CS151 Intro to Data Structures

Interfaces

Algorithm Analysis

Warmup

Assume correct `insert` and `expand` methods exist. Write the `moveToFront` method.

It should move the specified element to the front of the expandable array. It should search for the first occurrence of the given element in the array. Upon finding it, the method shifts all elements between the start of the array and the found element one position to the right. The order of the remaining elements should be preserved.

Draw 3 "test" arrays and execute your method on them. Again, make sure to consider edge cases.

```
public class ExpandableArray<E> {
    private E[] data;
    private int size;


    public void moveToFront(E data) {

    }

}
```

Algorithmic Analysis

What is the big-o notation of the following operations?

1. `moveToFront` - DoublyLinkedList
2. `moveToFront` - ExpandableArray

Compare the two operations. Is one cheaper? Why or why not?

Announcements

- HW02 and Lab3 due Monday
- Lab4 today
  - due 2/18 (next Wednesday)
  - manual checkoff

Java Memory Diagrams

Given a segment of code:

- Trace the program by going through line by line
  - Draw a memory diagram
  - As program variables are updated, update their values in the diagram

Example 2

Given a segment of code:

- Trace the program by going through line by line
  - Draw a memory diagram
  - As program variables are updated, update their values in the diagram

Interfaces

- An interface is a contract - A set of shared methods that users **must** implement
- create a program to calculate the area of different shapes, such as circles, rectangles, triangles etc.
- For each shape, you should be able to print the shape name and area
- Every time someone adds a new shape, they **must** include the methods for getName() and getArea()

Interfaces

- For any new shape that is created, we want to **enforce** that these methods are also implemented.

```
interface Shape {
    public double getArea();
    public String getName();
}
```

```
class Circle implements Shape {
```

Interfaces

A contract - A set of shared methods that users **must** implement

A collection of method signatures with no bodies

A class can implement more than one interface

Interfaces

An interface is not a class!

A class is what an object **is**

An interface is what an object **does**

- can not be instantiated
- no constructors
- incomplete methods

Interface

No modifier - implicitly `public`

No instance variables except for constants (`static final`)

Object Comparison

Object Equality

A custom class must define (override) its own `equals`

Object Comparison

- What if we wanted to compare two students by GPA?

```
int compareTo(T o)
```

Parameters:

`o - the object to be compared.`

Returns:

`a negative integer, zero, or a positive integer as this object is less than, equal to, or greater than the specified object.`

`compareTo`

`compareTo` returns an `int`, not a `Boolean`

Why?

because it needs to convey three outcomes:

- `negative` if smaller compared to the parameter
- `0` if equal
- `positive` if larger compared to the parameter

`Comparable` interface

The `Comparable` interface is designed for objects that have an ordering

```
public interface Comparable<T> {
  int compareTo(T o);
}
```

`Comparable` interface

When would we want to use this? **Let's see in code :)**

Now, what if we wanted to sort from highest to lowest GPA

Custom Exceptions

Making Custom Exceptions

Often times we need to raise a custom exception

Extend `Exception` or `RuntimeException`

Custom Exceptions

What is the difference between extending from `Exception` rather than `RuntimeException`?

Subclass of `Exception` are checked exceptions – must be treated/caught

Subclass of `RuntimeException` are not checkable during compile time

Summary

- Every non-primitive is an OBJECT in Java
- Every Object is a REFERENCE

- Interfaces are a *contract* of which methods you will implement

- You can compare objects with compareTo