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 end, then swap until it is in the right place

registerNewPatient("Hermione")

Harry  Ron  Snape  Hermione  Snape  Hermione  Snape
Harry  Hermione  Ron  Snape  Hermione  Snape  Hermione
Harry  Hermione  Ron  Snape  Hermione  Snape  Hermione
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) 11-1
b) 10
c) 10-1
d) -1-11
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**)

Administrivia

• HW 3 / Quiz 3 available by tomorrow
  • Application: browser tabs stored in a linked list
  • practice using a testing tool (JUnit)

• Section today
  • fixing bugs in the SLinkedList from pre-lab
  • learn to use JUnit to debug a program
  • remember: prelab question #1 must be turned in on paper beginning of lab
HW3: linked lists applied to browser tabs

You will write methods for a LinkedList class and debug with test cases. 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
How do you know your program works?

What evidence do 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

```java
public void append(int d) {
}
```

the specification for 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
you will mostly rely on tests that you write

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

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

• **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

Evidence in CS2230
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

work-in-progress on new features in a separate copy. (might break things)

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

```java
// 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

identify a new feature (e.g., a new method)
write minimal code to make the tests pass
write 1 or more failing tests for the feature
1. Write JUnit tests for FLinkedList methods
2. Implement the FLinkedList methods, fixing the tests one-by-one
What does this JUnit message tell us?

A. FLinkedList passed the test case called testInsertFirst
B. all assert statements in testInsertFirst passed
C. we have evidence supporting that FLinkedList’s insertFirst method is correct
D. we have proven FLinkedList’s insertFirst method is correct
E. the test case called testInsertFirst took 0.002 seconds to run
What does this JUnit message tell us?

A. FLinkedLIst failed the test called testRemoveFromListSize1.
B. 50% of our code is correct
C. There is a failed assertion at line 37 of FLinkedListTest.java
D. There are 37 failed assertions in FLinkedListTest.java
E. There is a bug in junit.framework
F. An assertion failed because our code returned null, but it was supposed to return “a”
G. An assertion failed because our code returned “a”, but it was supposed to return null
FLinkedListTest.java

```java
@Test
public void testRemoveFromListSize1() {
   // check remove from small list
   FLinkedList cut = new FLinkedList();

   cut.insertFirst("a");
   String expected = "a";
   String actual = cut.removeFirst();
   assertEquals(expected, actual);
}
```
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?

```java
import org.junit.Assert;

public static void assertEquals(Object expected, Object actual)
{
    Assert.assertEquals(expected, actual);
}
```

assertEquals(expected, actual);

assertEq

assertEquals(Object expected, Object actual)
assertEquals(Object[] expecteds, Object[] actuals)
assertEquals(double expected, double actual)
assertEquals(long expected, long actual)
assertEquals(String message, Object expected, Object actual)

it’s time to talk about Java equality...
== isn’t always equal?

In Java, == does the expected for primitives.

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

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

```java
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

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

the code that does that...

```java
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;
    }
}
```

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`
Peer instruction

```java
boolean equals(Object o) {
    if (o instanceof Cat) {
        Cat c = (Cat) o;
        return this.breed.equals(o.breed);
    }
    return false;
}
```

Object o1 = new Object();
Object o2 = new Cat("Siamese");
Cat o3 = new Cat("Tabby");
Cat o4 = new Cat("Siamese");
Cat o5 = o2;

Which are true statements?
A. o1 == o2
B. o2 == o4
C. o2 == o5
D. o4 == o5
E. o2.equals(o4)
F. o2.equals(o5)
G. o4.equals(o5)
H. o2.equals(o3)

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