“It’s classes all the way down”

Introduction

Typically, a discussion of classes in Java focuses heavily on object-oriented programming (OOP), and how the fundamental features of OOP are (or aren’t) implemented by the class mechanism of Java. However, this misses a basic point: Java classes aren’t just for OOP.

But let’s begin with a reminder—in the form of a cosmological parallel—that in Java, all classes extend (directly or indirectly) the `Object` class.

> After a lecture on cosmology and the structure of the solar system, William James was accosted by a little old lady.
>
> “Your theory that the sun is the centre of the solar system, and the earth is a ball which rotates around it has a very convincing ring to it, Mr. James, but it’s wrong. I’ve got a better theory,” said the little old lady.
>
> “And what is that, madam?” Inquired James politely.
>
> “That we live on a crust of earth which is on the back of a giant turtle.”
>
> Not wishing to demolish this absurd little theory by bringing to bear the masses of scientific evidence he had at his command, James decided to gently dissuade his opponent by making her see some of the inadequacies of her position.
>
> “If your theory is correct, madam," he asked, "what does this turtle stand on?”
>
> “You’re a very clever man, Mr. James, and that’s a very good question,” replied the little old lady, “but I have an answer to it. And it is this: The first turtle stands on the back of a second, far larger, turtle, who stands directly under him.”
>
> “But what does this second turtle stand on?” persisted James patiently.
>
> To this the little old lady crowed triumphantly. “It’s no use, Mr. James—it’s turtles all the way down.”
>
> — J. R. Ross, “Constraints on Variables in Syntax,” 1967

In Java, with the exception of primitives, it’s objects all the way down. But in the Java version of the turtle story, there actually would be a bottom of the pile: the `Object` class.

Different kinds of classes for different purposes

Java classes can be informally categorized by their intended purpose; I divide them into 4 general groups:

1. A class may serve to define and include one or more *entry points* or *lifecycle methods* for an executable unit of code. In fact, let's state this in even stronger terms: *we can't really run anything in Java without putting it in a class*. For example, execution of a Java application always starts with the loading of a startup class, and the invocation of the `main` method in that class.

   In general, entry points are very specific: the *signature* (the name, parameter types, and parameter order), return type, and modifiers of the method must match those expected for the given type of entry point. So far, we've looked at Java applications, which have as their entry point a `main` method that is `public` and `static`, with a single parameter of type `String[]`, and a `void` return type. There are other kinds of entry points (e.g. the `run` method for a Java thread) and lifecycle methods (the `init`, `start`, `stop`, and `destroy` methods of a Java applet), as well; we'll work with some of these later in the course. (By the way, the `main` method is not just the entry point for a Java application; it is also the only standard lifecycle method common to all Java applications: The application starts with the invocation of the `main` method, and it completes when `main` returns, or when it terminates through an exception.)

2. A class may act as a coherent collection of fields and methods, related by the nature of the operations they perform or the types of data involved. We might think of this type of class as a drawer in a mechanic's toolbox, filled with tools for performing a set of related tasks.

   A good example of this type of class is the `java.lang.Math` class, which includes methods for performing and returning the results of mathematical computations.

3. We can define a class for the purpose of creating a new type of object, with specialized *state* (fields or attributes) and *behavior* (methods). When we create objects based on such a class, the objects are referred to as *instances* of the class.

   In some programming languages, such a class can be an entirely new type—that is, its definition isn't based on any other class. (This is more typical of an *object-based* language, rather than an *object-oriented* language.) In Java, however, all Java classes are defined in a class hierarchy, with `java.util.Object` at the top; if we don't explicitly extend an existing class when we define a new one, the new class implicitly extends the `Object` class. This kind of class definition—extending a general class by defining a more specialized one—is what we're usually talking about when we discuss classes in the context of object-oriented programming.

   Definition of new types based on others is called *inheritance*. Further, if type *B* is based on type *A*, and we can treat an object of type *B* as if it were an object of type *A*, then it's also *subtyping*, which is also part of what is meant by *polymorphism*. (In this context, *type* and *class* are synonymous.) When we talk about inheritance in Java programming, we're implicitly discussing aspects of polymorphism—including subtyping—as well.

   We can see an analogue of this concept in the taxonomic relationship between the domesticated dog and the wolf. *Canis lupus familiaris* (dog) is a subspecies of *Canis lupus* (wolf), and many of the attributes and behaviors of *Canis lupus* are shared by *Canis lupus familiaris*; this is *inheritance*, in OOP terms. In most contexts, we treat dogs as dogs. However, dogs are still members of the species *Canis lupus*, and in some contexts, we treat them as such, without distinguishing them from wolves; this is *subtyping*. Finally, there are some behaviors which, while exhibited by both dogs and wolves, are distinctively different between the two; this is a kind of *polymorphism*, where behaviors with the same name differ, depending on the type of object.

   You'll create lots of classes of this kind in your projects, in this bootcamp and beyond. Also, one of the classes you'll use in virtually every project fits mostly into this group: the `java.lang.String` class. In Java, a string—even literal quoted text, like "Hello World!"—is actually an instance of the `String` class, with *identity* (two strings with identical text content are still distinguishable from each other), *state* (the text content), and *behavior* (methods that can be used to retrieve information about the state, or modify the state—in the case of `String`, methods that modify the state actually create and return a new `String` with that modified state).

4. Finally, we have a category that has a lot of overlap with the previous category—but the distinctions are important enough that I prefer to treat this as a category of its own.

   We can define a class not primarily for the purpose of being able to creating and use instances of the class in our code, but in order for other, more specialized classes to extend it further. In some cases, we will even make it *impossible* to create instances of the more general class we're defining.

   Taking another example from the animal kingdom, consider the genus *Felis*: there are many species in the genus (including *Felis catus*, the domestic cat), and while there are attributes and behaviors that are common to all members of the genus, there aren't any actual cats that are just of that genus; instead, there are cats belonging to one of the species in the genus. Nonetheless, it makes sense to have the genus defined, and to identify the distinguishing characteristics of the genus, even though we know that any actual cats in the genus will be more specialized than that. When we define classes like this in Java, we call them *abstract classes*.

Many classes primarily in the 1st or 2nd category are not intended to be used to create objects, or to be further specialized; thus. the position of such a class in a hierarchy of types is secondary. For example, your `HelloWorld` class serves as an entry point to a Java application, and is not used to create new instances of the `HelloWorld` class; thus, the fact that the `HelloWorld` class is a subclass of `Object` not only doesn't interest us—it's entirely unimportant. Many Java application startup classes have this in common; in fact, a common complaint about Java is that running even a very simple application requires a class, though that function has little to do with object-oriented programming.

The `java.lang.Math` class is an example of the 2nd category of class: it can't be used to create `Math` objects (instead, it operates on values of the intrinsic `int`, `long`, `float`, and `double` types), and can't be specialized further by creating subclasses of `Math`. (Of course, that doesn't mean we can't write a different class to perform similar or improved mathematical computations, and use that class in place of `Math`.)

The above categories aren't mutually exclusive, and few classes fit exactly into this scheme. In practice, most of the classes you'll create, and most of the existing classes you'll use, fit into more than one of the categories. However, this categorization can be a useful way of looking at a class—in terms of its primary purpose, and how that purpose is reflected in the code.

Members and modifiers

Constructors

When the purpose of a class is to define a new object type, we usually include non-`private` (see below) *constructors* in the class definition. The constructor's job is to initialize the state of an instance of the class. A constructor looks like a method, with a couple of important differences:

- A constructor *always* has the same name (including case) as the class itself. For example, a constructor for a class named `MyClass` would also be named `MyClass`. In fact, we see this in action whenever we write an initialization statement such as

  ```
  SomeClass somevar = new SomeClass();
  ```

  When we use the `new` keyword to create objects, `new` creates the instance, then invokes the constructor to initialize it, and then returns a reference to the new instance—which may be assigned to an object variable such as `someVar`. (Array objects are created in essentially the same fashion, apart from the specification of constructors.)

- A constructor has no return type, not even `void`.

```java
class MyClass {

  MyClass() {
    // Initialize the state of a MyClass instance.
  }

}
```

Access modifiers

Sometimes, the categorization of a class's intended purpose is reflected in *access modifiers* used in the class definition, and in the definitions of the class *members* (the fields, methods, and constructors in the class).

`public`

> As we've already discussed, a `public` class is one that is visible outside the package where it's defined. In general, any class intended for use as an entry point *must* be `public`. Similarly, if we write a class as a collection of related fields and methods, intending it for use as a toolbox for other developers, it would probably defeat that purpose if we don't declare the class `public`.
>
> On the other hand, a class whose purpose is to define a new object type may or may not be `public`, depending on our needs. In some cases, we might define a new object type that is only intended for use in our own code, and just in the package where the class is defined; in that case, we might not make the class `public`.

`public` can also be used with class *members* . This serves to make them visible outside the class—even outside the package where the class is defined.

(Note that Java allows no more than one `public` class to be defined in a `.java` source file. As a matter of practice, we'll take that one step further, and include no more than one top-level class—whether `public` or not—in a source file.)

`private`

We can use the `private` modifier on members of a class. By doing that, we make those class members visible only inside the class; code in other classes—even in the same package—can't access those members.

When we're defining a class that's primarily in the 1st or 2nd category—that is, when we don't intend for it to be used to create objects—we often define a `private` constructor with no parameters. (Note that if we don't define any constructors at all, Java defines a `public` constructor with no parameters as a *default constructor*.)

`protected`

The `protected` access modifier is used on members of a class, to make them accessible only within the class, and from subclasses of the class. Thus, we tend to use `protected` only when we're defining a class that falls at least partly in the 4th category—that is, a class that we anticipate will be further specialized by subclasses.

(default: *package-private*)

When no access modifier is included in a class or member definition, the default *package-private* access rule is in force. With this access level, classes are visible and accessible to code in the same package; similarly, class members with package-private access are visible and accessible to code in the same class and in the package in which the class is defined.

Other modifiers

In addition to the access modifiers, there are several other modifiers that can be applied to classes and members. Some of them reflect the purpose of a class, and its categorization in the above scheme; these are summarized here.

`static`

This modifier is used on class members, to associate them with the class as a whole, rather than with objects created from the class. We've already seen one very important use of this modifier, in Java applications: The Java application launcher expects to find a `main` method in the startup class—one that is not only `public` (otherwise, the launcher won't be able to see it) but also `static` (so the launcher can invoke it immediately after loading the class itself into memory, without having to create an instance of the class).

Within a non-`static` method, the current instance can be referenced via the `this` keyword. (Note that when doing so does not result in ambiguity, the `this` keyword can be omitted from code in instance methods when referring to fields and other methods of the same instance.) However, in a `static` method, there's no instance to reference; thus, `this` can't be used. On the other hand, in both `static` and non-`static` methods, the current class can be referenced by name—thus, since `static` members are accessible in the context of the class, they can be referenced via the class name. (Once again, if no ambiguity would result, the class name can be omitted from code in instance or `static` methods, when referring to `static` fields and methods of the same class.)

A class that's primarily in the 1st or 2nd category, and not the 3rd, will often have only `static` fields and methods, since it might be neither necessary nor desirable to create instances of the class (and thus, non-`static` fields and methods will never be used anyway). Many other classes will have a combination of `static` and non-`static` members. For example, the `java.awt.Color` class has `static` fields (representing specific color constants), some `static` methods (used for various color lookups and conversions), and some instance methods (primarily for obtaining information about `Color` instances).

`abstract`

We'll learn more about this modifier when we look at `abstract` classes in detail. For now, a quick summary: `abstract` can be applied to classes, and to methods (if used with a method, it *must* also be applied to the class as a whole). An `abstract` method is one which is is not yet implemented: all that appears is the declaration, with no method body. An `abstract` class is one which cannot be instantiated—that is, we can't create object instances of that class. Instead, we must define a subclass of the `abstract` class, implementing any `abstract` methods in the process, and then create instances of the subclass. Thus, a class of this type falls primarily into the 4th category: its purpose is to establish a branch in the type hierarchy, for further specialization by one or more subclasses based on it.

`final`

This modifier can be applied to classes and members. When applied to a class, it specifies that no further specialization is allowed—that is, subclasses of a `final` class can't be defined. When applied to a field, it indicates that once a value is assigned to that field (either in the declaration, or in a constructor, or in a static initializer—more about that last one later), it can't be changed. When assigned to a method, it means that the method can't be overridden or hidden in a subclass.

Applied to fields, `final` is often used in combination with `static` to define a constant. For example, in `java.lang.Math`, there is a declaration for `public static final PI`, with an assigned value of `3.141592653589793`. If we attempt to write code that assigns a new value to `Math.PI`, compilation will fail, since the value of a `final` field cannot be modified after the initial assignment.

As noted above, when applied to methods and classes, `final` is used to limit (or eliminate entirely) the possibility of further specialization. This might be done for a class that is primarily not in the 4th category (i.e. it's purpose is not to extend the class hierarchy); it might also be done for a class or method that is already highly specialized, and the developer feels that it will be difficult to specialize it further without introducing errors.

Summary

Most of the time we spend programming in Java is focused on creating classes of one kind or another. However, the clearer we are from the start about the intended purpose of a class (an entry point; a collection of related methods; a new object type; or a general type intended for specialization by subtypes) the better the chance that we can deliver a well-written class, with minimal effort spent on reworking our code in an effort to “get it right this time.”