Scoping, Static Variables, Overloading, Packages

In this lecture, we will examine in more detail the notion of **scope** for variables. We've already indicated that variables only "exist" within the block they are defined. This is because they are allocated on the stack and are popped off when the block exits. If we want to pass return values from a method, we need to supply a return value or pass variables by reference (through a class). This lecture will explain in more detail about where variables may be accessed depending on where they are defined.

Scoping

In the last lecture, we said that local variables are those declared within a block or those passed as value parameters. The block does not have to be the body of a method—local identifiers can be declared within *any* block. In a large program, there may be several variables with the same identifier— *name*, for example. How do we know which variable is meant and where each variable is accessible? The answers to these questions are provided by scope rules. Java uses a type of scoping called **static scoping**. It is called static because the scope of identifiers is determined at compile time. In contrast, some languages (like Lisp) use dynamic scope, where the scope is determined at runtime.

There are two categories of scope for an identifier in Java that are generally used: class scope, and local scope. Both adhere to the same basic rule: **a variable is accessible everywhere within the set of curly braces where it is declared, including code within nested curly braces**.

When a variable is declared within the class, it is accessible from all other methods in that class. These are either class or instance variables (we'll describe the difference in a moment):

```
public class Foo
{
        private int var;
        public int Method1()
        {
                …   // var accessible anywhere here
        }
        …
        public int MethodN()
        {
                … // var accessible anywhere here
        }
}

public class Foo2
{
        var defined in class Foo not accessible here
}
```

Note that variables defined in the class give us a means to share information among different methods in the class. Any value stored in these variables is lost when the object is destroyed (e.g., a variable referencing it goes away). We can also control access to these variables with the public and private modifiers.

We do not have the public and private modifiers when a variable is declared within a method. These are variables with local scope; they are accessible only within the braces which they are defined, and disappear when method exits. This means that variables defined in a method are not directly accessible outside that method. This includes variables passed as parameters:

```
public class Foo
{
        public int Method1(int x1)
        {
                int x2;
                // only x1 and x2 accessible here
                // destroyed when Method1 exits
        }
        …
        public int MethodN(String s1, String s2)
        {
                // only s1 and s2 accessible here
                // destroyed when MethodN exits
        }
}

public class Foo2
{
        // No variables from Foo's methods accessible here
}
```

In normal usage, variable scoping is as simple as defined above. However, things get trickier when variables have the same name. For example, consider the following scenario:

```
public class Foo
{
        private int x;
        public int Method1()
        {
                int x;
                x = 10;         // Which x is changed?
        }
}
```

In this example we have two variables named x. One has class scope, the other has local scope. Which variable is referenced when there is this ambiguity?

The rule used in Java is that the scope of a variable begins with its most recent declaration. This means that local variables take precedence over class variables. In the example above, the local variable is changed while the class variable remains unchanged. Here is an example:

```
public class OutputTest
{
  public int a;

  public void DoIt()
  {
     int a;                       // Local variable a
     a=3;
     System.out.println(a);  // Outputs 3
     {                            // Sub-block
            a = 5;              // Same local variable a
     }
     System.out.println(a);  // Outputs 5
     return;                      // Local "a" destroyed
  }

  public static void main(String[] args)
  {
     OutputTest x = new OutputTest();
     x.a = 100;
     x.DoIt();
     System.out.println(x.a);  // Outputs 100
  }
}
```

The output of this program is

```
3
5
100
```

Inside main, we only have access to class variable a. Inside DoIt, we reference the local variable a while the class variable remains unchanged.

Java does not allow us to have multiple local variables with the same name within the same scope. The following is illegal;

```
public void DoIt()
{
    int a;                  // Local variable a
    a=3;
    {                       // Sub-block
        int a;              // NOT ALLOWED
        a = 5;              // Same local variable a
    }
}
```

However, Java does allow us to have multiple sub-blocks with the same local variable name. Each local variable is unique only to its block:

```
public class OutputTest
{
  public int a;
  public void DoIt()
  {
    int i=0;
    {                       // Sub-block
        int a;
        a = 1;
        System.out.println(a);
    }                       // "a" destroyed
    {
        int a;
        a = 2;
        System.out.println(a);
    }
    while (i<1)
    {   // More typical formation of sub-block
        int a=3;
        System.out.println(a);
        i++;
    }
  }

  public static void main(String[] args)
  {
    OutputTest x = new OutputTest();
    x.DoIt();
  }
}
```

The output of this program is:

```
1
2
3
```

Each variable “a” is defined and exists only within its sub-block. This means the variable “a” is created and destroyed three times in this example.

The keyword “this”

From these examples, it may seem like it is impossible to access a variable defined in the class if it has the same name as a local variable. This is true if we just use the variable name. However, Java provides a keyword called “this” that refers to the current object. From “this” we can access class variables. For example:

```
public class OutputTest
{
  public int a;
  public void DoIt()
 {
     int a;                    // Local variable a
     a=3;
     System.out.println(a);  // Outputs 3
     this.a = 50;              // Changes class instance variable
     return;                   // Local "a" destroyed
  }

  public static void main(String[] args)
 {
     OutputTest x = new OutputTest();
     x.a = 100;
     x.DoIt();
     System.out.println(x.a);  // Outputs 50
  }
}
```

This program outputs:

```
3
50
```

The class variable is changed from the DoIt method when we preface the variable with “this” rather than the local variable.

If class identifiers are accessible to all methods, why don't we just make all identifiers global and do away with parameter lists? Good programming practice dictates that communication between the modules of our program should be explicitly stated. This practice limits the possibility of one module accidentally interfering with another. In other words, each method is given only what it needs to know. This is how we write "good" programs.

Consider a large project where many programmers are working on different sections of the program. If everyone used class variables, each programmer would have to be extremely careful not to pick a variable name that another programmer was using, or the program might not compile or may produce buggy output. The existence of local variables avoids this problem by allowing each programmer to define and have access to their own “private” set of variables that will never conflict with someone else’s code.

Static Variables

Previously we discussed static methods. Static methods are defined and exist within the class definition, or blueprint. This means that there is one and only one method, and this means that we don’t have to create an instance of the class to reference that method.

We can do the same thing with variables. If we add the keyword **static** in front of a variable defined in the class, then this becomes a static class variable. There is only one copy of this variable that is shared among all instances of the class.

Static variables are useful when we need to share the same information among all instances of a class. A method that is defined as static only has access to static class variables. Here is an example that illustrates the use of static variables. We would like a counter to keep track of how many times instances of class Foo have been created. Each time the class is created, in the constructor we increment the static counter. Each individual instance has access to this counter as well as an individual ID number:

```
public class OutputTest
{
  private static int counter = 0;       // # of instances created
  private int num;                      // Which number am I?

  // Constructor keeps track of number of times created
  public OutputTest()
  {
   counter++;         // Increment “global” static variable
   num = counter;     // Set local instance to “global” value
  }

  // Print out our information
  public void PrintInfo()
  {
        System.out.println("I am instance " + num +
           " of " + counter);
  }
```

```
     public static void main(String[] args)
     {
      OutputTest i1 = new OutputTest();
      i1.PrintInfo();
      OutputTest i2 = new OutputTest();
      OutputTest i3 = new OutputTest();
      i3.PrintInfo();
      OutputTest i4 = new OutputTest();
      i3.PrintInfo();
      i4.PrintInfo();
     }
   }
```

The output from this program is:

```
      I am instance 1 of 1
      I am instance 3 of 3
      I am instance 3 of 4
      I am instance 4 of 4
```

Each time we create a new instance, the static counter is increased by one. This information is accessible to each class.

Method Overloading

Let's say that we want to write a method, AbsoluteValue, that can operate on integers, floats, and doubles and return the absolute value of the number that is passed in. Using what we know so far we would have to write three separate functions:

```
public class Absolutes
{
      public static int AbsoluteValueInt(int x)
      {
            if (x<0) return (-1*x);
            return x;
      }

      public static float AbsoluteValueFloat(float x)
      {
            if (x<0) return (-1*x);
            return x;
      }

      public static double AbsoluteValueDouble(double x)
      {
            if (x<0) return (-1*x);
            return x;
      }
}
```

From main, we could invoke these methods as:

```
public static int main(String[] args)
{
	int x=-3;
	float f=(float) 34.5;
	double d=-3.812312;

	System.out.println(Absolutes.AbsoluteValueInt(x));
	System.out.println(Absolutes.AbsoluteValueFloat(f));
	System.out.println(Absolutes.AbsoluteValueDouble(d));
}
```

We would have to write a separate function for each data type, and give each function a different name to distinguish one from the other. Fortunately, Java supports a feature called method overloading that lets us assign the same name to methods that have different parameter types. The compiler is able to distinguish which function we want to use by matching up the data types in the parameters. For example we could write:

```
public static int AbsoluteValue(int x)
{
	if (x<0) return (-1*x);
	return x;
}

public static float AbsoluteValue(float x)
{
	if (x<0) return (-1*x);
	return x;
}

public static double AbsoluteValue(double x)
{
	if (x<0) return (-1*x);
	return x;
}

public static int main(String[] args)
{
	int x=-3;
	float f=(float) 34.5;
	double d=-3.812312;

	System.out.println(Absolutes.AbsoluteValue(x));
	System.out.println(Absolutes.AbsoluteValue(f));
	System.out.println(Absolutes.AbsoluteValue(d));
}
```

In this case, we have named all of the functions the same thing, “AbsoluteValue”. The compiler is able to invoke the proper function call by matching up the parameters. You might notice that all of the AbsoluteValue methods are basically the same except the data type. In a future class we’ll discuss **templates** that allow us to write the method only once.

As long as there is a unique set of parameters, the compiler can make the distinction. For example, the following would not be valid:

```
public static float AbsoluteValue(float x)
{
      if (x<0) return (-1*x);
      return x;
}

public static double AbsoluteValue(float x)
{
      if (x<0) return (-1*x);
      return x;
}
```

In this case, even though both methods return different data types, they both expect the same parameters. Consequently, the compiler won't be able to distinguish one from the other, resulting in a compile-time error.

Note that overloading also works if we don't have the same number of parameters. The following functions can be overloaded successfully based on the different number (as opposed to type) of parameters:

```
int foo(int x)
{
      return 3;
}

int foo(int x, int y)
{
      return 4;
}
```

A function making the invocation of either foo(x) or foo(x,y) will successfully invoke the proper method.

Packages and Importing

A package is a collection of classes that have been grouped together and given a name. Generally the classes are all related in some way (e.g. utilities, math function, etc.) To give a file a package name, each file should have as the first line:

    package name_of_package;

The name of the package is generally all lowercase letters. For example:

    package mystuff.coolcode;

To use the classes in a package elsewhere, use the import statement at the top of the file (which we've already been using for various predefined Java classes):

```
import mystuff.coolcode.*;
```

The above would give us access to all classes in the package "mystuff.coolcode". If we wanted to be more specific we could identify specific classes:

```
import mystuff.coolcode.animation;
import mystuff.coolcode.funstuff;
```

The package name is not an arbitrary identifier, but tells the compiler where to find the classes on the disk. Each "." Corresponds to an actual directory:

```
mystuff.lib.math
```

Would be located in directory mystuff/lib/math. These directories are relative to the current working directory or to the CLASSPATH directory, which you can set as an environment variable (e.g. to c:\jdk\lib)

NetBeans lets you create packages quite easily, and by default will ask you to put your classes into packages.