

# Software Development (CS2500)

## Lecture 31: Abstract Classes and Methods

M.R.C. van Dongen

January 12, 2011

### Contents

1	Outline	1
2	Abstract Classes	1
3	Abstract Methods	3
4	The Object Class	4
4.1	Overriding equals . . . . .	7
4.2	Overriding hashCode . . . . .	8
4.3	Overriding toString . . . . .	8
4.4	Extreme Polymorphism . . . . .	8
5	For Friday	9
6	Bibliography	9

### 1 Outline

Today's lecture is about abstract classes and abstract methods. As the name suggests they give us more abstraction. In addition, they give us more flexibility.

### 2 Abstract Classes

Again consider the design of our Animal class hierarchy. For ease of reference it is depicted in Figure 1. We can write `Hippo hippo = new Hippo( )` and `Cat cat = new Cat( )`. Using polymorphism we can even write `Animal cat = new Cat( )`. After all, Cat extends Feline, and Feline extends Animal, so Cat

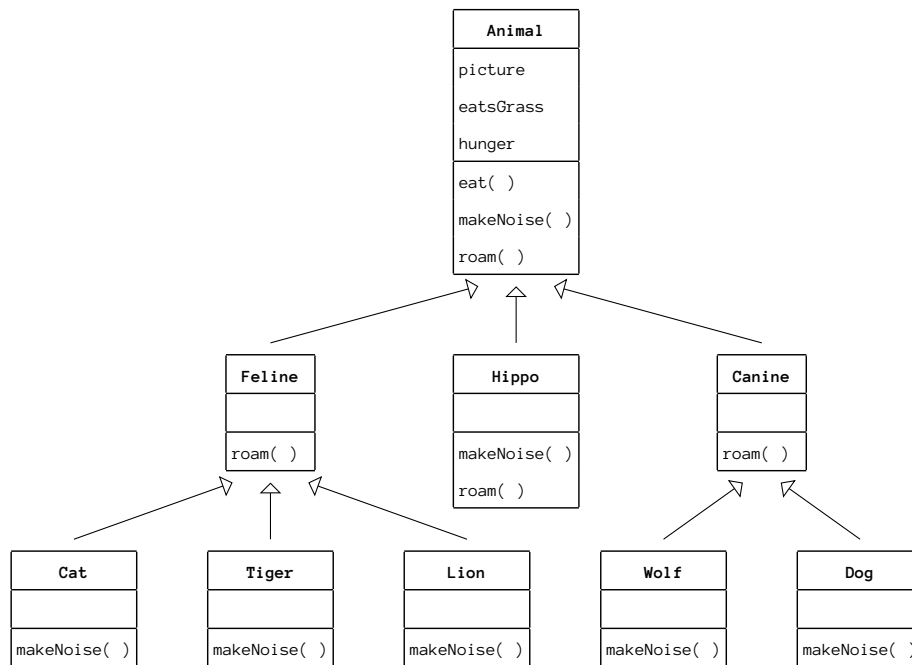


Figure 1: The Animal class hierarchy

is a subclass of Animal. Since Cat is a subclass of Animal, the substitution principle tells us that we may use a Cat where an Animal is expected: a Cat ISA Animal.

We're even allowed to write `Animal animal = new Animal( )`, but what kind of object is that supposed to give us? Arguably `Animal animal = new Animal( )` does not make any sense.

The Animal class serves only two purposes:

- We need it for inheritance, so we can share common code, and define a common protocol for Animals.
- We need it for polymorphism, so we can write code that will still work if we add subclasses.

We never intended the Animal class to be instantiated. We want Cat and Dog objects, but not Animal objects. Fortunately, there's a magic spell which prevents a class from ever being instantiated.

```
public abstract class Animal {
    ...
}
```

The keyword `abstract` prevents a class from being instantiated. You write it before the word `class` and the name of the class in the class definition. When a class "becomes" abstract `javac` won't let you instantiate it. So if Animal is abstract the following gives you a compile-time error.

```
Animal animal = new Animal( );
```

Needless to say, subclasses can be abstract too.

```
public abstract class Canine extends Animal {  
    ...  
}
```

A class is called *abstract* if it's defined with the keyword `abstract`. A class is called *concrete* if it is not abstract.

You can still use abstract type reference variables. This is useful for polymorphism.

```
Dog dog = new Dog( );  
Cat cat = new Cat( );  
Animal animal = dog;  
animal = cat;
```

But, you can only instantiate concrete classes.

```
Cat cat = new Cat( );  
Animal dog = new Dog( );
```

Instantiating an *array* of a concrete base class type is also allowed.

```
Animal[] animals = new Animal[ 3 ];
```

This is allowed because the result is a reference to an Array object.

### 3 Abstract Methods

Java also has *abstract methods*. Abstract methods are defined in abstract classes, they are defined with the keyword `abstract`, and they have no body. The following is an example.

```
public abstract void roam( );
```

As you can see from the example, a semi-colon is written where you usually have a body.

Abstract classes must be *extended*. Abstract methods must be *overridden*. Abstract methods still let you define (part of) a common protocol. The advantage is that they let you do this without having to implement default behaviour.

Abstract methods have no body. They only occur in abstract classes. They have no default behaviour. Still, subclasses need abstract method behaviour as part of a common protocol. Therefore, you have to *implement* the abstract method. Here, *implementing the abstract method* means providing the method's body. Every abstract method should be implemented somewhere "along" every path leading from the abstract class to the concrete subclasses. This may mean implementing the abstract method in an abstract subclass. Of course, a method may be overridden, and overridden, .... All abstract methods must be implemented somewhere.

You implement a method just like you override the method. So, if you have an abstract method `makeNoise` as follows.

```
public abstract class Animal {  
    public abstract void makeNoise( );  
}
```

Java

Then you implement it as follows:

```
public class Dog extends Animal {  
    @Override  
    public void makeNoise( ) { ... }  
}
```

Java

Each abstract method should be implemented somewhere along each path leading from the abstract class, which defines the method, to some concrete subclass. To see how this works, again consider Figure 1. Let's assume the methods `roam( )` and `makeNoise( )` are abstract methods in the abstract class `Animal`. Furthermore, let's assume the `Feline` and `Canine` classes are abstract. There are six concrete classes: `Cat`, `Tiger`, `Lion`, `Hippo`, `Dog`, and `Wolf`. For each concrete class, the two abstract methods should be implemented along the path from the abstract class to the concrete class.

- Let's first look at the method `makeNoise( )`. In our original design we implemented the method in the concrete classes. Let's do the same here. Clearly, the method is implemented along each path leading from the abstract class to some concrete class.
- Next let's look at the method `roam( )`. In our original design we implemented this method in the concrete `Hippo` class and the classes `Feline` and `Canine`, which are now abstract. Again let's do the same here. With this choice, which happens to be different, the method is also implemented along any path leading from the `Animal` class to some concrete class.

## 4 The Object Class

Let's assume we want to implement our own `ArrayList` class for storing `Dog` object references. For simplicity we'll store at most 5 `Dog` references. The following is a possible implementation.

```

public class DogList {
    private final Dog[] dogs = new Dog[ 5 ];
    private int size = 0;

    public void add( Dog dog ) {
        if (size < dogs.length) {
            dogs[ size ++ ] = dog;
        }
    }

    public Dog get( int index ) {
        return ((0 <= index) && (index < size))
            ? dogs[ index ] : null;
    }
}

```

The class works fine, but it is not very flexible: we can only add and get Dog object references. If we want to store Cat object references as well then we need to exploit polymorphism and implement the array as an array of a common supertype of Cat and Dog. The Animal class seems exactly what we're looking for.

```

public class AnimalList {
    private final Animal[] animals = new Animal[ 5 ];
    private int size = 0;

    public void add( Animal animal ) {
        if (size < animals.length) {
            animals[ size ++ ] = animal;
        }
    }

    public Animal get( int index ) {
        return ((0 <= index) && (index < size))
            ? animals[ index ] : null;
    }
}

```

Using our Animal class we can now write the following.

```

public class AnimalApplication {
    public static void main( String[] args ) {
        AnimalList list = new AnimalList( );
        list.add( new Dog( ) );
        list.add( new Cat( ) );
        list.add( new Dog( ) );
        list.get( 0 ).makeNoise( ); // Arf. Arf.
        list.get( 1 ).makeNoise( ); // Mew. Mew.
        list.get( 2 ).makeNoise( ); // Arf. Arf.
    }
}

```

Wow, this class works great, but again it is not very flexible: this time we can only add and get `Animal`s object references (including object references from subclasses of the `Animal` class. You get the drift. If we need a class to store and retrieve *any* object reference we need a class which is at the top of the object class hierarchy. The class we are looking for is the `Object` class.

Every class in Java extends the `Object` class. If we use an `ArrayList`, then we can store/retrieve any kind of object. The following are some of the instance methods which are provided by the `ArrayList` class.

- `boolean remove( Object elem )`
- `int indexOf( Object elem )`
- `Object get( int index )`
- `boolean add( Object elem )`

Many of the types of `ArrayList` methods use the ultimately polymorphic type `Object`. This gives us ultimate flexibility. (But see further on.)

We've already studied some of this, but it's good to have another look at what's defined in the `Object` class. The following are three important methods.

**`int hashCode( )`:** Returns the object's hashCode. An object's hashCode is a number which is associated with the object. An `Object` instance's hashCode is guaranteed to be unique, but it is possible to override the `hashCode` method and change this. Hashcodes have many important applications. More about that in another lecture.

**`boolean equals( Object that )`:** Returns true if and only if the object is equal to that.

**`String toString( )`:** Returns a `String` representation of the object.

It is strongly recommended that you override these methods.

## 4.1 Overriding equals

Overriding `equals` makes it easier to compare objects. It allows you to perform a proper *deep* object comparison.

**Shallow comparison:** A *shallow* comparison is for comparing object identity (by object reference value). Shallow comparison is carried out using `==`. Shallow comparison only returns `true` if two object references reference the same object.

**Deep comparison:** A *deep* comparison is for comparing objects by the values of their relevant attributes. A deep comparison is carried out with `equals`. A deep comparison only returns `true` if two object references reference two objects which represent the same “notion”.

Shallow comparisons are quick, whereas deep comparisons take more time.

The following may explain why it may be useful to implement `equals` in a different way than the value which is returned by this `==` that. After all, for most applications two numbers are the same if they have the same value.

```
public class IntModTwo {
    private boolean isEven;
    public Number( boolean even ) {
        isEven = even;
    }
    public void increment( ) {
        isEven = !isEven;
    }
    @Override
    public boolean equals( Object that ) {
        return (that instanceof IntModTwo)
            && this.isEven == that.isEven;
    }
    @Override
    public String toString( ) {
        return (isEven ? "0" : "1");
    }
}
```

There is one new ingredient in this example: `instanceof`. As you may have already guessed from the name, `A instanceof B` is true if and only if the instance `A` is an instance of class `B`. So, `Dog instanceof Animal` and `Animal instanceof Animal` are true, but `Animal instanceof Dog` is not. However, if `A` is equal to `null` then `A instanceof B` will always give you false.

You should only override `equals` if it makes sense. Specifically, you should always obey the *contract* for `equals` [Bloch, 2008, Item 8]. The contract for `equals` is that it should define an equivalence class. This means that the following should be true for all object references `o1`, `o2`, and `o3` such that `o1 != null` and `o2 != null`:

**Reflexivity:** `o1.equals( o1 )`.

**Symmetry:** If `o1.equals( o2 )` then `o2.equals( o1 )`.

**Transitivity:** If `o1.equals( o2 )` and `o2.equals( o3 )` then `o1.equals( o3 )`.

**Consistency:** `o1.equals( o3 )` should always return the same. (Unless the state of the referenced objects changes.)

**“Sense”:** `o1.equals( null )` should return `false`.

Many classes rely on this contract. If you fail to implement it correctly your application may fail.

## 4.2 Overriding hashCode

Always override `hashCode` when you override `equals` [Bloch, 2008, Item 9]. The following is the general contract:

- The general rule is that if the state for the computation of `equals` does not change then the `hashCode` also shouldn't change.
- Two object references which are deeply equal should have the same `hashCode`.
- Note that it is allowed to have two object references with the same `hashCode` which are not deeply equal.

In short you should only let the `hashCode` depend on the attributes which are needed for the computation of `equals`.

## 4.3 Overriding toString

Always override `toString` [Bloch, 2008, Item 10]. A good overriding implementation of `toString` makes your class a nicer one. Include *all* interesting information in the result. The result should be self-explanatory.

## 4.4 Extreme Polymorphism

There is a tradeoff between making your class as polymorphic as possible and the safety you get from the type system. Using polymorphic reference variables should be easy and safe. Avoid using polymorphic `Object` reference variables if you can. Using them is not always easy. Using them is not always safe. Instead, try using polymorphic reference variables from as specific a class as possible. Using them is easier. Using them is safer.

The following shows why using polymorphic non-`Object` reference variables is usually easier. In the example which follows shortly, we add things to and get things from a list of type `ArrayList<Object>`. With this list we can add object references of any type. However, when we want to get object references from the list it gets complicated.



In the example, our first complication is that we need to cast the `Object` reference to a `Dog` reference as the compiler isn't aware of the types of the actual instances which are referenced by the things in the list. This makes using the list awkward.

Our second complication is that we may write programs which are fine at compile time but which fail at runtime. For example, the call `list.get( 2 )` returns is a `House` reference. When the program is run, casting the `House` reference to a `Dog` reference will cause the program to halt with an exception.

```
Dog dog1 = new Dog( );
Object dog2 = new Dog( );
House house = new House( );
ArrayList dogList = new ArrayList( );
dogList.add( dog1 );           // Grand: Dog is subtype of Object.
dogList.add( dog2 );           // Grand: This is an Object reference.
dogList.add( house );          // Grand: House is subtype of Object.
dog1 = (Dog)dogList.get( 0 );  // We need the cast.
dog1.makeNoise( );            // Arf. Arf.
dog1 = (Dog)dogList.get( 1 );  // We need the cast.
dog1.makeNoise( );            // Arf. Arf.
dog1 = (Dog)dogList.get( 2 );  // Compiler allows it but runtime exception.
dog1.makeNoise( );            // We don't even get here.
```

Don't Try this at Home

If we use an `ArrayList<Dog>` for our list then several things change. First we're not allowed to add the `Object` and `House` object references to the list. (The compiler doesn't know that `dog2` is actually referencing a `Dog` object at the moment of the call `dogList.add( dog2 )`. All it looks at is the type of the reference variable.) Second, we no longer need casts to get object references from the list. Finally, we can be sure that whatever comes from the list is a `Dog` reference. In effect using a restricted type for our polymorphic reference variable made it easier to write this version of the program. In addition it allowed the compiler to disallow certain kinds of statements which were not safe.

```
Dog dog1 = new Dog( );
Object dog2 = new Dog( );
House house = new House( );
ArrayList<Dog> dogList = new ArrayList<Dog>( );
dogList.add( dog1 );           // Grand: Dog is subtype of Dog.
// dogList.add( dog2 );        // Not allowed: Object doesn't extend Dog
// dogList.add( house );       // Not allowed: House doesn't extend Dog
dog1 = dogList.get( 0 );       // We no longer need the cast.
dog1.makeNoise( );            // Arf. Arf.
```

Don't Try this at Home

## 5 For Friday

Study the lecture notes, and study Pages 197–216.

## 6 Bibliography

### References

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