CS200: Migrating to Java – Inheritence and Abstract Classes

Kathi Fisler

January 30, 2026

Motivating Question

How can we share common code and fields across classes?

1 Abstracting over Common Methods in Different Classes

Recall our `isNormalSize` methods on animals:

```
1  // in the Boa class
2  public boolean isNormalSize () {
3    return 30 <= this.length && this.length <= 60;
4  }
5
6  // in the Dillo class
7  public boolean isNormalSize () {
8    return 12 <= this.length && this.length <= 24;
9  }
```

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:

```
1 public boolean isNormalSize(int low, int high) { ... }
2
3 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 with 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 Inheritence

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 capture information related to animal size, we name the class `SizedAnimal`.

We initially populate `SizedAnimal` 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 `SizedAnimal` needs the following contents:

```
1  public class SizedAnimal {
2    int length;
3
4    // the constructor
5    public SizedAnimal (int length) {
6      this.length = length;
7    }
8
9    /**
10     * Helper to determine whether length is within bounds
11     */
12   public boolean isLenWithin(int low, int high) {
13     return low <= this.length && this.length <= high;
14   }
15 }
```

Next, we need a way to say that `Boa` and `Dillo` should extend on what is already defined in `SizedAnimal`. In standard OO (object-oriented) terminology, `Boa` and `Dillo` should **inherit** from `SizedAnimal`. We indicate inheritence using a new keyword called **extends** in the class definition:

```
1  class Dillo extends SizedAnimal implements IAnimal {
2    ...
3  }
4
5  class Boa extends SizedAnimal implements IAnimal {
6    ...
7  }
```

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

2.1 Simplifying the Boa/Dillo Classes

As a result of using **`extends`**, every field and method in `SizedAnimal` 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 `SizedAnimal` constructor, not the `Boa` constructor, should set the value of the `length` field. The `Boa` constructor passes the value for `length` to the `SizedAnimal` 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:

```
1  public class Boa extends SizedAnimal implements IAnimal{
2    String name;
3    String eats;
4
5    public Boa (String name, int length, String eats) {
6      super(length);
7      this.name = name;
8      this.eats = eats;
9    }
10   ...
11 }
```

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

Now, we can clean up `Boa` to use the `isLenWithin` method that is in `SizedAnimal`.

```
1  public class Boa extends SizedAnimal implements IAnimal {
2    String name;
3    String eats;
4
5    public Boa (String name, int length, String eats) {
6      super(length);
7      this.name = name;
8      this.eats = eats;
9    }
10
11   /**
12    * check whether boa's length is considered normal
13    */
14   public boolean isNormalSize() {
15     return super.isLenWithin(30, 60);
16   }
17 }
```

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.

2.2 Sharing the `implements` annotation

You might have noticed that both `Boa` and `Dillo` implement the `IAnimal` interface. Since that is also common, we can move that annotation up to `SizedAnimal` as well:

```
1    public class SizedAnimal implements IAnimal {
2      int length;
3      ...
4   }
5
6  class Boa extends SizedAnimal {
7    ...
8  }
```

This version of the code runs just as the previous version did. The **`implements`** annotation on `SizedAnimal` must be satisfied by every one of its subclasses.

3 Abstract Classes

One last detail: our current code allows someone to write **`new`** `SizedAnimal(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 `SizedAnimal` 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:

```
1  public abstract class SizedAnimal {
2    int length;
3
4    ...
5  }
```

Now, **new** `SizedAnimal(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, which in turn capture shared *behavior*. If you just need a type name that spans multiple classes, create an interface.
- *Super classes* capture common *structure* (and corresponding code about that structure). If you have common or shared fields with corresponding methods, create a super class.
- *Abstract classes* exist ONLY to share common fields/methods, 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 fields, 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.

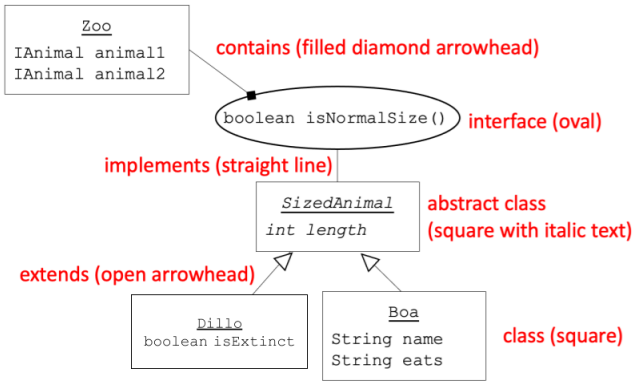
Interfaces also recognize that the world is not neatly hierarchical. Different kinds of real world objects have all sorts of different properties that affect how we use them. For example, within Brown's information system, I am each of a faculty member, first-year advisor, and a director of undergraduate 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.

5 Drawing Relationships Among Classes

Sometimes, it helps for us to create diagrams showing the relationships among classes and interfaces. This is particularly useful when we have many classes over many files, since a single drawing can summarize information that is otherwise split across many `.java` files.

Here is a diagram showing the relationships among our classes so far:



The rectangles are classes and the ovals are interfaces. The different lines and arrowheads that connect classes and interfaces show three different relationships (so far):

- plain lines connect classes and interfaces
- lines with open-headed arrows points from classes to their superclasses
- lines with solid diamondheads indicate that one class has multiple fields with the type of the other

We will work with these diagrams through the course. This is your warmup with them.

5.1 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 SizedAnimal, 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:

```
1 public class FruitFly implements IAnimal {
2  public FruitFly() {}
3
4  public boolean isNormalSize() {
5     return true;
6  }
7 }
```

6 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:

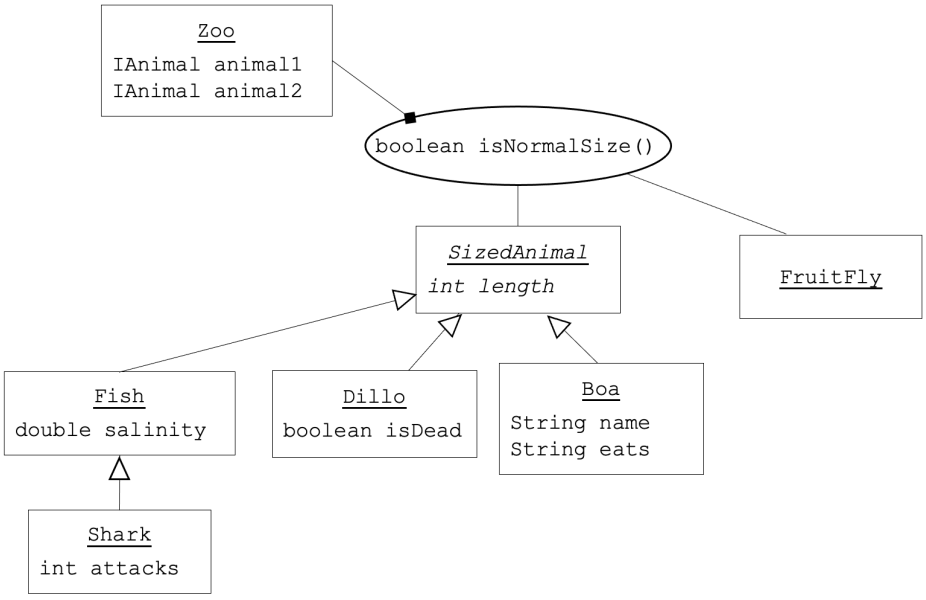
```
1 public class Fish extends SizedAnimal {
2  double salinity;
3
4  public Fish (int length, double salinity) {
5    super(length);
```

```
6     this.salinity = salinity;
7   }
8
9   /**
10   * check whether fish's length is considered normal
11   */
12  public boolean isNormalSize () {
13    return isLenWithin(3, 15);
14  }
15 }
16
17 public class Shark extends Fish {
18  int attacks;
19
20  public Shark (int length, int attacks) {
21    super(length, 3.75);
22    this.attacks = attacks;
23  }
24 }
```

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 `Shark`s.
- 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.

Here is a diagram showing the classes and relationships that we have added:



7 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.