Lecture 4: Inheritance and Abstract Classes

Contents

1 Abstracting over Common Methods in Different Classes

2 Inheritance

2.1 Simplifying the Boa/Dillo Classes

2.1.1 Fields and Constructor

2.1.2 The isLenWithin Helper

3 Abstract Classes

4 Abstract/Super Classes versus Interfaces

4.1 Example: Adding FruitFlies

5 Class Extension without Abstraction

6 Summary

Motivating Question

How can we share common code and fields across classes?

Objectives

By the end of this lecture, you will know:

- How to share common code across classes
- What inheritance is in object-oriented programming
- What super classes and abstract classes are in Java
- The difference between abstract classes and interfaces

By the end of this lecture, you will be able to:

- Share common code across classes
- Create class hierarchies
1 Abstracting over Common Methods in Different Classes

Recall our `isNormalSize` methods on animals:

```java
// in the Boa class
public boolean isNormalSize () {
    return 5 <= this.length && this.length <= 10;
}

// in the Dillo class
public boolean isNormalSize () {
    return 2 <= this.length && this.length <= 3;
}
```

The method bodies on `Boa` and `Dillo` differ only in the numbers for the low and high bounds. We know that we should create helper functions to share common code in cases such as this. How do we do this in Java?

The first question is what helper function we should create. There are two main options:

```java
public boolean isNormalSize(int low, int high) { ... }
public boolean isLenWithin(int low, int high) { ... }
```

While these functions have the same signature, they signal different expectations about the code. The first suggests that the `isNormalSize` computation will be pretty much the same across all animals, while the second suggests that the common computation is about the length being within bounds, and that there could be other conditions to consider in determining whether an animal is normal size (such as its height, wing span, etc).

Since there could be many other kinds of animals, we’ll go with option 2.

2 Inheritance

Fortunately, Java (and all other object-oriented languages) follows a model of `class hierarchies` in which one class can build upon (or extend) the definitions in another. We can define a class for the shared information between `Boa` and `Dillo` and make `isLenWithin` a method in that class. Since our new class exists to abstract over animals, we name the class `AbsAnimal`.

We initially populate `AbsAnimal` with the code that is common to `Boa` and `Dillo`. We have already noted that the `isNormalSize` method is (largely) common, and that we want to have a common helper called `isLenWithin`. Looking at the two classes, however, we see that the `length` field is also common. This suggests that `AbsAnimal` needs the following contents:

```java
public class AbsAnimal {
    int length;

    // the constructor
    public AbsAnimal (int length) {
        this.length = length;
    }
```
Next, we need a way to say that Boa and Dillo should extend on what is already defined in AbsAnimal. In standard OO (object-oriented) terminology, Boa and Dillo should inherit from AbsAnimal. We indicate inheritance using a new keyword called extends in the class definition:

```java
class Dillo extends AbsAnimal implements IAnimal {
    ...
}

class Boa extends AbsAnimal implements IAnimal {
    ...
}
```

In OO terminology, AbsAnimal is the superclass of each of Boa and Dillo. Each of Boa and Dillo is a subclass of AbsAnimal.

### 2.1 Simplifying the Boa/Dillo Classes

As a result of using extends, every field and method in AbsAnimal is now part of Boa and Dillo. This means that we can remove some code from each of these classes. We'll work just with Boa in these notes (the changes to Dillo are similar).

#### 2.1.1 Fields and Constructor

The first thing to note is that we can remove the length field from Boa. Furthermore, the AbsAnimal constructor, not the Boa constructor, should set the value of the length field. The Boa constructor passes the value for length to the AbsAnimal constructor by calling super(length). In general, super refers to the superclass; here, it is the name of the constructor in the superclass. This gives the following code:

```java
public class Boa extends AbsAnimal implements IAnimal{
    String name;
    String eats;

    public Boa (String name, int length, String eats) {
        super(length);
        this.name = name;
        this.eats = eats;
    }
    ...
}
```
In Java, each class may extend at most one other class, so the meaning of `super` is unambiguous. Note that the call to `super` must be the first line in the `Boa` constructor (this is again one of the rules of Java).

2.1.2 The `isLenWithin` Helper

Finally, we need to clean up `Boa` to use the `isLenWithin` method that is in `AbsAnimal`.

```java
public class Boa extends AbsAnimal implements IAnimal {
    String name;
    String eats;

    public Boa (String name, int length, String eats) {
        super(length);
        this.name = name;
        this.eats = eats;
    }

    /**
     * check whether boa's length is considered normal
     */
    public boolean isNormalSize() {
        return super.isLenWithin(30, 60);
    }
}
```

We now have code that reflects the code sharing that you learned to do in CS17. We’ve also left ourselves the ability to tailor `isNormalSize` differently for different animals, while still sharing the common length computation.

3 Abstract Classes

One last detail: our current code allows someone to write `new AbsAnimal(6)`. This would mean "some animal of length 6". If our goal were only to create instances of specific animals (which seems reasonable), this object wouldn’t make much sense. We therefore want to prevent someone from creating `AbsAnimal` objects (allowing only `Boa` and `Dillo` objects).

An abstract class is a class that can be extended but not instantiated as a standalone object. We specify this through the keyword `abstract` when defining a class:

```java
public abstract class AbsAnimal {
    int length;

    ...
}
```

Now, `new AbsAnimal(6)` yields an error.
4 Abstract/Super Classes versus Interfaces

We now have two mechanisms, abstract/super classes and interfaces, through which classes can share information. What is each one best used for?

- **Interfaces** specify new types. If you need a type name, create an interface.

- **Super classes** capture common code. If you have common or shared code, create a super class.

- **Abstract classes** exist ONLY to share common code, but not to create data. If you have a class that should not be used to create new objects, mark it as abstract.

Those with prior Java experience may have learned to use abstract classes for both creating types and sharing code, but this is not good OO programming. Interfaces are fairly permissive: a class must provide methods that implement those outlined in the interface, but how that implementation works (including what data structures get used) is entirely up to the author of the class. Abstract classes provide actual code, which means that any class which uses an abstract class must do so in a way that is consistent with the existing code or data structures. The restriction that a class may only ever extend one other class reflects this consistency problem. In contrast, a class can implement any number of interfaces, because interfaces do not constrain the implementation of behavior.

Interfaces also recognize that the world is not neatly hierarchical. Different kinds of real world objects have all sorts of different properties that affect how we use them. For example, within Brown’s information system, I am each of a faculty member, first-year advisor, and a director of undergraduate studies. Different tasks within the university view me as having these different roles. Interfaces let a program say "I need an object that has the methods associated with this particular role”. Class hierarchies can’t do this (because of the single-extension restriction).

Put differently, there is an important distinction between stating which operations are required and stating how those operations are implemented (we’ll see a lot of this in the coming days). The former is called specification; the latter implementation. Interfaces are for specification; abstract classes for implementation.

4.1 Example: Adding FruitFlies

What if I wanted to expand my zoo to include Fruit Flies? Fruit flies are so small that we don’t worry about tracking their length – they will always be considered normal size. If I make a Fruit fly class extend AbsAnimal, then I end up tracking a field (length) that I don’t need. But if I have that class extend the IAnimal interface, I can still put FruitFlies in my zoo. Specifically:

```java
public class FruitFly implements IAnimal {
    public FruitFly() {}

    public boolean isNormalSize() {
        return true;
    }
}
```
5 Class Extension without Abstraction

We didn’t cover this section in class, but it could serve as a useful exercise for review.

Abstract classes support field and method abstraction, but class extensions are also used to express hierarchy among data. For example, let’s add two kinds of animals to our class hierarchy: Fish, which have a length and an optimal saline level for water in their tanks; and Sharks, which are fish for which we record the number of times they attacked people. The new classes appear as follows:

```java
public class Fish extends AbsAnimal {
    double salinity;

    public Fish (int length, double salinity) {
        super(length);
        this.salinity = salinity;
    }

    /**
     * check whether fish's length is considered normal
     */
    public boolean isNormalSize () {
        return isLenWithin(3, 15);
    }
}

public class Shark extends Fish {
    int attacks;

    public Shark (int length, int attacks) {
        super(length, 3.75);
        this.attacks = attacks;
    }
}
```

A few things to note here:

- The salinity data has type `double`; this is a common type to use for real numbers.
- `Shark` extends `Fish`, but `Fish` is not an abstract class. It still makes sense to create `Fish` that are not also `Sharks`.
- Constructors do not need to take all of the initial field data as parameters. For example, if we assume that all sharks have the same saline level, then the `Shark` constructor asks for only the length and number of attacks; it provides the fixed saline level to the `Fish` constructor on the call to `super`.
- `Shark` does not need to define `isLenWithin`, since it inherits the definition from `Fish`. If you wanted `Shark` to have its own definition (since it might have a different normal size), you could provide one in the `Shark` class. Java calls the most specific method for each object.
6 Summary

This lecture introduced the following concepts:

- Classes can be organized into hierarchies. Subclasses inherit data and methods from their superclasses.

- Parameters to otherwise common methods are passed as constructor arguments to superclasses.

- Abstract classes enable sharing but not instantiation (i.e., you can’t use the `new` keyword to make an object of an abstract class).

- A class can have at most one superclass. The constructor for a subclass should call `super` to initialize the superclass.

- Class hierarchies are only used for capturing shared code (implementation). Shared requirements belong in interfaces. Unless your code relies on a common implementation across classes, use an interface.

Please let us know if you find any mistakes, inconsistencies, or confusing language in this or any other CS18 document by filling out the anonymous feedback form: [https://cs.brown.edu/courses/cs018/feedback](https://cs.brown.edu/courses/cs018/feedback)