A few more details about `Object` class methods from last lecture

One point I didn't mention last lecture is that there is a relationship between the `equals(Object)` and `hashCode()` methods. The Java API for Object.hashCode() recommends that if `o1.equals(o2)` is true, then `o1.hashCode() == o2.hashCode()` also should be true.[1] Why? One reason is that the `hashCode()` method is used in the `HashMap<K,V>` class and other classes such as `HashSet<E>` as well. In particular, for the `HashMap<K,V>` class, suppose we have two variables `key1` and `key2` of type `K`. Suppose we put an entry into the hashmap using `put(key1, value)` and then later we have the instruction `get(key2)`. If `key1.equals(key2)` is true, then we would expect `get(key1)` to return the same value as `get(key2)`. Otherwise, what would be the point of saying that two keys are equal!? But we can only ensure that they get the same value if indeed they have the same hash codes.

Java `Object clone()` method

Another commonly used method in class `Object` is `clone()`. Recall this method from Assignment 1, when you cloned large `NaturalNumber` objects, for example, in the subtraction method. In general, the `clone()` method creates a different object, which is of the same class as the invoking object and which has fields that have identical values to those of the invoking object at the time of the invocation.

Cloned objects are supposed to obey the following:

- The expression `x == x.clone()` should return false.
- The expression `x.equals( x.clone())` should returns true (suggested, but not required).

These two conditions make intuitive sense. The point of cloning is to create a different object instance (first condition), but the clone is suppose to be the same as the original in whatever sense we define "same" to mean for that object's class. Note that the second condition doesn't hold for `Object` objects. But that's ok, as one rarely clones `Object` objects.

One subtle aspect of cloning is that objects can reference other objects. So if we clone an object which has reference fields, then should we also clone the objects that are referenced? For example, if we clone a `LinkedList` object, then do we want the list and the objects in the list to be cloned? This is called a *deep copy*. Or do we just want to the list to be cloned but the objects that the list references should not be cloned? This is called a *shallow copy* . If you look up the Java API for `LinkedList`, you'll see that it specifies to make a shallow copy. But in general, there is no correct answer to what one should do. It is an issue of design, and what is appropriate for the situation in question.

ADTs versus APIs

[ASIDE: This discussion reiterates point made at the end of lecture 9.]

We have seen many abstract data types (ADTs): lists, stacks, queues, trees, binary search trees, priority queues, graphs, maps. Each ADT consists of a set of operations that one performs on the data. These operations are defined independently of the implementation in some programming

[1] The converse need not hold. E.g. Two different Java strings might have the same hashCode().

language. It is sometimes useful to keep the implementation details hidden, since they may just be distracting and irrelevant.

Although ADT's are meant to be independent of any particular programming language, in fact they are similar to concrete quantities in programming, namely interfaces that are given to a programmer. In Java, for example, there is the *Java API*, which you are familiar with. The API (*application program interface*) gives a user many predefined classes with implemented methods. What makes this an "interface" is that the implementation is hidden. You are only given the names of classes and methods (and possibly fields) and comments on what these methods do.

The word "interface" within "Java API" should not be confused with related but different usage of the word, namely the Java reserved word `interface`, which is what this lecture is about.

Java `interface`

A typical first step in designing an object oriented program is to define the classes and the method signatures within each class and to specify (as comments) what operations the methods should perform. Eventually, you implement the methods.

A user (client) of the class should not need to see the implementation of a method to be able to use it, however. The user only sees the API. If the design is good, then all the client needs is a description of what the method does, and the method signatures: the return type, method name, and parameters with their types. (This hiding of the implementation is called *encapsulation*.)

In Java, if we write *only* the signatures of a set of methods in some class, then technically we don't have a class. What we have instead is an `interface`. So, an `interface` is a Java program component that declares the method signatures but does not provide the method bodies.

We say that a class `implements` an interface if the class implements each method that is defined in the interface. In particular, the method signatures must be the same as in the interface. So, if we say a class `C` implements an interface `I`, then `C` must implement all the methods from interface `I`, which means that `C` specifies the body of these methods. In addition, the class `C` can have other methods.

e.g. Java `List` interface

Consider the two classes `ArrayList<T>` and `LinkedList<T>` which are used to implement lists. As such, these two classes share many method signatures. Of course, the underlying implementations of the methods are very different, but the result of the methods are the same, in the sense of maintaining a list. For example, if you have a list and then you remove the 3rd item, you get a well defined result. This new list should not depend on whether the original list was implemented with a linked list or with an array.

The `List<T>` interface includes familiar method signatures such as:

```
void       add(T o)
void       add(int index, T element)
boolean  isEmpty()
T          get(int index)
T          remove(int index)
int        size()
```

Both `ArrayList<T>` and `LinkedList<T>` implement this interface, namely they implement all the methods in this interface.

The `ArrayList` class also has other methods that are not part of the `List` interface. For example, the `ensureCapacity(int)` method will expand the underlying array to the number of slots of the input argument if the current array has fewer than that many slots. The `trimToSize( )` method does the opposite. It will shrink the length of the underlying array so that the number of slots is equal to the current number of elements in the list. Note that these two methods make no sense for an `LinkedList`.

There are also methods for the `LinkedList` class that would not be suitable for `ArrayList` class, namely `addFirst` and `removeFirst`. There is nothing special about these operations for array lists. They would be implemented exactly the same as on any other index and they would be expensive because of the shifts necessary. If these operations are commonly needed, then one would tend to use a linked list instead since these operations are inexpensive for linked lists.

Why is the `List` interface useful? Sometimes you may wish to write a program that uses either an `ArrayList<T>` or a `LinkedList<T>` but you may not care which. You want to be flexible, so that the code will work regardless of which type is used. In this case, you can use the generic Java interface `List<T>`. For example,

```
void myMethod( List<String> list ){
       :
    list.add("hello");
       :
}
```

Java allows you to do this. The compiler will see the `List` type in the parameter, and it will infer that the argument passed to this method will be an object of some class that implements the `List` interface. As long as `list` only invokes methods from the `List` interface, the compiler will not complain. You can also do thinks such as the following, namely have the same variable `list` reference different types of lists at different times in the program.

```
List<String>        list;

list  =  new  ArrayList<String>();
list.add("hello");
:
list  =  new  LinkedList<String>();
list.add( new String(hi) );
```

See the lecture slides for a similar example. I defined a `Shape` interface which has two methods: `getArea()` and `getPerimeter`, where the latter is the length of the boundary of the shape. I then defined two classes `Rectangle` and `Circle` that implement the `Shape` interface, namely they provide method bodies.

The following example is used to illustrate one of the limitations of interfaces, and motivates the use of abstract classes discussed next.

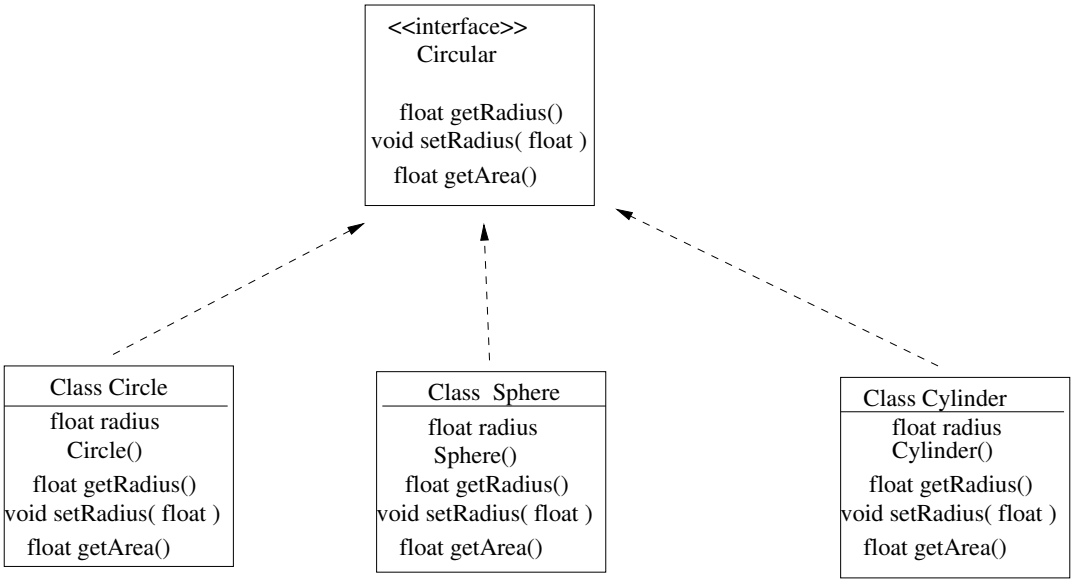
Example: Circular

Many geometrical shapes have a `radius`, for example, of a circle, sphere, and cylinder. Suppose we wanted to define classes `Circle`, `Sphere`, `Cylinder` of shapes that have a radius. In each case, we might have a private field `radius` and public methods `getRadius()` and `setRadius()`. We might also want a `getArea()` method.

We could define an interface `Circular`

```
public interface Circular{
   public double getRadius();
   public void   setRadius(double radius);
   public double getArea();
}
```

and define each of these classes to implement this interface. The problem with such a design is that we would need to define each class to have a local variable `radius` and (identical) methods `getRadius()` and `setRadius()`. Only the `getArea()` methods would differ between classes. We could do this, but there is a better way to deal with these class relationships.



Abstract classes

The better way is to use a hybrid of a class and an interface in which some methods are implemented but other methods are specified only by their signature. This hybrid is called an `abstract class`. One adds the modifier `abstract` to the definition of the class and to each method that is missing its body. For example:

```
public abstract class Circular{

    private double radius;
    Circular(){};

    Circular(double radius){
        this.radius = radius;
    };

    public double getRadius(){
       return radius;
    }

    public void   setRadius(double radius){
        this.radius = radius;
    }

    public abstract double getArea();
}
```

This abstract class has just one abstract method that would need to be implemented by the subclass `Circle`, `Cylinder`, or `Sphere`.

Note that the subclass `Circle` might also have a method `getPerimeter()`. Such a method would make no sense for a `Sphere` or `Cylinder` since perimeter is defined for 2D shapes, not 3D shapes. Similarly, `getVolume()` would make sense for a `Sphere` and `Cylinder`, but not for a `Circle`.

An abstract class *cannot* be instantiated. However, abstract classes do have constructors. This seems like a contradiction, but it is not. Abstract classes are extended by concrete subclasses which provide the missing method bodies. When these subclasses are instantiated, they must inherit the fields and methods of the superclass. In particular, the values of the inherited subclass fields are set by the superclass constructor (either via an explicit `super()` call, or by default). Thus, even if the superclass is abstract, it still needs a constructor.

```
public class Circle extends Circular{

   Circle(double radius){
      super(radius);
   }

   double getArea(){
     double  r = this.getRadius();
     return  Math.PI * r*r;
   }
}
```

```
public class Cylinder extends Circular{
   double height;

   Cylinder(double radius, double h){
       super(radius);
       this.height = h;
   }

   double getArea(){
      return 2* Math.PI * r * height;
   }
}
```

Abstract classes also appear in class hierarchies/diagrams, along with interfaces as above:

- a class (abstract or not) "implements" an interface
- a class (abstract or not) "extends" a class (abstract or not)

A class can implement more than one interface. However, a class cannot extend two abstract classes. The reason for this policy is the same for why a class cannot extend two classes – namely if the two superclasses were to contain two different versions of a method with the same signature then it wouldn't be clear which of these two methods gets inherited by the subclass.

Finally, one can declare variables to have a type that is an abstract class, just as one can declare a variable to be of type class or of type interface.