Software Development (CS2500)

Lecture 25: Inheritance (Continued)

M.R.C. van Dongen

December 1, 2010

Contents

1 Introduction

This lecture continues our study of *inheritance*. Specifically, we shall:

- Exploit the *transitivity* of inheritance to maximise code reuse, save coding effort, and increase program maintenance.

- Use the *is-a test* to cross-check our inheritance hierarchy.
- Use the *has-a test* to determine when object reference attributes are required.
- Learn how to control inheritance.
- Study the difference between overriding and overloading.

This lecture is based on [Sierra and Bates, 2004, Chapter 7] and parts of [Lewis and Loftus, 2009, Chapter 9].

2 Last Monday

Last Monday we studied inheritance. Inheritance increases the ability to reuse implementation effort.

- General-purpose code is put in a superclass.
- More specific code is put in subclasses.
- The superclass defines more general behaviour.
- The subclasses defines more specific behaviour. A subclass can *inherit* behaviour from its superclass. It can *override* the general behaviour with more specific behaviour. Finally, a subclass can provide additional behaviour.

Inheritance separates class-specific from more general code. This allows us to make local changes in a subclass without affecting code in other classes. Inheritance also defines a common protocol. If the superclass defines a method with a given signature then method calls with this signature can be used in the superclass and in any of its subclasses.

3 Transitivity

In this section we shall study *transitivity* and how it relates to inheritance.

Let $\oplus$ be a binary relation. We write $a \oplus b$ if and only if $a \oplus b$ is true. The following are some examples.

- If $\oplus$ is $<$ then $1 \oplus 3$ is true.
- If $\oplus$ is $=$ then $1 \oplus 3$ is false.
- If $\oplus$ is $\neq$ then $1 \oplus 3$ is true.
- If $\oplus$ is 'is a parent of' then `Homer` $\oplus$ `Bart` is true.

We say '$a \oplus b$' if $a \oplus b$ is true. We say 'not $a \oplus b$' if $a \oplus b$ is false.

Relation $\oplus$ is called *commutative* if for all a and b the following holds:

- $a \oplus b$ is true if and only if $b \oplus a$ is true.

Both $=$ and $\neq$ are commutative. Most binary relations are not commutative: $1 < 2$ but not $2 < 1$. The 'is a parent of' relation is also not commutative. For example `Homer` is a parent of `Bart` is true but `Bart` is a parent of `Homer` is not true. The integer divisability relation is also not commutative: 1 divides 2 but not vice versa.

Relation $\oplus$ is called *transitive* if for all a, b, and c the following holds:

- If $a \oplus b$ is true; and
- If $b \oplus c$ is also true;
- Then $a \oplus c$ is also true.

Both $<$ and $\geq$ are transitive. The reachability relation in graphs is transative.[1] For example, if node a can be reached from node b and if node b can be reached from node c then node a can be reached from node c. Since this is true for all a, b, and c it follows that reachability is transative. The relation 'is an ancestor of' is also transitive: `Abraham` is an ancestor of `Homer` and `Homer` is an ancestor of `Bart`. Therefore `Abraham` is an ancestor of `Bart`. Most binary relations are not transitive: $1 \neq 2$ and $2 \neq 1$ but not $1 \neq 1$. The 'is a parent of' relationship is also not transitive. For example, '`Abraham` is a parent of `Homer`', and '`Homer` is a parent of `Lisa`', but not '`Abraham` is a parent of `Lisa`'.

The subclass and superclass relationships are both transitive. This is best seen by considering the class extension relationship. If class A extends class B then A is more specific than B.

- A `Dog` is more specific than a `Canine`; and
- A `Canine` is more specific than an `Animal`.
- Therefore, a `Dog` is more specific than an `Animal`.

We can also use the 'is more general than' relationship:

- An `Animal` is more general than a `Canine`; and
- A `Canine` is more general than a `Dog`.
- Therefore, an `Animal` is more general than a `Dog`.

A method may be overridden only once per class. However it may be overridden in subclasses of that class. And overridden in subclasses of that subclasses. And so on. When a method is called, the `Java` virtual machine will always call the method using the lowest method definition in the class hierarchy. By exploiting transitivity you can maximise code reuse. You should try and avoid overriding a given method in all classes at the botom of the class hierarchy. Instead you should try and override method definitions as high up the inheritance hierarchy as possible. By doing this, the overridden method can be used anywhere below in the hierarchy. Effectively, this maximises code reuse,saves coding effort, and increases program maintenance.

[1] Here we say that a node a in a graph is reachable from a node b in a graph if we can reach a from b by following a sequence of edges in the graph.

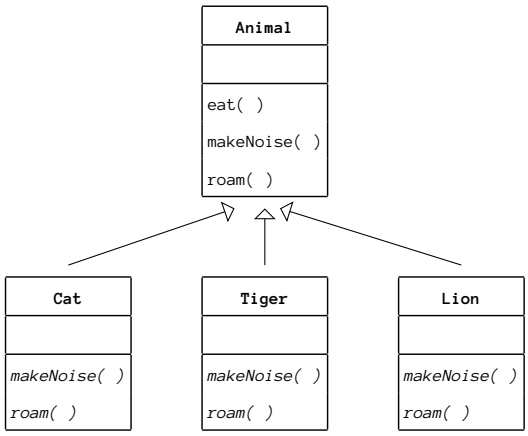

Figure 1: Larry's class design.

Figure 1 depicts a part of Larry's class design. This design does not take into account that all `Animals` at the bottom of this class diagram are `Feline`. In Larry's class design, there is no structure: there are two class levels and all overrides are at the lowest level. In total there are 6 overrides: 3 for `makeNoise( )` and 3 for `roam( )`.

Figures 2 and 3 look more like Brad's class design. These designs have an additional intermediate class level which sits between the `Animal` class and the subclasses of the `Filine` class.

The design which is depicted in Figure 2 has all overrides at the lowest level. It requires 3 different overrides for `makeNoise( )` and 3 identical overrides for the method `roam( )`.

In the design which is depicted in Figure 3 the overrides for the method `roam( )` are factored out and moved as high up the class hierarchy as possible. This class design also requires 3 overrides for `makeNoise( )` but only one override for `roam( )`. For this design it results in an overall saving of two overriding definitions.

4 The Is-A and Has-A Tests

4.1 A Test for Inheritance

Designing a class hierarchy is an art, more than a science. It is almost never possible to get things right from the start. For example, which classes should you use? Even if you have the classes, then it may not always be clear which classes to put at the top levels, which classes to put at the intermediate levels, and which classes to put at the bottom levels of the class hierarchy.

The *is-a test* provides some help to catch early mistakes. The test is designed to check when a subclass relationship is correct. The following describes how it works. Let A and B be two classes (nouns). If 'every

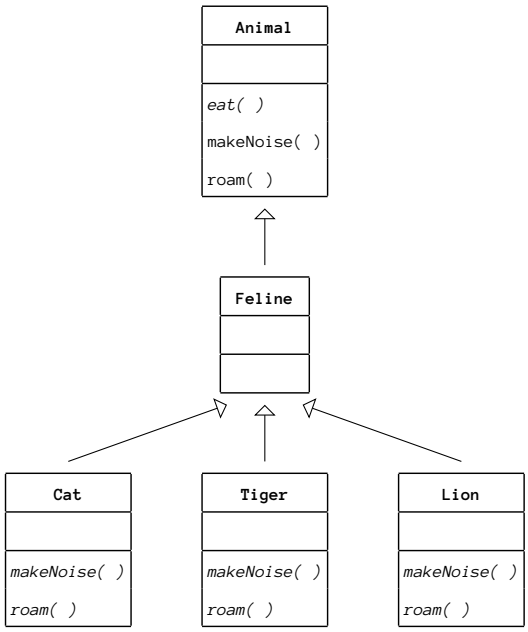

Figure 2: Class design with intermediate class and low overrides.

A is-a B' is correct (at the noun level), then making class *A* a subclass of *B* is correct.

The following are some examples.

- Every `Dog` is-an `Animal`. This is true so `Dog` can be a subclass of `Animal`.
- Every `Animal` is-a `Dog`. This isn't true so `Animal` cannot be a subclass of `Dog`.
- Every `Apple` is-a `Pear`. This isn't true so `Apple` cannot be a subclass of `Pear`.
- Every `Pear` is-an `Apple`. This also isn't true so `Pear` also cannot be a subclass of `Apple`.
- Every `Cat` is-a `Feline`. This is true so `Cat` can be a subclass of `Feline`.
- Every `Feline` is-a `Cat`. This isn't true so `Feline` cannot be a subclass of `Cat`.

The "extends test" is not so robust:

- `Cat` extends `Feline`. This is true so `Cat` can be a subclass of `Feline`.
- `Feline` extends `Cat`. This isn't true so `Feline` cannot be a subclass of `Cat`.

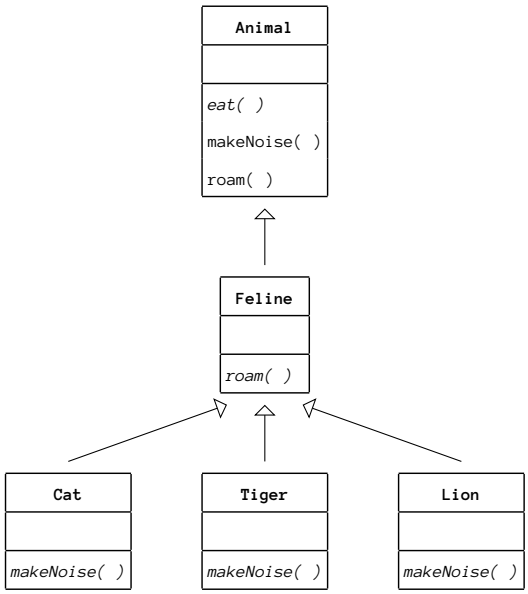

Figure 3: Class design with intermediate class and high overrides.

- `Conservatory` (`Sunroom`) extends `House`. This is true but `Conservatory` cannot be a subclass of `House`. For example, let's assume that `Conservatory` extends `House`. Then `Conservatory` should inherit all `House` methods and `Conservatory.ringDoorBell( )` does not make sense, and `Conservatory.lightFireplace( )` also does not make sense.

4.2 An Association Test

The previous example demonstrates that even if it makes sense for a class to use another class, it may not always make sense to make the other class a subclass. The `House` uses/requires/has access to the `Conservatory`. We've seen that `Conservatory` cannot extend `House`. Still it makes perfectly sense for the `House` class to have a `Conservatory` attribute.

```Java
public class House {
    private Bell doorBell;
    private Window[] groundfloorWindows;
    private Window[] firstFloorWindows;
    private Conservatory conservatory;
    …
}
```

The following is another example. `MouseCursor` cannot be a subclass of `Window`. Still it makes perfectly

sense for the `Window` class to have a `MouseCursor` attribute.

```Java
public class Window {
    private Position currentPosition;
    private Point lowerLeft;
    private Point upperRight;
    private MouseCursor cursor;
    …
}
```

If a class *A* has a class-*B* attribute then class *A uses B*. For example:

- `Window` uses a `MouseCursor`.
- `House` uses a `Conservatory`.

The *has-a* test determines when a class uses another class. If '*A* has-a *B*' then *A* can have a class-*B* attribute.

The following are some examples.

- Every `House` has-a `Conservatory` (possibly `null`). This is true so `House` should have a `Conservatory` attribute.
- Every `Window` has-a `MouseCursor`. This is true so `Window` should have a `MouseCursor` attribute.
- Every `Animal` has-a `Cat`. This isn't true so `Animal` shouldn't have a `Cat` attribute.
- Every `Cat` has-an `Animal`. This also isn't true so `Cat` shouldn't have an `Animal` attribute.

5 Controlling Inheritance

In `Java` a subclass inherits all `public` methods and attributes. There are some techniques to control inheritance and method overrides:

- Make the superclass `final`. Making a class `final` ensures the class cannot be extended.
- Make a method `final`. Making a method `final` ensures the method cannot be overridden.

5.1 Making the Class Final

The following demonstrates how to make a class `final`: you simply add the `final` keyword before the `class` keyword. By making the class `final` it becomes impossible to extend it.

```java
public final class Europe {
    // You may not extend this class.
    …
    public void countdown( ) {
        System.out.println( "We're leaving together" );
        System.out.println( "But still it's farewell" );
        System.out.println( "And maybe we'll come back," );
        System.out.println( "To earth who can tell?" );
        …
    }
}
```

Why would you ever want to make a class `final`? The answer is related to security and maintenance. In both cases, the underlying reason is that inheritance violates encapsulation. By allowing method overrides, you allow client classes to change the intended method behaviour. In theory this is almost as bad as providing the client classes direct access to superclass attributes. This time methods — not attributes — are exposed to modification.

Security: The first reason for making a class `final` is security: you want to make sure the class is used as it is supposed to be used. If you don't make the class `final`, the class can be extended and its non-`final` methods can be overridden. If methods are overridden they can be made behave the wrong way. If methods start to behave incorrectly it may become impossible to enforce certain invariants. For example, a `String` should behave as as `String`. In particular this means that the length of the string should be equal to the number of characters in the `String`. If the `length( )` method is overridden and returns the wrong length then this may pose problems for a method that displays the `String`.

Maintenance: A second reason for not allowing class extensions is related to maintenance. Here it is important to avoid client classes which start to rely on different behaviour. This subclass behaviour may break if a superclass implementation is upgraded.

5.2 Simulating Inheritance

In this section we shall study a technique which lets us simulate inheritance. The key to the technique is *class composition*.

Class A is *composed* of class B if

- A uses a class B attribute.
- Class A objects own the attribute: there should be only one object that uses the attribute: aliases should not be allowed.
- Class A controls object B: no other object can change object B.

Composition lets you safely simulate inheritance.

```Java
// class Animal is now final.
public class Dog {
    // Dog owns Animal object reference.
    private Animal animal = new Animal( );
    …
    // Default eating behaviour.
    public void eat( )  { animal.eat( ); }
    // Special roaming behaviour.
    public void roam( ) { … }
}
```

5.3 Making the Method Final

The following demonstrates how to make a method `final`: you simply add the `final` keyword before the method's return type. By making the method `final` it becomes impossible to override it. In the following example, the class `Europe` may be extended. Subclasses may override the method `playOtherDreadfulTune( )`, but they may not override the method `countdown( )` because it's `final`.

```Java
public class Europe {
    // You may not override this method.
    public final void countdown( ) {
        System.out.println( "We're leaving together" );
        System.out.println( "But still it's farewell" );
        System.out.println( "And maybe we'll come back," );
        System.out.println( "To earth who can tell?" );
        …
    }
    // You may override this method.
    public void playOtherDreadfulTune( ) { … }
}
```

6 Method Overloading

6.1 Method Signatures

Formally, the *signature* of a method is the list consisting of the method's argument types. The signature does not include the method visibility. The signature does not include the `return` type.

The following is an example with three method definitions. The first method has signature '`int`, `int`', the first method has signature '`int`, `int`, `int`', and the last method has signature '`double`, `double`'.

```Java
...
private int sumTwo( int a, int b ) {
    return a + b;
}
public int sumThree( int a, int b, int c ) {
    return sumTwo( a, sumTwo( b, c ) );
}
public double sumFour( double a, double b ) {
    return a + b;
}
```

The order of the types in the argument list matters, so the signatures of the following methods are different. The first signature is '`int`, `char`' and the second signature is '`char`, `int`'.

```Java
...
private void exampleA( int a, char b ) {
    ⟨stuff⟩
}

private void exampleB( char a, int b ) {
    ⟨stuff⟩
}
```

6.2 Method Overloading

Java allows several *signatures* for methods with the same name. Two methods are said to *overload* each other if: they have the same name, and they have different signatures. When two methods overload each other, "the" method — the name — is said to be overloaded.

The following is an example with three method definitions. Each of the defined methods is called '`sum`' but their signatures are different. Because the three methods have the same name but different signatures, the method(s) `sum` are overloaded.

```Java
...
private int sum( int a, int b ) {
    return a + b;
}
public int sum( int a, int b, int c ) {
    return sum( a, sum( b, c ) );
}
public double sum( double a, double b ) {
    return a + b;
}
```

6.3 Overriding versus Overloading

It is a common mistake to confuse overriding with overloading.

Overriding: Overriding means redefining an *existing* method of a superclass.

Overloading: Overloading means defining a *new* method with a different signature.

The following is an example.

```java
public class Dog extends Animal {
    ...
    // new method
    public void eatPedigree( ) {
        // ⟨eat one portion of branded dog food⟩
    }

    // Overloading.
    public void eatPedigree( int portions ) {
        // ⟨eat several portions of branded dog food⟩
        while (portions –) {
            eatPedigree( );
        }
    }

    // Overriding
    @Override
    public void roam( ) { ... }

    // Overloading.
    public void roam( Dog friend ) {
        roam( );
        friend.roam( );
    }
}
```

7 For Friday

For Friday: study the notes, study Chapter 7, Pages 177–191, and carry out the exercises on Pages 192 and 193.

References

[Lewis and Loftus, 2009] John Lewis and William Loftus. *Java Software Solutions* Foundations of Program Design. Pearson International, 2009.

[Sierra and Bates, 2004] Kathy Sierra and Bert Bates. Head First *Java*. O'Reilly, 2004.