Chapter 4

Defining Your Own Classes
Part 1

Animated Version

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

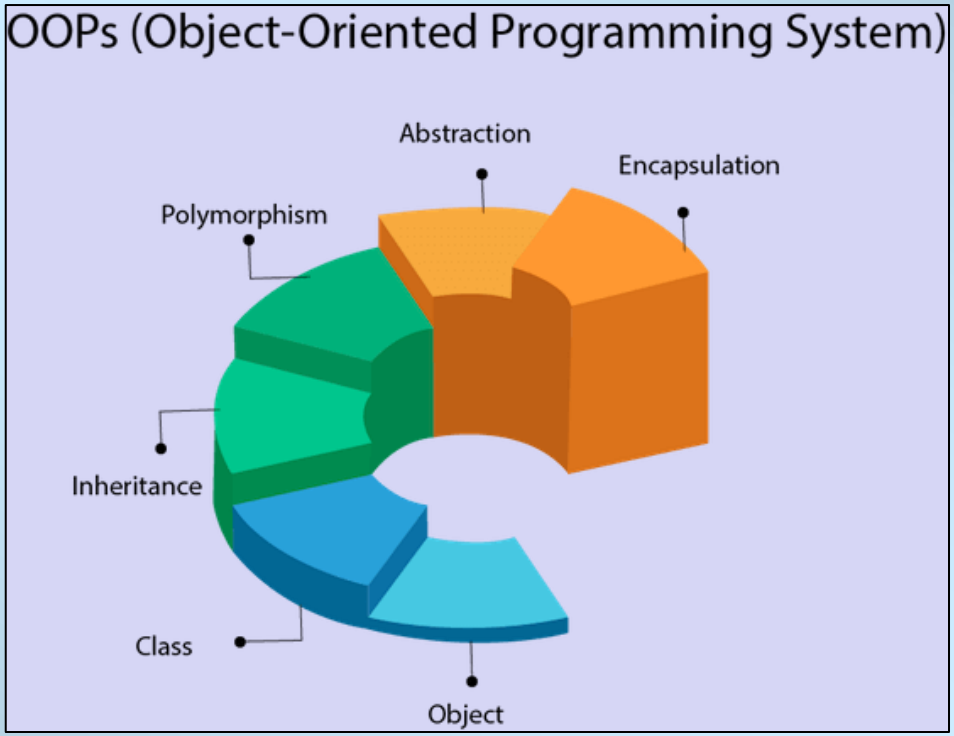
- Using just the String, GregorianCalendar, JFrame and other standard classes will not meet all of our needs. We need to be able to define our own classes customized for our applications.
- Learning how to define our own classes is the first step toward mastering the skills necessary in building large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

Why Design a Class First?

When we start to design a program in Java, we design a class as the first step to create a blueprint or template for objects. We then create an object as an instance of that class to make use of the blueprint.

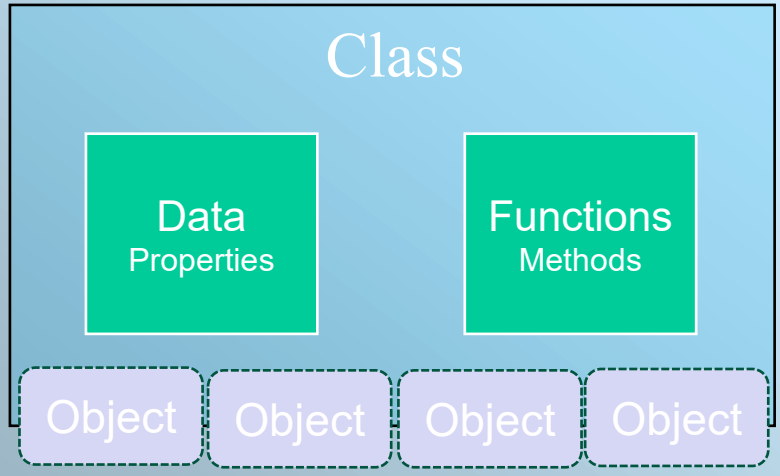
Why Design a Class First?

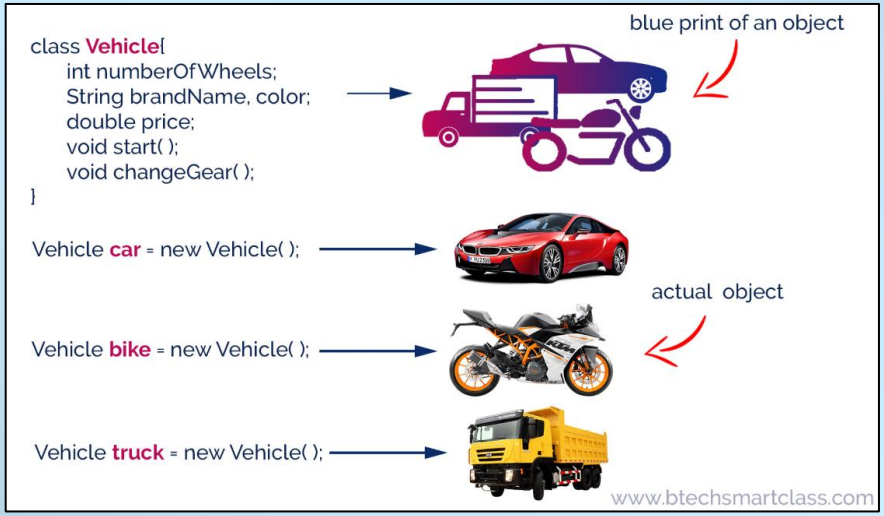
A class in Java is a **blueprint** for creating objects. Think of a class as the architectural plan for a house. The plan defines what the house will have: how many rooms, where the windows will be, and so on. Similarly, a class defines the attributes (data or properties) and methods (actions or behaviors) that an object will possess.



Why Create Objects?

- **State and Behavior:** An object has a **unique state** (the specific values of its attributes) and can perform actions (its methods).
  For example, a Car class might have attributes like color and speed. An object created from this class, let's say **myCar**, might have the specific state of color = **'red'** and speed = **60**. You can then call methods on **myCar** like **accelerate**() or **brake**().
- **Execution: Objects** are the **entities** that actually execute the program's logic. They store the **data** and perform the **actions** defined in their class. Without objects, the class is just an inactive definition.
- **Real-World Modeling: Creating objects** allows you to **model real-world entities** in your program. If you're building a system for a school, you might have **classes** like **Student** and **Teacher**, and then create **individual** student and teacher objects with their own **unique data**.





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

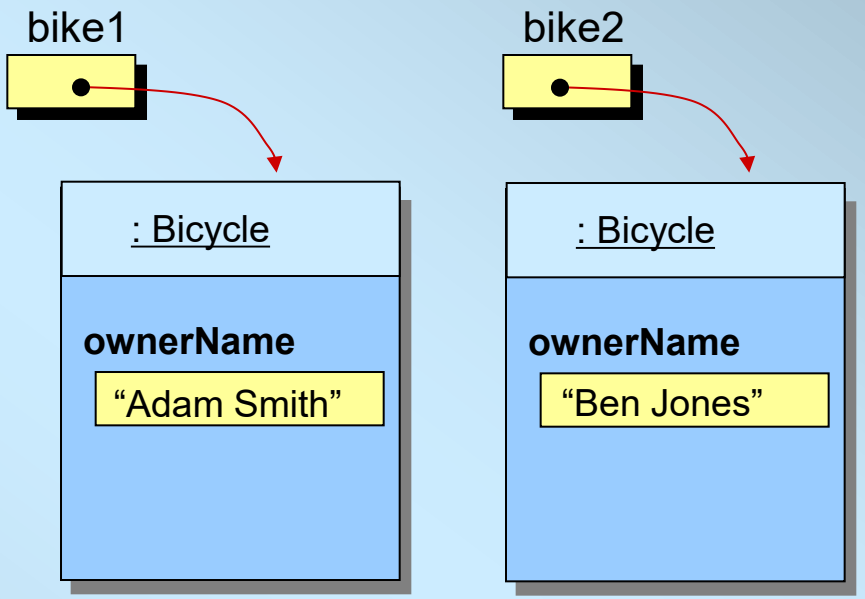
```
class Bicycle {

    // Data Member
    private String ownerName;

    //Constructor: Initialzes the data member
    public 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

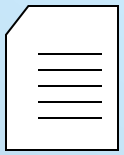
- Once the Bicycle class is defined, we can create multiple instances.

```
Bicycle bike1, bike2;

bike1 = new Bicycle( );
bike1.setOwnerName("Adam Smith");

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

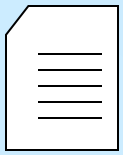
Sample Code



The Program Structure and Source Files



BicycleRegistration.java

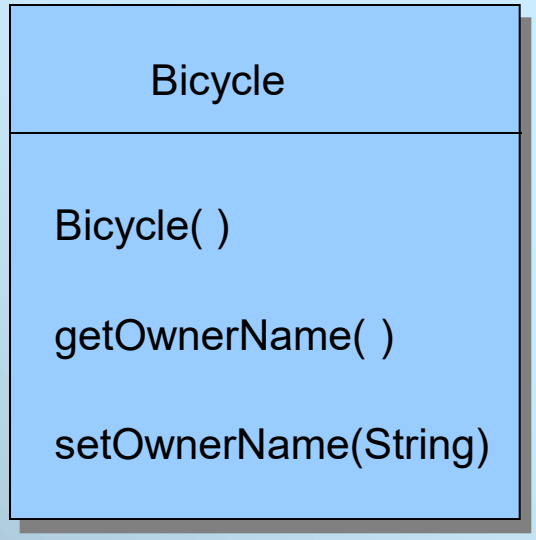


Bicycle.java

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

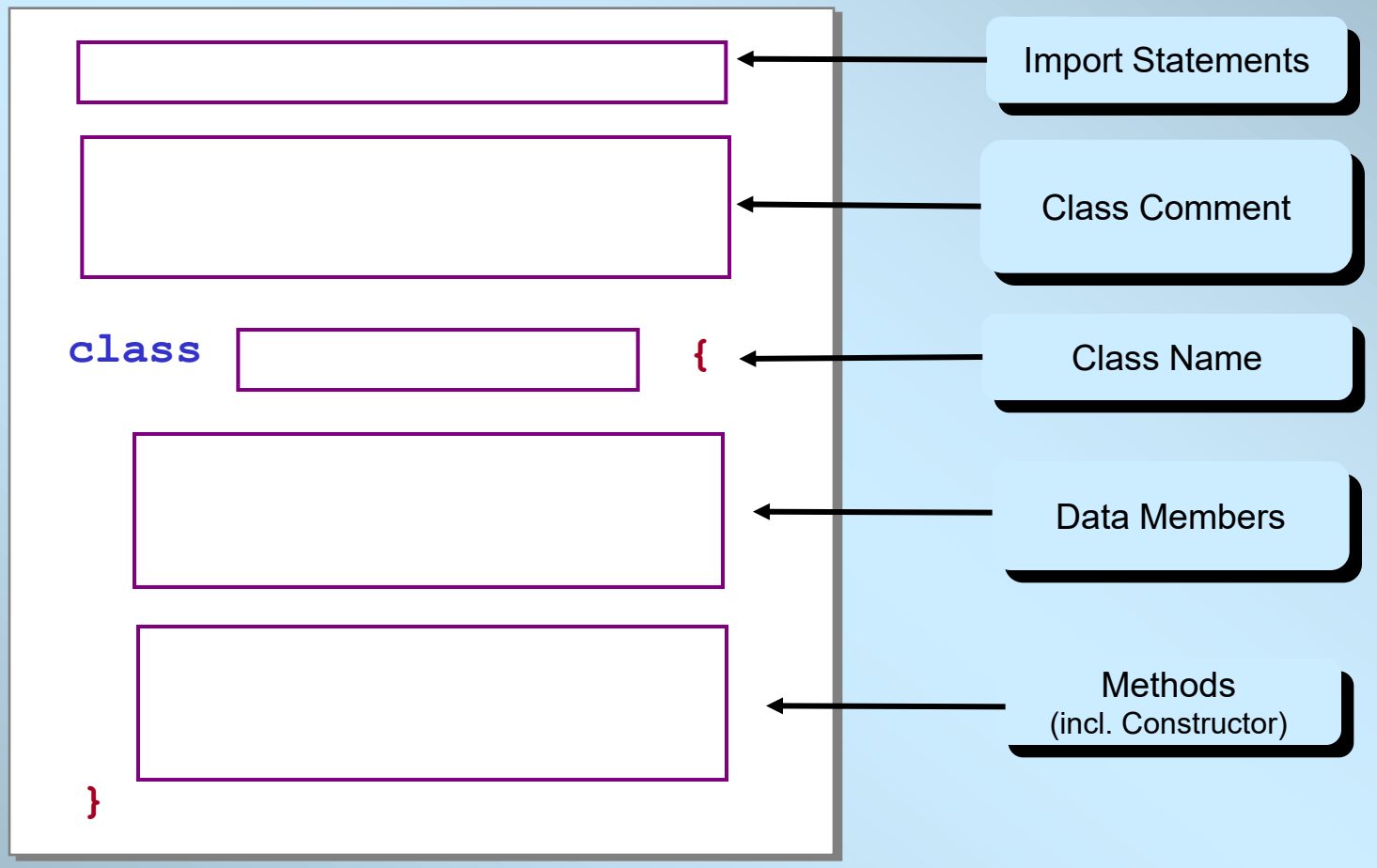
To run the program: 1. javac Bicycle.java (compile)
2. javac BicycleRegistration.java (compile)
3. java BicycleRegistration (run)

Class Diagram for Bicycle



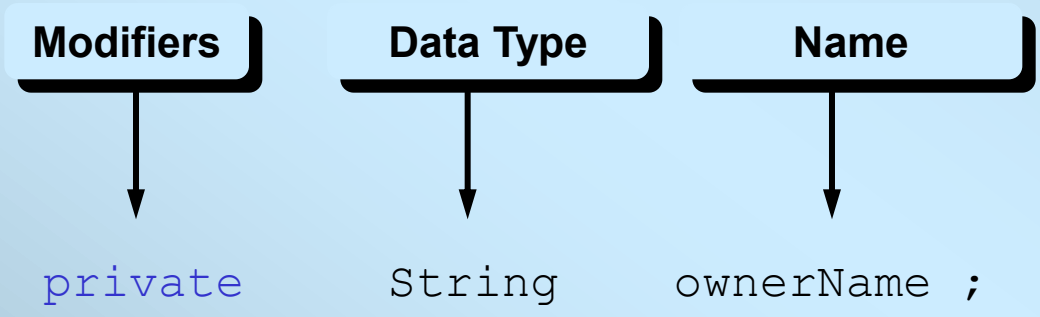
Method Listing
We list the name and the data type of an argument passed to the method.

Template for Class Definition



Data Member Declaration

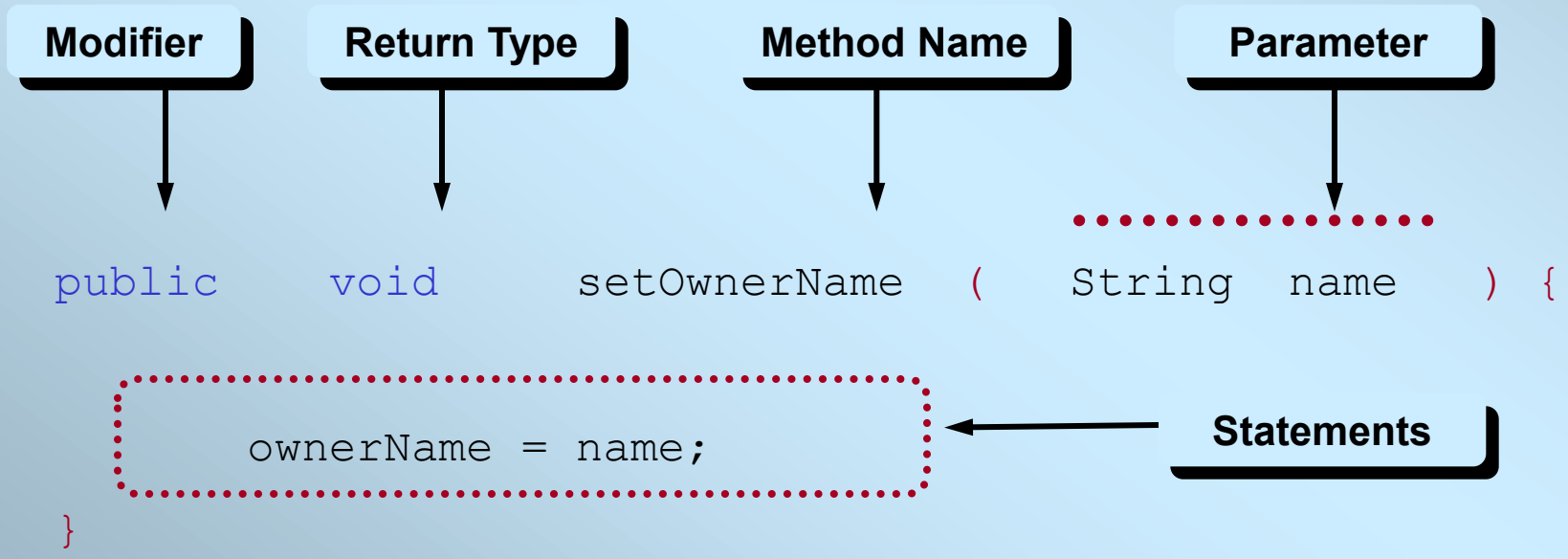
```
<modifiers>  <data type> <name> ;
```



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

Method Declaration

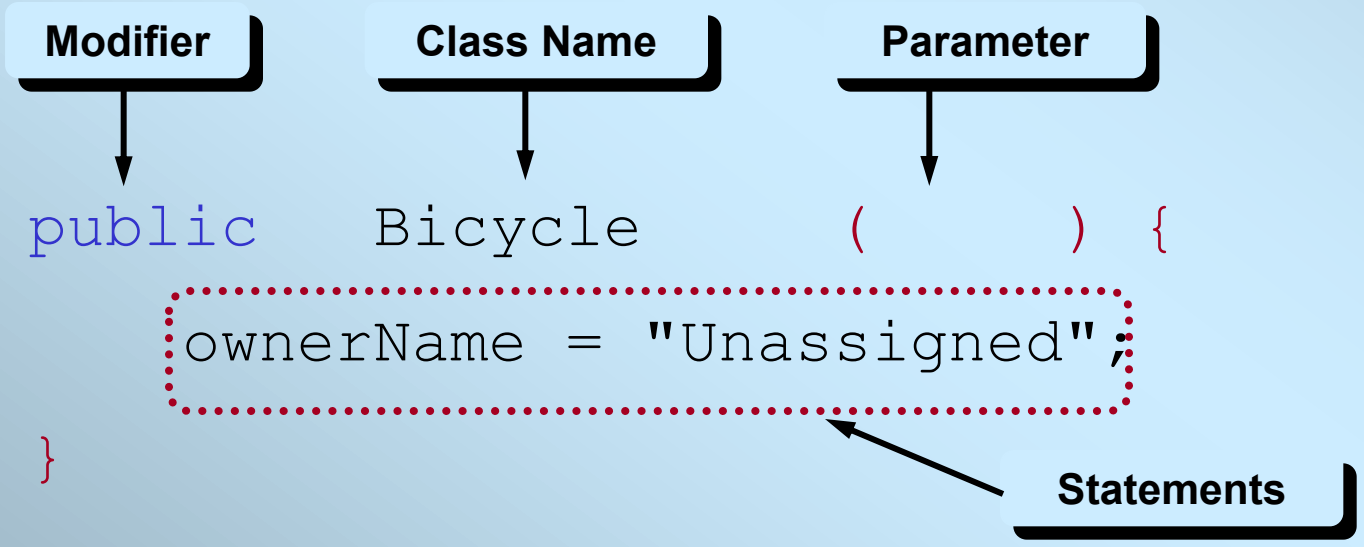
```
<modifier>  <return type>  <method name>  ( <parameters>  ){
        <statements>
}
```



Constructor

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

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



Second Example: Using Bicycle and Account

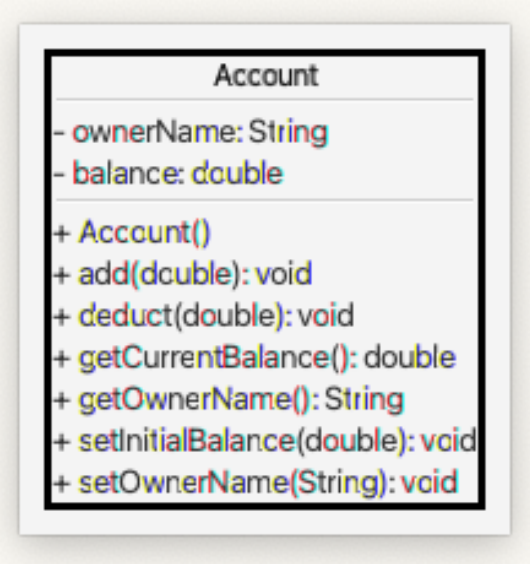
```
class SecondMain {

    //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;
    }

    public String getOwnerName( ) {

        return ownerName;
    }
```

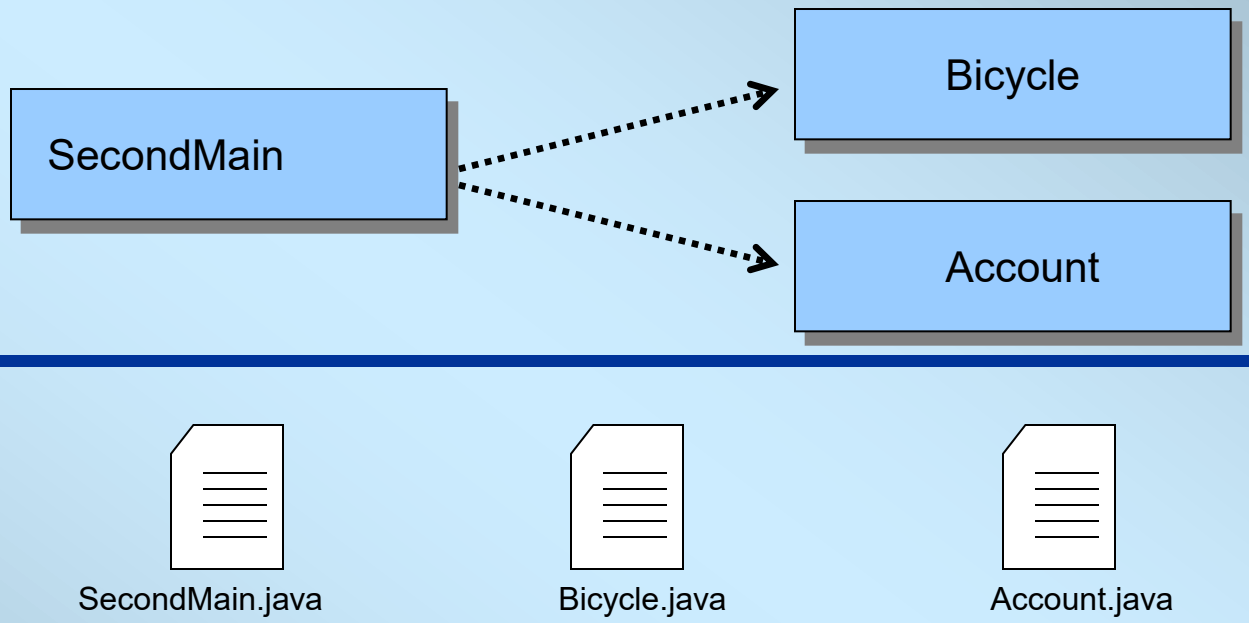
Page 1

```
    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);
      . . .
    }

    . . .
 }
```

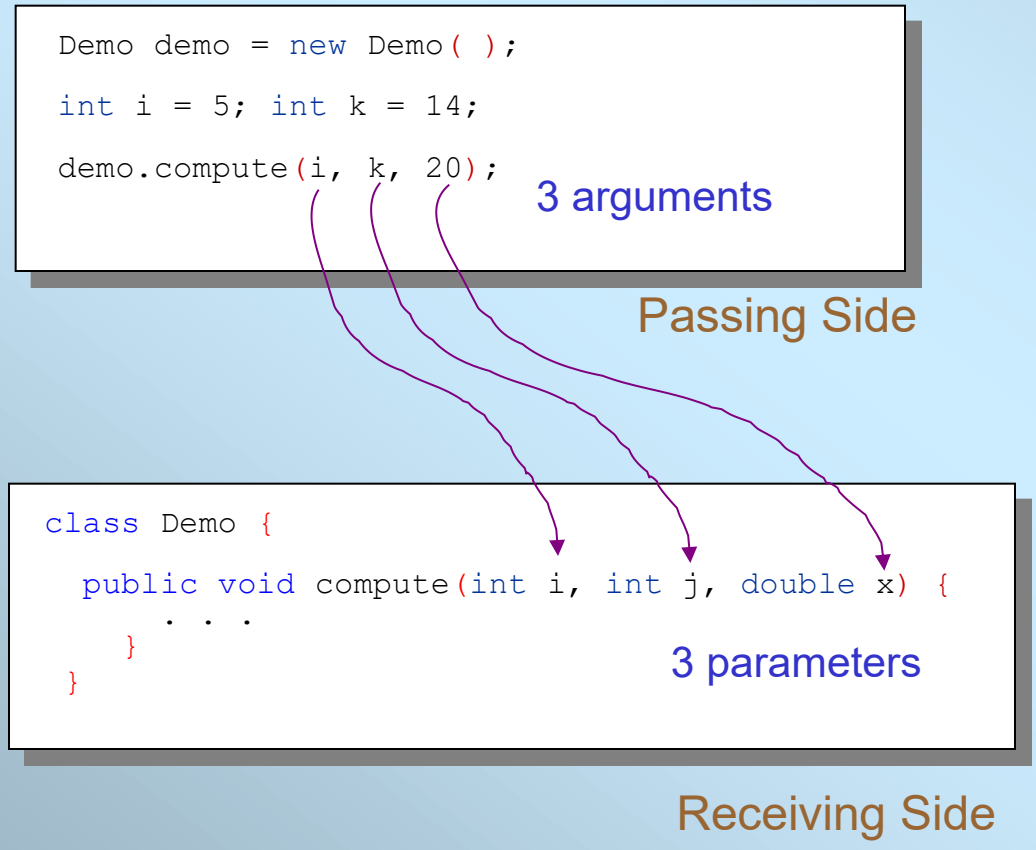
argument

```
class Account {

    . . .

    public void add(double amt) {

        balance = balance + amt;
    }

    . . .
}
```

parameter

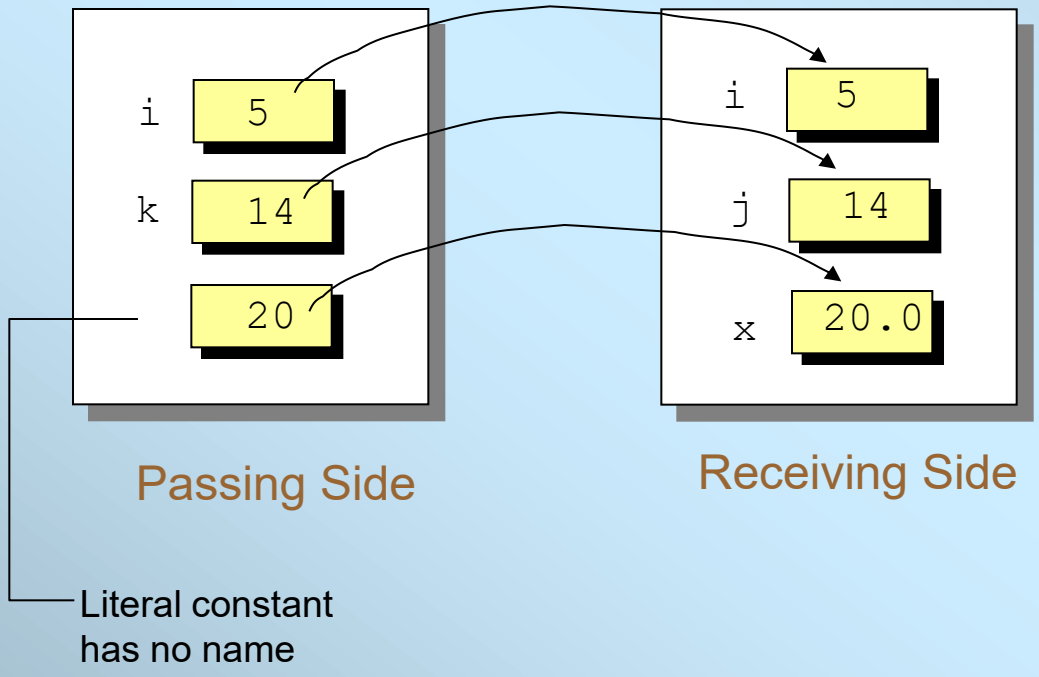
- An argument is a value we pass to a method
- A parameter is a placeholder in the called method to hold the value of the passed argument.

Matching Arguments and Parameters



- 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);
```

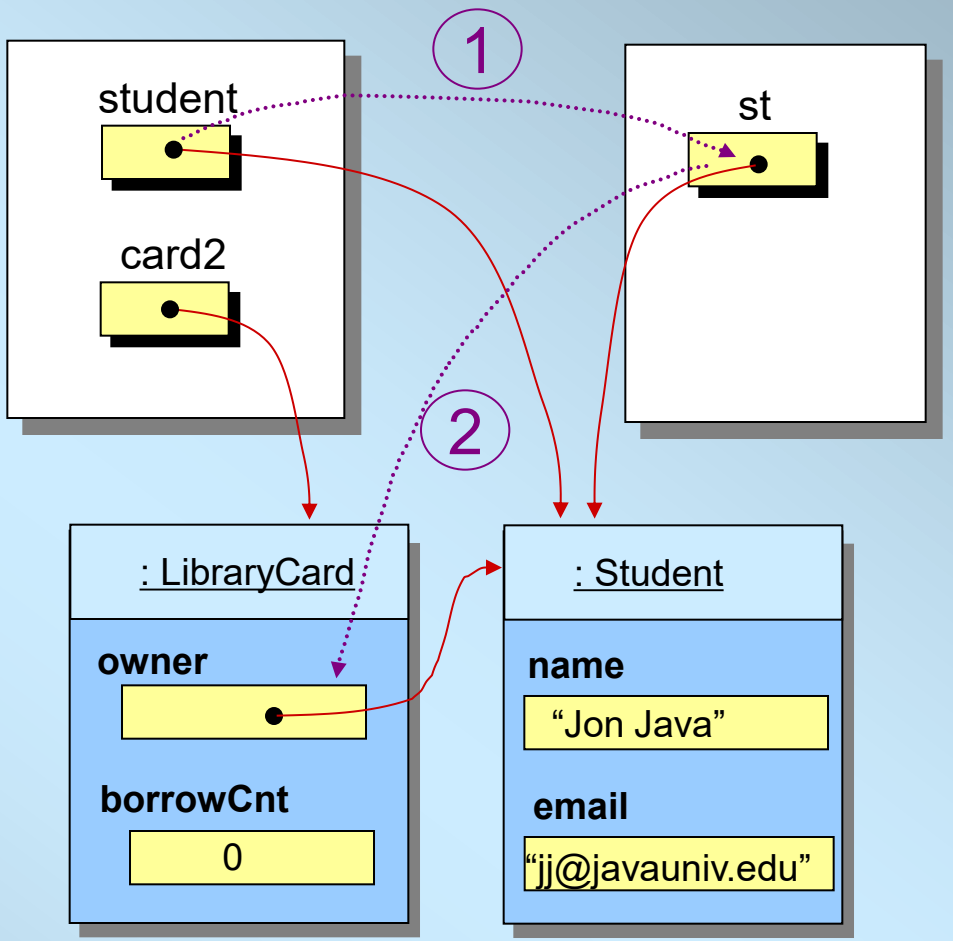
Passing Side

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

      owner = st;
  }
}
```

Receiving Side

1 Argument is passed

2 Value is assigned to the data member

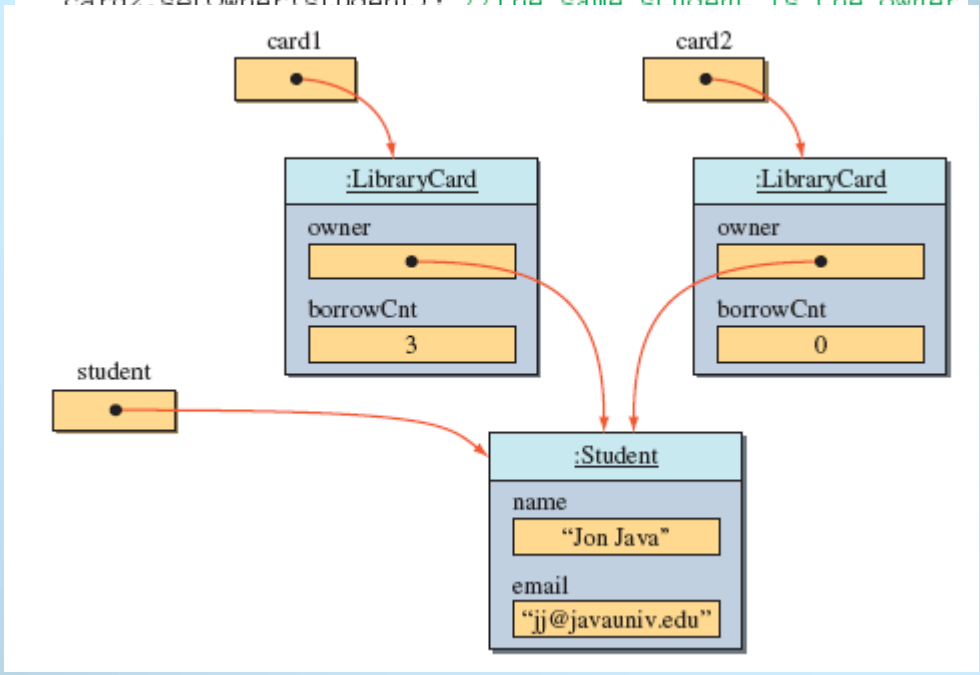


State of Memory

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
```

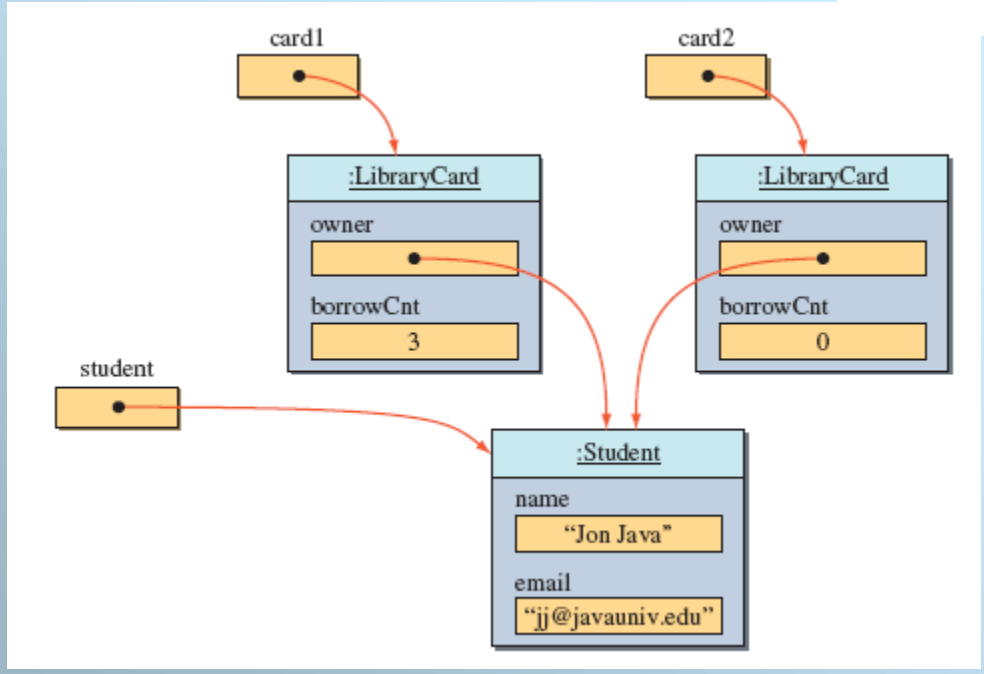


- Since we are actually passing a reference to the same object, it results in the owner of two LibraryCard objects pointing to the same Student object

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();
…
```

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

    public void doOne() {

    …

    }
    private void doTwo() {

    …

    }
}
```

Client

Service

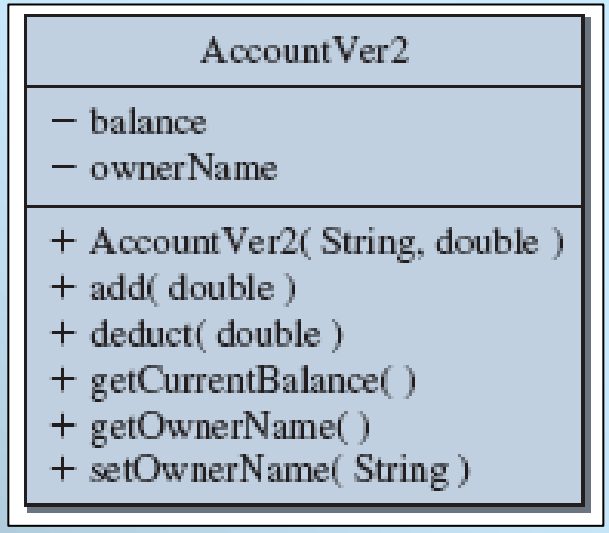
Data Members Should Be private

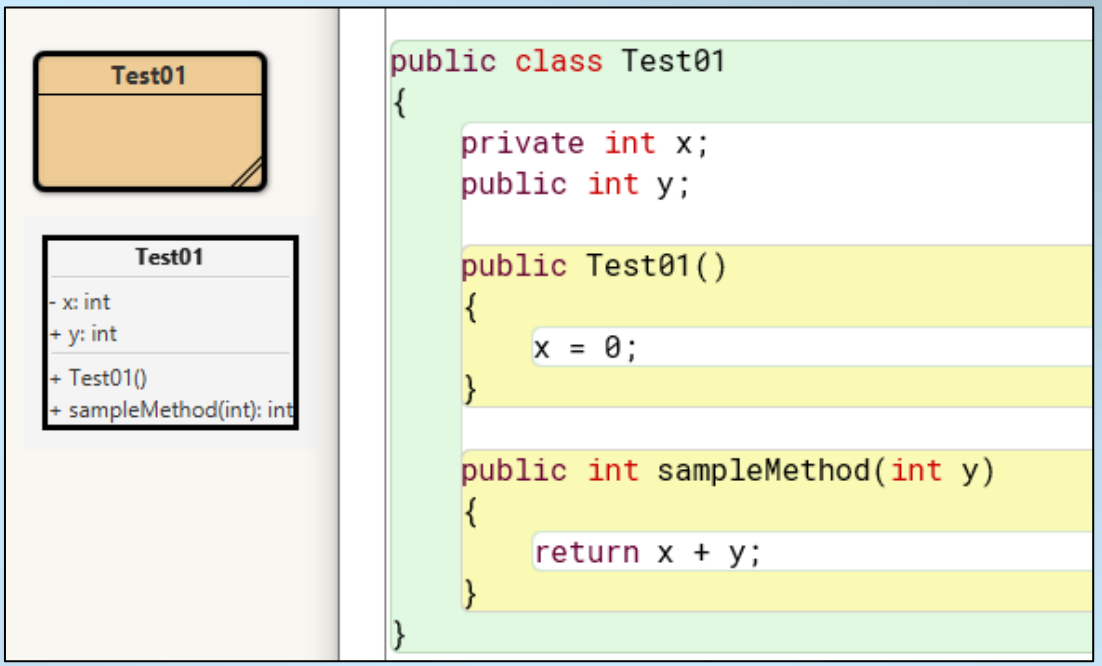
- Data members are the implementation details of the class, so they should be invisible to the clients. Declare them private .
- Exception: Constants can (should) be declared public if they are meant to be used directly by the outside methods.

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 (-)

BlueJ 5+ Extensions

Reference URL : https://www.bluej.org/extensions/extensions2.html

Installing extensions

Extensions are installed by placing the extension jar file into an extension directory.

Class Card - A Better UML Extension

Description:

An improvement over the original UML Extension, this one allows you to show more than one class display, and to move class displays around the screen. Available in English and German.

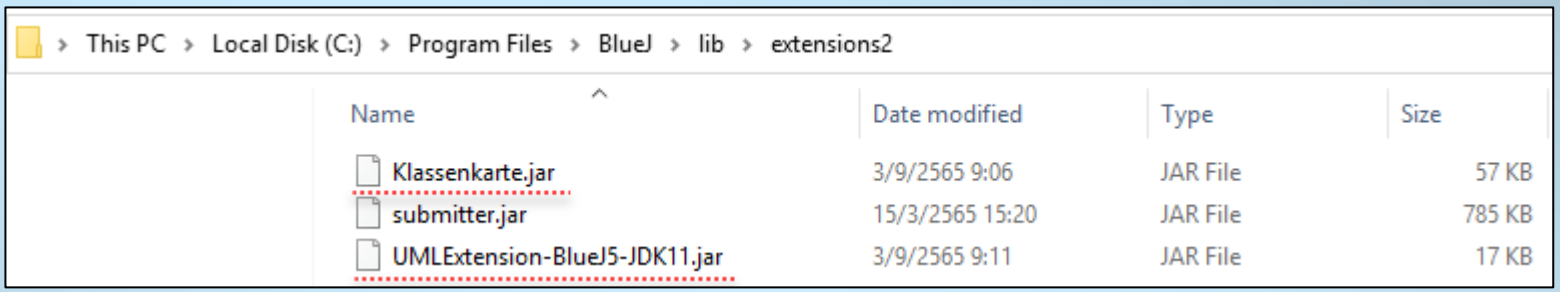
UML Extension

Description:

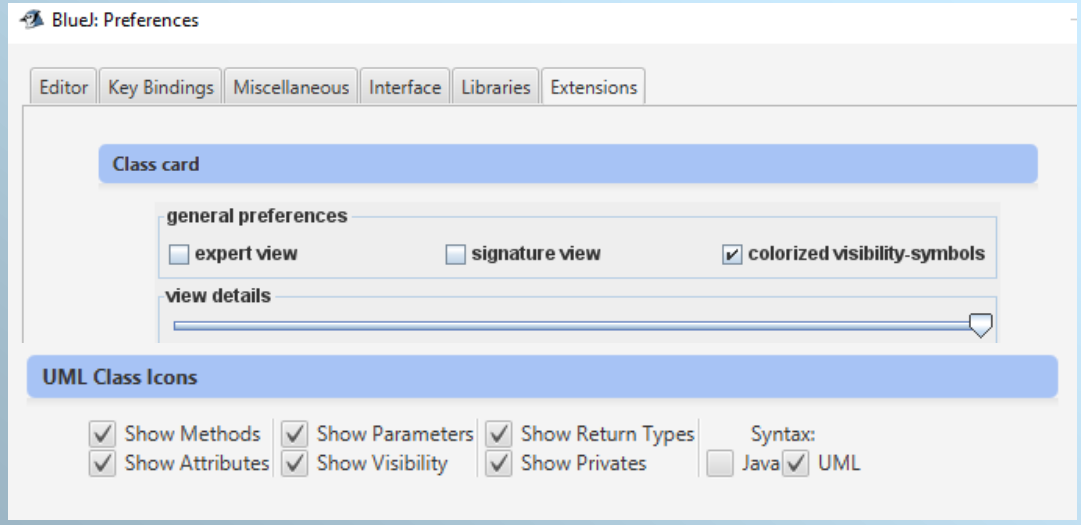
Update from the original extension: displays a simple popup for a BlueJ class in the form of a UML Class icon. Details of what will be displayed, and the precise syntax used can be configured via the preferences panel, or the BlueJ properties files. New functionality: drag icons and double click event to close window.

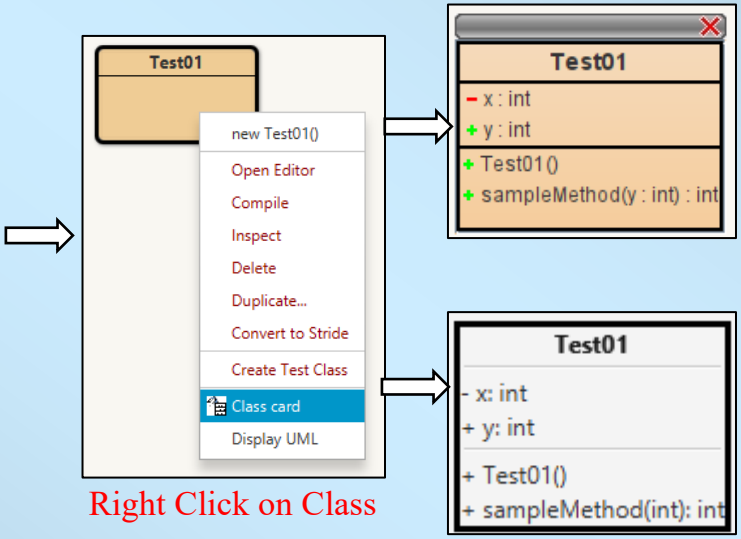
BlueJ 5+ Extensions

Installing Location: C:\Program Files\BlueJ\lib\extensions2



Set Extension: BlueJ > Tools > Preferences… > Extension





Right Click on Class

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

```
class Dice {

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

    private int number;

    public Dice( ) {
       number = NO_NUMBER;
    }

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

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

Local Variables

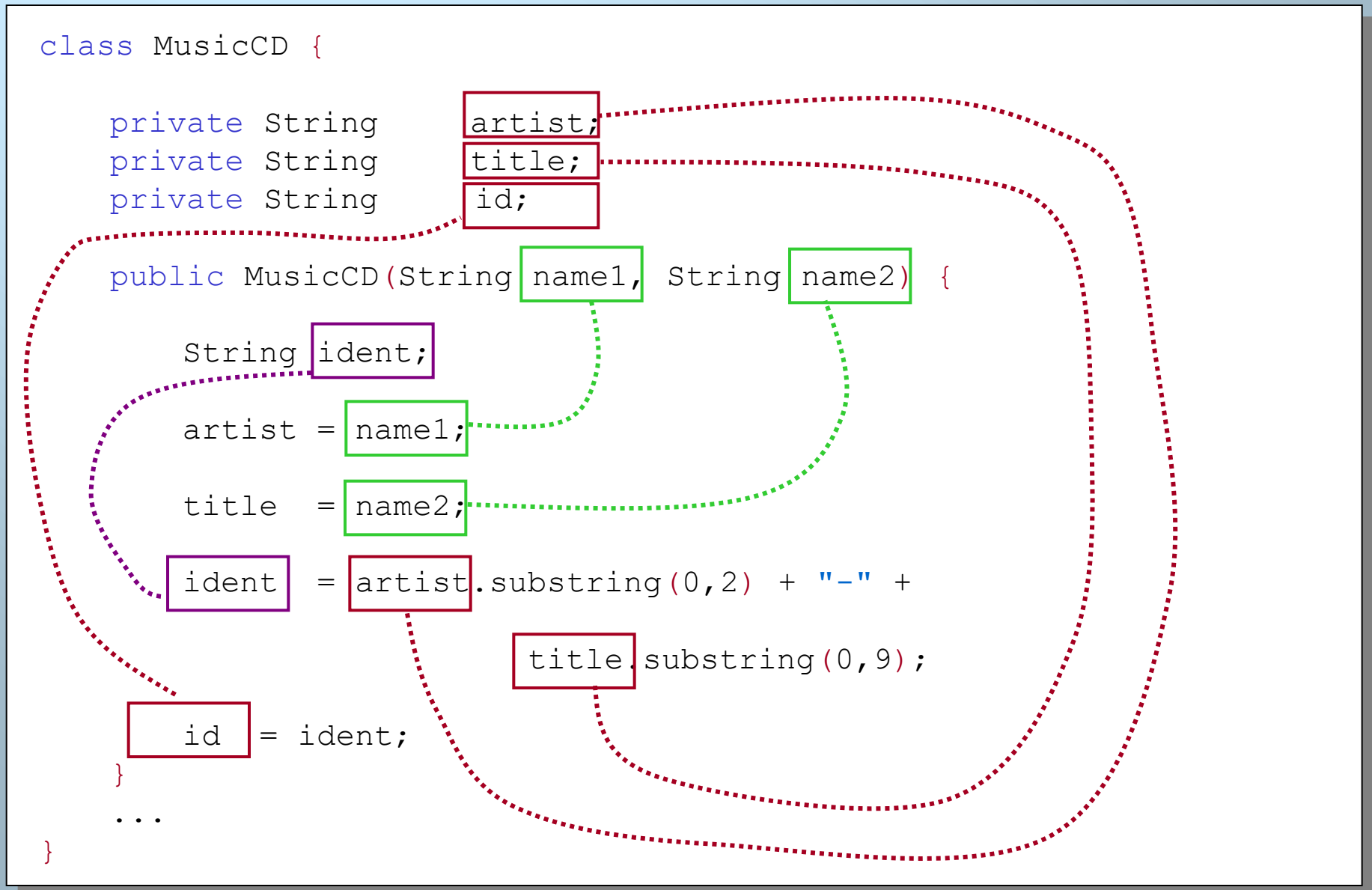
- Local variables are declared within a method declaration and used for temporary services, such as storing intermediate computation results.

```
public double convert(int num) {

    double result;

    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



String Data Type Allocate Memory

```
String str1 = "Test123456";
String str2 = "Test123456";

String str3 = str1.substring(0,2) + "-
" + str2.substring(0,9) ;
```

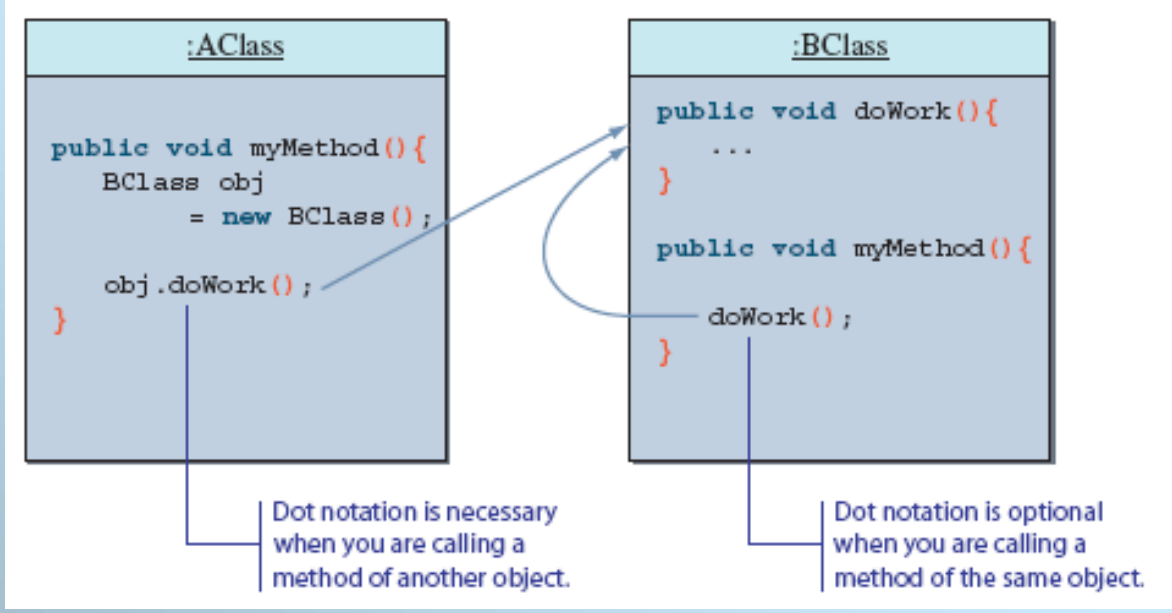
Compute Step

1. String Pool **No.1.** => str1.substring(0,2)
2. String Pool **No.2.** => "-"
3. String Pool **No.3.** => str2.substring(0,9)
4. String Pool **No.4.** => No.1. + No.2.
5. String Pool **No.5.** => No.4. + No.3.

FINAL REF => **str3** point to **No.5**

Calling Methods of the Same Class

- So far, we have been calling a method of another class (object).
- It is possible to call method of a class from another method of the same class.
  - in this case, we simply refer to a method without dot notation



Changing Any Class to a Main Class

- Any class can be set to be a main class.
- All you have to do is to include the main method.

```
class Bicycle {

    //definition of the class as shown before comes here

    //The main method that shows a sample
    //use of the Bicycle class
    public static void main(String[] args) {

        Bicycle myBike;

        myBike = new Bicycle( );
        myBike.setOwnerName("Jon Java");

        System.out.println(myBike.getOwnerName() + "owns a bicycle");
    }
}
```

Problem Statement

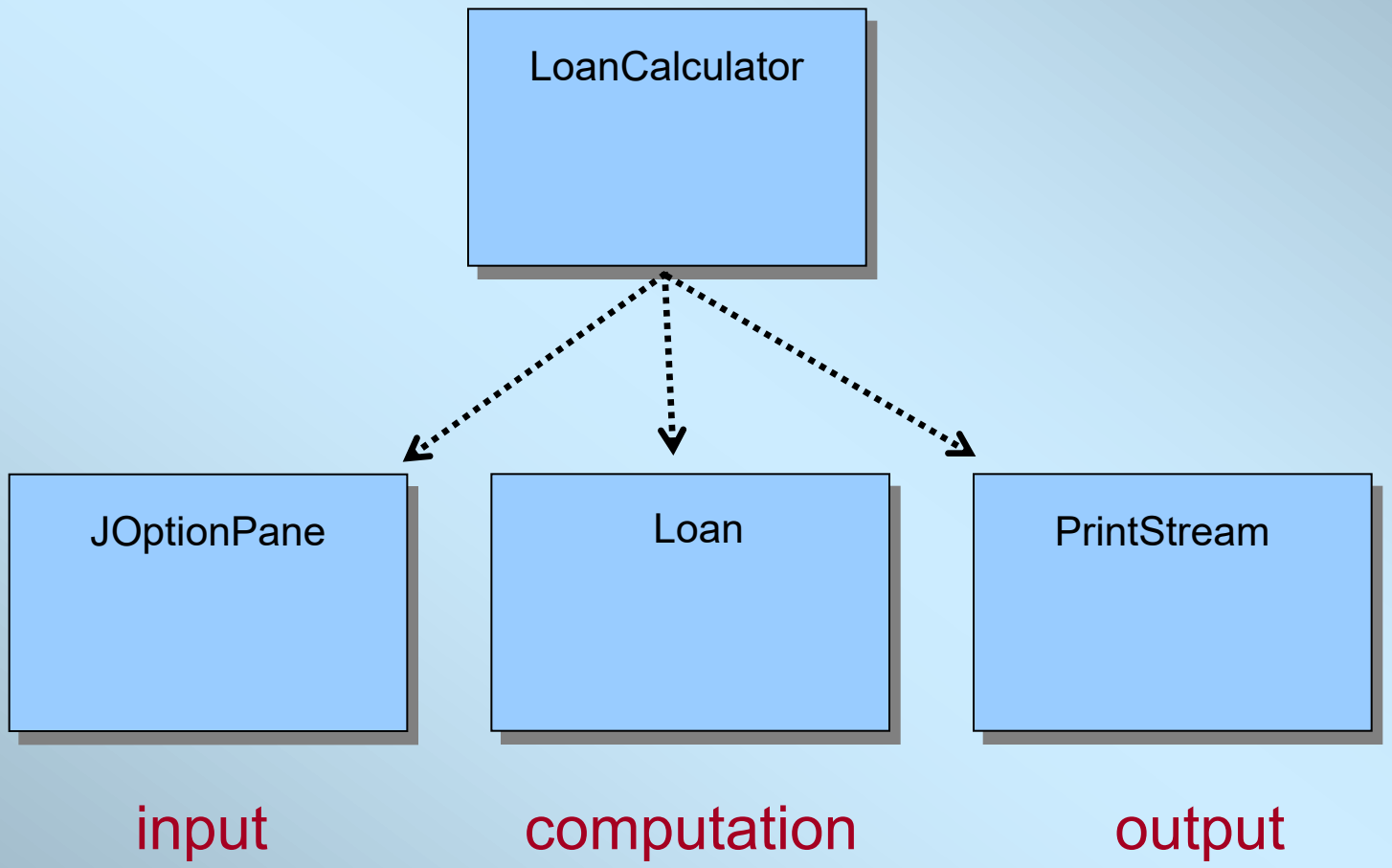
- Problem statement:

  Write a loan calculator program that computes both monthly and total payments for a given loan amount, annual interest rate, and loan period.

Overall Plan

- Tasks:
  - Get three input values: **loanAmount**, **interestRate**, and **loanPeriod**.
  - Compute the monthly and total payments.
  - Output the results.

Required Classes



Development Steps

- We will develop this program in five steps:

1. Start with the main class LoanCalculator. Define a temporary placeholder Loan class.
2. Implement the input routine to accept three input values.
3. Implement the output routine to display the results.
4. Implement the computation routine to compute the monthly and total payments.
5. Finalize the program.

Step 1 Design

- The methods of the LoanCalculator class

Method	Visibility	Purpose
start	public	Starts the loan calcution. Calls other methods
computePayment	private	Give three parameters, compute the monthly and total payments
describeProgram	private	Displays a short description of a program
displayOutput	private	Displays the output
getInput	private	Gets three input values

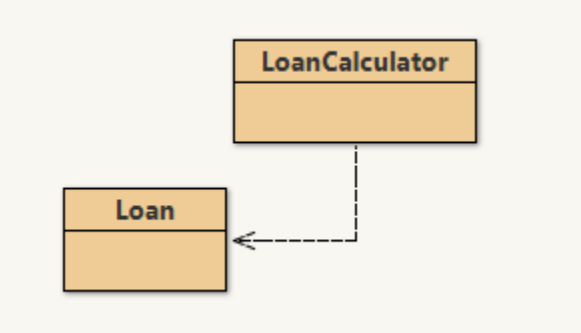
Step 1 Code

Program source file is too big to list here. From now on, we ask you to view the source files using your Java IDE.

Directory: Chapter4/Step1

Source Files:

LoanCalculator.java
Loan.java



Create Class LoanCalulator and Loan

```
class Loan {

//---------------------------------
//    Data Members
//---------------------------------

// Default contructor.
    public Loan( ) {

    }
}
```

```
class LoanCalculator {

//    Data Members
// This object does the actual loan computation
    private Loan loan;

//---------------------------------
//  Main Method
//---------------------------------
    public static void main(String[] arg) {
        LoanCalculator calculator = new LoanCalculator();
        calculator.start();
    }
```

```
//-----------------------------------
//    Constructors
//-----------------------------------

    public LoanCalculator()
    {
        loan = new Loan();
    }
```

```
/**
 * Top level method that calls other private methods
 * to compute the monthly and total loan payments
 */
public void start()
{

    describeProgram();   //tell what the program does
    getInput();          //get three input values
    computePayment();    //compute the monthly payment and total
    displayOutput();     //diaply the results
}
```

Add Method to LoanCalculator

```
/**
 * Computes the monthly and total loan payments.
 */
private void computePayment()
{
    System.out.println("inside computePayment");   //TEMP
}

/**
 * Provides a brief explaination of the program to the user.
 */
private void describeProgram()
{
    System.out.println("inside describeProgram");   //TEMP
}

/**
 * Display the input values and monthly and total payments.
 */
private void displayOutput()
{
    System.out.println("inside displayOutput");   //TEMP
}

/**
 * Gets three input values--loan amount, interest rate, and
 * loan period--using an InputBox object
 */
private void getInput()
{
    System.out.println("inside getInput");   //TEMP
}
```

Step 1 Test

- In the testing phase, we run the program multiple times and verify that we get the following output

```
inside describeProgram
inside getInput
inside computePayment
inside displayOutput
```

Step 2 Design

- Design the input routines
  - LoanCalculator will handle the user interaction of prompting and getting three input values
  - LoanCalculator calls the setAmount, setRate and setPeriod of a Loan object.

```
 * loan period--using an InputBox object
 */
private void getInput() {

    double  loanAmount, annualInterestRate;

    int     loanPeriod;

    String  inputStr;

    inputStr           = JOptionPane.showInputDialog(null,
                                "Loan Amount (Dollars+Cents):");
    loanAmount         = Double.parseDouble(inputStr);

    inputStr           = JOptionPane.showInputDialog(null,
                                "Annual Interest Rate (e.g., 9.5):");
    annualInterestRate = Double.parseDouble(inputStr);

    inputStr           = JOptionPane.showInputDialog(null,
                                "Loan Period - # of years:");
    loanPeriod         = Integer.parseInt(inputStr);

    //assign input values to the loan object
    loan.setAmount(loanAmount        );
    loan.setRate  (annualInterestRate);
    loan.setPeriod(loanPeriod        );

    //TEMP
    System.out.println("Loan Amount: $" + loan.getAmount());
    System.out.println("Annual Interest Rate:"
                       + loan.getRate() + "%");
    System.out.println("Loan Period (years):" + loan.getPeriod());
    //TEMP
}
```

Create Method in Class Loan

Data Member =>

```
public static final MONTHS_IN_YEAR;

private double loanAmount;
private double monthlyInterestRate;
private int numberOfPayments
```

Method =>

```
/**
 * Sets the loan amount of this loan.
 *
 * @param amount the loan amount
 */
public void setAmount(double amount) {
    loanAmount = amount;
}
```

```
/**
 * Sets the interest rate of this loan.
 *
 * @param annualRate the annual interest rate
 */
public void setRate(double annualRate) {
    monthlyInterestRate = annualRate / MONTHS_IN_YEAR  / 100;
}
```

```
/**
 * Sets the loan period of this loan.
 *
 * @param period the loan period expressed in
 *               number of years.
 */
public void setPeriod(int periodInYears) {
    numberOfPayments = periodInYears * MONTHS_IN_YEAR;
}
```

Step 2 Code

Directory: Chapter4/Step2

Source Files:

LoanCalculator.java
Loan.java

Step 2 Test

- We run the program numerous times with different input values
- Check the correctness of input values by echo printing

```
System.out.println("Loan Amount: $"
                              + loan.getAmount());

System.out.println("Annual Interest Rate:"
                              + loan.getRate() + "%");

System.out.println("Loan Period (years):"
                              + loan.getPeriod());
```

Step 3 Design

- We will implement the displayOutput method.
- We will reuse the same design we adopted in Chapter 3 sample development.

```java
/**
 * Provides a brief explaination of the program to the user.
 */
private void describeProgram() {
    System.out.println("This program computes the monthly and total");
    System.out.println("payments for a given loan amount, annual ");
    System.out.println("interest rate, and loan period (# of years).");
    System.out.println("\n");
}
```

```java
/**
 * Display the input values and monthly and total payments.
 */
private void displayOutput() {

    DecimalFormat df = new DecimalFormat("0.00");

    System.out.println("Loan Amount: $" + loan.getAmount());
    System.out.println("Annual Interest Rate:"
                       + loan.getRate() + "%");
    System.out.println("Loan Period (years): " + loan.getPeriod());

    System.out.println("Monthly payment is $ " + df.format(loan.getMonthlyPayment()));
    System.out.println("  TOTAL payment is $ " + df.format(loan.getTotalPayment()));

}
```

Step 3 Code

Directory: Chapter4/Step3

Source Files:

LoanCalculator.java
Loan.java

Step 3 Test

- We run the program numerous times with different input values and check the output display format.
- Adjust the formatting as appropriate

Step 4 Design

- Two methods getMonthlyPayment and getTotalPayment are defined for the Loan class
- We will implement them so that they work independent of each other.
- It is considered a poor design if the clients must call getMonthlyPayment before calling getTotalPayment.

```java
/**
 * Returns the loan amount.
 *
 * @return the loan amount
 */
public double getAmount( ) {
    return loanAmount;
}

/**
 * Returns the loan period in the number of years.
 *
 * @return the loan period in the number of years
 */
public int getPeriod( ) {
    return numberOfPayments / MONTHS_IN_YEAR;
}

/**
 * Returns the annual interest rate.
 *
 * @return the annual interest rate
 */
public double getRate( ) {
    return monthlyInterestRate * MONTHS_IN_YEAR;
}
```

```java
/**
 * Returns the monthly payment for
 * the currently set amount, interest rate,
 * and loan period
 *
 * @return the monthly payment
 */
public double getMonthlyPayment( ) {
    double monthlyPayment;

    monthlyPayment = (loanAmount * monthlyInterestRate)
                      /
                     (1 - Math.pow(1/(1 + monthlyInterestRate),
                                   numberOfPayments ) );
    return monthlyPayment;
}
```

```java
/**
 * Returns the total payment for
 * the currently set amount, interest rate,
 * and loan period
 *
 * @return the total payment
 */
public double getTotalPayment( ) {
    double totalPayment;

    totalPayment = getMonthlyPayment( ) * numberOfPayments;

    return totalPayment;
}
```

Step 4 Test

- We run the program numerous times with different types of input values and check the results.

Input			Output (shown up to three decimal places only)	
Loan Amount	Annual Interest Rate	Loan Period (in Years)	Monthly Payment	Total Payment
10000	10	10	132.151	15858.088
15000	7	15	134.824	24268.363
10000	12	10	143.471	17216.514
0	10	5	0.000	0.000
30	8.5	50	0.216	129.373

Step 5: Finalize

- We will implement the describeProgram method
- We will format the monthly and total payments to two decimal places using DecimalFormat.