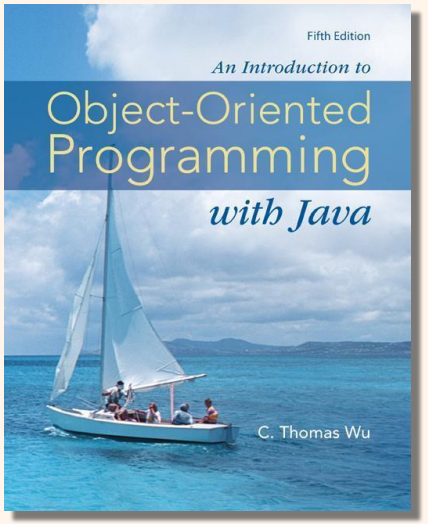


Chapter 4

Defining Your Own Classes
Part 1

Objectives

After you have read and studied this chapter, you should be able to

- Define a class with multiple methods and data members
- Differentiate the local and instance variables
- Define and use value-returning methods
- Distinguish private and public methods
- Distinguish private and public data members
- Pass both primitive data and objects to a method

Why Programmer-Defined Classes

- Using just the String, GregorianCalendar, JFrame and other standard classes will not meet all of our needs. We need to be able to define our own classes customized for our applications.
- Learning how to define our own classes is the first step toward mastering the skills necessary in building large programs.
- Classes we define ourselves are called programmer-defined classes.

First Example: Using the Bicycle Class

```
class BicycleRegistration {
    public static void main(String[] args) {
        Bicycle bike1, bike2;
          String  owner1, owner2;

        bike1 = new Bicycle( );   //Create and assign values to bike1
        bike1.setOwnerName("Adam Smith");

        bike2 = new Bicycle( );   //Create and assign values to bike2
        bike2.setOwnerName("Ben Jones");

          owner1 = bike1.getOwnerName( ); //Output the information
          owner2 = bike2.getOwnerName( );

        System.out.println(owner1 + " owns a bicycle.");
        System.out.println(owner2 + " also owns a bicycle.");
    }
}
```

The Definition of the Bicycle Class

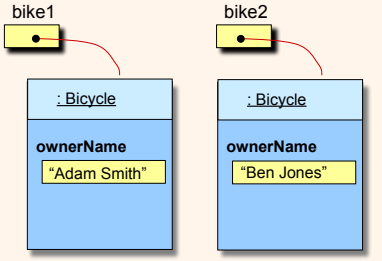
```
class Bicycle {

    // Data Member
    private String ownerName;

     //Constructor: Initialzes the data member
     public void Bicycle( ) {
          ownerName = "Unknown";
     }

    //Returns the name of this bicycle's owner
    public String getOwnerName( ) {

        return ownerName;
    }

    //Assigns the name of this bicycle's owner
    public void setOwnerName(String name) {

        ownerName = name;
    }
}
```


Multiple Instances

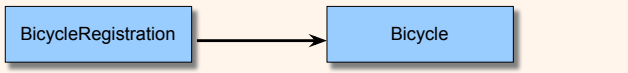
- Once the Bicycle class is defined, we can create multiple instances.

```
Bicycle bike1, bike2;

bike1 = new Bicycle( );
bike1.setOwnerName ("Adam
Smith");
bike2 = new Bicycle( );
bike2.setOwnerName ("Ben Jones");
```

Sample Code



The Program Structure and Source Files



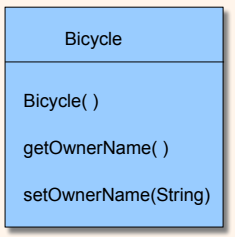

BicycleRegistration.java


Bicycle.java

There are two source files. Each class definition is stored in a separate file.

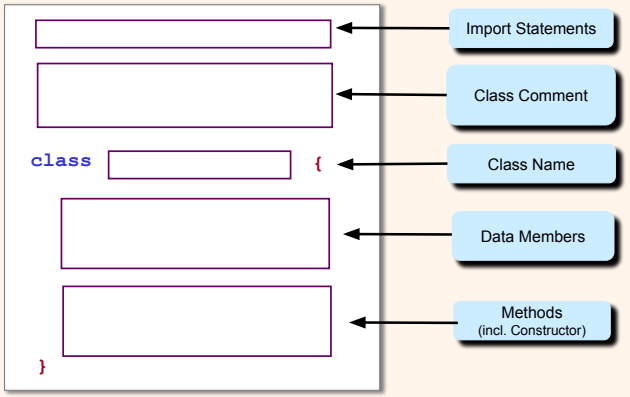
To run the program: 1. javac Bicycle.java (compile)
2. javac BicycleRegistration.java (compile)
3. java BicycleRegistration (run)

Class Diagram for Bicycle



Method Listing
We list the name and the data type of an argument passed to the method.

Template for Class Definition



Data Member Declaration

```
<modifiers>  <data type> <name> ;
```

Modifiers → `private` | Data Type → `String` | Name → `ownerName ;`

Note: There's only one modifier in this example.

Method Declaration

```
<modifier>  <return type>  <method name>  ( <parameters>  ) {
      <statements>
}
```

Modifier → `public` | Return Type → `void` | Method Name → `setOwnerName` | Parameter → `( String name )`

```
{
    ownerName = name;
}
```

Statements → `ownerName = name;`

Constructor

- A *constructor* is a special method that is executed when a new instance of the class is created.

```
public <class name> ( <parameters> ){
    <statements>
}
```

Modifier → `public` | Class Name → `Bicycle` | Parameter → `( )`

```
public Bicycle ( ) {
    ownerName = "Unassigned";
}
```

Statements → `ownerName = "Unassigned";`

Second Example: Using Bicycle and Account

```
class SecondMain {

    //This sample program uses both the Bicycle and Account classes

    public static void main(String[] args) {

        Bicycle bike;
        Account acct;

        String  myName = "Jon Java";

        bike = new Bicycle( );
        bike.setOwnerName(myName);

        acct = new Account( );
        acct.setOwnerName(myName);
        acct.setInitialBalance(250.00);

        acct.add(25.00);
        acct.deduct(50);

        //Output some information
        System.out.println(bike.getOwnerName() + " owns a bicycle and");
        System.out.println("has $ " + acct.getCurrentBalance() +
                " left in the bank");

    }
}
```


The Account Class

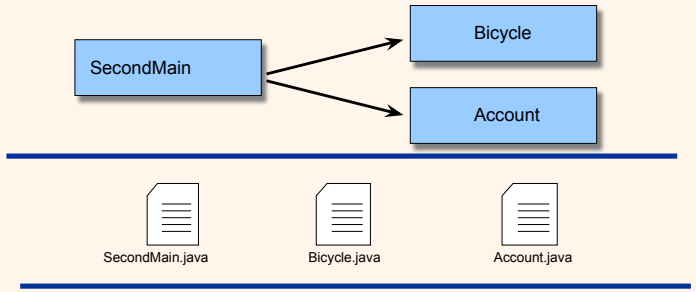
```
class Account {

    private String ownerName;

    private double balance;

    public Account( ) {
         ownerName = "Unassigned";
         balance = 0.0;
    }

    public void add(double amt) {
        balance = balance + amt;
    }

    public void deduct(double amt) {
        balance = balance - amt;
    }

    public double getCurrentBalance( ) {
         return balance;
    }
```

Page 1

```
    public String getOwnerName( ) {

         return ownerName;
    }

    public void setInitialBalance
                              (double bal) {

          balance = bal;
    }

    public void setOwnerName
                          (String name) {

         ownerName = name;
    }
}
```

Page 2

The Program Structure for SecondMain



To run the program:
1. javac Bicycle.java (compile)
2. javac Account.java (compile)
2. javac SecondMain.java (compile)
3. java SecondMain (run)

Note: You only need to compile the class once. Recompile only when you made changes in the code.

Arguments and Parameters

```
class Sample {
    public static void
           main(String[] arg) {

      Account acct = new Account();
      . . .

      acct.add(400);
      . . .
    }
    . . .
}
```

argument

```
class Account {
    . . .
    public void add(double amt) {
        balance = balance + amt;
    }
    . . .
}
```

parameter

- An argument is a value we pass to a method
- A parameter is a placeholder in the called method to hold the value of the passed argument.

Matching Arguments and Parameters

```
Demo demo = new Demo( );
int i = 5; int k = 14;
demo.compute(i, k, 20);
```

3 arguments

Passing Side

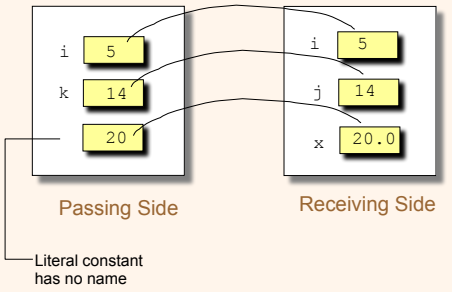
```
class Demo {
  public void compute(int i, int j, double x) {
    . . .
  }
}
```

3 parameters

Receiving Side

- The number or arguments and the parameters must be the same
- Arguments and parameters are paired left to right
- The matched pair must be assignment-compatible (e.g. you cannot pass a double argument to a int parameter)

Memory Allocation



- Separate memory space is allocated for the receiving method.
- Values of arguments are passed into memory allocated for parameters.

Passing Objects to a Method

- As we can pass int and double values, we can also pass an object to a method.
- When we pass an object, we are actually passing the reference (name) of an object
  - it means a duplicate of an object is NOT created in the called method

Passing a Student Object

```
LibraryCard card2;
card2 = new LibraryCard();
card2.setOwner(student);
```

Passing Side

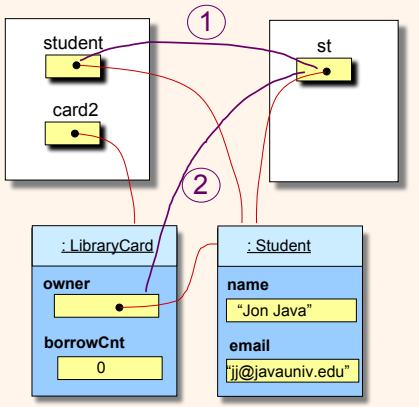
(1)

```
class LibraryCard {
  private Student owner;
  public void setOwner(Student st) {
    owner = st;  (2)
  }
}
```

Receiving Side

(1) Argument is passed

(2) Value is assigned to the data member



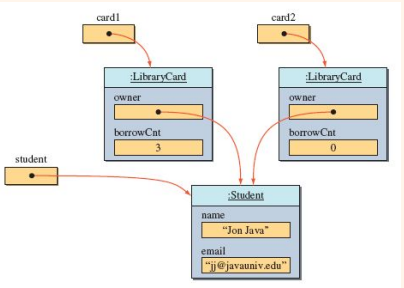
State of Memory

Sharing an Object

- We pass the same Student object to card1 and card2

```
Student    student;
LibraryCard card1, card2;

student = new Student( );
student.setName('Jon Java');
student.setEmail('jj@javauniv.edu");

card1 = new LibraryCard( );
card1.setOwner(student);
card1.checkOut(3);

card2 = new LibraryCard( );
card2.setOwner(student); //the same student is the owner
                         //of the second card, too
```

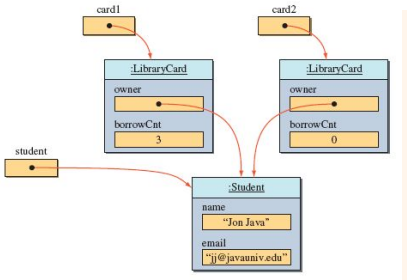
- Since we are actually passing a reference to the same object, it results in the owner of two LibraryCard objects pointing to the same Student object



Sharing an Object

- We pass the same Student object to card1 and card2

```
Student    student;
LibraryCard card1, card2;

student = new Student( );
student.setName('Jon Java');
student.setEmail('jj@javauniv.edu");

card1 = new LibraryCard( );
card1.setOwner(student);
card1.checkOut(3);

card2 = new LibraryCard( );
card2.setOwner(student); //the same student is the owner
                         //of the second card, too
```



- Since we are actually passing a reference to the same object, it results in owner of two LibraryCard objects pointing to the same Student object

Information Hiding and Visibility Modifiers

- The modifiers **public** and **private** designate the accessibility of data members and methods.
- If a class component (data member or method) is declared private, **client classes cannot access it**.
- If a class component is declared public, **client classes can access it**.
- Internal details of a class are declared private and hidden from the clients.

  This is information hiding.

Accessibility Example

```
...
Service obj = new Service();
obj.memberOne = 10;          ✓
obj.memberTwo = 20;          ✗
obj.doOne();                 ✓
obj.doTwo();                 ✗
...
```

Client

```
class Service {
    public  int memberOne;
    private int memberTwo;

    public void doOne() {
    ...
    }
    private void doTwo() {
    ...
    }
}
```

Service

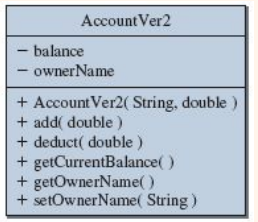
Data Members Should Be private

- Data members are the implementation details of the class, so they should be invisible to the clients. Declare them **private** .
- *Exception*: **Constants** can (should) be declared public if they are meant to be used directly by the outside methods.

Guideline for Visibility Modifiers

- Guidelines in determining the visibility of data members and methods:
  - Declare the class and instance variables private.
  - Declare the class and instance methods private if they are used only by the other methods in the same class.
  - Declare the class constants public if you want to make their values directly readable by the client programs. If the class constants are used for internal purposes only, then declare them private.

Diagram Notation for Visibility



public – plus symbol (+)
private – minus symbol (-)

Class Constants

- In Chapter 3, we introduced the use of constants.
- We illustrate the use of constants in programmer-defined service classes here.
- Remember, the use of constants
  - provides a meaningful description of what the values stand for. number = UNDEFINED; is more meaningful than number = -1;
  - provides easier program maintenance. We only need to change the value in the constant declaration instead of locating all occurrences of the same value in the program code

A Sample Use of Constants

```
import java.util.Random;
class Die {

    private static final int MAX_NUMBER =  6;
    private static final int MIN_NUMBER = 1;
    private static final int NO_NUMBER = 0;

    private int number;
    private Random random;

    public Dice( ) {
       random = new Random();
       number = NO_NUMBER;
    }

    //Rolls the dice
    public void roll( ) {
       number = random.nextInt(MAX_NUMBER - MIN_NUMBER + 1) + MIN_NUMBER;
    }

    //Returns the number on this dice
    public int getNumber( ) {
        return number;
    }
 }
```


Local Variables

- Local variables are declared within a method declaration and used for temporary services, such as storing intermediate computation results.

```
public double convert(int num) {

    double result;      <-- local variable

    result = Math.sqrt(num * num);

    return result;
}
```


Local, Parameter & Data Member

- An identifier appearing inside a method can be a local variable, a parameter, or a data member.
- The rules are
  - If there's a matching local variable declaration or a parameter, then the identifier refers to the local variable or the parameter.
  - Otherwise, if there's a matching data member declaration, then the identifier refers to the data member.
  - Otherwise, it is an error because there's no matching declaration.

Sample Matching

```
class MusicCD {

    private String artist;
    private String title;
    private String id;

    public MusicCD(String name1, String name2) {

        String ident;

        artist = name1;

        title  = name2;

        ident  = artist.substring(0,2) + "-" +
                 title.substring(0,9);

        id = ident;
    }
    ...
}
```