Interfaces, abstract classes, and polymorphism

Last lecture I introduced interfaces and abstract classes. Interestingly, these define types. That is, we can define the type of a reference variable to be an interface or an abstract class. What does that mean? If the type of a variable is an interface, this means that the variable references an object whose class implements the interface (or whose class extends a class that implements the interface). In particular, the object has the methods defined in the interface. Similarly, if the type of a variable is an abstract class, this means that the variable references an object whose class extends that abstract class. What do we gain by allowing variables to have a type that is an interface or an abstract class? One thing we gain is polymorphism.

Example: abstract class

To understand this, recall the class `Dog` and its subclasses `Beagle`, `Doberman`, etc. You might have an application in which it makes no sense to instantiate a `Dog` object, i.e. you would only be instantiating particular breed of dog. In this case, you could make `Dog` an abstract class. That way, if you want to instantiate a `Dog`, then you have to do so by instantiating the appropriate subclass, namely `Beagle`, `Doberman`, `Mongrel`, etc.

In the example below, the method `bark()` could be abstract method in the `Dog` class. Although all dogs bark, different breeds make very different barking sounds and, for example, the tone and yappiness depends on the size and breed. Each subclass would have its own concrete implementation of `bark()`.

```
Dog  myDog = new Beagle();     // myDog is polymorphic
         :
myDog = new Doberman();
         :
myDog.bark();
```

Example: interface

What does it mean for the type of a variable to be an interface? It just means that the object belongs to a class that implements the interface. Again, this is useful because it provides runtime flexibility. For example, suppose classes `A` and `B` implement interface `I`. Then the following would be allowed (by the compiler):

```
I  v = new A();       //  v is polymorphic
v = new B();
```

The only thing we are requiring of `v` is that it has the methods of interface `I`. For example, the interface `I` could have a single method.

Can you see any disadvantages of this? One disadvantage is that since `v` is of type `I`, the compiler will give an error if you invoke any method other than the one(s) in interface `I`. So there is a tradeoff here. If `I` has only a single method, you have great flexibility in that many classes can implement this interface[1]. But `v` cannot invoke any methods other than those in the interface.

[1]Careful here: It is not enough that the class contains the method(s) of that interface. The class has to be defined to implement the interface.

Casting

Primitive types (review)

You should be familiar with the basics of primitive types and casting in Java from COMP 202 (or the equivalent). You learned that (except for `boolean`) primitive types are ordered from "narrow" to "wide":

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

Widening conversions occur automatically, but a narrowing conversion requires an explicit *cast*, otherwise you will get a compiler error.

```
int    i = 3;
double x = 4.0;
x = i;           // widening conversion occurs as part of an assignment
x = 5.3 * i;     // widening conversion by promotion
i = (int) x;     // narrowing conversion by casting
```

Narrowing typically leads to an approximation. For example, when you convert from a `float` to an `int`, you discard the fractional part. Interestingly, though, approximations can occur with widening conversions as well. How to see this? Note that there are 32 bits used to represent an `int` and 32 bits used to represent a `float`, which means that you can represent 2^{32} different possible values of each. Most `float`'s have a fractional part[2] and similarly most `int` values cannot be represented exactly as `float`.

Reference types

We can think of "narrowing" and "widening" in a class hierarchy as well. For example, if class `B` extends class `A` then class `B` is considered to be narrower than `A`, i.e. `A` is wider than `B`.

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 (in the number of bits used) then you will notice that the bigger object has a narrower type. This is the opposite of what happens with primitive types, where the narrower object uses at most as many bits as the wider object.

Another difference to watch out for 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 object being referenced. A few examples will illustrate the point.

First, though, some 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.

[2]You will learn exactly how floats are defined in COMP 273. Its very cool, but now is not the time to explain – i.e. these details don't concern the general point I am making here.

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

This is analogous to widening by promotion:

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

We have not seen downcasting before, however. Let us look at a few examples of both.

Example 1 (wrapper classes)

```
Number n;              //  This is an abstract class.  Check out Java API.
Integer i;
n = new Integer(3);    //  Upcasting.
i = (Integer) n;       //  Downcasting.  Compiler allows it.  No error
                       //    runtime since object is an Integer.
n = new Double(3.14);  //  Upcasting.
i = (Integer) n;       //  Downcasting.  Compiler allows it.  But
                       //  there will be a class exception at runtime
                       //  since object is not an Integer.
```

I didn't mention this in the lecture but.. if the last line were replaced by

```
if (n instanceof Integer)    //  note keyword  "instanceof"
   i = (Integer) n;
```

then the instruction will not be executed at runtime. The keyword `instanceof` is very useful for catching class exceptions at runtime.

Example2

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

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

Notice that when we say that a "cast" occurs here, we shouldn't think of the object 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 3

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

((Beagle) myDog).chaseRabbit();  //  Explicit cast ok and no compiler
                                 //  error.  But if myDog references
                         // a Doberman 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).chaseRabbit();
}
```

Example 4

I did not cover the next example in class, so treat it as an exercise. Look at the code on the left and cover up the comments on the right. Ask yourself what each line means and which lines give a compiler error. (WARNING: This code doesn't do anything useful.)

```
  public interface I{
     public void m(I  other);
  }

  public class A implements I{
     public void m(I  a){
         A  myA = (A) a;      //  Programmer expects type A  argument
               :              //  (or descendent of A).
               :
     }
  }

  public class B implements I {
    public void m(I  b) {

        B  myB = (B) b;
        A  myA = (A) b;
        A    a = myB;       //  compiler error (cannot convert from B to A)
           myA = (A) myB;   //  compiler error (cannot cast from B to A)
             :
    }
}
```