CSCI 3302 Lesson 1 – Introduction to Data Structures

Learning Objectives:

- Understand the purpose and organization of the course.
- Define abstraction and information hiding.
- Describe the concept of an Abstract Data Type (ADT)

Readings:

- Review Chapter 1 of the text.
- Read Chapter 2, Section 1 of the text.

Course Admin (30 minutes):

- Slides – Lesson 1

- What is a data structure?
    - What this course is about and how it fits into total CS education.
    - "What is a data structure?"
    - Example: Library setting. How are the books organized? Alphabetically? Topically? By the color of the book jacket?
        - In a bookstore, fiction is arranged alphabetically by author, whereas non-fiction is typically arranged topically.
        - There is an intuition about how someone might want to find books. If they liked the last Stephen King novel, perhaps they are searching for more books by this author. If someone is looking a travel guide for their trip to Italy it would be helpful for all these types of books to be on the same shelf.
    - The analog in computer science is that when we want to access data within a program, we would like it to be organized efficiently based on how we intend to use it. Do we need to access all of our data sequentially? Do we need to be able to quickly look-up a specific piece of data (or perhaps we just want to know if a piece of data is there or not)?

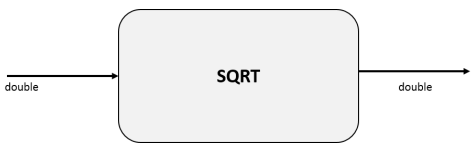
- Review course syllabus:
    - Summarize course objectives
        - Briefly summarize the course objectives. They really boil down to 3 main points. We want students to be able to
            1. Design/implement data structures on their own,
            2. Use appropriate data structures to solve problems (esp. those that would come standard with a high-level programming language), and
            3. Become a better programmer overall.
    - Prerequisites
    - Office hours
    - Textbook and readings
    - Homework
        - Must compile/run for any credit
        - Individual effort/Paired Programming

    - Student expectations
- HW: Watch videos and load first assignment

Lesson (45 minutes):

- **Problem Solving and Software Engineering:**
    - **Problem solving** refers to the entire process of taking the statement of a problem and developing a computer program that solves that problem.
        - Typically, a **solution** consists of two components: algorithms and ways to store data.
        - When constructing a solution, you must organize your data collection so that you can operate on the data easily in the manner that the algorithm requires.
        - Most of what we will do in this course is about how to organize the data.
    - An **algorithm** is a step-by-step specification of a method to solve a problem within a finite amount of time.
        - One action that an algorithm often performs is to operate on a collection of data

- **Abstraction and Information Hiding:**
    - What is abstraction? We can think about how a car works at different levels of abstraction.
        - At the level of the operator, we insert the key and turn, use the steering wheel, foot pedals, and gear shift.
        - At the level of the mechanic, we understand how the various parts work together. The fuel injection system loads the cylinders with gasoline; the spark plugs ignite the fuel which turn the pistons; this turns the crankshaft which turn the wheels, etc.
        - At the level of the physicist, we understand the chemical process of adding a spark to fuel; we can calculate the force generated; we can measure the heat build-up and the friction of the piston in the cylinder.
    - When you go through the drive-through at McDonalds, you probably don't care exactly how many cooks are on duty, or the arrangement of the patties, lettuce, and tomatoes around the griddle. What you care about is the input (the cost of the food; the time spent getting it) and the output (the quality of the food; the friendliness of the cashier).
    - **Procedural abstraction** separates the purpose of a method from its implementation.
        - The idea here is that what is done is important and how it's done is not as important.
        - Think about the Math.sqrt() command. We can visualize it as a black box where we provide a number as input and we get a number as output.



We don't need to know how it works in order to use it. We could, perhaps, write our own method using an implementation of the Newton-Raphson method. But even then, we may not have a good understanding of how the method is actually operating at the machine level (i.e., what are the machine instructions that were generated by the compiler). We may not understand how all of this is actually being carried out at the physical level with electrons and silicon gates. The ability to abstract away these lower details makes our job easier as a programmer.

- Going up in abstraction, instead of down, think about a large programming project with multiple programmers. Procedural abstraction is essential to team projects. After all, in a team situation, you will have to use methods written by others, frequently without knowledge of their algorithm.

- **Data abstraction** focuses on what the operations do instead of on how you will implement them. In this form of abstraction, instead of just focusing on operations, we focus on data first and then the operations that manipulate the data.

- An **Abstract Data Type (ADT)** is a collection of data and a set of operations on the data.
  - Enables you to focus on what operations you will perform on the data instead of how you will perform them.

  - Ultimately, someone—perhaps you—will implement the ADT by using a data structure, which is a construct that you can define within a programming language to store a collection of data.

  - The principle of **information hiding** helps you to think about what details of a module should be visible from the outside, or **public**, view and what details should be kept **private**.
    - Information hiding limits the ways in which you need to deal with methods and data.

    - As a user of a module, you do not worry about the details of an implementation.

    - As an implementer of a module, you do not worry about its uses.

D2L Brightspace (10 minutes):

- Organization of course website on Brightspace

Lesson 02

Learning Objectives:

- Familiarize with the SDLC.
- Use GitHub for assignments.
- Review double arrays and loops.
- Understand how to work with RGB.

- Go to D2L, GitHub and select PP01.
    - Load Assignment
    - Go over VS Code and Loading Project
        - Different Files
        - Problems/TODOs
        - Committing Changes

- **Double arrays review**
    - Syntax: `type[][] = new type[num_rows][num_cols]`
    - Example: `int[][] grades = new int[20][5]`
        - Say for 20 students each with 5 grades
        - Traversal – TestIM.java print array in repo PP01-Assignment

- Colors – Slides

- ImagesManipulator

```
public int[][] grayScale() {
        // TODO (in class): average the R G B values
        int w = pixels.length;
        int h = pixels[0].length;

        int[][] newPixels = new int[w][h];
        for (int x = 0; x < w; x++)
            for (int y = 0; y < h; y++) {
                // average the red, blue, and green values
                // of the pixel and set the corresponding
                // grayscale pixel to this average value
                Color color = new Color( this.pixels[x][y]);

                int red = color.getRed();
                int blue = color.getBlue();
                int green = color.getGreen();
                int gray = (red + blue + green) / 3;

                newPixels[x][y] = new Color(gray,gray,gray).getRGB();
            }

        return newPixels;
 }

public int[][] filter(int mask) {

        // TODO (in class): use & to filter by mask
        int w = pixels.length;
        int h = pixels[0].length;

        int[][] newPixels = new int[w][h];
        for (int x = 0; x < w; x++)
            for (int y = 0; y < h; y++) {
                int color = this.pixels[x][y];
                newPixels[x][y] = mask & color;
            }

        return newPixels;
```

- **The Software Lifecycle:**
  - The development of good software involves a lengthy and continuing process known as the software's **life cycle**:
    - Specification:
      - Given an initial statement of the software's purpose, you must specify all aspects of the problem.
      - The specification phase requires that you bring precision and detail to the original problem statement and that you communicate with both programmers and nonprogrammers.

    - Design:
      - Once you have completed the specification phase, you must design a solution to the problem.
      - The best way to simplify the problem-solving process is to divide a large problem into small, manageable parts. The resulting program will contain **modules**, which are self-contained units of code.
      - Classes should be designed so that the objects are independent, or loosely coupled. Coupling is the degree to which objects in a program are interdependent.
      - Classes should also be designed so that objects are highly cohesive. Cohesion is the degree to which the data and methods of an object are related.
      - Methods should also be highly cohesive; each should perform one well-defined task.
      - Ideally, each object should represent one component in the solution.
      - Objects interact by sending messages to each other through method calls, which in turn represents the **data flow** among objects.
        - You should specify in detail the assumptions, input, and output for each method.
        - You can view these specifications as the terms of a **contract** between your method and the code that calls it.

    - Implementation
      - The coding phase involves translating the design into a particular programming language and removing the syntax errors.

    - Testing/Refining
      - During the testing phase, you need to remove as many logical errors as you can. One approach is to test the individual methods of the objects first, using valid input data that leads to a known result.
      - Often the best approach to solving a problem is first to make some simplifying assumptions during the design of the solution and next to develop a complete working program under these assumptions. You can then add more sophisticated input and output routines, additional features, and more error checks to the working program.
      - Also, realize that any time you modify a program—no matter how trivial the changes might seem—you must thoroughly test it again.

- Production/Deployment
  - When the software product is complete, it is distributed to its intended users, installed on their computers, and used.

- Maintenance
  - Users of your software invariably will detect errors that you did not discover during the testing phase. Correcting these errors is part of maintaining the software.
  - Another aspect of the maintenance phase involves enhancing the software by adding more features or by modifying existing portions to suit the users better.

Lesson 3 – Recursion I

Lesson Objectives:

- Define recursion.
- Know the parts of a recursive algorithm.
- Trace through the execution of a simple recursive method.
- Write a simple recursive method.

Readings:

- Read Chapter 3, Section 1.

Recursive methods:

A method that calls itself is said to be recursive. A method **f1** is also recursive if it calls a method **f2**, which under some circumstances calls **f1**, creating a cycle in the sequence of calls.

Problems that lend themselves to a recursive solution have the following characteristics:

- One or more simple cases of the problem have a straightforward, non-recursive solution.
- The other cases can be redefined in terms of problems that are closer to the simple cases.
- By applying the redefinition process every time the recursive method is called, eventually the problem is reduced entirely to simple cases, which are relatively easy to solve.

Recursive algorithms generally consist of an if statement in the form:

```
if this is a simple case
    solve it
else
    redefine the problem using recursion
```

```
// calculate the factorial, n!
```

```
public static int fact(int x)
{
      if (x == 0)
            return 1;
      else
            return x * fact(x - 1);
}
```

Exercise: Write a method that recursively sums numbers from 0 to n.

$$sum(n) = \begin{cases} 0 & if\, n = 0 \\ n + sum(n-1) & otherwise \end{cases}$$

```
// Write a method that recursively sums numbers from 0 to n.
public static int sum_seq(int x)
{
      if (x == 0)
            return 0;
      else
            return x + sum_seq(x - 1);
}
```

Multiple recursive calls:

A recursive method sometimes makes multiple recursive calls: within the same

$$f(x) = \begin{cases} 1 & if\, x = 0 \\ 1 & if\, x = 1 \\ f(x-1) + f(x-2) & otherwise \end{cases}$$

```
// calculate the nth Fibonacci number
public static int fib(int n) {
      if ((n == 0) || (n == 1))
            return 1;
      else
            return fib(n - 1) + fib(n - 2);
}
```

Head recursion vs. tail recursion:

- We can also organize recursion by head recursion and tail recursion:
    - In head recursion, the recursive call when it happens, comes before other processing in the function

if this is a simple case

solve it
else
recursive call
process data

- In tail recursion, it's the opposite; the processing occurs before the recursive call.

if this is a simple case
solve it
else
process data
recursive call

- Choosing between the two styles may seem arbitrary, but the choice can make a significant difference.

- For example:

```
public static void head(int n) {
  if (n == 0)
    System.out.println(n);
  else {
    head(n - 1);
    System.out.println(n);
  }
}


public static void tail(int n){
  if (n == 0)
    System.out.println(n);
  else {
    System.out.println(n);
    tail(n - 1);
  }
}

public static void main(String[] args) {
  head(5);

  System.out.println();

  tail(5);
}
```

- ***Recursively processing an attribute array:***
  - We can use private helper methods to provide a cleaner signature for client code.
    - We do this when we want to use recursion, but there is a difference between the parameters needed for the recursion and the parameters needed for the client code.

  - Though we can recursively process an array that is passed as an argument, we can also recursively process an array that is a class attribute:

```
// An array attribute of the class
```

```
private int[] intArray;

// Constructor creating the array attribute
public ConstructorName() {
     intArray = { 23, 34, 45, 56, 67, 78 };
}

// calculate the sum of the elements of an array using head recursion
private static int sumByHead(int index) {
    if (index == intArray.length - 1)
      return intArray[index];
    else
      return intArray[index] + sumByHead(index + 1);
}

public static int sumByHead() {
    return sumByHead(0);
}

// calculate the sum of the elements of an array using tail recursion
private static int sumByTail(int index, int s) {
    if (index == intArray.length - 1)
      return s + intArray[index];
    else
      return sumByTail(index + 1, s + intArray[index]);
}

public static int sumByTail() {
    return sumByTail(0, 0);
}
```

- In this example, we are assuming that all locations in the array contain relevant integers. Also, the array to be processed is passed as an argument to the method.

- ***In-class Exercise:***
  - Write a method that uses head recursion to find the maximum value in an array.

  - Write a method that uses tail recursion to find the maximum value in an array.

```
private int findMaxHR(int index) {
    if (index == intArray.length - 1)
       return intArray[index];
    else {
      int maxFromRest = findMaxHR(index + 1);
       if (maxFromRest > intArray[index])
         return maxFromRest;
       else
          return intArray[index];
     }
}

public static double findMaxHR() {
```

```
    return findMaxHR(values, 0);
}


private static double findMaxTR(int index, int maxSoFar) {
    if (index == intArray.length)
       return maxSoFar;
    else {
      if (intArray[index] > maxSoFar)
         return findMaxTR(index + 1, values[index]);
      else
         return findMaxTR(values, index + 1, maxSoFar);
      }
}



public static double findMaxTR(double[] values) {
     return findMaxTR(values, 0, 0);
}
```

Lesson 4 – Recursion II

Lesson Objectives:

- Describe the difference between head recursion and tail recursion.
- Write a function that employs head recursion.
- Write a function that employs tail recursion.
- Explain the utility of using helper methods with recursion.
- Describe how to conduct a recursive binary search.

Readings:

- Read Chapter 3, Sections 2-5.

Lesson:

- ***Recursively processing an attribute array:***
    - We can use private helper methods to provide a cleaner signature for client code.
        - We do this when we want to use recursion, but there is a difference between the parameters needed for the recursion and the parameters needed for the client code.

    - Though we can recursively process an array that is passed as an argument, we can also recursively process an array that is a class attribute:

```
// An array attribute of the class
private int[] intArray;
```

```
// Constructor creating the array attribute
public ConstructorName() {
     intArray = { 23, 34, 45, 56, 67, 78 };
}

// calculate the sum of the elements of an array using head recursion
private static int sumByHead(int index)
{
    if (index == intArray.length - 1)
      return intArray[index];
    else
      return intArray[index] + sumByHead(index + 1);
}

public static int sumByHead()
{
    return sumByHead(0);
}

// calculate the sum of the elements of an array using tail recursion
private static int sumByTail(int index, int s)
{
    if (index == intArray.length - 1)
      return s + intArray[index];
    else
      return sumByTail(index + 1, s + intArray[index]);
}

public static int sumByTail()
{
    return sumByTail(0, 0);
}
```

- In this example, we are assuming that all locations in the array contain relevant integers. Also, the array to be processed is passed as an argument to the method.

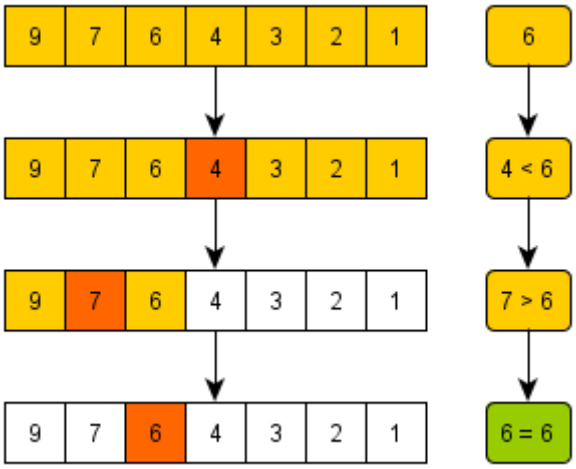
- ***In-class Exercise:***
  - Write a method that uses head recursion to find the maximum value in an array.

  - Write a method that uses tail recursion to find the maximum value in an array.

```
private int findMaxHR(int index)
{
    if (index == intArray.length - 1)
       return intArray[index];
    else {
      int maxFromRest = findMaxHR(index + 1);
       if (maxFromRest > intArray[index])
         return maxFromRest;
       else
          return intArray[index];
     }
```

```
}

public static double findMaxHR()
{
    return findMaxHR(values, 0);
}


private static double findMaxTR(int index, int maxSoFar)
{
    if (index == intArray.length)
       return maxSoFar;
    else {
      if (intArray[index] > maxSoFar)
         return findMaxTR(index + 1, values[index]);
      else
         return findMaxTR(values, index + 1, maxSoFar);
      }
}



public static double findMaxTR(double[] values)
{
     return findMaxTR(values, 0, 0);
}
```

- ***Recursively Searching a Sorted Array using Binary Search:***
    - Suppose that you have a sorted array `lsnArray` of integers and you want to find a specific integer. You could construct an iterative solution without too much difficulty, but instead consider a recursive formulation that is actually more efficient than examining every item:



```
// A recursive binary search helper function. It returns
// location of x in given array arr[l..r] is present,
// otherwise -1
private static int binarySearch(int[] arr, int l, int r, int x)
{
    if (r >= l) {
        int mid = l + (r - l) / 2;

        // If the element is present at the middle
        // itself
        if (arr[mid] == x)
            return mid;

        // If element is smaller than mid, then
        // it can only be present in left subarray
        if (arr[mid] > x)
            return binarySearch(arr, l, mid - 1, x);

        // Else the element can only be present
        // in right subarray
        return binarySearch(arr, mid + 1, r, x);
    }

    // We reach here when element is not
    // present in array
    return -1;
}

// The public method for conducting a binary
// search of the array arr.
public static int binarySearch(int[] arr, int x)
{
    return binarySearch(arr, x);
}
```

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.

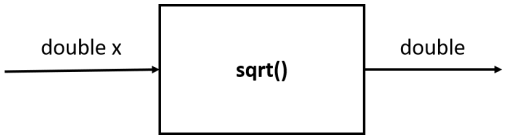
Reading:

- 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.



- 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();
-     }
- }
-
```

```java
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.

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.

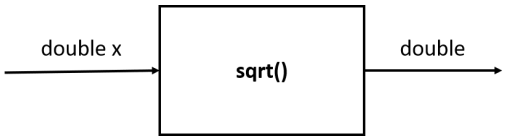
Reading:

- 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.



- 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();
-       }
-   }
-
```

```java
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.

Lesson 8 – Linked Lists II

Learning Objectives:

- Use a node class to build a reference-base list (linked list).
- Write methods to insert and remove items within a linked list.
- Implement a reference-based linked-list in Java for storing integers.
- Explain alternative variations of the linked list.

Reading:

- Chapter 5, Section 3-4 of the text.

The Linked-List:

- A close examination of the array-based implementation of the ADT list reveals that an array is not always the best data structure to use to maintain a collection of data. An array has a fixed-size—at least in most commonly used programming languages—but the ADT List can have an arbitrary length.

- Also, an array orders its items physically and, you must shift data when you insert or delete an item at a specified position. Shifting data can be a time-consuming process that should be avoided, if possible.

- Like you implemented in the set, we can modify the array implementation so that, when the array is full, we create a new, larger array and copy the contents over. This process is time consuming, as well.

- We want a list implementation that does not involve shifting and copying data to a new array.

Object References:

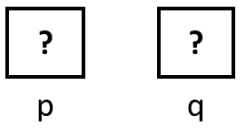
- Note that an object of a given class does not come into existence until you apply the **`new`** operator.

- When you declare a variable that refers to the object, you are creating a **reference** to the object. A **reference variable**, or simply a **reference**, contains the location, or **address** in memory, of an object.

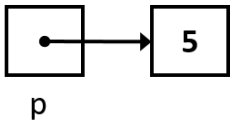
- By using a reference to a particular object, you can locate the object and, for example, access the object's public members.

- For example:

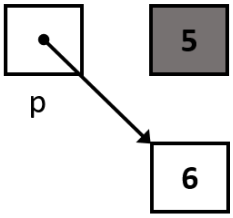
Integer p;
Integer q;



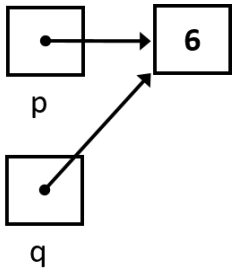
p = **new** Integer(5);



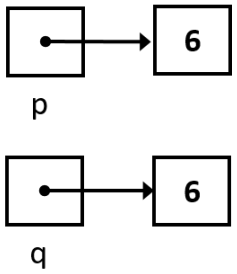
p = **new** Integer(5);



q = p



q = **new** Integer(9);



- Key Concepts:
  - The declaration

    ```
    Integer intRef;
    ```

    statically allocates a reference variable `intRef` whose value is `null`. When a reference variable contains `null`, it does not reference anything.
  - `intRef` can reference an `Integer` object. The statement

```
intRef = new Integer(5);
```

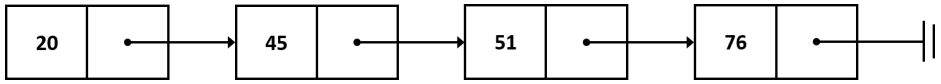
dynamically allocates an `Integer` object referenced by `intRef`. (However, see item 3 on the list.)

- If, for some reason, `new` cannot instantiate an object of the class represented, it may throw a `java.lang.InstantiationException` or a `java.lang.IllegalAccessException`. Thus, you can place the following statement within a try block to test whether memory was successfully allocated:

```
intRef = new Integer(5);
```

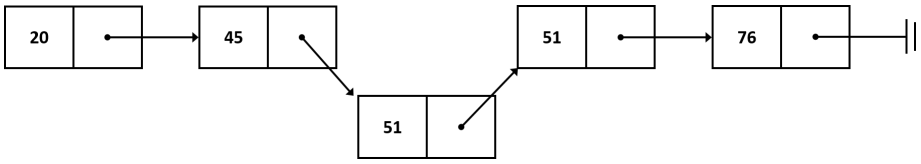
- When the last reference to an object is removed, the object is marked for garbage collection.

- To get a conceptual notion of a list implementation that does not involve shifting, consider the following figure:

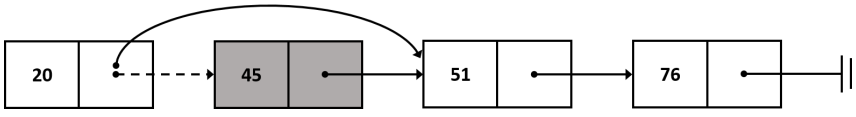


- In the diagram, each item of the list is actually linked to the next item. Thus, if you know where an item is, you can determine its successor, which can be physically located anywhere in memory.

- This flexibility not only allows you to insert and delete data items without shifting data, but it also allows you to increase the size of the list easily.
  - If you need to insert a new item, you simply find its place in the list and set two **links**.



  - Similarly, to delete an item, you find the item and change a link to bypass the item.



- Because the items in this data structure are linked to one another, it is called a **linked list**. Using this kind of structure, a linked list is able to grow as needed. In many applications, this flexibility gives a linked list a significant advantage.

Reference-Based Linked Lists:

- A linked list contains components that are linked to one another. Each component—usually called a node—contains both data and a "link" to the next item.

The Node Class:

```
   public class Node<E> {

    private E item;
    private Node<E> nextNode;

    public Node(E nodeItem) {
        this.item = nodeItem;
        this.nextNode = null;
    }

    public E getItem() {
        return this.item;
    }
    public void setNext(Node<E> nextNode){
        this.nextNode = nextNode;
    }
    public Node<E> getNext() {
        return this.nextNode;
    }

    @Override
// Wait until the end of class to implement this
// method in order to demonstrate recursion within
// the Node class.
    public String toString() {
        if (this.nextNode == null)
            return "(" + item.toString() + ", null";
        else
            return "(" + item.toString() + ", " + nextNode.toString() +
")";
    }

}
```

```java
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;
    }
}
```

The Linked-List:

```java
public class FriendsList extends ListADT<Friend> {
    private Node<Friend> head;
    private int numFriends;

    public FriendsList() {
        this.head = null;
        this.numFriends = 0;
    }

    public FriendsList(Friend firstFriend) {
        this.head = new Node<Friend>(firstFriend);
        this.numFriends = 1;
    }

    @Override
    public boolean isEmpty() {
        return this.numFriends == 0;
    }
```

```
    @Override
    public int size() {
        return this.numFriends;
    }

    @Override
    public void removeAll() {
        // makes all nodes unreachable
        // and thus garbage collected
        this.head = null;
        this.numFriends = 0;
    }

    // Pre : 0 <= i < numItems
    private Node<Friend> getNodeAt(int i) {
        Node<Friend> n = this.head;
        for (int k = 1; k <= i; k++)
            n = n.getNext();

        return n;

    }

    @Override
    public void add(int index, Friend item) throws ListException {
        if (index < 0 || index > this.size())
            throw new ListException("Index " + index + " is invalid for a
list of size " + this.size());

        Node<Friend> newFriend = new Node<Friend>(item);
        if (index == 0) {
            newFriend.setNext(this.head);
            this.head = newFriend;
        } else {
            Node<Friend> previous = this.getNodeAt(index - 1);
            Node<Friend> current = previous.getNext();

            // previous should not go to the new Node
            previous.setNext(newFriend);
            newFriend.setNext(current);
        }
        this.numFriends++;
    }
```

```
    @Override
    public Friend 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());

        Node<Friend> node = getNodeAt(index);
        return node.getItem();
    }

    @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());

        if (index == 0)
            this.head = this.head.getNext();
        else {
            Node<Friend> previous = getNodeAt(index - 1);
            Node<Friend> nodeToRemove = previous.getNext();
            Node<Friend> nodeAfter = nodeToRemove.getNext();

            previous.setNext(nodeAfter);
        }
        this.numFriends--;
    }

    @Override
    public String toString() {
        String s = "[";
        if (this.size() != 0) {
            Node<Friend> current = this.head;
            for (int i = 0; i < this.size() - 1; i++) {
                s = s + current.getItem().toString() + ",";
                current = current.getNext();
            }
            s = s + current.getItem().toString();
        }

        return s + "]";
    }

}
```

The Main Method:

```
public class TestFriendList {
    public static void main(String[] args) throws ListException {
        FriendsList friendList = new FriendsList();
        System.out.println("Testing add: ");
        friendList.add(0, new Friend("magicschoolbusdropout", false));
        friendList.add(1, new Friend("Lezduit", true));
        friendList.add(2, new Friend("HoosierDaddy", true));
        System.out.println(friendList.toString());

        System.out.println();
        System.out.println("Testing remove: ");
        friendList.remove(1);
        System.out.println(friendList);

        System.out.println();
        System.out.println("Testing remove all: ");
        friendList.removeAll();
        System.out.println(friendList);
    }
}
```

Announcements:

- Exam 01 is in one week.
    - True/False; Multiple Choice; Short Answer; Coding

Exam I:

- Points: 100
- Exam sections:
    - True/False (30 points)
    - Multiple Choice (40 points)
    - Short answer (15 points)
    - Coding (15points)
- Review:
    - Know basic Linux commands.
    - Understand how reference variables work.

- Know the difference between
  - ADTs,
  - abstraction (procedural abstraction and data abstraction), and
  - information hiding.
- Know how to write a class diagram.
- Know the pros/cons of recursion.
- Know what an interface is.
- Know what the keyword static is and what it does.
- Know the principles of OOP.
- Be able to trace through a recursive method.
- Be able to draw a representation of a linked-list.
- Compare/contrast the array-based List with the reference-based List.
- Write a recursive method
- Implement a method in a List
- Implement a method in a linked-list