CSC142 Supplemental Reading: Week09

Contents

1 Abstract Classes

An **abstract class** is a special kind of class that differs from a normal class in just two ways:

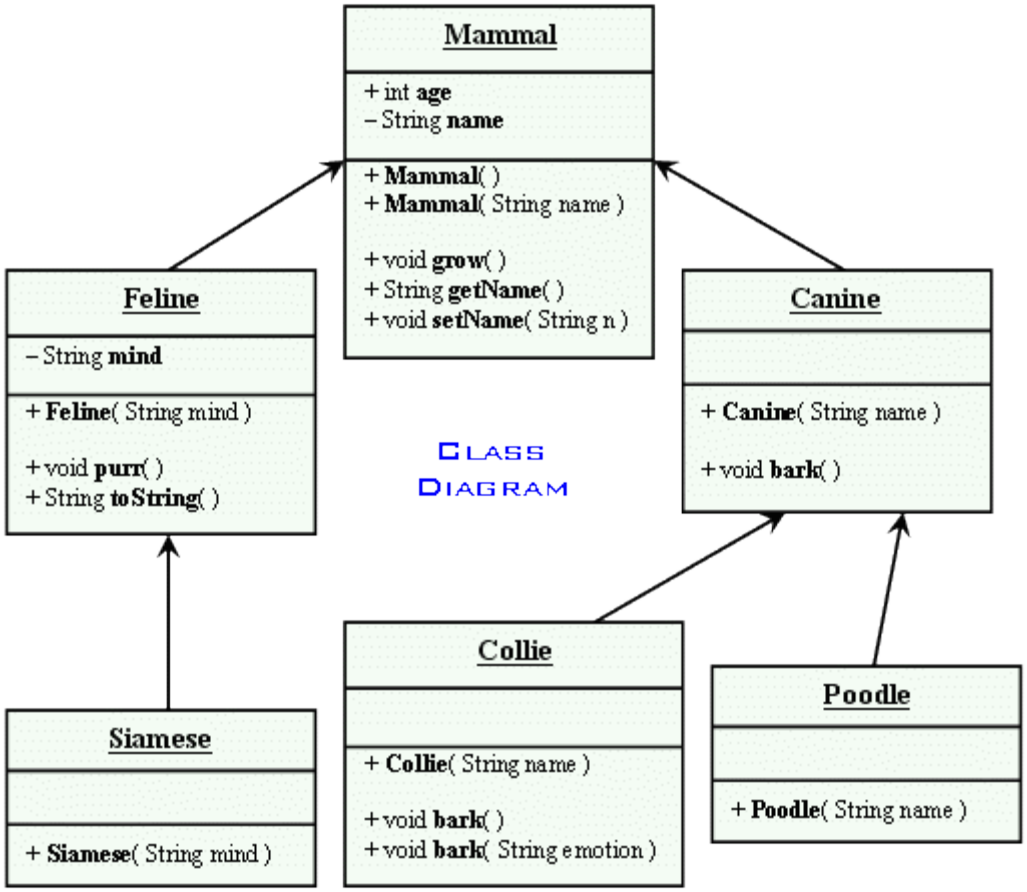
- **An abstract class cannot be instantiated.**
- **An abstract class may contain abstract methods.**

To emphasize these differences, a normal class (the only kind we have seen up until now) is sometimes referred to as a **concrete class**. (The word 'concrete' seems like a reasonable conceptual opposite of 'abstract'.)

Those are the only two differences -- everything else is the same: an abstract class may contain instance variables, instance methods, class variables, class methods -- even constructors. However, the differences are very important. For example, what is the point of having a class that has any instance members if it cannot be instantiated? Why would such a class need a constructor anyway? The answer:

- **There is only one reason to have an abstract class -- so it can be extended.**

An abstract class cannot be instantiated, but all of its capabilities will be **inherited** by any concrete class that extends -- and specializes -- the abstract class. Sometimes this organization makes sense in a complex system of classes. Consider the 'Mammals' example used in a previous note. Here's the original class diagram again for reference:



In this system, things like collies and poodles are quite 'concrete' -- you can picture what a collie looks like. But what about a mammal? A mammal is more of an abstract concept -- a category or **classification** that includes some features that all mammals have in common. For example, mammals have warm blood, etc. In the same way that the classification 'mammal' represents things that all mammals have in common, the class Mammal can specify and implement things that all its subclasses have in common. So, Collie objects and Siamese objects can both understand grow() and getName() messages. If we were to change the class definition for class Mammal from

```
public class Mammal { ... }
```

to

```
public abstract class Mammal { ... }
```

nothing whatsoever would change in this project except that we would no longer be able to instantiate class Mammal -- which we probably never have wanted to do in any case.

Realize this distinction: even though we cannot instantiate an abstract class, we ***can*** have reference variables whose static type is that abstract class. So polymorphism and dynamic dispatch will still work.

1.1 Abstract Methods

But why bother with abstract classes? Why take the trouble (well, it wasn't so much trouble, was it?) to declare the class abstract? Well, one minor benefit is that the compiler will prevent it from being instantiated -- which might prevent a logic error somewhere down the road. However, the real benefit comes with the extra capability that abstract classes have: the ability to declare abstract methods.

An **abstract method** is simply **a method signature declaration without an implementation** -- no body. Here's an example:

```
public abstract void grow();
```

Notice: no curly braces -- so no method body -- just a semicolon after the parentheses. The keyword **abstract** is used to declare that this is an abstract method. If this method were written in class Mammal, then that class would be **required** to be declared abstract. Any class that contains an abstract method must itself be declared abstract.

If this declaration is placed in class Mammal, then any class that extends Mammal must implement this method. In a sense, class Mammal is saying "If you want to be a Mammal, you need to be able to grow, and I'm not going to tell you how to do that -- you'll need to implement that yourself." **The presence of an abstract method in a superclass places an obligation on every concrete subclass to implement that method (to override the abstract declaration).**

Because every concrete subclass of Mammal will have a **grow()** method, and because we can use Mammal as a static type that also recognizes the **grow()** method, some interesting things are possible. For example, we might want to maintain a collection of different kinds of mammals and tell them all to 'grow' as some point, like this:

```
Mammal[ ] animals = new Mammal[100];
... // fill the array with different objects that conform to type Mammal
for ( int i = 0; i < animals.length; i++ )
      animal[i].grow(); // make each Mammal grow
```

Notice that we do not need any type casts to call the **grow()** method -- type Mammal recognizes the **grow()** method because of the method signature in the Mammal class.

An abstract method doesn't need to be void, and it can have parameters. In fact, any method signature is fine. Here's another example:

```
public abstract double calculateSomething( int param1, double param2 );
```

To summarize:

1. An abstract method uses the keyword abstract and contains no body, no implementation. A semicolon is used after the parameter list.
2. A subclass must override an inherited abstract method, otherwise the subclass will be abstract as well.
3. By having the abstract method signature in the super class, dynamic dispatch works properly.

2 Java Interfaces

[Let's begin this note with some terminology clarification. We have used the word 'interface' in this course already -- when referring to the 'public interface' of a class (those members that a class exposes to client code by declaring them *public*). That is one use of the word 'interface' in the context of Java and OOP, but that is not what this note is about.]

In the tutorial on Interfaces, an interface is defined as "*a reference type, similar to a class, that can contain only constants, method signatures, and nested types.*" One important point to recognize in that statement: an interface creates a new reference type. Here's an example:

```
public interface Petable
{
   public void petMe();
}
```

That's it -- and because it's public, this needs to be in a separate file called Petable.java. An interface is a top-level structure -- like a class. (Method and variable declarations, on the other hand, always need to be defined inside another structure -- inside a class, for example.)

Notice the definition of the method **petMe** -- just a method signature followed by a semicolon (identical to the syntax for an abstract method). If the method needed parameters or a return type, those would also need to be specified, but a method body (i.e. an 'implementation') is not allowed.

Similar to abstract classes, interfaces cannot be instantiated. But we can declare reference variables of their type.

The example interface definition gives us the ability to declare a variable with static type **Petable** and call the **petMe** method, like this:

```
Petable pet = [ well, it's not yet clear what object we could put here... ];
pet.petMe();
```

There is a piece missing in the code above -- and indeed, an interface is of no value until some class **implements** it. In fact, ***implements*** is the keyword in Java that we use. Here's an example:

```
public class Siamese extends Feline implements Petable {

   public Siamese( String m ) {
      super( m );
   }
   public void petMe() {
      purr();
   }
}
```

When a class implements an interface, two things happen:

- The class conforms to the data type created by the interface (**Siamese "Is-A" Petable**).
- The class must override all methods specified in the interface and provide implementations for them.

If Siamese failed to properly override the **petMe** method, a syntax error would result. With this code properly compiled, we can now finish our previous program:

```
Petable pet = new Siamese( "Fluffy" );
pet.petMe();
```

Why Bother?

Why not just make pet type Siamese? Well, for this short bit of code, that would work fine. Interfaces are generally used as a way to organize large hierarchies of classes when there is a desire to ***separate*** the **abstract idea** of something from the **actual implementation**. For example, there is an interface named **List** in the Java API that specifies many of the behaviors of an ArrayList. Class ArrayList implements interface List, and so do several other classes. In a situation where only the 'listness' of the object matters, the data type of the variable might be specified as List, allowing any kind of list to be used.

Another major organizational benefit of interfaces is this: while every class extends exactly one other class (a rigidly hierarchical organization), **a class may implement any number of interfaces**. So, we can create conformance paths that run through a large hierarchy of classes and interfaces in whatever way best describes the relationships we want them to have.

Here's a more concrete example. Let's have class **Canine** implement the **Petable** interface (and also the interface **Tagged** -- just to illustrate the syntax for implementing multiple interfaces):

```
public class Collie extends Canine implements Petable, Tagged { ... }
```

Now we have a classification -- Petable -- that applies to at least two different classes. We could use this data type as the basis for a collection, like this:

```
ArrayList<Petable> pets = new ArrayList<Petable>();
pets.add( new Siamese( "Fluffy" ) );
pets.add( new Collie( "Lassie" ) );
... // add more
for ( int i = 0; i < pets.size(); i++ ) {
   Petable pet = pets.get( i );
   pet.petMe();
}
```

Here's a key point. You could argue that Siamese and Collie also both conform to type Mammal (or even Object), so why not use that type in the collection. The problem is that not all mammals are petable. This is a classification we are assigning very deliberately. So, Siamese and Collie objects are Petable, but Tiger and Wolf objects (if we create them) are not. We can gather together objects that share something in common and use that shared behavior very conveniently when an interface is used to specify those relationships.

A good real-world analogy for a Java interface is a license -- like a driving license. You don't really 'inherit' anything from a driving license except **obligations**. For example, to get the license, you need to pass a test -- you may need to demonstrate a left-turn hand signal, for example. Think of that as a method you need to implement -- a behavior you can exhibit. Some people have a driving license, some don't. Some people hold more than one license -- maybe a pilot's license or a CPR certification. Some classes implement the Petable interface, some don't. Some classes (like Collie above) implement more than one interface.

Interfaces are most valuable as a tool for organizing a large group of classes. However, even if you don't plan on creating a large group of classes, you still need to know about interfaces because they are used extensively in the organization of the classes built into Java. For example, **Iterator is an interface**. You will continue to encounter interfaces as you learn more about Java.

Look up Iterator and List in the Java API. You'll notice that interface names are in italics.