STUDENT OUTLINE

Lesson 6 – Defining and Using Classes

INTRODUCTION: The previous lessons have discussed how to use objects and their methods. But all the objects have been created using a class from one of the Java libraries or curriculum supplied classes. This lesson discusses how to define your own classes and objects.

The key topics for this lesson are:

A. Designing a Class
B. Determining Object Behavior
C. Instance Variables
D. Implementing Methods
E. Constructors
F. Using Classes

VOCABULARY:

ATTRIBUTES	ACCESS SPECIFIER
BEHAVIORS	CONSTRUCTOR
ENCAPSULATION	INSTANCE VARIABLE
METHOD CALLS	OVERLOADED
REFERENCE	

DISCUSSION:

A. Designing a Class

1. One of the advantages of object-oriented design is it allows a programmer to create a new abstract data type that is reusable in other situations

2. When designing a new data type, two components must be identified - attributes and behaviors.

3. Consider the icons used in computer operating systems. The attributes that describe the icon are things like a graphic pattern, colors, size, name, and its position on the screen. Some of its behaviors include changing color and moving its position.

4. The attributes of an object are the nouns that describe that object. For example, in our checking account example below, the attributes are "current balance" and "account number". These will become the private data members of a class.

5. The behaviors of an object are the verbs that denote the actions of that object or what it does. For example, in our checking account example below, behaviors are "accept a deposit", "process a check", etc. These will become

the member functions of a class. In a Java program, behaviors of an object are described by methods.

B. Determining Object Behavior

1. In this section, you will learn how to create a simple class that describes the behavior of a bank account. Before you start programming, you need to understand how the objects of your class behave. Operations that can be carried out with a checking account could consist of:

   - Accept a deposit
   - Withdraw from the account
   - Get the current balance

2. In Java, these operations are expressed as *method calls*. For example, assume we have an object `checking` of type `CheckingAccount`. The methods that invoke the required behaviors

   ```
   checking.deposit(1000)
   checking.withdraw(250)
   System.out.println("Balance: " + checking.getBalance());
   ```

   are represented by the set of methods

   - `deposit`
   - `withdraw`
   - `getBalance`

   These methods form the behavior of the `CheckingAccount` class. The behavior is the complete list of the methods that you can apply to objects of a given class. An object of type `CheckingAccount` can be viewed as a "black box" that can carry out its methods.

3. To construct objects of the `CheckingAccount` class, it is necessary to declare an object variable

   ```
   CheckingAccount checking;
   ```

   Object variables such as `checking` are *references* to objects. Instead of holding an object itself, a reference variable holds the information necessary to find the object in memory.

4. This object variable `checking` does not refer to any object at all. An attempt to invoke a method on this variable would cause the compiler to generate an error indicating that the variable had not been initialized. To initialize the variable, it is necessary to create a new `CheckingAccount` object using the **`new`** operator

   ```
   checking = new CheckingAccount();
   ```

   This call creates a new object and returns a reference to the newly created object. To use an object, you must assign that reference to an object variable.

5. We will implement (that is, create and write the code for) the `CheckingAccount` object so that the account has an initial balance of 1000.0 dollars.

```
// open a new account
double initialDeposit = 1000.0;
CheckingAccount checking = new CheckingAccount();

// set initial balance to 1000.0
checking.deposit(intialDeposit);
```

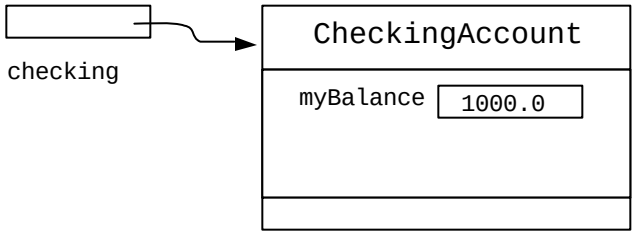


Figure 6-1. Creating a New Object

6. Objects of the `CheckingAccount` class can be used to carry out meaningful tasks without knowing how the `CheckingAccount` objects store their data or how the `CheckingAccount` methods do their work. This is an important aspect of object-oriented programming.

7. Once we understand how to use objects of the `CheckingAccount` class, it is possible to design a Java class that implements its behaviors. To describe object behavior, you first need to implement a class, and then implement methods within that class.

```
public class CheckingAccount
{
  // CheckingAccount data

  // CheckingAccount constructors

  // CheckingAccount methods
}
```

Next we implement the three methods that have already been identified:

- `deposit`
- `withdraw`
- `getBalance`

```
public class CheckingAccount
{
  // CheckingAccount data

  // CheckingAccount constructors
```

```
public void deposit( double amount )
{
  // method implementation
}
```

```
  public void withdraw( double amount )
  {
    // method implementation
  }

  public double getBalance()
  {
    // method implementation
  }

}
```

8. A method header consists of the following parts:

   access_specifier return_type *method_name* (*parameters*)

   a. An ***access_specifier*** (such as **public**). The access specifier controls which other methods can call this method. Most methods should be declared as public so all other methods in your program can call them.

   b. The ***return_type*** of the method (such as **double** or **void**). The return type is the type of the value that the method computes. For example, in the `CheckingAccount` class, the `getBalance` method returns the current account balance, which is a floating-point number, so its return type is **double**. The `deposit` and `withdraw` methods don't return any value. To indicate that a method does not return a value, you use the special type **void**.

   c. The *method_name* (such as `deposit`).

   d. A list of the *parameters* of the method. The parameters are the input to the method. The `deposit` and `withdraw` methods each have one parameter, the amount of money to deposit or withdraw. The type of parameter, such as **double**, and name for each parameter, such as `amount`, must be specified. If a method has no parameters, like `getBalance`, it is still necessary to supply a pair of parentheses `()` behind the method name.

9. Once the method header has been specified, the implementation of the method must be supplied in a block that is delimited by braces `{...}`. The `CheckingAccount` methods will be implemented later in Section D.

C. Instance Variables

1. Each object must store its current *state*. The state is the set of values that describe the object and that influence how an object reacts to method calls. In the case of our checking account objects, the state is the current balance and an account identifier.

2. Each object stores its state in one or more *instance variables*.

```
public class CheckingAccount
{
  ...
  private double myBalance;
  private String myAccountNumber;

  // CheckingAccount methods
}
```

3. An instance variable declaration consists of the following parts:

access_specifier type *variable_name*

   a. An ***access_specifier*** (such as **private**). Instance variables are generally declared with the access specifier **private**. That means they can be accessed only by methods of the *same class*, not by any other method. In particular, the balance variable can be accessed only by the `deposit`, `withdraw`, and `getBalance` methods.
   b. The ***type*** of the variable (such as **double**).
   c. The *variable_name* (such as `myBalance`).

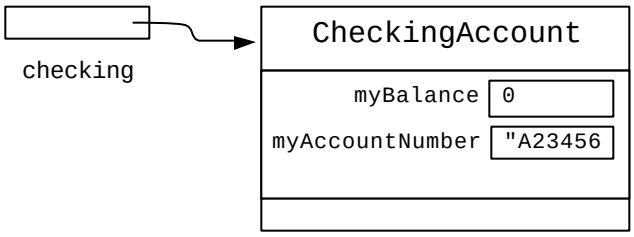


Figure 6-2. Instance Variables

4. If instance variables are declared private, then all data access must occur through the public methods. This means that the instance variables of an object are effectively hidden from the programmer who only uses a class. They are available only to the programmer who implements the class, that is, the one who writes or revises the methods. The process of hiding data is called *encapsulation*. Although it is possible in Java to define instance variables as **public** (leave them unencapsulated), it is very uncommon in practice. We will always make instance variables **private** in this curriculum guide.

5. For example, because the `myBalance` instance variable is **private**, it cannot be accessed in other code:

```
double balance = checking.mybalance;  // compiler ERROR!
```

However, the public `getBalance` method to inquire about the balance can be called:

```
double balance = checking.getBalance();  // OK
```

D. Implementing Methods

1. An implementation must be provided for every method of the class. The implementation for three methods of the `CheckingAccount` class is given below.

```
public class CheckingAccount
{
  private double myBalance;
  private String myAccountNumber;

  public double getBalance()
  {
    return myBalance;
  }

  public void deposit(double amount)
  {
    myBalance =  myBalance + amount;
  }

  public void withdraw(double amount)
  {
    myBalance =  myBalance – amount;
  }
}
```

2. The implementation of the methods is straightforward. When some amount of money is deposited or withdrawn, the balance increases or decreases by that amount.

3. The `getBalance` method simply *returns* the current balance. A **`return`** statement obtains the value of a variable and exits the method immediately. The return value becomes the value of the method call expression. The syntax of a **`return`** statement is:

```
return expression;
```

or

```
return;  // Exits the method without bringing back a value
```

E. Constructors

1. The final requirement to implement the `CheckingAccount` class is to define a *constructor*, whose purpose is to initialize the values of instance variables of an object.

```
public class CheckingAccount
{
  ...
```

```
  public CheckingAccount()  // constructor
  {
    myBalance = 0;
    myAccountNumber = "NEW";
  }
  ...
}
```

2. Constructors always have the same name as their class. Similar to methods, constructors are generally declared as public to enable any code in a program to construct new objects of the class. Unlike methods, constructors do not have return types.

3. Constructors are always invoked together with the **new** operator:

```
new CheckingAccount();
```

The **new** operator allocates memory for the objects, and the constructor initializes it. The value of the new operator is the reference to the newly allocated and constructed object.

In most cases, you want to declare and store a reference to an object in an object variable as follows:

```
CheckingAccount checking = new CheckingAccount();
```

4. If you do not initialize an instance variable that is a number, it is initialized automatically to zero. Even though, initialization is handled automatically for instance variables, it's a matter of good style to initialize all instance variables explicitly.

5. Many classes have more than one constructor. For example, you can supply a second constructor for the `CheckingAccount` class that sets the `myBalance` and `accountNumber` instance variables to initial values, which are the parameters of the constructor:

```
public class CheckingAccount
{
  ...

  public CheckingAccount() // constructor defines values
  {
    myBalance = 0;
    myAccountNumber = "NEW";
  }

  public CheckingAccount(double initialBalance, String acctNum)
   // contructor gets values elsewhere
  {
    myBalance = initialBalance;
    myAccountNumber = acctNum;
  }
  ...
}
```

The second constructor is used if you supply a number and a string as construction parameters.

```
CheckingAccount checking = new CheckingAccount(5000, "A123");
```

6. Note that in the above example there are two constructors of the same name. Whenever you have multiple methods (or constructors) with the same name, the name is said to be *overloaded*. The compiler figures out which one to call by looking at the parameters of each method.

   For example, if you construct a new `checkingAccount` object with

   ```
   CheckingAccount checking = new CheckingAccount();
   ```

   then the compiler picks the first constructor. If you construct an object with

   ```
   CheckingAccount checking = new CheckingAccount(5000, "A123");
   ```

   then the compiler picks the second constructor.

7. The implementation of the `CheckingAccount` class is complete and is given below:

```
public class CheckingAccount
{
  private double myBalance;
  private String myAccountNumber;

  public CheckingAccount()
  {
    myBalance = 0;
    myAccountNumber = "NEW";
  }

  public CheckingAccount(double initialBalance, String acctNum)
  {
    myBalance = initialBalance;
    myAccountNumber = acctNum;
   }

  public double getBalance()
  {
    return myBalance;
  }

  public void deposit(double amount)
  {
    myBalance =  myBalance + amount;
  }

  public void withdraw( double amount )
  {
    myBalance =  myBalance – amount;
  }
}
```

F. Using Classes

1. Using the `CheckingAccount` class is best demonstrated by writing a program that solves a specific problem. We want to study the following scenario:

   > A interest bearing checking account is created with a balance of $1,000. For two years in a row, add 2.5% interest. How much money is in the account after two years?

2. Two classes are required: the `CheckingAccount` class that was developed in the preceding sections, and a second class called `CheckingTester`. The main method of the `CheckingTester` class constructs a `CheckingAccount` object, adds the interest twice, then prints out the balance.

```
class CheckingTester
{
  public static void main(String[] args)
  {
    CheckingAccount checking =
        new CheckingAccount(1000,"A123");

    final double INTEREST_RATE = 2.5;
    double interest;

    interest = checking.getBalance() * INTEREST_RATE / 100;
    checking.deposit(interest);

    System.out.println("Balance after year 1 is $"
        + checking.getBalance());

    interest = checking.getBalance() * INTEREST_RATE / 100;
    checking.deposit(interest);

    System.out.println("Balance after year 2 is $"
        + checking.getBalance());
  }
}
```

3. The classes can be distributed over multiple files or kept together in a single file. If kept together, the class with the `main` method must be declared as **`public`**. The **`public`** attribute cannot be specified for any other class in the same file since a Java source file can contain only one **`public`** class.

4. Care must be taken to ensure that the name of the file matches the name of the public class. For example, a single file containing both the `CheckingAccount` class and the `CheckingTester` class must be contained in a file called `CheckingTester.java`, not `CheckingAccount.java`.

SUMMARY/ REVIEW: The topics in this lesson are critical in your study of computer science. The concepts of abstraction and object-oriented programming (OOP) will continue to be developed in future lessons. Before you solve the lab exercise, you are encouraged to play with the `CheckingAccount` class and implement objects using all the behaviors of the class.

ASSIGNMENT: Lab Exercise, L.A.6.1, *MPG*
Lab Exercise, L.A.6.2, *Rectangle*

LAB EXERCISE

MPG

Background:

1. Professional programmers carefully design the classes they need before any coding is done. With well-designed classes, programming is much easier and the program has fewer bugs. Object-oriented design consists of deciding what classes are needed, what data they will hold, and how they will behave. All these decisions are documented (written up) and then examined. If something doesn't look right, it is fixed before any programming is done.

2. The specifications of a class that models the fuel efficiency of a car would be:

   Variables

```
int myStartMiles;        // Starting odometer reading
int myEndMiles;          // Ending odometer reading
double myGallonsUsed;    // Gallons of gas used between the readings
```

   Constructors

```
// Creates a new instance of a Car object with the starting
//   odometer readings.
Car(int odometerReading)
```

   Methods

```
// Simulates filling up the tank. Record the current odometer reading
//   and the number of gallons to fill the tank
void fillUp(int odometerReading, double gallons)

// Calculates and returns the miles per gallon for the car.
double calculateMPG()
```

Assignment:

1. Implement a `Car` class with the following properties.

   a. A `Car` keeps track of the start odometer reading, ending odometer reading, and the number of gallons used between readings.

   b. The initial odometer reading is specified in the constructor

   c. A method `calculateMPG` calculates and returns the mile per gallon for the car..

   d. A method `fillup` simulates filling up the tank at a gas station: `odometerReading` is the current odometer reading and `gallons` is the number of gallons that filled the tank. Save these values in instance variables.

e. With this information, miles per gallon can be calculated. Write the method so that it updates the instance variables each time it is called (simulating another visit to the pumps). After each call, `calculateMPG` will calculate the latest miles per gallon.

2. Write a testing class with a `main` method that constructs a car and calls `fillUp` and `calculateMPG` a few times. Sample usage would be

```
Car auto = new Car(15);  // initial odometer reading of 15 miles
auto.fillUp(250, 10);    // odometer is at 250 miles
                         //   fillup with 10 gallons of gas
                         // repeat auto.fillup line for additional fillups

System.out.println(auto.calculateMPG()) // print miles per gallon
```

3. Write a testing class with a `main` method that constructs a car and calls `fillUp` and `calculateMPG` a few times. A sample run of the program would give (values in ***bold italics*** represent input from the user):

```
New car odometer reading: 15

Filling Station Visit
  odometer reading: 250
  gallons to fill tank: 10

Miles per gallon: 23.50

Filling Station Visit
  odometer reading: 455
  gallons to fill tank: 12.5

Miles per gallon: 16.40
```

4. Format the output as shown. Miles per gallon should be rounded to 2 decimal places.

5. Turn in the source code with the run output attached. It is recommended that the `Car` class and the testing class be combined in one source file (*MilesPerGallon.java*)

LAB EXERCISE

RECTANGLE

Background:

1. Professional programmers carefully design the classes they need before any coding is done. With well-designed classes, programming is much easier and the program has fewer bugs. Object-oriented design consists of deciding what classes are needed, what data they will hold, and how they will behave. All these decisions are documented (written up) and then examined. If something doesn't look right, it is fixed before any programming is done.

2. The specifications of a class that models a rectangular shape would be:

   Variables

```
private double myX;      // the x coordinate of the rectangle
private double myY;      // the y coordinate of the rectangle
private double myWidth;  // the width of the rectangle
private double myHeight; // the height of the rectangle

// Creates a 500 x 500 SketchPad with a DrawingTool, pen, that is used
// to display Rectangle objects. The Drawingtool is declared static
// so that multiple Rectangle objects can be drawn on the Sketchpad
// at the same time.
private static DrawingTool pen =
   new DrawingTool(new SketchPad(500, 500));
```

   Constructors

```
// Creates a default instance of a Rectangle object with all dimensions
//   set to zero.
Rectangle()

// Creates a new instance of a Rectangle object with the left and right
// edges of the rectangle at x and x + width. The top and bottom edges
// are at y and y + height.
Rectangle(double x, double y, double width, double height)
```

   Methods

```
// calculates and returns the perimeter of the rectangle
public double getPerimeter()

// Calculates and returns the are of the rectangle.
public double getArea()

// Draws a new instance of a Rectangle object with the left and right
// edges of the rectangle at x and x + width. The top and bottom edges
// are at y and y + height.
public void draw()
```

Assignment:

1. Implement a `Rectangle` class with the following properties.

   a. A `Rectangle` object is specified in the constructor with the left and right edges of the rectangle at `x` and `x + width`. The top and bottom edges are at `y` and `y + height`.

   b. A method `getPerimeter` calculates and returns the perimeter of the `Rectangle`.

   c. A method `getArea` calculates and returns the area of the `Rectangle`.

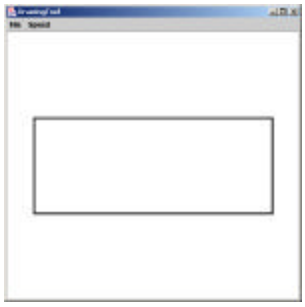
   d. A method `draw` displays a new instance of a `Rectangle` object. Refer to handout, *H.A.1.1 – DrawingTool*, for details on `DrawingTools` methods.

2. Write a testing class with a `main` method that constructs a `Rectangle` and calls `getPerimeter` and `getArea` for each `Rectangle` created. Sample usage would be:

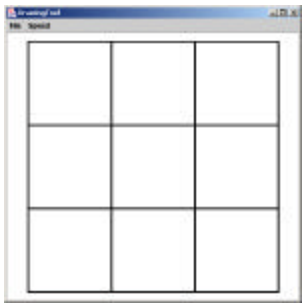
```
// Construct a 400 x 160 rectangle at location -200, -80.
Rectangle rectA = new Rectangle(-200, -80, 400, 160);
rectA.draw();  // draw the rectangle

System.out.println("Perimeter = " + rectA.getPerimeter());
System.out.println("Area = " + rectA.getArea());
```

   The resulting images would be similar to the one shown below:



3. Construct a 3x3 grid of `Rectangle` objects as show below. You should be able to produce the grid with only 3 rectangles. In addition, calculate and display the perimeter and area of the rectangles.



4. Turn in the source code with the run output attached. It is recommended that the `Rectangle` class and the testing class be combined in one source file (*RectangleTest.java*).