3.3 Account Class: Initializing Objects with Constructors

As mentioned in Section 3.2, when an object of class `Account` (Fig. 3.1) is created, its `String` instance variable `name` is initialized to `null` by *default*. But what if you want to provide a name when you *create* an `Account` object?

Each class you declare can optionally provide a *constructor* with parameters that can be used to initialize an object of a class when the object is created. Java *requires* a constructor call for *every* object that's created, so this is the ideal point to initialize an object's instance variables. The next example enhances class `Account` (Fig. 3.5) with a constructor that can receive a name and use it to initialize instance variable `name` when an `Account` object is created (Fig. 3.6).

```
// Fig. 3.5: Account.java
// Account class with a constructor that initial

public class Account {
   private String name; // instance variable

   // constructor initializes name with paramete
   public Account(String name) { // constructor
      this.name = name;
   }

   // method to set the name
   public void setName(String name) {1
      this.name = name;
   }

   // method to retrieve the name
   public String getName() {
      return name;
   }
}
```

Fig. 3.5

`Account` class with a constructor that initializes the `name`.

3.3.1 Declaring an `Account` Constructor for Custom Object Initialization

When you declare a class, you can provide your own constructor to specify *custom initialization* for objects of your class. For example, you might want to specify a name for an `Account` object when the object is created, as you’ll see in line 8 of Fig. 3.6:

```
Account account1 = new Account("Jane Green");
```

In this case, the `String` argument `"Jane Green"` is passed to the `Account` object’s constructor and used to initialize the `name` instance variable. The preceding statement requires that the class provide a constructor that takes only a `String` parameter. Figure 3.5 contains a modified `Account` class with such a constructor.

Account Constructor Declaration

Lines 8–10 of Fig. 3.5 declare `Account`’s constructor, which *must* have the *same name* as the class. A constructor’s *parameter list* specifies that the constructor requires zero or more pieces of data to perform its task. Line 8 indicates that the constructor has exactly one parameter—a `String` called `name`. When you create a new `Account` object, you’ll pass a person’s name to the constructor’s `name` parameter. The constructor will then assign the `name` parameter’s value to the *instance variable* `name` (line 9).

Error-Prevention Tip 3.2

Even though it’s possible to do so, do not call methods from constructors. We’ll explain this in Chapter 10, Object-Oriented Programming: Polymorphism and Interfaces.

Parameter `name` of Class `Account`’s Constructor and Method `setName`

Recall from Section 3.2.1 that method parameters are local variables. In Fig. 3.5, the constructor and method `setName` both have a parameter called `name`. Although these parameters have the same identifier (`name`), the parameter in line 8 is a local variable of the constructor that’s *not* visible to method `setName`, and the one in line 13 is a local variable of `setName` that’s *not* visible to the constructor.

3.3.2 Class `AccountTest`: Initializing `Account` Objects When They’re Created

The `AccountTest` program (Fig. 3.6) initializes two `Account` objects using the constructor. Line 8 creates and initializes the `Account` object `account1`. Keyword `new` requests memory from the system to store the `Account` object, then implicitly calls the class’s constructor to *initialize* the object. The call is indicated by the parentheses after the class name, which contain the *argument* `"Jane Green"` that’s used to initialize the new object’s name. Line 8 assigns the new object to the variable `account1`. Line 9 repeats this process, passing the argument `"John Blue"` to initialize the name for `account2`. Lines 12–13 use each object’s `getName` method to obtain the names and show that they were indeed initialized when the objects were *created*. The output shows *different* names, confirming that each `Account` maintains its *own copy* of the instance variable `name`.

```
1   // Fig. 3.6: AccountTest.java
2   // Using the Account constructor to initialize t
3   // variable at the time each Account object is c
4
5   public class AccountTest {
6      public static void main(String[] args) {
7         // create two Account objects
8         Account account1 = new Account("Jane Green
9         Account account2 = new Account("John Blue"
10
11        // display initial value of name for each
12        System.out.printf("account1 name is: %s%n"
13        System.out.printf("account2 name is: %s%n"
14     }
15  }
```

```
account1 name is: Jane Green
account2 name is: John Blue
```

Fig. 3.6

Using the `Account` constructor to initialize the `name` instance variable at the time each `Account` object is created.

Constructors Cannot Return Values

An important difference between constructors and methods is that *constructors cannot return values*, so they *cannot* specify a return type (not even `void`). Normally, constructors are declared `public`—later in the book we’ll explain when to use `private` constructors.

Default Constructor

Recall that line 11 of Fig. 3.2

```
Account myAccount = new Account();
```

used `new` to create an `Account` object. The *empty* parentheses after “`new Account`” indicate a call to the class’s **default constructor**—in any class that does *not* explicitly declare a constructor, the compiler provides a default constructor (which always has no parameters). When a class has only the default constructor, the class’s instance variables are initialized to their *default values*. In Section 8.5, you’ll learn that classes can have multiple constructors.

There’s No Default

Constructor in a Class That Declares a Constructor

If you declare a constructor for a class, the compiler will *not* create a *default constructor* for that class. In that case, you will not be able to create an `Account` object with the class instance creation expression `new Account()` as we did in Fig. 3.2—unless the custom constructor you declare takes *no* parameters.

Software Engineering Observation 3.3

Unless default initialization of your class's instance variables is acceptable, provide a custom constructor to ensure that your instance variables are properly initialized with meaningful values when each new object of your class is created.

Adding the Constructor to Class `Account`'s UML Class Diagram

The UML class diagram of Fig. 3.7 models class `Account` of Fig. 3.5, which has a constructor with a `String name` parameter. As with operations, the UML models constructors in the *third* compartment of a class diagram. To distinguish a constructor from the class's operations, the UML requires that the word "constructor" be enclosed in **guillemets (« and »)** and placed before the constructor's name. It's customary to list constructors *before* other operations in the third compartment.

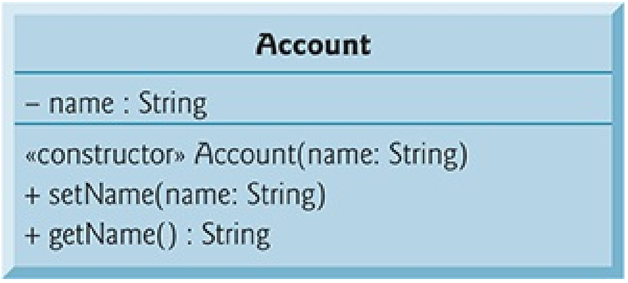

Fig. 3.7

UML class diagram for `Account` class of Fig. 3.5.

Description