Working With Statics

So what does the term *static* mean in Java? It’s used to describe a special type of

field or method that isn’t associated with a particular instance of a class. Instead,

static fields and methods are associated with the class itself, which means that

you don’t have to create an instance of the class to access a static field or methods.

Instead, you access a static field or method by specifying the class name, not a

variable that references an object.

* The value of a static field is the same across all instances

private static int ballCount;

static private int ballCount;

Convention to put visibility keyword first

The best-known static method is main, which is called by the Java runtime to start

an application. The main method must be static — which means that applications

are run in a static context by default.

One of the basic rules of working with static methods is that you can’t access a

nonstatic method or field from a static method, because the static method doesn’t

have an instance of the class to use to reference instance methods or fields.

You *can* access static methods and fields from an instance method

One common use for static variables is to keep track of how many instances of a

class have been created.

Sometimes you want to create a class that can’t be instantiated at all. Such a class

consists entirely of static fields and methods.

All you have to do to prevent a class instance from being created, then, is provide

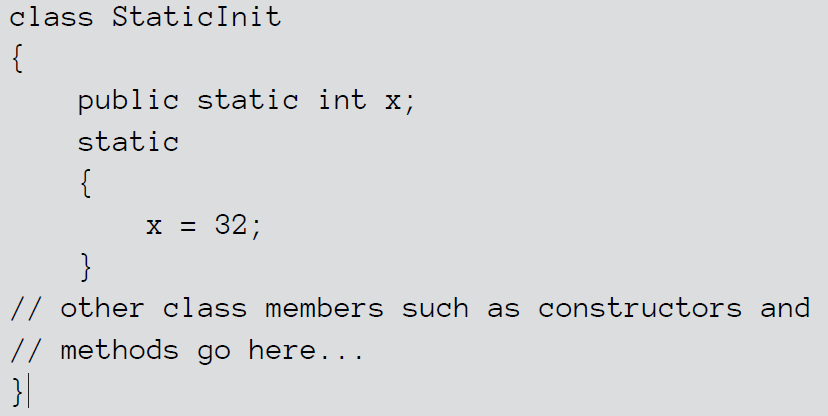
a single private constructor.

Java provides a feature called a *static initializer* that’s designed specifically to let

you initialize static fields.

If a class has more than one static initializer, the initializers are executed in the

order in which they appear in the program.



Using the Object and Class Classes

* Every object is a Object
* No extends on a class then that class extends Object
* Object can be a type
* ArrayList constructor accepts Object as a argument
* Primitives aren’t Objects

# Methods

clone: This method is commonly used to make copies of objects, and

overriding it in your own classes is not uncommon. I explain this method in

detail later in this chapter, in the section “The clone Method.”

equals: This method is commonly used to compare objects. Any class that

represents an object that can be compared with another object should

override this method. Turn to the section “The equals Method,” later in this

chapter, for more info.

finalize: This method is called when the garbage collector realizes that an

object is no longer being used and can be discarded. The intent of this

method is to let you create objects that clean up after themselves by closing

open files and performing other cleanup tasks before being discarded. But

because of the way the Java garbage collector works, there’s no guarantee

that the finalize method is ever actually called. As a result, this method isn’t

commonly used.

getClass: This method is sometimes used in conjunction with the Class

class, which I describe later in this chapter, in the section “The Class Class.”

hashCode: Every Java object has a *hash code,* which is an int representation of

the class that’s useful for certain operations. This method isn’t terribly

important until you start to work with hash tables — which is a pretty

advanced technique, best left to people with pocket protectors and tape

holding their glasses together.

toString: This method is one of the most commonly used methods in Java.

I describe it in the section “The toString Method,” later in this chapter.

getName(): Returns a String representing the name of the classgetSuperclass(): Returns another Class object representing this Class object’s superclass

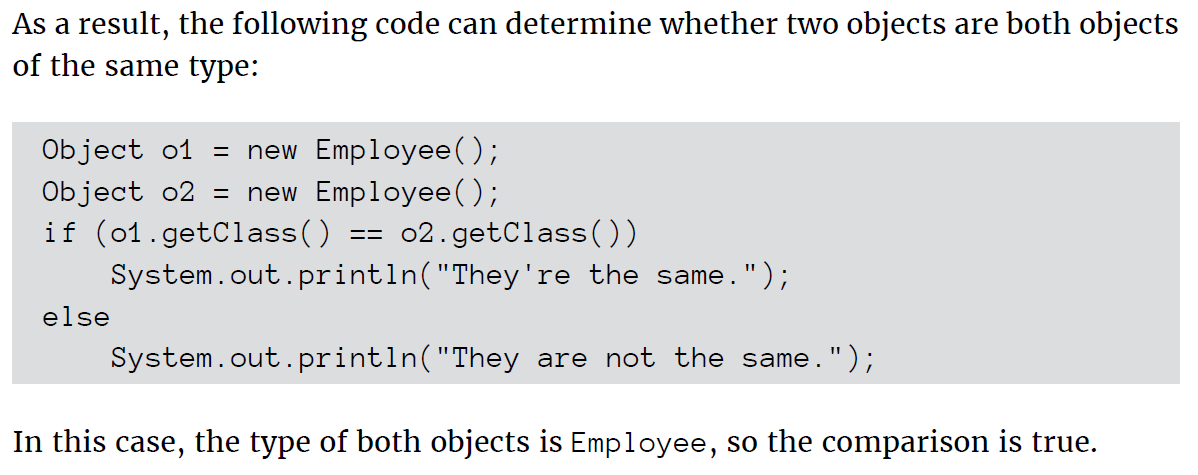
# The Class Class

Every class used by a Java application is represented in memory by an object of

type Class. If your program uses Employee objects, for example, there’s also a

Class object for the Employee class. This Class object has information not about

specific employees but about the Employee class itself.



Creating Generic Collection Classes

ArrayList<Employee> empList = new ArrayList<Employee>();

In short, if you want to create a method that accepts any type of ArrayList, you

have to code the method like this:

public void addItems(ArrayList<?> list)

In that case, you can add an extends clause to the wildcard, like this:

public void addItems(ArrayList<? extends Employee> list)

Then you can call the addItems method with an ArrayList of type Employee,

HourlyEmployee, or SalariedEmployee.

Now, before you call it a day, take this example one step further: Suppose this

addItems method appears in a generic class that uses a formal type parameter

<E> to specify the type of elements the class accepts, and you want the addItems

method to accept an ArrayList of type E or any of its subclasses. To do that, you’d

declare the addItems method like this:

public void addItems(ArrayList<? extends E> list)

Here the wildcard type parameter <? extends E> simply means that the Array

List can be of type E or any type that extends E.

Java 7 introduced a new feature called the *diamond operator* that lets you skip the

type when you call the constructor, like this:

ArrayList<String> nums = new ArrayList<>();

Here’s another example:

ArrayList<String> nums;

nums = new ArrayList<>();  
The diamond operator simply deduces the correct type based on the variable’s

type.