CS112 Lecture: Declarations and Scope

Last revised 2/18/08

Objectives:

1. To introduce the technical terms "identifier", "qualified", "scope", "shadow" and "visibility"
2. To review the use of "static"

Materials:

1. Scope Rules Examples Slide Show
2. Executable demonstration of program (in My Java Demos)

I. Identifiers and Declarations

One of the marks of a professional in any field is that he/she can use the technical vocabulary of the field with precision. That is one of the key issues today.

A. In a programming language, whenever a word occurs (outside of quotation marks or a comment) it is understood as being one of two things:

1. A keyword - one that has special meaning in the language. In any programming language, there is a fixed list keywords that are part of the language definition. In most languages, these are treated as reserved words that cannot be used for any other purpose in a program. (For what it's worth, a complete list of the reserved words of Java appears in section 3.9 of the Java language specification.)

2. An identifier.

3. In a language like Java, any word that is not a reserved word is considered to be an identifier.

   Example:

   ```
   for (int i = 0; i < n; i ++)
   ```

   The compiler recognizes `for` and `int` as reserved words. It considers `i` and `n` to be identifiers

B. The association of an identifier with a particular feature of a program is called a declaration.

1. Languages like Java require that all identifiers that appear in a program must be declared.

Example: `int width;`

`width` is identifier, and `int width;` is a declaration of the identifier `width` as being the name of a variable of type `int`.

2. Some languages allow implicit declarations of a variable by using it in an assignment statement - e.g.in some languages `width = 50` would be taken as a declaration of the variable `width`. The danger of this is that typographical errors can go uncaught - if one mistakenly typed `widh = 50`, it could be understood as an implicit declaration of a new variable `widh`, rather than a typo.

C. We have looked at quite a number of kinds of declarations that can occur in a Java program. Let's review:

1. Variable declarations

   a) Instance variables

   b) Class variables

   c) Local variables

   d) Parameters

2. Constant declarations

3. Method declarations

4. Constructor declarations

D. Whenever a identifier appears in a program, it must be associated with some declaration, whch is taken as defining the meaning of that name. (An identifier that cannot be associated with some declaration is reported as being an error - an undeclared identifier.)

1. Example:

```
Robot karel;
...
karel.move();
```

   In this case, the identifier `karel` in the second statement is associated with the declaration given in the first statement.

2. In the case of Java, the compiler treats variable/constant declarations and method declarations distinctly. If an identifier occurs without parentheses after it, only variable/constant declarations are considered. If it is followed by a left parenthesis, then only method declarations are considered.

   a) In effect, the Java compiler divides identifiers into two classes - variable identifiers and constant identifiers.

   b) Thus, it is possible (though not good practice) for a Java program to define the same word as <u>both</u> the name of a variable and the name of a method.

   Example: the following is legal, but not desirable!

```
void foo() {
   System.out.println("foo() called");
   int foo = 3;
   System.out.println(foo);
   foo();
}
```

   If a program calls `foo()`, the following output is produced ad infinitum:

```
foo() called
3
foo() called
...
```

E. In general, when an identifier is used, it can be <u>qualified</u> by some other identifier

   Example:

```
karel.move();
```

   The identifier `move()` is qualified by the identifier `karel`.

   1. A qualified name specifies the context in which an identifier is to be interpreted - in the above, the identifier `move()` is to be understood in the context of the object named by `karel`.

   2. An identifier that has not qualifier is called, of course, an <u>unqualified identifier.</u>

   3. In declarations, identifiers are <u>never</u> qualified. Only uses of identifiers can be qualified.

II. Scope

A. For any declaration, the scope of a declaration is the region of the program text where the identifier that has been declared can be used without qualification.

Examples:

1.
```
class Foo
{
     public int bar;

     ...
}
```

The identifier `bar` can be used anywhere inside class `Foo` without qualification. The scope of the declaration for `bar` is the entire class.

2.
```
class Foo
{
     public void something()
     {
          int bar;
          ...
     }
     ...
}
```

The identifier `bar` can be used anywhere inside the method `something()` without qualification. The scope of the declaration for `bar` is the method `something()`.

B. In Java, certain kinds of variables can never be qualified. Thus, they can only be used within their scope.

C. Before looking at the scope rules, we need to consider the concept of a block. In brief, a block is a body of code enclosed by braces - including any blocks contained within it.

Example:

```
public boolean rightIsClear()
{
    turnRight();
    if (frontIsClear())
    {
        turnLeft();
        return true;
    }
    else
    {
        turnLeft();
        return false;
    }
}
```

There are three blocks in this example, as shown by the three rectangles. Note that the outer block includes the inner two.

D. Here is a summary of the scope rules of Java for identifiers we have learned about so far. (There are actually a lot more! In particular, we don't consider class names.)

Type of declaration	Scope	Can be used elsewhere with qualification
method	entire class block - even before the declaration	yes
instance variable	entire class block - even before the declaration	yes
parameter	entire method - including rest of the header, plus complete block	no
local variable	rest of the block - beginning at the point of declaration to the end of the block	no
variable declared in for loop	for loop header plus statement	no

Examples of each (PROJECT - STOP WITH loop control variable - do not go onto second instance variable)

E. One interesting question that arises is what if we have a declaration for a given name occurring within the scope of another declaration for the same name?

1. In many cases, Java does not allow this to occur. In particular:

   a) A class cannot contain two methods with the same signature (name + parameter types)

   b) A method cannot contain a parameter or local or loop control variable declaration for the same name as another parameter of the method.

   c) A method cannot contain a local or loop control variable declaration for the same name as a local variable previously declared in the same method.

2. There is one place, however, where Java does allow this to occur: a method can contain a parameter or local or loop control variable declaration for the same name as an instance variable.

   a) In this case, the local declaration is said to shadow the instance variable. In particular, any unqualified use of the identifier in question will refer to the local declaration, not the instance variable. We say that the local declaration creates a hole in scope in the instance variable declaration.

      PROJECT final slide

   b) Given all this, what should the demonstration program print?

      ASK

      DEMO

3. In a case like this, how can one refer to the shadowed instance variable?

   ASK

   By qualifying the name with `this` - a reserved word that always refers to the current object.

   DEMO modifying program to say `this.var2` in the second `println` statement.

4. One place where this is commonly done is in constructors

   Example: In Lab 4, you created a PongBall class with a constructor that looked like this:

```
public PongBall(int initX, int initY, int initXV, int initYV) {
      x = initX;
      y = initY;
      xVelocity = initXV;
      yVelocity = initYV;
}
```

Suppose, instead, you had used the same names for the Constructor parameters as for the instance variables - not unreasonable, since the values specified are, in fact, the values for the instance variables. Then you could have written the constructor this way:

```
public PongBall(int x, int y, int xVelocity, int yVelocity) {
      this.x = x;
      this.y = y;
      this.xVelocity = xVelocity;
      this.yVelocity = yVelocity;
}
```

This would work correctly, because the unqualified names appearing on the right hand side of each of the assignments would be understood as referring to the parameters (because of shadowing); while the qualified names on the left hand side would necessarily refer to the instance variables.

III. Visibility

A. Another issue that arises with some Java declarations is visibility. Visibility has to do with whether a declaration may be used outside its scope (with an appropriate qualifier, of course).

B. In Java, parameters, local variables, and loop control variables are visible only within their scope. In fact, there is not even any way to qualify the name outside the scope.

C. However, identifiers declared as method names or instance variables can, in principle, be used outside their scope by qualifying them with the name of an object. Therefore, method and instance variable declarations carry an explicit visibility specifier which governs how the identifier being declared can be used. For now, you know just two visibility specifiers; there are, in fact, two more possible that you will learn about later.

1. When a method or instance variable is declared `public`, a qualified reference to the identifier declared is permissible anywhere in a program.

   Most methods of a class will be declared `public`, because they represent behaviors that can reasonably be invoked elsewhere in the program.

2. When a method or instance variable is declared `private`, a qualified reference to the identifier declared is permissible only in the same class.

   a) For now, you should always declare the instance variables of a class `private`.

   b) Instance methods are sometimes declared `private`, when they exist only to help other (`public`) instance methods do their job.

   c) Note that a `private` specifier does allow access from other objects of the same class.

      Example: many classes declare an `equals()` method that determines whether two different objects have the same value. This is typically done by comparing their instance variables - which requires that the `equals()` method of one object access the instance variables of another object of the same class, which is permitted.

IV. Static

A. We have already met the "`static`" modifier that can be used in class declarations. What does it mean?

   ASK

   A feature declared `static` belongs to the class as a whole. All the instances of the class share a single common copy.

B. Only methods and "top-level" variables can be declared `static`.

C. Anywhere within a class, `static` features are in scope, and can be referred to without being qualified. But outside a class, a `static` feature be qualified, typically by the class name.

   Example: the class `static` defines the `static` constant `PI`. To refer to this elsewhere in a program, one writes `Math.PI`. However, within the body of the class, the unqualfied form can be used. For example, the following is the actual body of the `toRadians()` method defined in the class:

```
public static double toRadians(double angdeg) {
      return angdeg / 180.0 * PI;
}
```