# Software Development (CS2500)

Lecture 34: **Interfaces**

M.R.C. van Dongen

January 17, 2011

## Contents

## 1 Outline

Today's lecture starts by finishing our study of polymorphism. We shall continue by studying *multiple inheritance*. We shall finish by studying *interfaces*.

## 2 Polymorphism

This section completes our investigation of polymorphism. It starts by recalling some of the theory from the previous lecture. This is continued by explaining the relationship between object reference type, the type of the referenced instance, and the instance methods which are available using the object reference.

We've already seen that the following doesn't work. According to the method's signature, `copy( )` returns an `Object` reference and as far as the compiler is concerned that's all it knows. The first two assignments are fine, but the third is not: `Java` only lets you assign a (sub)⟨class⟩ reference value to a given ⟨class⟩ reference variable.

Don't Try this at Home

```
public static void main( String[] args ) {
    Cat felix   = new Cat( );
    Object copy = copy( felix );
    Cat copyCat = copy( felix );
}

private static Object copy( Object object ) {
    return object;
}
```

The following does work because of the *(Liskov) substitution principle*, which says that if a reference is expected of an instance of a certain class then you may substitute a reference to an instance of any subclass of that class.

Java

```
public static void main( String[] args ) {
    Cat felix = new Cat( );
    Object copyCat = felix;
}
```

The following also doesn't work. The reason why this doesn't work is that the compiler's decision whether you can use a method is based on the object *reference* type, not on the actual instance type. Here the reference type is `Object` and the `Object` class does not define the instance method `makeNoise`.

Don't Try this at Home

```
public static void main( String[] args ) {
    Cat felix = new Cat( );
    Object copyCat = felix;
    object.makeNoise( );
}
```

The following shows the general technique to overcome the previous problem. We have two options.

1. We assign the object reference to a `Cat` object reference variable and use this variable to invoke the method `makeNoise( )`. To do this we first convince the compiler that this is OK. We do this by casting the `Object` reference which is returned by the expression `copyCat` to a `Cat` reference.

2. We don't use the assignment but call the method on the expression which "does" the casting.

The following demonstrates both techniques in one example.

```java
public static void main( String[] args ) {
    Cat felix = new Cat( );
    Object copyCat = felix;
    Cat copy = (Cat)copyCat;
    copy.makeNoise( );
    ((Cat).copyCat).makeNoise( );
}
```

* * *

We shall finish this section by once more recalling the basic rules which are used to determine which instance methods we can use when working with polymorphic reference variables.

Consider a class Class. Let SubClass be a subclass of Class. Furthermore, let's assume we have two variables: `Class super`, and `SubClass sub`. Finally, let's assume that `sub` references a `SubClass` instance. Polymorphism and the substitution principle lets us assign the value of `sub` to `super`.

```java
SubClass sub = new SubClass;
Class super  = sub;
```

- Using `sub` we can use all `SubClass`-specific instance methods/attributes.
- Using `super` we can't: the compiler uses *the reference type* to determine which methods/attributes can be used and not the type of the referenced object.
- Using `super` we can use all `Class`-specific instance methods/attributes.
- By inheritance, `sub` also lets us use them.
- Using our remote control analogy, we may regard `super` as a cheaper version of remote control. Both remotes have the buttons for the `Class` instance methods/attributes. Only `sub` has the buttons for the `SubClass` instance methods/attributes.

## 3 Multiple Inheritance

In this section we shall study *multiple inheritance*, which involves having several immediate superclasses. We shall start with a case study which demonstrates what happens if we want to add some functionality to the `Dog` and `Cat` classes of our Fota application. Next we shall study multiple inheritance.

In our case study we are going to add a method `void beFriendly( )` for pets (`Dog` and `Cat` objects) in our Fota application. We're going to look at four different ways to achieve our goal. As we shall see, each of them has their own advantages and disadvantages. In the next section we shall do it the proper way.

### 3.1 Adding the `Pet` Method to the `Animal` Class

As our first option we may decide to add the `Pet` method to the `Animal` class. The following are some advantages and disadvantages:

**Pros:** The are two main advantages: (1) all `Pets` will inherit `Pet` behaviour, and (2) `Animal` can act as a polymorphic type for `Pets`.

**Cons:** There are also disadvantages: (1) all non-`Pets` will also get `beFriendly( )` behaviour, and (2) more than likely we'll have to override the `beFriendly( )` method in the `Dog` and `Cat` classes because `Dogs` and `Cats` have different friendly behaviour. Then again, it is probably unavoidable that we have to provide different overrides for this method in the `Dog` and the `Cat` class.

**Conclusion:** Clearly the disadvantages — especially (1) — outweigh the advantages.

**Cause:** The reason for the bad design is that the Is-A test fails for non-`Pets`. For example, the test `Hippo` is-an `Animal` makes no sense if each `Animal` is supposed to have an instance method `beFriendly( )`.

## 3.2 Making `Animal` Class Abstract

For our second option we also decide to add the `Pet` method to the `Animal` class. However, this time we make the `Animal` class abstract and make `beFriendly( )` an abstract method. The following are some advantages and disadvantages:

**Pros:** The advantages are better than before. This time we are forced to properly implement `beFriendly( )` for `Pet` and non-`Pet` classes. For example, we cannot forget implement the method as it is abstract and has to be implemented for the concrete classes `CaT` and `Dog`. As with the previous design, `Animal` can act as a polymorphic type for `Pets`.

**Cons:** Our major disadvantage is that we must override `beFriendly( )` for *all* concrete classes.

**Conclusion:** This design is worse than Option I.

**Cause:** The reason for the bad design is that with this design the Is-A test fails for non-`Pets`.

## 3.3 Putting the `Pet` Method where it Belongs

For our third option we only add the `Pet` method to the `Dog` and `Cat` classes. However, we do not provide a superclass defining the method `beFriendly( )`. The following are some advantages and disadvantages:

**Pros:** The following are some advantages. This time `beFriendly( )` is only needed in the classes where it belongs. Implementing `beFriendly( )` requires little effort. As a consequence all `Animals` now behave properly.

**Cons:** The following are some disadvantages. (1) The method `beFriendly( )` not being abstract, the compiler cannot help us implement it properly. For example, we may implement the method as '`public void beFriendly( )`' in the `Cat` class but as '`public void befriendly( )`' in the `Dog` class. The definition of the method in the `Dog` class has a typo. As a result, the `Cat` and `Dog` classes fail to implement the common protocol. (2) The main disadvantage is that we lose a proper polymorphic type for `Pets`. Even if we use `Object` or `Animal` as a polymorphic for `Pets` we still won't have an easy way to call `Pet`-specific methods.

**Conclusion:** This design makes `Pets` difficult to work with.

**Cause:** The reason why this design fails is that it is incapable of providing polymorphism, which makes most applications “tick”.

## 3.4 Two Superclasses for `Pets`

As a fourth (hypothetical) option we may decide to introduce an abstract `Pet` class and make it a second superclass of the `Cat` and `Dog` classes.

**Pros:** The following are the advantages. The `beFriendly( )` method is where it belongs. Implementing `beFriendly( )` requires little effort. A clever compiler should be able to help us properly override `beFriendly( )`. Most importantly, `Pet` can act as a polymorphic type for pets.

**Cons:** The main disadvantage is that in `Java` a class can have no more than one immediate superclass. Languages which do support more than one immediate superclass are said to support *multiple inheritance*.

**Conclusion:** This design is ideal but impossible.

**Cause:** A decision of the `Java` language designers.

## 3.5 Why is Multiple Inheritance Not Allowed?

The last section showed that `Java` does not allow multiple inheritance. As a matter of fact there are at least two good reason for disallowing multiple inheritance. These reasons are best explained by studying the class diagram which is depicted in Figure 1.

The first reason for disallowing multiple inheritance in a class design like this is that it is very messy to allow `CDBurner` and `DVDBurner` have access to the shared variable `i`. For example, both classes may have different assumptions about the allowed values for this variable. Furthermore, the classes have no means to “communicate” with each other, which makes it impossible to enforce any invariants even if the two classes wanted to. If we allow this kind of shared access, we might just as well make all our variables `public`.

A second disadvantage is that it is not immediately clear which of the methods `burn` should be inherited if the class `ComboDrive` doesn’t override `burn( )`. Should it be the method from `CDBurner` or should it be the method from `DVDBurner`? To allow multiple inheritance one needs extra rules or notation to decide which method is inherited. Arguably this is a disadvantage because it makes it more difficult to learn the resulting language and use it without making errors.

# 4 Interfaces

A *`Java` interface* is like an abstract class. However, interfaces basically have no implementation. They can only have: abstract instance methods, and class attributes. The methods and attributes of an interface cannot be `private`. We shall see shortly that interfaces resolve the multiple inheritance problem.

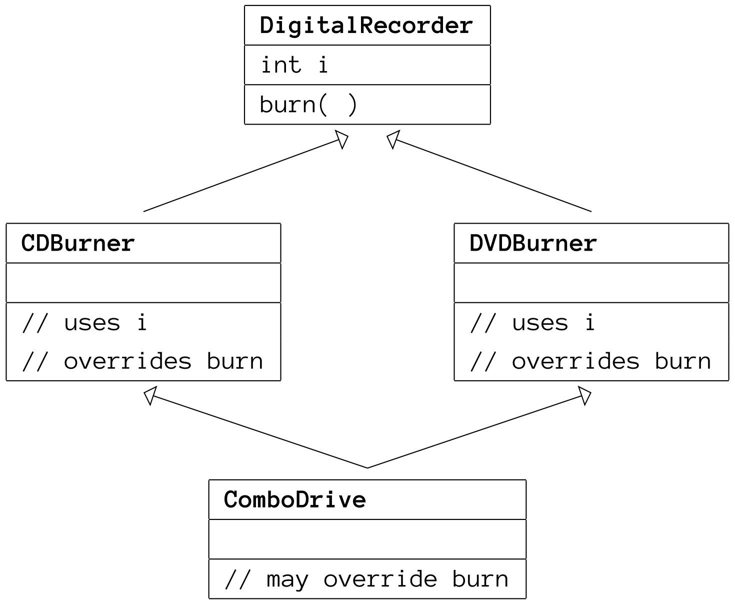

Figure 1: Deadly diamond of death (diamond problem)

## 4.1 Using Interfaces

Using interfaces is as easy as using abstract classes. For simplicity we shall only allow (abstract) methods in our interfaces.

**Creating:** You create the interface by writing the abstract interface methods in the body of the file that defines the interface.

**Implementing:** The result of *implementing* an interface is a class. It is analogous to extending an abstract class and implementing all abstract methods which are defined in the abstract class.

- You override abstract methods which are defined in the interfaces which are implemented by the class. If the class is concrete then you override *all* abstract methods.
- The new interface now acts as a "subclass" of the interface.

**Extending:** The result of *extending* an interface is another interface. It is analogous to writing an abstract class that extends an abstract class.

- You add zero or more new abstract methods.
- The new interface now acts as a "sub-interface" of the interface.

## 4.2 Creating an Interface

You create a `Java` interface just like you write an abstract `Java` class. However, this time you use the keyword `interface` instead of `class`.

```java
public interface Pet {
    public void beFriendly( );
}
```

## 4.3 Implementing an Interface

The result of *implementing* an interface is a class. The resulting class is concrete it should override all abstract methods of the interface. If it is abstract you may override any number of abstract methods.

Given the interface `Pet` from the previous example you can *implement* it by defining a class which overrides the abstract method `beFriendly` which is defined by the interface.

Let's see how this works by implementing a class `Dog` which implements the interface `Pet`. This works just as extending an (abstract) class `Pet`. When you extend a class you write `extends`. When you implement an interface you write `implements`.

```java
public class Dog implements Pet {
    @Override
    public void beFriendly( ) {
        System.out.println( "Be excited when master arrives." );
    }
}
```

## 4.4 Extending an Interface

An interface is *extended* by defining a new interface which defines zero or more new abstract methods.

Given the interface `Pet` from the previous example you can *extend* it by writing a new interface which defines one or several abstract methods.

Let's see how this works and extend the interface `Pet` by defining a new interface called `LostPet` which defines a new abstract method. Specifically, the interface defines a method `hasBeenFound` which returns `true` if the lost `Pet` has been found and a method `getPoundName` which returns the name of the pound the found `Pet` is in. This time you use the keyword `extends`.

```java
public interface LostPet extends Pet {
    public boolean hasBeenFound( );
    public String  getPoundName( );
}
```

You can extend or implement one or several interfaces. You can also combine this with the extension of one class.

```java
public class Cat extends Animal implements Pet {
    …
}
```

## 4.5 “Abstract” Template Interfaces

No doubt you’ve already noticed: interfaces don’t really allow you to share common code. Fortunately, there’s a very easy way to overcome this problem: use an abstract class that implements one or several interfaces. This technique is commonly used in the `Java` collections.

To see how this works, consider the following source code.

```java
public abstract class AbstractPet implements Pet {
    @Override
    public void beFriendly( ) {
        System.out.println( "I’m so excited." );
        System.out.println( "And I just can’t hide it." );
    }
}
```

We can now use this abstract class as the basis for the implementation of a `Cat` and `Dog` class. We simply extend the abstract class and all code is immediately inherited. Both resulting classes share the code from the abstract class. The following is an example. (For simplicity we’re not letting the classes extend the `Animal` class.)

```java
public class Dog extends AbstractPet {
    public void makeNoise( ) {
        System.out.println( "Arf. Arf." );
    }
}
```

```java
public class Cat extends AbstractPet {
    public void makeNoise( ) {
        System.out.println( "Mew. Mew." );
    }
}
```

## 4.6 Delegation

You may not have noticed it but by *extending* the abstract *class* `AbstractPet` in the `Dog` and `Cat` classes at the end of the previous section the two classes lose the right to extend other classes. For example, the classes are now no longer allowed to extend the `Animal` class. This section shows how we may extend a class while at the same time implement an interface with the default behaviour for that interface.

As with the last example we’ll use the `AbstractPet` class for the default behaviour. With some modification, you can also use this technique to “extend” several superclasses. The following demonstrates the technique.

```java
public class Dog extends Animal implements Pet {
    private final Pet gopher = new AbstractPet( );

    @Override
    public void makeNoise( ) {
        // This is Animal behaviour.
        System.out.println( "Arf. Arf." );
    }

    @Override
    public void beFriendly( ) {
        // This is Pet behaviour.
        gopher.beFriendly( );
    }
}
```

We introduce a private `Pet` reference variable `gopher` and assign it some concrete (polymorphic) `Pet` reference. In our example the concrete `Pet` reference is an `AbstractPet` reference, but in general other concrete `Pet` references may also work. By initialising `gopher` this way it references an object which has all default `Pet` behaviour.

To implement `Pet` behaviour, we simply *delegate* the work to the object which is referenced by `gopher`. In this example, we simply call `gopher.beFriendly( )` in the definition of `beFriendly`. This general technique of delegation is very useful.

Further information about abstract template classes may be found in [Bloch, 2008, Item 18].

### 4.7 What the Experts Say

Interfaces don't depend on an implementation. That's why they are more flexible to use than classes. The advantage of interfaces is well known by the experts in the field:

- Coding to an interface, rather than to an implementation, makes your code easier to extend [McLaughlin *et al.*, 2007, Page 224].
- By coding to an interface, you reduce the dependencies between different parts in your implementation ... and "loosely coupled" is a good thing [McLaughlin *et al.*, 2007, Page 282].
- Prefer interfaces to abstract classes [Bloch, 2008, Item 18].

## 5 For Wednesday

Study the lecture notes, and study Chapter 8.

## 6 Bibliography

### References

[Bloch, 2008] Joshua Bloch. *Effective* Java. Addison–Wesley, 2008.

[McLaughlin *et al.*, 2007] Brett D. McLaughlin, Gary Pollice, and David West. Head First *Object-Oriented Analysis & Design*. O'Reilly, 2007.