JAVA PROGRAMMING 1

Summer 2018 - Christian Hur

# Unit 4 Lecture - More Object Concepts

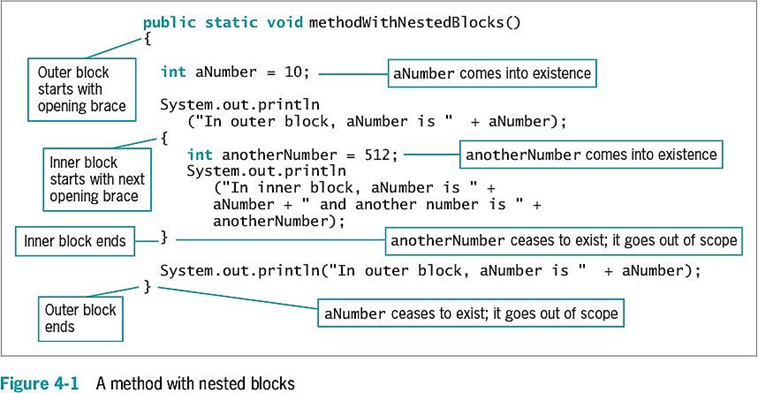
Reading: Chapter 4

Chapter 4 introduces blocks and scope, overriding, and overloading. You will learn to create overloaded methods and constructors. You will learn about static variables and how to create constants using the final keyword. Finally, you will learn to use prewritten classes in the java.lang and java.util packages, and other packages.

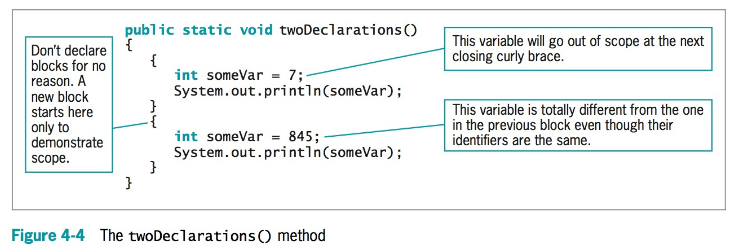
## Understanding Blocks and Scope

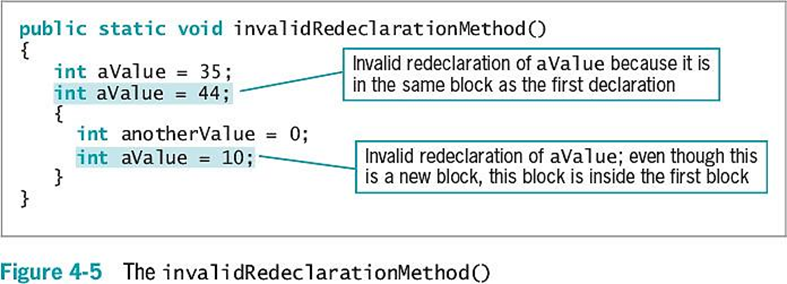
A code **block** is any code that is contained within a pair of curly braces. Blocks can reside within a class or method and they can be nested inside other blocks forming **outer blocks** and **inner blocks**.

Example: Nested blocks - local variables

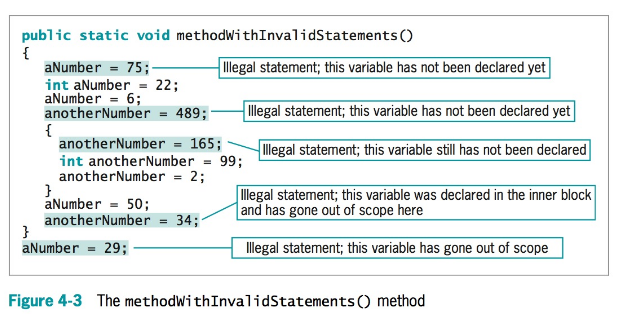


**Scope** refers to the area in which a data item (i.e. a variable) is visible to a program, and in which you can refer to it using its simple identifier. A variable can have global or class level scope, local scope (within a method), block scope (within a pair of curly braces). In Java, a variable with block-level scope can be declared in the class space (class level) or within a method (local). A variable **comes into scope** (existence) when it’s declared and **goes out of scope** (cease to exist, is destroyed) at the end of the code block.



Variables cannot be redeclared more than once within a block. The following is illegal:  
  


You cannot use a data item that is not in scope.



## 

## 

# Overrides

Overrides refers to how a locally declared variable always masks or hides another variable with the same name elsewhere in the class. When you use the variable’s name within the method in which it is declared, it takes precedence over, or **overrides**, any other variable with the same name in another method.

Example:

public static void main(String[] args) {

int number = 10;

System.out.println(number); //result ==> 10

display(number);

}

public static void display(int n)

{

int number = 55 + n;

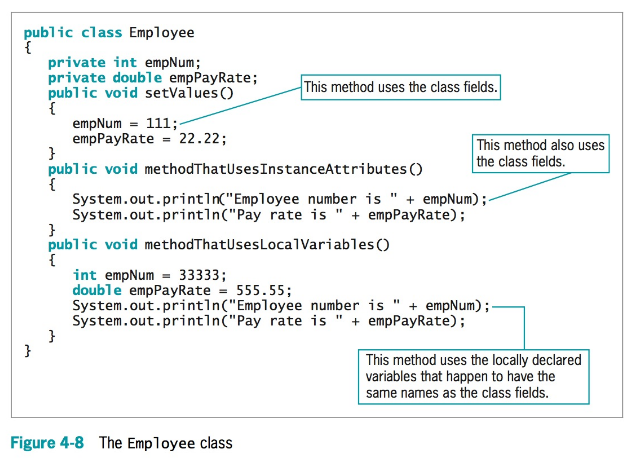
System.out.println(number); //result ==> 55

n = 34;

}

# Shadowing

Shadowing is the phenomenon in which a variable that hides another shadows it. When variables have the same name, variables within methods override or hide the class’s fields.



# Overloading a Method

**Overloading** a method allows you to use one identifier to execute diverse tasks. In Java, it more specifically means writing multiple methods in the same scope that have the same name but different parameter lists (aka different **signatures**). In overloaded methods, the parameter identifiers do not have to be different, but the parameter lists must satisfy one or both of these conditions:

* The lists must have different numbers of parameters. For example, one list could have one double, another list could have two doubles, and a third list could have 10 doubles.
* The lists must have parameter data types in different orders. For example, one list could have two **doubles**, another could have an **int** and a **double**, and a third could have a **double** and an **int**.

When you overload a Java method, multiple methods share a name, and the compiler understands which one to use based on the arguments (signatures) in the method call.

Examples:

public static void main(String[] args) {

int number = 10;

System.out.println(number); //result ==> 10

display();

}

//main method - no argument

public static void display()

{

int number = 55;

System.out.println(number); //result ==> 55

display(number); //pass an integer to a double (auto-type promotion)

}

//main method - one argument of double type

public static void display(double a)

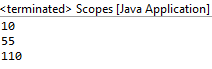
{

int number = (int)(a \* 2); //type-cast

System.out.println(number); //result ==> 110

}

Result:



## 

## 

# Ambiguity

When you overload methods, you risk creating an ambiguous situation—one in which the compiler cannot determine which method to use.

Example 1:

//display method - one argument of double type

public static void display(double a)

{

int number = (int)(a \* 2);

System.out.println("Double: " + number);

}

//display method - one argument of int type

public static void display(int a)

{

int number = a \* 3;

System.out.println("Int: " + number);

}

Example 2:

public static void display(int a, double b)

{

double number = a \* b;

System.out.println("I-D: " + number); //result ==> 110

}

public static void display(double a, int b)

{

double number = b \* 3;

System.out.println("D-I: " + number); //result ==> 110

}

display(1, 2.2); //OK

display(1.5,.2); //OK

display(1, 2); //Problem - ambiguous

The compiler can’t determine which version of the **display**() method to use, and the program does not compile.

Methods with identical names that have identical parameter lists but ***different return types are not overloaded***—they are illegal.

Example:

public **int** aMethod(int x);

public **void** aMethod(int x); //illegal - does not compile

The compiler determines which of several versions of a method to call based on the arguments (signatures) in the method call, and does not consider the return type.

# Constructor Overloads

As with any other method, you can overload constructors. Overloading constructors provides you with a way to create objects with different initializing arguments, or none, as needed. As long as the constructor parameter lists differ, the constructors are not ambiguous.

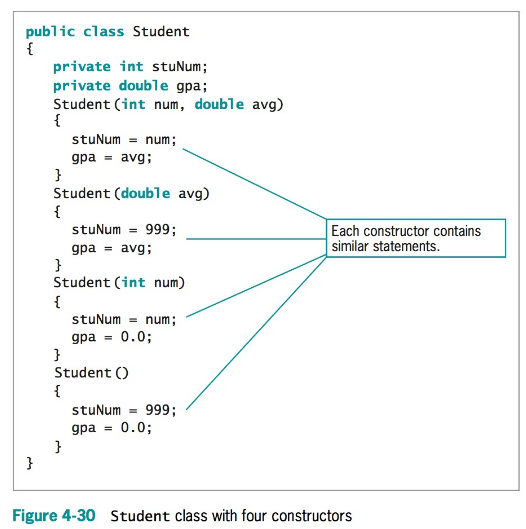
# The **this** reference

You learned that if a field or method name is preceded by the keyword static when it is declared, only one field or method exists, no matter how many objects are instantiated. In other words, if a field is static, then only one copy of the field exists, and all objects created have the same value for that field.

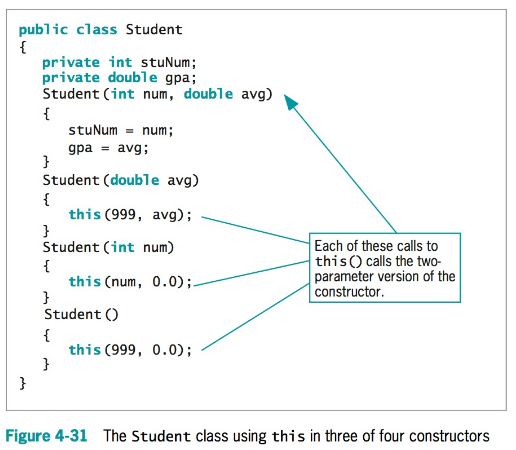
When you create a method that uses a nonstatic field value for a class—for example, to get or set the field value—the method must be nonstatic. That means it performs in a different way for each object.

The compiler accesses the correct object’s field because every time you call a nonstatic method, a **reference**—an object’s memory address—is implicitly understood. The reference to an object that is passed to any object’s nonstatic method is called the **this** reference. Only nonstatic, instance methods have a **this** reference.

# Using the this Reference to Make Overloaded Constructors More Efficient



You can write each constructor to call one master constructor to save coding and reduce the chance for errors.



# The Math Class

The class java.lang.Math contains constants and methods that you can use to perform common mathematical functions. All of the constants and methods in the Math class are static—they are class variables and class methods. In other words, you do not create any Math objects when you use the class.

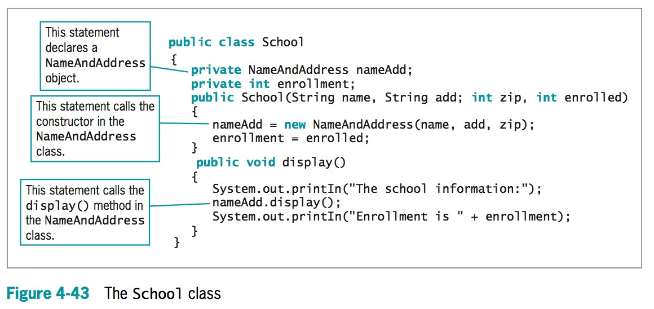
areaOfCircle = Math.PI \* radius \* radius;

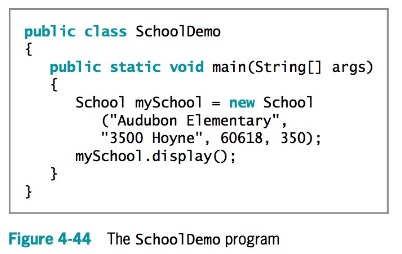
See Table-1 (page 222)

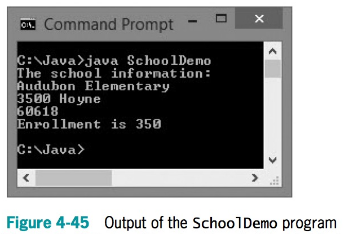
# Composition (has-a relationship)

The fields in a class can be simple data types like int and double, but they can also be class types. **Composition** describes the relationship between classes when an object of one class is a data field within another class. You have already studied many classes that contain String object fields. These classes employ composition. The relationship created is sometimes called a **has-a** relationship because one class “**has an**” instance of another.

|  |
| --- |
| Note: When a class is a subclass of a base class, the relationship created is called an **is-a** relationshiop because the subclass “**is a**” child class of the base class (parent class) - inheritance. |







# Nested Classes - not recommended

* Every class you have studied so far has been stored in its own file, and the filename has always matched the class name. In Java, you can create a class within another class and store them together; such classes are nested classes. The containing class is the top-level class. There are four types of nested classes:
* **static member classes**: A static member class has access to all static methods of the top-level class.
* **Nonstatic member classes**, also known as inner classes: This type of class requires an instance; it has access to all data and methods of the top-level class.
* **Local classes**: These are local to a block of code.
* **Anonymous classes**: These are local classes that have no identifier.