## Type Conversion

You should already be 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`

and the number of bytes used by each is

$$1, 2, 2, 4, 8, 4, 8$$

respectively.

Widening conversions occur automatically. Narrowing conversions require an explicit *cast* in the code; otherwise you will get a compiler error. Here are some examples.

```
int    i = 3;
double d = 4.2;

d = i;                  // widening (in assignment)
d = 5.3 * i;            // widening in a binary expression (by "promotion")
i = (int) d;            // narrowing (by casting)
float f = (float) d;    //  "

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

*These type conversions all change the bit representations of the values.* In many cases, this leads to an approximation of the value that is represented. You will learn the details of these representations in COMP 273. (See my 273 lecture notes if you are interested. )

We use similar concepts of "narrowing" and "widening" for reference types 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 the narrower object can be thought of as 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. (Even with primitive types, it is difficult to order by number of bytes since `float` is wider than `long`, yet a `float` has fewer bytes).

Conversions can also occur between reference types. However, *reference type conversions do not change 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. Widening conversions from a subclass to superclass occur automatically. Here we say that we are casting *upwards* (upcasting). Upcasting is sometimes called *implicit casting.* We cast *downwards* ("downcasting") when we are casting from a superclass to a subclass. Like with primitive types, we need to be explicit when we downcast reference types.

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. A few examples are given below.

```
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 (explicit downcast)

myPoodle.show()                 //  runtime error if myPoodle referenced
                           //  a Dog object that has no show() method

((Poodle) myDog).show();  //  Explicit down cast ok: no compiler error.
```

In the last example, if `myDog` references a `Doberman` at runtime, then you get a runtime error since Dobermans aren't show dogs.

## Polymorphism (introduction)

We have seen that the declared type of a reference variable does not entirely determine the class of object that the variable can reference at runtime. At runtime, a variable can reference an object of its declared type, or it can also reference an object that is a subtype of the variable's declared type. This property, that the object type can be narrower than the declared type, is called *polymorphism*[1].

There are three separate cases to consider, depending on whether a variable's declared type is a class, an interface, or an abstract class. Suppose a reference variable has a declared type that is a class C. At runtime, that variable can reference any object of class C or any object of a class that extends C. If a variable has a declared type that is an abstract class A, then at runtime that variable can reference any object whose class extends A. If a variable has a declared type that is an interface I, then at runtime that variable can reference any object whose class implements A.

When we discussed type conversion above, we concentrated on the type checking that is done by the compiler. When we discuss polymorphism, we assume a program has compiled fine, and we are concerned with which method is invoked at runtime. The method is determined by the class that the object belongs to. Consider, for example:

[1]from Greek: poly means "many" and "morph" means forms

```
boolean  b;
Object  obj;
   :                     //  some code not specified here
if (b)
   obj = new float[23];   // an array of floats
else
   obj = new Dog();
System.out.print(obj);   //  invokes the toString() method (*)
```

The compiler cannot say for sure which `toString()` method will be invoked since the compiler doesn't know for sure what the value of `b` will be when the `if (b)` condition is evaluated. Rather, the `toString()` method must be determined at runtime, when (*) is executed and the variable `obj` references either a `float[]` or a `Dog`. In each case, there will be a `toString()` method used which is appropriate for the object. (Recall that every class has a `toString()` method.)

To understand more generally how polymorphism works, we need to understand how classes are represented in a running program.

## The Class class

When you define a class by typing some ASCII code into a `.java` file, your class definition includes various things: the name of the class, a list of fields and types, a list of methods and their signatures and the instructions of each method, modifiers such as `public`, and other info. When you compile the class, the compiler makes a `.class` file, which is stored in a directory on your computer. That directory is determined by the package name that you write in the first line of the `.java` file.

When a Java program uses a particular class, it loads this class file and uses the information in this class file to makes an object, which I will call a " class descriptor." This class descriptor is different from a `.class` file generated by a compiler, since a class descriptor is an object in a running program rather than a file. That said, the information in a class descriptor is the same (for our purposes, anyhow) as the the information in a class file. In particular, class descriptors contain the name of the class, an array of fields and types, an array of methods including the instructions of each method, a reference to the superclass, etc.

Class descriptors are objects in a running program, just like the objects that are generated by `new` commands. What class do these class descriptor objects belong to? Answer: the `Class` class.

In lecture 30, I mentioned several methods in the `Object` class, such as `hashCode(), toString(), equals(), clone()`. Another one is `getClass()`. It returns the class descriptor of the class that this object belongs to, that is, the class whose constructor invoked this object. So, the return type of `getClass()` is `Class`. For example, `myDog.getClass()` would return a reference to the `Dog` class descriptor, which is an object of type `Class`.

The `Class` class has several methods. For example, `getSuperClass()` returns the class descriptor of the superclass. The return type is `Class`. So, if `myDog` were a `Beagle`, then `myDog.getClass()` would return the `Beagle` class descriptor, and `myDog.getClass().getSuperClass()` would return the `Dog` class descriptor.

Note that the `Object` class does not have a superclass, so the following returns `null`.

```
Object obj = new Object();
System.out.println(obj.getClass().getSuperclass());
```