CSS 142

Lecture 12

Arkady Retik aretik@uw.edu

TODAY'S CONTENT

1. **HW feedback**
2. **Quiz**
3. **Classes: Methods, Constructors**
   - **Pass by value vs pass by reference**

Key topic

4. **Intro to Arrays**
5. **HoA : 5 <optional> ???**

If time allows

Reading Wedn

- **Arrays 6.1; 6.2**

HW4

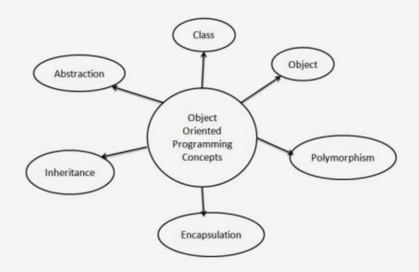
Average Score:	27.48
High Score:	30
Low Score:	18.5
Total Graded Submissions:	42 submissions
Median score:	28.5

CLASSES

Quiz: refresher

A short quiz to refresh and check

your reading of Chapter 4



- **Two fundamental blocks of a program are…**
  - Data and Methods
- **OOP –**
  - reasoning about a program as a set of objects rather than a set of actions
- **Object –**
  - a programming entity that contains state (data) and behavior (methods)
- **State** -
  - a set of values stored in an object.
- **Behavior –**
  - a set of actions an object can perform, i.e. reporting or modifying its internal state.

Classes vs Objects

- write your answer down:

- One is the cookie-cutter, the other - the cookies
  - which one is a cookie?

- One is a text file; the other a chunk of RAM
  - which one is a text file?
  - which one is persistent?

True/False

- An object of class A is an instance of class A.
    - True
- In a method invocation, there must be exactly the same number of arguments in parentheses as there are formal parameters in the method definition heading.
    - True
- Inside a Java method definition, you can use the keyword ***`this`*** as a name for the calling object.
    - True
- Boolean expressions may be used to control ***if-else*** or ***while*** statements.
    - True
- The modifier ***private*** means that an instance variable can be accessed by name outside of the class definition.
    - False

- It is considered good programming practice to validate a value passed to a mutator method before setting the instance variable.
    - True
- Mutator methods can return integer values indicating if the change of the instance variable was a success.
    - False
- Method overloading is when two or more methods of the same class have the same name but differ in number or types of parameters.
    - True
- Java supports operator overloading.
    - False
- Only the default constructor has the `this` parameter.
    - False

Multiple Choice

- The ***new*** operator:
  - allocates memory
  - is used to create an object of a class
  - associates an object with a variable that names it.
  - all of the above.
  - none of the above

  A : D

- A method that performs some action other than returning a value is called a __________ method.
  - null
  - void
  - public
  - private

  A : B

- The body of a method that returns a value must contain at least one _________ statement.
  - void
  - invocation
  - throws
  - return

  A : D

- A variable whose meaning is confined to an object of a class is called:
  - instance variable
  - local variable
  - global variable
  - none of the above
    - A

- A variable whose meaning is confined to a method definition is called an/a
  - instance variable
  - local variable
  - global variable
  - none of the above
    - B

- In Java, a block is delimited by:
  - ( )
  - /* */
  - “ “
  - { }
    - D

- In Java, call-by-value is only used with:
  - objects
  - primitive types
  - this
  - all of the above
    - B

- The parameter ***this*** refers to
  - instance variables
  - local variables
  - global variables
  - the calling object
    - D

- When you want the parameters in a method to be the same as the instance variables you can use the ______ parameter.
  - String
  - hidden
  - default
  - this
    - D

- Two methods that are expected to be in all Java classes are:
  - getName and setName
  - toString and equals
  - compareTo and charAt
  - toLowerCase and toUpperCase
    - B

- A program whose only task is to test a method is called a:
  - driver program
  - stub
  - bottom-up test
  - recursive method
    - A

- Java has a way of officially hiding details of a class definition. To hide details, you mark them as _________.
  - public
  - protected
  - private
  - all of the above
    - C

- Accessor methods:
  - return something equivalent to the value of an instance variable.
  - promotes abstraction
  - both A and B
  - none of the above
    - C

- A _________ states what is assumed to be true when the method is called.
  - prescript
  - postscript
  - precondition
  - postcondition
    - C

- The name of a method and the list of ________ types in the heading of the method definition is called the method signature.
  - parameter
  - argument
  - return
  - primitive
    - A

Classes and Objects:

continue

CLASS: INTRO

- Classes are the most important language feature that make *object-oriented programming* (*OOP*) possible
- Programming in Java consists of defining a number of classes
  - Every program is a class
  - All helping software consists of classes
  - All programmer-defined types are classes
- Classes are central to Java

CLASS: INTRO

- You already know how to use **classes** and the **objects created from them**, and how to invoke their methods
  - For example, you have already been using the predefined `String`, `Scanner`, `Random`, `File` classes
- Now you will learn how to define your own classes and their methods, and how to create your own objects from them

A Class Is a Type

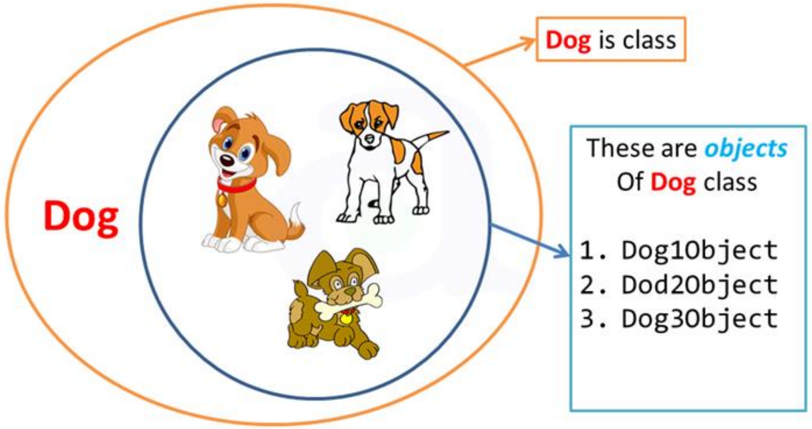
NB!

- A class is a special kind of programmer-defined **type**, and variables can be declared of a **class type**
- A value of a class type is called an **object** or ***an instance of the class***
- A class determines the **types of data** that an **object can contain**, as well as **the actions** it can perform

Primitive Type Values vs. Class Type Values

NB!

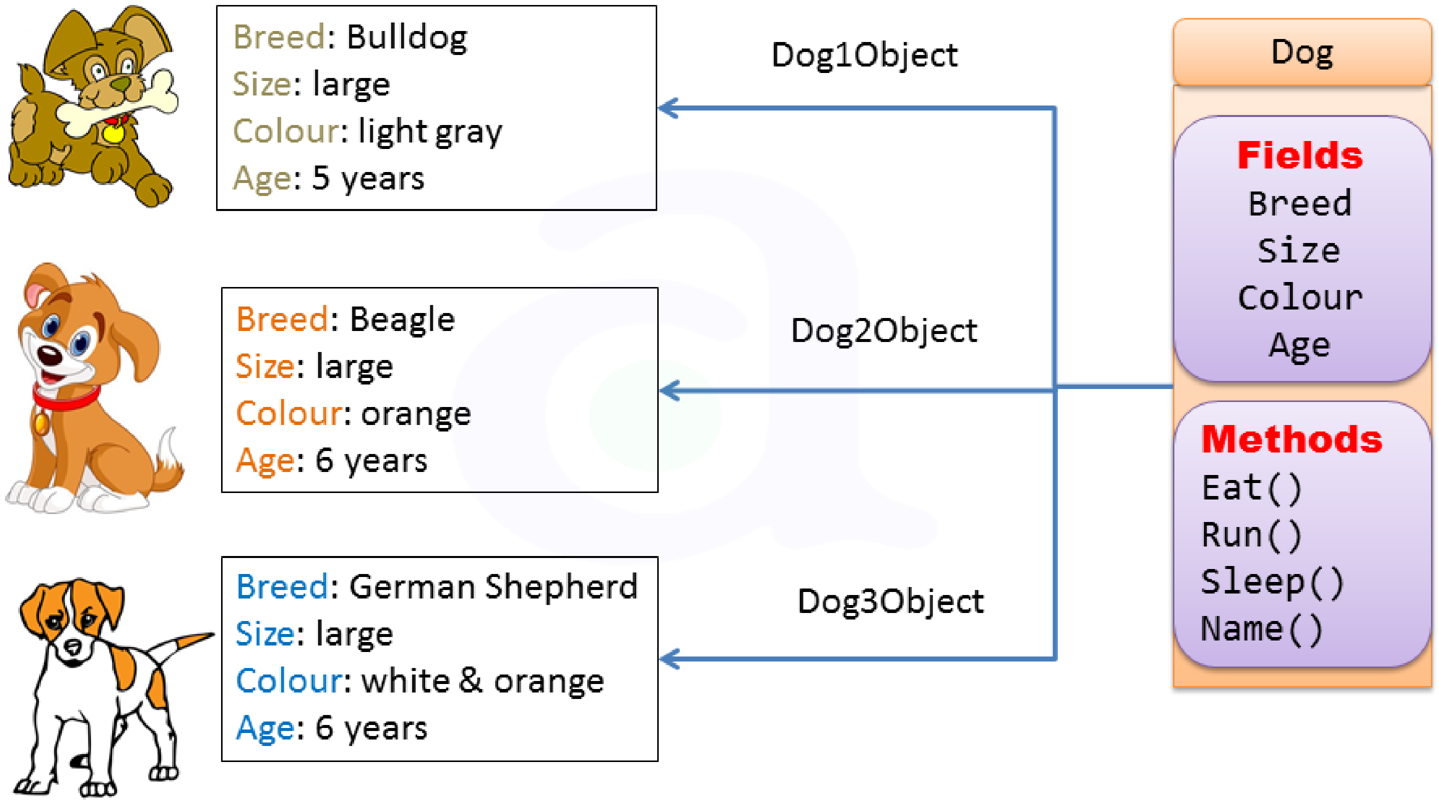
- A primitive type value is **a single piece of data**
- A class type value or object can have **multiple pieces of data**, as well as actions called ***methods***
  - All objects of a class have the same **methods**
  - All objects of a class have the same pieces of **data** (i.e., name, type, and number)
  - For a given object, each piece of data can hold a different value



2 mins:

Write down an examples for a Dog Class and one or two objects

A Class Is a Type



Breed: Bulldog
Size: large
Colour: light gray
Age: 5 years

Breed: Beagle
Size: large
Colour: orange
Age: 6 years

Breed: German Shepherd
Size: large
Colour: white & orange
Age: 6 years

Dog1Object

Dog2Object

Dog3Object

Dog

Fields
Breed
Size
Colour
Age

Methods
Eat()
Run()
Sleep()
Name()

The Contents of a Class Definition

- A class definition specifies the **data items** and **methods** that all of its objects will have
- These data items and methods are sometimes called *members* of the object
- Data items are called ***fields*** or ***instance variables***
- Instance variable declarations and method definitions can be placed in any order within the class definition

The `new` Operator

- An object of a class is named or declared by a variable of the class type:

```
ClassName  classVar;
```

- The `new` operator **must** then be used to create the object and associate it with its variable name:

```
classVar = new ClassName();
```

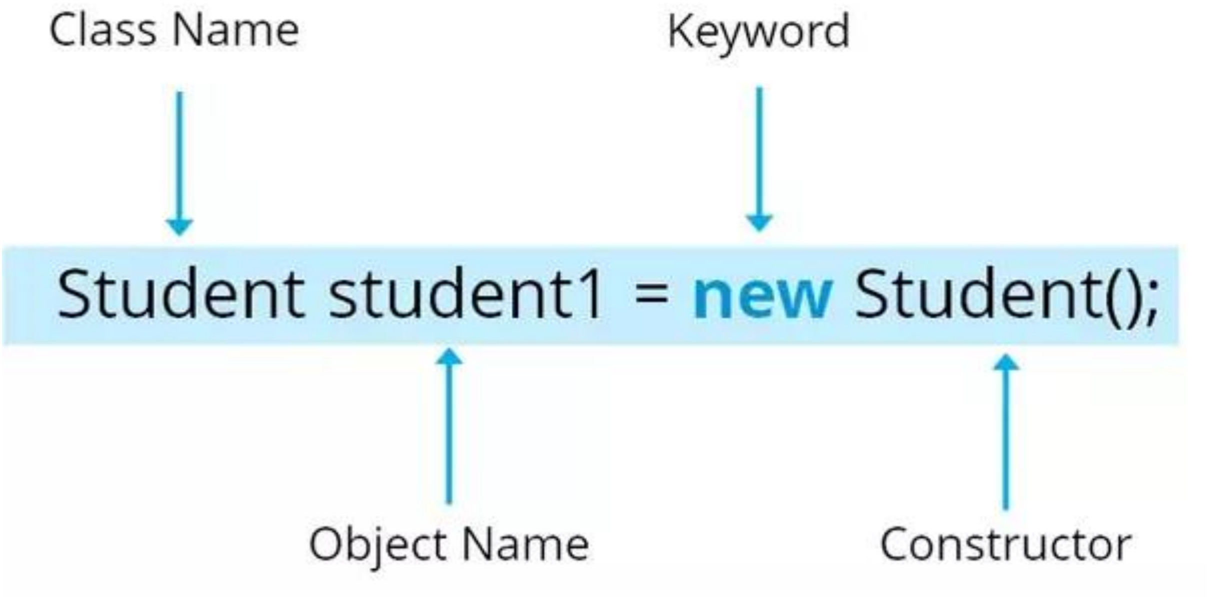
- These can be combined as follows:

```
ClassName classVar = new ClassName();
```

The **new** Operator

1 mins:

Write down two examples for a Class Date or Class Student we constructed last time



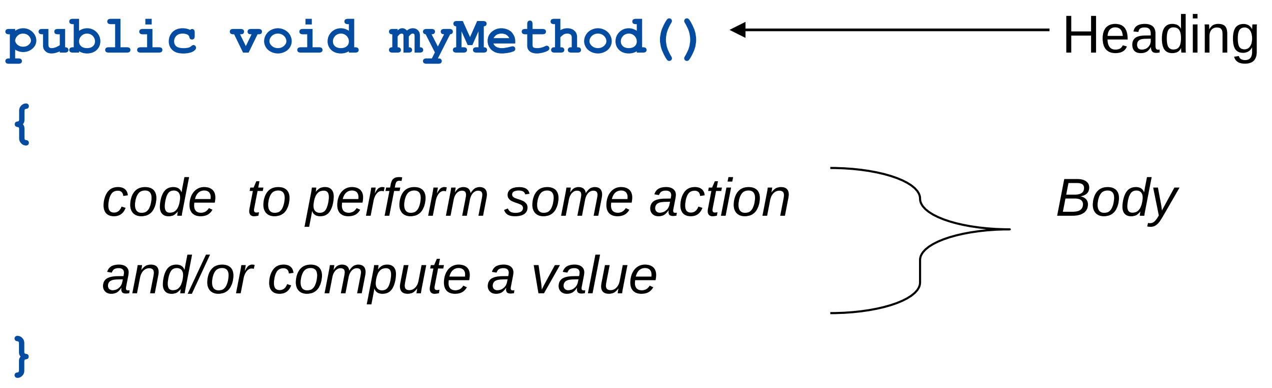
```
Student student1 = new Student();
```

Instance Variables and Methods

- Instance variables can be defined as in the following two examples
  - Note the `public` **modifier** (for now):
  - `public String  instanceVar1;`
  - `public int  instanceVar2;`
- In order to refer to a particular instance variable, preface it with its object name as follows:
  - `objectName.instanceVar1`
  - `objectName.instanceVar2`

Instance Variables and Methods

- Method definitions are divided into two parts: a *heading* and a *method body:*



```
public void myMethod()
{
      code  to perform some action
      and/or compute a value
}
```

- Methods are invoked using the name of the calling object and the method name as follows:

```
classVar.myMethod();
```

- Invoking a method is equivalent to executing the method body

File Names and Locations

- Reminder: a Java file must be given the same name as the class it contains with an added `.java` at the end
  - For example, a class named `MyClass` must be in a file named `MyClass.java`
- For now, your program and all the classes it uses should be **in the same directory or folder**

More About Methods

NB!

- There are two kinds of methods:
  - Methods that compute and return a value
  - Methods that perform an action
    - this type of method does not return a value, and is called a `void` method
- Each type of method differs slightly in how it is defined as well as how it is (usually) invoked

More About Methods

- A method that returns a value **must specify the type of that value in its heading**:

```
public typeReturned methodName(paramList)
```

- A `void` method uses the keyword `void` in its heading to show that it does not return a value :

```
public void methodName(paramList)
```

1 min

Write two examples

on method headings:

void and one that

returns value

`main` is a `void` Method

- A program in Java is just a class that has a `main` method
- When you give a command to run a Java program, the run-time system invokes the method `main`
- Note that `main` is a `void` method, as indicated by its heading:

```
public static void main(String[] args)
```

`return` Statements

- The body of both types of methods contains a list of declarations and statements enclosed in a pair of braces

```
public <void or typeReturned> myMethod()
{
      declarations
      statements
}
```

Body

`return` Statements

- The body of a method that returns a value must also contain **one or more** `return` statements
  - A `return` statement specifies the **value** returned and ends the method invocation:
  - `return Expression;`
  - `Expression` can be any expression that evaluates to something of the **type returned listed in the method heading**

1 min

Write two examples

`return` Statements

- A `void` method need **not** contain a `return` statement, unless there is a situation that requires the method to end before all its code is executed
- In this context, since it does not return a value, a `return` statement is used without an expression:
  - `return;`

Method Definitions

NB!

- An invocation of a method that returns a value can be used as an expression anyplace that a value of the **`typeReturned`** can be used:

```
typeReturned tRVariable;
tRVariable = objectName.methodName();
```

- An invocation of a **`void`** method is simply a statement:

```
objectName.methodName();
```

Any Method Can Be Used As a `void` Method

- A method that returns a value can also perform an action

- If you want the action performed, but do not need the returned value, you can invoke the method as if it were a `void` method, and the returned value will be discarded:
  - `objectName.returnedValueMethod();`

Local Variables

NB!

- A variable declared within a method definition is called a ***local variable***
  - All variables declared in the `main` method are local variables
  - All method parameters are local variables
- If two methods each have a local variable of the same name, they are still two entirely different variables

Global Variables

- Some programming languages include another kind of variable called a *global* variable
- The Java language does **not** have global variables

Blocks

- A *block* is another name for a compound statement, that is, a set of Java statements enclosed in braces, **{ }**
- A variable declared within a block is local to that block, and cannot be used outside the block
- Once a variable has been declared within a block, its name cannot be used for anything else within the same method definition

Declaring Variables in a `for` Statement

NB!

- You can declare one or more variables within the initialization portion of a `for` statement
- A variable so declared will be **local** to the `for` loop, and cannot be used outside of the loop
- If you need to use such a variable outside of a loop, then declare it outside the loop

Parameters of a Primitive Type

- The methods seen so far have had no parameters, indicated by an empty set of parentheses in the method heading
- Some methods need to receive additional data via a list of ***parameters*** in order to perform their work
    - These *parameters* are also called *formal parameters*

Parameters of a Primitive Type

NB!

- A parameter list provides a description of the data required by a method
  - It indicates the **number** and **types** of data pieces needed, the **order** in which they must be given, and the **local name** for these pieces as used in the method

```
public double myMethod(int p1, int p2, double p3)
```

Parameters of a Primitive Type

NB!

- When a method is invoked, the appropriate values must be passed to the method in the form of ***arguments***
  - Arguments are also called ***actual parameters***
- The number and order of the arguments **must exactly match** that of the parameter list
- The type of each argument must be compatible with the type of the corresponding parameter

```
int a=1,b=2,c=3;
double result = myMethod(a,b,c);
```

Parameters of a Primitive Type

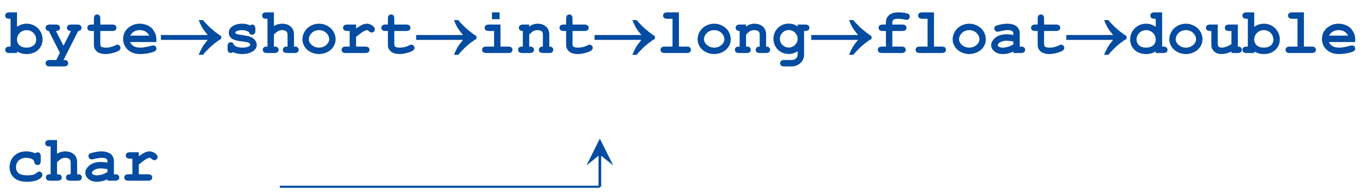
- In the preceding example, the value of each argument (not the variable name) is plugged into the corresponding method parameter
    - This method of plugging in arguments for formal parameters is known as the ***call-by-value** mechanism*

NB!

```
int a=1,b=2,c=3;
double result = myMethod(a,b,c);
```

Parameters of a Primitive Type

- If argument and parameter types do not match exactly, Java will attempt to make an automatic type conversion
  - In the preceding example, the **`int`** value of argument **`c`** would be cast to a **`double`**
  - A primitive argument can be **automatically type cast** from any of the following types, to any of the types that appear to its right:



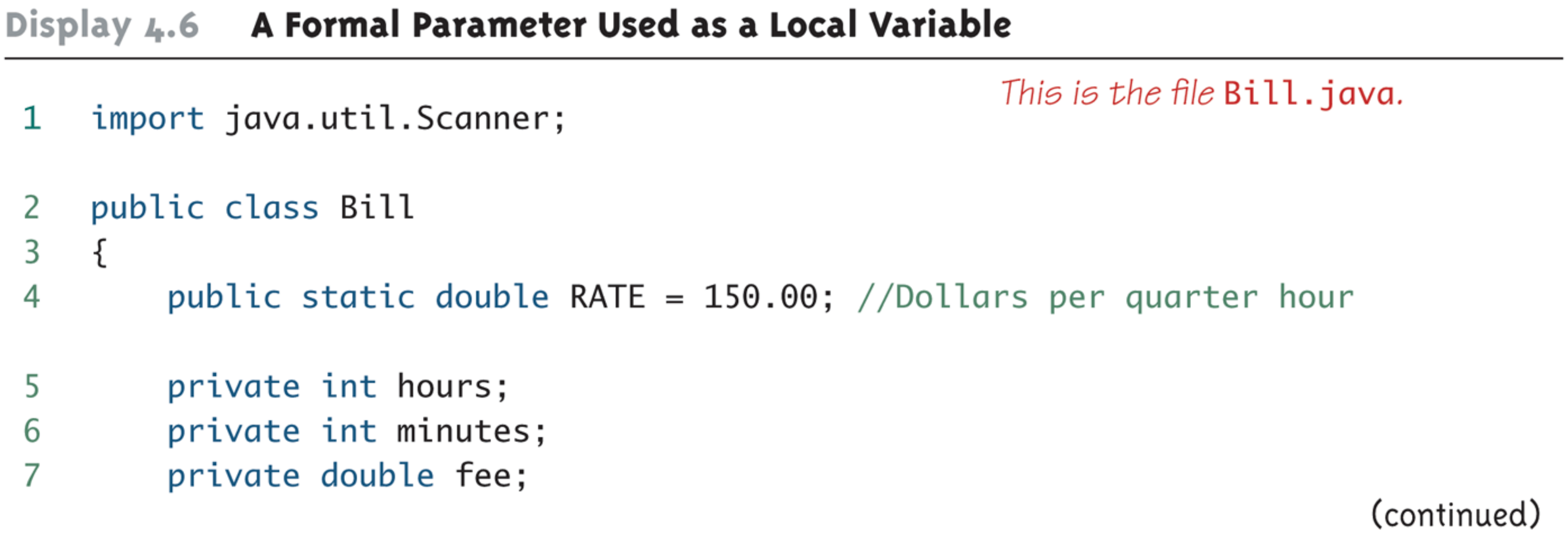
`byte→short→int→long→float→double`

`char`

Parameters of a Primitive Type

- A parameters is often thought of as a blank or placeholder that is filled in by the value of its corresponding argument
- However, a parameter is more than that: it is actually a **local variable**
- When a method is invoked, the value of its argument is computed, and the corresponding parameter (i.e., local variable) is initialized to this value
- Even if the value of a formal parameter is changed within a method (i.e., it is used as a local variable) the value of the argument cannot be changed

A Formal Parameter Used as a Local Variable (Part 1 of 5)



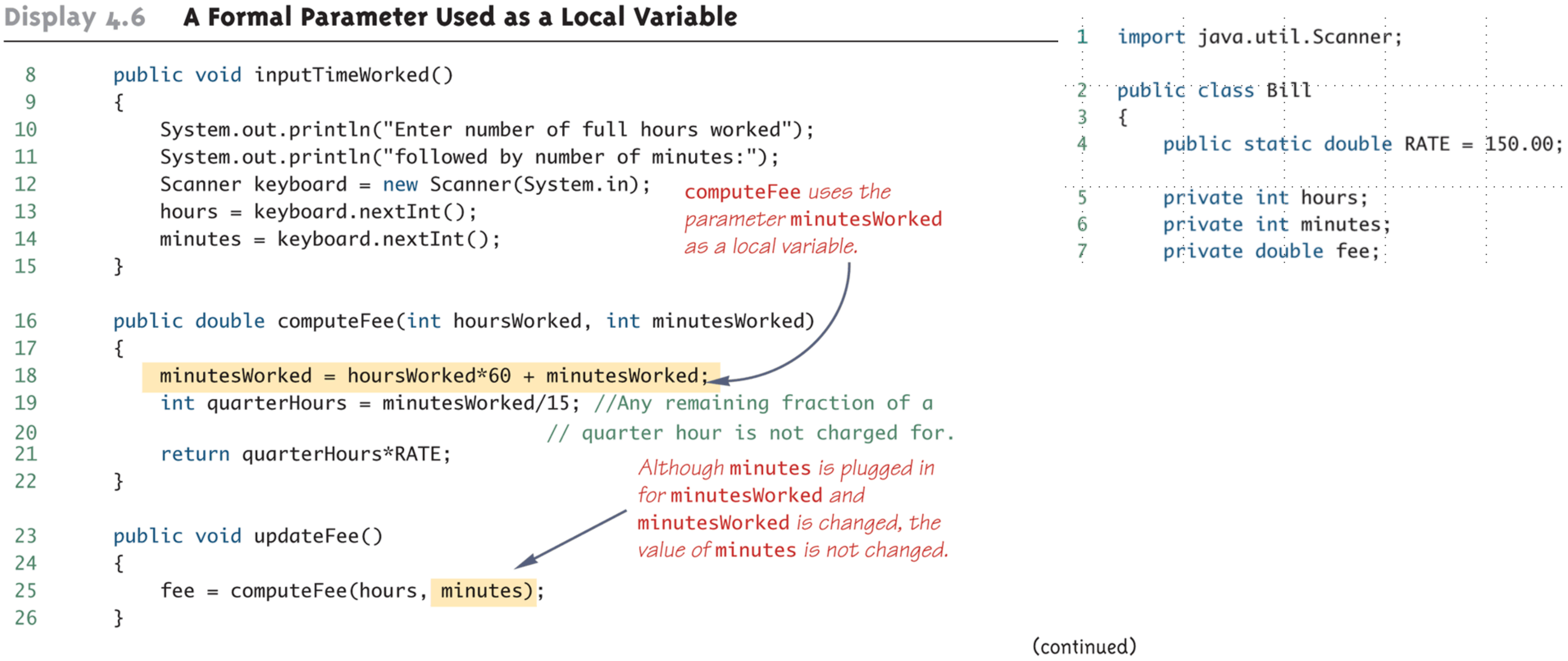
Display 4.6 A Formal Parameter Used as a Local Variable

This is the file **Bill.java**.

```
import java.util.Scanner;

public class Bill
{
    public static double RATE = 150.00; //Dollars per quarter hour

    private int hours;
    private int minutes;
    private double fee;
```

(continued)

A Formal Parameter Used as a Local Variable (Part 2 of 5)



Display 4.6 A Formal Parameter Used as a Local Variable

```java
import java.util.Scanner;

public class Bill
{
    public static double RATE = 150.00;

    private int hours;
    private int minutes;
    private double fee;
```

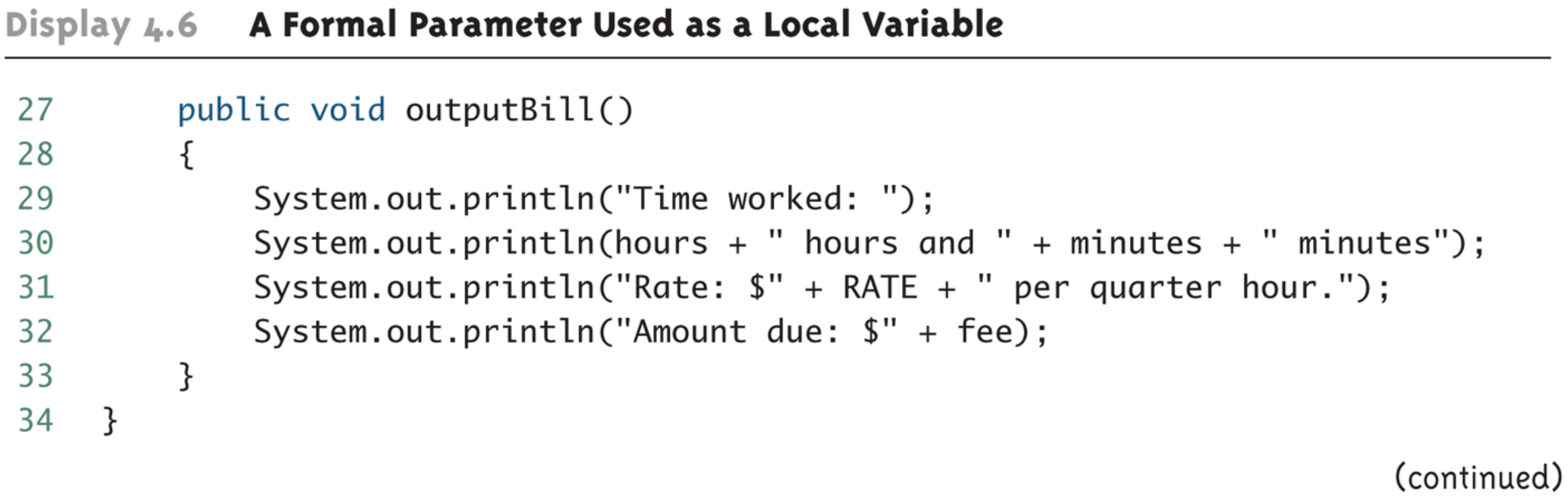
```java
    public void inputTimeWorked()
    {
        System.out.println("Enter number of full hours worked");
        System.out.println("followed by number of minutes:");
        Scanner keyboard = new Scanner(System.in);
        hours = keyboard.nextInt();
        minutes = keyboard.nextInt();
    }

    public double computeFee(int hoursWorked, int minutesWorked)
    {
        minutesWorked = hoursWorked*60 + minutesWorked;
        int quarterHours = minutesWorked/15; //Any remaining fraction of a
                                             // quarter hour is not charged for.
        return quarterHours*RATE;
    }

    public void updateFee()
    {
        fee = computeFee(hours, minutes);
    }
```

computeFee uses the parameter minutesWorked as a local variable.

Although minutes is plugged in for minutesWorked and minutesWorked is changed, the value of minutes is not changed.

(continued)

A Formal Parameter Used as a Local Variable (Part 3 of 5)

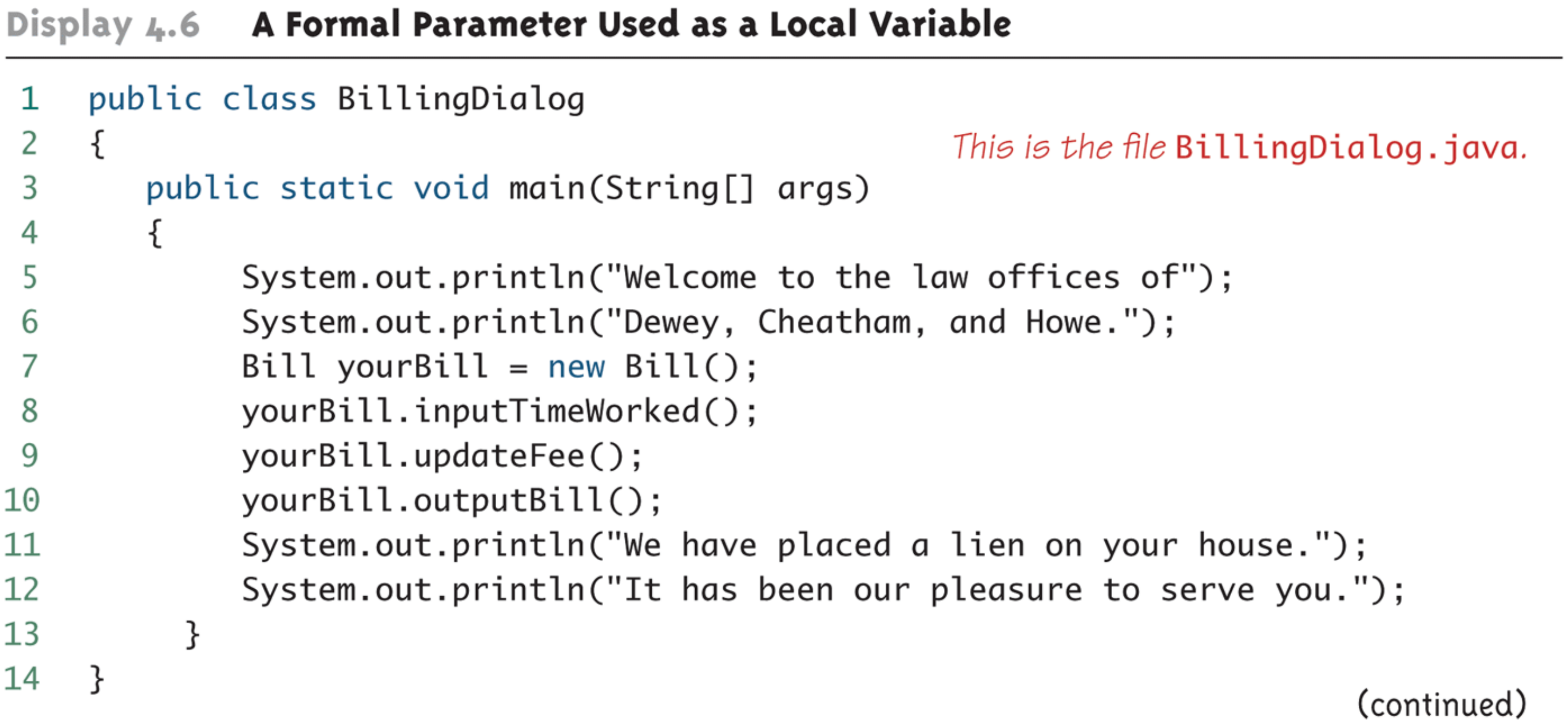


Display 4.6 **A Formal Parameter Used as a Local Variable**

```
    public void outputBill()
    {
        System.out.println("Time worked: ");
        System.out.println(hours + " hours and " + minutes + " minutes");
        System.out.println("Rate: $" + RATE + " per quarter hour.");
        System.out.println("Amount due: $" + fee);
    }
}
```

(continued)

A Formal Parameter Used as a Local Variable (Part 4 of 5)



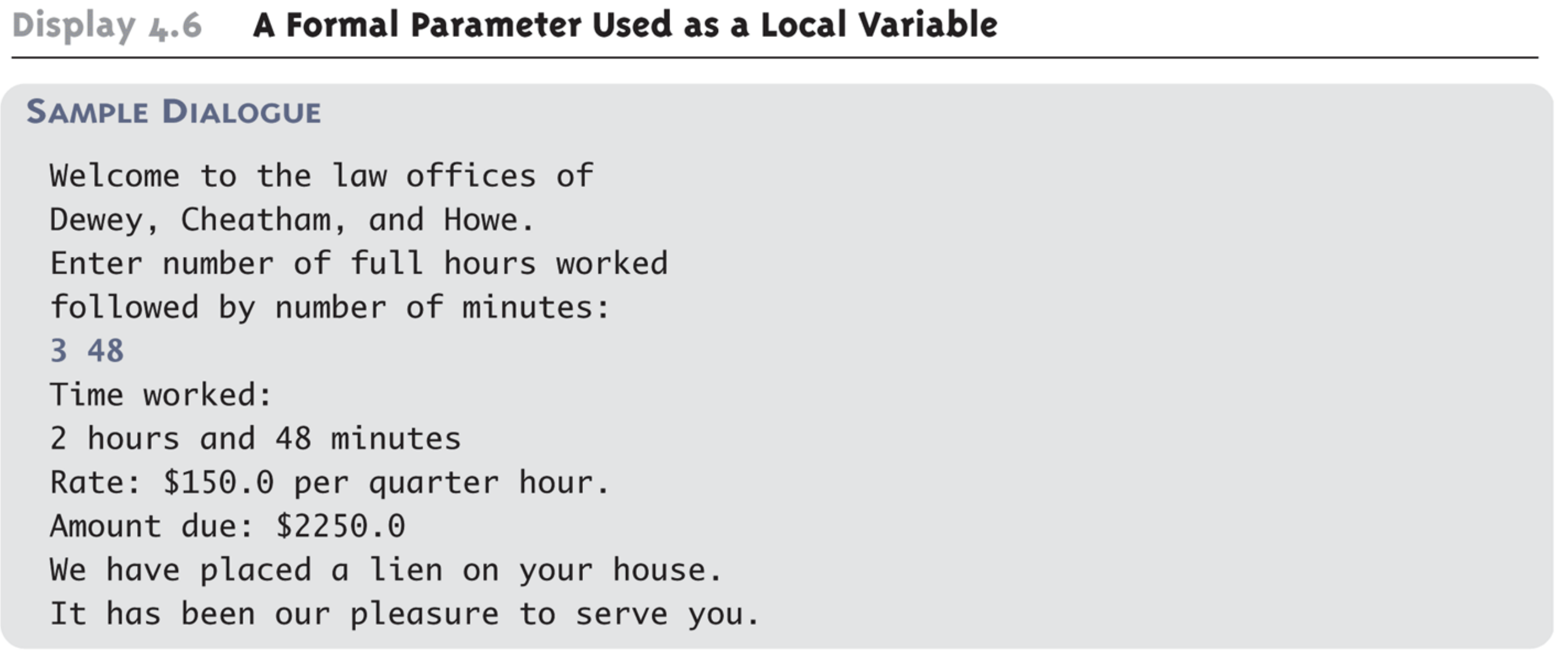
Display 4.6 A Formal Parameter Used as a Local Variable

```
public class BillingDialog
{
   public static void main(String[] args)
   {
        System.out.println("Welcome to the law offices of");
        System.out.println("Dewey, Cheatham, and Howe.");
        Bill yourBill = new Bill();
        yourBill.inputTimeWorked();
        yourBill.updateFee();
        yourBill.outputBill();
        System.out.println("We have placed a lien on your house.");
        System.out.println("It has been our pleasure to serve you.");
     }
}
```

This is the file `BillingDialog.java`.

(continued)

A Formal Parameter Used as a Local Variable (Part 5 of 5)

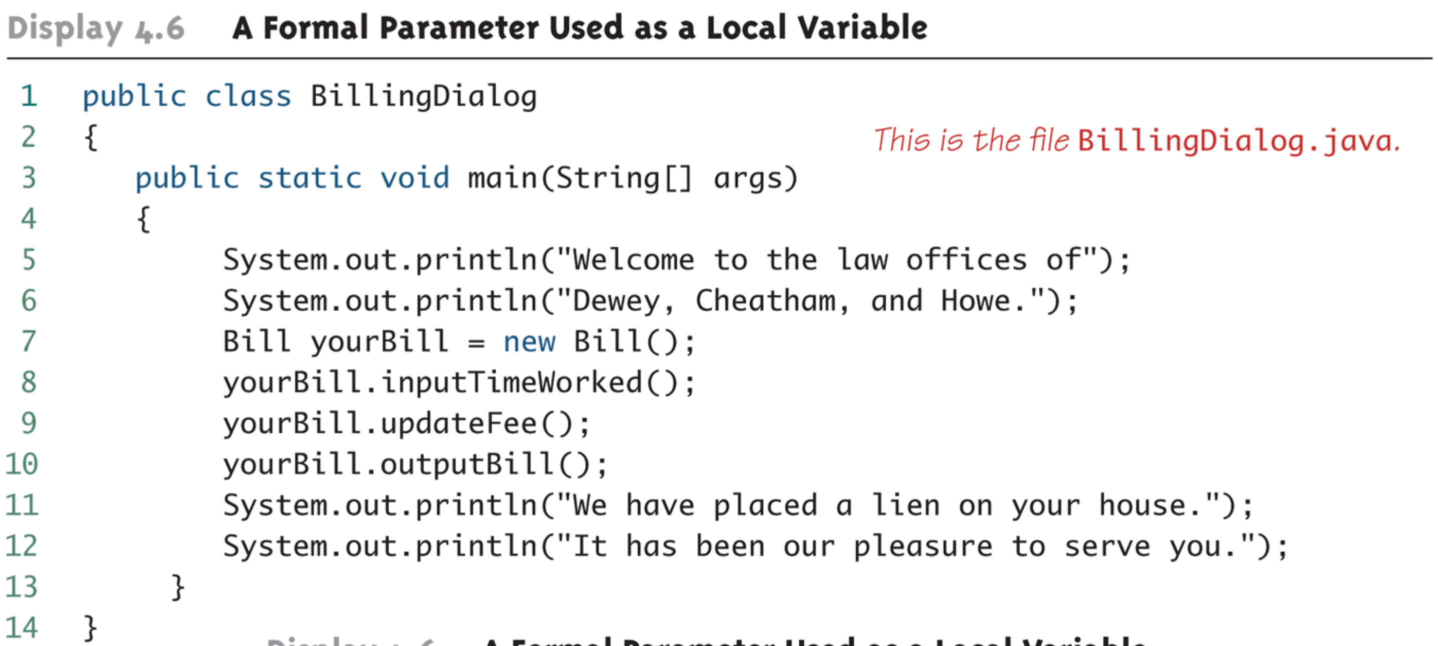


Display 4.6 **A Formal Parameter Used as a Local Variable**

Sample Dialogue

```
Welcome to the law offices of
Dewey, Cheatham, and Howe.
Enter number of full hours worked
followed by number of minutes:
3 48
Time worked:
2 hours and 48 minutes
Rate: $150.0 per quarter hour.
Amount due: $2250.0
We have placed a lien on your house.
It has been our pleasure to serve you.
```

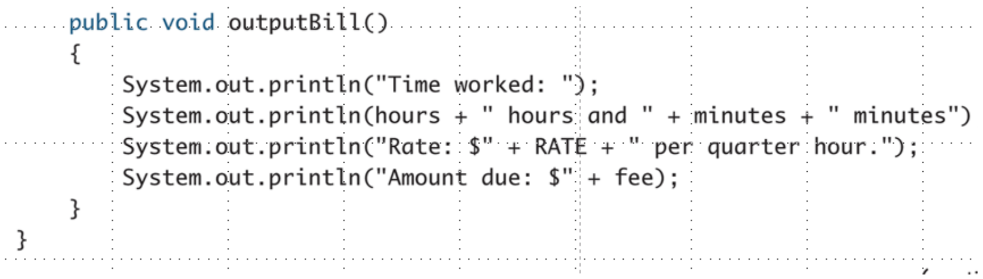
A Formal Parameter Used as a Local Variable



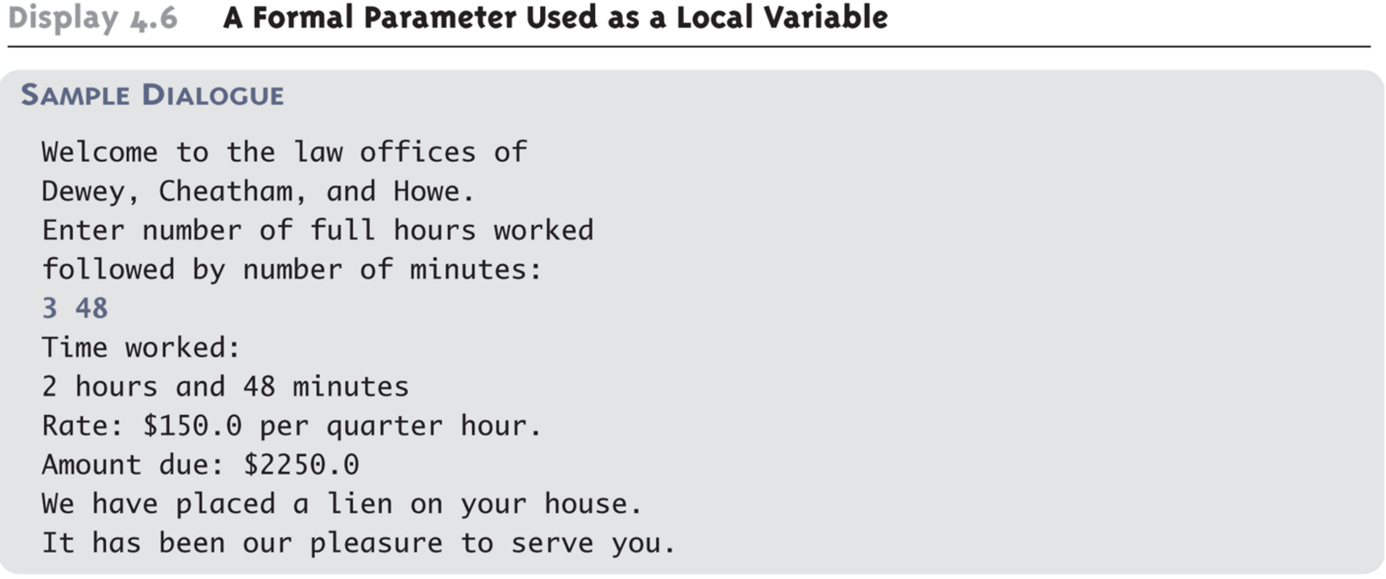
Display 4.6 A Formal Parameter Used as a Local Variable

This is the file `BillingDialog.java`.

```java
public class BillingDialog
{
    public static void main(String[] args)
    {
          System.out.println("Welcome to the law offices of");
          System.out.println("Dewey, Cheatham, and Howe.");
          Bill yourBill = new Bill();
          yourBill.inputTimeWorked();
          yourBill.updateFee();
          yourBill.outputBill();
          System.out.println("We have placed a lien on your house.");
          System.out.println("It has been our pleasure to serve you.");
      }
}
```



```java
    public void outputBill()
    {
        System.out.println("Time worked: ");
        System.out.println(hours + " hours and " + minutes + " minutes");
        System.out.println("Rate: $" + RATE + " per quarter hour.");
        System.out.println("Amount due: $" + fee);
    }
}
```



Display 4.6 A Formal Parameter Used as a Local Variable

Sample Dialogue

```
Welcome to the law offices of
Dewey, Cheatham, and Howe.
Enter number of full hours worked
followed by number of minutes:
3 48
Time worked:
2 hours and 48 minutes
Rate: $150.0 per quarter hour.
Amount due: $2250.0
We have placed a lien on your house.
It has been our pleasure to serve you.
```

Pitfall: Use of the Terms "Parameter" and "Argument"

- Do not be surprised to find that people often use the terms parameter and argument interchangeably
- When you see these terms, you may have to determine their exact meaning from context

NB!

The `this` Parameter

- All instance variables are understood to have `<the calling object>.` in front of them
- If an **explicit** name for the calling object is needed, the keyword `this` can be used

`myInstanceVariable` always means and is always interchangeable with `this.myInstanceVariable`

The `this` Parameter

- `this` ***must*** be used if a parameter or other local variable with the same name is used in the method
  - Otherwise, all instances of the variable name will be interpreted as local

```
int someVariable = this.someVariable
```

local

instance

The **`this`** Parameter

- The **`this`** parameter is a kind of hidden parameter
- Even though it does not appear on the parameter list of a method, it is still a parameter
- When a method is invoked, the calling object is automatically plugged in for **`this`**

Methods That Return a Boolean Value

- An invocation of a method that returns a value of type **`boolean`** returns either **`true`** or **`false`**
- Therefore, it is common practice to use an invocation of such a method to control statements and loops where a Boolean expression is expected
  - **`if-else`** statements, **`while`** loops, etc.

1 min

Write two examples

The methods `equals` and `toString`

- Java expects certain methods, such as `equals` and `toString`, to be in all, or almost all, classes
- The purpose of `equals`, a `boolean` valued method, is to compare two objects of the class to see if they satisfy the notion of "being equal"
  - Note: You cannot use `==` to compare objects

```
public boolean equals(ClassName objectName)
```

- The purpose of the `toString` method is to return a `String` value that represents the data in the object

```
public String toString()
```

Testing Methods

- Each method should be tested in a program in which it is the only untested program
    - A program whose only purpose is to test a method is called a *driver program*
- One method often invokes other methods, so one way to do this is to first test all the methods invoked by that method, and then test the method itself
    - This is called ***bottom-up testing***
- Sometimes it is necessary to test a method before another method it depends on is finished or tested
    - In this case, use a simplified version of the method, called a *stub,* to return a value for testing

The Fundamental Rule for Testing Methods

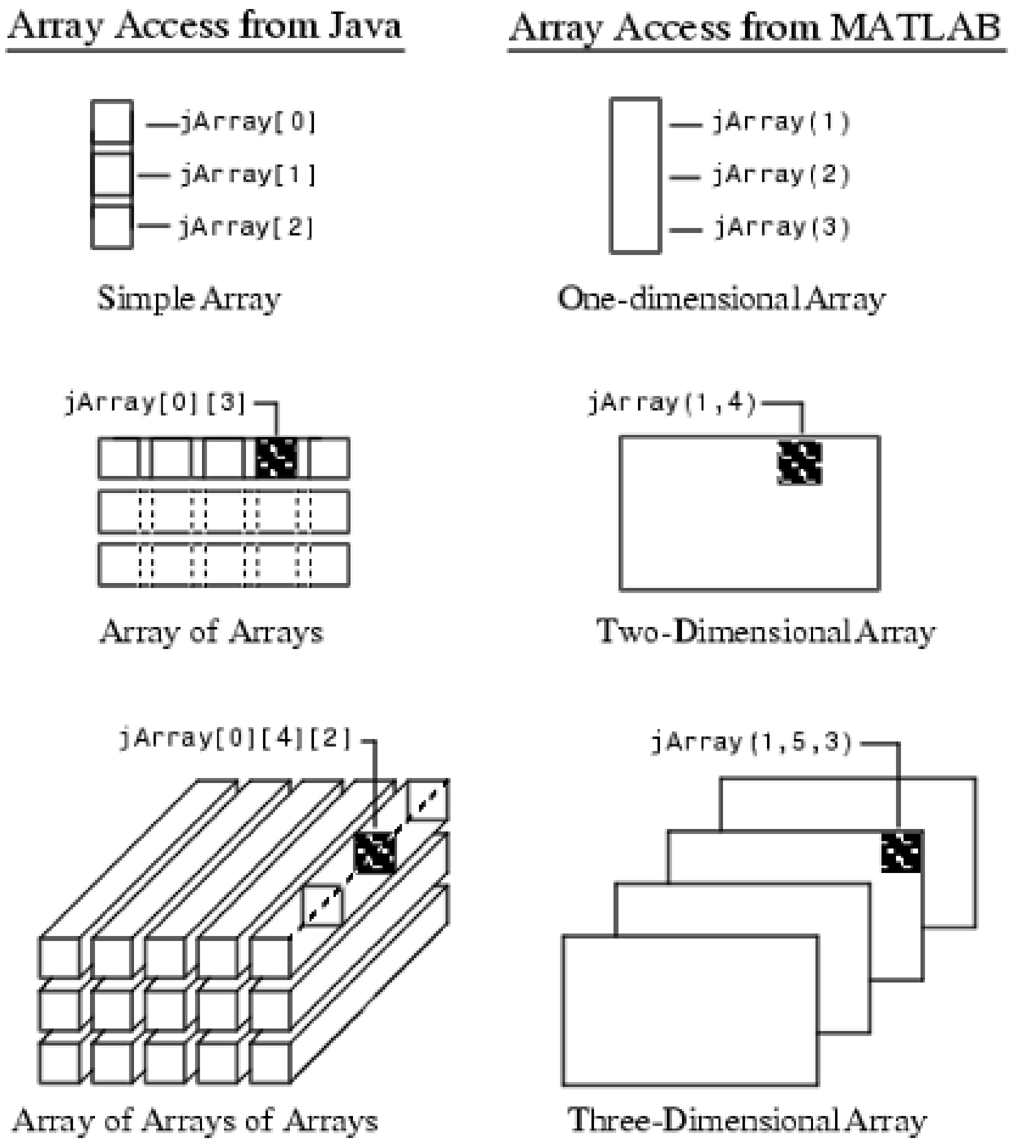
- ***Every method should be tested in a program in which every other method in the testing program has already been fully tested and debugged***

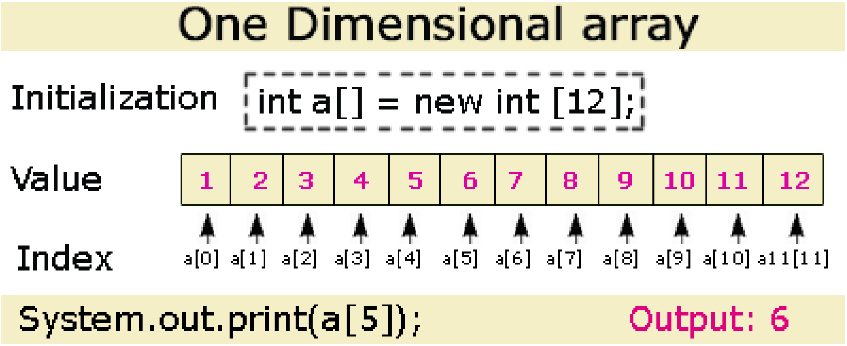
ARRAYS

Chapter 6 in Savitch

(a quick refresher)

Arrays: refresher and examples





One Dimensional array

Initialization `int a[] = new int [12];`

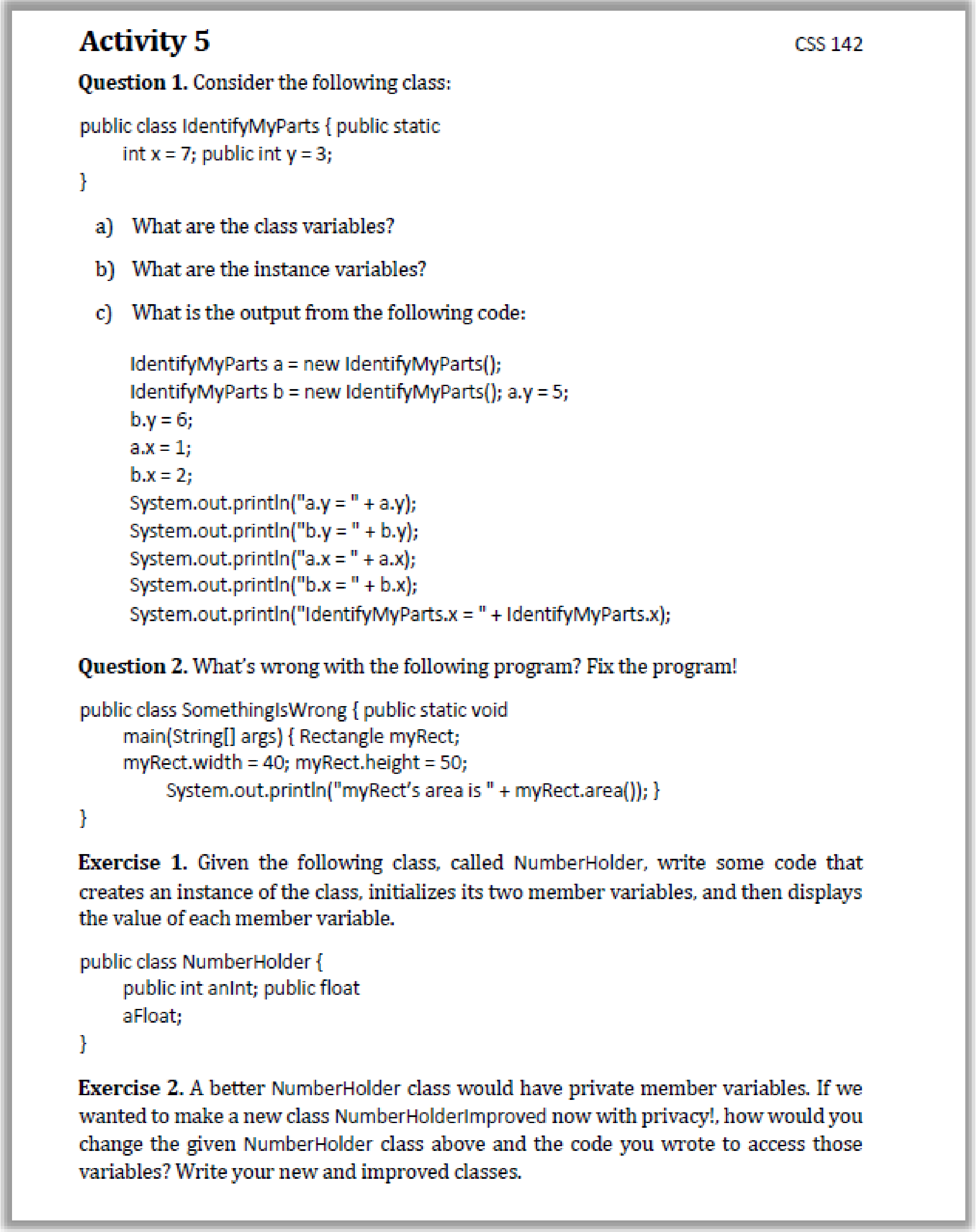
Value	1	2	3	4	5	6	7	8	9	10	11	12
Index	a[0]	a[1]	a[2]	a[3]	a[4]	a[5]	a[6]	a[7]	a[8]	a[9]	a[10]	a11[11]

`System.out.print(a[5]);` Output: 6

Examples of One-Dimensional, Two-Dimensional and Three-Dimensional Arrays?

Hands on Assignment 5:

optional



Activity 5

CSS 142

Question 1. Consider the following class:

```
public class IdentifyMyParts { public static
      int x = 7; public int y = 3;
}
```

a) What are the class variables?

b) What are the instance variables?

c) What is the output from the following code:

```
IdentifyMyParts a = new IdentifyMyParts();
IdentifyMyParts b = new IdentifyMyParts(); a.y = 5;
b.y = 6;
a.x = 1;
b.x = 2;
System.out.println("a.y = " + a.y);
System.out.println("b.y = " + b.y);
System.out.println("a.x = " + a.x);
System.out.println("b.x = " + b.x);
System.out.println("IdentifyMyParts.x = " + IdentifyMyParts.x);
```

Question 2. What's wrong with the following program? Fix the program!

```
public class SomethingIsWrong { public static void
      main(String[] args) { Rectangle myRect;
      myRect.width = 40; myRect.height = 50;
            System.out.println("myRect's area is " + myRect.area()); }
}
```

Exercise 1. Given the following class, called NumberHolder, write some code that creates an instance of the class, initializes its two member variables, and then displays the value of each member variable.

```
public class NumberHolder {
      public int anInt; public float
      aFloat;
}
```

Exercise 2. A better NumberHolder class would have private member variables. If we wanted to make a new class NumberHolderImproved now with privacy!, how would you change the given NumberHolder class above and the code you wrote to access those variables? Write your new and improved classes.