v1.0

THE UNIVERSITY OF MELBOURNE

Programming and Software Development
COMP90041

Lecture 4

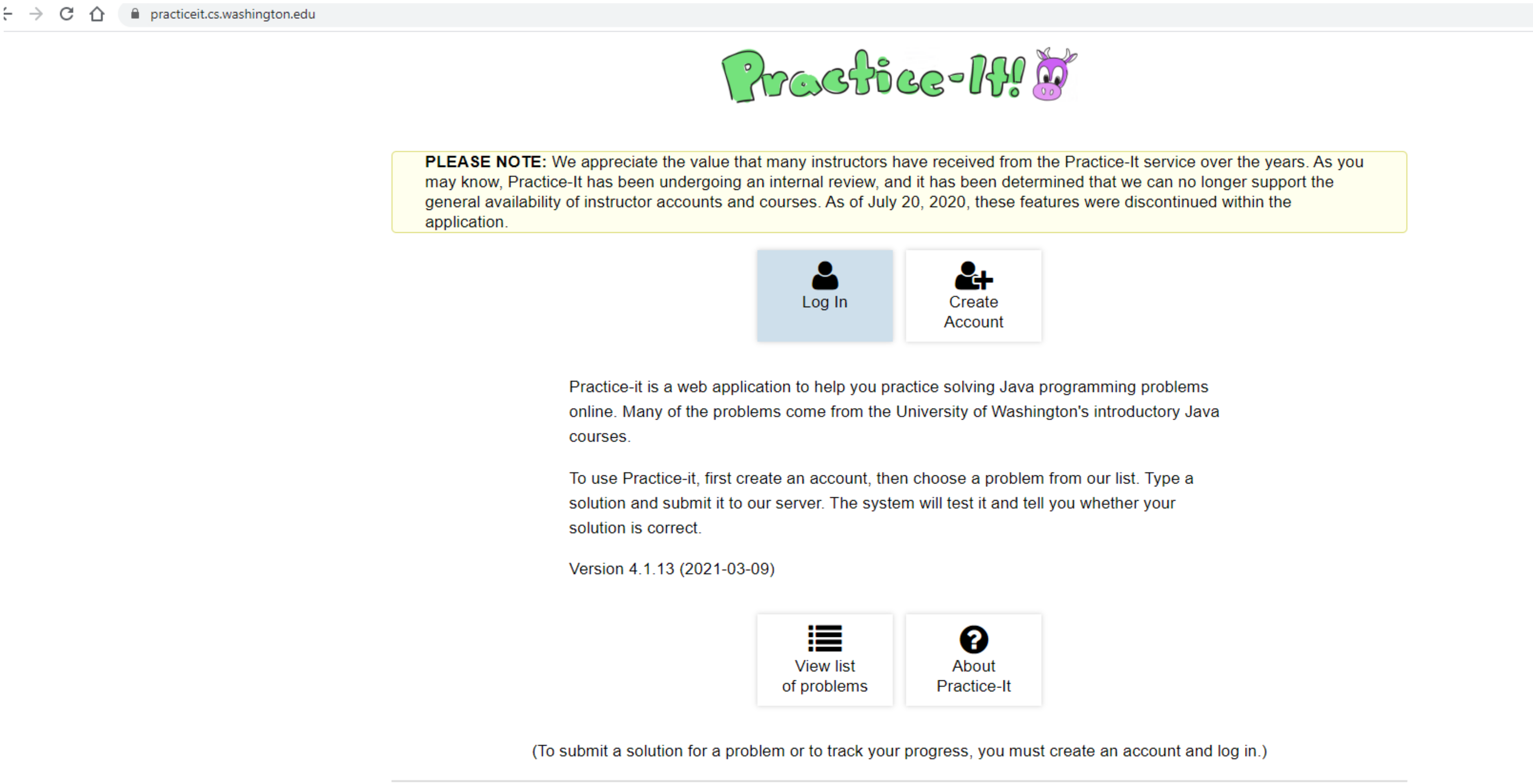
Classes and Methods

NOTE: Some of the Material in these slides are adopted from
* Lectures Notes prepared by Dr. Peter Schachte and
* the Textbook resources

Udaya Parampalli & Thuan Pham
Semester 1 2021, 20-02-2021

- Topics: Chapter 3 of the textbook
  - Branching mechanisms
  - Evaluating Boolean expressions
  - Different ways of constructing Loops
  - Debugging
- Tutorial – Week 3
  - Practice small Java programs
  - Formatted output
  - Running command line arguments

Review



PLEASE NOTE: We appreciate the value that many instructors have received from the Practice-It service over the years. As you may know, Practice-It has been undergoing an internal review, and it has been determined that we can no longer support the general availability of instructor accounts and courses. As of July 20, 2020, these features were discontinued within the application.

Practice-it is a web application to help you practice solving Java programming problems online. Many of the problems come from the University of Washington's introductory Java courses.

To use Practice-it, first create an account, then choose a problem from our list. Type a solution and submit it to our server. The system will test it and tell you whether your solution is correct.

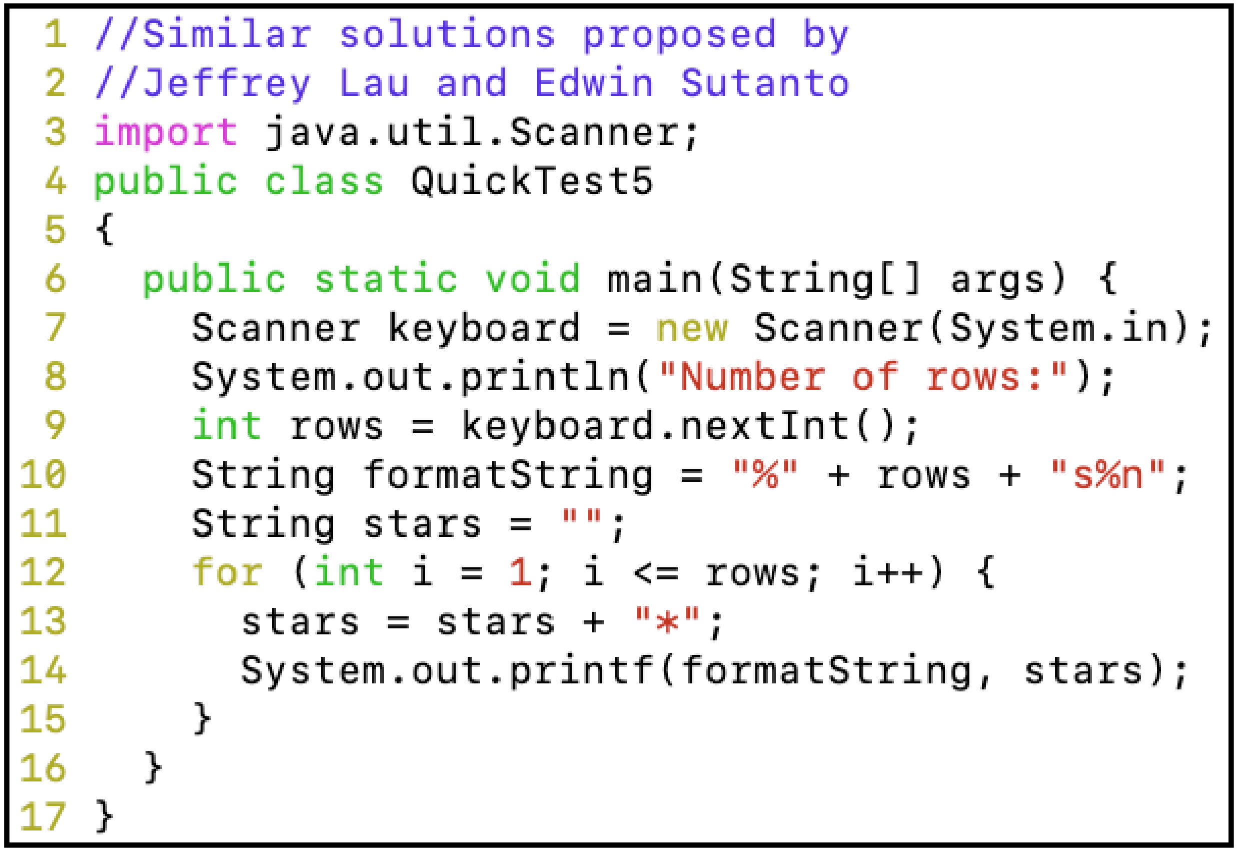
Version 4.1.13 (2021-03-09)

(To submit a solution for a problem or to track your progress, you must create an account and log in.)

Practice-IT!

Write a program that prints out the following triangle to the screen using a nested for loop.

```
         *
        **
       ***
      ****
     *****
    ******
   *******
  ********
 *********
**********
```



```java
//Similar solutions proposed by
//Jeffrey Lau and Edwin Sutanto
import java.util.Scanner;
public class QuickTest5
{
  public static void main(String[] args) {
    Scanner keyboard = new Scanner(System.in);
    System.out.println("Number of rows:");
    int rows = keyboard.nextInt();
    String formatString = "%" + rows + "s%n";
    String stars = "";
    for (int i = 1; i <= rows; i++) {
      stars = stars + "*";
      System.out.printf(formatString, stars);
    }
  }
}
```

Quick Test - 5

- Class definitions
  - Class structure
  - Variables
  - Methods
- Encapsulation
  - Access modifiers (e.g., public vs private)
  - Accessor and mutator methods
- Overloading
- Constructors

Outline

Methods and Classes

- A Java program is made up of interacting objects from various classes.
- A method is an operation defined by a class
- i.e., it defines how to do something
- The Java library defines many methods And you can define your own
- Similar to functions, subroutines, procedures in other languages
- Java supports two kinds of methods:
  - Class or static methods, and
  - Instance or non-static methods
- Instance methods are more common, but Class methods are simpler, so we start there

- Class definitions
  - Class structure
  - Variables
  - Methods
- Encapsulation
  - Access modifiers (e.g., public vs private)
  - Accessor and mutator methods
- Overloading
- Constructors

Outline

- Classes are central to Java
- 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

A Java program consists of
objects from various classes
Interacting with one another

Introduction

- A class is a type and you can declare variables of a class type (e.g., Car myCar)
- A value of a class type is called an ***object*** or *an instance of the class*
- An object has both ***data*** and ***actions***
  - **actions** are called **methods**
- Each object can have different data, but all objects of a class have the same types of data and all objects of a class have the same methods (e.g., myCar vs yourCar)

Terminology

Primitive Type Values vs. Class Type Values

- 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

- 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

Class Definition

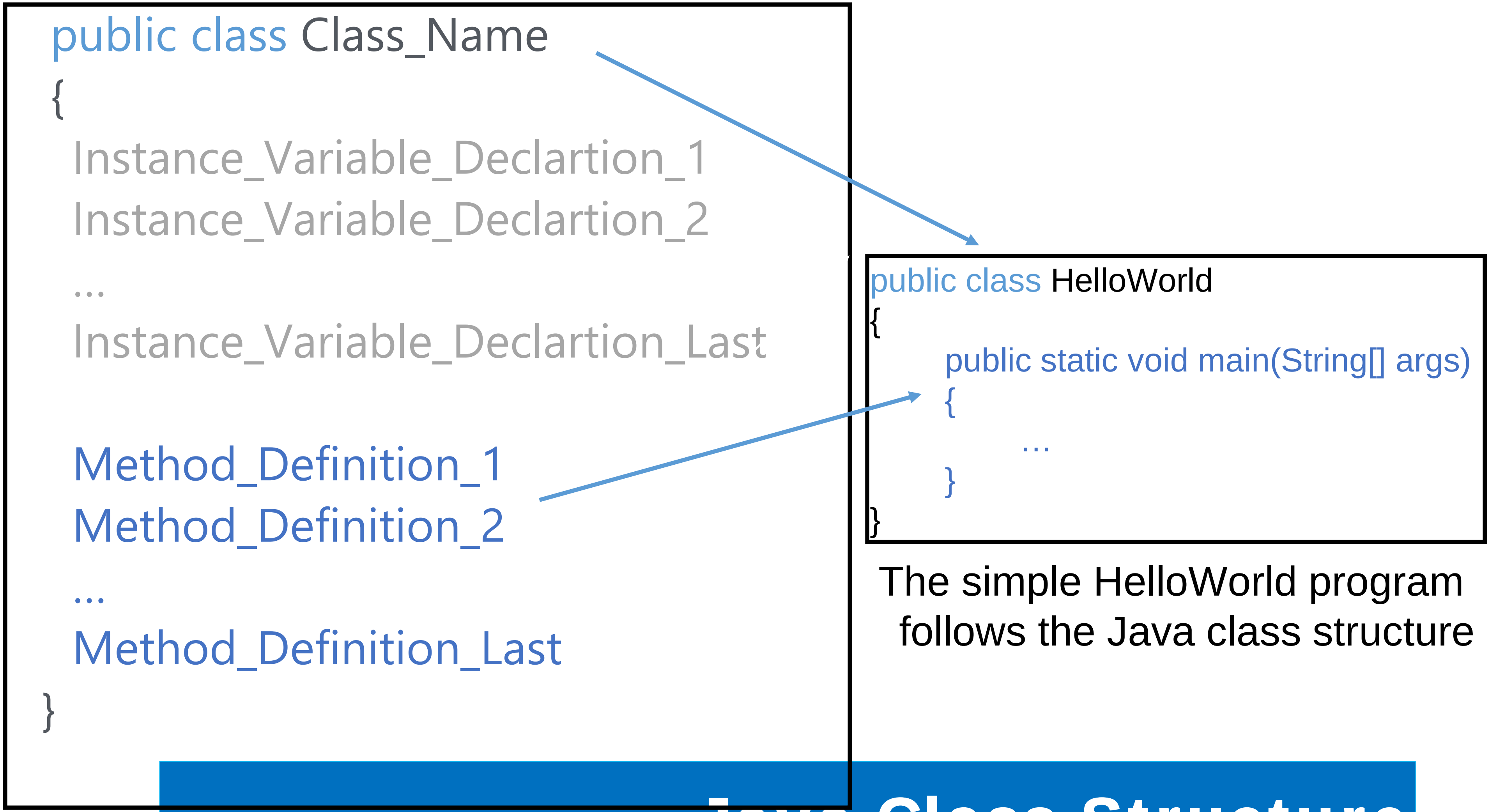
```
public class Class_Name
{
  Instance_Variable_Declartion_1
  Instance_Variable_Declartion_2
  …
  Instance_Variable_Declartion_Last

  Method_Definition_1
  Method_Definition_2
  …
  Method_Definition_Last
}
```

Data

Actions

Java Class Structure

Java Class Structure



```
public class Class_Name
{
  Instance_Variable_Declartion_1
  Instance_Variable_Declartion_2
  …
  Instance_Variable_Declartion_Last

  Method_Definition_1
  Method_Definition_2
  …
  Method_Definition_Last
}
```

```
public class HelloWorld
{
        public static void main(String[] args)
        {
                …
        }
}
```

The simple HelloWorld program follows the Java class structure

- 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();

E.g. Date.java and DateDemo.java

The new Operator

- Instance variables can be defined as in the following two examples
  - Note the **public** modifier (for now):

```
public String  instanceVar1;
public int  instanceVar2;
```

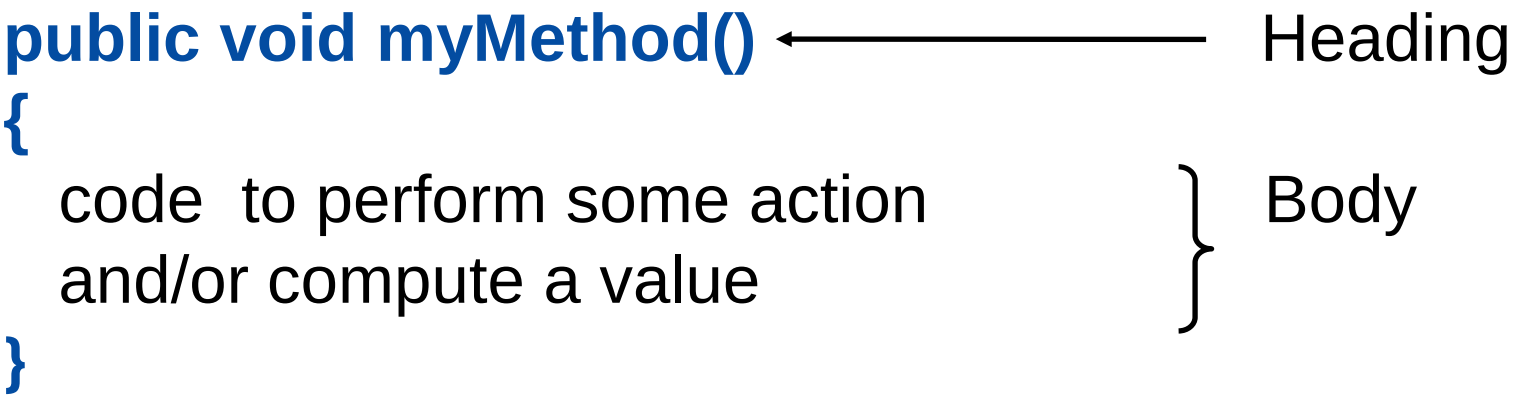
- In order to refer to a particular instance variable, preface it with its object name as follows:

```
objectName.instanceVar1
objectName.instanceVar2
```

Instance Variables and Methods

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:

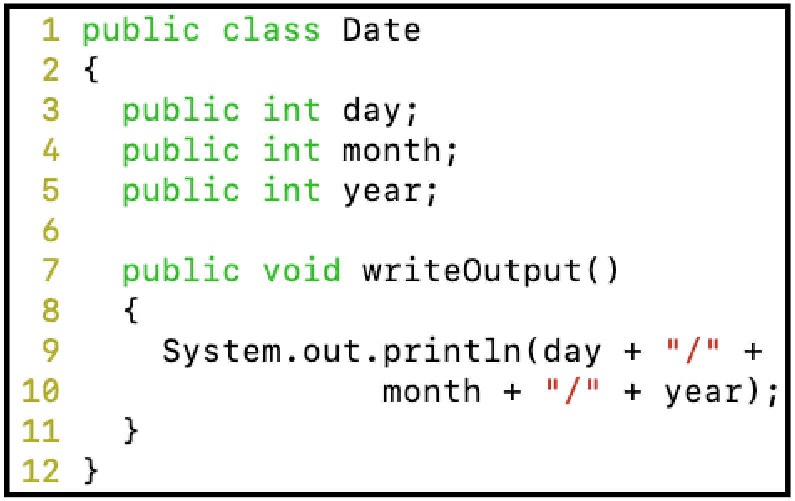
```
classVar.myMethod();
```

- Invoking a method is equivalent to executing the method body

- 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

File Names and Locations

Example-1: Simple class definition



```java
public class Date
{
  public int day;
  public int month;
  public int year;

  public void writeOutput()
  {
    System.out.println(day + "/" +
                month + "/" + year);
  }
}
```

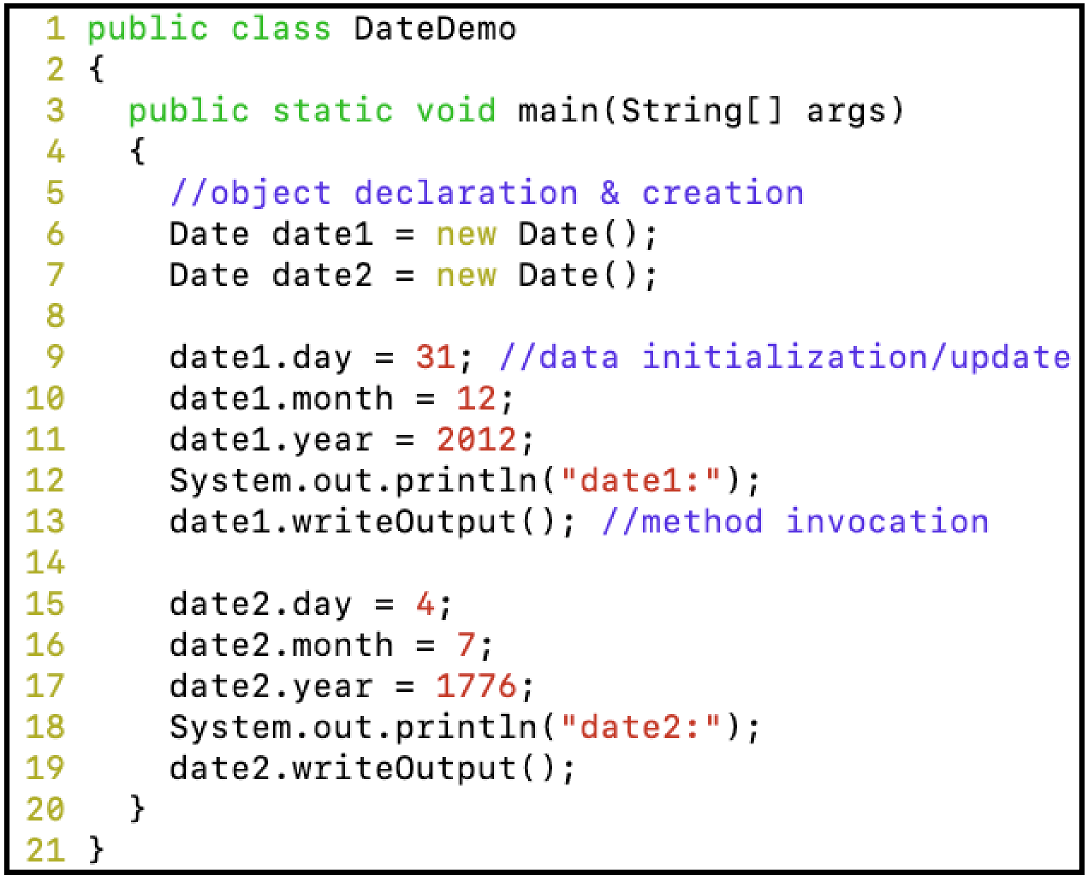
Date.java

Compilation:

javac DateDemo.java

Execution:

java DateDemo



```java
public class DateDemo
{
  public static void main(String[] args)
  {
    //object declaration & creation
    Date date1 = new Date();
    Date date2 = new Date();

    date1.day = 31; //data initialization/update
    date1.month = 12;
    date1.year = 2012;
    System.out.println("date1:");
    date1.writeOutput(); //method invocation

    date2.day = 4;
    date2.month = 7;
    date2.year = 1776;
    System.out.println("date2:");
    date2.writeOutput();
  }
}
```

DateDemo.java

- 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(parameter_List)

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

  public void methodName(parameter_List)

Method definitions

- 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)
```

`main` is a `void` Method

- 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 (for ***local variables***)
    statements
}

Method body

- 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

`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;

`return` Statements

- 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();
```

Method invocation

- 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.

Methods That Return a Boolean Value

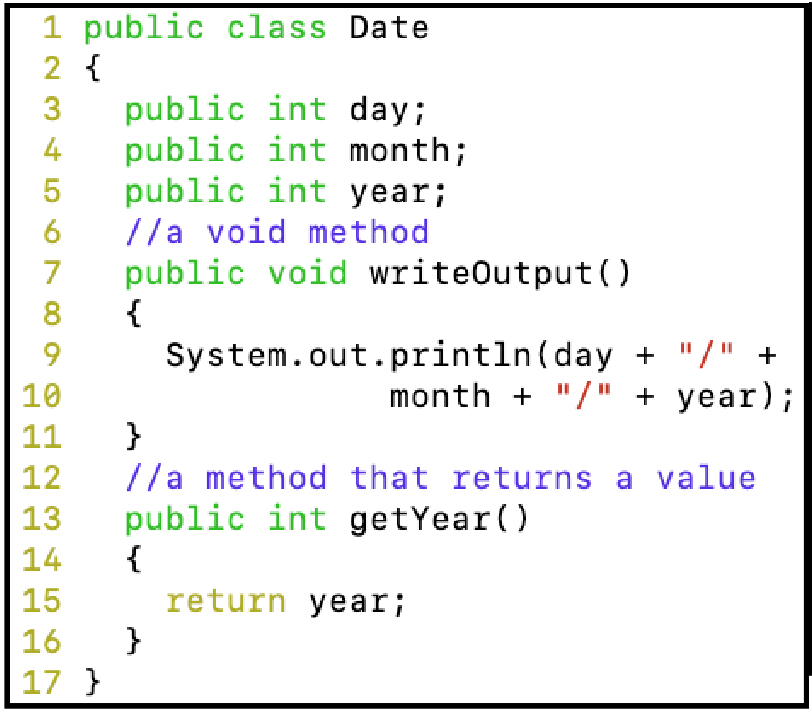
- A method that returns a value can also perform an action

- 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();
```

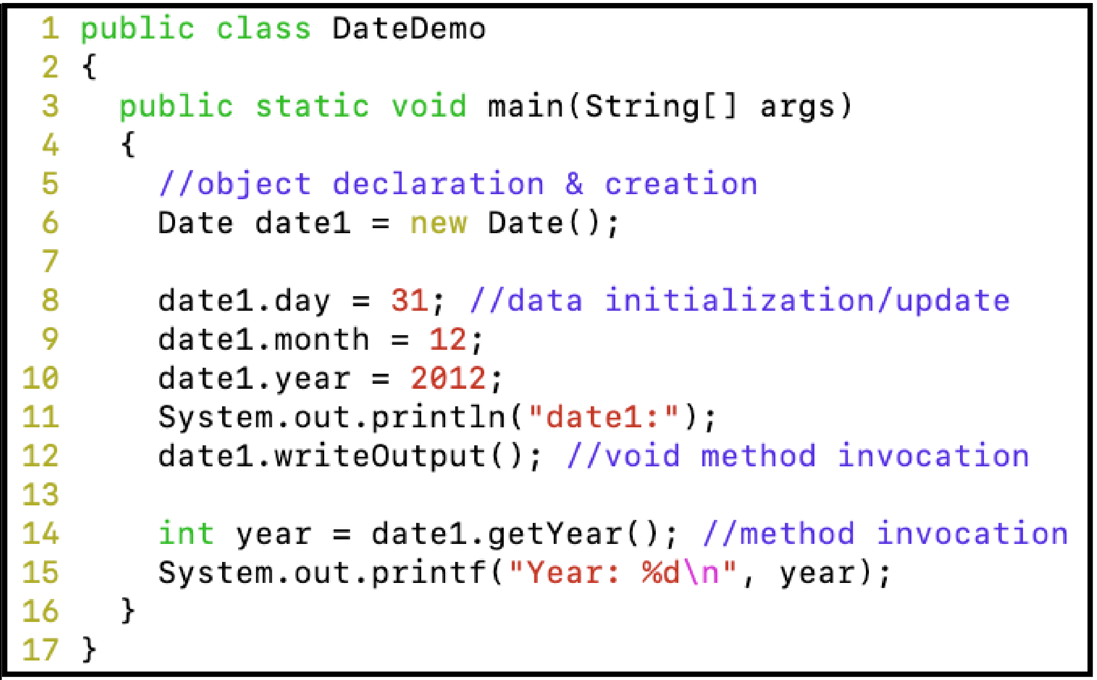
Any Method Can Be Used As a void Method

Example-2: Types of methods



```java
public class Date
{
  public int day;
  public int month;
  public int year;
  //a void method
  public void writeOutput()
  {
    System.out.println(day + "/" +
               month + "/" + year);
  }
  //a method that returns a value
  public int getYear()
  {
    return year;
  }
}
```

Date.java



```java
public class DateDemo
{
  public static void main(String[] args)
  {
    //object declaration & creation
    Date date1 = new Date();

    date1.day = 31; //data initialization/update
    date1.month = 12;
    date1.year = 2012;
    System.out.println("date1:");
    date1.writeOutput(); //void method invocation

    int year = date1.getYear(); //method invocation
    System.out.printf("Year: %d\n", year);
  }
}
```

DateDemo.java

- A variable declared within a method definition is called a ***local variable***
  - 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

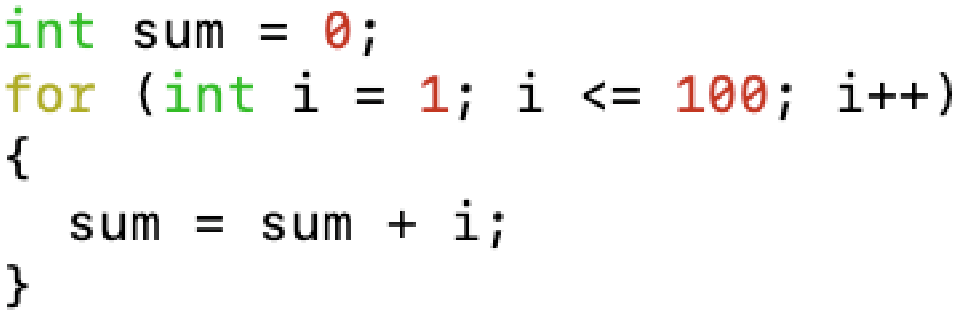
- **Note:** Some programming languages include another kind of variable called a *global* variables. The Java language does **not** have global variables

Local Variables

- 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
  - When the block ends, all variables declared within the block disappear
- **Note:** in Java, you cannot have two variables with the same name inside a single method definition (e.g., inside a block and outside a block)

Blocks

- You can declare one or more variables within the initialization portion of a **for** statement



```
int sum = 0;
for (int i = 1; i <= 100; i++)
{
   sum = sum + i;
}
```

- The variable **i** is 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

Declaring Variables in a for Statement

- 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

- 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

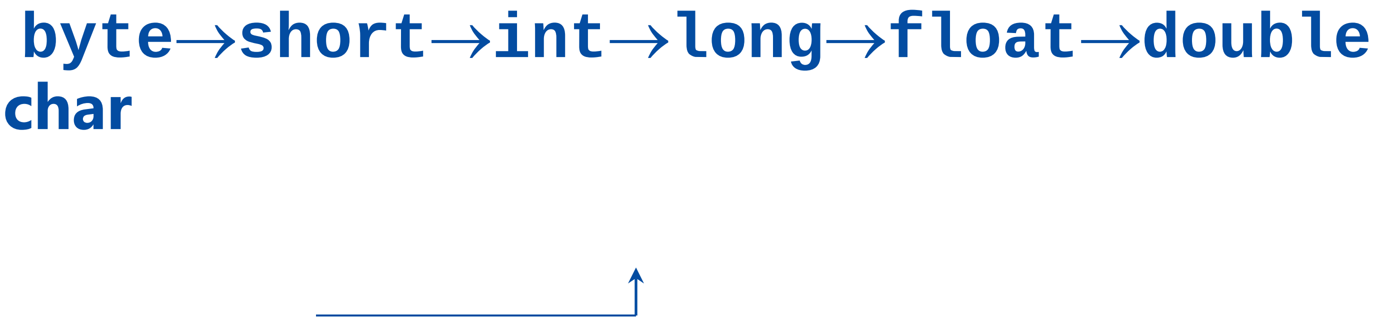
- When a method is invoked, the appropriate values must be passed to the method in the form of ***arguments***
  - Arguments are also called ***actual parameters***
- The *number and order* of the arguments must exactly match that of the parameter list
- The type of each argument must be compatible with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

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

Parameters of a Primitive Type

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

- 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

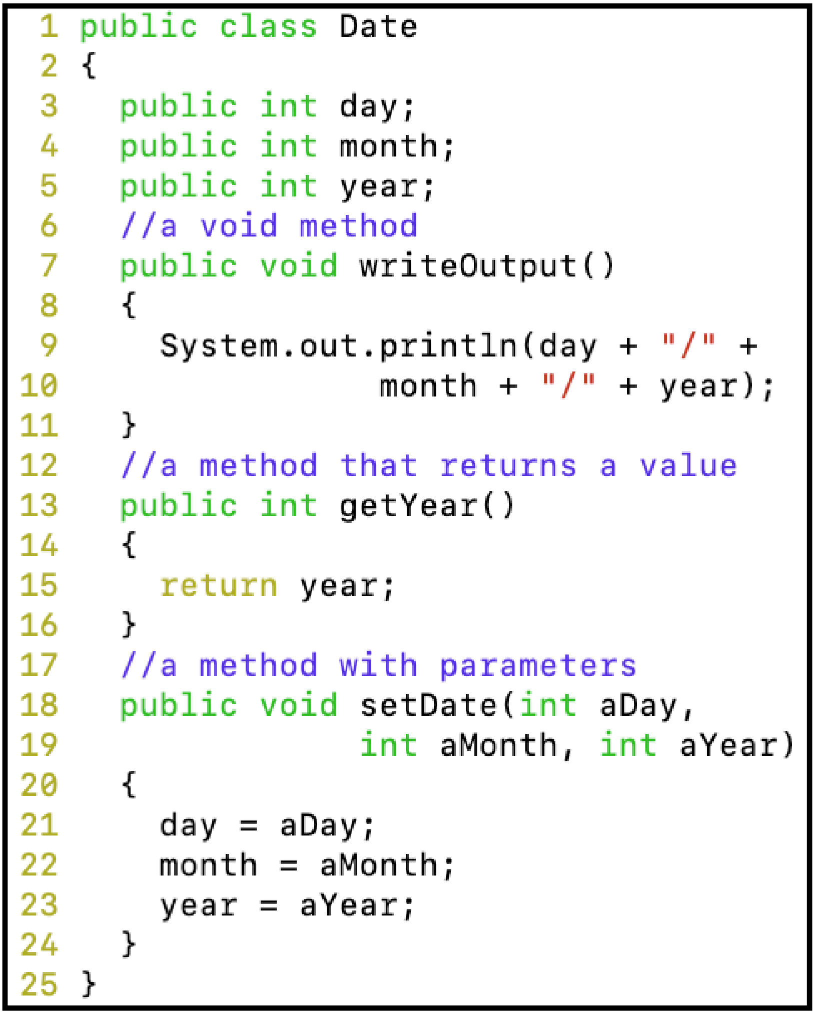
    call-by-value mechanism

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


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

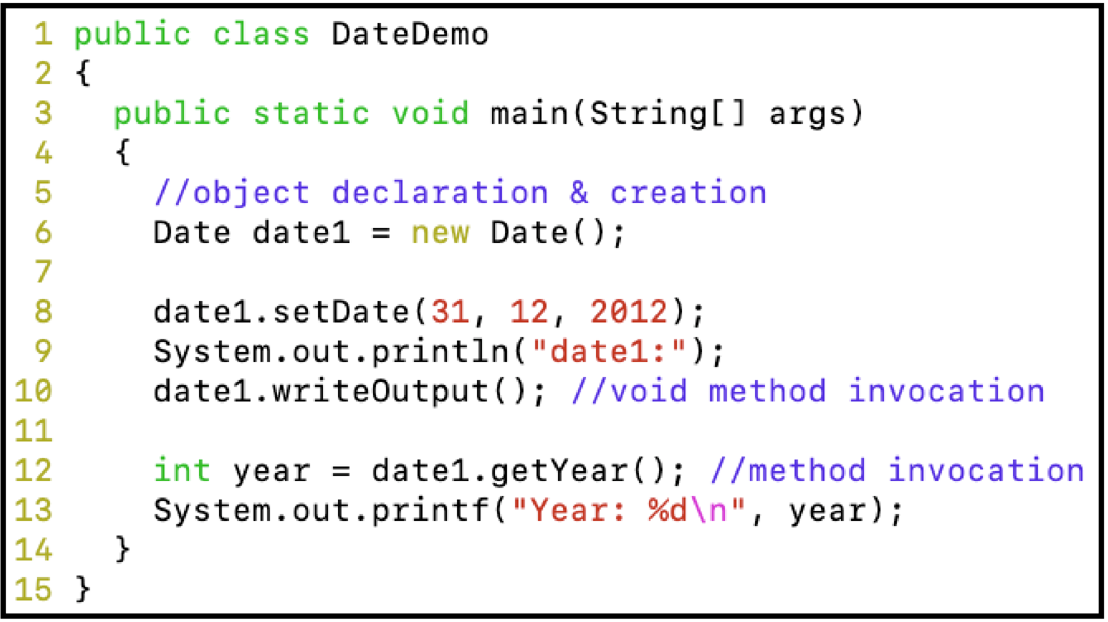
Parameters of a Primitive Type

- A parameter is filled in by the value of its corresponding argument
- A parameter is actually a local variable
- When a method is invoked, the value of its argument is computed/evaluated, 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*

Call-by-value mechanism



```
public class Date
{
  public int day;
  public int month;
  public int year;
  //a void method
  public void writeOutput()
  {
    System.out.println(day + "/" +
               month + "/" + year);
  }
  //a method that returns a value
  public int getYear()
  {
    return year;
  }
  //a method with parameters
  public void setDate(int aDay,
              int aMonth, int aYear)
  {
    day = aDay;
    month = aMonth;
    year = aYear;
  }
}
```



```
public class DateDemo
{
  public static void main(String[] args)
  {
    //object declaration & creation
    Date date1 = new Date();

    date1.setDate(31, 12, 2012);
    System.out.println("date1:");
    date1.writeOutput(); //void method invocation

    int year = date1.getYear(); //method invocation
    System.out.printf("Year: %d\n", year);
  }
}
```

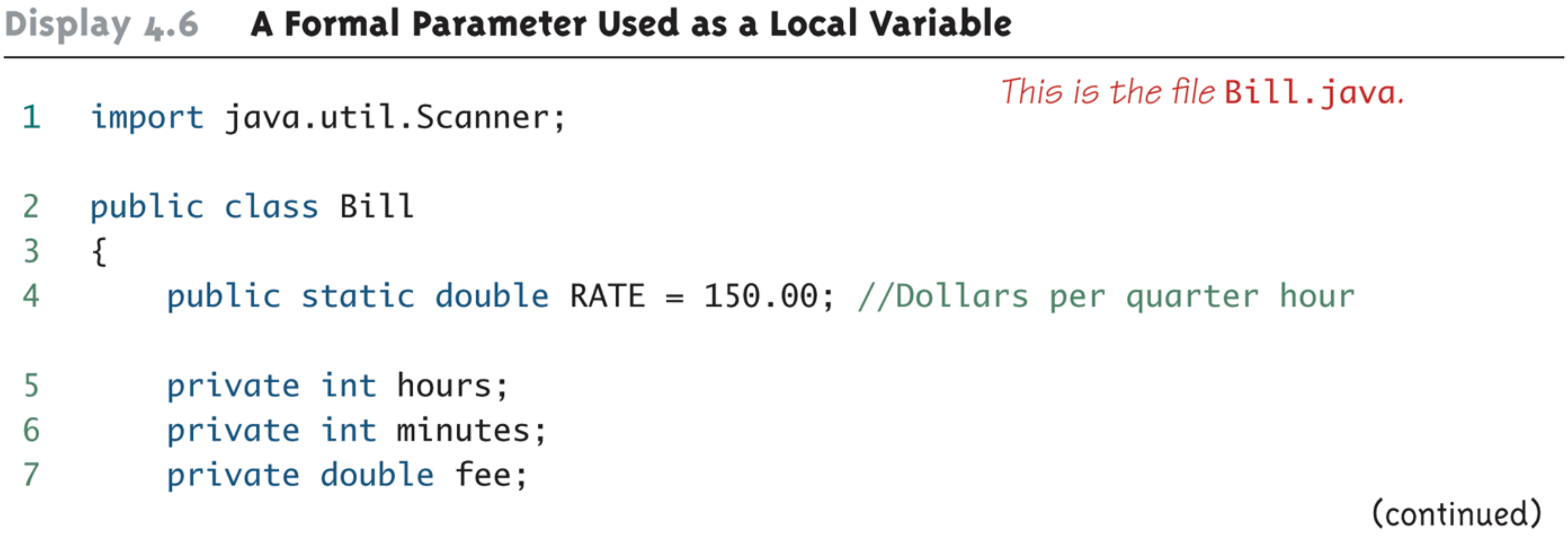
DateDemo.java

Date.java

Example-3: Primitive parameters

- Use a method parameter as a local variable
  - Update the value of the parameter inside the method

Another example



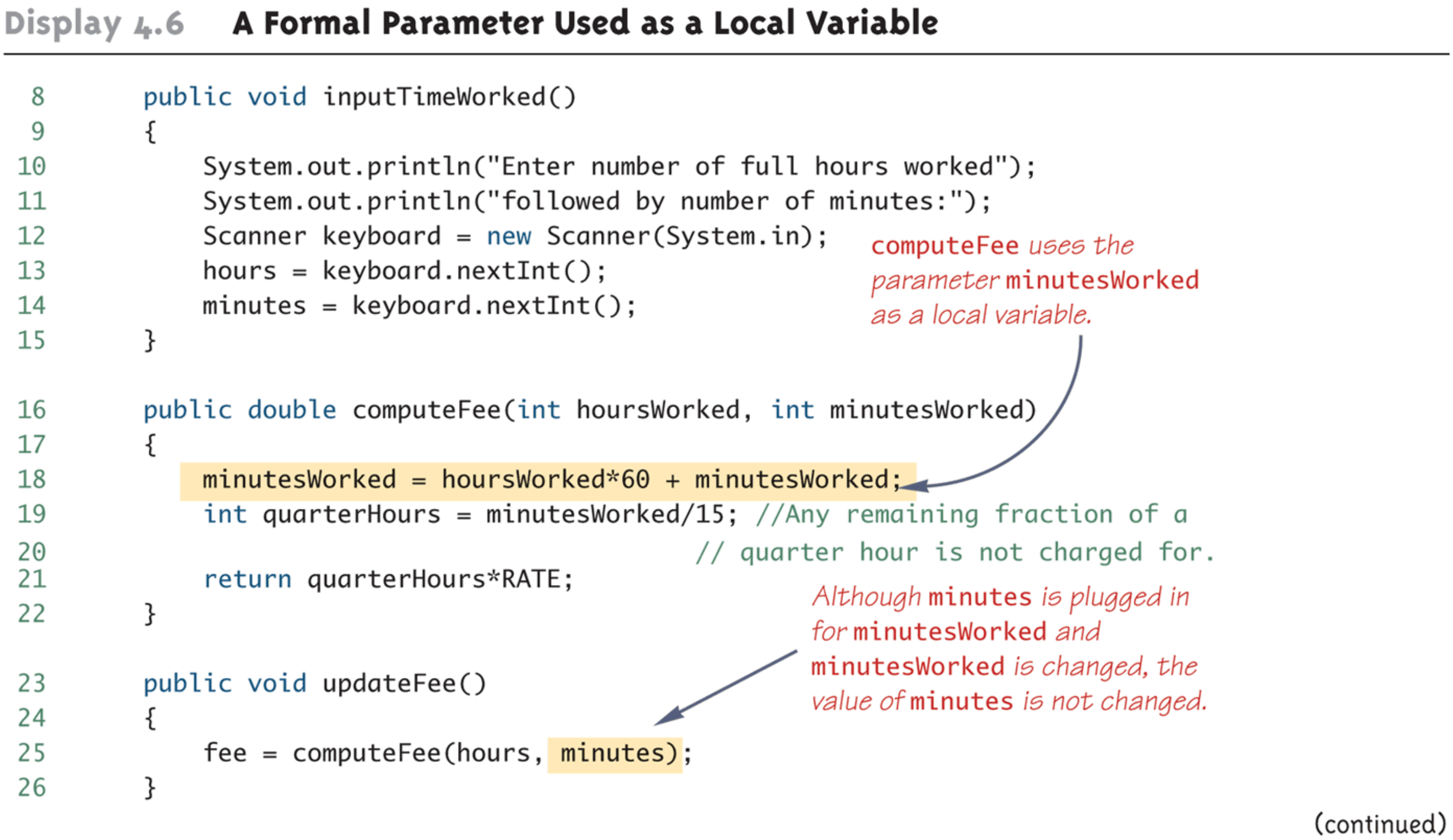
Display 4.6 **A Formal Parameter Used as a Local Variable**

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 1 of 5)



Display 4.6 A Formal Parameter Used as a Local Variable

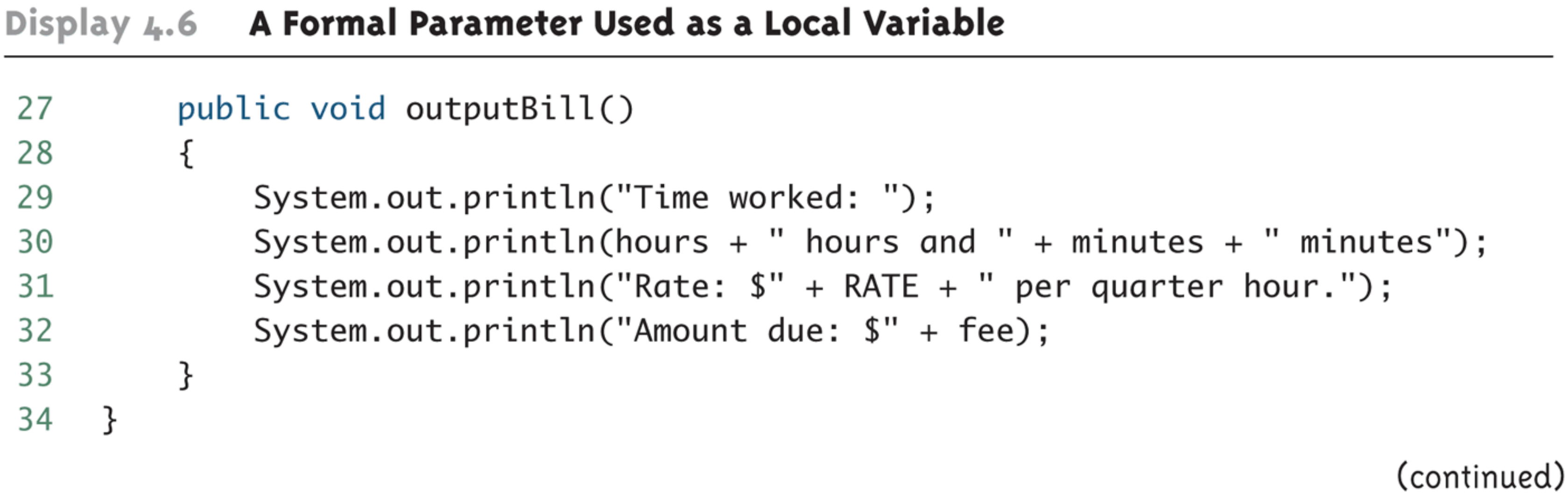
```
public void inputTimeWorked()
{
    System.out.println("Enter number of full hours worked");
    System.out.println("followed by number of minutes:");
    Scanner keyboard = new Scanner(System.in);
    hours = keyboard.nextInt();
    minutes = keyboard.nextInt();
}

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 2 of 5)



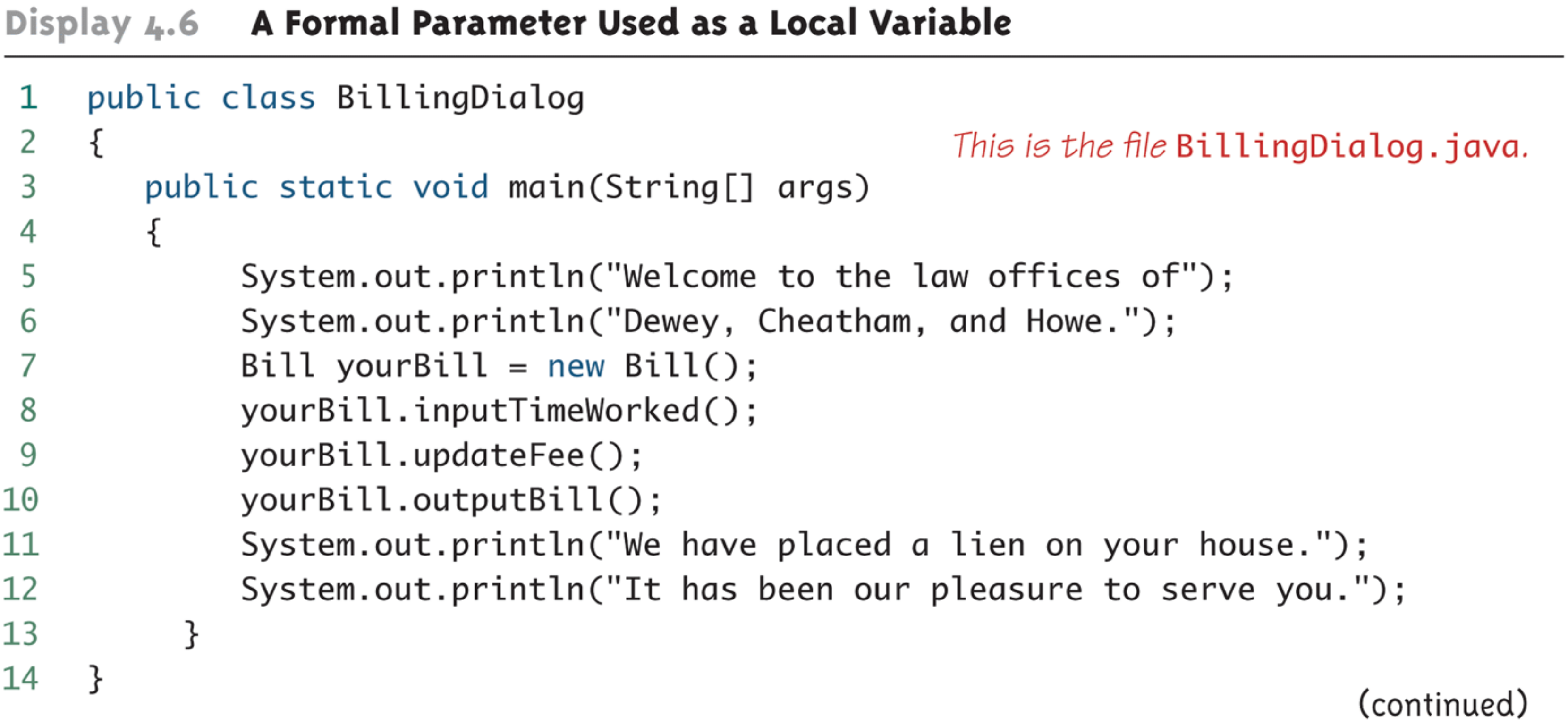
Display 4.6 A Formal Parameter Used as a Local Variable

```
    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 3 of 5)

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



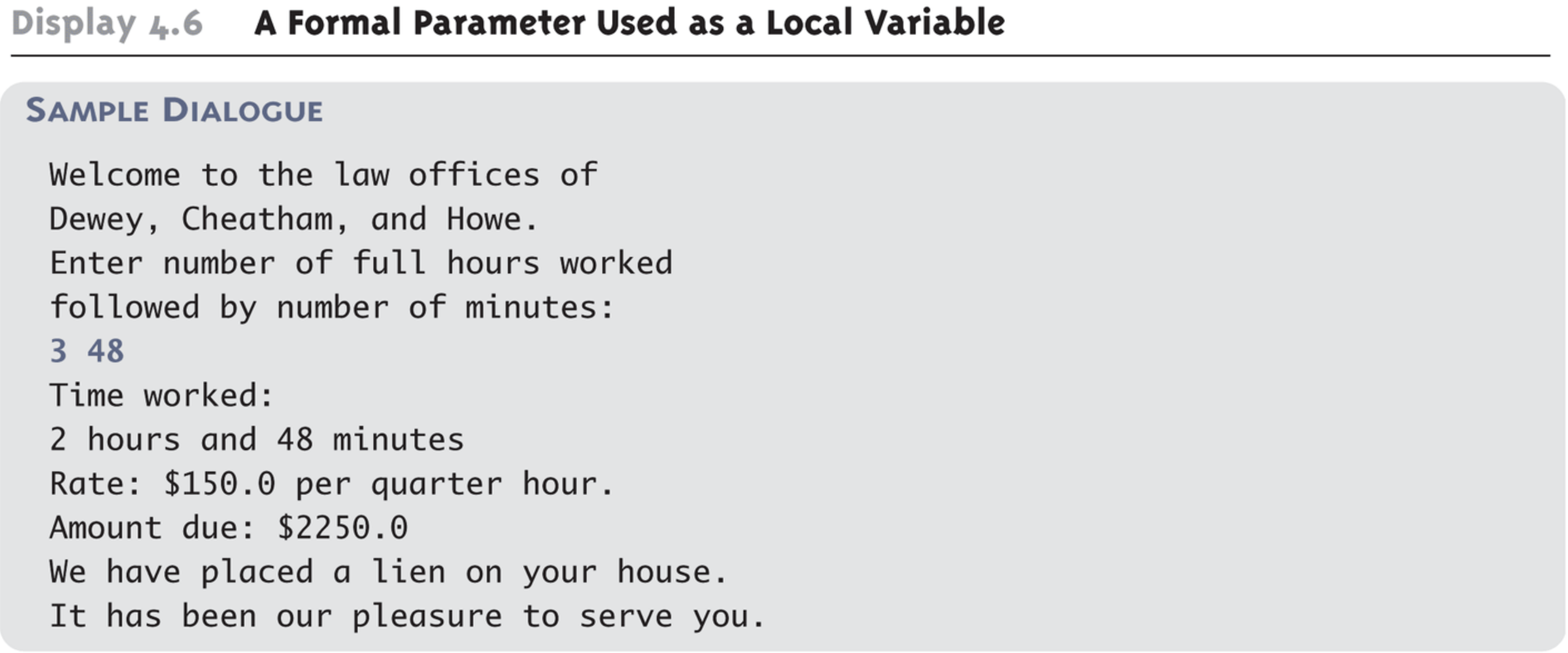
Display 4.6 A Formal Parameter Used as a Local Variable

This is the file `BillingDialog.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.");
     }
}
```

(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.
```

- 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

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

- 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

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

- What will happen if we make the following changes to the **setDate** method in Example-3?

```
public void setDate(int day, int month, int year)
{
     day = day;
     month = month;
     year = year;
}
```

Example-4: 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`**

The this Parameter

The methods equals and toString

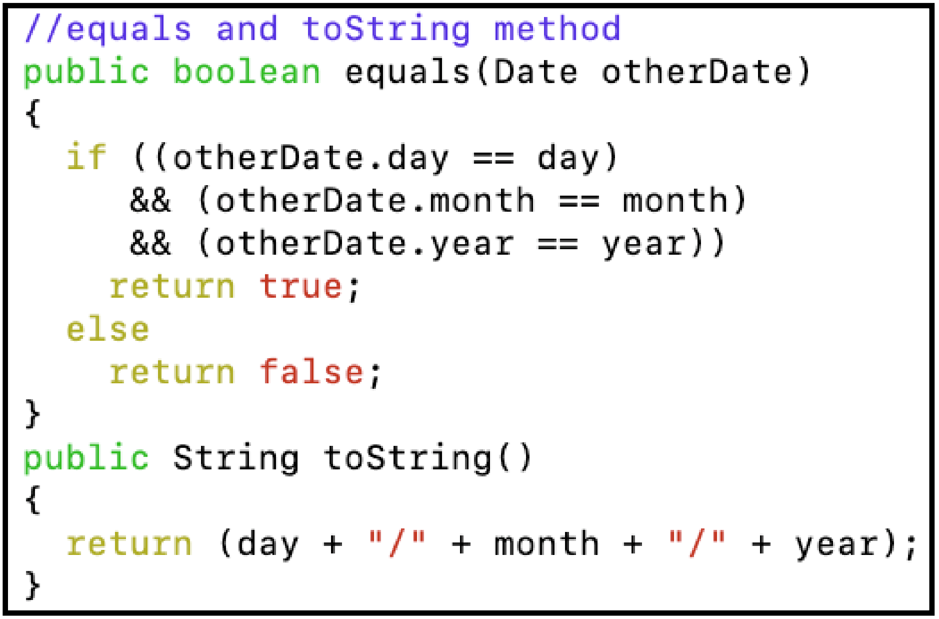
- Java expects certain methods, such as **equals** and **toString**, to be in all, or almost all, classes
- The purpose of **equals**, a **boolean** valued method, is to compare two objects of the class to see if they satisfy the notion of "being equal"
  - Note: You cannot use **==** to compare objects

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

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

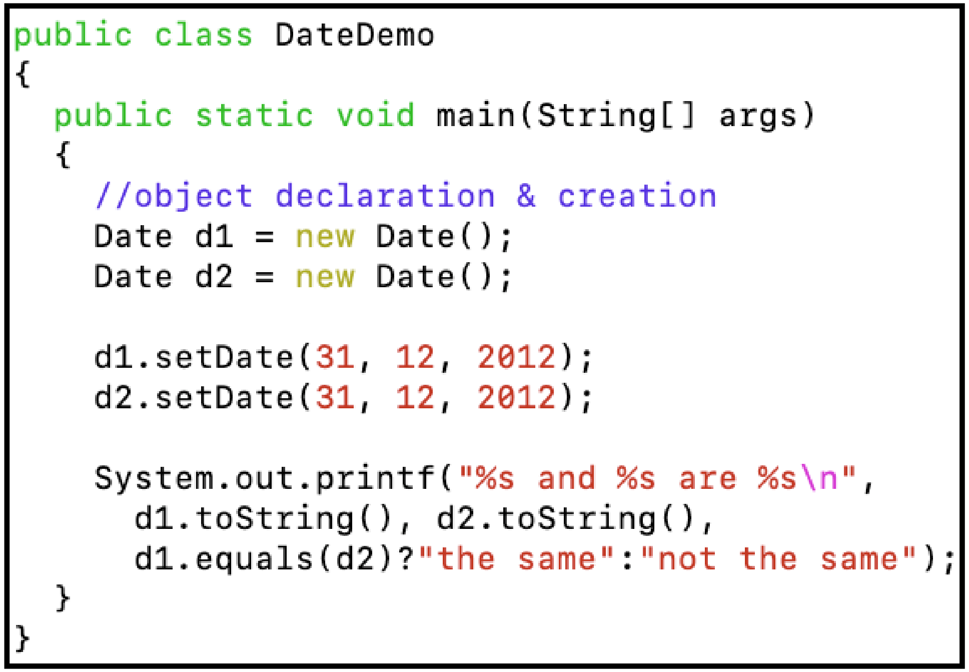
```
public String toString()
```

Example-5: equals and toString methods



```
//equals and toString method
public boolean equals(Date otherDate)
{
  if ((otherDate.day == day)
     && (otherDate.month == month)
     && (otherDate.year == year))
    return true;
  else
    return false;
}
public String toString()
{
  return (day + "/" + month + "/" + year);
}
```

Newly added methods
Inside Date.java



```
public class DateDemo
{
  public static void main(String[] args)
  {
    //object declaration & creation
    Date d1 = new Date();
    Date d2 = new Date();

    d1.setDate(31, 12, 2012);
    d2.setDate(31, 12, 2012);

    System.out.printf("%s and %s are %s\n",
      d1.toString(), d2.toString(),
      d1.equals(d2)?"the same":"not the same");
  }
}
```

DateDemo.java

- Each method should be tested in a 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

Testing Methods

- Class definitions
  - Class structure
  - Variables
  - Methods
- **Encapsulation**
  - **Access modifiers (e.g., public vs private)**
  - **Accessor and mutator methods**
- Overloading
- Constructors

Outline

- ***Information hiding*** is the practice of separating how to use a class from the details of its implementation
  - ***Abstraction*** is another term used to express the concept of discarding details in order to avoid information overload
- ***Encapsulation*** means that the data and methods of a class are combined into a single unit (i.e., a class object), which hides the implementation details
  - Knowing the details is unnecessary because interaction with the object occurs via a well-defined and simple interface

Information Hiding and Encapsulation

- The **API** or ***application programming interface*** for a class is a description of how to use the class
  - A programmer need only read the API in order to use a well designed class
- An **ADT** or ***abstract data type*** is a data type that is written using good information-hiding techniques

Important Acronyms: API and ADT

- The modifier **`public`** means that there are no restrictions on where an instance variable or method can be used
- The modifier **`private`** means that an instance variable or method cannot be accessed by name outside of the class
- It is considered good programming practice to make **all** instance variables **`private`**
- Most methods are **`public`**, and thus provide controlled access to the object
- Usually, methods are **`private`** only if used as helping methods for other methods in the class

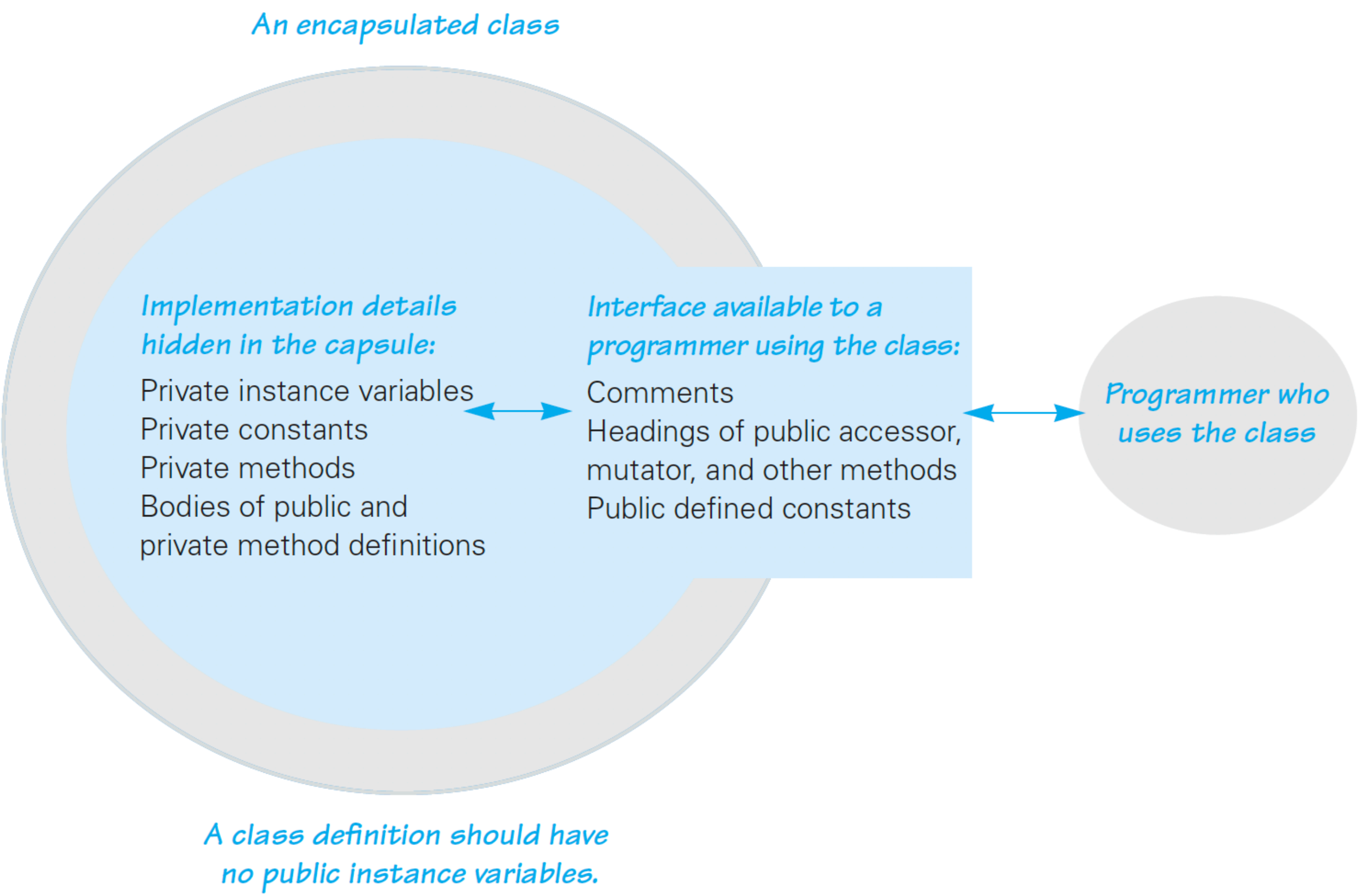
`public` and `private` Modifiers

- ***Accessor*** methods allow the programmer to obtain the value of an object's instance variables
  - The data can be accessed but not changed
  - The name of an accessor method typically starts with the word **`get`**
- ***Mutator*** methods allow the programmer to change the value of an object's instance variables in a controlled manner
  - Incoming data is typically tested and/or filtered
  - The name of a mutator method typically starts with the word **`set`**

Accessor and Mutator Methods

Encapsulation

Display 4.10 Encapsulation



An encapsulated class

Implementation details hidden in the capsule:

Private instance variables
Private constants
Private methods
Bodies of public and
private method definitions

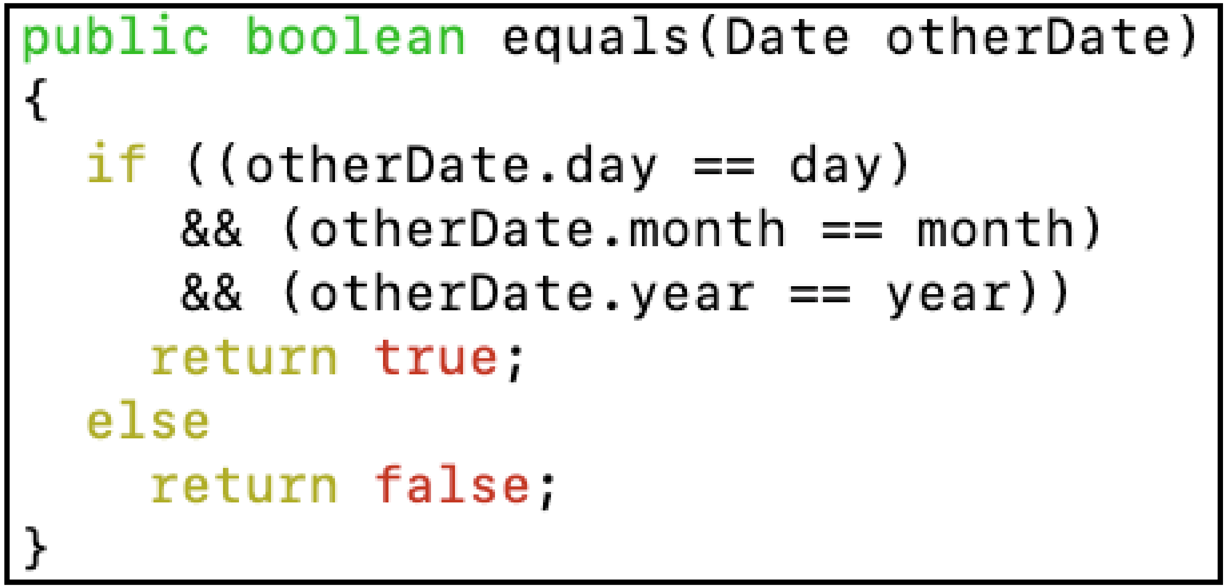
Interface available to a programmer using the class:

Comments
Headings of public accessor,
mutator, and other methods
Public defined constants

Programmer who uses the class

A class definition should have no public instance variables.

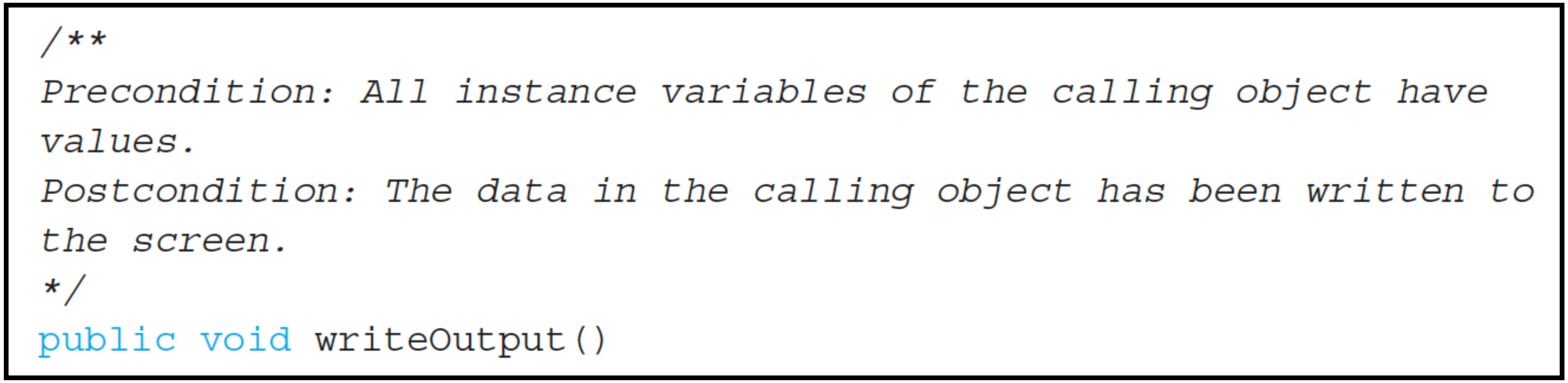
- Within the definition of a class, private members of **any** object of the class can be accessed, not just private members of the calling object



```
public boolean equals(Date otherDate)
{
  if ((otherDate.day == day)
     && (otherDate.month == month)
     && (otherDate.year == year))
    return true;
  else
    return false;
}
```

A Class Has Access to Private Members of All Objects of the Class

- The ***precondition*** of a method states what is assumed to be true when the method is called
- The ***postcondition*** of a method states what will be true after the method is executed, as long as the precondition holds
- It is a good practice to always think in terms of preconditions and postconditions when designing a method, and when writing the method comment



```
/**
Precondition: All instance variables of the calling object have
values.
Postcondition: The data in the calling object has been written to
the screen.
*/
public void writeOutput()
```

Preconditions and Postconditions

- Class definitions
  - Class structure
  - Variables
  - Methods
- Encapsulation
  - Access modifiers (e.g., public vs private)
  - Accessor and mutator methods
- **Overloading**
- Constructors

Outline

- ***Overloading*** is when two or more methods *in the same class* have the same method name
- To be valid, any two definitions of the method name must have different ***signatures***
  - A signature consists of the name of a method together with its parameter list
  - Differing signatures must have different numbers and/or types of parameters

Overloading

- If Java cannot find a method signature that exactly matches a method invocation, it will try to use automatic type conversion
- The interaction of overloading and automatic type conversion can have unintended results
- In some cases of overloading, because of automatic type conversion, a single method invocation can be resolved in multiple ways
  - Ambiguous method invocations will produce an error in Java

Overloading and Automatic Type Conversion

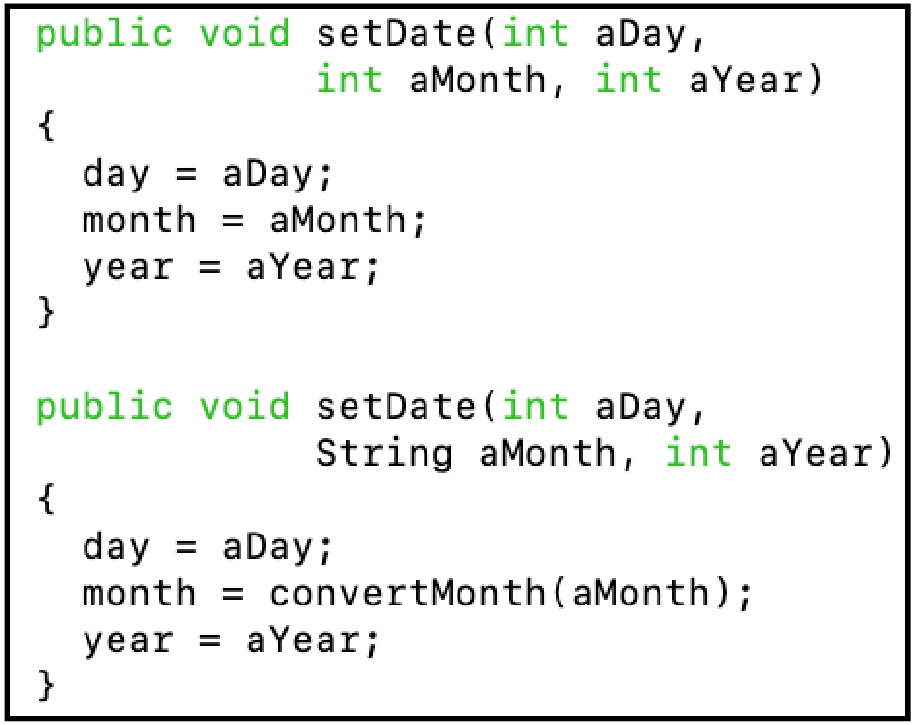
- The signature of a method only includes the method name and its parameter types
  - The signature does **not** include the type returned
- Java does not permit methods with the same name and different return types in the same class

Pitfall: You Can Not Overload Based on the Type Returned

- Although many programming languages, such as C++, allow you to overload operators (+, -, etc.), Java does not permit this
  - You may only use a method name and ordinary method syntax to carry out the operations you desire

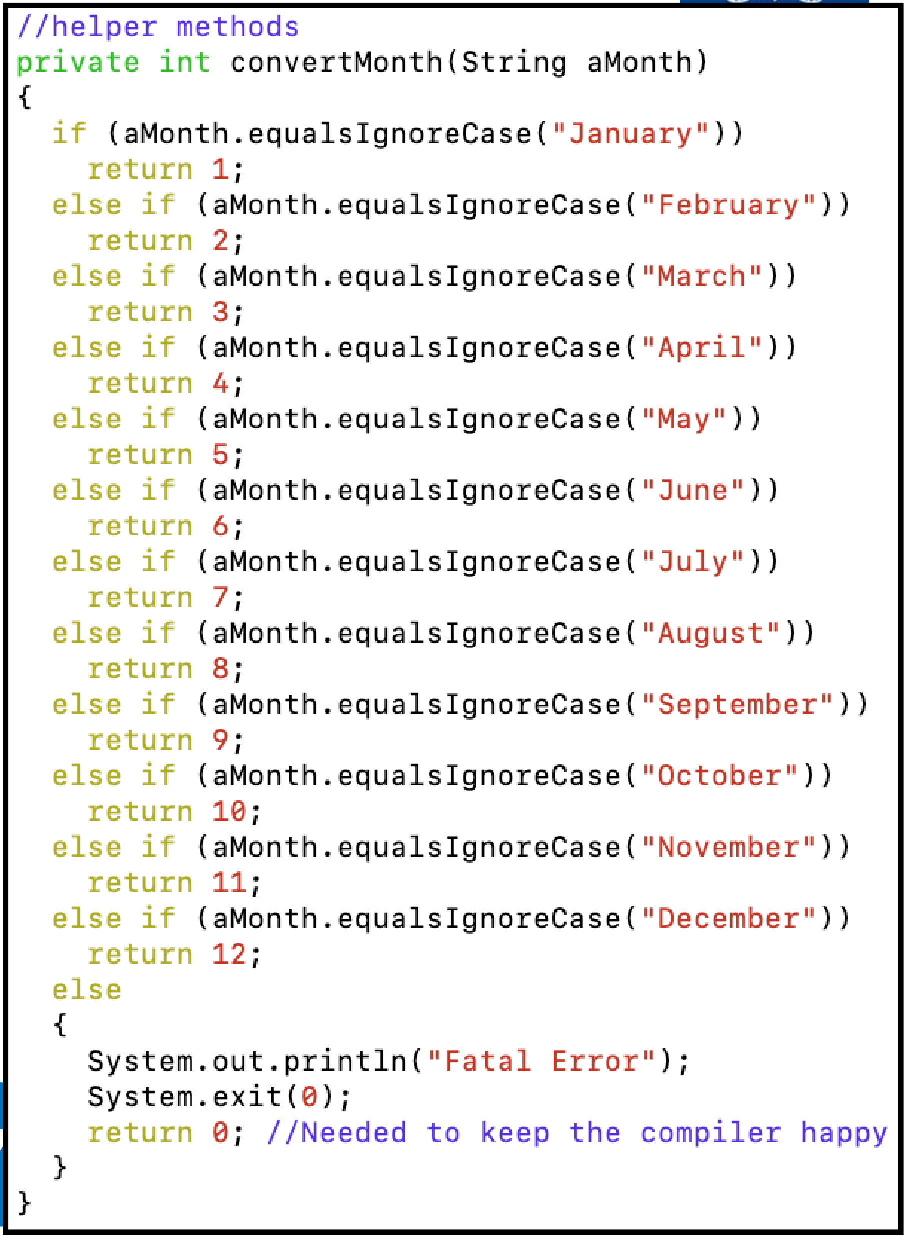
You Can Not Overload Operators in Java

Example-7



```
public void setDate(int aDay,
             int aMonth, int aYear)
{
  day = aDay;
  month = aMonth;
  year = aYear;
}

public void setDate(int aDay,
             String aMonth, int aYear)
{
  day = aDay;
  month = convertMonth(aMonth);
  year = aYear;
}
```

Two setDate methods having different signatures in Date.java



```
//helper methods
private int convertMonth(String aMonth)
{
  if (aMonth.equalsIgnoreCase("January"))
    return 1;
  else if (aMonth.equalsIgnoreCase("February"))
    return 2;
  else if (aMonth.equalsIgnoreCase("March"))
    return 3;
  else if (aMonth.equalsIgnoreCase("April"))
    return 4;
  else if (aMonth.equalsIgnoreCase("May"))
    return 5;
  else if (aMonth.equalsIgnoreCase("June"))
    return 6;
  else if (aMonth.equalsIgnoreCase("July"))
    return 7;
  else if (aMonth.equalsIgnoreCase("August"))
    return 8;
  else if (aMonth.equalsIgnoreCase("September"))
    return 9;
  else if (aMonth.equalsIgnoreCase("October"))
    return 10;
  else if (aMonth.equalsIgnoreCase("November"))
    return 11;
  else if (aMonth.equalsIgnoreCase("December"))
    return 12;
  else
  {
    System.out.println("Fatal Error");
    System.exit(0);
    return 0; //Needed to keep the compiler happy
  }
}
```

- Class definitions
  - Class structure
  - Variables
  - Methods
- Encapsulation
  - Access modifiers (e.g., public vs private)
  - Accessor and mutator methods
- Overloading
- Constructors

Outline

- A ***constructor*** is a special kind of method that is designed to initialize the instance variables for an object:
  public ClassName(anyParameters){code}
  - A constructor must have the same name as the class
  - A constructor has no type returned, not even **`void`**
  - Constructors are typically overloaded

Constructors

- A constructor is called when an object of the class is created using **new**

  ClassName objectName = new ClassName(anyArgs);

  - The name of the constructor and its parenthesized list of arguments (if any) must follow the **new** operator
  - This is the **only** valid way to invoke a constructor: a constructor cannot be invoked like an ordinary method
- If a constructor is invoked again (using **new**), the first object is discarded and an entirely new object is created
  - If you need to change the values of instance variables of the object, use mutator methods instead

Constructors

- The first action taken by a constructor is to create an object with instance variables
- Therefore, it is legal to invoke another method within the definition of a constructor, since it has the newly created object as its calling object
  - For example, mutator methods can be used to set the values of the instance variables
  - It is even possible for one constructor to invoke another

You Can Invoke Another Method in a Constructor

- Like any ordinary method, every constructor has a **`this`** parameter
- The **`this`** parameter can be used explicitly, but is more often understood to be there than written down
- The first action taken by a constructor is to automatically create an object with instance variables
- Then within the definition of a constructor, the **`this`** parameter refers to the object created by the constructor

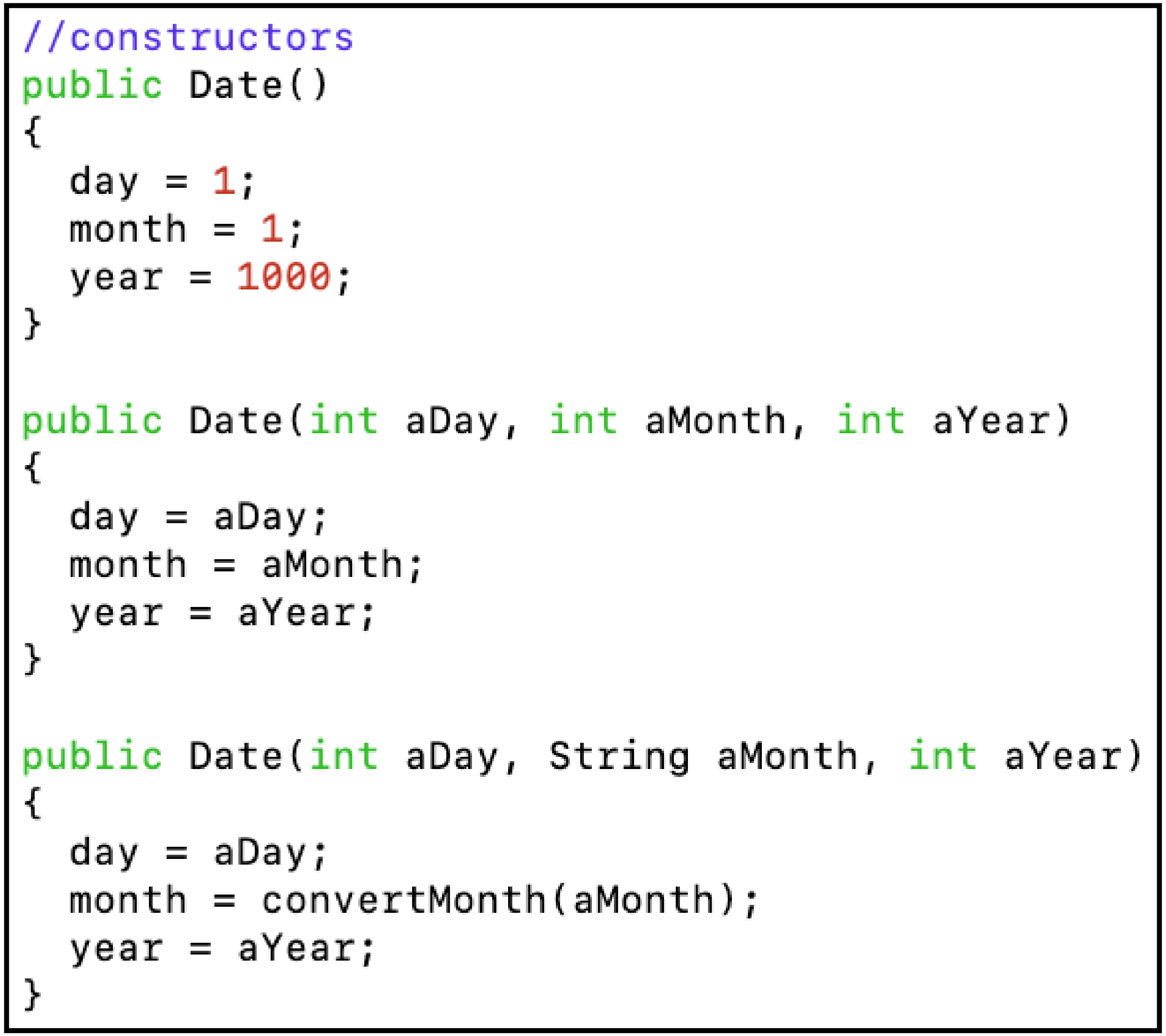
A Constructor Has a this Parameter

- If you do not include any constructors in your class, Java will automatically create a *default* or *no-argument* constructor that takes no arguments, performs no initializations, but allows the object to be created
- If you include even one constructor in your class, Java will not provide this default constructor
- If you include any constructors in your class, normally you should provide your own no-argument constructor.

Include a No-Argument Constructor

- Instance variables are automatically initialized in Java
  - **boolean** types are initialized to false
  - Other primitives are initialized to the zero of their type
  - Class types are initialized to **`null`**
- However, it is a better practice to explicitly initialize instance variables in a constructor
- **Note**: Local variables are not automatically initialized

Default Variable Initializations



```
//constructors
public Date()
{
  day = 1;
  month = 1;
  year = 1000;
}

public Date(int aDay, int aMonth, int aYear)
{
  day = aDay;
  month = aMonth;
  year = aYear;
}

public Date(int aDay, String aMonth, int aYear)
{
  day = aDay;
  month = convertMonth(aMonth);
  year = aYear;
}
```

Example-8: Constructors

- Class structure
- Instance variables and methods
- Different types of methods and their invocation
- Information hiding & Encapsulation
- Overloading methods
- Class constructors

Learning Outcomes