6.11 Scope of Declarations

You've seen declarations of various Java entities, such as classes, methods, variables and parameters. Declarations introduce names that can be used to refer to such Java entities. The **scope** of a declaration is the portion of the program that can refer to the declared entity by its name. Such an entity is said to be "in scope" for that portion of the program. This section introduces several important scope issues.

The basic scope rules are as follows:

1. The scope of a parameter declaration is the body of the method in which the declaration appears.
2. The scope of a local-variable declaration is from the point at which the declaration appears to the end of that block.
3. The scope of a local-variable declaration that appears in the initialization section of a `for` statement's header is the body of the `for` statement and the other expressions in the header.
4. A method or field's scope is the entire body of the class. This enables a class's instance methods to use the fields and other methods of the class.

Any block may contain variable declarations. If a local variable or parameter in a method has the same name as a field of the class, the field is *hidden* until the block terminates execution—this is called **shadowing**. To access a shadowed field in a block:

- If the field is an instance variable, precede its name with the `this`

keyword and a dot (`.`), as in `this.x`.

- If the field is a `static` class variable, precede its name with the class's name and a dot (`.`), as in *ClassName*`.x`.

It's a compilation error if multiple *local* variables have the same name in the same method.

Figure 6.9 demonstrates field and local-variable scopes. Line 6 declares and initializes the field `x` to `1`. This field is *shadowed* in any block (or method) that declares a local variable named `x`. Method `main` declares a local variable `x` (line 11) and initializes it to `5`. This local variable's value is output to show that the field `x` (whose value is `1`) is *shadowed* in `main`.

```
1 // Fig. 6.9: Scope.java
2 // Scope class demonstrates field and local-variab
3
4 public class Scope {
5    // field that is accessible to all methods of t
6    private static int x = 1;
7
8    // method main creates and initializes local va
9    // and calls methods useLocalVariable and useFi
10   public static void main(String[] args) {
11      int x = 5; // method's local variable x shad
12
13      System.out.printf("local x in main is %d%n",
14
15      useLocalVariable(); // useLocalVariable has
16      useField(); // useField uses class Scope's f
17      useLocalVariable(); // useLocalVariable rein
18      useField(); // class Scope's field x retains
19
20      System.out.printf("%nlocal x in main is %d%n
21   }
```

```
                        22
23    // create and initialize local variable x durin
  24    public static void useLocalVariable() {
25       int x = 25; // initialized each time useLoca
                        26
           27       System.out.printf(
28          "%nlocal x on entering method useLocalVar
29       ++x; // modifies this method's local variabl
           30       System.out.printf(
31          "local x before exiting method useLocalVa
                    32    }
                      33
34    // modify class Scope's field x during each cal
      35    public static void useField() {
           36       System.out.printf(
37          "%nfield x on entering method useField is
38       x *= 10; // modifies class Scope's field x
           39       System.out.printf(
40          "field x before exiting method useField i
                    41    }
                     42 }
```

```
                local x in main is 5
local x on entering method useLocalVariable is 25
local x before exiting method useLocalVariable is 26
     field x on entering method useField is 1
   field x before exiting method useField is 10
local x on entering method useLocalVariable is 25
local x before exiting method useLocalVariable is 26
    field x on entering method useField is 10
  field x before exiting method useField is 100
                local x in main is 5
```

Fig. 6.9

`Scope` class demonstrates field and local-variable scopes.

The program declares two other methods —`useLocalVariable` (lines 24–32) and `useField` (lines 35–41)—that take no arguments and return no results. Method `main` calls each method twice (lines 15–18). Method `useLocalVariable` declares local variable `x` (line 25). When `useLocalVariable` is first called (line 15), it creates local variable `x` and initializes it to `25`, outputs the value of `x` (lines 27–28), increments `x` (line 29) and outputs the value of `x` again (lines 30–31). When `uselLocalVariable` is called a second time (line 17), it *recreates* local variable `x` and *reinitializes* it to `25`, so the output of each `useLocal-Variable` call is identical.

Method `useField` does not declare any local variables. Therefore, when it refers to `x`, the field `x` (line 6) of the class is used. When method `useField` is first called (line 16), it outputs the value (`1`) of field `x` (lines 36–37), multiplies the field `x` by `10` (line 38) and outputs the value (`10`) of field `x` again (lines 39–40) before returning. The next time method `useField` is called (line 18), the field has its modified value (`10`), so the method outputs `10`, then `100`. Finally, in method `main`, the program outputs the value of local variable `x` again (line 20) to show that none of the method calls modified `main`'s local variable `x`, because the methods all referred to variables named `x` in other scopes.

Principle of Least Privilege

In a general sense, “things” should have the capabilities they need to get their job done, but no more. An example is the scope of a variable. A variable should not be visible when it’s not needed.

Good Programming Practice 6.3

Declare variables as close to where they’re first used as possible.