```
public class MedicalImage {
   public Picture picture;
   public Date date;
   public Location where;
.
.
.
}
```

```
public class Link {
   public int value;
   public Date date;
   public Link next;
.
.
.
}
```

```
public class Simulation {
   public Atom[] atoms;
   public double temp;
.
.
.
}
```

```
public class Atom {
   public double x;
   public double y;
   public double vx;
   public double vy;
.
.
.
}
```

a) ***Create a new Atom and set its position to (5,8); create another and set its position to (7, 10):***

```
// Inside Universe.java
public static void main(String[] args) {

}
```

b) ***How would we compare two atoms? You CANNOT edit Atom.java.***

1) Create a class (static) method that takes two parameters 'a1, a2' of type Atom that returns true if two Atoms have exactly the same x and y.

Hint static => no this pointer!

How would you invoke this method?

c) ***What if you CAN edit Atom.java?*** *Hint 1: think of comparing two string objects, s1.equals(s2). Hint 2: you will need to use 'this.x'*

2) Create an instance (non-static) method 'equals' in Atom that takes a pointer to an atom and returns true if the atom has the same position as the given atom.

```
// Atom.java continued…
```

Write some code that uses your method:

Where would your code fail if the atom parameter value was null?

d) ***Create an instance method "moving" in Atom that takes no parameters and returns true iff the atom's vx or vy values are non-zero:***

Write code to print the result from invoking the method:

e) ***Create an instance method "init" in Simulation that initializes the atoms array with 100 elements and creates 100 atoms at random locations:***

```
// Simulation.java continued…
```

How would you invoke this method:

Write code that prints the contents of the array?

```
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!


}
```

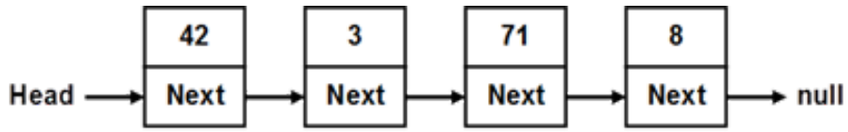
Data structures

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;

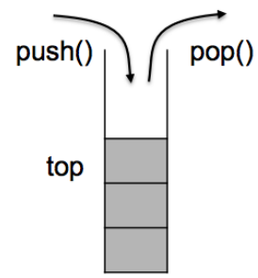
Linked List: collection of node objects; node elements keep information about their next element; nodes can be added or removed anywhere in list; traversal via next.

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*

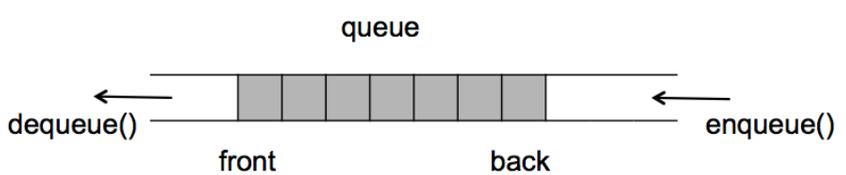


Implementation: arrays, linked lists, bill's closet

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



CS 125 - Lecture 23

Objectives: Encapsulation; MP5 tips and tricks

Up next: MP5 out; Due in 7 days (Next Friday, 8PM)

Enhanced learning under different contexts and environments

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 : EncapTest.java */
public class EncapTest{

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

   public int getAge() {
      return age;
   }

   public String getName() {
      return name;
   }

   public String getNetID() {
      return idNum;
   }

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

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

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

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.**

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 EncapTest class can be accessed as below:

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

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

      System.out.print("Name : " + encap.getName() +
                                 " Age : "+ encap.getAge() +
                                 " Email: "+encap.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.