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.

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

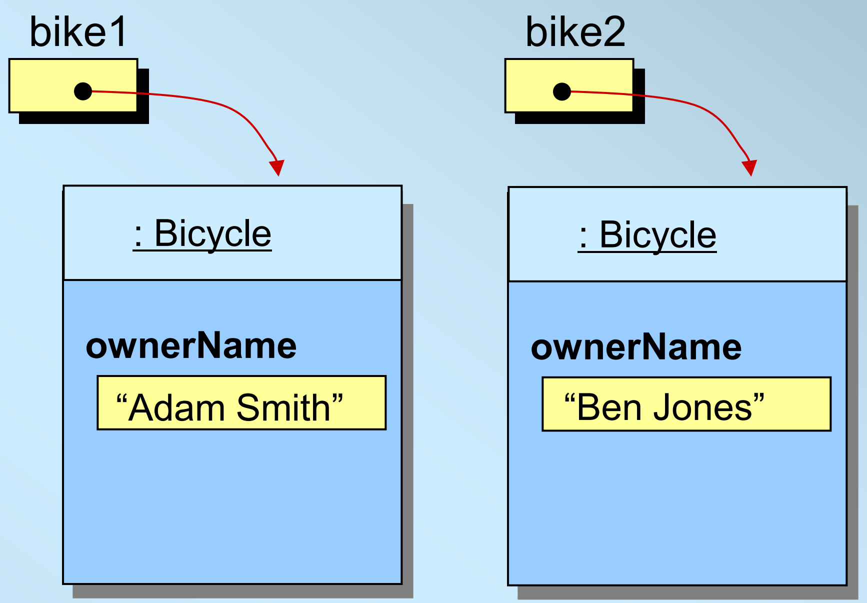
```
class Bicycle {

    // 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

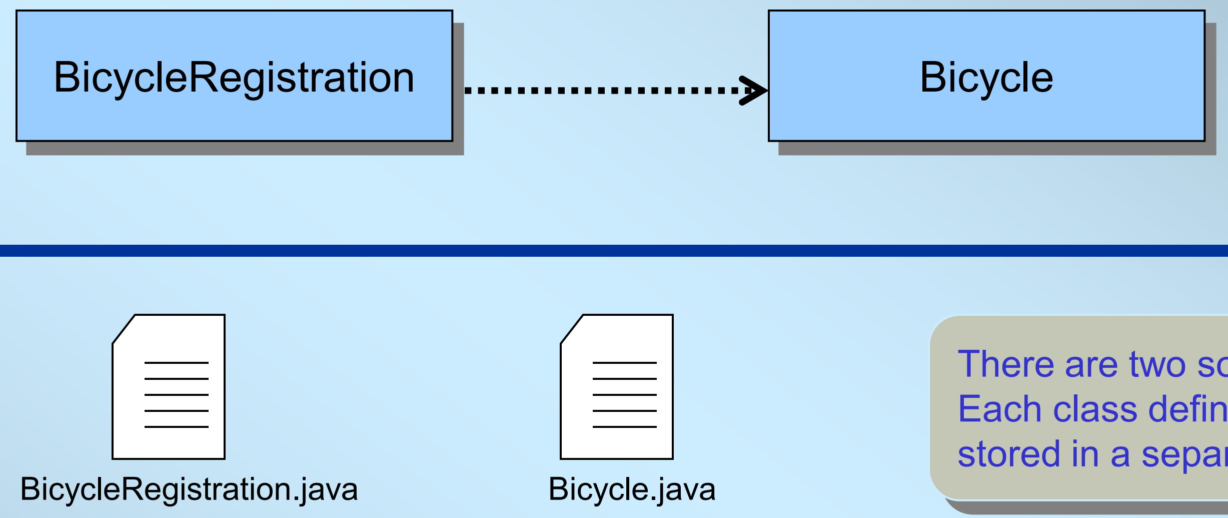
- Once the Bicycle class is defined, we can create multiple instances.

```
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



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

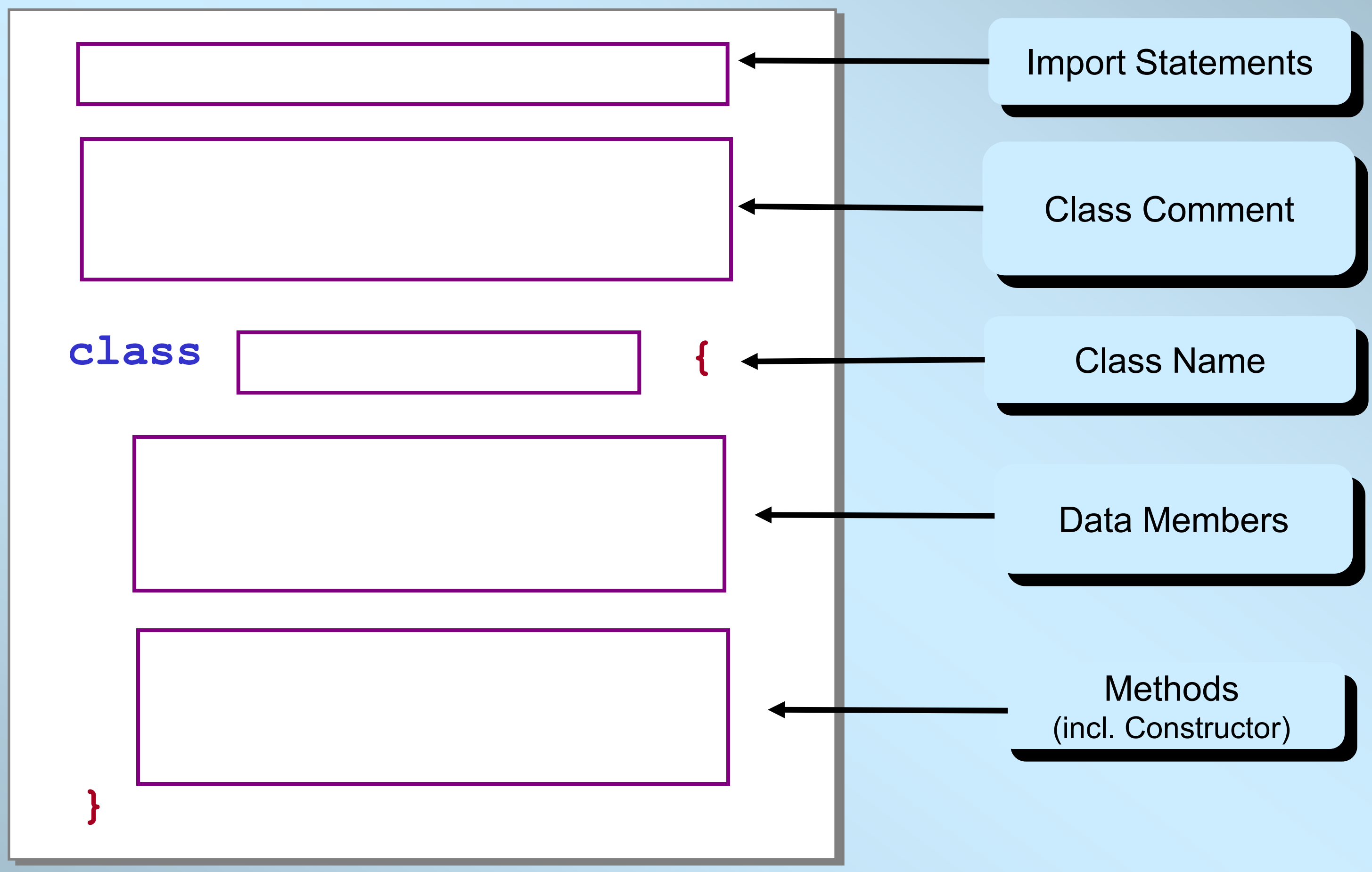
To run the program: 1. javac Bicycle.java (compile)
2. javac BicycleRegistration.java (compile)
3. java BicycleRegistration (run)

Class Diagram for Bicycle

Bicycle
Bicycle( ) getOwnerName( ) setOwnerName(String)

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 **Data Type** **Name**

```
private     String    ownerName ;
```

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

Method Declaration

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

Modifier **Return Type** **Method Name** **Parameter**

```
public    void    setOwnerName  (  String  name  ) {

      ownerName = name;
}
```

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>
}
```

Modifier

Class Name

Parameter

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

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

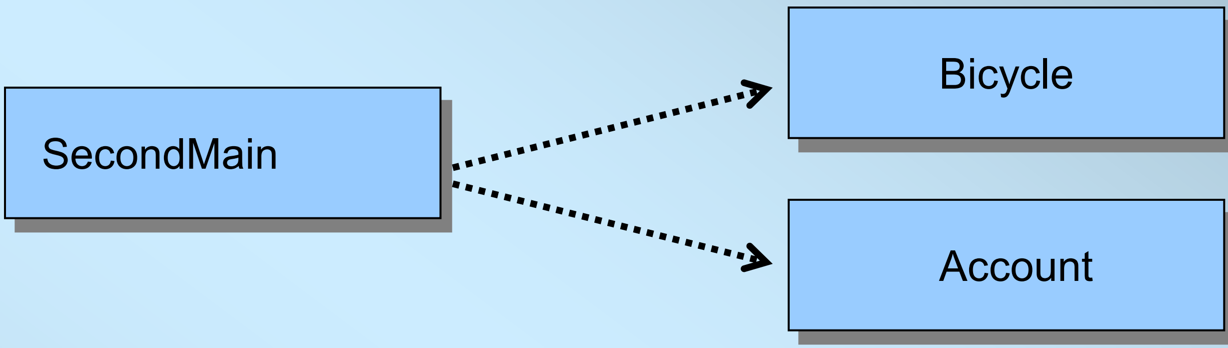
```java
class Account {

    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;
    }
```

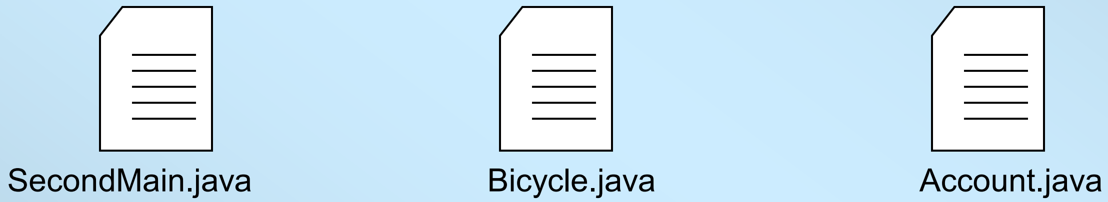
Page 1

```java
    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

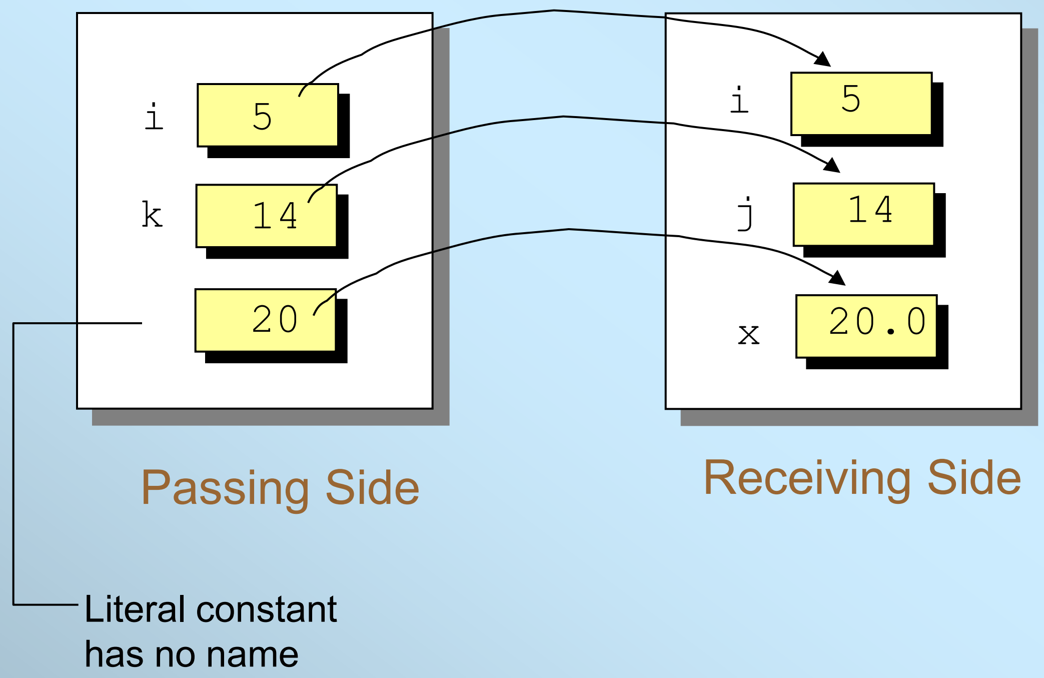
```
class Demo {

  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);
```

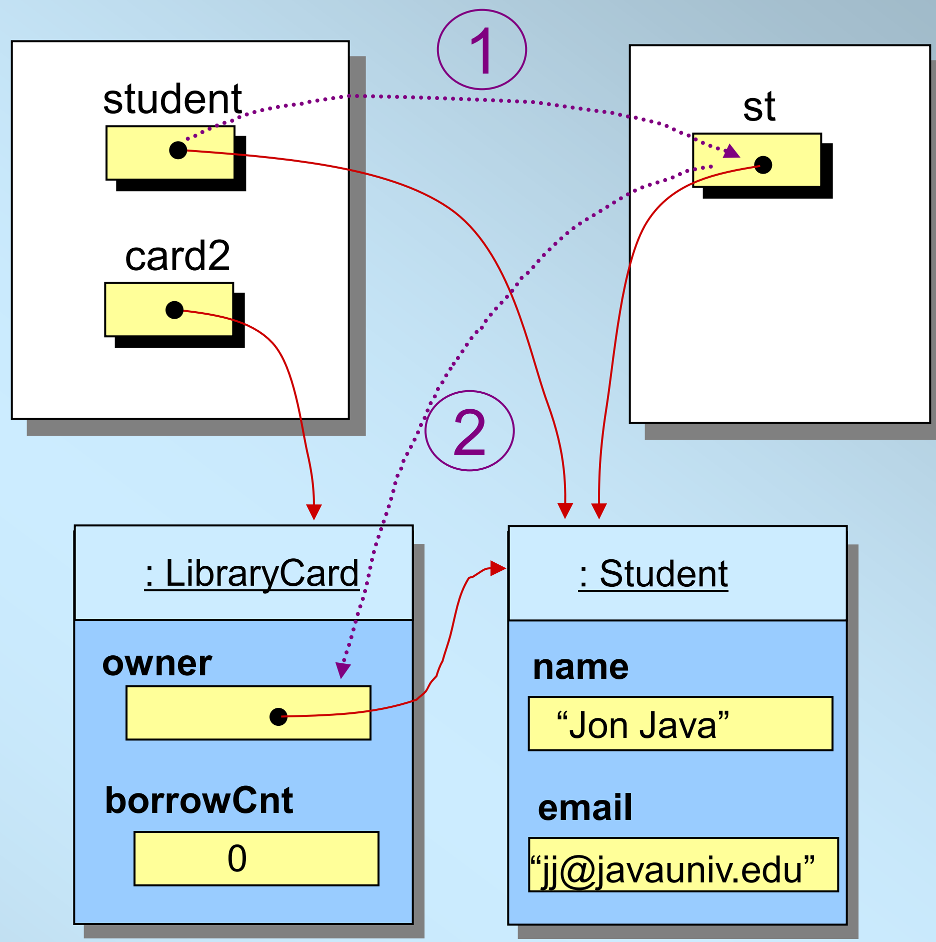
Passing Side

```
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



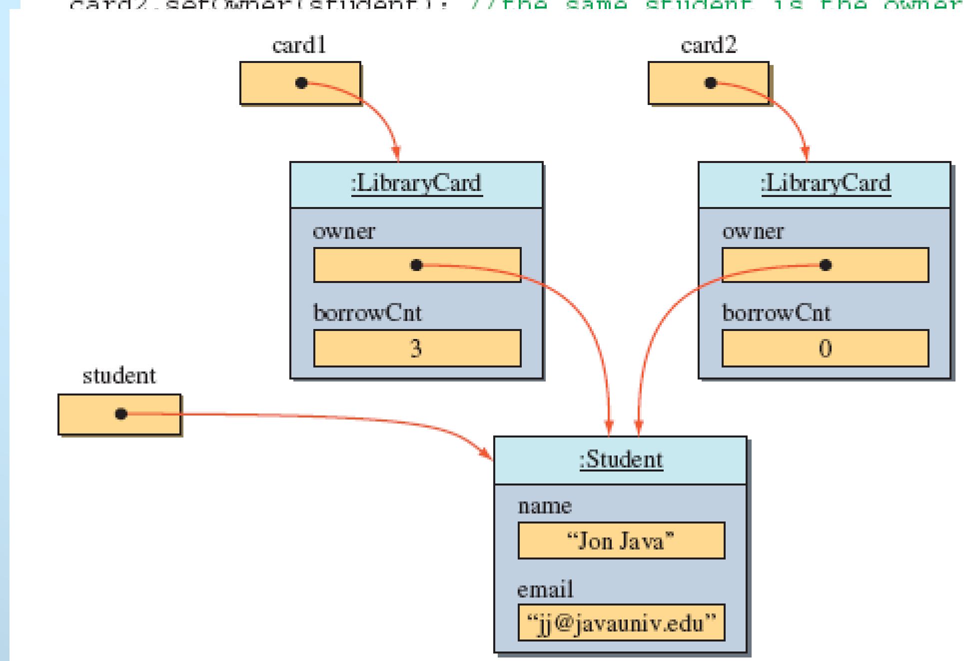
State of Memory

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
```

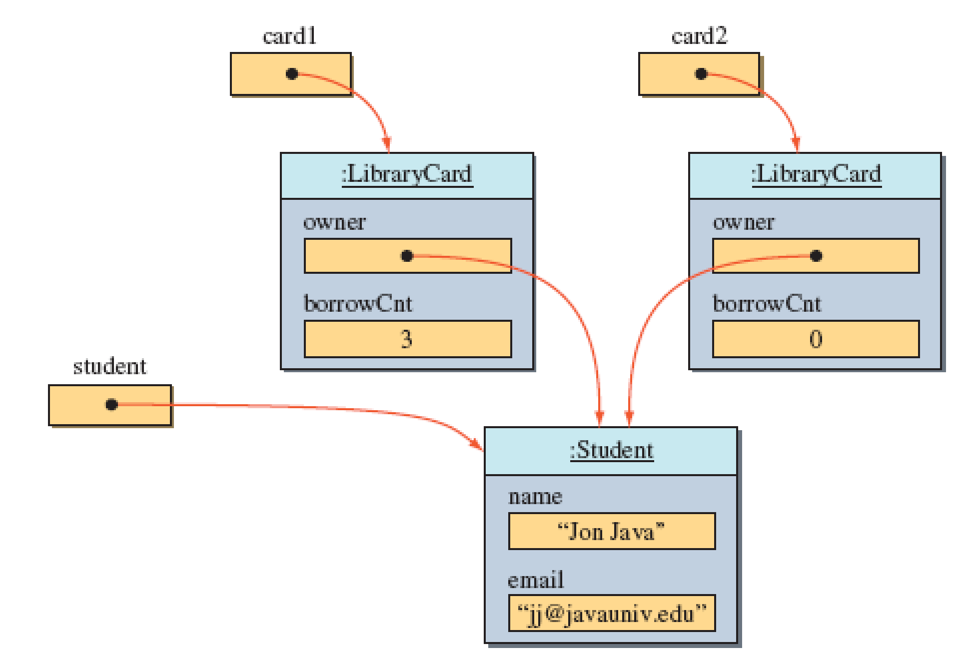
- Since we are actually passing a reference to the same object, it results in the owner of two LibraryCard objects pointing to the same Student object



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

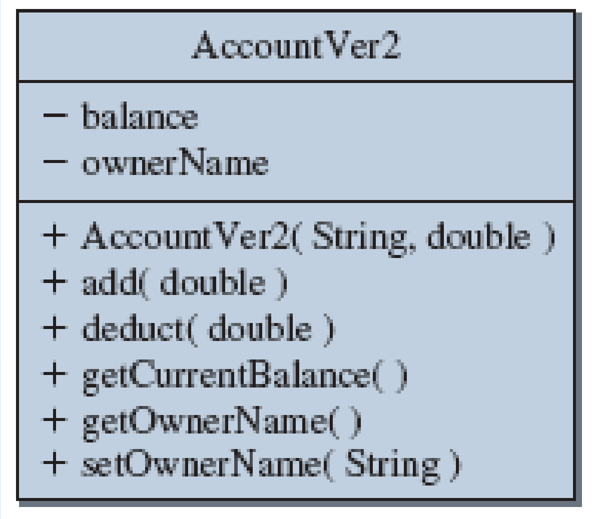
Data Members Should Be private

- 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 (-)

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;        local variable

    result = Math.sqrt(num * num);

    return result;
}
```

Local, Parameter & Data Member

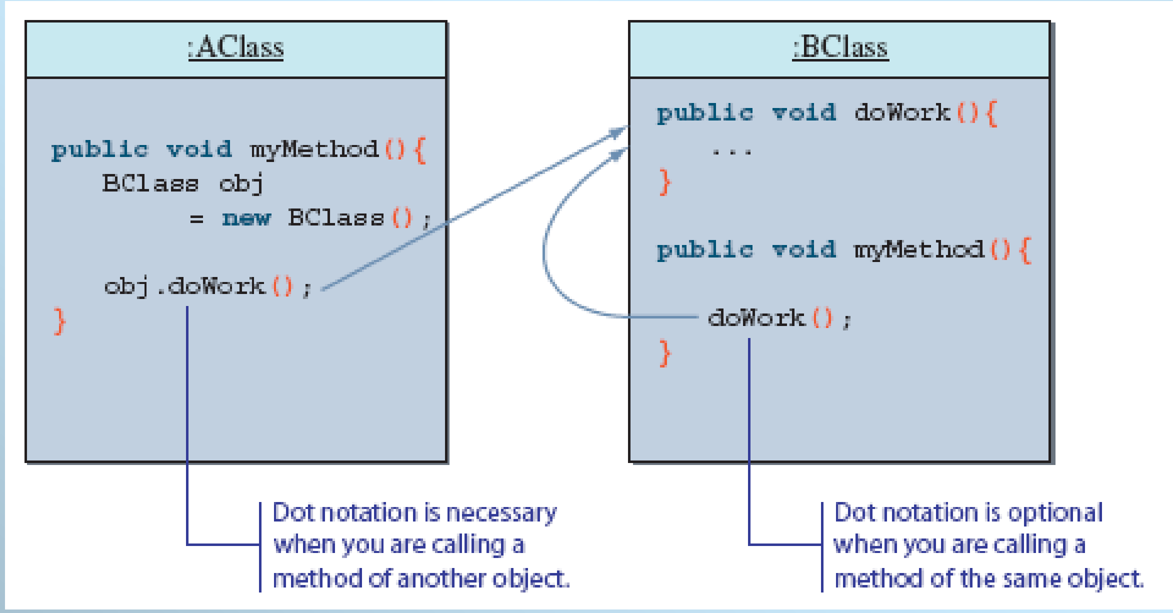
- An identifier appearing inside a method can be a local variable, a parameter, or a data member.
- The rules are
  - If there's a matching local variable declaration or a parameter, then the identifier refers to the local variable or the parameter.
  - Otherwise, if there's a matching data member declaration, 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;
    }
    ...
}
```

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

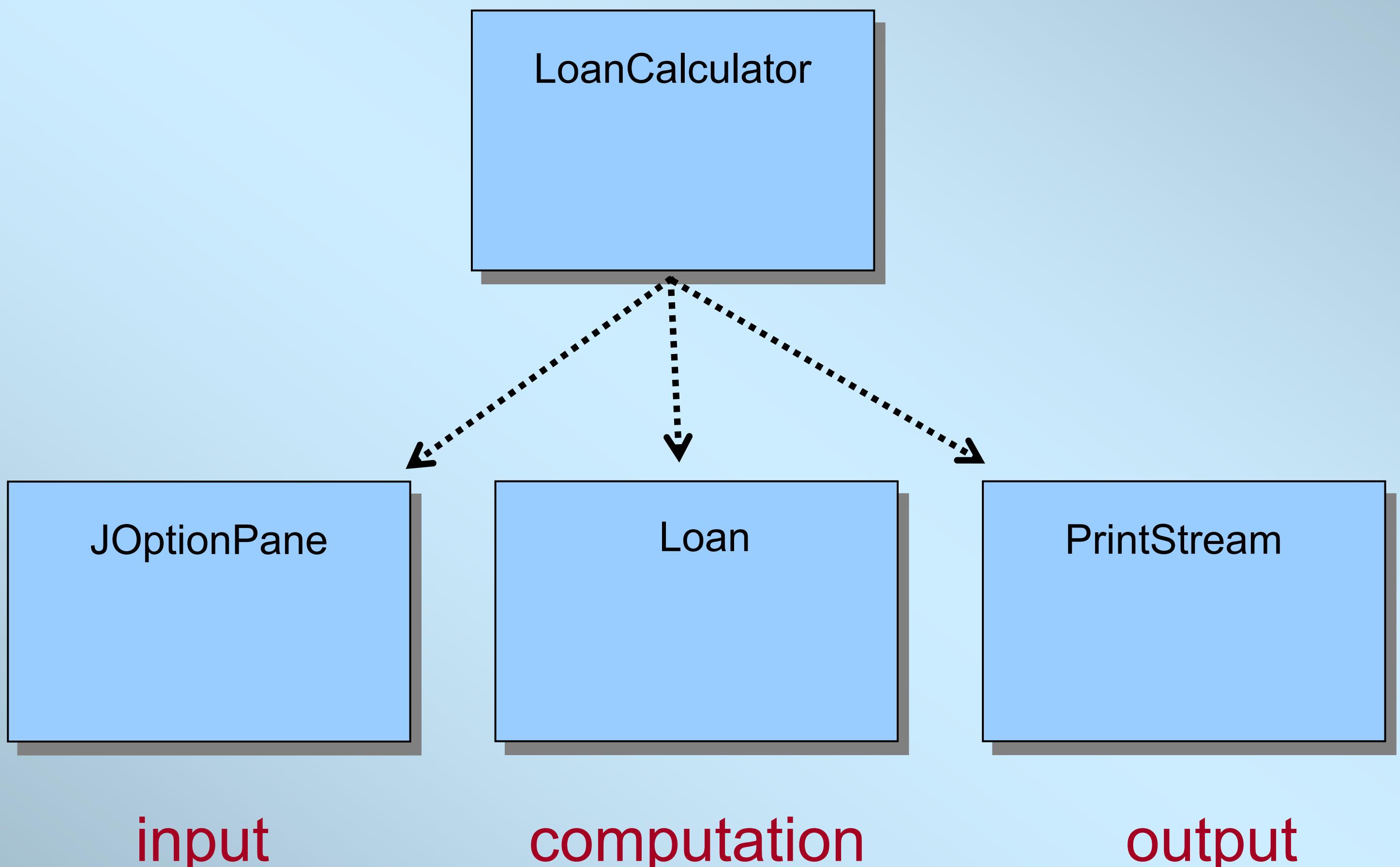
- Problem statement:

  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

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.

Step 2 Code

Directory: Chapter4/Step2

Source Files:

LoanCalculator.java
Loan.java

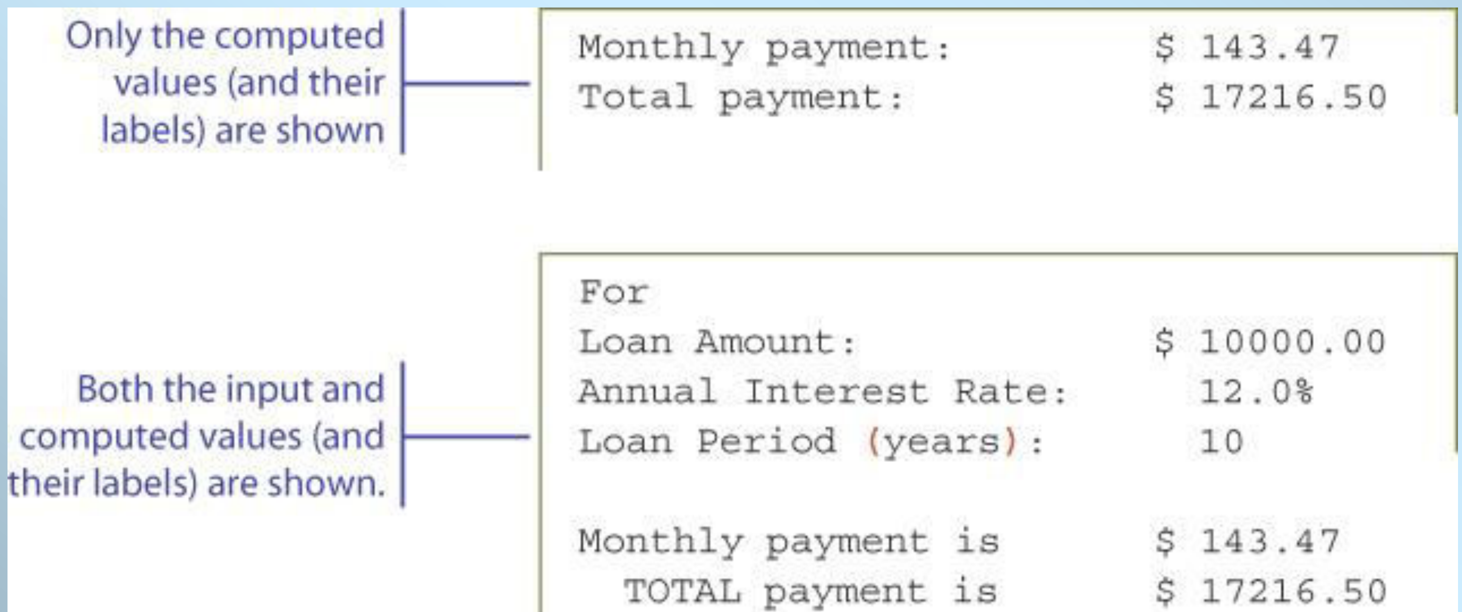
Step 2 Test

- 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.



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.

Step 4 Code

Directory: Chapter4/Step4

Source Files:

LoanCalculator.java
Loan.java

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.

Directory: Chapter4/Step5

Source Files (final version):
LoanCalculator.java
Loan.java

Chapter 7

Defining Your Own Classes
Part 2

Animated Version

Objectives

- After you have read and studied this chapter, you should be able to
  - Describe how objects are returned from methods
  - Describe how the reserved word this is used
  - Define overloaded methods and constructors
  - Define class methods and variables
  - Describe how the arguments are passed to the parameters using the pass-by-value scheme
  - Document classes with javadoc comments
  - Organize classes into a package

Returning an Object from a Method

- As we can return a primitive data value from a method, we can return an object from a method also.
- We return an object from a method, we are actually returning a reference (or an address) of an object.
  - This means we are not returning a copy of an object, but only the reference of this object

Sample Object-Returning Method

- Here's a sample method that returns an object:

```
public Fraction simplify( ) {

    Fraction simp;

    int num   = getNumberator();
    int denom = getDenominator();
    int gcd   = gcd(num, denom);

    simp = new Fraction(num/gcd, denom/gcd);

    return simp;
}
```

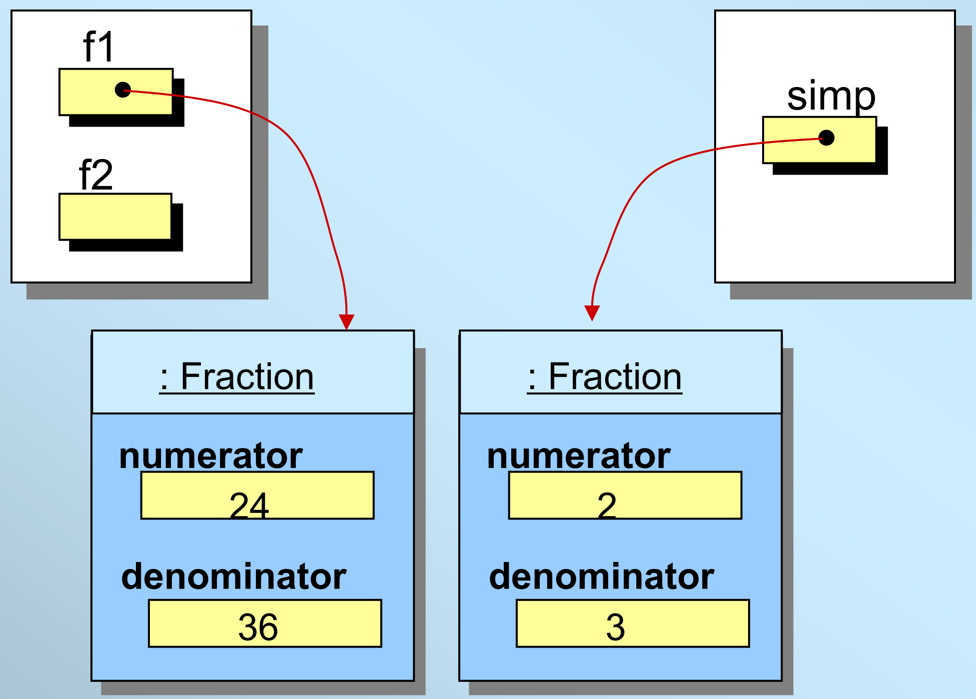
Return type indicates the class of an object we're returning from the method.

Return an instance of the Fraction class

A Sample Call to simplify

```
f1 = new Fraction(24, 26);

f2 = f1.simplify();
```

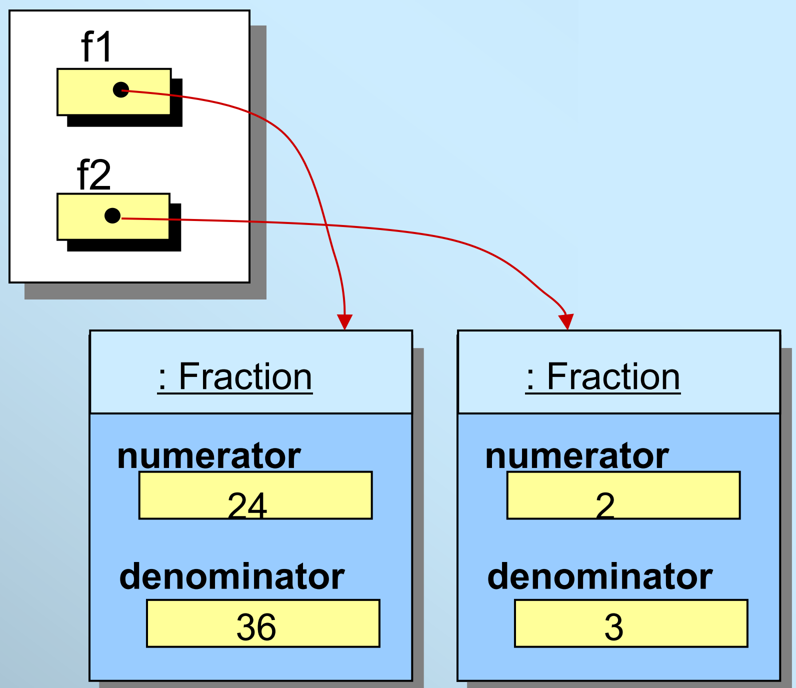
```
public Fraction simplify( ) {

   int num   = getNumerator();
   int denom = getDenominator();
   int gcd   = gcd(num, denom);

   Fraction simp = new
        Fraction(num/gcd, denom/gcd);

   return simp;
}
```



A Sample Call to simplify (cont'd)

```
f1 = new Fraction(24, 26);

f2 = f1.simplify();
```

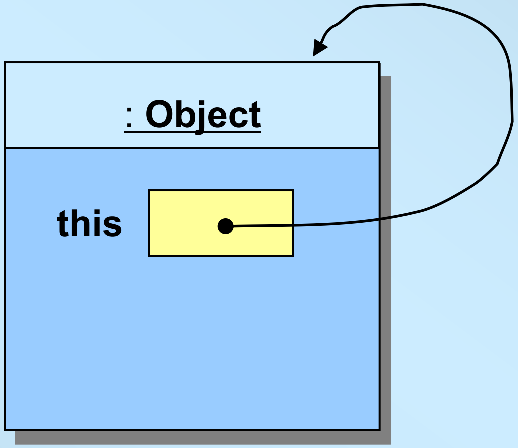
```
public Fraction simplify( ) {

   int num   = getNumerator();
   int denom = getDenominator();
   int gcd   = gcd(num, denom);

   Fraction simp = new
        Fraction(num/gcd, denom/gcd);

   return simp;
}
```



The value of simp, which is a reference, is returned and assigned to f2.

Reserved Word this

- The reserved word this is called a *self-referencing pointer* because it refers to an object from the object's method.

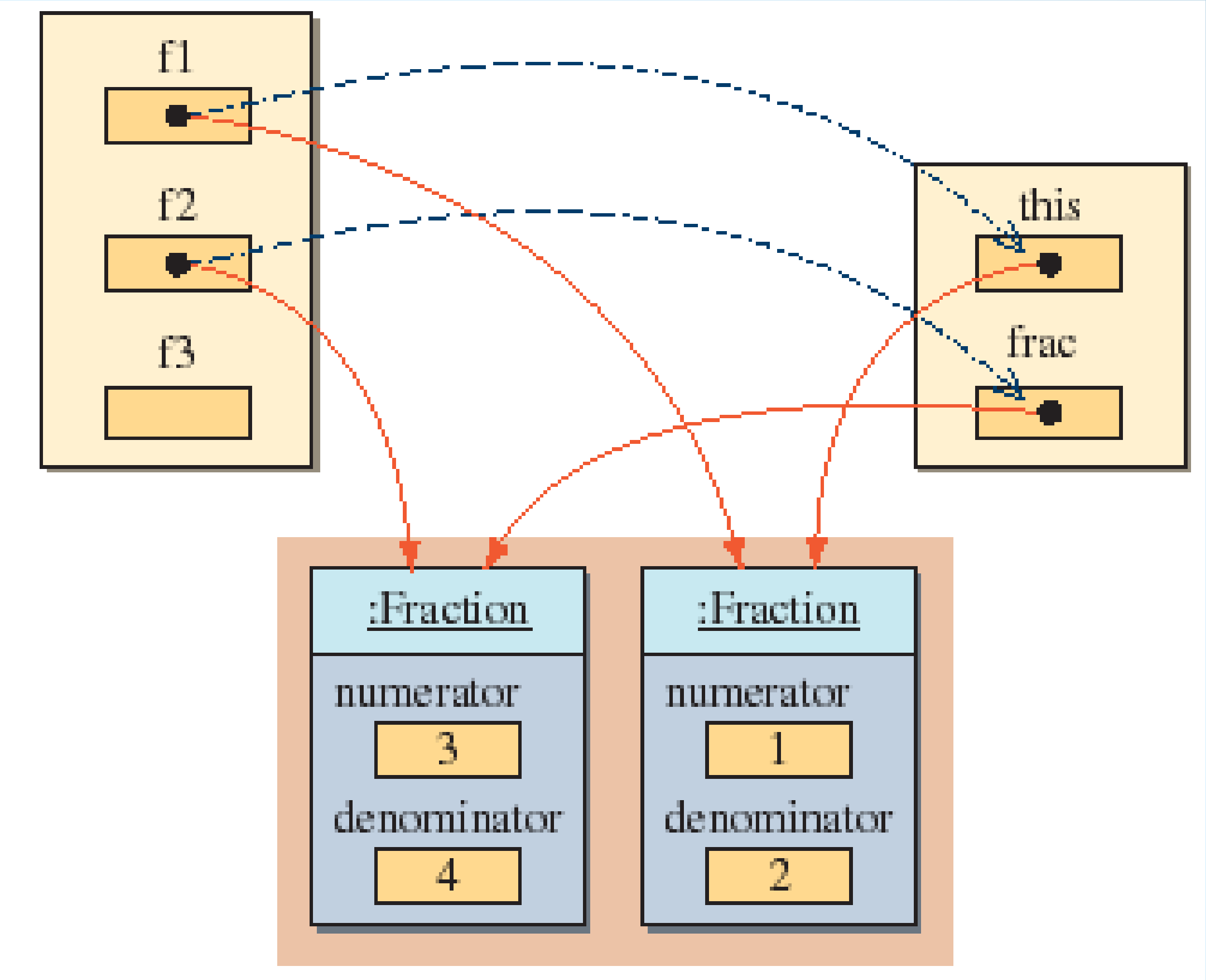


- The reserved word this can be used in three different ways. We will see all three uses in this chapter.

The Use of this in the add Method

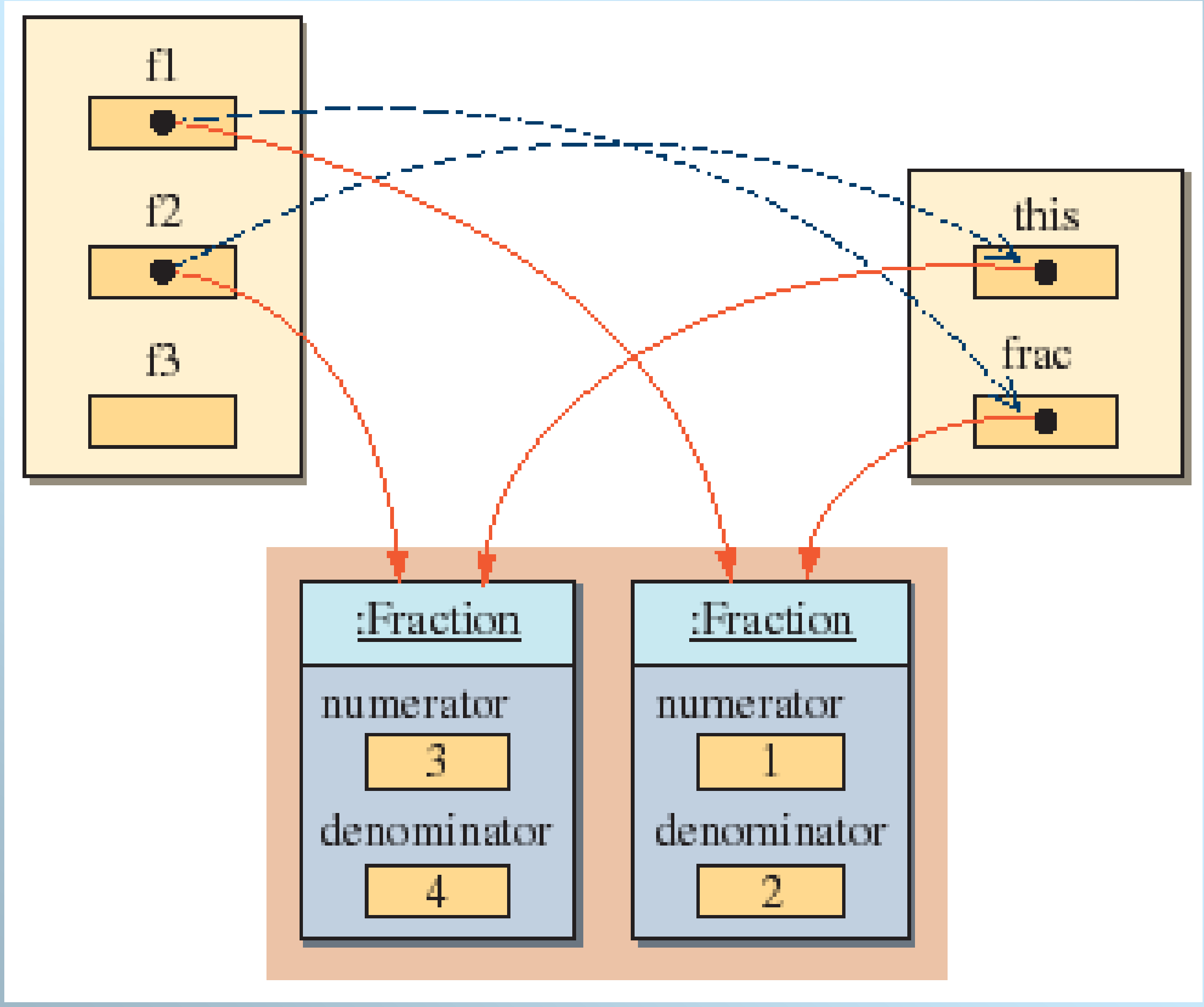
```
public Fraction add(Fraction frac) {

   int      a, b, c, d;
   Fraction sum;

   a = this.getNumerator();   //get the receiving
   b = this.getDenominator(); //object's num and denom

   c = frac.getNumerator();   //get frac's num
   d = frac.getDenominator(); //and denom

   sum = new Fraction(a*d + b*c, b*d);

   return sum;
}
```

f3 = f1.add(f2)



Because f1 is the receiving object (we're calling f1's method), so the reserved word this is referring to f1.

f3 = f2.add(f1)



This time, we're calling f2's method, so the reserved word this is referring to f2.

Using this to Refer to Data Members

- In the previous example, we showed the use of this to call a method of a receiving object.
- It can be used to refer to a data member as well.

```
class Person {

   int  age;

    public void setAge(int val) {
       this.age = val;
   }
   . . .
}
```

Overloaded Methods

- Methods can share the same name as long as
  - they have a different number of parameters (Rule 1) or
  - their parameters are of different data types when the number of parameters is the same (Rule 2)

```
public void myMethod(int x, int y) { ... }
public void myMethod(int x) { ... }
```

✓ Rule 1

```
public void myMethod(double x) { ... }
public void myMethod(int x) { ... }
```

✓ Rule 2

Overloaded Constructor

- The same rules apply for overloaded constructors
  - this is how we can define more than one constructor to a class

```
public Person( ) { ... }

public Person(int age) { ... }
```

✓ Rule 1

```
public Pet(int age) { ... }

public Pet(String name) { ... }
```

✓ Rule 2

Constructors and this

- To call a constructor from another constructor of the same class, we use the reserved word **this**.

```
public Fraction( ) {
    //creates 0/1
    this(0. 1);
}

public Fraction(int number) {
    //creates number/1
    this(number, 1);
}

public Fraction(Fraction frac) {
    //copy constructor
    this(frac.getNumerator(),
          frac.getDenominator());
}

public Fraction(int num, int denom) {
    setNumerator(num);
    setDenominator(denom);
}
```

Class Methods

- We use the reserved word static to define a class method.

```
public static int gcd(int m, int n) {

   //the code implementing the Euclidean algorithm
}


public static Fraction min(Fraction f1, Fraction f2) {

   //convert to decimals and then compare

}
```

Call-by-Value Parameter Passing

- When a method is called,
  - the value of the argument is passed to the matching parameter, and
  - separate memory space is allocated to store this value.
- This way of passing the value of arguments is called a *pass-by-value* or *call-by-value scheme*.
- Since separate memory space is allocated for each parameter during the execution of the method,
  - the parameter is local to the method, and therefore
  - changes made to the parameter will not affect the value of the corresponding argument.

Call-by-Value Example

```
class Tester {
    public void myMethod(int one, double two ) {
        one = 25;
        two = 35.4;
    }
}
```

```
Tester tester;
int x, y;
tester = new Tester();
x = 10;
y = 20;
tester.myMethod(x, y);
System.out.println(x + " " + y);
```

produces

```
10 20
```

Memory Allocation for Parameters

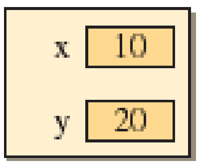
1

```
x = 10;
y = 20;
tester.myMethod( x, y );
```

execution flow

```
public void myMethod( int one, double two ) {

  one = 25;
  two = 35.4;
}
```

at 1 before calling myMethod



state of memory

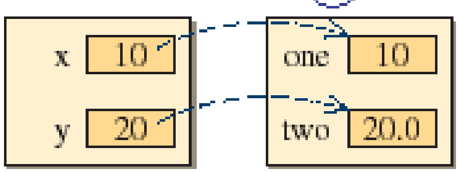
Local variables do not exist
before the method execution.

2

```
x = 10;
y = 20;
tester.myMethod( x, y );
```

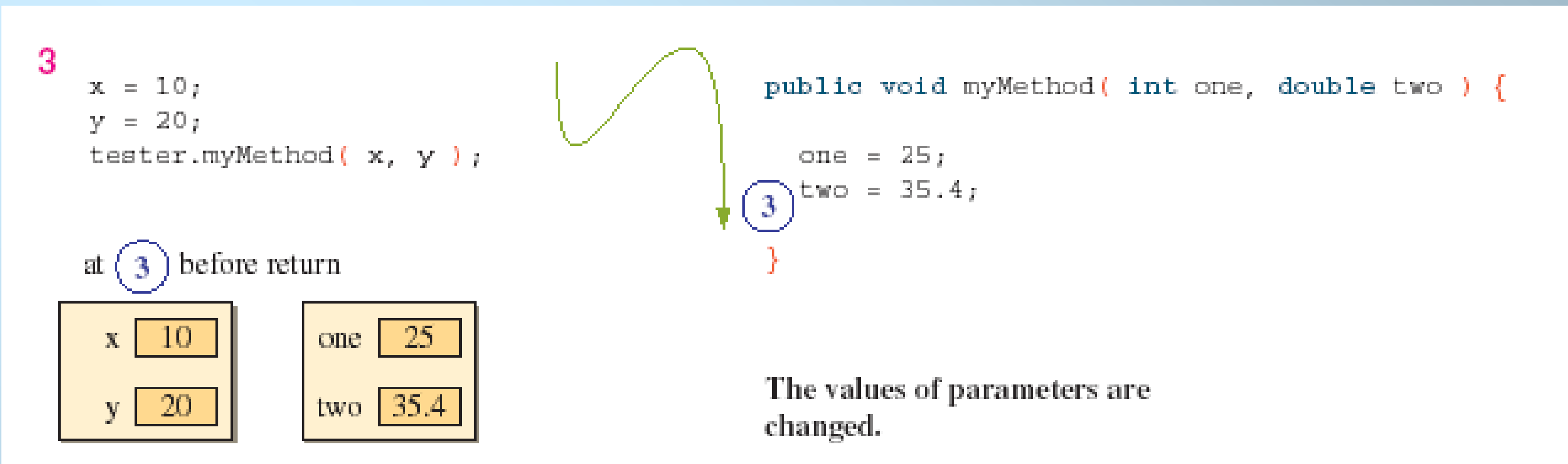
```
public void myMethod( int one, double two ) {  2

  one = 25;
  two = 35.4;
}
```

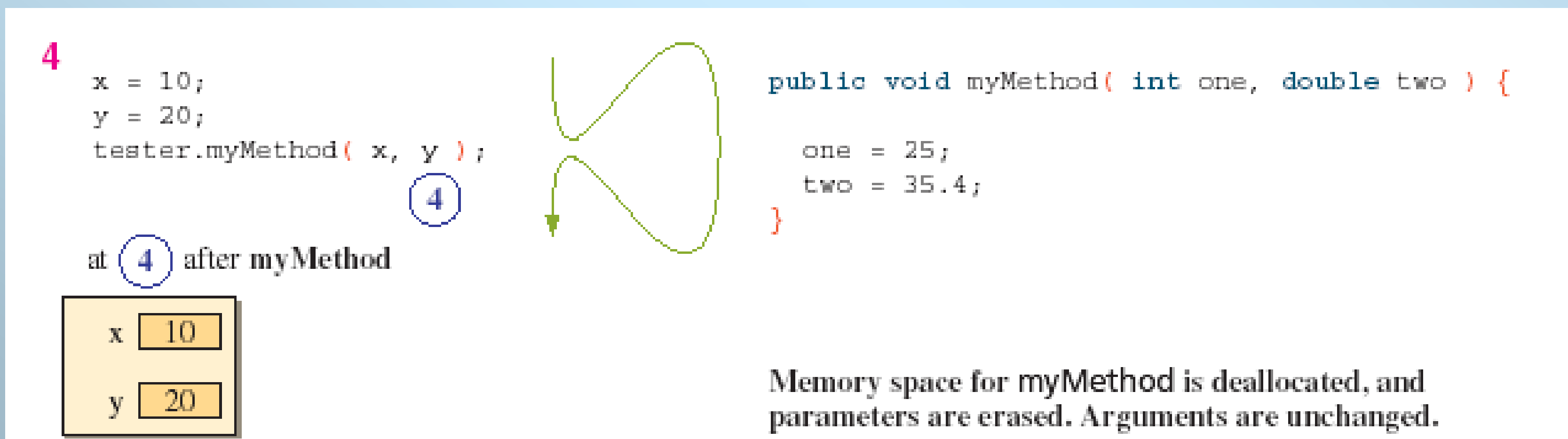
values are copied at 2



Memory space for myMethod is allocated, and the values
of arguments are copied to the parameters.

Memory Allocation for Parameters (cont'd)





Parameter Passing: Key Points

1. *Arguments are passed to a method by using the pass-by- value scheme.*
2. *Arguments are matched to the parameters from left to right.The data type of an argument must be assignment-compatible with the data type of the matching parameter.*
3. *The number of arguments in the method call must match the number of parameters in the method definition.*
4. *Parameters and arguments do not have to have the same name.*
5. *Local copies, which are distinct from arguments,are created even if the parameters and arguments share the same name.*
6. *Parameters are input to a method, and they are local to the method.Changes made to the parameters will not affect the value of corresponding arguments.*

Organizing Classes into a Package

- For a class A to use class B, their bytecode files must be located in the same directory.
  - This is not practical if we want to reuse programmer-defined classes in many different programs
- The correct way to reuse programmer-defined classes from many different programs is to place reusable classes in a package.
- A *package* is a Java class library.

Creating a Package

- The following steps illustrate the process of creating a package name myutil that includes the Fraction class.

  1. Include the statement

  ```
  package myutil;
  ```

  as the first statement of the source file for the Fraction class.

  2. The class declaration must include the visibility modifier public as

  ```
  public class Fraction {
     ...
  }
  ```

  3. Create a folder named myutil, the same name as the package name. In Java, the package must have a one-to-one correspondence with the folder.

  4. Place the modified Fraction class into the myutil folder and compile it.

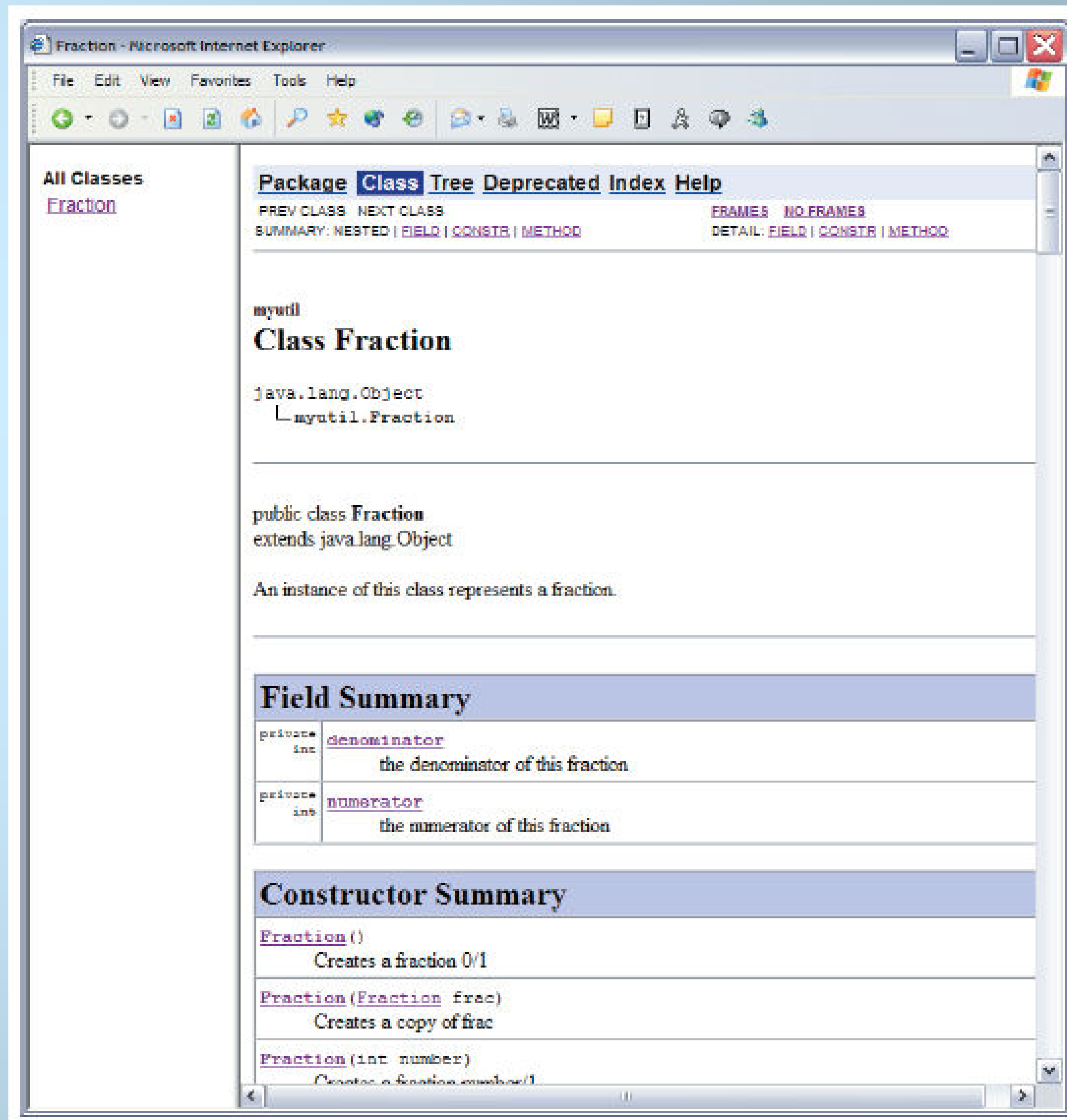
  5. Modify the CLASSPATH environment variable to include the folder that contains the myutil folder.

Using Javadoc Comments

- Many of the programmer-defined classes we design are intended to be used by other programmers.
  - It is, therefore, very important to provide meaningful documentation to the client programmers so they can understand how to use our classes correctly.
- By adding javadoc comments to the classes we design, we can provide a consistent style of documenting the classes.
- Once the javadoc comments are added to a class, we can generate HTML files for documentation by using the javadoc command.

javadoc for Fraction

- This is a portion of the HTML documentation for the Fraction class shown in a browser.
- This HTML file is produced by processing the javadoc comments in the source file of the Fraction class.



javadoc Tags

- The javadoc comments begins with /** and ends with */
- Special information such as the authors, parameters, return values, and others are indicated by the @ marker

    @param

    @author

    @return

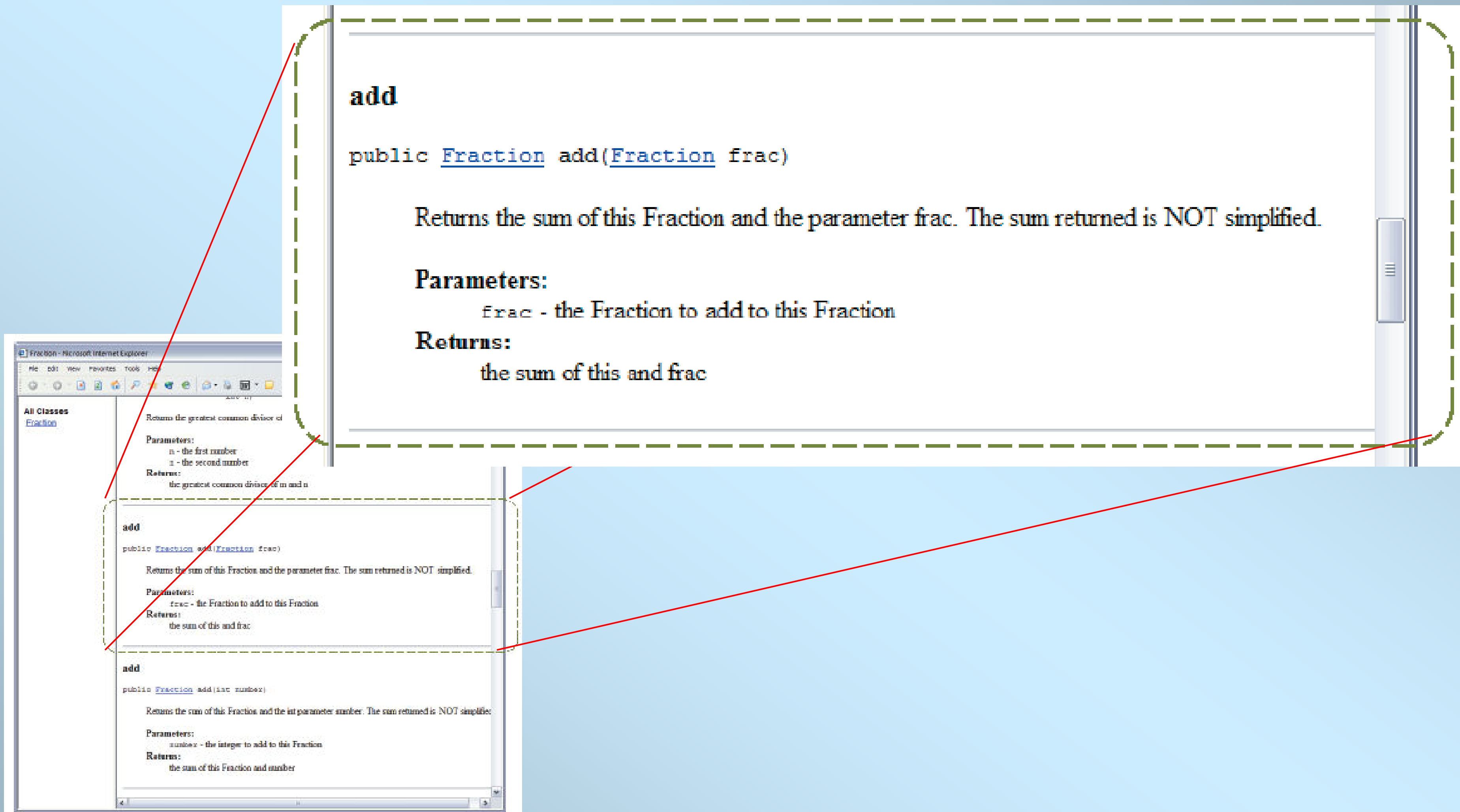
    etc

Example: javadoc Source

```
. . .

/**
* Returns the sum of this Fraction
* and the parameter frac. The sum
* returned is NOT simplified.
*
* @param frac the Fraction to add to this
*             Fraction
*
* @return the sum of this and frac
*/
public Fraction add(Fraction frac) {
   ...
}
. . .
```

this javadoc will produce

Example: javadoc Output



javadoc Resources

- General information on javadoc is located at

  http://java.sun.com/j2se/javadoc

- Detailed reference on how to use javadoc on Windows is located at

  http://java.sun.com/j2se/1.5/docs/tooldocs/windows/javadoc.html

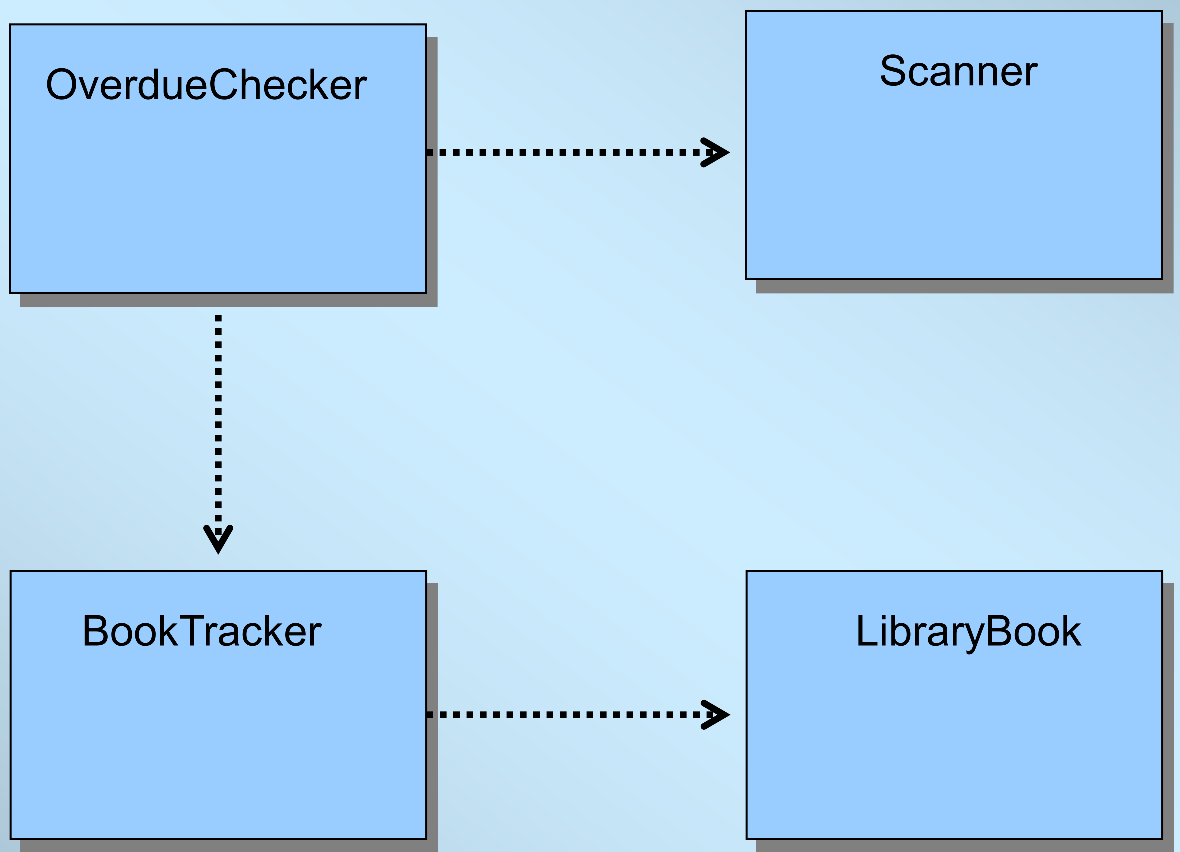
Problem Statement

Write an application that computes the total charges for the overdue library books. For each library book, the user enters the due date and (optionally) the overdue charge per day,the maximum charge, and the title. If the optional values are not entered, then the preset default values are used. A complete list of book information is displayed when the user finishes entering the input data.The user can enter different return dates to compare the overdue charges.

Overall Plan

- Tasks:
  1. Get the information for all books
  2. Display the entered book information
  3. Ask for the return date and display the total charge. Repeat this step until the user quits.

Required Classes



helper class

Development Steps

- We will develop this program in five steps:

1. Define the basic LibraryBook class.
2. Explore the given BookTracker class and integrate it with the LibraryBook class.
3. Define the top-level OverdueChecker class. Implement the complete input routines.
4. Complete the LibraryBook class by fully implementing the overdue charge computation.
5. Finalize the program by tying up loose ends.

Step 1 Design

- Develop the basic LibraryBook class.
- The key design task is to identify the data members for storing relevant information.
- We will include multiple constructors for ease of creating LibraryBook objects.
  - Make sure that an instance will be initiated correctly no matter which constructor is used.

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: Chapter7/Step1

Source Files: LibraryBook.java
Step1Main.java (test program)

Step 1 Test

- In the testing phase, we run the test main program Step1Main and confirm that we get the expected output:

```
Title unknown                         $ 0.50      $  50.00      03/14/04
Introduction to OOP with Java         $ 0.75      $  50.00      02/13/04
Java for Smarties                     $ 1.00      $ 100.00      01/12/04
Me and My Java                        $ 1.50      $ 230.00      01/01/04
```

Step 2 Design

- Explore the helper BookTracker class and incorporate it into the program.
- Adjust the LibraryBook class to make it compatible with the BookTracker class.

Step 2 Code

Directory: Chapter7/Step2

Source Files: LibraryBook.java
Step2Main.java (test program)

Step 2 Test

- In the testing phase, we run the test main program Step2Main and confirm that we get the expected output.
- We run the program multiple times trying different variations each time.

Step 3 Design

- We implement the top-level control class OverdueChecker.
- The top-level controller manages a single BookTracker object and multiple LibraryBook objects.
- The top-level controller manages the input and output routines
  - If the input and output routines are complex, then we would consider designing separate classes to delegate the I/O tasks.

Step 3 Pseudocode

```
GregorianCalendar returnDate;
String reply, table;
double totalCharge;

inputBooks(); //read in all book information

table = bookTracker.getList();
System.out.println(table);

//try different return dates
do {
    returnDate = read return date ;
    totalCharge = bookTracker.getCharge(returnDate);
    displayTotalCharge(totalCharge);
    reply = prompt the user to continue or not;
} while ( reply is yes );
```

Step 3 Code

Directory: Chapter7/Step3

Source Files: OverdueChecker.java
LibraryBook.java

Step 3 Test

- Now we run the program multiple times, trying different input types and values.
- We confirm that all control loops are implemented and working correctly.
  - At this point, the code to compute the overdue charge is still a stub, so we will always get the same overdue charge for the same number of books.
- After we verify that everything is working as expected,we proceed to the next step.

Step 4: Compute the Charge

- To compute the overdue charge, we need two dates: the due date and the date the books are or to be returned.
- The getTimeInMillis method returns the time elasped since the epoch to the date in milliseconds.
- By subtracting this since-the-epoch milliseconds value of the due date from the same of the return date, we can find the difference between the two.
  - If the difference is negative, then it's not past due, so there's no charge.
  - If the difference is positive, then we convert the milliseconds to the equivalent number of days and multiply it by the per-day charge to compute the total charge.

Step 4 Code

Directory: Chapter7/Step3

Source Files: OverdueChecker.java
LibraryBook.java

Step 4 Test

- We run the program mutiple times again, possibly using the same set of input data.
- We enter different input variations to try out all possible cases for the computeCharge method.
  - Try cases such as the return date and due date are the same, the return date occurs before the due date, the charge is beyond the maximum, and so forth.
- After we verify the program,we move on to the next step.

Step 5: Finalize / Extend

- Program Review
  - Are all the possible cases handled?
  - Are the input routines easy to use?
  - Will it be better if we allow different formats for entering the date information?
- Possible Extensions
  - Warn the user, say, by popping a warning window or ringing an alarm, when the due date is approaching.
  - Provide a special form window to enter data

    (Note: To implement these extensions, we need techniques not covered yet.)

Chapter 6

Repetition Statements

Animated Version

Objectives

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

- Implement repetition control in a program using while statements.
- Implement repetition control in a program using do-while statements.
- Implement a generic loop-and-a-half repetition control statement
- Implement repetition control in a program using for statements.
- Nest a loop repetition statement inside another repetition statement.
- Choose the appropriate repetition control statement for a given task
- Prompt the user for a yes-no reply using the showConfirmDialog method of JOptionPane.
- (Optional) Write simple recursive methods

Definition

- Repetition statements control a block of code to be executed for a fixed number of times or until a certain condition is met.
- Count-controlled repetitions terminate the execution of the block after it is executed for a fixed number of times.
- Sentinel-controlled repetitions terminate the execution of the block after one of the designated values called a *sentinel* is encountered.
- Repetition statements are called loop statements also.

The while Statement

```
int sum = 0, number = 1;

while ( number <= 100 ) {

   sum    =  sum + number;

   number = number + 1;

}
```

These statements are executed as long as number is less than or equal to 100.

Syntax for the while Statement

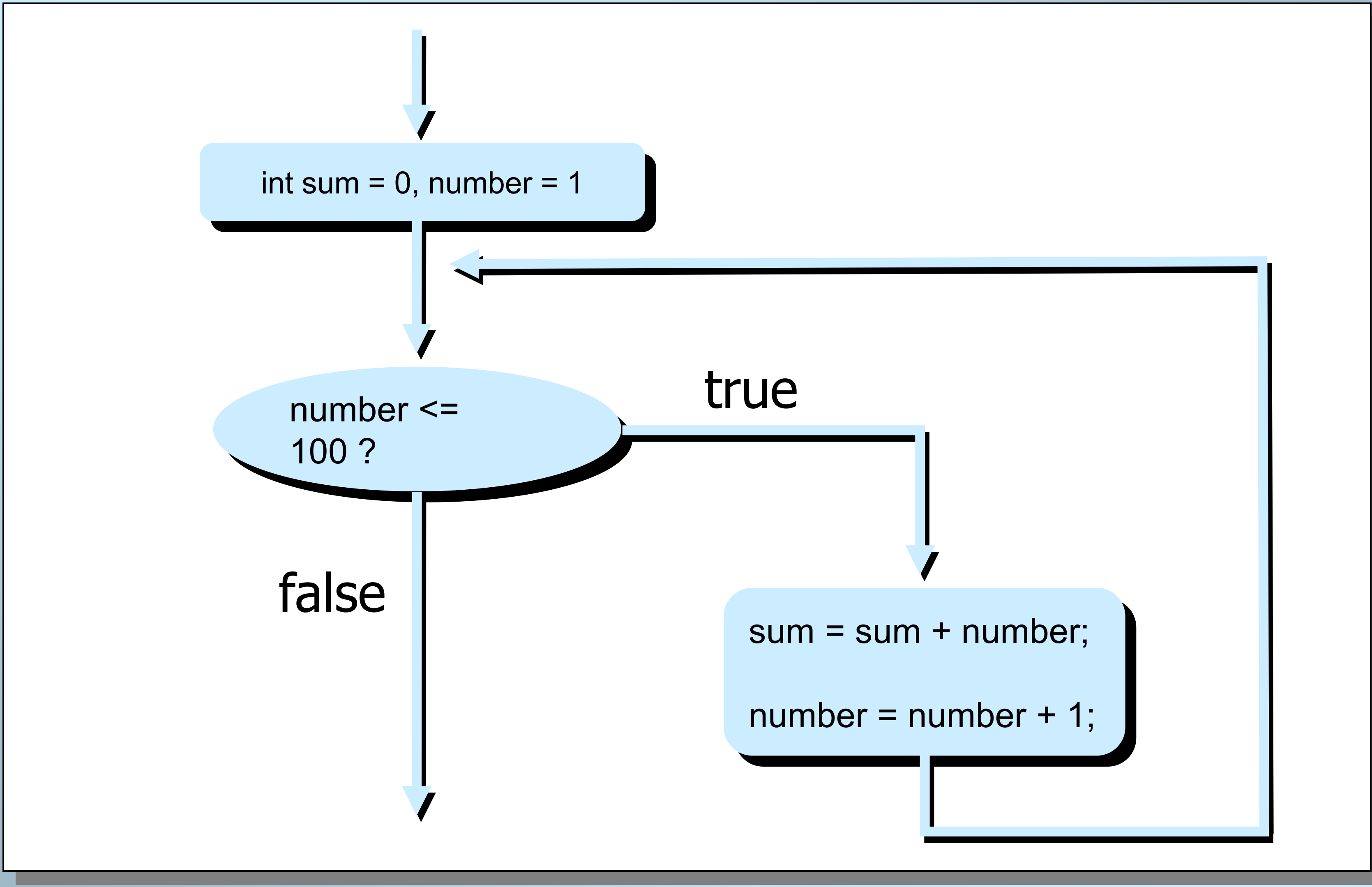
```
while ( <boolean expression> )

      <statement>
```

Boolean Expression

Statement (loop body)

```
while (  number <= 100  ) {

    sum    =  sum + number;

    number = number + 1;
}
```

Control Flow of while



More Examples

1

```
int sum = 0, number = 1;

while ( sum <= 1000000 ) {
     sum    =  sum + number;
     number = number + 1;
}
```

Keeps adding the numbers 1, 2, 3, … until the sum becomes larger than 1,000,000.

2

```
int product =  1, number = 1,
    count   = 20, lastNumber;


lastNumber = 2 * count - 1;


while (number <= lastNumber) {
     product = product * number;
     number  = number + 2;
}
```

Computes the product of the first 20 odd integers.

Finding GCD

```java
public int gcd_bruteforce(int m, int n) {

    //assume m, n >= 1

    int last = Math.min(m, n);

    int gcd;
    int i = 1;

    while (i <= last) {

        if (m % i == 0 && n % i == 0) {

            gcd = i;
        }

        i++;
    }

    return gcd;
}
```

Direct Approach

```java
public int gcd(int m, int n) {

    //it doesn't matter which of n and m is bigger
    //this method will work fine either way

    //assume m,n >= 1

    int r = n % m;

    while (r !=0) {

        n = m;

        m = r;

        r = n % m;
    }

    return m;
}
```

More Efficient Approach

Example: Testing Input Data

Priming Read

```
String inputStr;
int    age;

inputStr = JOptionPane.showInputDialog(null,
                                   "Your Age (between 0 and 130):");
age      = Integer.parseInt(inputStr);

while (age < 0 || age > 130) {
    JOptionPane.showMessageDialog(null,
        "An invalid age was entered. Please try again.");

    inputStr = JOptionPane.showInputDialog(null,
                                   "Your Age (between 0 and 130):");

    age = Integer.parseInt(inputStr);
}
```

Useful Shorthand Operators

```
sum = sum + number;
```

is equivalent to

```
sum += number;
```

Operator	Usage	Meaning
`+=`	`a += b;`	`a = a + b;`
`-=`	`a -= b;`	`a = a - b;`
`*=`	`a *= b;`	`a = a * b;`
`/=`	`a /= b;`	`a = a / b;`
`%=`	`a %= b;`	`a = a % b;`

Watch Out for Pitfalls

1. Watch out for the off-by-one error (OBOE).
2. Make sure the loop body contains a statement that will eventually cause the loop to terminate.
3. Make sure the loop repeats exactly the correct number of times.
4. If you want to execute the loop body N times, then initialize the counter to 0 and use the test condition counter < N or initialize the counter to 1 and use the test condition counter <= N.

Loop Pitfall - 1

1

```
int product = 0;

while ( product < 500000 ) {
    product = product * 5;
}
```

2

```
int count = 1;

while ( count != 10 ) {
    count = count + 2;
}
```

Infinite Loops

Both loops will not terminate because the boolean expressions will never become false.

Overflow

- An infinite loop often results in an overflow error.
- An **overflow error** occurs when you attempt to assign a value larger than the maximum value the variable can hold.
- In Java, an overflow does not cause program termination. With types **float** and **double**, a value that represents infinity is assigned to the variable. With type **int**, the value “wraps around” and becomes a negative value.

Loop Pitfall - 2

1

```
float count = 0.0f;

while ( count != 1.0f ) {
    count = count + 0.3333333f;
}                   //seven 3s
```

2

```
float count = 0.0f;

while ( count != 1.0f ) {
    count = count + 0.33333333f;
}                   //eight 3s
```

Using Real Numbers

Loop 2 terminates, but Loop 1 does not because only an approximation of a real number can be stored in a computer memory.

Loop Pitfall – 2a

1

```
int result = 0; double cnt = 1.0;
while (cnt <= 10.0){
    cnt += 1.0;
    result++;
}
System.out.println(result);
```

→ 10

2

```
int result = 0; double cnt = 0.0;
while (cnt <= 1.0){
    cnt += 0.1;
    result++;
}
System.out.println(result);
```

→ 11

Using Real Numbers

Loop 1 prints out 10, as expected, but Loop 2 prints out 11. The value 0.1 cannot be stored precisely in computer memory.

Loop Pitfall - 3

- Goal: Execute the loop body 10 times.

1

```
count = 1;
while ( count < 10 ){
    . . .
    count++;
}
```

2

```
count = 1;
while ( count <= 10 ){
    . . .
    count++;
}
```

3

```
count = 0;
while ( count <= 10 ){
    . . .
    count++;
}
```

4

```
count = 0;
while ( count < 10 ){
    . . .
    count++;
}
```

1 and 3 exhibit off-by-one error.

The do-while Statement

```
int sum = 0, number = 1;

do {

    sum += number;
    number++;

} while ( sum <= 1000000 );
```

These statements are executed as long as sum is less than or equal to 1,000,000.

Syntax for the do-while Statement

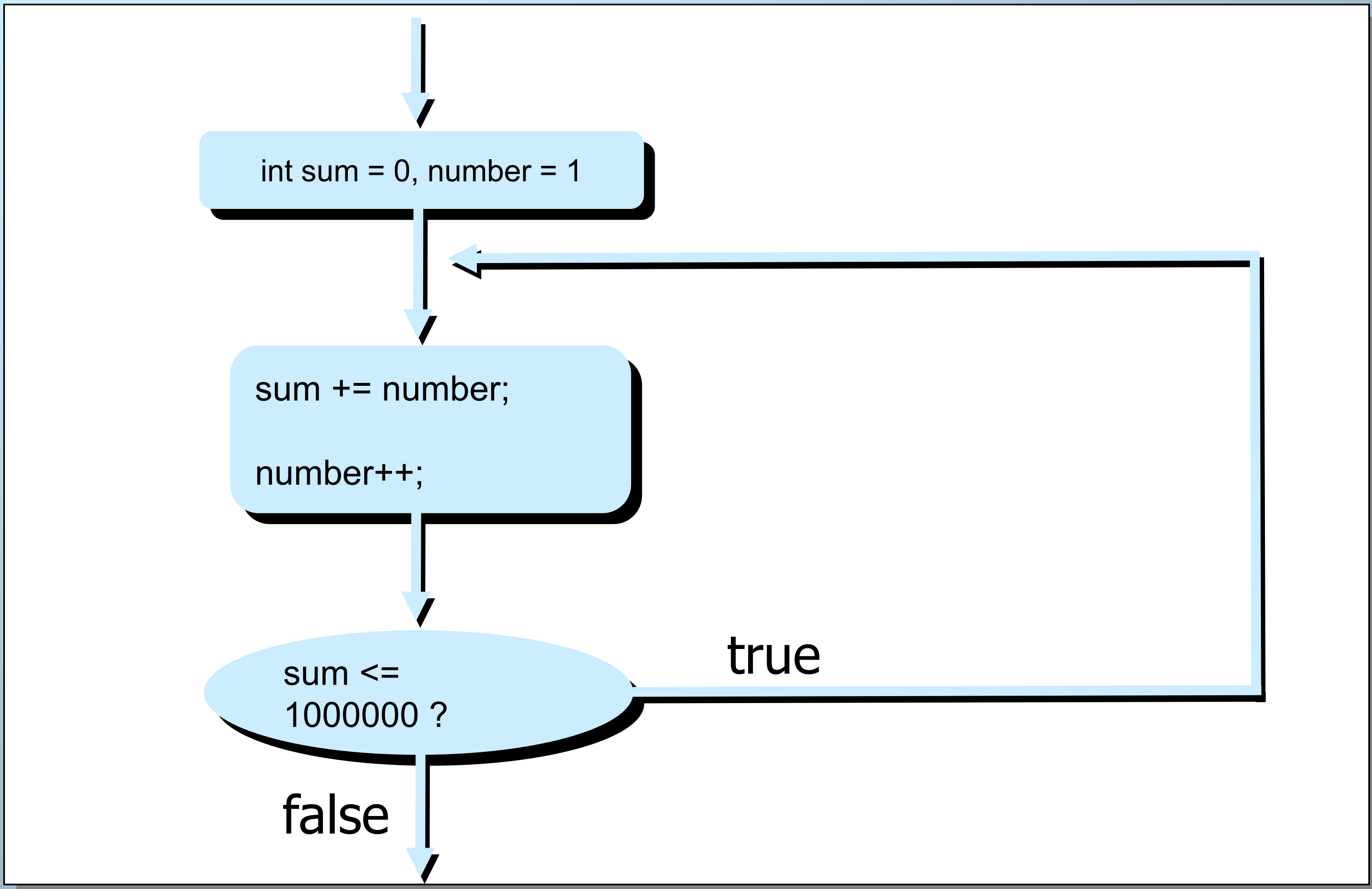
```
do
      <statement>

while ( <boolean expression> ) ;
```

```
do    {

   sum += number;
   number++;

}   while (  sum <= 1000000   );
```

Statement (loop body)

Boolean Expression

Control Flow of do-while



Loop-and-a-Half Repetition Control

- *Loop-and-a-half repetition control* can be used to test a loop's terminating condition in the middle of the loop body.
- It is implemented by using reserved words **while,** **if**, and **break**.

Example: Loop-and-a-Half Control

```
String name;

while (true){

    name = JOptionPane.showInputDialog(null, "Your name");

    if (name.length() > 0) break;

    JOptionPane.showMessageDialog(null, "Invalid Entry." +
                  "You must enter at least one character.");
}
```

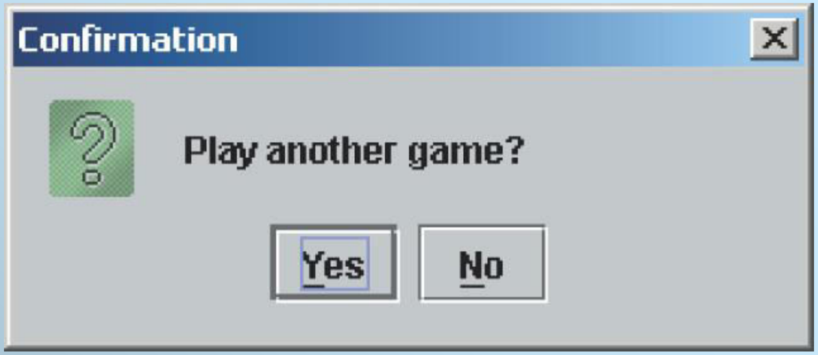
Pitfalls for Loop-and-a-Half Control

- Be aware of two concerns when using the loop-and-a-half control:
  - **The danger of an infinite loop.** The boolean expression of the `while` statement is true, which will always evaluate to true. If we forget to include an `if` statement to break out of the loop, it will result in an infinite loop.
  - **Multiple exit points.** It is possible, although complex, to write a correct control loop with multiple exit points (`break`s). It is good practice to enforce the *one-entry one-exit control* flow.

Confirmation Dialog

- A confirmation dialog can be used to prompt the user to determine whether to continue a repetition or not.

```
JOptionPane.showConfirmDialog(null,
        /*prompt*/           "Play Another Game?",
        /*dialog title*/     "Confirmation",
        /*button options*/ JOptionPane.YES_NO_OPTION);
```



Example: Confirmation Dialog

```
boolean keepPlaying = true;
int      selection;

while (keepPlaying){

    //code to play one game comes here
    // . . .

    selection = JOptionPane.showConfirmDialog(null,
                                              "Play Another Game?",
                                              "Confirmation",
                                              JOptionPane.YES_NO_OPTION);

    keepPlaying = (selection == JOptionPane.YES_OPTION);
}
```

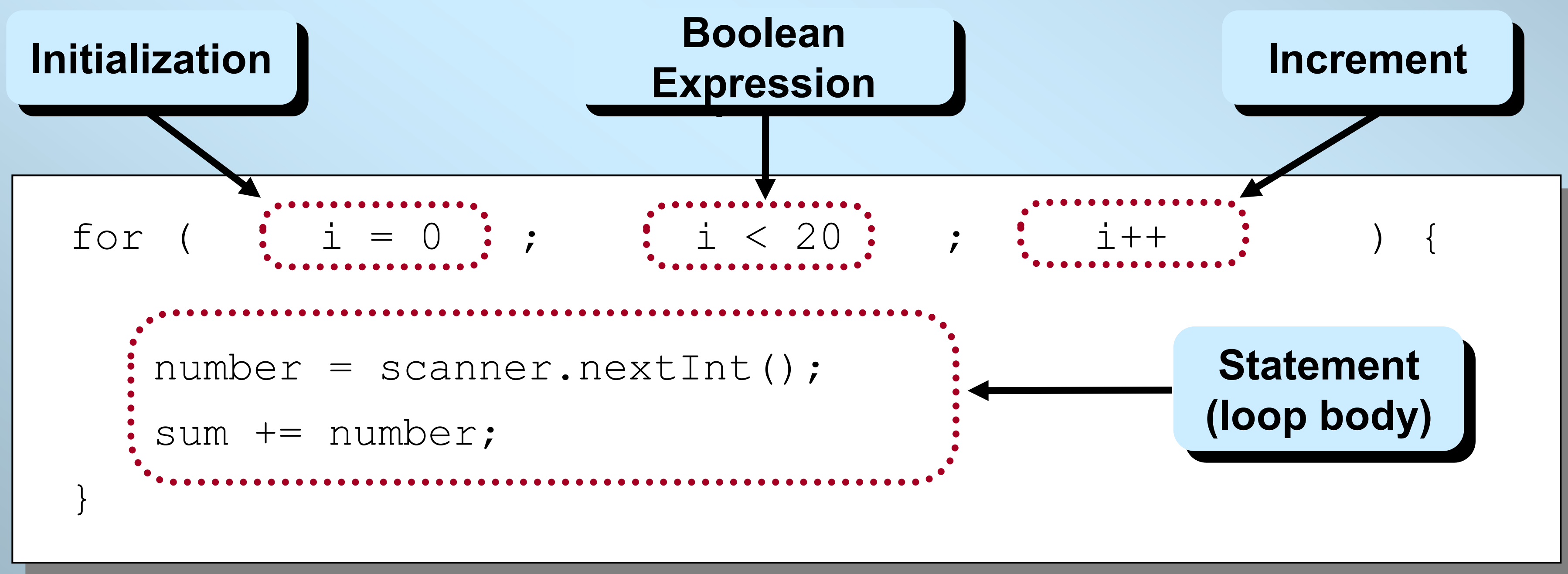
The for Statement

```
int i, sum = 0, number;

for (i = 0; i < 20; i++) {

    number = scanner.nextInt( );
    sum += number;

}
```

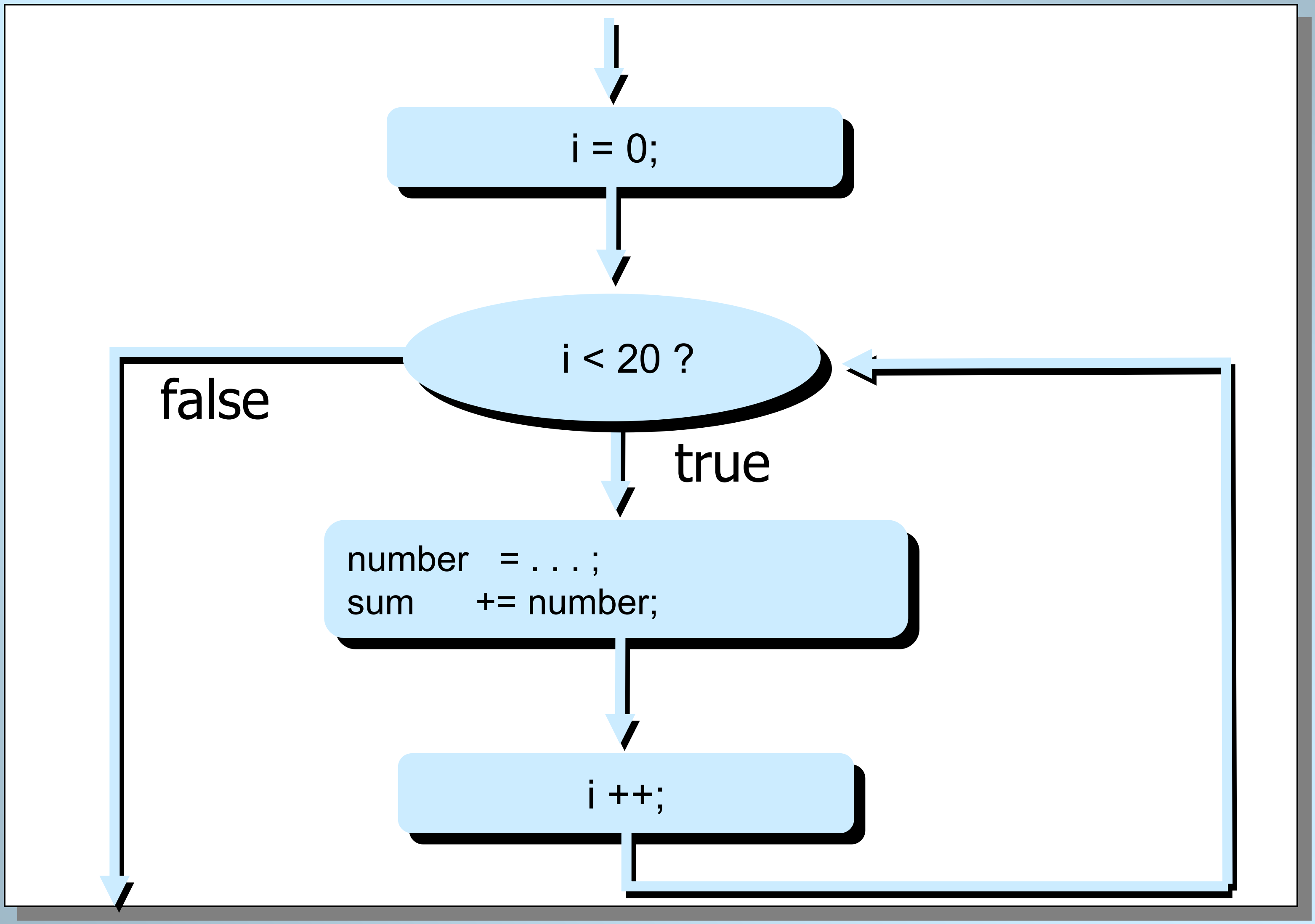
These statements are executed for 20 times (i = 0, 1, 2, … , 19).

Syntax for the for Statement

```
for ( <initialization>; <boolean expression>; <increment>  )

                    <statement>
```



Control Flow of for



More for Loop Examples

1

```
for (int i = 0; i < 100; i += 5)
```

i = 0, 5, 10, … , 95

2

```
for (int j = 2; j < 40; j *= 2)
```

j = 2, 4, 8, 16, 32

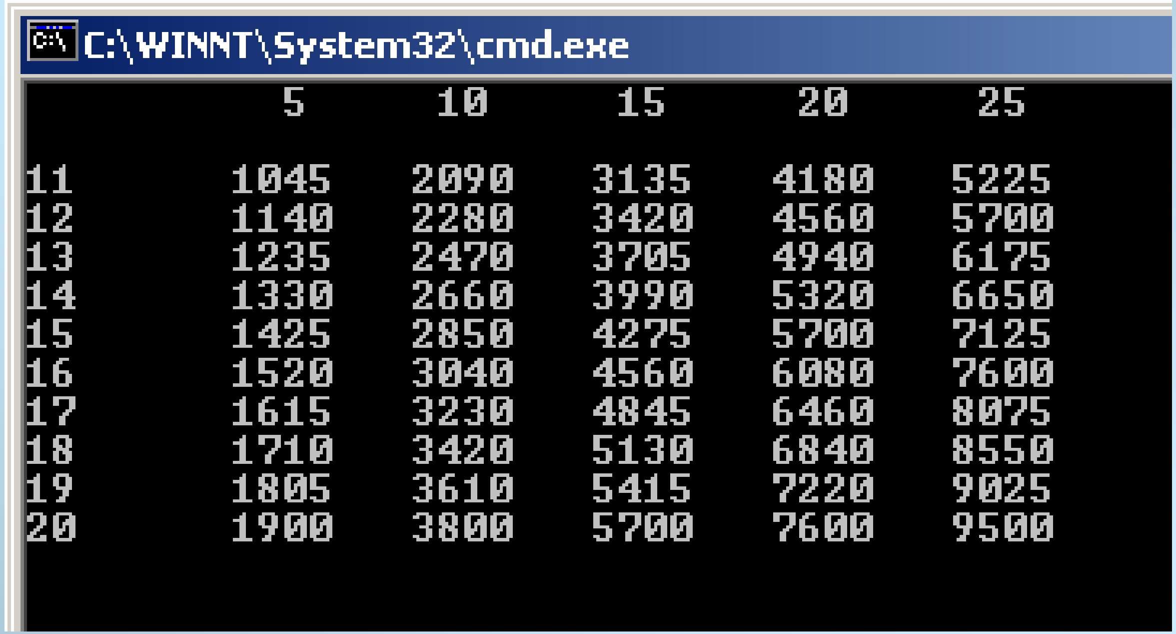
3

```
for (int k = 100; k > 0; k--) )
```

k = 100, 99, 98, 97, ..., 1

The Nested-for Statement

- Nesting a for statement inside another for statement is commonly used technique in programming.
- Let’s generate the following table using nested-for statement.



	5	10	15	20	25
11	1045	2090	3135	4180	5225
12	1140	2280	3420	4560	5700
13	1235	2470	3705	4940	6175
14	1330	2660	3990	5320	6650
15	1425	2850	4275	5700	7125
16	1520	3040	4560	6080	7600
17	1615	3230	4845	6460	8075
18	1710	3420	5130	6840	8550
19	1805	3610	5415	7220	9025
20	1900	3800	5700	7600	9500

Generating the Table

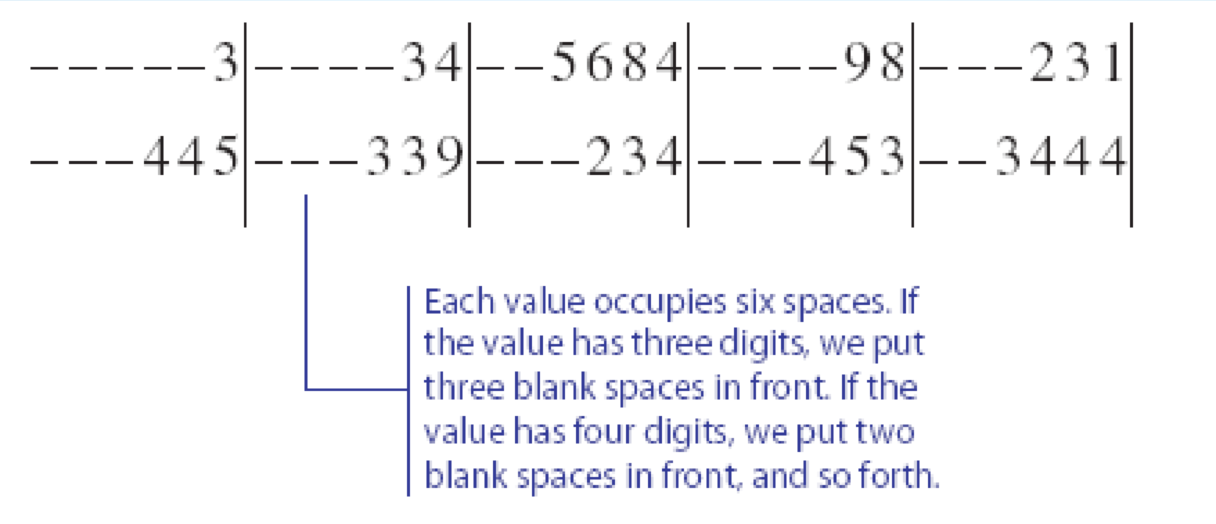
OUTER

INNER

```
int price;
for (int width = 11; width <=20, width++){

    for (int length = 5, length <=25, length+=5){

      price = width * length * 19; //$19 per sq. ft.
      System.out.print ("  " + price);
    }
    //finished one row; move on to next row
    System.out.println("");
}
```

Formatting Output

- We call the space occupied by an output value the *field*. The number of characters allocated to a field is the *field width*. The diagram shows the field width of 6.
- From Java 5.0, we can use the Formatter class. System.out (PrintStream) also includes the format method.



The Formatter Class

- We use the **Formatter** class to format the output.
- First we create an instance of the class

```
Formatter formatter = new Formatter(System.out);
```

- Then we call its format method

```
    int num = 467;
    formatter.format("%6d", num);
```

- This will output the value with the field width of 6.

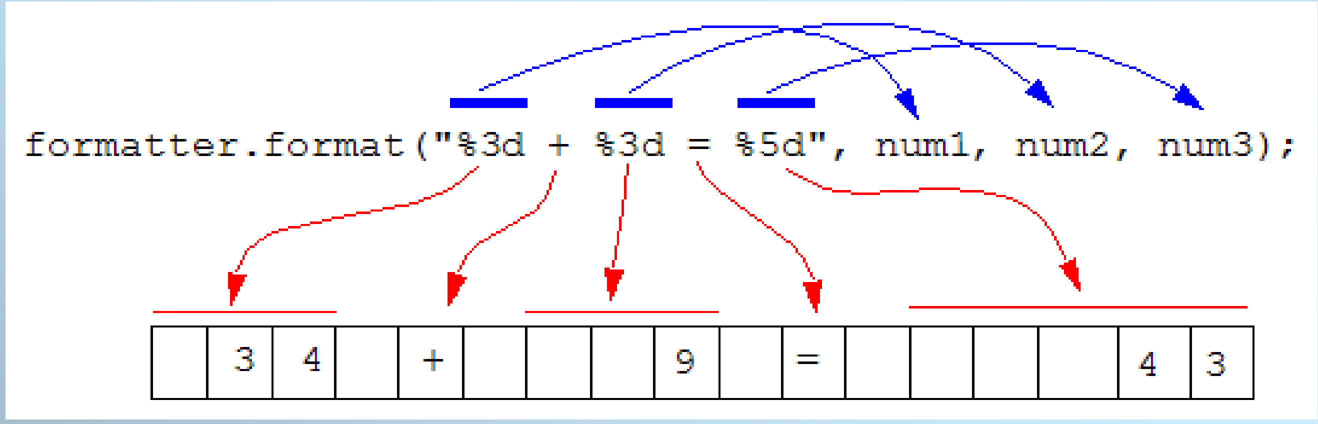
The format Method of Formatter

- The general syntax is

```
format(<control string>, <expr1>, <expr2>, . . . )
```

Example:

```
int num1 = 34, num2 = 9;
int num3 = num1 + num2;
formatter.format("%3d + %3d = %5d", num1, num2, num3);
```



The format Method of PrintStream

- Instead of using the Formatter class directly, we can achieve the same result by using the format method of PrintStream (System.out)

```
Formatter formatter = new Formatter(System.out);
formatter.format("%6d", 498);
```

is equivalent to

```
System.out.format("%6d", 498);
```

Control Strings

- Integers

  `% <field width> d`

- Real Numbers

  `% <field width> . <decimal places> f`

- Strings

  `% s`

- For other data types and more formatting options, please consult the Java API for the Formatter class.

Estimating the Execution Time

- In many situations, we would like to know how long it took to execute a piece of code. For example,
  - Execution time of a loop statement that finds the greatest common divisor of two very large numbers, or
  - Execution time of a loop statement to display all prime numbers between 1 and 100 million
- Execution time can be measured easily by using the Date class.

Using the Date Class

- Here's one way to measure the execution time

```
Date startTime = new Date();

//code you want to measure the execution time

Date endTime = new Date();

long elapsedTimeInMilliSec =
          endTime.getTime() - startTime.getTime();
```

Problem Statement

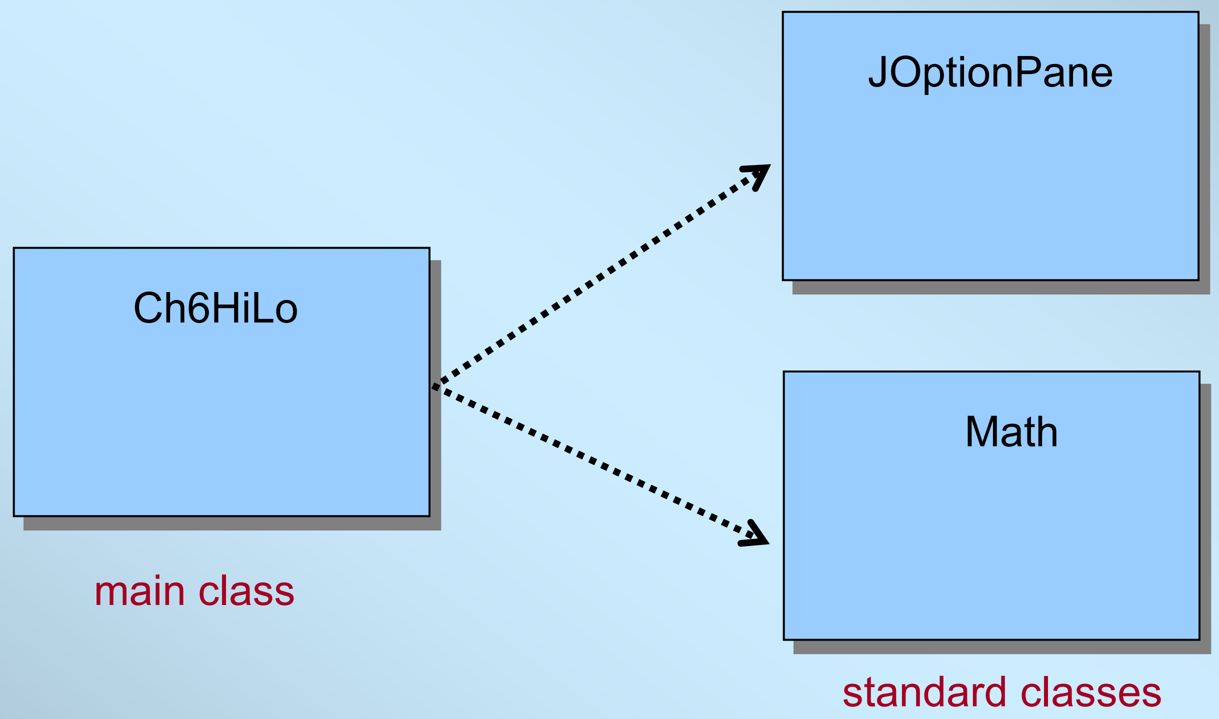
Write an application that will play Hi-Lo games with the user. The objective of the game is for the user to guess the computer-generated secret number in the least number of tries. The secret number is an integer between 1 and 100, inclusive. When the user makes a guess, the program replies with HI or LO depending on whether the guess is higher or lower than the secret number. The maximum number of tries allowed for each game is six. The user can play as many games as she wants.

Overall Plan

- Tasks:

```
do {

    Task 1: generate a secret number;

    Task 2: play one game;

} while ( the user wants to play );
```

Required Classes



Development Steps

- We will develop this program in four steps:

1. Start with a skeleton Ch6HiLo class.
2. Add code to the Ch6HiLo class to play a game using a dummy secret number.
3. Add code to the Ch6HiLo class to generate a random number.
4. Finalize the code by tying up loose ends.

Step 1 Design

- The topmost control logic of HiLo

```
1. describe the game rules;
2. prompt the user to play a game or not;

while ( answer is yes ) {

    3. generate the secret number;
    4. play one game;
    5. prompt the user to play another game or
          not;
}
```

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: Chapter6/Step1

Source Files: Ch6HiLo.java

Step 1 Test

- In the testing phase, we run the program and verify confirm that the topmost control loop terminates correctly under different conditions.
- Play the game
  - zero times
  - one time
  - one or more times

Step 2 Design

- Implement the playGame method that plays one game of HiLo.
- Use a dummy secret number
  - By using a fix number such as 45 as a dummy secret number, we will be able to test the correctness of the playGame method

The Logic of playGame

```
int guessCount = 0;
do {
    get next guess;

    guessCount++;

    if (guess < secretNumber) {
        print the hint LO;
    } else if (guess > secretNumber) {
        print the hint HI;
    }

} while (guessCount < number of guesses allowed
            && guess != secretNumber );

if (guess  == secretNumber) {
    print the winning message;
} else {
    print the losing message;
}
```

Step 2 Code

Directory: Chapter6/Step2

Source Files: Ch6HiLo.java

Step 2 Test

- We compile and run the program numerous times
- To test getNextGuess, enter
  - a number less than 1
  - a number greater than 100
  - a number between 2 and 99
  - the number 1 and the number 100
- To test playGame, enter
  - a guess less than 45
  - a guess greater than 45
  - 45
  - six wrong guesses

Step 3 Design

- We complete the generateSecretNumber method.
- We want to generate a number between 1 and 100 inclusively.

```
private void generateSecretNumber( ) {
    double X = Math.random();

    secretNumber = (int) Math.floor( X * 100 ) + 1;

    System.out.println("Secret Number: "
                                  + secretNumber);   // TEMP
    return secretNumber;
}
```

Step 3 Code

Directory: Chapter6/Step3

Source Files: Ch6HiLo.java

Step 3 Test

- We use a separate test driver to generate 1000 secret numbers.
- We run the program numerous times with different input values and check the results.
- Try both valid and invalid input values and confirm the response is appropriate

Step 4: Finalize

- Program Completion
  - Finish the describeRules method
  - Remove all temporary statements
- Possible Extensions
  - Allow the user to set her desired min and max for secret numbers
  - Allow the user to set the number of guesses allowed
  - Keep the score—the number of guesses made — while playing games and display the average score when the user quits the program

Chapter 5

Selection Statements

Animated Version

Objectives

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

- Implement a selection control using if statements
- Implement a selection control using switch statements
- Write boolean expressions using relational and boolean expressions
- Evaluate given boolean expressions correctly
- Nest an if statement inside another if statement
- Describe how objects are compared
- Choose the appropriate selection control statement for a given task

The if Statement

```
int testScore;

testScore = //get test score input

if (testScore < 70)

    JOptionPane.showMessageDialog(null,
                                  "You did not pass" );

else

    JOptionPane.showMessageDialog(null,
                                  "You did pass" );
```

This statement is executed if the testScore is less than 70.

This statement is executed if the testScore is 70 or higher.

Syntax for the if Statement

```
if ( <boolean expression> )

        <then block>
else

        <else block>
```

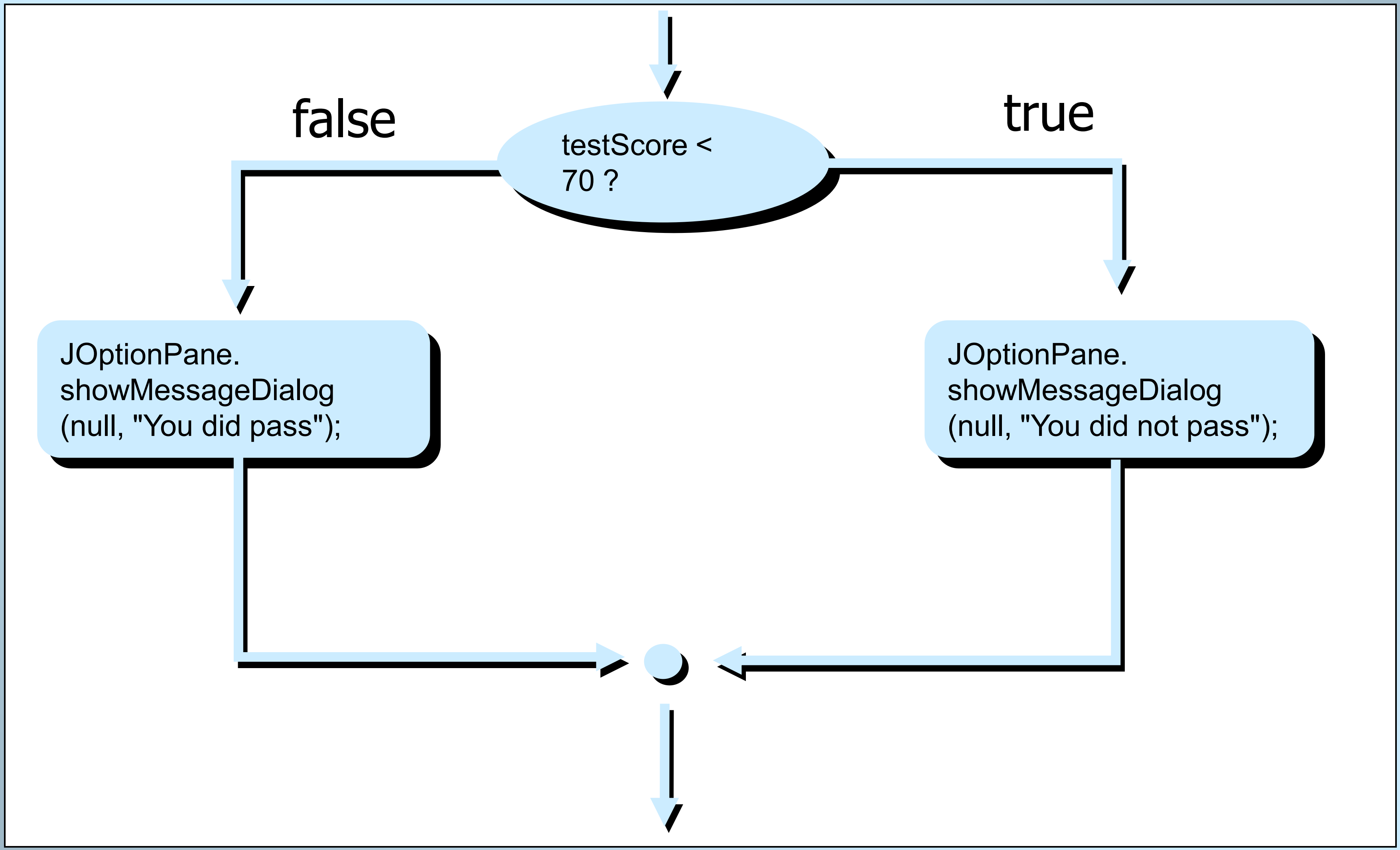
Boolean Expression

Then Block

Else Block

```
if (    testScore < 70    )

    JOptionPane.showMessageDialog(null,
                    "You did not pass" );

else

    JOptionPane.showMessageDialog(null,
                    "You did pass " );
```

Control Flow



Relational Operators

```
<        //less than
<=       //less than or equal to
==       //equal to
!=       //not equal to
>        //greater than
>=       //greater than or equal to
```

```
testScore < 80
testScore * 2 >= 350
30 < w / (h * h)
x + y != 2 * (a + b)
2 * Math.PI * radius <= 359.99
```

Compound Statements

- Use braces if the <then> or <else> block has multiple statements.

```
if (testScore < 70)
{
    JOptionPane.showMessageDialog(null,
                        "You did not pass" );
    JOptionPane.showMessageDialog(null,
                    "Try harder next time" );
}
else
{
    JOptionPane.showMessageDialog(null,
                      "You did pass" );
    JOptionPane.showMessageDialog(null,
                    "Keep up the good work" );
}
```

Then Block

Else Block

Style Guide

```
if ( <boolean expression> ) {
    …
}
else {
    …
}
```

Style 1

```
if ( <boolean expression> )
{
    …
}
else
{
    …
}
```

Style 2

The if-then Statement

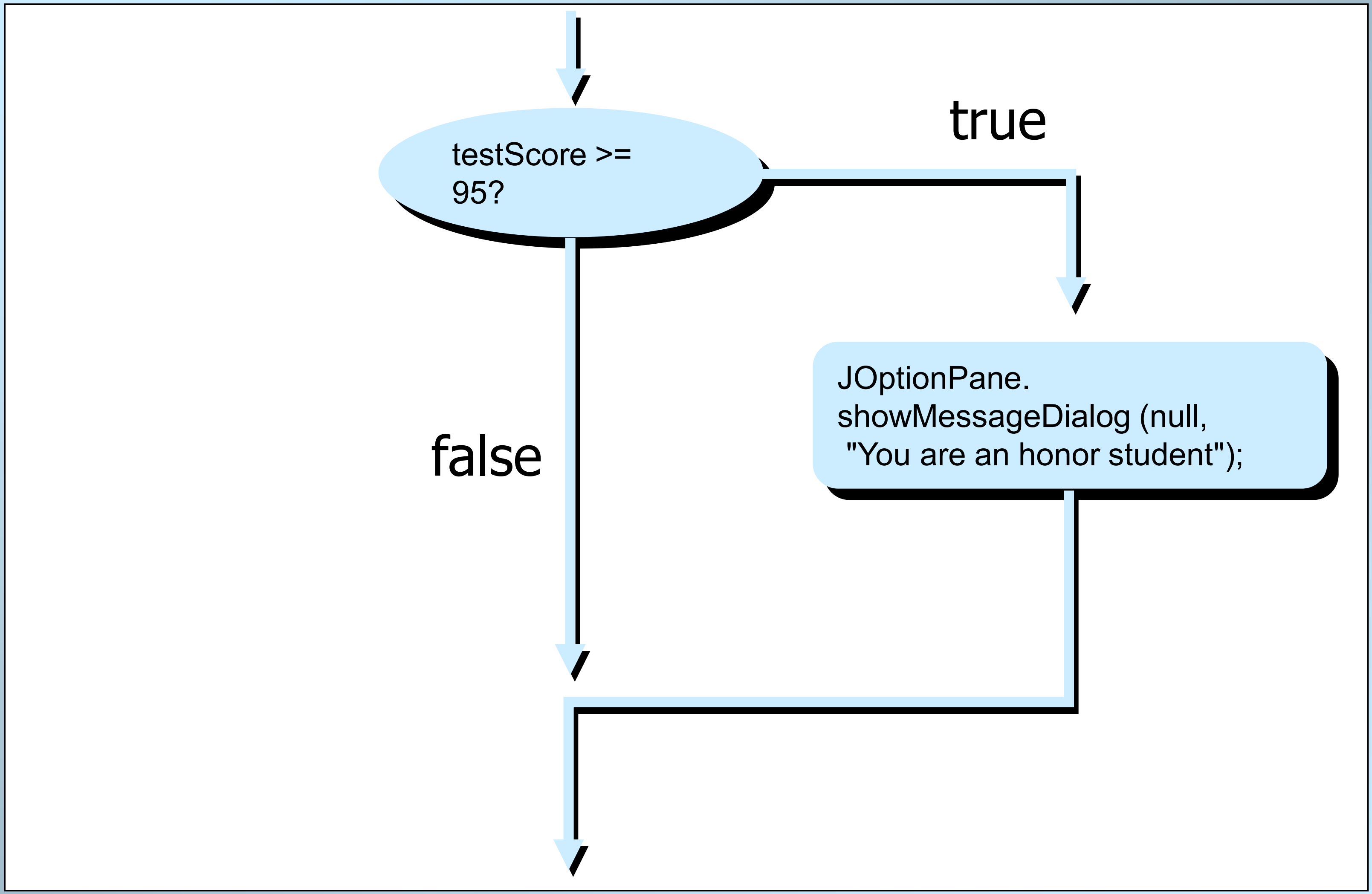
```
if ( <boolean expression> )

        <then block>
```

Boolean Expression

Then Block

```
if (     testScore >= 95     )

    JOptionPane.showMessageDialog(null,
                     "You are an honor student");
```

Control Flow of if-then

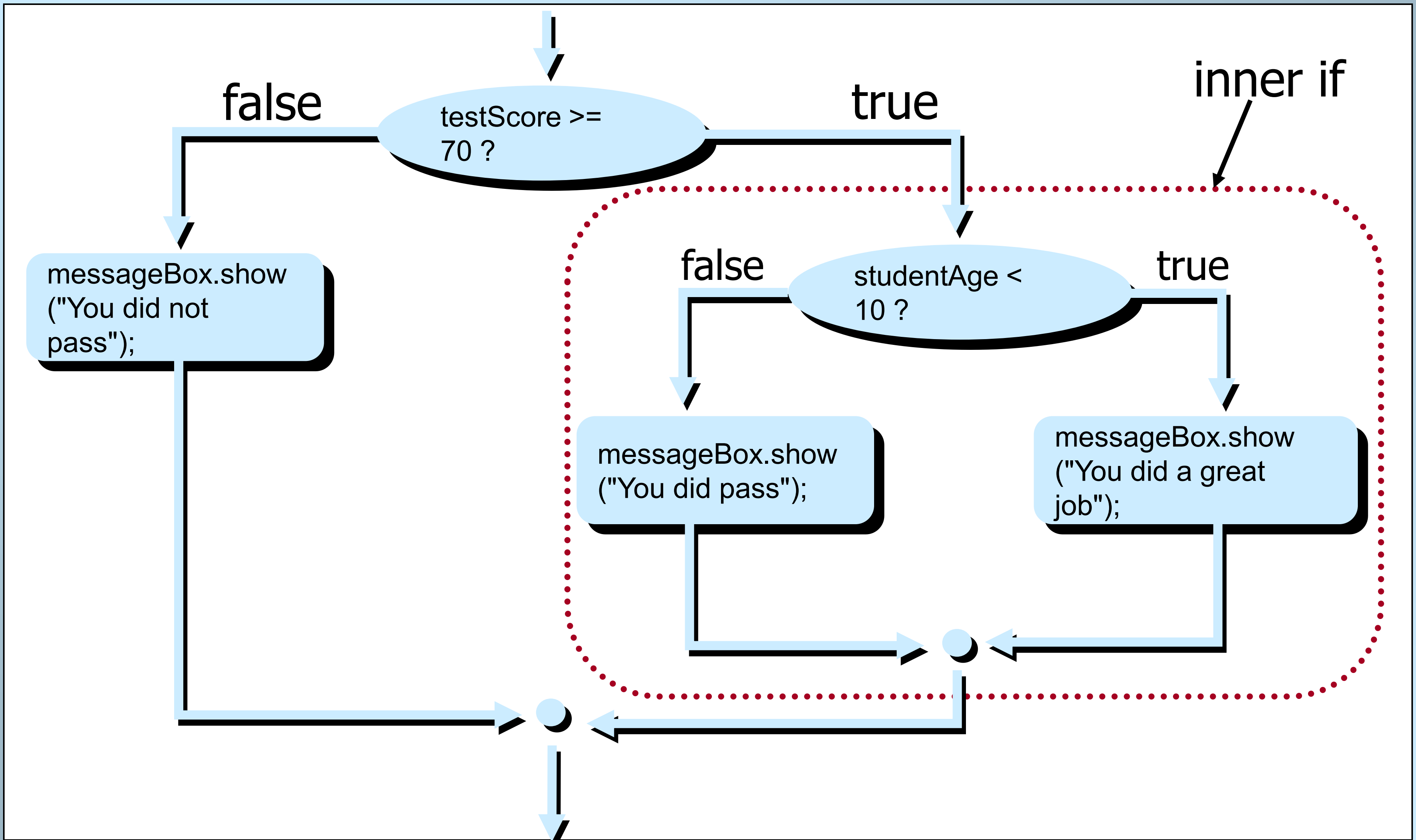


The Nested-if Statement

- The then and else block of an if statement can contain any valid statements, including other if statements. An if statement containing another if statement is called a nested-if statement.

```
if (testScore >= 70) {
     if (studentAge < 10) {
          System.out.println("You did a great job");
     } else {
          System.out.println("You did pass"); //test score >= 70
     }                                        //and age >= 10
} else { //test score < 70
      System.out.println("You did not pass");
}
```

Control Flow of Nested-if Statement



Writing a Proper if Control

```
if (num1 < 0)
    if (num2 < 0)
        if (num3 < 0)
            negativeCount = 3;
        else
            negativeCount = 2;
    else
        if (num3 < 0)
            negativeCount = 2;
        else
            negativeCount = 1;
else
    if (num2 < 0)
        if (num3 < 0)
            negativeCount = 2;
        else
            negativeCount = 1;
    else
        if (num3 < 0)
            negativeCount = 1;
        else
            negativeCount = 0;
```

```
negativeCount = 0;

if (num1 < 0)
        negativeCount++;
if (num2 < 0)
        negativeCount++;
if (num3 < 0)
        negativeCount++;
```

The statement

negativeCount++;

increments the variable by one

if – else if Control

Test Score	Grade
$90 \le$ score	A
$80 \le$ score < 90	B
$70 \le$ score < 80	C
$60 \le$ score < 70	D
score < 60	F

```
if (score >= 90)
   System.out.print("Your grade is A");

else if (score >= 80)
    System.out.print("Your grade is B");

else if (score >= 70)
    System.out.print("Your grade is C");

else if (score >= 60)
    System.out.print("Your grade is D");

else
    System.out.print("Your grade is F");
```

Matching else

Are A and B different?

A

```
if (x < y)
   if (x < z)
          System.out.print("Hello");
else
      System.out.print("Good bye");
```

B

```
if (x < y)
    if (x < z)
      System.out.print("Hello");
    else
      System.out.print("Good bye");
```

Both A and B means...

```
if (x < y) {
    if (x < z) {
       System.out.print("Hello");
    } else {
       System.out.print("Good bye");
    }
}
```

Boolean Operators

- A *boolean operator* takes boolean values as its operands and returns a boolean value.
- The three boolean operators are
  - and: &&
  - or: ||
  - not !

```
if (temperature >= 65 && distanceToDestination < 2) {
    System.out.println("Let's walk");
} else {
    System.out.println("Let's drive");
}
```

Semantics of Boolean Operators

- Boolean operators and their meanings:

P	Q	P && Q	P || Q	!P
false	false	false	false	true
false	true	false	true	true
true	false	false	true	false
true	true	true	true	false

De Morgan's Law

- De Morgan's Law allows us to rewrite boolean expressions in different ways

```
Rule 1: !(P && Q) ⟷ !P || !Q

Rule 2: !(P || Q) ⟷ !P && !Q
```

```
!(temp >= 65 && dist < 2)

    ⟷ !(temp >=65) || !(dist < 2) by Rule 1

    ⟷ (temp < 65 || dist >= 2)
```

Short-Circuit Evaluation

- Consider the following boolean expression:

```
x > y || x > z
```

- The expression is evaluated left to right. If x > y is true, then there's no need to evaluate x > z because the whole expression will be true whether x > z is true or not.
- To stop the evaluation once the result of the whole expression is known is called *short-circuit evaluation*.
- What would happen if the short-circuit evaluation is not done for the following expression?

```
z == 0 || x / z > 20
```

Operator Precedence Rules

Group	Operator	Precedence	Associativity
Subexpression	( )	10 (If parentheses are nested, then innermost subexpression is evaluated first.)	Left to right
Postfix increment and decrement operators	++ --	9	Right to left
Unary operators	- !	8	Right to left
Multiplicative operators	* / %	7	Left to right
Additive operators	+ -	6	Left to right
Relational operators	< <= > >=	5	Left to right
Equality operators	== !=	4	Left to right
Boolean AND	&&	3	Left to right
Boolean OR	||	2	Left to right
Assignment	=	1	Right to left

Boolean Variables

- The result of a boolean expression is either **true** or **false**. These are the two values of data type **boolean**.
- We can declare a variable of data type **boolean** and assign a boolean value to it.

```
boolean pass, done;
pass = 70 < x;
done = true;
if (pass) {
        …
} else {
        …
}
```

Boolean Methods

- A method that returns a boolean value, such as

```
private boolean  isValid(int value) {
     if (value < MAX_ALLOWED)
           return true;
     } else {
           return false;
     }
}
```

Can be used as

```
if (isValid(30)) {
     …
} else {
     …
}
```

Comparing Objects

- With primitive data types, we have only one way to compare them, but with objects (reference data type), we have two ways to compare them.
    1. We can test whether two variables point to the same object (use ==), or
    2. We can test whether two distinct objects have the same contents.

Using == With Objects (Sample 1)

```
String str1 = new String("Java");
String str2 = new String("Java");

if (str1 == str2) {
    System.out.println("They are equal");
} else {
    System.out.println("They are not equal");
}
```

```
They are not equal
```

Not equal because str1 and str2 point to different String objects.

Using == With Objects (Sample 2)

```
String str1 = new String("Java");
String str2 = str1;

if (str1 == str2) {
    System.out.println("They are equal");
} else {
    System.out.println("They are not equal");
}
```

```
They are equal
```

It's equal here because str1 and str2 point to the same object.

Using equals with String

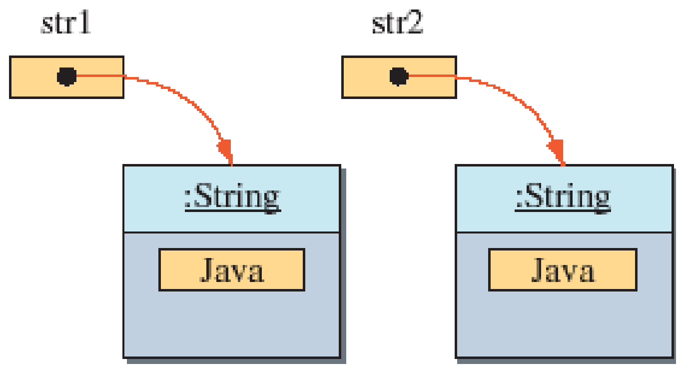
```
String str1 = new String("Java");
String str2 = new String("Java");

if (str1.equals(str2)) {
    System.out.println("They are equal");
} else {
    System.out.println("They are not equal");
}
```

```
They are equal
```

It's equal here because str1 and str2 have the same sequence of characters.

The Semantics of ==

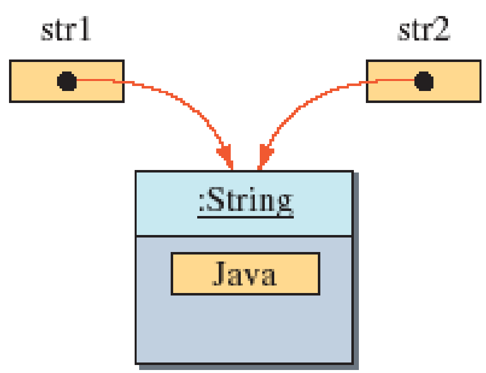
Case A: Two variables refer to two different objects.




```
String str1, str2;

str1 = new String("Java");
str2 = new String("Java");
```

str1 == str2 ⟶ false

Case B: Two variables refer to the same object.



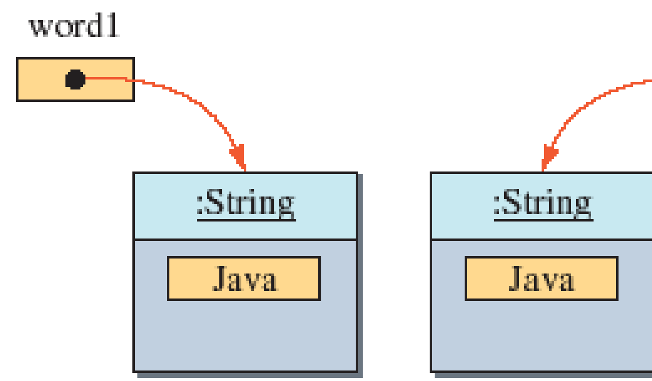

```
String str1, str2;

str1 = new String("Java");
str2 = str1;
```

str1 == str2 ⟶ true

In Creating String Objects

```
String word1, word2;

word1 = new String("Java");

word2 = new String("Java");
```

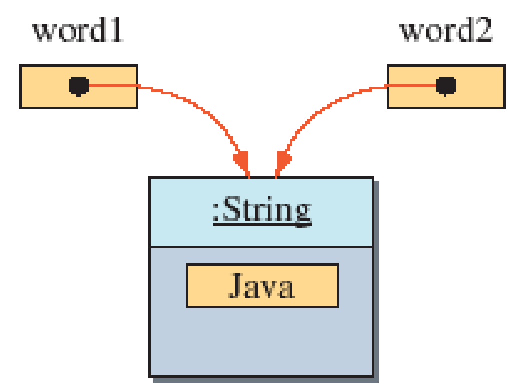
Whenever the new operator is used, there will be a new object.



word1 == word2 → false

```
String word1, word2;

word1 = "Java";

word2 = "Java";
```

Literal string constant such as “Java” will always refer to one object.



word1 == word2 → true

The switch Statement

```
int gradeLevel;
gradeLevel = JOptionPane.showInputDialog("Grade (Frosh-1,Soph-2,...):" );

switch (gradeLevel) {

    case 1: System.out.print("Go to the Gymnasium");
            break;

    case 2: System.out.print("Go to the Science Auditorium");
            break;

    case 3: System.out.print("Go to Harris Hall Rm A3");
            break;

    case 4: System.out.print("Go to Bolt Hall Rm 101");
            break;
}
```

This statement is executed if the gradeLevel is equal to 1.

This statement is executed if the gradeLevel is equal to 4.

Syntax for the switch Statement

```
switch ( <arithmetic expression> ) {
        <case label 1> : <case body 1>
        …
        <case label n> : <case body n>
}
```

```
switch (  gradeLevel  ) {
    case 1: System.out.print( "Go to the Gymnasium" );
            break;
    case 2: System.out.print( "Go to the Science Auditorium" );
            break;
    case 3: System.out.print( "Go to Harris Hall Rm A3" );
            break;
    case 4: System.out.print( "Go to Bolt Hall Rm 101" );
            break;
}
```

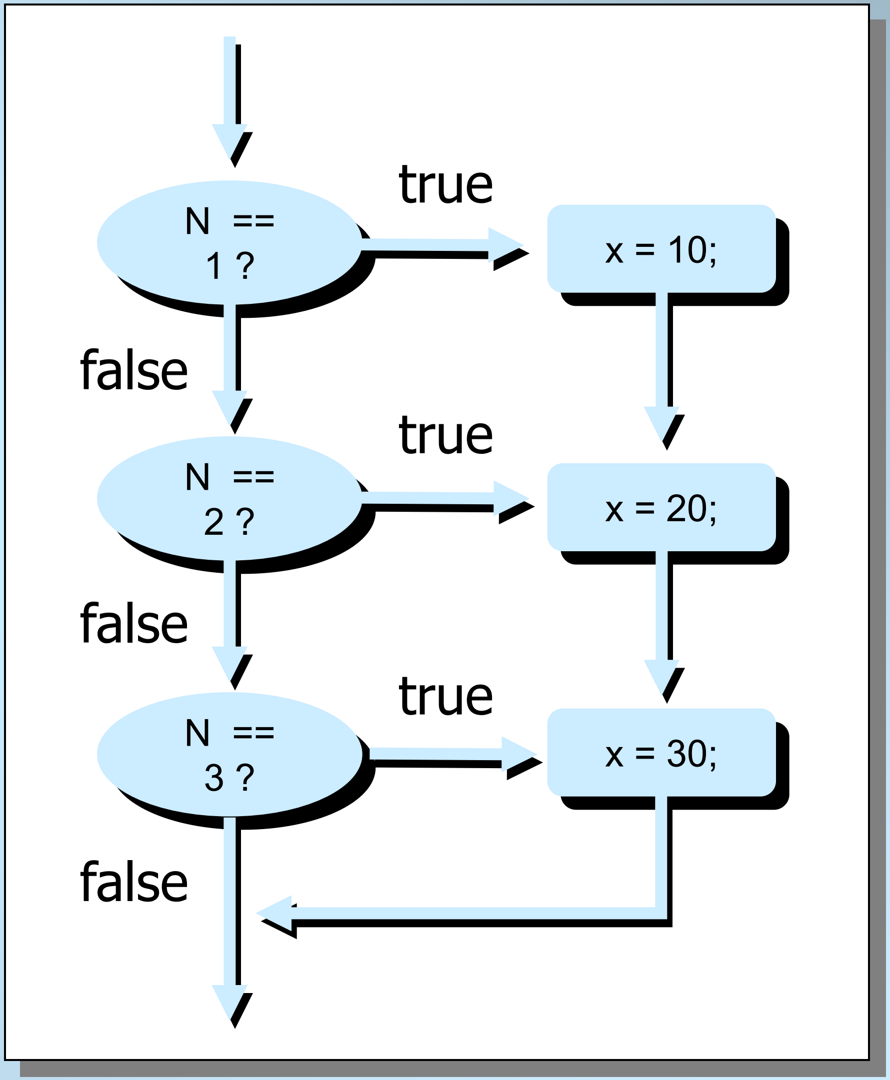
Arithmetic Expression

Case Label

Case Body

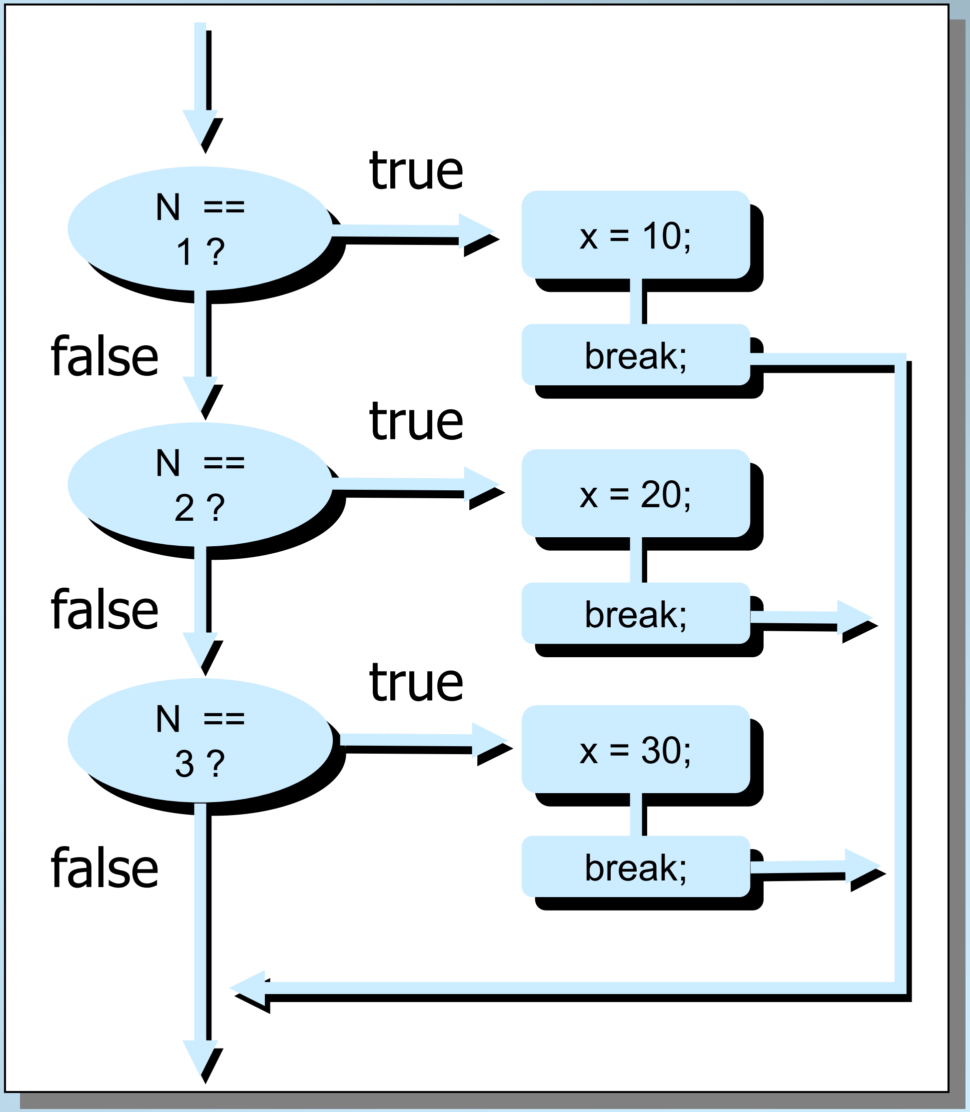
switch With No break Statements

```
switch ( N ) {
   case 1: x = 10;
   case 2: x = 20;
   case 3: x = 30;
}
```



switch With break Statements

```
switch ( N ) {
    case 1: x = 10;
            break;
    case 2: x = 20;
            break;
    case 3: x = 30;
            break;
}
```



switch With the default Block

```
switch (ranking) {

    case 10:
    case  9:
    case  8: System.out.print("Master");
              break;

    case  7:
    case  6: System.out.print("Journeyman");
              break;

    case  5:
    case  4: System.out.print("Apprentice");
              break;

    default: System.out.print("Input error: Invalid Data");
              break;
}
```

Drawing Graphics

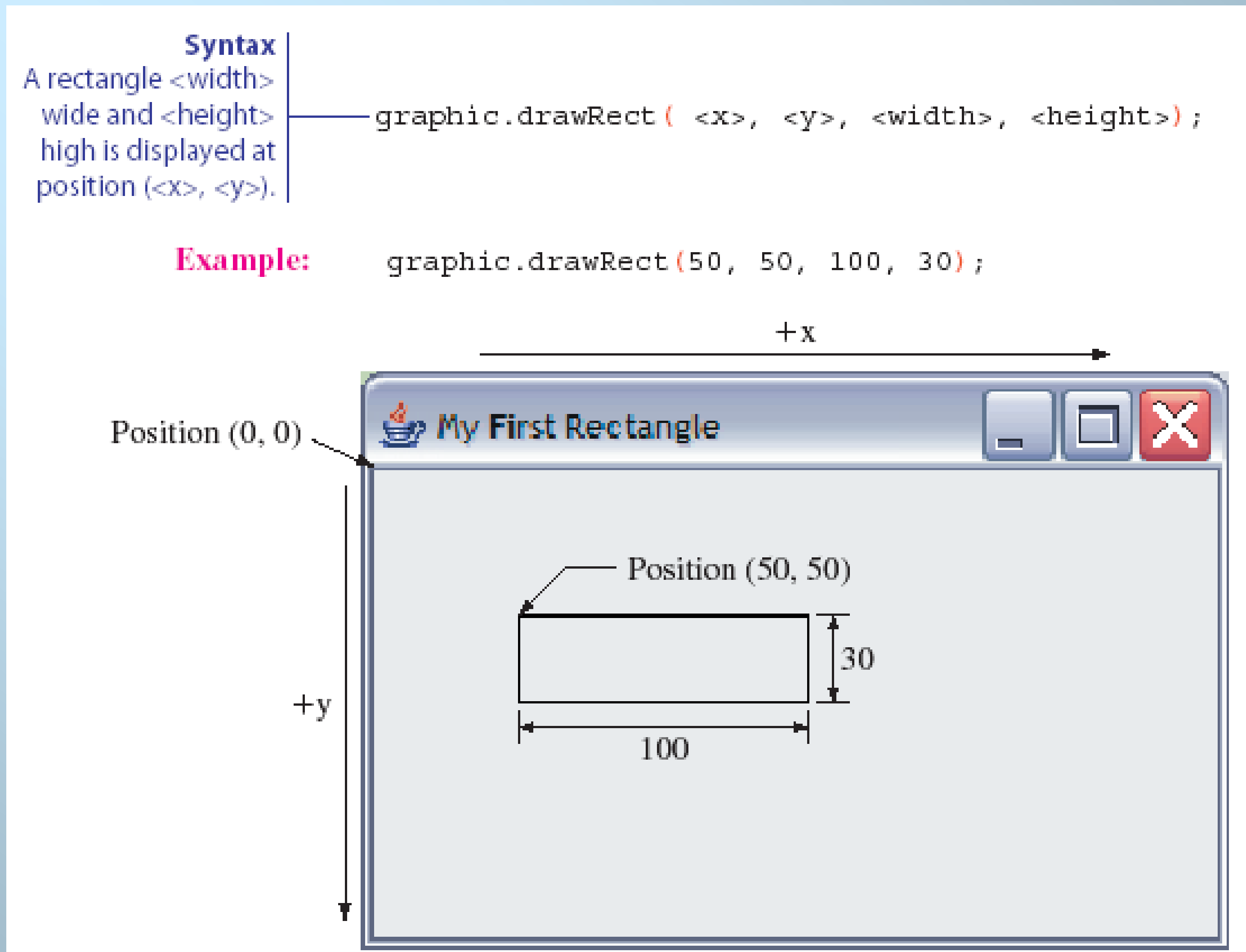
- Chapter 5 introduces four standard classes related to drawing geometric shapes. They are
  - java.awt.Graphics
  - java.awt.Color
  - java.awt.Point
  - java.awt.Dimension
- These classes are used in the Sample Development section
- Please refer to Java API for details

Sample Drawing

```
import javax.swing.*; //for JFrame
import java.awt.*; //for Graphics and Container

class Ch5SampleGraphics {

    public static void main( String[] args ) {

        JFrame    win;
        Container contentPane;
        Graphics  g;

        win = new JFrame("My First Rectangle");
        win.setSize(300, 200);
        win.setLocation(100,100);
        win.setVisible(true);

        contentPane = win.getContentPane();
        g = contentPane.getGraphics();
        g.drawRect(50,50,100,30);
    }
}
```

win must be visible on the screen before you get its content pane.

The Effect of drawRect



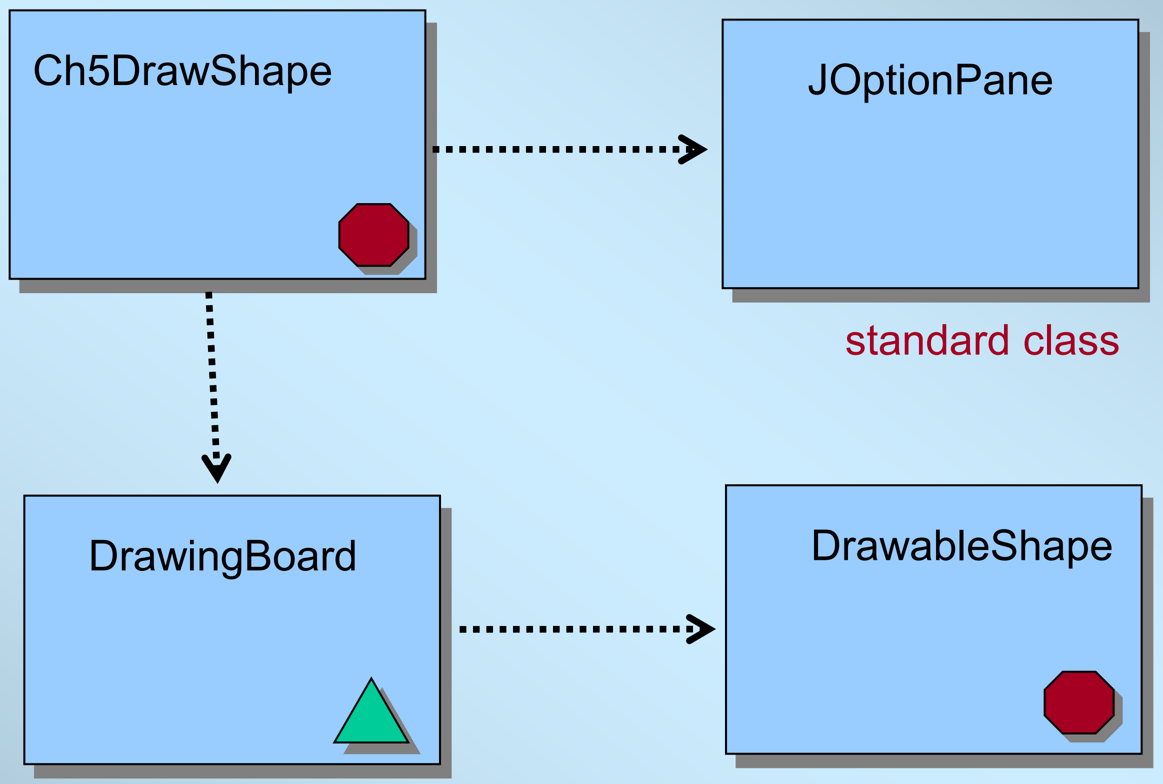
Problem Statement

Write an application that simulates a screensaver by drawing various geometric shapes in different colors. The user has an option of choosing a type (ellipse or rectangle), color, and movement (stationary, smooth, or random).

Overall Plan

- Tasks:
  - Get the shape the user wants to draw.
  - Get the color of the chosen shape.
  - Get the type of movement the user wants to use.
  - Start the drawing.

Required Classes



class we implement

helper class given to us

Development Steps

- We will develop this program in six steps:

1. Start with a program skeleton. Explore the DrawingBoard class.
2. Define an experimental DrawableShape class that draws a dummy shape.
3. Add code to allow the user to select a shape. Extend the DrawableShape and other classes as necessary.
4. Add code to allow the user to specify the color. Extend the DrawableShape and other classes as necessary.
5. Add code to allow the user to specify the motion type. Extend the DrawableShape and other classes as necessary.
6. Finalize the code by tying up loose ends.

Step 1 Design

- The methods of the DrawingBoard class
  - `public void addShape(DrawableShape shape)`

    Adds a shape to the DrawingBoard. No limit to the number shapes you can add
  - `public void setBackground(java.awt.Color color)`

    Sets the background color of a window to the designated color
  - `public void setDelayTime(double delay)`

    Sets the delay time between drawings to delay seconds
  - `public void setMovement(int type)`

    Sets the movement type to STATIONARY, RANDOM, or SMOOTH
  - `public void setVisible(boolean state)`

    Sets the background color of a window to the designated color
  - `public void start( )`

    Starts the drawing of added shapes using the designated movement type and delay time.

Step 1 Code

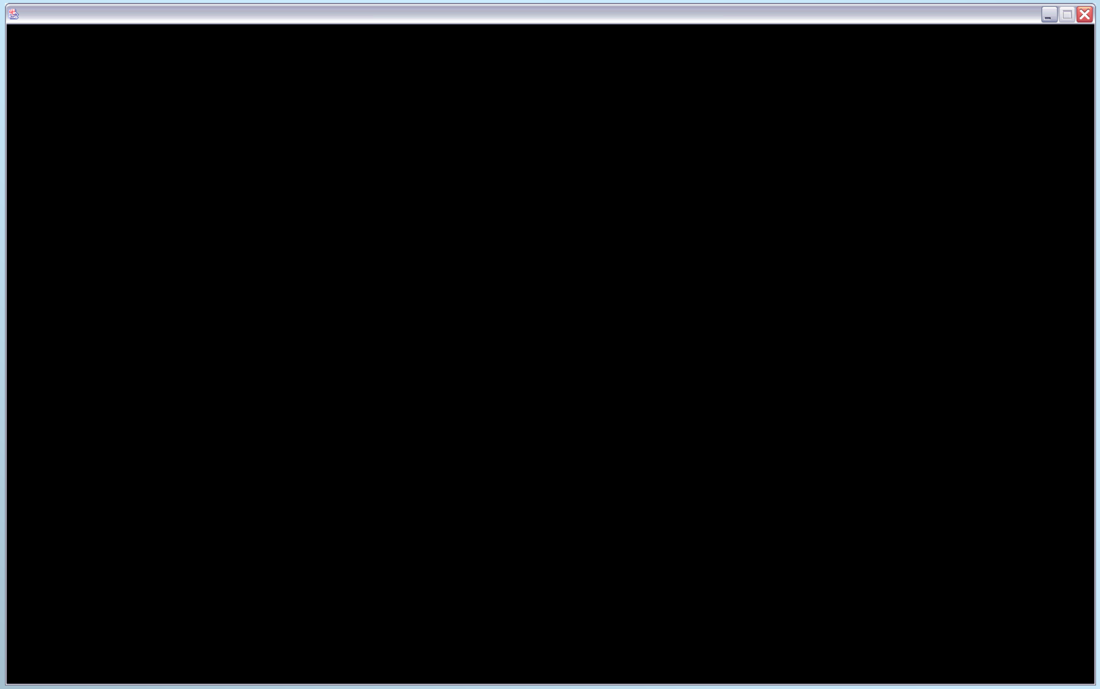
Program source file is too big to list here. From now on, we ask you to view the source files using your Java IDE.

Directory: Chapter5/Step1

Source Files: Ch5DrawShape.java

Step 1 Test

- In the testing phase, we run the program and verify that a DrawingBoard window with black background appears on the screen and fills the whole screen.



Step 2 Design

- Define a preliminary DrawableShape class

- The required methods of this class are

  - `public void draw(java.awt.Graphics g)`

    Draws a shape on Graphics object g.
  - `public java.awt.Point getCenterPoint( )`

    Returns the center point of this shape
  - `public java.awt.Dimension getDimension( )`

    Returns the bounding rectangle of this shape
  - `public void setCenterPoint(java.awt.Point pt)`

    Sets the center point of this shape to pt.

Step 2 Code

Directory: Chapter5/Step2

Source Files: Ch5DrawShape.java
DrawableShape.java

Step 2 Test

- We compile and run the program numerous times
- We confirm the movement types STATIONARY, RANDOM, and SMOOTH.
- We experiment with different delay times
- We try out different background colors

Step 3 Design

- We extend the main class to allow the user to select a shape information.
- We will give three choices of shapes to the user: Ellipse, Rectangle, and Rounded Rectangle
- We also need input routines for the user to enter the dimension and center point. The center point determines where the shape will appear on the DrawingBoard.
- Three input methods are

```
private int            inputShapeType( )
private Dimension      inputDimension( )
private Point          inputCenterPoint( )
```

Step 3 Code

Directory: Chapter5/Step3

Source Files: Ch5DrawShape.java
DrawableShape.java

Step 3 Test

- We run the program numerous times with different input values and check the results.
- Try both valid and invalid input values and confirm the response is appropriate

Step 4 Design

- We extend the main class to allow the user to select a color.
- We follow the input pattern of Step 3.
- We will allow the user to select one of the five colors.
- The color input method is

```
private Color     inputColor( )
```

Step 4 Code

Directory: Chapter5/Step4

Source Files: Ch5DrawShape.java
DrawableShape.java

Step 4 Test

- We run the program numerous times with different color input.
- Try both valid and invalid input values and confirm the response is appropriate

Step 5 Design

- We extend the main class to allow the user to select a movement type.
- We follow the input pattern of Step 3.
- We will allow the user to select one of the three movement types.
- The movement input method is

```
private int     inputMotionType( )
```

Step 5 Code

Directory: Chapter5/Step5

Source Files: Ch5DrawShape.java
DrawableShape.java

Step 5 Test

- We run the program numerous times with different movement input.
- Try both valid and invalid input values and confirm the response is appropriate

Step 6: Finalize

- Possible Extensions
  – Morphing the object shape
  – Changing the object color
  – Drawing multiple objects
  – Drawing scrolling text