Advanced Programming

Defining Classes I
Introduction

• 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 Definitions

• 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** and **Scanner** 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

- 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.
  - If A is a class, then the phrases "bla is of type A," "bla is an object of the class A," and "bla is an instance of the class A" mean the same thing.
- 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

• 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
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*
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();
  ```
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:

```java
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:

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

• Invoking a method is equivalent to executing the method body
```java
public class DateFirstTry {
    public String month;
    public int day;
    public int year; // a four digit number.

    public void writeOutput() {
        System.out.println(month + " " + day + ", " + year);
    }
}
```
public class DateFirstTryDemo {
    public static void main(String[] args) {
        DateFirstTry date1, date2;
        date1 = new DateFirstTry();
        date2 = new DateFirstTry();
        date1.month = "December";
        date1.day = 31;
        date1.year = 2007;
        System.out.println("date1:");
        date1.writeObject();

        date2.month = "July";
        date2.day = 4;
        date2.year = 1776;
        System.out.println("date2:");
        date2.writeObject();
    }
}
Class Definition

The following shows the form of a class definition that is most commonly used; however, it is legal to intermix the method definitions and the instance variable declarations.

SYNTAX

```java
public class Class_Name
{
    Instance_Variable_Declaration_1
    Instance_Variable_Declaration_2
        ...
    Instance_Variable_Declaration_Last

    Method_Definition_1
    Method_Definition_2
        ...
    Method_Definition_Last
}
```
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
More About Methods

• 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:
  
  ```java
  public typeReturned methodName(paramList)
  ```

• A `void` method uses the keyword `void` in its heading to show that it does not return a value:
  
  ```java
  public void methodName(paramList)
  ```
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

```java
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
• **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

• An invocation of a method that returns a value can be used as an expression anywhere that a value of the type `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();
  ```
```java
import java.util.Scanner;

public class DateSecondTry {

    private String month;
    private int day;
    private int year; //a four digit number.

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

    public void readInput()
    {
        Scanner keyboard = new Scanner(System.in);
        System.out.println("Enter month, day, and year.");
        System.out.println("Do not use a comma.");
        month = keyboard.next();
        day = keyboard.nextInt();
    }
}
```
    year = keyboard.nextInt();

    }  

public int getDay()
{  
    return day;

};  

public int getYear()
{  
    return year;

};  

public int getMonth()
{  
if (month.equalsIgnoreCase("January"))
    return 1;
else if (month.equalsIgnoreCase("February"))
    return 2;
else if (month.equalsIgnoreCase("March"))
    return 3;
else if (month.equalsIgnoreCase("April"))
    return 4;
else if (month.equalsIgnoreCase("May"))
    return 5;
else if (month.equalsIgnoreCase("June"))
    return 6;
else if (month.equalsIgnoreCase("July"))
    return 7;
else if (month.equalsIgnoreCase("August"))
    return 8;
else if (month.equalsIgnoreCase("September"))
    return 9;
else if (month.equalsIgnoreCase("October"))
    return 10;
else if (month.equalsIgnoreCase("November"))
    return 11;
else if (month.equalsIgnoreCase("December"))
    return 12;
else
{
    System.out.println("Fatal Error");
    System.exit(0);
    return 0; //Needed to keep the compiler happy
}
return 0;
```java
public class DemoOfDateSecondTry {
    public static void main(String[] args) {
        DateSecondTry date = new DateSecondTry();
        date.readInput();

        int dayNumber = date.getDay();
        System.out.println("That is the " + dayNumber + "th day of the month.");
    }
}
```

Sample Dialogue

Enter month, day, and year.
Do not use a comma.
July 4 1776
That is the 4th day of the month.
Local Variables

• A variable declared within a method definition is called a *local variable* (e.g. *keyboard*)
  – 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
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
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

```java
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

```java
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`
Parameters of a Primitive Type

• A parameter 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

```java
1 import java.util.Scanner;

2 public class Bill {
3     public static double RATE = 150.00; // Dollars per quarter hour
4     private int hours;
5     private int minutes;
6     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
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();
}

calculateFee uses the parameter minutesWorked as a local variable.

```

```java
public double calculateFee(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;
}

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

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

(continued)
A Formal Parameter Used as a Local Variable

(Part 3 of 5)

Display 4.6  A Formal Parameter Used as a Local Variable

```java
27       public void outputBill()
28           {
29               System.out.println("Time worked: ");
30               System.out.println(hours + " hours and " + minutes + " minutes");
31               System.out.println("Rate: $" + RATE + " per quarter hour.");
32               System.out.println("Amount due: $" + fee);
33           }
34       }
```

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

Display 4.6   A Formal Parameter Used as a Local Variable

1   public class BillingDialog
2      {
3             public static void main(String[] args)
4         {
5                 System.out.println("Welcome to the law offices of");
6                 System.out.println("Dewey, Cheatham, and Howe.");
7                 Bill yourBill = new Bill();
8                 yourBill.inputTimeWorked();
9                 yourBill.updateFee();
10                yourBill.outputBill();
11                System.out.println("We have placed a lien on your house.");
12                System.out.println("It has been our pleasure to serve you.");
13         }
14     }

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: $150.00 per quarter hour.
Amount due: $2250.00

We have placed a lien on your house.
It has been our pleasure to serve you.
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`.
• If today is of type DateSecondTry, then

```java
today.writeObject();
```

is equivalent to

```java
System.out.println(today.month + " " + today.day + ", " + today.year);
```

This is because, although the definition of `writeObject` reads

```java
public void writeObject()
{
    System.out.println(month + ", " + day + ", " + year);
}
```

it really means

```java
public void writeObject()
{
    System.out.println(<the calling object>.month + " " + <the calling object>.day + ", " + <the calling object>.year);
}
```
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.

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

```java
public void writeOutput()
{
    System.out.println(this.month + " " + this.day
                        + ", " + this.year);
}
```
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**.
Not Correct

```java
public void setDate(int month, int day, int year) //Not correct
{
    month = monthString(month);
    day = day;
    year = year;
}
```
public void setDate(int month, int day, int year)
{
    this.month = monthString(month);
    this.day = day;
    this.year = year;
}
Material Covered

• Chapter 4: pages 200 – 225
  – Absolute Java, 4th edition, Savitch