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. 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 \( I_2 \)) can extend another interface (say \( I_1 \)). This means that \( I_2 \) inherits all the method signatures from \( I_1 \). We don’t need to write the method signatures out again in the definition of \( I_2 \). In the class diagram, we would put a dashed line from \( I_2 \) to \( I_1 \).

- Although each class (other than `Object`) directly extends exactly one other class\(^2\) 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:

  ```java
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.

---

\(^1\)But nodes do not have references to their children, i.e. subclasses.

\(^2\) 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

```java
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.

Abstract classes

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. 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;
    }
}

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)

In general, that a class cannot extend two abstract classes. The reason for this policy is the same as why a class cannot extend two classes – namely if the two superclasses were to contain two versions of a method with the same signature then it wouldn’t be clear which of these two methods gets inherited by the subclass.

One last fact is that 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. It will become more clear why, when we discuss polymorphism in an upcoming lecture.

Casting

You are already familiar with the basics of primitive types and how conversions can occur between them. Primitive types are ordered from “narrow” to “wide”, for example, char widens to int and the following go from narrow to wide:

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

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

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

    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; // "
Narrowing typically leads to an approximation. For example, when you convert from a \texttt{float} to an \texttt{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 \texttt{int} and 32 bits used to represent a \texttt{float}. This means that you can represent $2^{32}$ different possible values of each. Most \texttt{float}'s have a fractional part\footnote{You will learn exactly how floats are defined in COMP 273.} It turns out that most \texttt{int} values cannot be represented exactly as \texttt{float} values either.

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

One possible confusion is 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. 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. \textit{Casting reference types is different, however, since no change occurs to the referenced object}. Rather, the cast only tells the compiler that you (the programmer) 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 \textit{downwards} (“downcasting”) when we are casting from a superclass to a subclass, and we cast \textit{upwards} (upcasting) when we cast from a subclass to a superclass. Upcasting occurs automatically, and so it is sometimes called \textit{implicit casting}. We have seen upcasting before, e.g.

\begin{verbatim}
Dog  myDog = new Beagle();
\end{verbatim}

This is analogous to:

\begin{verbatim}
double  myDouble = 3;  // from int to double.
\end{verbatim}

We have not seen downcasting before (at least not for reference types). We will see it below.

\textbf{Example 1}

\begin{verbatim}
Dog  myDog = new Doberman();  // Upcasting.
:
Beagle  myBeagle = myDog;  // Compiler error.
   // (implicit downcast Dog to Beagle not allowed).
\end{verbatim}
Beagle myBeagle = (Beagle) myDog;  // Allowed (though runtime error
   // would occur if myDog references
   // a Doberman object).

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();
}

A question came up in class about the rules of the instanceof operator. This operator takes two arguments: the first is a reference type variable var; the second is a class 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 indeed has access to all the methods of C, e.g. whether or not myDog can invoke hunt().

A related example came up in class. One student suggested a class HuntingDog such that Beagle extends HuntingDog and HuntingDog extends Dog. Suppose the method hunt() were defined in HuntingDog.

Dog myDog = new Beagle();
if (myDog instanceof HuntingDog){ // returns true
   ((Beagle) myDog).hunt(); // compiler allows this.
}