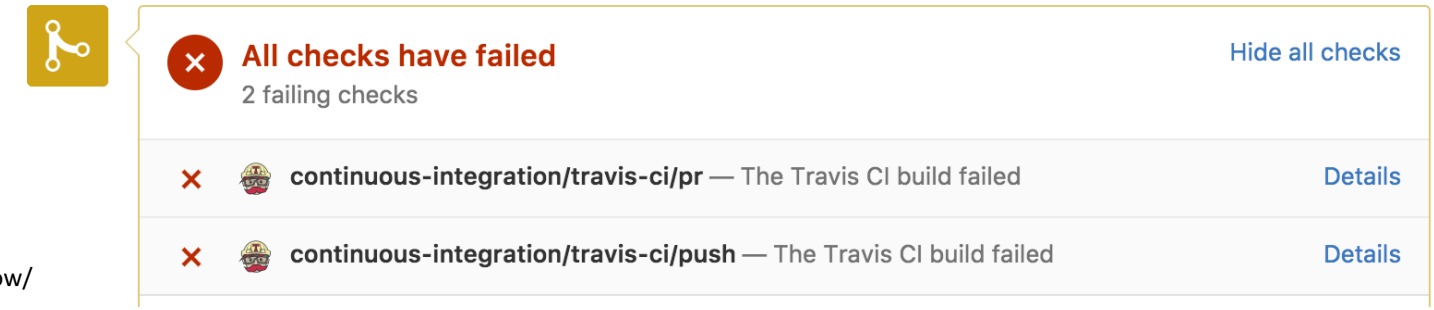
https://b.socrative.com/login/student/
room CS2230X ids 1000-2999
room CS2230Y ids 3000+

In many software projects, the tests get the final say

There is a “golden copy” of the code that passes all the tests



not allowed to “merge” the new feature into the golden copy unless all the tests pass



https://guides.github.com/introduction/flow/

Goal for today

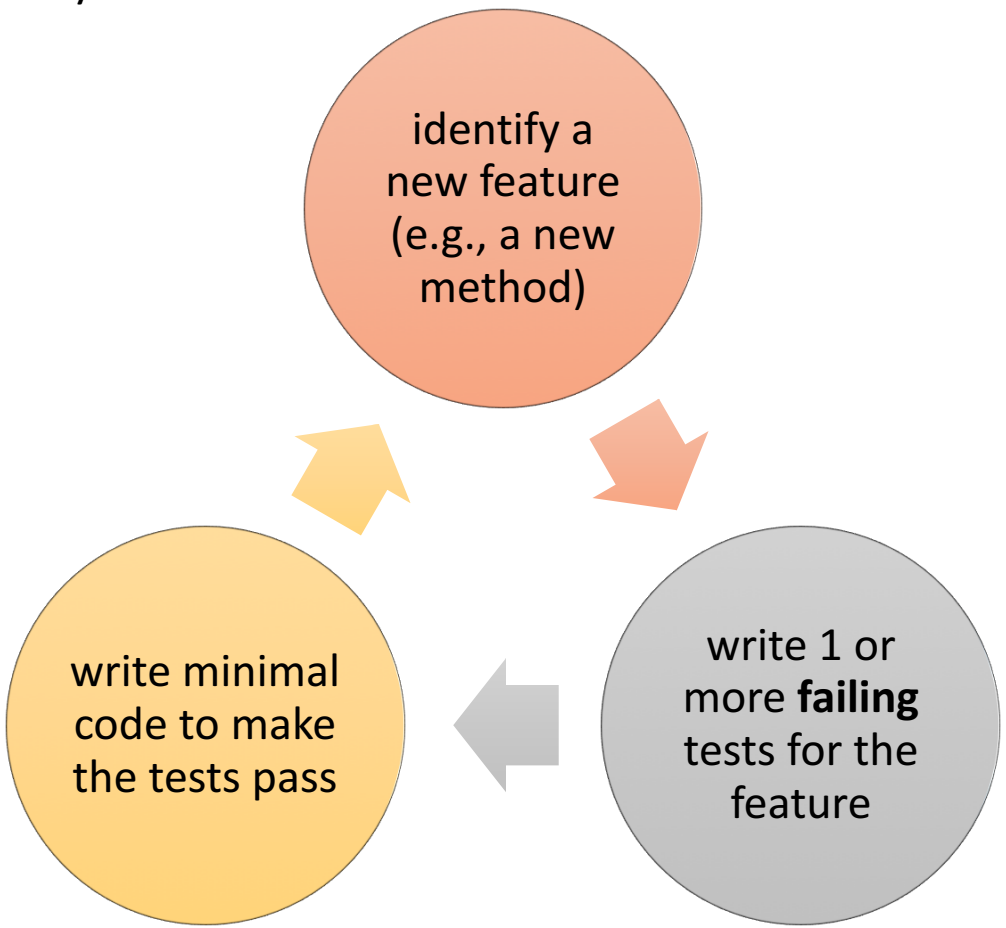
create a class with this specification and test it

```
// insert the String s at the front of the list
void insertFirst(String s) { }


// Remove the String at the front of the list
// and return it. Returns null if the list is empty.
String removeFirst() { }


// Print out the elements of the list in order
void printList() { }
```

Approach for today

We'll use a methodology called test-driven development

- not the only choice
- but it can be very useful



Write JUnit tests for FLinkedList.removeFirst()

Implement FLinkedList.removeFirst() and fix it until all of our existing tests pass

What does a partially red bar in JUnit's output mean?

A. all the tests failed
B. a test has a bug in it
C. the code that is being tested has a bug in it
D. all the tests passed
E. none of the above

https://b.socrative.com/login/student/
room CS2230X ids 1000-2999
room CS2230Y ids 3000+

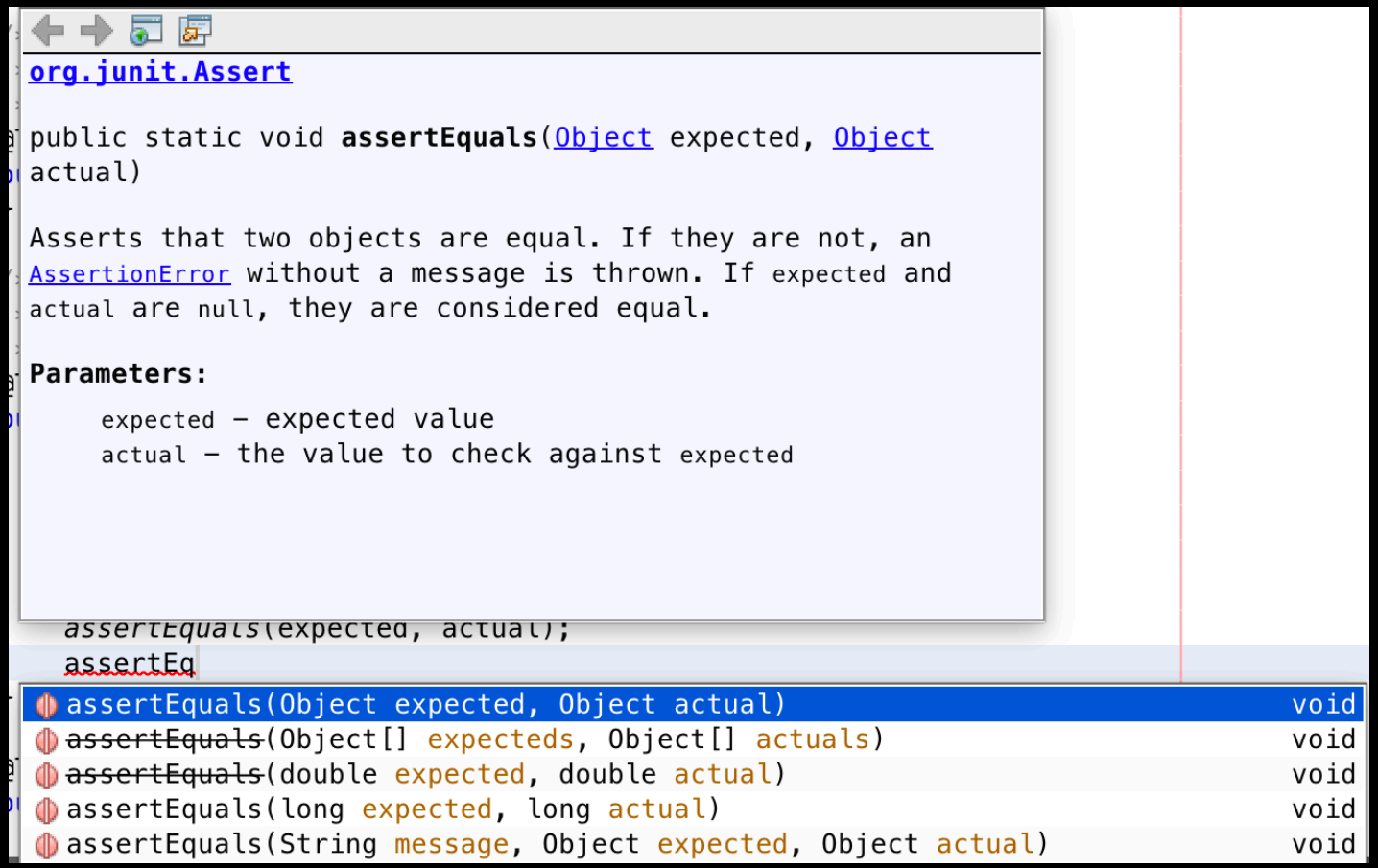
Write JUnit tests for FLinkedList.printList()?

writing tests for methods that print to the console is possible but a bit more complicated

Instead, let's write a test for a helper method that would be useful to printList()!

specifically, toString()

You are writing JUnit tests now?



it's time to talk about Java equality...

== isn't always equal?

In Java, == does the expected for primitives.

```
int a = 26;
int b = 26;
// a == b is true
```

```
int a = 13;
int b = 26;
// a == b is false
```

Comparing two references checks if they are pointing to the same object

```
Patient p1 = new Patient("Marion", 100);
Patient p2 = new Patient("Marion", 100);
Patient p3 = p1;
// p1 == p2  is false
// p1 == p3  is true
```

Not pointing to the same object? not ==

The equals() method

```
class Patient {
    String name;
    int height;
}
```

We decide that two Patients are equal() when they have the same name and height

the code that does that...

```
public boolean equals(Object o) {
        if (o instanceof Patient){
                Patient op = (Patient) o;
                return this.height==op.height &&
                        this.name.equals(op.name);
        } else {
                return false;
        }
}
```

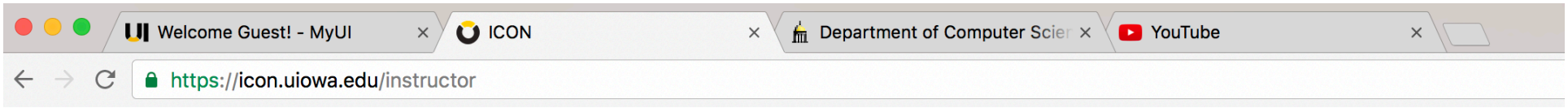
String.equals compares the content of the String instead of references

Every Java class already has an invisible `equals` method defined. But you have to *override* it with your own if you want to do something smarter like compare the fields.

Secondary new things in this snippet of code

- instanceof to check if o is a Patient
- casting o from Object to Patient

HW3: linked lists applied to browser tabs



You will write methods for a LinkedList class. Each method will provide new functionality for tabs (reorder, open, close, display, ...).

Today's big ideas 2

- Testing
  - you should write tests
  - JUnit helps you organize and run your tests
  - a look at test driven development

- Different kinds of equality in Java