ADT (Abstract Data Type)

In the last several lectures I have presented algorithms but expressed them using *pseudo-code*. I did this because I was emphasizing the sequence of operations that are performed to solve a given problem, and I was not so concerned with a particular implementation.

Writing algorithms using pseudocode is very common when you are first planning out how to solve a problem. Typically, when you have in mind a certain type[1] of data and a set of operations that you want to perform on the data, including what are the operations performed on the data, what inputs these operations require and what outputs they produce. Notice that these are issues of *what*, not *how*. One often uses the term "abstract data type" (ADT) to refer to the type of data and the operations to be performed on the data. When you are talking about an ADT, you generally mean that you are not (yet) concerned with a specific implementation.

For example, we have seen several examples of "lists", namely a set of objects of a given type and that are arranged in an order. We have seen that a list can be implemented in Java either using an "array" or using a "linked list". These two implementations are quite different but for both of them we can talk about operations such as $add(e), add(i, e), remove(i), clear(), replace(i, e), getEntry(i), contains(e), getLength(), isEmpty()$, etc where i is an index and e is a data element in the list. We can specify these operations without even saying what programming language we are planning to use.

Java Interfaces

Once we decide to use Java as our programming language, we can translate the ADT into Java code. A common first step is to define the method signatures that correspond to the operations we want to perform. We might only provide the method signatures, and not the method bodies. The idea is that we can write the bodies (the implementation) afterwards or we can hire someone to write the implementation.

Another reason to only consider the method signature is that that users of our classes should not need to know how the classes are implemented. (This hiding of the implementation is typically called *encapsulation*.) By only writing the signatures along with some comments that specify how the each method is to be used, we force ourselves to see the methods from the user's point of view.

If we write only the signatures of a set of methods in some class, then technically we don't yet have a class. What we have an `interface`. An interface is a Java program component that declares the method signatures but does not provide the method bodies.[2]

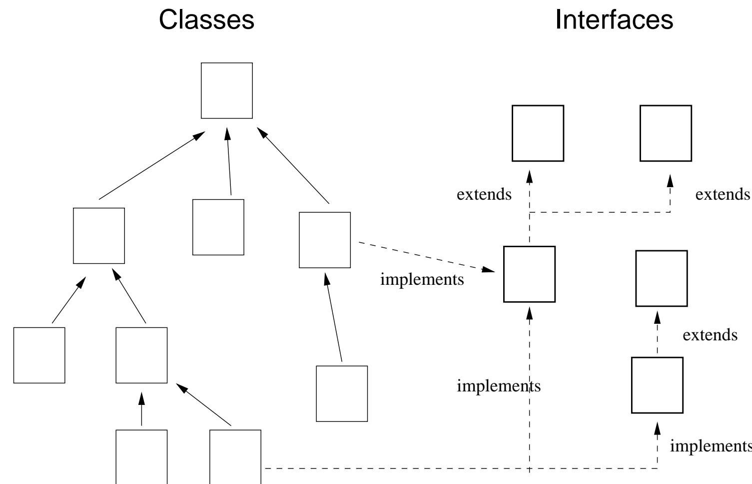
Interfaces and Inheritance

We say that a class `implements` an interface if it provides the method body for each method in the interface. If a class `C` implements an interface `I`, then `C` must implement all the methods from interface `I`, meaning that `C` must specify the body of these methods. In addition, `C` can have other methods, such as a `C` constructor, as well as fields both public and private.

[1] the term "type" here is used in the general English sense, not in the sense of a particular programming language

[2] There is a bit more to the definition of an interface than this. For example, the methods have to be `public`. Also, it is ok to include possibly public named constants in a method signature.

Earlier when we discussed classes and their inheritance relationships, we pictured a hierarchy where each class (except `Object`) extends some other unique class (see sketch below on the left). How do interfaces fit into such hierarchies?



An interface just becomes a new node in the hierarchy, as shown above right. We used dashed arrows to indicate inheritance relationships that involve interfaces.

- If a class `C` implements an interface `I` then we put a dashed arrow from `C` to `I` in the hierarchy.
- One interface (say `I2`) can extend another interface (say `I1`). This means that `I2` inherits all the method signatures from `I1`. We don't need to write the method signatures out again in the definition of `I2`. In the class hierarchy, we put a dashed line from `I2` to `I1`.
- Wherease each class can (directly) extend exactly one other class,[3] a class `C` can implement multiple interfaces. The parent interfaces can even contain the same method signature. This is no problem since the interfaces only contain the signatures (not the bodies), so there can be no conflict. We would say: `public class C2 extends C1 implements I1, I2, I3`

Let's look at a particular example. This example serves more to illustrate the definition, than to do something useful.

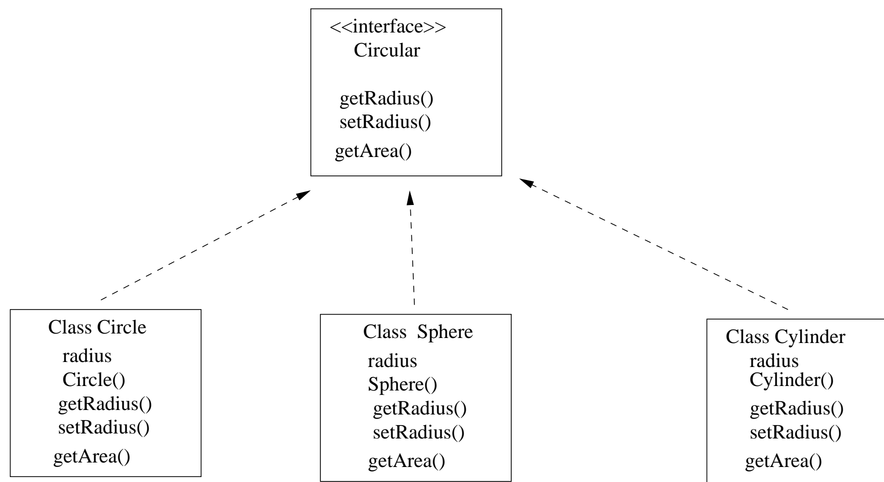
Example: Circular

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

We can also talk about an area for each shape, though this has different meanings for the three types of shape. So we want to define a method `getArea()`.

[3] If this 'unique parent' constraint were not in place, and a class `C` were allowed to extend multiple classes (say **A** and `B`), then it could happen that there is method conflict – class **A** and `B` could contain a method with the same signature but with different bodies. In that case, if an object that belongs to class `C`, then it would be ambiguous which of these methods from `A` or `B` would be inherited by `C`.

How might we capture this? Suppose we defined three classes: `Circle`, `Sphere`, `Cylinder`. For each of them, we would have a local variable *radius* and methods `getRadius()` and `setRadius()`.



We could, for example, define an interface:

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

and then define classes that implement this interface,

```
public class Circle implements Circular{
    private radius;
    public Circle();
    public double getRadius(){
        return radius;
    }
    public void   setRadius(double radius){
        this.radius = radius;
    }
    public void   getArea(){
        return Math.PI * radius * radius;
}
```

```
public class Sphere implements Circular{
    private radius;
    public Sphere();
    public double getRadius(){
        return radius;
    }
    public void   setRadius(double radius){
        this.radius = radius;
    }
    public void   getArea(){
        return 4/3*Math.PI * radius * radius * radius;
    }
}

public class Cylinder implements Circular {
    private radius;
    private length;
    public Cylinder();
    public double getRadius(){
        return radius;
    }
    public void   setRadius(double radius){
        this.radius = radius;
    }
    public void   getArea(){
        return     length * Math.PI * radius * radius;
    }
}
```

Abstract classes

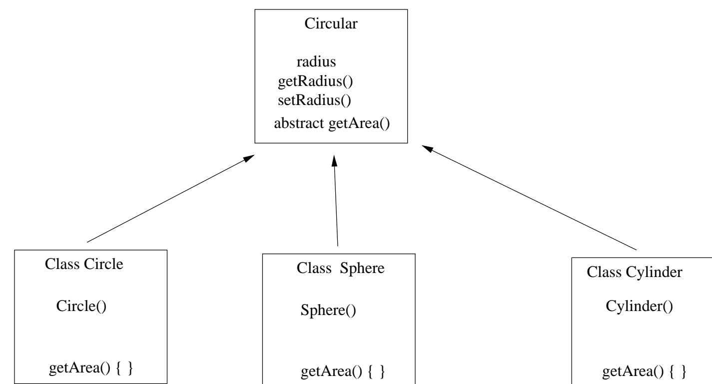
It often occurs that one wants to define a class with some methods fully specified, but some methods specified only by their signature. Think of this as a hybrid between a full class and an interface. In Java, this hybrid is called an `abstract class`. One adds the modifier `abstract` to the definition of the class and to each method that is missing its body (see example below).

An abstract classes cannot be instantiated. However, abstract classes do still have and need constructors. To understand why, note that abstract classes are extended by concrete (non-abstract) 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 fields are set by the superclass constructor (either via an explicit `super()` call, or by default). Whether the superclass is abstract or not, it needs a constructor.

Finally, note that abstract classes also appear in class hierarchies: a class (abstract or not) "implements" an interface; an class (abstract or not) "extends" a class (abstract or not).

We will see examples in the coming lectures. For now, let's redo the example of Circular by using an abstract class instead of an interface.

Example: Circular




```
public abstract class Circular{
    private double radius;
    public double getRadius(){
        return radius;
    }
    public void   setRadius(double radius){
        this.radius = radius;
    }
    public abstract double getArea();
}

public class Circle extends Circular{
    public double getArea(){
        double    radius =  getRadius();   //  getRadius() is inherited
        return    Math.PI * radius * radius;
    }
}

public class Sphere extends Circular{
    public double getArea(){
        double    radius =  getRadius();
        return    4/3*Math.PI * radius * radius * radius;
    }
}

public class Cylinder extends Circular{
    public double getArea(){
        double    radius =  getRadius();   //  getRadius() is inherited
        return    length * Math.PI * radius * radius;
    }
}
```