CS 125 - Lecture 24

Objectives: Encapsulation; MP5 tips and tricks; constructors

Up next: MP5 out; Due in 5 days? Quiz in lab section this week!

Enhanced learning under different contexts and environments

1. **Encapsulation can be described as a protective barrier that prevents the code and data being randomly accessed by other code defined outside the class. Access to the data and code is tightly controlled by an interface.**

2. **Encapsulation** is the technique of making the fields in a class private and providing access to the fields via public methods. If a field is declared private, it cannot be accessed by anyone outside the class, thereby hiding the fields within the class. For this reason, encapsulation is also referred to as data hiding.

```
/* File name : Astronaut.java */
public class Astronaut{

   private String name;
   private String netID;
   private int age;

   public int getAge() {
      return age;
   }

   public String getName() {
      return name;
   }

   public String getNetID() {
      return netID;
   }

   public void setAge( int newAge) {
      age = newAge;
   }

   public void setName( String newName) {
      name = newName;
   }

   public void setNetID( String newId) {
      netID = newId;
   }
}
```

3. The public methods are the access points to this class' fields from the outside java world. Normally, these methods are referred as getters and setters. Therefore any class that wants to access the variables should access them through these getters and setters - collectively called *'accessor methods'.*

The variables of the Student class can be accessed as below:

```
/* File name : RunEncap.java */
public class RunEncap{

   public static void main(String args[]) {
      Astronaut candidate = new Astronaut();
      candidate.setName("Chris");
      candidate.setAge(42);
      candidate.setNetID("astrochris2");

      System.out.print("Name : " + candidate.getName() +
                       " Age : "+ candidate.getAge() +
                       " Email: "+candidate.getNetID() + "@illinois.edu");
    }
}
```

Output:____

4. **Benefits of encapsulation:**

a) The fields of a class can be made read-only or write-only.

b) A class can have total control over what is stored in its fields.

c) The users of a class do not know how the class stores its data. A class can change the data type of a field and users of the class do not need to change any of their code.

d) It's how all the cool kids do it.

```
public class SimpleList {
   private double[] data ;

   public int length() { ______________________________ }

// Add a value to end of the list
   public void add(double value) {











   }
   public double removeAt(int index) {












   }
   public double removeFromFront() {
   // add to the end and always remove from
   // the front: our list can be used as a queue!


   }
   public double removeFromEnd() {
   // our list can be used as a stack!


   }
```

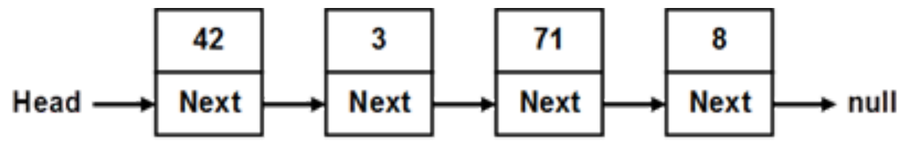
Data structures

List: contains a sequence of elements. The list has a property **length** (count of elements) and its elements are **arranged consecutively**. The list allows adding elements on different positions, removing them and incremental crawling.

Implementation: partially-filled arrays;

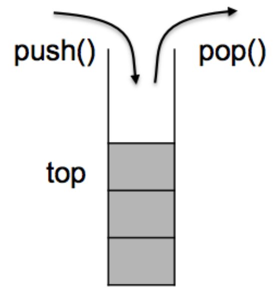
Linked List: collection of node objects; node elements keep information about their next element; nodes can be added or removed anywhere in list; traversal via next.

Implementation: objects with 'next' instance variables



Stack :

elements can only be added and removed on the top of the stack. *Last-in First-out* (LIFO) data collection - the last element that is inserted is the first to be removed. *pushOntoStack (); popOffOfStack() methods*

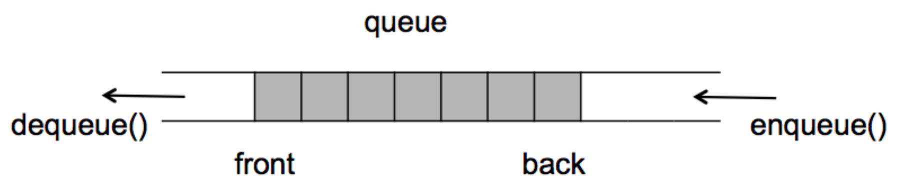


Implementation: arrays, linked lists, bill's closet

Queue :

add elements only on the back and retrieve elements only at the front; First-in First-out (FIFO) data collection - first element that is inserted is the first to be removed;

Implementation: arrays, linked lists, concert ticket orders, print jobs



5. **Overloading methods** … two or more methods with the same name, but different parameter lists.

Constructors … special method(s) used to create instances of a class, often using method overloading to initialize the object's instance variables in a variety of ways. You must manually include a *default* if overloading others.

```
public Time(int Hour, int Minute, int Seconds) { }
public Time(Time other) {  }
public Time(int secondsFrom1_1_1970) { }
public Time() { }
```

Implementing constructors:

7. Complete the US State class below so that we can create states in the following way :

```
State s1 = new State();

// 52% are democ. voters, 45% are repub. votes. 3% other-
State ptr = new State("Michigan",0.52, 0.45);

State copyOfMichigan = new State(ptr);

class State {
   private String name
   private double dem; // likelihood of democratic result 0..1
   private double repub; // likelihood of republican result 0..1
   private double other; //likelihood of independent results 0..1
   double getDem()   {                              }
   double getRepub() {                              }
   double getOther() {                              }
   String getName()  {                              }
   String toString() { return getName() + ": " + getDem()
                 + ", " + getRepub() + ", " + getOther(); }
```

6. Rewrite the following code to **use constructors** instead of the `set___` and `copy` methods.

```
Ghost g1 = new Ghost();
g1.setX(10); g1.setY(20);
g1.setEdible(true);

Ghost g2 = g1.copy();
```

… and write the two new Ghost constructors for the Ghost class: i) A constructor that takes 3 parameters and ii) a copy-constructor that takes a reference to another ghost.

```
class Ghost {
   private int x,y;
   private boolean edible;
```

```
2.public class SimpleList {
   private double[] data ;

   public int length() { ______________________________ }

// Add a value to end of the list
   public void add(double value) {









   }
   public double removeAt(int index) {











   }
   public double removeFromFront() {
   // add to the end and always remove from
   // the front: our list can be used as a queue!
```

1. What is encapsulation and why should we use it?

6. ***Make a list of US States.***

```
  StateList list = new StateList();
  State ptr = new State("Michigan",0.52, 0.45);
  list.add(ptr);
.
.
.

public class StateList {
     private State[] array = new State[0]; // empty array of pointers.
          // Note Each time add is called we'll make a larger array.

     public State getState(int i) { return array[i];}
     public int getSize() { return array.length; }

     public void add(State s) {
        State[] temp = new State[ this.array.length + 1];
       for (int i=0;i<state.length;i++) temp[i] =  array[i];   ???

       temp[ temp.length - 1 ] =s;                             ???
       this.array = temp; // array pointer now looks at new array
     }

     public void addAll(StateList other) { // Spot the error :-)  ???
       for(int x=0; x < other.length;x++)
           add(other.getState(x));
     }
     // returns states where state.repub > 0.5
     public StateList getRepublicanStates() {
          StateList result = new StateList();
          for(int x=0; x< array.length; x++) {
               State state = getState(x);
               if(state.getRepub() > 0.5)
                    result.add( state );
          }
          return result;
     }
     // ---- CONSTRUCTORS ----
     public StateList() { // do nothin'
     }
     public StateList( StateList other) {
          array = new State[ other.getSize() ];
          for(int x=0; x< array.length; x++) {
               array[x] = other.getState(x); // SHALLOW COPY

         or    array[x] = ____________________________// DEEP
           }
     }
}
```