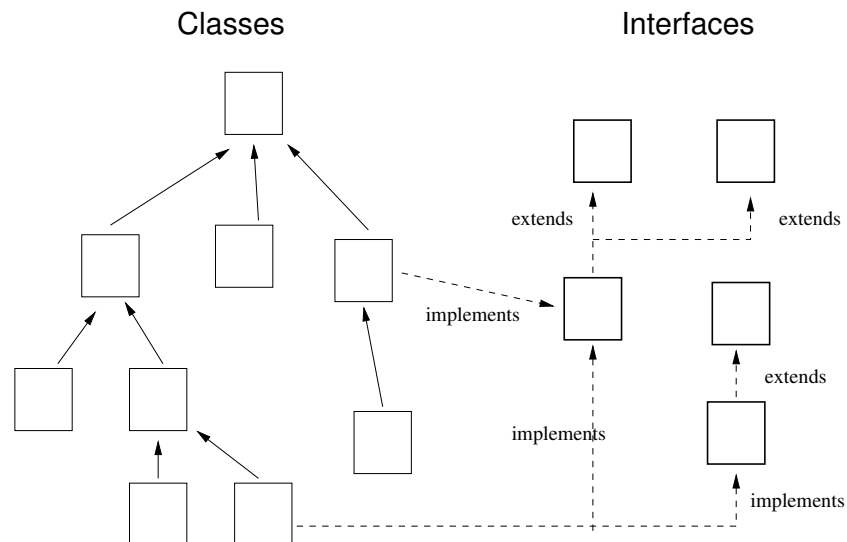


## Interfaces (revisited)

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 node having a reference to its parent<sup>1</sup> i.e. superclass. How, if at all, 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`.
- Although each class (other than `Object`) directly extends exactly one other class,<sup>2</sup> 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
```

We have seen examples of interfaces in lecture 22. The following example is used to illustrate one of the limitations of interfaces, and motivates the use of abstract classes discussed next.

<sup>1</sup>But nodes do not have references to their children, i.e. subclasses.

<sup>2</sup> 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 – 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?

## Example: Circular

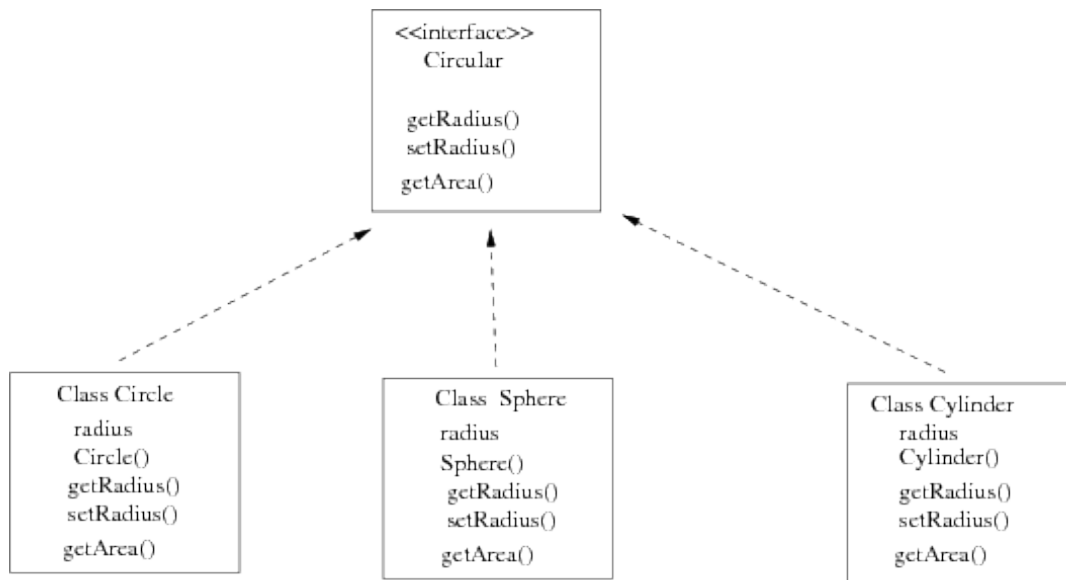
Many geometrical shapes have a **radius**, for example, of a circle, sphere, and cylinder. Suppose we wanted to define classes `Circle`, `Sphere`, `Cylinder` of shapes that have a radius. In each case, we might have a private field `radius` and public methods `getRadius()` and `setRadius()`. We might 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. This would allow us to use polymorphism. For example, we could define a variable to be of type `Circular` and it could use any of the methods of that interface.

The problem with such a design, though, 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 its not so elegant.



## Abstract classes

What we would be better for the last example is a hybrid between a class and an interface. We would like some methods to be implemented but other methods to be specified only by their signature. 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. 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`. 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 fields are set by the superclass constructor (either via an explicit `super()` call, or by default). Thus, whether the superclass is abstract or not, it needs a constructor.

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

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

One can declare variables to have a type that is an abstract class, just as one can declare variables to be of type (concrete) class or of type interface. Polymorphism is again used to decide which method is invoked at runtime.

## Casting

From COMP 202 (or equivalent), you should be familiar with the basics of primitive types and how conversions can occur between them. For example, you learned that primitive types are ordered

from “narrow” to “wide”, for example,

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

A widening conversion occurs automatically, but a narrowing conversion requires an explicit *cast*, otherwise you will get a compiler error.

```
int    i = 3;
double d = 4.2;
d = i;          // widening conversion (as part of an assignment)
d = 5.3 * i;    // widening conversion (by "promotion")
i = (int) d;    // narrowing conversion (by casting)
float f = (float) d; // "
```

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.

One possible confusion to watch out for is the following. An object of a subclass typically has more fields and methods than an object of its superclass, and 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 happens with primitive types, where the wider type typically uses more bytes.

Another difference to keep in mind between primitive and reference types is that, 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. Casting reference types is different, however, since no change occurs to the referenced object. Rather, the cast serves to tell the compiler that you expect the object to be a certain type at runtime, and specifically that you want to apply a certain method that is defined objects of that runtime class. 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 (at least not for reference types).

**Example 1**

```
Dog myDog = new Doberman(); // Upcasting.
:
Beagle myBeagle = myDog;    // Compiler error.
                             // (implicit downcast Dog to Beagle not allowed).

Beagle myBeagle = (Beagle) myDog; // Allowed (though runtime error
                                   // would occur if myDog references
                                   // a Doberman object).
```

I emphasize: when we say that a “cast” occurs here, don’t think of the object as changing. It doesn’t. Rather, the cast tells the compiler that the programmer is expecting `myDog` to reference a `Beagle`. (If this fails to be the case, a class exception error will occur at runtime.)

**Example 2**

```
Dog myDog = new Beagle();
myDog.hunt(); // Gives a compiler error, if hunt() is
              // defined in Beagle but not in class Dog.

((Beagle) myDog).hunt(); // Explicit cast ok and no compiler error.
                        // But if myDog references a Doberman at
                        // runtime, then you get a runtime error.

// 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();
}
```