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

March 9, 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.
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);
}
/*
 * MTList.java
 * Represents an empty list
 */

public class MTList 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 string representation of this list
    public int length() {
        return 0;
    }

    // Returns 0: no ints in this IList
    public int sum() {
        return 0;
    }
}

attach new int to this IList
nothing to double, return this IList
empty list has 0 elements
nothing to add to sum
// Returns a String representation of this MTList
public String toString(){
    return "()";
}

Write a predicate method that returns true if a given int is in the IList.

Write a method that removes a given number from an IList.
/*  
* ConsList.java  
*  
* Represents a pair, or cons cell, with fields  
* for first and rest.  
*/

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);
    }

}
// Doubles each number in this IList, producing a new IList
public IList doubleList() {
    return new ConsList(this.first * 2, rest.doubleList());
}

// Returns number that results from adding all list numbers
public int sum() {
    return this.first + this.rest.sum();
}

// Returns an integer equal to the length of this list
public int length() {
    return 1 + this.rest.length();
}

// Returns a String representation of this ConsList
public String toString() {
    return "("+this.first+" "+ this.rest.toString()+" ) ");
}
public class TestIList {

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

    // class to generate random numbers
    private Random generator;

    //List to contain random numbers
    private IList numbers;
/* Creates a new instance of TestIList */
public TestIList() {
    this.generator = new Random();
    this.numbers = new MTList();
    // 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());

// Cons a new number onto the IList
System.out.println("\nCons number 100 onto front of list:");
System.out.println(this.numbers.cons(100));
}

/**
 * Returns a random number between 0 and 99
 */
private int genNumber() {
    return Math.abs(this.generator.nextInt() % 100);
}

/**
 * main method of program
 */
public static void main(String[] args) {
    new TestIList();

}
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.
Singly-Linked Lists

The common pattern is to start at the head of the list (usually called 'Head'), then move a pointer from each node to the next by following the pointer in the node, stopping when the null that marks the end of the list is reached (i.e., when pointer == null).

Node runner;
// A pointer that will be used to traverse the list.
runner = head;
// Start with runner pointing to the head of the list.
// Continue until null is encountered.
while ( runner != null ) {
    System.out.println( runner.item );
    // Do something with the item in the current node.
    runner = runner.next;
    // Move on to the next node in the list.
}
Singly-Linked Lists

A version of the code on the last slide with a for loop that has different syntax than usual.

Node runner;
// A pointer that will be used to traverse the list.
runner = head;

// Start with runner pointing to the head of the list.
// Uses the pointer runner as the loop counter.
for ( Node runner = head;
     runner != null; runner = runner.next ) {
    System.out.println( runner.item );
}
A Node in a SLL

// Imagine this code is inside a class called
// SinglyLinkedList

public class IntNode {
    int item; // One of the integers in list.
    IntNode next; // Pointer to the next node in list.
}

public void intNodeSum() {
    int sum = 0;
    IntNode runner = head;
    while (runner != null) {
        sum = sum + runner.item;
        // Add current item to the sum.
        runner = runner.next;
    }
    System.out.println("The sum of the list is " + sum);
}