8.11 `static` Class Members

Every object has its own copy of all the instance variables of the class. In certain cases, only one copy of a particular variable should be *shared* by all objects of a class. A `static` **field**—called a **class variable**—is used in such cases. A `static` variable represents **classwide information**—all objects of the class share the *same* piece of data. The declaration of a `static` variable begins with the keyword `static`.

Motivating `static`

Let's motivate `static` data with an example. Suppose that we have a video game with `Martian`s and other space creatures. Each `Martian` tends to be brave and willing to attack other space creatures when the `Martian` is aware that at least four other `Martian`s are present. If fewer than five `Martian`s are present, each of them becomes cowardly. Thus, each `Martian` needs to know the `martianCount`. We could endow class `Martian` with `martianCount` as an *instance variable*. If we do this, then every `Martian` will have *a separate copy* of the instance variable, and every time we create a new `Martian`, we'll have to update the instance

variable `martianCount` in every `Martian` object. This wastes space with the redundant copies, wastes time in updating the separate copies and is error prone. Instead, we declare `martianCount` to be `static`, making `martianCount` classwide data. Every `Martian` can see the `martianCount` as if it were an instance variable of class `Martian`, but only *one* copy of the `static martianCount` is maintained. This saves space. We save time by having the `Martian` constructor increment the `static martianCount`—there's only one copy, so we do not have to increment separate copies for each `Martian` object.

Software Engineering Observation 8.9

Use a `static` *variable when all objects of a class must use the same copy of the variable.*

Class Scope

Static variables have *class scope*—they can be used in all of the class's methods. We can access a class's `public static` members through a reference to any object of the class, or by qualifying the member name with the class name and a dot (`.`), as in `Math.sqrt(2)`. A class's `private`

`static` class members can be accessed by client code only through methods of the class. Actually, `static` *class members exist even when no objects of the class exist*—they're available as soon as the class is loaded into memory at execution time. To access a `public static` member when no objects of the class exist (and even when they do), prefix the class name and a dot (`.`) to the `static` member, as in `Math.PI`. To access a `private static` member when no objects of the class exist, provide a `public static` method and call it by qualifying its name with the class name and a dot.

Software Engineering Observation 8.10

Static class variables and methods exist, and can be used, even if no objects of that class have been instantiated.

`static` Methods Cannot Directly Access Instance Variables and Instance Methods

A `static` method *cannot* directly access a class's instance

variables and instance methods, because a `static` method can be called even when no objects of the class have been instantiated. For the same reason, the `this` reference *cannot* be used in a `static` method. The `this` reference must refer to a specific object of the class, and when a `static` method is called, there might not be any objects of its class in memory.

Common Programming Error 8.5

A compilation error occurs if a `static` *method calls an instance method in the same class by using only the method name. Similarly, a compilation error occurs if a* `static` *method attempts to access an instance variable in the same class by using only the variable name.*

Common Programming Error 8.6

Referring to `this` *in a* `static` *method is a compilation error.*

Tracking the Number of

Employee Objects That Have Been Created

Our next program declares two classes—`Employee` (Fig. 8.12) and `EmployeeTest` (Fig. 8.13). Class `Employee` declares `private static` variable `count` (Fig. 8.12, line 6) and `public static` method `getCount` (lines 32–34). The `static` variable `count` maintains a count of the number of objects of class `Employee` that have been created so far. This class variable is initialized to `0` in line 6. If a `static` variable is *not* initialized, the compiler assigns it a default value—in this case `0`, the default value for type `int`.

```
          1   // Fig. 8.12: Employee.java
 2   // static variable used to maintain a count of t
        3   // Employee objects in memory.
                        4
            5   public class Employee {
 6      private static int count = 0; // number of Em
          7      private String firstName;
          8      private String lastName;
                        9
10      // initialize Employee, add 1 to static count
11      // output String indicating that constructor
12      public Employee(String firstName, String last
      13          this.firstName = firstName;
       14          this.lastName = lastName;
                        15
16         ++count; // increment static count of empl
17         System.out.printf("Employee constructor: %
    18             firstName, lastName, count);
                    19      }
                        20
            21      // get first name
```

```
22      public String getFirstName() {
23         return firstName;
24      }
25
26      // get last name
27      public String getLastName() {
28         return lastName;
29      }
30
31      // static method to get static count value
32      public static int getCount() {
33         return count;
34      }
35   }
```

Fig. 8.12

`static` variable used to maintain a count of the number of `Employee` objects in memory.

When `Employee` objects exist, variable `count` can be used in any method of an `Employee` object—this example increments `count` in the constructor (line 16). The `public static` method `getCount` (lines 32–34) returns the number of `Employee` objects that have been created so far. When no objects of class `Employee` exist, client code can access variable `count` by calling method `getCount` via the class name, as in `Employee.getCount()`.

Good Programming Practice 8.2

Invoke every `static` *method by using the class name and a dot (*`.`*) to emphasize that the method being called is a* `static` *method.*

When objects exist, `static` method `getCount` also can be called via any reference to an `Employee` object. This contradicts the preceding Good Programming Practice and, in fact, the Java SE 9 compiler issues warnings on lines 16–17 of Fig. 8.13.

Class `EmployeeTest`

`EmployeeTest` method `main` (Fig. 8.13) instantiates two `Employee` objects (lines 11–12). When each `Employee` object's constructor is invoked, lines 13–14 of Fig. 8.12 assign the `Employee`'s first name and last name to instance variables `firstName` and `lastName`. These statements do *not* copy the original `String` arguments. `String`s in Java are **immutable**—they cannot be modified after they're created. Therefore, it's safe to have *many* references to one `String` object. This is not normally the case for objects of most other classes in Java. If `String` objects are immutable, you might wonder why we're able to use operators `+` and `+=` to concatenate `String` objects. String concatenation actually

results in a *new* `String` object containing the concatenated values. The original `String` objects are *not* modified.

```
1   // Fig. 8.13: EmployeeTest.java
2   // static member demonstration.
3
4   public class EmployeeTest {
5      public static void main(String[] args) {
6         // show that count is 0 before creating Em
7         System.out.printf("Employees before instan
8            Employee.getCount());
9
10        // create two Employees; count should be 2
11        Employee e1 = new Employee("Susan", "Baker
12        Employee e2 = new Employee("Bob", "Blue");
13
14        // show that count is 2 after creating two
15        System.out.printf("%nEmployees after insta
16        System.out.printf("via e1.getCount(): %d%n
17        System.out.printf("via e2.getCount(): %d%n
18        System.out.printf("via Employee.getCount()
19           Employee.getCount());
20
21        // get names of Employees
22        System.out.printf("%nEmployee 1: %s %s%nEm
23           e1.getFirstName(), e1.getLastName(),
24           e2.getFirstName(), e2.getLastName());
25     }
26  }
```

```
Employees before instantiation: 0
Employee constructor: Susan Baker; count = 1
Employee constructor: Bob Blue; count = 2
Employees after instantiation:
via e1.getCount(): 2
via e2.getCount(): 2
```

```
via Employee.getCount(): 2

Employee 1: Susan Baker
Employee 2: Bob Blue
```

Fig. 8.13

`static` member demonstration.

When `main` terminates, local variables `e1` and `e2` are discarded—remember that a local variable exists *only* until the block in which it's declared completes execution. Because `e1` and `e2` were the only references to the `Employee` objects created in lines 11–12 (Fig. 8.13), these objects become "eligible for garbage collection" as `main` terminates.

In a typical app, the garbage collector *might* eventually reclaim the memory for any objects that are eligible for collection. If any objects are not reclaimed before the program terminates, the operating system will reclaim the memory used by the program. The JVM does *not* guarantee when, or even whether, the garbage collector will execute. When it does, it's possible that no objects or only a subset of the eligible objects will be collected.