**CS1401 – Programming Assignment 10**

**Due: Monday December 7, 2015 by 8:00 a.m.**

**To be submitted via Piazza in the folder “lab10”**

**Welcome to Lab 10!**

|  |
| --- |
| This is your last lab for the semester.  This week and next, you will keep practicing on user-defined types. You will be given a class and you will have to implement two types of containers for multiple items of this new type. You will also have to handle exceptions.  We hope you enjoyed the labs this semester!  Now, let’s get started and then move on to CS2401 adventures! |

** You should expect to work about 6 to 8 extra hours outside the lab session to complete this assignment.**

**It means that you need to make sure that Java works on your own computer, or that you go to open labs to work some more.**

**Extra time on labs includes completing the activities and taking the time to make sure that your submission is picture perfect!**

**Storage options: sedentary & fixed VS itinerant & flexible ☺**

**Some background.**  
Earlier this semester, you learned about **arrays** (arrays are also one of the topics of the challenge lab # 2). Arrays are convenient as they allow you to manipulate and store at once multiple items of the same type. More specifically, when using arrays, you should know exactly (or at the very least, at most) how many items you will need to store and manipulate in this array because the size of the storage is fixed. This can be seen as a drawback to this useful concept since often times, we do not know how many pieces of information we will have to handle. But let’s not worry about it for now.

So assuming that you know exactly how many pieces of a given type of information you need to handle, you can then easily “ask the computer” for memory to store it, as an array. Conveniently, since the size of the necessary memory is known (and assuming there is enough space on your computer), the computer will be able to assign the requested amount of memory as a contiguous chunk of memory. This is particularly important to know because this way, once you know where the array is stored, you have instant access to each piece of information within.

*As you see below, the idea of an array is very similar to that of a row of town houses. Once you know where the first one is, you have easy and almost instant access to all houses in the row.*



So arrays are really convenient, once you have acquired the necessary memory, manipulating the information / data within is extremely straightforward.

Now, what if we do not know how many pieces of information we will need. What can we do? Let’s say that you thought you would need to store 100 pieces of information and you only have 50. In order to save memory, you may want to copy these 50 pieces of information inside a smaller array of size 50 (hoping you will not need more memory later…). On the other hand, suppose you asked for storage for 100 pieces of information and you now need 200. You would need to request a larger storage (larger array), copy the 100 pieces of information already in the original array, and then keep filling it with the additional data you have to store. In both cases, the process of switching to a larger or smaller array is costly (imagine doing this on a much larger scale, with millions of items… that would be very expensive…).

So what can we really do when we do not know how much data we need to store? Another way of storing data is via what are called linked lists.

**Linked lists** allow you to store “lists” of items, just like arrays do, but on a request-basis: you get exactly the amount of memory that you need, as you go. The downside of it is that since you do not request the full amount of memory at once, the computer is not likely to provide contiguous storage. A consequence of that is that accessing any item in the list (once data has been stored) is not as easy as with arrays. Such lists are more similar to scavenger hunts than to town houses: to get to item # 3, you do need to go through items #1 and 2 first.

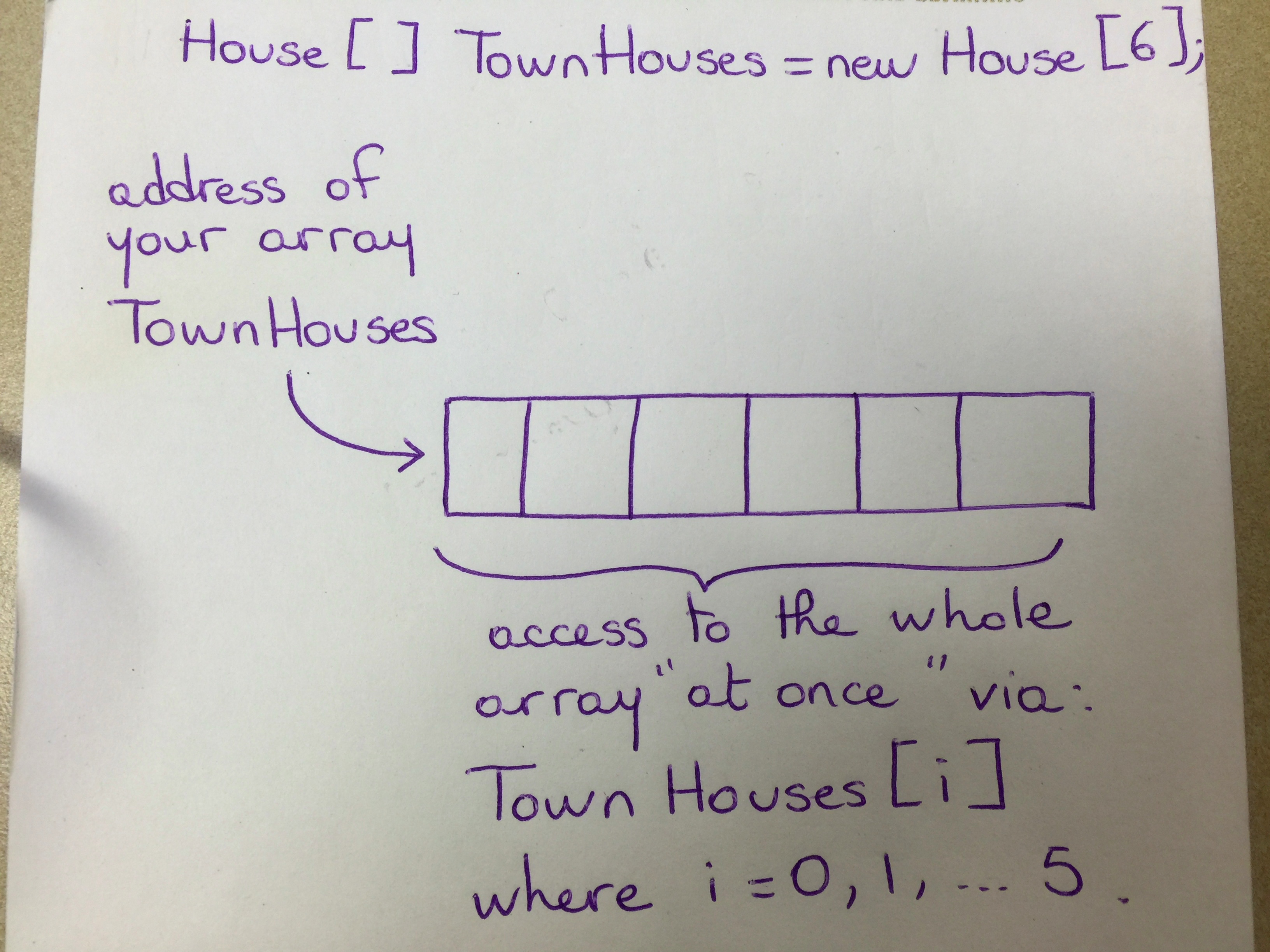


*On the above map for instance, in order to go from the boat to the moose, you need to go through each of the objects / locations in between. The idea is that each item contains a clue to access the next clue.*

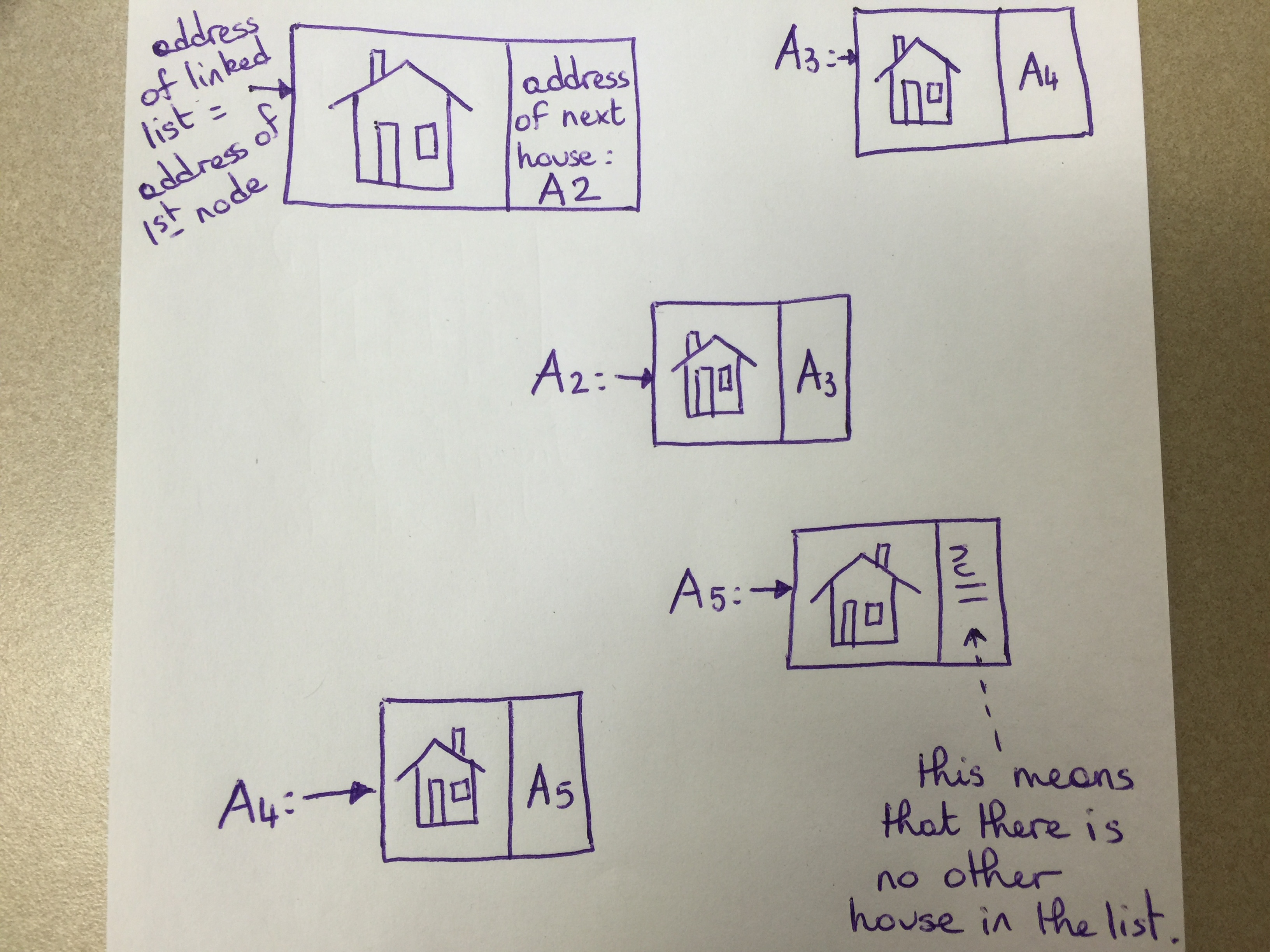
More specifically, each stored data is paired with an extra piece of information, which is the address of the next piece of information (which is null if the piece of information at hand is the last one in the list). Note: the address – extra piece of information – is not necessary with arrays because the next item in the list is known to be “right next to” the current one.

***Let’s look at the following first comparison between arrays and linked list:***

When you have an array, the address of your array, namely what you use as its name / identifier, gives you access to the whole array (see below):



When you have a linked list, the address of the linked-list, i.e., its identifier, only gives you access to the starting point of the list, i.e., its first element. Then, the first element, in turn, gives you access to the second, etc. (see below with an example).



In the above example, the list contains the first house followed by houses at address A2, A3, A4, and A5.

As you can see, a list has to be a set of nodes, each node encompassing the following information:

* The data you want to store (a house information for instance); and
* The address to the next node: the address to any node is actually what we use to access a node, which is of type node.

This type of data calls for a new user-defined type because two types of information need to be manipulated at once (the data of interest and the address to the next node).

Below is one way to build such a new user-defined type. We will call it HouseNode for now:

|  |
| --- |
| public class HouseNode {  // attributes  House myHouse;  HouseNode addressToNextHouse; // Note: the address to next is of type HouseNode  // followed by relevant constructors, accessors, mutators, and other methods  // …  } |

Please note that here, when dealing with linked lists, we do not use ArrayList. You should not use ArrayList in your lab.

**In this lab,** you will have to handle vacation destinations. Vacation destinations are defined by:

* The name of the city (String);
* The distance from home (int);
* The total amount of money spent during this vacation (double);
* The date at which you were on vacation at this destination (int defined by YYYYMMDD; e.g., November 10, 2015 would be coded as 20151110); and
* Whether you enjoyed this vacation or not (Boolean).

**Activity 1.**

This certainly calls for a new user-defined type: vacationDestination. You will implement this new type, with the above attributes, and corresponding methods (constructors, accessors (one per attribute), mutators (one per attribute), and a print method that prints the information in vacationDestination), in the file **vacationDestination.java** that is provided to you.

The print method for an object whose attributes are as follows:

cityName: Tucson

distance: 400

cost: 1050.99

date: 20150901

enjoyed: true

should print the following:

|  |
| --- |
| Destination: Tucson on 9/1/2015  Distance from home: 400 miles  Total expenses: 1050.99  This vacation was great! |

And if attribute enjoyed had been false, the print method should have printed:

|  |
| --- |
| Destination: Tucson on 9/1/2015  Distance from home: 400 miles  Total expenses: 1050.99  This vacation was not so great… |

**Activity 2.**

Then in another java file, **manageVacations.java**, you will implement the following methods:

1. A method, **ReadFromFile**, that takes a file name as a string – this file contains vacation information, and returns an array of items of type vacationDestination. This method should handle file reading errors via exception handling.
2. A method, **SortByDate**, that takes an array of items of type vacationDestination, and sorts them by date, from oldest to newest.
3. A method, **earliest**, that takes an array of items of type vacationDestination,

and returns the date your oldest vacation destinations.

1. A method, **Enjoyed**, that takes an array of items of type vacationDestination, and returns the number of vacation destinations that were enjoyed.
2. A recursive method, **recEnjoyed**, that takes an array of items of type vacationDestination and an integer n, and returns the number of vacation destinations that were enjoyed starting at index n in the array.
3. A method, **betterClose**, that takes:
   * an array of items of type vacationDestination; as well as
   * an integer named dist that represents a distance in miles,

and returns:

* + true if there are more (>=) vacation destinations in the array that are closer than (<=) dist from home that were enjoyed, than vacation destinations in the array that are farther than (>) dist from home and that were enjoyed; and
  + false if there are less (<) vacation destinations in the array that are closer than (<=) dist from home that were enjoyed, than vacation destinations in the array that are farther than (>) dist from home and that were enjoyed.

1. The **main** method that allows you to test all above methods.

**Activity 3.**

Activity 2 assumes that you know exactly how many vacation information you will have to handle. In particular, it assumes that what you read from the file is final: nothing should be added or removed from it.

In Activity 3, we will implement tools for the forgetful user ☺. We will assume that 1/ the user asks the computer to use the readFile method, 2/ sortDate, and 3/ he/she remembers that he/she forgot a destination. At that point, since we do not want to copy the content of the array of vacationDestination to another array (this might happen more than once and we do not want to risk the cost), we will need to become more flexible, i.e., use a linked list of vacationDestination.

In this activity, you will have to complete the implementation of another user-defined type: vacDestinationLL, in **vacDestinationLL.java**. This includes all necessary constructors, accessors (one per attribute), mutators (one per attribute), and the following methods:

1. A (non static) method, **printLL**, that prints a linked list;
2. A (non static) method, **sizeLL**, that returns the number of items in the list;
3. A (non static) recursive method, **sizeLLR**, that returns the number of items in the list;
4. Bonus: A (non static) method, **removeHead**, that modifies the original list by cropping out its first node;
5. A (non static) method, **addTail**, that takes a vacationDestination VD and modifies the original list where VD has been added as the last node in the list.
6. Bonus: A (non static) method, **addNth**, that takes a vacationDestination VD and an integer n, and modifies the original list where VD has been added as the nth node in the list.

**Activity 4.**

Finally, in **manageVacations.java**, you will implement the following two methods:

A method **buildLL** that takes an array of items of type vacationDestination, and returns a linked list of the vacationDestination items in the array.

A method **chronoLLinsert** that takes:

* 1. an array of vacationDestination vacations; and
  2. a vacationDestination extraVacation,

and returns a linked-list of all of the items in vacations in chronological order (from oldest to newest) as well as extraVacation inserted at the right chronological position to ensure that the resulting list is sorting in ascending chronological order.

Hint: you will need to use the method sortDate and the method addTail.

**Additional Activity on Exceptions.**

In many of the above methods, it makes sense to handle exceptions. We ask that, in the course of implementing all of these methods, you handle exceptions in at least 5 different places in your code to get full credit for handling exceptions. For any additional meaningful exception that you handle, and up to 5 additional exceptions, you will receive 10 extra points (therefore up to 50 extra points).

**What you have to turn in:**

* In a docx file, write the pseudocode of each of the above methods as well as a clear description of the new classes vacationDestination and vacDestinationLL: remember that a good pseudocode can be handed out to a programmer without further explanation and the programmer should be able to translate it to code. *This is the standard that you will be held to.*
* In this same docx file, you will describe and justify each of the exceptions you handle in your code.
* In the three java files that were provided to you, complete the description of the methods as described above and according to your pseudocode + include in one of the three files, the main method the lines of code that are necessary to meet the above specification.

**Important Note:** If your code is not properly indented, it will not be graded (you will receive 0 for the coding part of your grade).

**Important notes:**

* Indent your code properly following guidelines available at: <http://www.oracle.com/technetwork/java/javase/documentation/codeconventions-136091.html>. Badly-indented code will be graded 0.
* Spend time working on your pseudocode as the amount of points you get for the pseudocode is bigger than the amount of points you get for your code (usually, close to a 60/40 ratio).

**That’s all! Looking forward to seeing you in lab!**

**After that, you will be ready to successfully move on to CS2!**