# FIT ICT Software Development

Lecture 6

Lecturer : Charles Tyner


- Scope and Duration
- Modifiers
- Method Overloading

# Scope and Duration

# Automatic Variables

So far, we have used identifiers for variable names.

The attributes of variables include:

- **name**
- **type**
- **size**
- **value**

For eg String name1 = “Tom”;

Each identifier in a program also has other attributes:

- **Scope**
- **Duration**

# Scope

An identifiers **scope** is where the identifier can be referenced in a program, ie where the program can use the variable.

- Some identifiers can be references throughout a program.
- Others can be referenced from only limited portions of a program.

## Scope Rules

The scope of an identifier is the portion of the program in which the identifier can be referenced.

A local variable declared in a block can be referenced only in that or blocks nested within that block.

The scopes for an identifier are:

- class scope
- block scope.

## Class Scope

Methods and instance variables of a class have class scope.

Class scope begins at the opening left brace "{" of the class definition and terminates at the closing right brace "}" of the class definition.

Class scope enables methods of a class to **directly invoke all methods defined in the same class or inherited into the class** (such as methods inherited from the Applet class) and to directly access all instance variables defined in the class.

All instance variables and methods of a class can be considered global to the methods of the class in which they are defined.

The methods can modify the instance variables directly and invoke other methods of the class.

# Block Scope

An identifier declared inside a block has block scope.

Block scope begins at the identifier's declaration and ends at the terminating right brace "}" of the block.

Local variables declared at the beginning of a method have block scope as do method parameters, which are also local variables of the method.

Any block may contain variable declarations.

If a local variable in a method has the same name as an instance variable, the instance variable is "hidden" until the block terminates execution.

# Block Scope

## Let's look at an example:

```
// An example of calling a method to draw stars

public class DrawStarsApp
{
   public static void main( String args[] )
   {

       // call method to draw stars
       drawstars();

       // print blank line
       System.out.println(" ");

       System.out.println("Hello World");

       // call method to drawstars
       drawstars();

       // print blank line
       System.out.println(" ");

   }


   // method to draw stars
   public static void drawstars()
   {
       // THIS VARIABLE HAS BLOCK SCOPE
       int j;

       for (j =1 ; j < 50; j = j + 1)
       {
           System.out.print("*");
       }
   }

}
```

→ Variable **j** in method drawstars has **BLOCK SCOPE**

# Duration

An identifiers **duration** (also called its **lifetime**) determines the period during which that identifier exists in memory.

- Some identifiers exist briefly
- Some are repeatedly created and destroyed.
- Others exist for the entire execution of a program.

# Duration

Identifiers that represent **local variables in a method** (parameters and variables declared in the method body) have **automatic duration**.

Automatic duration variables are created when program control enters the block in which they are declared.

- They exist while the block is active
- They are **destroyed** when the block is exited.

**Variables of automatic duration are referred to as automatic variables.**

# Duration

Let's return to our previous example:

```
// An example of calling a method to draw stars

public class DrawStarsApp
{
   public static void main( String args[] )
   {

       // call method to draw stars
       drawstars();

       // print blank line
       System.out.println(" ");

       System.out.println("Hello World");

       // call method to drawstars
       drawstars();

       // print blank line
       System.out.println(" ");

   }


   // method to draw stars
   public static void drawstars()
   {
       // THIS VARIABLE j HAS BLOCK SCOPE
       // THIS VARIABLE j HAS AUTOMATIC DURATION
       int j;

       for (j =1 ; j < 50; j = j + 1)
       {
           System.out.print("*");
       }
   }

}
```

→ The variable **j** has **Automatic Duration**

# Duration

**Automatic duration** is a means of conserving memory because automatic duration variables are created when the block is which they are created is entered and are destroyed when the block is exited.

Automatic duration is an example of the **principle of least privilege**.

This principle states that each component of a system should have sufficient rights and privileges to accomplish its designated task, but no additional rights or privileges. This helps prevent accidental and/or malicious errors from occurring in systems.

The compiler automatically initialises the instance variables of a class if the programmer does not provide initial values.

Variables of the primitive data types are initialised to **zero** except boolean variables which are initialised to **false**.

References are initialised to **null**.

# Static Duration

Java also has identifiers of static duration.

Variables and methods of static duration exist from the point at which the class that defines them is loaded into memory for execution until the program terminates.

For static duration variables, storage is allocated and initialised once their classes are loaded into memory.

For static duration methods, the names of the methods exist when their classes are loaded into memory.

# Program Example

```
1  // Fig. 5.9: Scope.java
2  // Scope class demonstrates field and local variable scopes.
3
4  public class Scope
5  {
6     // field that is accessible to all methods of this class
7     private static int x = 1;
8
9     // method main creates and initializes local variable x
10    // and calls methods useLocalVariable and useField
11    public static void main( String[] args )
12    {
13       int x = 5; // method's local variable x shadows field x
14
15       System.out.printf( "local x in method begin is %d\n", x );
16
17       useLocalVariable(); // useLocalVariable has local x
18       useField(); // useField uses class Scope's field x
19       useLocalVariable(); // useLocalVariable reinitializes local x
20       useField(); // class Scope's field x retains its value
21
22       System.out.printf( "\nlocal x in method begin is %d\n", x );
23    } // end method begin
24
```

**Fig. 5.9** | Scope class demonstrating scopes of a field and local variables. (Part 1 of 3.)

# Program Example(Contd.)

```
25      // create and initialize local variable x during each call
26      public static void useLocalVariable()
27      {
28         int x = 25; // initialized each time useLocalVariable is called
29
30         System.out.printf(
31            "\nlocal x on entering method useLocalVariable is %d\n", x );
32         ++x; // modifies this method's local variable x
33         System.out.printf(
34            "local x before exiting method useLocalVariable is %d\n", x );
35      } // end method useLocalVariable
36
37      // modify class Scope's field x during each call
38      public static void useField()
39      {
40         System.out.printf(
41            "\nfield x on entering method useField is %d\n", x );
42         x *= 10; // modifies class Scope's field x
43         System.out.printf(
44            "field x before exiting method useField is %d\n", x );
45      } // end method useField
46   } // end class Scope
```

**Fig. 5.9** | Scope class demonstrating scopes of a field and local variables. (Part 2 of 3.)

# Program Result

```
local x in method begin 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 method begin is 5
```

**Fig. 5.9** | Scope class demonstrating scopes of a field and local variables. (Part 3 of 3.)

# Modifiers

Java provides modifiers to control access to data, methods and classes.

**static**

Defines data and methods. It represents class-wide information that is shared by all instances of the class.

**public**

Defines classes, methods, and data in such a way that all programs can access them.

**private**

Defines methods and data in such a way that the declaring class can access them, but not any other class.

# Method Overloading

## Method overloading

Java enables several methods of the same name to be identified as long as these methods have different sets of parameters.

- based on the number of parameters,
- based on the types of parameters,
- based on the order of parameters,

This is called method overloading.

## Method Overloading

When an overloaded method is called, … the Java compiler selects the proper method by examining the **number, types and order** of the arguments in the call.

Method overloading is commonly used to create several methods of the same name that perform **similar tasks**, but on different data

Overloading methods that perform closely related tasks can make programs more reusable and understandable.

## Method Overloading

- Methods of the same name can be declared in the same class, as long as they have different sets of parameters
  - Called method overloading
  - Java compiler selects the appropriate method to call by examining the number, types and order of the arguments in the call
- Used to create several methods that perform the same or similar tasks on different types or different numbers of arguments

```
1  // Fig. 5.10: MethodOverload.java
2  // Overloaded method declarations.
3
4  public class MethodOverload
5  {
6     // test overloaded square methods
7     public static void main( String[] args )
8     {
9        System.out.printf( "Square of integer 7 is %d\n", square( 7 ) );
10       System.out.printf( "Square of double 7.5 is %f\n", square( 7.5 ) );
11    } // end method testOverloadedMethods
12
13    // square method with int argument
14    public static int square( int intValue )
15    {
16       System.out.printf( "\nCalled square with int argument: %d\n",
17          intValue );
18       return intValue * intValue;
19    } // end method square with int argument
20
```

**Fig. 5.10** | Overloaded method declarations. (Part 1 of 2.)

```
21      // square method with double argument
22      public static double square( double doubleValue )
23      {
24         System.out.printf( "\nCalled square with double argument: %f\n",
25            doubleValue );
26         return doubleValue * doubleValue;
27      } // end method square with double argument
28   } // end class MethodOverload
```

```
Called square with int argument: 7
Square of integer 7 is 49

Called square with double argument: 7.500000
Square of double 7.5 is 56.250000
```

**Fig. 5.10** | Overloaded method declarations. (Part 2 of 2.)

# A note on Strings

# Processing Strings

• A string is a sequence of characters

• It is denoted by double quote marks, eg “This is a string”

## Declaring Strings

String variables are declared as follows:
String s1 = “This is a string”;
String s2;

## String Handling Methods

### Equality

| | |
|---|---|
| s1.equals(s2) | This method returns true if s1,s2 contain the same sequence of characters, otherwise it returns false. |

## Example

```
class L2EX2
{
  public static void main(String[] args)
  {
     String s1 = "TESTSTRING";
     String s2 = "TESTSTRING";

     if(s1.equals(s2))

            System.out.println("I love rock music");
     else

            System.out.println("Programming is cool");

}
}
```

## String Handling Methods

| | |
|---|---|
| s1.length() | This method returns the number of characters in the string s1 |

# Week 6 Lab Exercise

- For this week in the lab students should type up all the code from lecture 6 and compile and run the code.
- Students should ensure that they completely understand the topics covered this week in the lecture including the code examples that relate to the topics.
- Students should refresh their memories by reading up on arrays before our next lecture.