CS 228: Introduction to Data Structures
Lecture 3
Friday, January 16, 2015

Recap

Java ***interfaces*** formalize and enforce the "separation of interface from implementation" that is one of the benefits of encapsulation. Using interfaces allows us to reduce coupling. For instance, consider the class `IntCollection` and its implementation `UnsortedIntCollection`, mentioned in the previous two lectures. Suppose we have a program that uses an `IntCollection` called `c`, declared as follows.

```
IntCollection c =
  new UnsortedIntCollection();
```

Now, suppose we're provided a much faster implementation of `IntCollection`, called `FastIntCollection`. Then, to use this new implementation, we simply replace the above line by

```
IntCollection c =
  new FastIntCollection();
```

Nothing else needs to be changed in the program! The profound usefulness of interfaces will become even clearer over the next few weeks.

Interface implementation — whereby one provides code for all the methods that an interface declares — is one of the ways inheritance is implemented in Java. In last lecture's example, `Bird` and `Person` are both implementations of type `ISpeaking`. Thus `Bird` and `Person` objects inherit the `speak()` method from `ISpeaking`. Of course, since `ISpeaking` has no implementation of `speak()`, `Bird` and `Person` need to provide their own implementations.

Recall that

- it is ***legal*** to declare an object whose type is an interface, but

- it is ***illegal*** to try to instantiate an interface variable

If an interface variable refers to an object, that object must belong to a class that implements that interface. For example:

```
ISpeaking b = new Bird();  // OK
b.speak(); // prints “tweet”
```

After the preceding statements, it is OK to do:

```
b = new Person();  // OK
b.speak(); // prints “Hi!”
```

This works because `Person` is an `ISpeaking` object, so b can hold a reference to a `Person` object. More precisely, the ***static type*** (or ***compile-time type***) of b is `ISpeaking`. The type of object b references is its ***dynamic type*** (or ***run-time type***). As we shall discuss in detail later, method invocations are always done using an object’s dynamic type.

Inheritance by Class Extension

Class extension is another form of inheritance. It allows a subclass to inherit all attributes and operations of its superclass. This helps allows the reuse of code. Additionally, class extension allows us to

- ***add*** new attributes or behavior (new instance variables and/or methods) to a class and

- ***modify*** behavior by ***overriding*** existing methods.

Example. Consider the interfaces

```
public interface ISpeaking
{
  void speak();
}

public interface ILicensable
{
  License getLicense();
}
```

Let us ignore the `License` class, which is not needed for this discussion.

Next, we define a class `Dog`, whose objects have a name and license.

```
public class Dog implements ISpeaking,
ILicensable
{
  private String name;
  private License license;

  public Dog(String name, License license)
  {
    this.name = name;
    this.license = license;
  }

  @Override
  public void speak()
  {
    System.out.println("woof");
  }
```

```
  @Override
  public License getLicense()
  {
    return license;
  }

  public String getName()
  {
    return name;
  }
}
```

Now, define a class `Retriever` that extends `Dog`. `Retriever` ***inherits*** the existing attributes and operations of `Dog`: the *name* and `License` fields and the `getName` and `getLicense` methods. It also

- ***adds*** some behavior (the `retrieve` method), and
- ***modifies*** the existing behavior (***overrides*** the `speak` method).

```java
public class Retriever extends Dog
{
  public Retriever(String name,
    License license)
  {
    super(name, license);
      //call superclass constructor
  }

  @Override
  public void speak()
  {
    System.out.println("raooou");
  }

  public Bird retrieve()
  {
    return new Bird();
  }
}
```

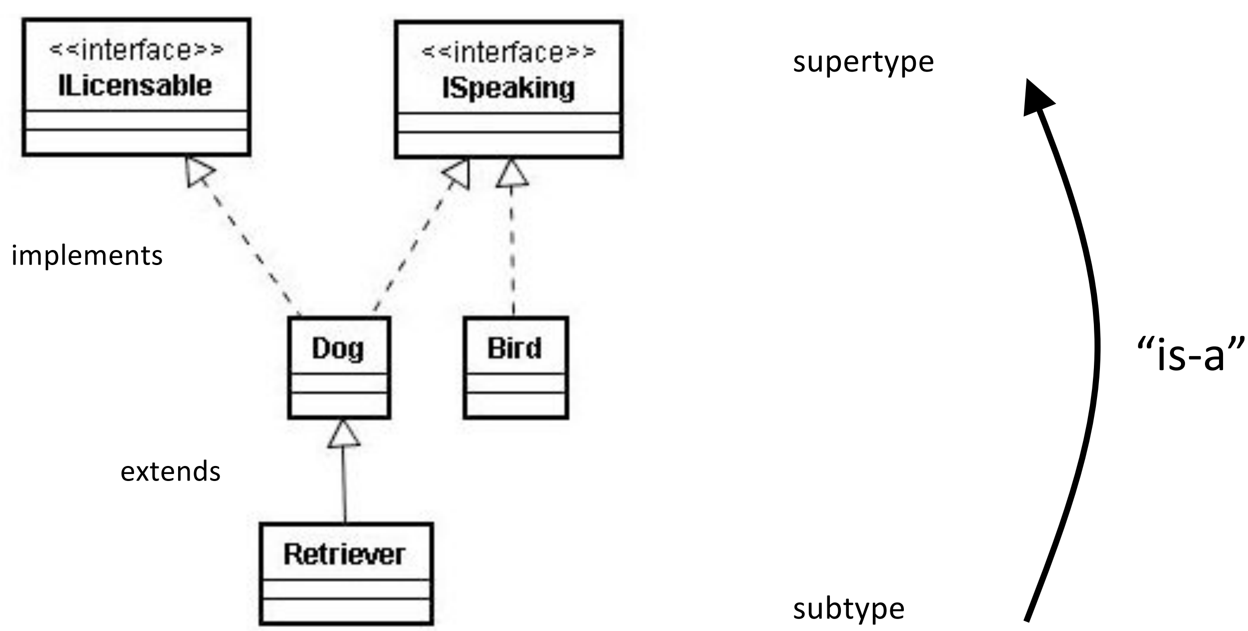
Note the use of `super` in `Retriever`. This serves to call the constructor for the superclass (`Dog`).

We say `Retriever` is a ***subclass*** or ***subtype*** of `Dog` (it is also a subtype of `ISpeaking` and of `ILicensable`) and `Dog` is a ***superclass*** or ***supertype*** of `Retriever`.

Notice that a Java class can ***implement*** more than one interface, but can ***extend*** only ***one*** other class.

Class Hierarchies

A ***class hierarchy*** can be represented by a ***class diagram***. For the example we have just seen, the diagram would look like this:



The superclass/subclass (supertype/subtype) relationships are indicated by the arrows with the big triangles. A dotted line means "***implements*** (an interface)" and a solid line means "***extends*** (a class)". Drawing all the arrows pointing upwards allows us to see the subtype-supertype relations easily.

We also call the subtype relation the "***is-a***" relation: A `Retriever` ***is a*** type of `Dog`, a `Dog` ***is a*** type of `ISpeaking`.

Note. The picture in the previous page is called a ***UML diagram***, where UML stands for "Unified Modeling Language". UML diagrams can get quite sophisticated. For example, you will often see the names of attributes and methods shown in the boxes too.

Polymorphism and Dynamic Binding

Polymorphism means that a variable of a given type T can hold a reference to an object of any of T's subtypes.

Consider the statement below.

```
1.  ISpeaking s = new Dog("Ralph", null);
```

Statement 1 does a few things:

- It invokes a constructor to instantiate an object of type `Dog`. The constructor returns a reference to the object.

- It declares a variable `s` of type `ISpeaking`. In fact, it declares that `s` will reference an `ISpeaking` object.

- It makes `s` point to the new `Dog` object.

Important point: A `Dog` object can masquerade as an `ISpeaking` object, because ***every `Dog` object is an `ISpeaking` object.***

However, the reverse is not true. See the UML diagram. The basic principle at work here is:

An object can hold a reference to any of its subtypes.

Memorize the following two definitions.

Static type: The type (class) of a variable. Also known as ***compile-time type***.

Dynamic type: The class of the object the variable references. Also known as ***run-time type***.

Thus, in the example above,

- the ***static type*** of `s` is `ISpeaking`
- the ***dynamic type*** of `s` is `Dog`

Now suppose we carry out the following statement.

```
2.  s = new Bird();
```

This is OK, because `Bird` is a subtype of `ISpeaking`. The static type of `s` remains `ISpeaking` (it will ***always*** be that), but its dynamic type has gone from `Dog` to `Bird`.

```
3.  Dog d = new Retriever(“Clover”, null);
```

This is OK, because `Retriever` is a subtype of `Dog`.

```
4. Retriever r = (Retriever) d;
```

This is also OK. The rule here is:

Even though the ***static type*** of `d` is `Dog`, its ***dynamic type*** is `Retriever`.

Therefore, `d` can be assigned to `r`. Nevertheless, an explicit downcast is still required. Here is the reason.

The Java compiler checks the correctness of each line ***based only on the static types*** of the variables, not on their ***run-time*** types.

Casts change the static type of variables. When Java looks at line 4, it does not know the dynamic type of `d`. It *does* know that `d` can refer to any sort of `Dog`, a plain one or a `Retriever`. The cast is our way to promise to Java that `d` indeed refers to a `Retriever` at this point in the execution.

The next statement yields a compiler error:

```
5.  d = new Bird();
```

`Bird` ***not*** a subtype of `Dog`; see the UML diagram.

The next statement compiles, but fails at run time.

```
6.  d = (Dog) s;
```

Since the compiler only considers static types, it sees nothing wrong with statement (6): the static type of `s` is `ISpeaking` and, since a `Dog` is an `ISpeaking` object, the (down) cast is OK, in principle. When we run the statement, though, the system discovers that the dynamic type of s is `Bird`. Since we cannot cast a `Bird` to a `Dog` (a `Bird` is not a `Dog`), the statement triggers a `ClassCastException` — the exception that is thrown when "the code has attempted to cast an object to a subclass of which it is not an instance."[1]

Now let's start invoking methods.

```
7.  Dog d = new Dog("Ralph", null);

8.  d.speak();                // "woof"

9.  d = new Retriever("Clover", null);

10. d.speak();                // "raooou"
```

[1] http://docs.oracle.com/javase/8/docs/api/java/lang/ClassCastException.html

Here is the basic rule:

> ***Dynamic Binding:*** When you invoke a method `m()` on a variable `v`, the code that gets executed is determined by the **run-time** type of `v`.

That is, when we invoke an overridden method, Java calls the method for the object's ***dynamic*** type, regardless of the variable's static type. This is called ***dynamic method lookup***, because Java automatically looks up the right method for a given object at run time.