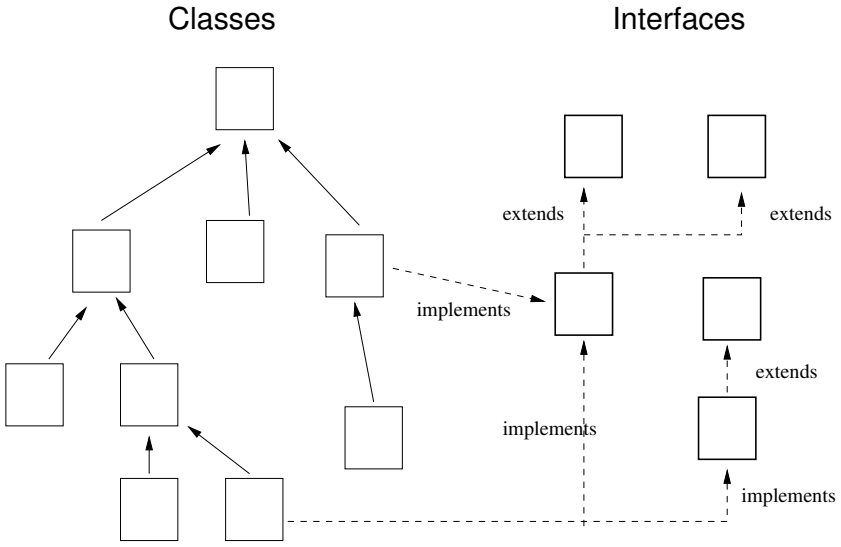
Interfaces (revisited)

Earlier when we discussed Java classes and their inheritance relationships, we considered a hierarchy where each class (except `Object`) extends some other unique class. See below left. Thus Java classes define a tree, with each note having a reference to its parent i.e. superclass. (Java classes do not have references to their subclasses. e.g. You are allowed to extend the `LinkedList` class, and in doing so, you don't change the `LinkedList` class.)



How do Java interfaces fit into the class hierarchy? As shown above right, an interface is another "node" in the inheritance diagram. 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`. Recall that a "class implements an interface" means that the class provides the method body for each method signature defined in the interface.
- 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 diagram, we would put a dashed line from `I2` to `I1`.
- Each class (other than `Object`) directly extends exactly one other class. Why? 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 might be a method conflict – superclasses **A** and `B` could contain a method with the same signature but with different bodies. Which of these methods would an object of class `C` inherit?

  However, 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:
  `class C2 extends C1 implements I1, I2, I3`

Abstract classes

We have seen examples of interfaces in lecture 29. The following example is used to illustrate one of the limitations of interfaces, and motivates the use of abstract classes discussed next.

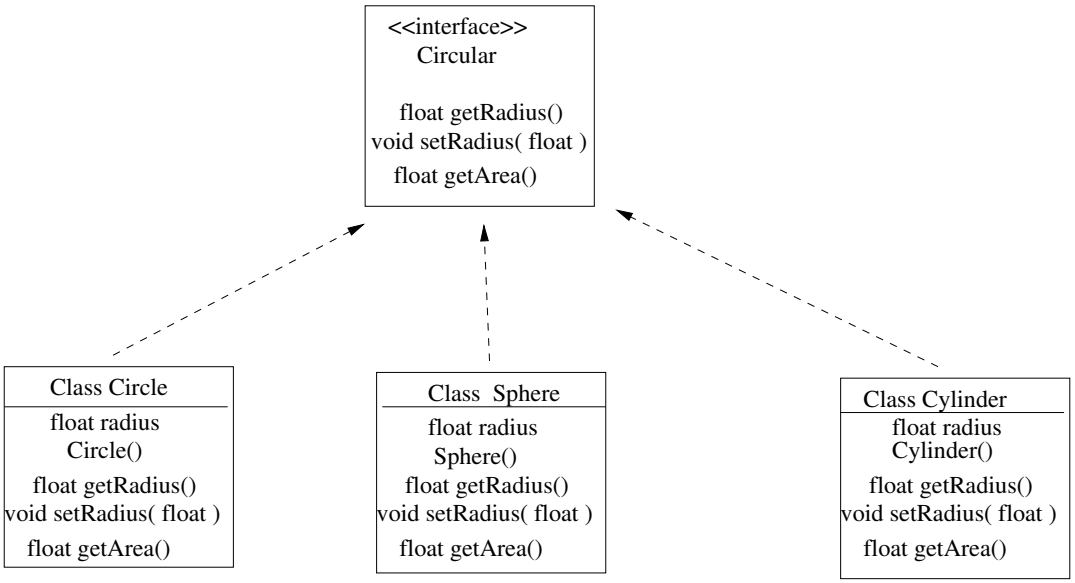
Example: Circular

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.



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;
    }
}
```

Here are a few more details before moving on. 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)

Although a class can implement more than one interface, 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.

Type Conversion

You should already be somewhat familiar with the basics of primitive types and how conversions can occur between them. Primitive types are ordered from "narrow" to "wide".

`byte`, `char`, `short`, `int`, `long`, `float`, `double`.

In fact, it is a bit more complicated than that, and if you care see
`http://docs.oracle.com/javase/specs/jls/se7/html/jls-5.html`

Widening conversions occur automatically, but narrowing conversions requires an explicit *cast*, otherwise you will get a compiler error. Here are some examples. *These type conversions require a change in the bit representations of the values.* For example, the above listed primitive types can have anywhere from 1 to 8 bytes. You will learn the details of these representations in COMP 273. (See my 273 lecture notes if you are interested.)

```
int    i = 3;
double d = 4.2;
       d = i;                  // widening (in assignment)
       d = 5.3 * i;            // widening  (by "promotion")
```

```
        i = (int) d;            // narrowing (by casting)
        float f = (float) d;    //  "

char    c = 'g';
int     index = c;            // widening
        c = (char) index;     // narrowing
```

Narrowing often leads to an approximation. For example, when you convert from a `float` to an `int`, you discard the fractional part. [ASIDE: Interestingly, though, approximations can occur with widening conversions as well. Four example, four bytes (32 bits) are used to represent an `int` and four bytes (32 bits) also used to represent a `float`. This means that you can represent 2^{32} different possible values of each. But most `float`'s have a fractional part, since that's what floats are for. So obviously the two representations cover quite a different set of values!]

We use similar concepts of "narrowing" and "widening" in a class hierarchy as well. If class `Beagle` extends class `Dog`, then class `Beagle` is narrower than `Dog`, or equivalently, `Dog` is wider than `Beagle`. In general, a subclass is narrower than its superclass; the superclass is wider than the subclass.

Notice that an object of a subclass typically has more fields and methods than an object of its superclass. So if you think of the relative "size" of the object (the number of fields and methods) then you will notice that the narrower object is bigger. This is the opposite of what generally happens with primitive types, where the wider type usually uses the same or more bytes than the narrower type.

There is another important difference to keep in mind between primitive and reference types. As I mentioned above, when we convert from one primitive type to another, in fact we perform an operation in which one binary representation of a value is replaced by another binary representation, possibly with a different number of bits. For example, a double uses 64 bits whereas a float uses only 32, and so converting from a float to a double (or double to float) requires re-coding the bits. *Reference type conversion is different, however, since no change occurs to the referenced object.* Rather, the conversion only tells the compiler that you (the programmer) expect or allow the object to be a certain type at runtime. A few examples below will illustrate this idea.

First, though, here is a bit more terminology. We cast *downwards* ("downcasting") when we are casting from a superclass to a subclass, and we cast *upwards* (upcasting) when we cast from a subclass to a superclass. Upcasting occurs automatically, and so it is sometimes called *implicit casting*. We have seen upcasting before, e.g.

```
Dog  myDog = new Beagle();
```

This is analogous to:

```
double  myDouble = 3;   //    from int to double.
```

We have not seen downcasting before for reference types. We will see it below.

Example 1

```
Dog  myDog = new Beagle();  // Upcasting.
```

```
  :
Poodle  myPoodle = myDog;     // Compiler error.
                              // (implicit downcast Dog to Poodle not   allowed).

 myDog.show();               // Gives a compiler error, since show()
                             //  is not defined in Dog class.


Poodle  myPoodle = (Poodle) myDog;   // Allowed

  myPoodle.show()                 //  runtime error if myPoodle referenced
                                  //  a Dog object that has no show() method
                                  //  (more on this next lecture)

((Poodle) myDog).show();  //  Explicit down cast ok: no compiler error.
                          //  But if myDog references a Doberman at
                          //  runtime, then you get a runtime error
                          //  since Dobermans aren't show dogs.

//  Alternatively, you could do the following which would not
//  produce a runtime error (if myDog references a Doberman) and
//  instead just wouldn't get executed.

if (myDog instanceOf Beagle){
   ((Beagle) myDog).hunt();
}
```

The `instanceof` operator takes two arguments: the first is a reference type variable `var`; the second is a class `C`, ie.

```
var instanceOf C
```

The operator returns true if and only if the object referenced by `var` is an instance of the class `C` or any class that extends `C`. The idea behind this rule is simple: we use the `instanceof` operator to verify whether the object referenced by `var` has access to the methods of `C`. In the above example, we are checking whether the *object* referenced by `myDog` can invoke `hunt()`.

Next lecture, I will discuss the above issues in more detail.