CS102

Introduction to data structures, algorithms, and object-oriented programming

March 28, 2016
Interfaces allow for polymorphism

Using structural recursion, you don't need to check for the end of the list. Why?

Because, even though an object is declared as the interface type, when a method call on an object of a subtype occurs, the method is called on the *instantiated* type, not the interface (the *declared* or *supertype*).
/**
 * IList.java
 */

public interface IList {

    // stub for method to double all values in this list
    public IList doubleList();

    // stub for method to return length of this list
    public int length();

    // stub for method to sum all integers in this list
    public int sum();

    // stub for method to add an integer to front of list
    public IList cons(int i);
}

/**
 * EmptyList.java
 * Represents an empty list
 */

public class EmptyList implements IList {

    // Inserts its argument onto the front (left) of this list
    public IList cons(int x) {
        return new ConsList(x, this);
    }

    // Doubles each number in this IList, producing an IList
    public IList doubleList() {
        return this;
    }

    // Returns length of this list
    public int length() {
        return 0;
    }

    // Returns 0: no ints in this IList
    public int sum() {
        return 0;
    }
}
// Returns a String representation of this EmptyList
global public String toString()
{ 
    return "()";
}

/ * ConsList.java
 */

public class ConsList implements IList {

  private int first;
  private IList rest;

  // Creates a new instance of ConsList
  public ConsList(int first, IList rest) {
    this.first = first;
    this.rest = rest;
  }

  // Inserts its argument onto the front (left) of this list
  public IList cons(int i) {
    return new ConsList(i, this);
  }

  // Initialize variables in constructor
  // Two instance variables
  // No recursive call...no need to traverse
  // the list, just put i at the front.
public IList doubleList() {
    return new ConsList(this.first * 2, rest.doubleList());
}

public int sum() {
    return this.first + this.rest.sum();
}

public int length() {
    return 1 + this.rest.length();
}

public String toString() {
    return "("+this.first+" "+ this.rest.toString()+" ) ");
}
/*
 * TestIList.java
 *
 * Program with constructor that makes a random list of
 * numbers and then calls all the methods in the IList
 * hierarchy on that list.
 */
import java.util.*;

public class TestIList {

    // declare size of list as a constant
    private static final int MAX = 5;

    // declare maximum range of numbers
    private static final int RANGE = 100

    // class to generate random numbers
    private Random generator;

    //List to contain random numbers
    private IList numbers;
public TestIList() {
    this.generator = new Random();
    this.numbers = new EmptyList();
    // Create IList of MAX random numbers
    for (int i = 0; i < MAX; i++) {
        this.numbers = new ConsList(genNumber(), this.numbers);
    }
    // Print out the original IList
    System.out.println("Original list: ");
    System.out.println(this.numbers);

    // Print out the list in which each number is doubled
    System.out.println("\nDoubled numbers: ");
    IList newList = this.numbers.doubleList();
    System.out.println(newList);

    // Print out the sum of integers in the given IList
    System.out.println("\nSum of list: ");
    System.out.println(this.numbers.sum());
}
System.out.println("\nLength of list:");
System.out.println(this.numbers.length());

} // end of constructor

/**
 * Returns a random number between 0 and RANGE-1
 */
private int genNumber() {
    return Math.abs(this.generator.nextInt() % RANGE);
}

/**
 * main method of program
 */
public static void main(String[] args) {
    new TestIList();
}
Additional methods for lists (lab7)

• Write a predicate method isXIn(x) that returns true if a given int x is in the list and false otherwise.

• Write a method getElementAt(i) that returns the key stored in position i of the list (or -1 if position i doesn't exist).

• Write a method append(x) that returns a given list with x at the end.
Singly-Linked Lists w/out Interface

When an object contains a reference to an object of the same type, then several objects can be linked together into a list. Each object refers to the next object.

For a list to be useful, there must be a variable that points to the first node in the list. Here, the variable named "head" serves this purpose.
Java does not support recursion as efficiently as Racket, Scheme, etc. Each recursive call puts another method frame on the method-call-stack. This area of memory is not infinite and can quickly fill up in a loop.

To traverse a list in Java, it is common practice to treat each node in the list as the same type and traverse the list using references (i.e. sometimes called "pointers"). The traversal is driven by a while loop that processes until the current node is null.
Traversing Singly-Linked Lists

Sometimes this traversal gets messy because you must account for the case when the list is null and when it is not null.

Calling a method on null always results in an error.
The SLIntList class represents a singly-linked list of an inner class called INode, a container for integers.

```java
public class SLIntList {
    // Inner class represents each node in list of ints
    public class INode {
        private int item;
        private INode next;
        // constructor
        INode(int i, INode n) {
            this.item = i;
            this.next = null;
        }
    } // end of INode inner class
    INode head = null;
    // remainder on next slides>
    ```
public void add(int i) {
    INode newNode = new INode(i, null);
    if (head == null) { // should only occur at start
        head = newNode;
    }
    else { // more nodes are attached in a chain to head
        newNode.next = head;
        head = newNode;
    }
    // No need to return head...it is global
} // end of cons method
Creating an SLIntList by adding to end

```java
public void append(int i) {
    INode newNode = new INode(i, null);
    INode runner = head; // set runner to head
    if (runner == null) {
        head = newNode;
    } else {
        while (runner.next != null) {
            runner = runner.next;
        }
        runner.next = newNode; // put newNode at end
    } // here, runner.next is null...end of list
} // end of append method
```
public int sumSLIntList() {
    int sum = 0;
    INode runner = head;
    while ( runner != null ) {
        sum += runner.item;
        // Add current item to the sum.
        runner = runner.next;
        // Move pointer to next item in
        // in list.
    }
    return sum;
}
Printing items in an SLIntList

```java
public void printSLIntList() {
    INode runner = head;

    while ( runner != null ) {
        System.out.println(runner.item);
        // Print current item and move on.
        runner = runner.next;
    }
}
```

Printing all the items in the list also requires an INode to traverse the list.
public void insert(int i) {
    INode newNode = INode(i, null);
    INode current = head;
    INode previous = head;

    while (current != null && current.item < i) {
        // move on
        previous = current;
        current = current.next;
    } // either current==null or current.item >= i
    if (previous == null) {
        newNode.next = current;
        head = newNode;
    }
    else if (previous != null) {
        newNode.next = current;
        previous.next = newNode;
    }
}
INode can implement a generic interface

// Inner class represents each node in list of ints
public class INode implements Comparable<INode> {
    // Comparable<INode> interface requires that all
    // items in the list must be INodes and that the
    // inner class must provide an implementation of
    // the compareTo method.
    int item;
    INode next;
    // constructor
    INode(int i, INode n) {
        this.item = i;
        this.next = null;
    }

    // see next slide for code here.
}

} // end of INode inner class
public int compareTo(INode o) {
    int cNum = 0;
    if (this.item > o.item) {  // no need to cast o, need to cast this
        cNum = 1;              // the compiler knows it is an INode
    }
    else if (this.item < o.item) {
        cNum = -1;
    }
    else {
        cNum = 0;
    }
    return cNum;
}

} // end of INode inner class
public void insert(int i) {
    INode newNode = INode(i, null);
    INode current = head;
    INode previous = null;

    while(current!=null && current.compareTo(newNode)<0){
        // move on
        previous = current;
        current = current.next;
    }  // either current==null or current.item >= i
    if (previous == null) {
        newNode.next = current;
        head = newNode;
    }
    else if (previous != null) {
        newNode.next = current;
        previous.next = newNode;
    }
}

Java Parameterized (generic) Types

1. Makes more bugs detectable at compile time

2. Types of objects in a container are specified by arguments inside <>s.

3. Eliminates need for some type-casting.
ArrayList\textless{}E\textgreater{}

Parameterized type that removes the complexity of dealing with references in a linked list.

ArrayList is implemented as a dynamic array because each item must be accessed quickly.

When you use an ArrayList, you should specify the type of object it can hold.
You can create a new ArrayList object without specifying the type, but everything removed from the ArrayList is considered to be type Object and must be type cast.

```java
ArrayList al = new ArrayList();
```

Legal, but not as easy to deal with as a parameterized ArrayList, such as

```java
ArrayList<Integer> intList = new ArrayList<Integer>();
```

One of the reasons wrapper types like the Integer class were developed was so that primitive types could be stored in containers such as ArrayList.
ArrayList\(<E>\>

- boolean add(E): appends item to end of list
- void add(i, E): inserts the item E at position i
- boolean contains(o): Returns true if o in list
- E get(i): Returns element at position i in list
- int indexOf(o): Returns index position of o in list
- int size(): Returns number of objects in list