Practice midterm posted later today
  • Much longer than the actual midterm will be!
  • Some sample solutions posted as well.

Midterm review next Wednesday (3/16)
  • Come with questions!

No class next Friday (3/18)
Sorting Objects
Sorting with `compareTo`

- Let’s add a `compareTo()` method to `Student`
- This method compares the name of this student
- How does this choice affect what a sorted vector looks like?
- Let’s try sorting `Student` with a `compareTo` method
Making InsertionSort generic

- We never used the fact that this is a vector of students (other than the `compareTo()` method)

- What kind of types can we sort?

- We want this class to have a `compareTo()` method. How can we require this?

- With an interface!
Comparable\(<T>\) Interface

- This is a Java interface, *not structure5*. (Built-in; don’t need to import anything.)

- Comparable\(<T>\) has only one method: public int compareTo(T other)

- Let’s tell Java that our Student class implements this interface
Generic Upper Bounds

- Way to tell Java that a generic type needs to meet certain requirements
- That way, at compile time, Java can make sure our types match up
- These are called **upper bounds**
- Let’s say we only want to accept objects that meet the requirements of the List interface. Rather than `<E>`, we write something like `<E extends List>`
  - (Yes, it’s extends and not implements. There are some good back-end reasons for this.)
- What do we want for our `insertionSort` method?
  - Want `<E extends Comparable<E>>`
  - That is to say: we want a type `E` that implements `Comparable<E>`. That is to say: need that objects of type `E` have a `compareTo` method that takes objects of type `E` as argument
Where we are

- Can sort any object so long as it implements `Comparable<E>`

- What are the downsides of this?

  - What if we want to sort objects that aren’t already comparable and we don’t want to modify the class?

  - Can only sort objects one way. (What if we want to sort `Students` by grade? Would need to rewrite the `Student` class!)

- There are upsides as well; we’ll come back to this after we talk about Comparators
• Way to sort objects without changing the class.

• Let’s try to sort students by age, without rewriting Student. How can we do that?

• Idea: use an object whose job it is to compare students
  • This object will not store any data
  • Will just have a int compare(Student, Student) method to compare two students

• In general: Comparator<T> has a int compare(T, T) method
  • compare returns < 0 if first is smaller; 0 if equal; > 0 if second is smaller
  • Only job: compare objects of type T
  • Need to import java.util.Comparator
Writing a `Comparator<Student>`

- This is a class that implements `Comparator<Student>`

- Goal: sort students by `age`
Using a Comparator

- Let’s say we want a sort method that uses a comparator, rather than a comparable object

- (Our sort should work for *any* comparator)

- Idea: take a comparator object as an argument. Then we can use its compare method to compare the objects!
Abstract Classes
Object Oriented Programming

- One advantage: Helps break down data and code into self-contained chunks
- Also: can use objects as building blocks to create other objects!
  - Improved portability, extensibility
  - *Avoid repetition!*
- Today: abstract classes
- Wraps up Linked Lists
Interfaces

• “Recipe” for the methods that must be available in any class implementing the interface

• Allows us to use multiple objects of different class types, through a united interface

• Limitation: can’t write any code in an interface.
  • (For now.)

• When is that a problem?

• What if several different classes implement the exact same method?
Example: Lists

- Vector and SinglyLinkedList both implement the List interface
- That means they both have a method `addFirst(E)`
- In fact, both have the same method!

```java
public void addFirst(E value) {
    add(0, value);
}
```
Abstract Classes

- If many classes have identical methods, want to only write that method once

- Idea: use an *abstract class* to store these methods
Abstract Class: Definition and Notation

• An abstract class is a partial implementation of the class; uses the abstract keyword

• Have some methods written out
  • Can also have instance variables

• Don’t need to write all methods, even if implementing an interface

• Like an interface: cannot instantiate an object of an abstract class type

• Idea: this is just a part of a class! Need to fill in the details with a non-abstract class
Abstract Class Usage

- We write some methods in an abstract class

- When writing a new class, can use the `extends` keyword to use the methods in that abstract class

- If we extend an abstract class, we can use any of its methods! Plus any additional ones we implement.
Abstract Classes with Lists

• In structure5, have an AbstractList class that implements methods that would be identical in all Lists
  • addFirst, addLast, contains, etc.

• Our lists then extend AbstractList to allow us to use these methods

• Let’s look at the code
The structure universe (almost)
The structure5 universe (so far)
The structure5 universe (after break)
Summary

• **abstract** keyword declares a class as abstract

• **extends** means that we are adding more methods on to an existing (abstract in this case) class

• We can replace abstract class methods with our own if we want, or use them as-is

• Cannot instantiate objects of abstract class type!