**Lesson 6 – Abstract Data Types II**

**Learning Objectives:**

* Describe the purpose and benefits of Abstract Data Types.
* List the benefits of modularity, procedural abstraction, and information hiding and describe how they are enforced in Java OO.
* List and describe the operations of the ADT List.
* Define a Java Interface for an ADT.
* Implement an array-based implementation of an ADT List.
* Implement an array to store items of type Object.

**Reading:**

* Read Chapter 4, Section 1-2.
* Chapter 4, Section 3 of the text.

***Abstract Data Types***

* **Modularity** is a technique that keeps the complexity of a large program manageable by systematically controlling the interaction of its components.
  + A modular program is easier to write, read, and modify because you can focus on one task at a time in a modular program without other distractions.
  + Modularity also isolates errors and eliminates redundancies.
* Modularity, thus, allows for **procedural abstraction**; that is, you can write the methods in relative isolation from one another, knowing what each one will do but not necessarily how each will eventually do it.

A picture containing black, darkness

Description automatically generated

* The principle of **information hiding** involves identifying details that you can hide within a module while writing a module’s specifications and then, **not only hiding these details, but also making them *inaccessible* from outside a module**.
  + One way to understand information hiding is to imagine walls around the various tasks a program performs. These walls prevent the tasks from becoming entangled.
  + The wall around each task *T* prevents the other tasks from “seeing” how *T* is performed. Thus, if task *Q* uses task *T*, and if the method for performing task *T* changes, task *Q* will not be affected.
  + The isolation of the modules cannot be total, however. Although task *Q* does not know *how* task *T* is performed, it must know *what* task *T* is and how to initiate it. What goes in and comes out of a module is governed by the terms of the method’s specifications, or **contract**: *If you use the method in this way, this is exactly what it will do for you*.
* **Data abstraction** asks that you think in terms of *what* you can do to a collection of data independently of *how* you can do it.
  + Data abstraction is a technique that allows you to develop each data structure in relative isolation from the rest of the solution.
  + The other modules of the solution will “know” what operations they can perform on the data, but they should not depend on how the data is stored or how the operations are performed.
  + Again, the terms of the contract are *what* and not *how*. Thus, data abstraction is a natural extension of procedural abstraction.
* An **abstract data type (ADT)** is a collection of data together with a set of operations on that data.
  + The ADT operations should not specify how the data is stored.
  + The description of an ADT’s operations must be rigorous enough to specify completely their effect on the data, yet it must not specify how to store the data nor how to carry out the operations.
  + Recall that a **data structure** is a construct that you can define within a programming language to store a collection of data. You choose a particular data structure when you implement an ADT.
  + When a program must perform data operations that are not directly supported by the language, you should first design an abstract data type and carefully specify what the ADT operations are to do (the contract). Then—and only then—should you implement the operations with a data structure.

**Specifying ADTs:**

* Let’s look deeper at this idea of a list (ignoring the Java implementations). Consider a list that you might encounter, such as a list of chores, a list of important dates, a list of addresses, or a grocery list.
* Except for the first and last items, each item has a unique **predecessor** and a unique **successor**. The first item—the **head** or front of the list—does not have a predecessor, and the last item—the **tail** or end of the list—does not have a successor.
* ADT list operations:
  + Create an empty list.
  + Determine whether a list is empty.
  + Determine the number of items on a list.
  + Add an item at a given position in the list.
  + Remove the item at a given position in the list.
  + Remove all the items from the list.
  + Retrieve (get) the item at a given position in the list.
* Pseudocode for the ADT List Operations

+createList()

// Creates an empty list.

+isEmpty() : boolean {query}

// Determines whether a list is empty.

+size() : integer {query}

// Returns the number of items that are in a list.

+add(in index : integer, in item : ListItemType)

// Inserts item at position index of a list, if

// 0 <= index <= size(). If index < size(),

// items are renumbered as follows: the item at

// index becomes the item at index+1; the item at

// index+1 becomes the item at index+2; and so on.

// Throws an exception when index is out of range

// or if the item cannot be placed on the list

// (list full).

+remove(in index : integer)

// Removes the item at position index of a list,

// if 0 <= index < size(). If index < size()-1,

// items are renumbered as follows: the item at

// index+1 becomes the item at index; the item

// at index+2 becomes the item at index+1; and

// so on.

// Throws an exception when index is out of range

// or if the list is empty.

+removeAll()

// Removes all the items in the list.

+get(in index : integer) : ListItemType {query}

// Returns the item at position index of a list if

// 0 <= index < size(). The list is left unchanged

// by this operation.

// Throws an exception if index is out of range.

* What does the specification of the ADT list tell you about its behavior? It is apparent that the list operations fall into three broad categories:
  + The operation add **adds** data to a data collection.
  + The operations remove and removeAll **remove** data from a data collection.
  + The operations isEmpty, size, and get **ask questions** about the data in a data collection.
* The specifications contain no mention of how to store the list or how to perform the operations; they tell you only what you can do to the list.
* It is of fundamental importance that the specification of an ADT *not* include implementation issues.
  + This restriction on the specification of an ADT is what allows you to build a wall between an implementation of an ADT and the **client** (the program that uses it).
  + The behavior of the operations is the only thing on which a program should depend.
* Once you have satisfactorily specified the behavior of an ADT, you can design applications that access and manipulate the ADT’s data solely in terms of its operations and without regard for its implementation.

**Java Interface for a List ADT :**

public abstract class **ListADT**<E> {

    public abstract boolean isEmpty();

    public abstract int size();

    public abstract void removeAll();

    public abstract void add(int index, E item) throws ListException;

    public abstract E get(int index) throws ListException;

    public abstract void remove(int index) throws ListException;

}

**Custom Exceptions for the Interface and Implementation:**

**public** **class** ListException **extends** RuntimeException {

**public** ListException(String s)

{

**super**(s);

}

}

**Favorites and Friends**

* **Favorites – up to 50, videos with name and link**
* **Friends – unlimited with username and status**
* import java.net.MalformedURLException;
* import java.net.URL;
* import java.util.Date;
* public class Favorite {
* Date date;
* String videoName;
* URL url;
* public Favorite(String name, String address) throws MalformedURLException {
* this.videoName = name;
* this.url = new URL(address);
* this.date = new Date();
* }
* public String toString() {
* String s = this.videoName + " " + this.date + "\n";
* return s + url.toString();
* }
* }

public class Friend {

    private String username;

    private boolean status = false; // false - offline; true online

    public Friend(String username, boolean status) {

        this.username = username;

        this.status = status;

    }

    public Friend(String username) {

        this(username, false);

    }

    public void setStatus(boolean online) {

        this.status = online;

    }

    @Override

    public String toString() {

        String s = this.status ?  "Online" : "Offline";

        return username + " " + s;

    }

}

**Java Array-based Implementation for a List ADT for Integers:**

public class **FavoriteList** extends ListADT<Favorite> {

    private static final int MAX\_FAVORITES = 50;

    private Favorite[] favorites;

    private int numFavorites;

    public FavoriteList() {

**this**.numFavorites = 0;

**this**.favorites = new Favorite[MAX\_FAVORITES];

    }

    @Override

    public boolean isEmpty() {

        return **this**.numFavorites == 0;

    }

    @Override

    public int size() {

        return **this**.numFavorites;

    }

    @Override

    public void removeAll() {

**this**.favorites = new Favorite[MAX\_FAVORITES];

**this**.numFavorites = 0;

    }

    @Override

    public void add(int index, Favorite item) throws ListException {

        if (index < 0 || index > **this**.size())

            throw new ListException("Index " + index + " is invalid for a list of size " + **this**.size());

        if (index >= MAX\_FAVORITES)

            throw new ListException("Index exceeds maximum allowed size of the list: " + MAX\_FAVORITES);

        for (int i = **this**.size(); i > index; i--)

**this**.favorites[i] = **this**.favorites[i - 1];

**this**.favorites[index] = item;

**this**.numFavorites++;

    }

    @Override

    public Favorite get(int index) throws ListException {

        if ((index < 0) || (index >= **this**.size()))

                throw new ListException("Index " + index + " is invalid for a list of size " + **this**.size());

        return **this**.favorites[index];

    }

    @Override

    public void remove(int index) throws ListException {

        if ((index < 0)||(index >= **this**.size()))

                throw new ListException("Index " + index + " is invalid for a list of size " + **this**.size());

            for (int position = index + 1; position < **this**.size(); position++)

**this**.favorites[position - 1] = **this**.favorites[position];

**this**.numFavorites--;

    }

    @Override

    public String toString() {

        String s = "";

        for (int i = 0; i < **this**.size(); i++) {

            s = s + i + ": " + **this**.get(i) + "\n";

        }

        return s;

    }

}

**We Can Use Our List Implementation Like This:**

import java.net.MalformedURLException;

public class **TestFavoriteList** {

    public static void main(String[] args) throws ListException, MalformedURLException {

        FavoriteList favList = new FavoriteList();

        System.out.println("Testing add: ");

        favList.add(0, new Favorite("Grounded", "https://www.youtube.com/watch?v=\_BqgJ9nW468"));

        favList.add(1, new Favorite("Vampire Survivor", "https://www.youtube.com/watch?v=6HXNxWbRgsg"));

        favList.add(2, new Favorite("High on Life", "https://www.youtube.com/watch?v=NyfneSMsr5U"));

        System.out.println(favList);

        System.out.println();

        System.out.println("Testing remove: ");

        favList.remove(1);

        System.out.println(favList);

        System.out.println();

        System.out.println("Testing remove all: ");

        favList.removeAll();

        System.out.println(favList);

    }

}

**The Java equals method:**

A couple of people used the Java equals method to compare different objects. Unfortunately, this does not work unless you override the equals method.

* equals will only compare what it is written to compare, no more, no less.
* if a class does not override the equals method, then it defaults to the equals(Object o) method of the closest parent class that has overridden this method.
* If no parent classes have provided an override, then it defaults to the method from the ultimate parent class, Object, and so you're left with the Object.equals(Object o) method. Per the Object API this is the same as ==; that is, it returns true if and only if both variables refer to the same object, if their references are one and the same. Thus you will be testing for object equality and not functional equality.
* Always remember to override hashCode if you override equals so as not to "break the contract". As per the API, the result returned from the hashCode method for two objects must be the same if their equals methods shows that they are equivalent. The converse is not necessarily true.