Methods in Java – Scope & Accessibility

A variable can generally be used at any point after it has been declared. The one exception to this rule is a variable declared in the control portion of a for loop, which can only be used within the loop. This is an example of *scope*, which identifies the range over which any *identifier* (e.g., variable or method) can be accessed.

In order to discuss *scope* in a precise manner, it is necessary to define a *block of code*. A *block* is a section of code enclosed by brace brackets. The scope of a variable spans from the point where it is declared to the end of the block where it was declared. Once we reach the brace bracket that closes the block, the variable can no longer be used.

Consider the following code:

```
public static void scopeSample (double a)  // scope of a begins
{
      int b = 0;                            // scope of b begins
      ...
      for (int c = 1; ...)                  // scope of c begins
      {
            int d = 2;                      // scope of d begins
            ...
      }                                     // scope of c,d ends
}                                           // scope of a,b ends
```

As you can see, our method included one parameter. The scope of a parameter is the entire method that uses the parameter, just as the scope of the variable in the `for` loop is the entire loop.

Once a block of code end, or terminates, there is no memory of any declarations from within the block. Thus it is possible (but not always advisable) to reuse an *identifier* in later blocks. Due to the structure of Java (and C, and C++), it is not possible for blocks to overlap, so scopes also do not overlap. This prevents the same identifier from being declared more than once in the same block, and an attempt to do so will produce a compiler error.

Since each method constitutes its own block of code, the declarations within one method are not visible to another method (including the `main` method). This is the reason why we can have variables with the same names in multiple methods, but it is still necessary to pass their values back and forth as parameters. In reality, these variables only share a name as seen by the programmer, but otherwise they occupy completely different spaces in memory.

Accessibility of Methods

So far, we have discussed the scope of variables within and between methods. There is also the question of scope as it relates to the methods themselves.

In a manner consistent with the scope of variables, any method can be accessed from within the same *block*. Since all methods are defined as part of a *class* (which has opening and closing brace brackets), we can say that all methods in a class are accessible to all other methods in the class. Note the one exception, the `main` method, which cannot be called by any other method in the class.

The Java language is made up of many classes spread across many files. We have already used some of these methods (e.g., `System.out.println` or `Math.sqrt`), so it must be possible to access methods across classes.

Just as a single class can contain multiple methods, it is possible to form larger groupings of multiple

classes. A collection of classes is known as a `package`, and the Java language is made up of a number of packages (e.g., `java.io` is a package of classes that handles input and output). When we create a file containing a class (or classes), we have the option of specifying a package for the class. If no package is specified, then the file belongs to the *default package* (the directory in which the file was created).

In the context of scope, it is worth observing that all of our methods, so far, have started with the word `public`. The word `public` is an example of an *access control modifier*. There is also a `private` access control modifier, and it is possible to omit the access control modifier completely (except for `main`, which must be `public`). There are other possibilities, but these three will suffice for our current needs.

The following table summarizes the effect of the *access control modifiers*.

access control modifier	summary
`public`	the method can be seen and called from anywhere – from any `method` in any `class` in any `package`
`private`	the method can only be accessed from within the `class` where it is defined
`<none>`	the method can only be accessed from within the `package` where it is defined

Exercises

1. In the method outline shown below, state which variables can be used at each numbered line.

```
public static void showScope (int i, double x)
{                                                    // 1
      int j;
      ...                                            // 2
      for (int k = 0; ... )
      {
            ...                                      // 3
            double y;
            ...                                      // 4
      }
      ...                                            // 5
      {
            float f;
            ...                                      // 6
      }
      double z;
      ...                                            // 7
}
```

2. What is a block?
3. What is:
   (a) the difference between defining a method as `public` and `private`?
   (b) the result of omitting both `public` and `private` from the header of a method?
4. For which method is the `public` modifier always required?