CS 228: Introduction to Data Structures
Lecture 4
Wednesday, January 21, 2015

Recap: Static Type Checking & Dynamic Binding

At compile time, Java uses ***static type checking***. That is, it checks for type errors using the ***static types*** of the variables, not their run-time types. Casts change the static type of variables.

At run time, Java uses ***dynamic binding***. That is, when the code invokes a method `m()` on a variable `v`, the code that gets executed is determined by the ***run-time*** (***dynamic***) type of `v`, regardless of `v`'s static type. Java implements dynamic binding via ***dynamic method lookup***.

More Examples

```
11. ISpeaking s = new Dog(“Ralph”, null);
```

```
s.speak();                  // “woof”
```

The next statement produces a compile-time error, since `ISpeaking` has no `getName` method

```
s.getName();
```

The following downcast is OK, since `s` really does refer to a `Dog` at runtime.

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

The next two statements go through:

```
d.getName();                // “Ralph”
d = new Retriever(“Clover”, null);
```

But the next one produces a compile-time error, because `Dog` has no `retrieve` method:

```
s = d.retrieve();
```

In contrast, the next two statements are OK, since d really refers to a `Retriever` at runtime.

```
s = ((Retriever) d).retrieve();

s.speak();                    // “tweet”
```

Abstract Classes

Suppose we introduce a `Cat` class:

```
class Cat implements ISpeaking
{
  private String name;

  public Cat(String name)
  {
    this.name = name;
  }

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

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

But now there is duplicated code in the `Cat` class and in the `Dog` class: There is a `name` field and a `getName()` method in both. Duplicated code is generally viewed as a bad thing.

To streamline the code, we can ***factor*** the common parts into a superclass of `Dog` and `Cat`, called, say, `Pet`. Then, we let `Dog` and `Cat` inherit the `getName()` method from `Pet`.

But it doesn’t make sense to try to implement the `speak()` method for `Pet`, since it could be either a `Cat` or a `Dog` (or something else). To allow the class to have a method that is declared but not implemented, we can make the class `abstract`:

```
public abstract class Pet implements
ISpeaking
{
  private String name;

  protected Pet(String name)
  {
    this.name = name;
  }

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

Abstract classes allow you to provide partial implementations, while leaving some details for subclasses to implement. The rules are:

- An abstract method is declared with the `abstract` keyword, and ends with a semicolon instead of a pair of braces with a method body.

- All methods of an interface are automatically abstract.

- If a class contains an abstract method, the class must also be declared abstract.

- You cannot create an instance of an abstract class with new.

Declaring that we implement the `ISpeaking` interface, but not actually providing a method body, is equivalent to having the following line in the `Pet` class

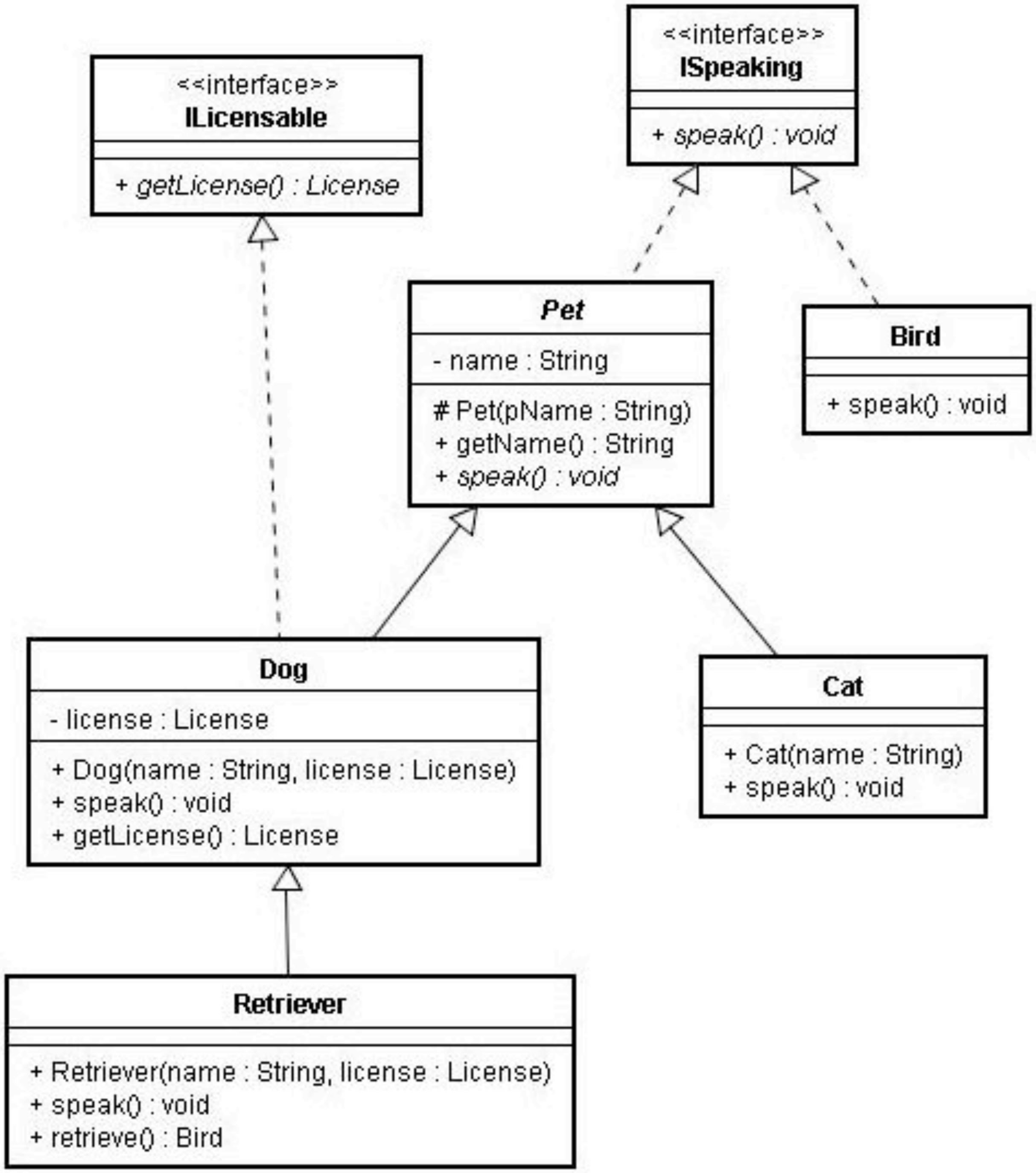
```
  public abstract void speak();
```

Now we can remove the `name` field and `getName` method from `Dog` and `Cat`, and update the constructors:

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

Note the use of the `protected` keyword in the `Pet` constructor; this makes the constructor accessible from subclasses or from any class within the package.

The next figure shows the updated class hierarchy.



<<interface>>
ILicensable
+ getLicense() : License
<<interface>>
ISpeaking
+ speak() : void
Pet
- name : String
# Pet(pName : String)
+ getName() : String
+ speak() : void
Bird
+ speak() : void
Dog
- license : License
+ Dog(name : String, license : License)
+ speak() : void
+ getLicense() : License
Cat
+ Cat(name : String)
+ speak() : void
Retriever
+ Retriever(name : String, license : License)
+ speak() : void
+ retrieve() : Bird

Interfaces versus Abstract Classes

A Java interface is in many ways like an abstract class. One difference is that, while a Java class can inherit from only one class — even if the superclass is `abstract` —, a class can implement (inherit from) as many Java interfaces as you like.

Up to Java 7, a Java interface could not implement any methods, nor could it include any fields except "`final static`" constants. It could only contain method prototypes and constants. Starting with Java 8, “`default`” and `static` methods may have be implemented in an interface definition. We will not use these new features in this course, but it is important to be aware that they exist.

The guiding principle behind the distinction between abstract classes and Java interfaces is to avoid the problems associated with "multiple inheritance" (MI), which is what happens when a subclass can inherit from several superclasses. This is allowed in languages like C++. MI is responsible for some of the scariest tricks and traps of C++. Problems arise, for example when two superclasses define differing methods or fields with the same name, but this is not the only problem that can occur. You will study this and other issues in programming language design in CS 342. Java avoids these problems by not having MI, except in the limited form of Java interfaces.

Access Modifiers

There are four access modifiers that may be applied to fields and methods.

- **`private`:** accessible only within the class, used for implementation details

- (none) aka **"package-private":** accessible from any class within the package

- **`protected`**: accessible from subclasses (or from any class within the package)

- **`public`:** accessible from any class

We sometimes use `protected` access when designing a class to be extended: if you need subclasses to have access to a variable or method that is not part of the public API, make it `protected`.

Note that if a variable is non-private, you should never ***shadow*** it (re-declare a variable with the same name) in the subclass, even though this is allowed by the compiler.

You can override methods, but there’s no such thing as overriding variables!

Package-private access is rarely used. The `Animals.java` example contains several classes and interfaces with package-private access. This was done simply to enable us to put them all in the same file, since a public class or interface is required to be in its own file.

Package-private (default) access is ***almost*** synonymous with `protected` access. The `protected` modifier specifies that the member can only be accessed within its own package (as with package-private) and, ***in addition***, by a subclass of its class in ***another*** package.

The Root of the Java Class Hierarchy

At the root of the Java class hierarchy is the class `java.lang.Object`. Every class in Java is a subclass of this class.

The `Object` class has several predefined methods. One of them is **`toString()`**, which returns a String representation of an object. When Java is asked to print an object `x` — for instance when we invoke `System.out.println(x)` — what is actually printed is `x.toString()`.

The default implementation of `toString()` returns a string consisting of the class name, the ‘`@`' character, and the unsigned hexadecimal representation of the “hash code” of the object (we’ll see hashing and hash codes towards the end of the semester). Since this string is typically meaningless to us, we will usually override `toString()` to provide a more useful description. Another predefined method in the `Object` class is **`getClass()`**, which returns the runtime class of an object. This class is represented by a special, unique, object of type `Class`.

You can find a complete list of the methods in the Object class online[1]. In the next few lectures, we will focus on two of these methods: `equals()` and `clone()`.

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