6.170 Recitation 2

1. ADT (Abstract Data Types)

Roughly speaking, an **ADT** is an object that supports a set of operations, without worrying about the actual data representation.

For example, a "set of integers" is an **ADT** that perhaps supports the following operations:

- Adding an integer to the set.
- Removing an integer from the set.
- Testing whether an integer is in the set or not.
- Finding out how many integers are in the set.

In Java, an **ADT** usually corresponds to an "interface" or an "abstract class". For example, the "set of integers" might be represented by the following interface:

```
interface Set {
    public boolean add(int n);
    public boolean remove(int n);
    public boolean contains(int n);
    public int size();
}
```

Notice that, the description above does not describe how the set is actually represented. Indeed, there are many ways of implementing this **ADT**.

For example, one way is to represent the set using an array of integers:

```
class ArrayBasedSet implements Set {

    public int size = 0;
    public int maxsize = 100;
    public int[] myarray = new int[maxsize];

    public boolean add(int n) {...}
    public boolean remove(int n) {...}
    public boolean contains(int n) {...}
    public int size() {...}
}
```

Now, if another piece of code needs to use "sets", the programmer does not (and should not) need to care whether the set is implemented using one representation or another.

```
// This method adds all the elements of an array into an existing
set.

void addArrayToSet(Set a,int[] array) {

  for(int i=0; i<array.length; i++) {
      a.add(array[i]);
  }

}
```

---

2. Representation Independence and Representation Exposure

Suppose we now wish to write the ***"set union"*** operation. That is, given two sets A and B, we wish to write a method that computes the union of A and B. The ADT definition given earlier is insufficient for implementing this method, since it does not provide any method for retrieving or enumerating elements in the set.

Since the **myarray** field is public, we might be tempted to just write the ***"set union"*** operation by reading the array directly:

```
// This method adds all the elements of Set B into Set A.

void setUnion(ArrayBasedSet a,ArrayBasedSet b) {

  for(int i=0; i<b.myarray.length; i++) {
      a.add(b.myarray[i]);
  }

}
```

But this is very undesirable, because it ties our implementation of ***"set union"*** with a particular representation of **Set**. If we were given a different implementation of **Set**, for example, where the set is represented using a linked list, then our code will no longer work:

```
import java.util.LinkedList;

class ListBasedSet implements Set {

    public LinkedList<Integer> mylist = new LinkedList<Integer>();

    public boolean add(int n) {...}
```

```
    public boolean remove(int n) {...}
    public boolean contains(int n) {...}
    public int size() {...}
}
```

In general, we want our code to be ***"representation independent"***. That means we should not depend on any implementation-specific features. In this case, the author of the **ArrayBasedSet** implementation shouldn't have made **myarray** public. By making it public, it exposes its particular implementation and allows clients to be dependent on it. When the details of implementation are exposed, it is called ***"rep exposure"***, and is highly undesirable.

The proper way to implement ***"Set Union"*** is to first introduce a new accessor method for the **Set** interface. The accessor should be ***flexible*** enough so that client software can hopefully perform all necessary operations through the accessor; at the same time, the accessor should also be ***generic*** enough so that it can be implemented efficiently in both **ArrayBasedSet** and **ListBasedSet**. One possible choice is to add a method that returns an ***iterator*** for the set:

```
public Iterator<Integer> iterator();
```

The best way to add this method to the **Set** interface is to extend the builtin interface **Iterable**, like this:

```
interface Set extends Iterable<Integer> {
    public boolean add(int n);
    public boolean remove(int n);
    public boolean contains(int n);
    public int size();
}
```

Now, any class that implements **Set** will have to provide 5 methods. On top of ***add***, ***remove***, ***contains***, and ***size***, you have to implement an extra accessor method called ***iterator***. This extra method is required because it is mandated by the interface **Iterable**, and **Set** now implements **Iterable**.

This accessor is flexible enough for implementing ***"Set Union"*** because you can use it to iterate through every element in the set:

```
void setUnion(Set a,Set b) {
  for(Iterator<Integer> i=b.iterator(); i.hasNext(); ) {
      a.add(i.next());
  }
}
```

```
Or, you can make it even simpler, like this:

void setUnion(Set a,Set b) {
  for(Integer i:b) {
      a.add(i);
  }
}
```

At the same time, it is not difficult or overly inefficient to implement it for both **ArrayBasedSet** and **ListBasedSet**:

```
class ListBasedSet implements Set {
    private LinkedList<Integer> mylist = new LinkedList<Integer>();
    ...
    public Iterator<Integer> iterator() { return mylist.iterator(); }
}
```

```
class ArrayBasedSet implements Set {

  private int size = 0;
  private int maxsize = 100;
  private int[] myarray = new int[maxsize];

  ...

  public Iterator<Integer> iterator() { return new MyIterator(); }

  private class MyIterator implements Iterator<Integer> {

      private int[] readOnlyCopy;
      private int remains;

      public MyIterator() {
          remains = size;
          if (size > 0) {
              readOnlyCopy = new int[size];
              for(int i=0;i<size;i++) readOnlyCopy[i]=myarray[i];
          }
      }

      public boolean hasNext() { return remains > 0 ; }

      public Integer next() {
          if (remains==0) throw new NoSuchElementException();
          remains--;
          return readOnlyCopy[remains];
      }

      public void remove() {
          throw new UnsupportedOperationException("unsupported
operation!");
      }
  }
}
```

3. Java Interfaces and Java Abstract Classes

In Java, ADT's can be embodied using either ***interfaces***, or ***abstract classes***. There are benefits and limitations for these two approaches.

interface:

An interface can only contain a list of method declarations. That is, it only lists the methods that an implementation of this interface should have. An interface never has any data fields, and it never has actual Java code attached to it. For example, the **Set** interface we saw earlier does not contain any code in any of its methods:

```
interface Set extends Iterable<Integer> {
    public boolean add(int n);
    public boolean remove(int n);
    public boolean contains(int n);
    public int size();
}
```

If you want your new class **ListBasedSet** to ***implement*** the above interface, you have to provide the code for all 5 methods listed below:

```
import java.util.LinkedList;
import java.util.Iterator;

class ListBasedSet implements Set {

    private LinkedList<Integer> mylist = new LinkedList<Integer>();

    public boolean add(int n) {
        if (contains(n)) return false;
        mylist.add(n);
        return true;
    }

    public boolean remove(int n) {
        if (!contains(n)) return false;
        mylist.remove(n);
        return true;
    }

    public boolean contains(int n) { return mylist.contains(n); }
```

```
    public int size() { return mylist.size(); }

    public Iterator<Integer> iterator() { return
mylist.listIterator(); }
}
```

abstract class:

An abstract class, on the other hand, can contain data fields and some method implementations. For example, the class below is an abstract class, because the **sayGreeting**() method doesn't have any code in it:

```
abstract class Person {
    private String name = "";
    public String getName() { return name; }
    public void setName(String n) { name=n; }
    abstract public String sayGreeting();
}
```

If you want your new class **EnglishPerson** to ***extend*** the above abstract class, you only have to provide the code for the missing method **sayGreeting**():

```
class EnglishPerson extends Person {
    public String sayGreeting() { return "Hello"; }
}
```

benefits and drawbacks:

An ***interface*** can not provide a "default" implementation. So every implementation has to explicitly implement every method, even if some of the methods are likely to be common among all implementations. In this case, an ***abstract class*** can provide a default implementation for the common methods, and allow other methods to be implemented in many different ways.

For example: the code for **setUnion**() is identical in both **ArrayBasedSet** and **ListBasedSet**. But since **Set** is an interface, it can not provide any code. So we had to implement **setUnion** twice. But if we change **Set** into an ***abstract class***, then we can put **setUnion**() into it, and every subclass can share that implementation without having to duplicate your code:

```
interface Set extends
```

```
abstract class Set implements
```

```
Iterable<Integer> {
  public boolean add(int n);
  public boolean remove(int n);
  public void removeAll();
}

class ListBasedSet implements
Set {
  public Iterator<Integer>
iterator() {..}
  public boolean add(int n) {..}
  public boolean remove(int n)
{..}
  public void removeAll() {..}
}

class ArrayBasedSet implements
Set {
  public Iterator<Integer>
iterator() {..}
  public boolean add(int n) {..}
  public boolean remove(int n)
{..}
  public void removeAll() {..}
}
```

```
Iterable<Integer> {
  abstract public boolean
add(int n);
  abstract public boolean
remove(int n);
  public void removeAll() {
    // ACTUAL CODE;
  }
}

class ListBasedSet extends Set {
  public Iterator<Integer>
iterator() {..}
  public boolean add(int n) {..}
  public boolean remove(int n)
{..}
}

class ArrayBasedSet extends Set
{
  public Iterator<Integer>
iterator() {..}
  public boolean add(int n) {..}
  public boolean remove(int n)
{..}
}
```

On the other hand, you can only ***extend*** one class at a time. So if you are already a subclass of something else, or if you want to conform to several different specifications, then ***abstract class*** is not for you (because you can not extend both an ***abstract class*** and some other class). Java ***interfaces*** do not pose such restrictions. You can simultaneously ***implement*** several interfaces by implementing all the methods mentioned in each interface:

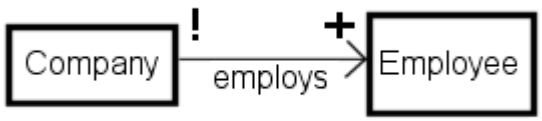
```
interface Drawable {
    public void draw();
}

interface Clickable {
    public void click();
}

interface Draggable {
    public void drag();
}

class Icon implements Drawable, Clickable, Draggable {
    public void draw() { System.out.println("drawing..."); }
    public void click() { System.out.println("clicking..."); }
```

```
    public void drag() { System.out.println("dragging..."); }
}
```

---

4. Object Model Diagrams

A useful way of looking at a software system is to think in terms of the objects in the system and relations among them. Here, we'll use a few examples to introduce some common notations:



Nodes in the diagram represents "sets of objects".
For example, **Company** and **Employee** are 2 sets of objects in this system.

Arrows represent "relations".
For example, **Employs** is a relation in this system.

The "!" and "+" on the two ends of the arrow are the **"multiplicity"** markers.
In particular:

- * means 0 or more (when the multiplicity is omitted, then this is the default)
- + means 1 or more
- ? means 0 or 1
- ! means exactly one

So this diagram says that every **Company** employs one or more **Employee(s)**,
and every **Employee** is employed by exactly one **Company**.

Exercise 4.1:
How would you change the relation to indicate that an employee can work for more than one company?

Answer: Change the '!' marker to the '+' marker.

Exercise 4.2:
Consider, for example, an MIT class scheduling application. The textual descriptions are given below.

Nodes would include:

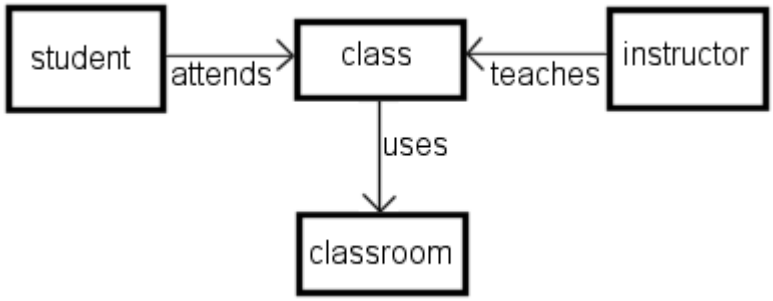
- Students.
- Classes.
- Instructors.
- Classrooms.

Relations would include:

- Attends: each student attends one or more classes.
- Teaches: each instructor teaches one or more classes.
  For ***simplicity***, we'll assume every class is taught by exactly one instructor.
- Uses: each class uses exactly 1 classroom. But a classroom can be used for different classes, since they don't meet at the same time.

Ignoring the multiplicity markers for the time being, what might an object model diagram for the system looks like?

Answer:



Exercise 4.3: Your task is to fill in the missing multiplicity markers in the diagram above.

Answer: