

Chapter 8 – Designing Classes

Chapter Goals

- To learn how to discover appropriate classes for a given problem
- To understand the concepts of cohesion and coupling
- To minimize the use of side effects
- To document the responsibilities of methods and their callers with preconditions and postconditions
- To understand static methods and variables
- To understand the scope rules for local variables and instance variables
- To learn about packages
- T** To learn about unit testing frameworks

Discovering Classes

- A class represents a single concept from the problem domain
- Name for a class should be a noun that describes concept
- Concepts from mathematics:

Point
Rectangle
Ellipse

- Concepts from real life:

BankAccount
CashRegister

Discovering Classes

- Actors (end in -er, -or) – objects do some kinds of work for you:

`Scanner`

`Random // better name: RandomNumberGenerator`

- Utility classes – no objects, only static methods and constants:

`Math`

- Program starters: only have a `main` method

- Don't turn actions into classes

- *Paycheck is a better name than ComputePaycheck*

Cohesion

- A class should represent a single concept
- The public interface of a class is *cohesive* if all of its features are related to the concept that the class represents
- This class lacks cohesion:

```
public class CashRegister
{
    public void enterPayment(int dollars, int quarters,
        int dimes, int nickels, int pennies)
        ...
    public static final double NICKEL_VALUE = 0.05;
    public static final double DIME_VALUE = 0.1;
    public static final double QUARTER_VALUE = 0.25;
    ...
}
```

Cohesion

- `CashRegister`, as described above, involves two concepts:
cash register and *coin*
- Solution: Make two classes:

```
public class Coin
{
    public Coin(double aValue, String aName) { ... }
    public double getValue() { ... }
    ...
}

public class CashRegister
{
    public void enterPayment(int coinCount, Coin coinType)
        { ... }
    ...
}
```

Coupling

- A class *depends* on another if it uses objects of that class
- `CashRegister` depends on `Coin` to determine the value of the payment
- `Coin` does not depend on `CashRegister`
- High coupling = Many class dependencies
- Minimize coupling to minimize the impact of interface changes
- To visualize relationships draw class diagrams
- UML: Unified Modeling Language
 - *Notation for object-oriented analysis and design*

Dependency

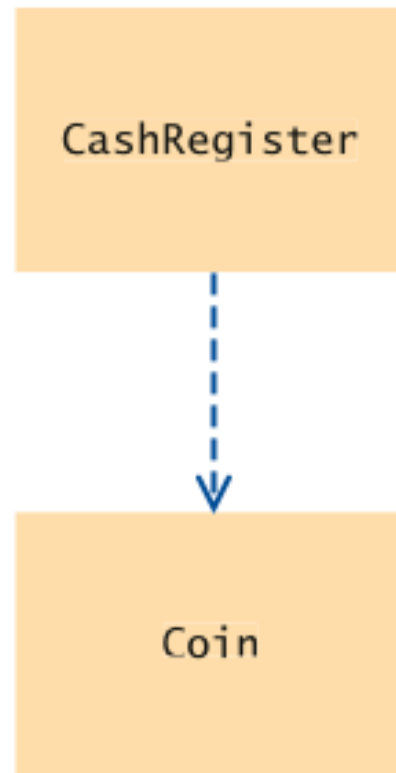


Figure 1
Dependency Relationship
Between the CashRegister
and Coin Classes

High and Low Coupling Between Classes

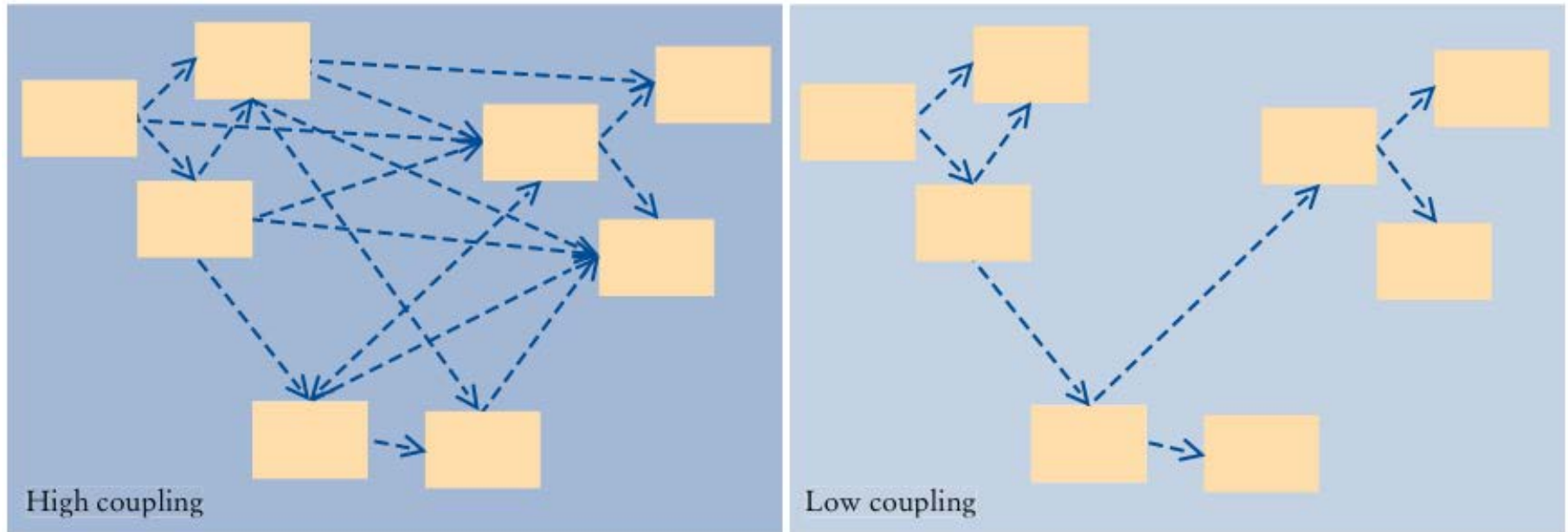


Figure 2 High and Low Coupling Between Classes

Immutable Classes

- **Accessor:** Does not change the state of the implicit parameter:

```
double balance = account.getBalance();
```

- **Mutator:** Modifies the object on which it is invoked:

```
account.deposit(1000);
```

- **Immutable class:** Has no mutator methods (e.g., `String`):

```
String name = "John Q. Public";  
String uppercased = name.toUpperCase();  
// name is not changed
```

- It is safe to give out references to objects of immutable classes; no code can modify the object at an unexpected time

Immutable Objects and Classes

If the contents of an object cannot be changed once the object is created, the object is called an *immutable object* and its class is called an *immutable class*. If you delete the set method in the Circle class in Listing 8.10, the class would be immutable because radius is private and cannot be changed without a set method.

A class with all private data fields and without mutators is not necessarily immutable. For example, the following class Student has all private data fields and no mutators, but it is mutable.

Example

```
public class Student {
    private int id;
    private BirthDate birthDate;

    public Student(int ssn,
        int year, int month, int day) {
        id = ssn;
        birthDate = new BirthDate(year, month, day);
    }

    public int getId() {
        return id;
    }

    public BirthDate getBirthDate() {
        return birthDate;
    }
}
```

```
public class BirthDate {
    private int year;
    private int month;
    private int day;

    public BirthDate(int newYear,
        int newMonth, int newDay) {
        year = newYear;
        month = newMonth;
        day = newDay;
    }

    public void setYear(int newYear) {
        year = newYear;
    }
}
```

```
public class Test {
    public static void main(String[] args) {
        Student student = new Student(111223333, 1970, 5, 3);
        BirthDate date = student.getBirthDate();
        date.setYear(2010); // Now the student birth year is changed!
    }
}
```

What Class is Immutable?

For a class to be immutable, it must mark all data fields private and provide no mutator methods and no accessor methods that would return a reference to a mutable data field object.

Side Effects

- **Side effect of a method:** Any externally observable data modification:

```
harrysChecking.deposit(1000);
```

- Modifying explicit parameter can be surprising to programmers— avoid it if possible:

```
public void addStudents(ArrayList<String> studentNames)
{
    while (studentNames.size() > 0)
    {
        String name = studentNames.remove(0);
        // Not recommended
        . . .
    }
}
```

Side Effects

- This method has the expected side effect of modifying the implicit parameter and the explicit parameter `other`:

```
public void transfer(double amount, BankAccount other
{
    balance = balance - amount;
    other.balance = other.balance + amount;
}
```

Side Effects

- Another example of a side effect is output:

```
public void printBalance() // Not recommended
{
    System.out.println("The balance is now $"
        + balance);
}
```

Bad idea: Message is in English, and relies on `System.out`

- Decouple input/output from the actual work of your classes
- Minimize side effects that go beyond modification of the implicit parameter

Common Error: Trying to Modify Primitive Type Parameters

- ```
void transfer(double amount, double otherBalance)
{
 balance = balance - amount;
 otherBalance = otherBalance + amount;
}
```

- Won't work

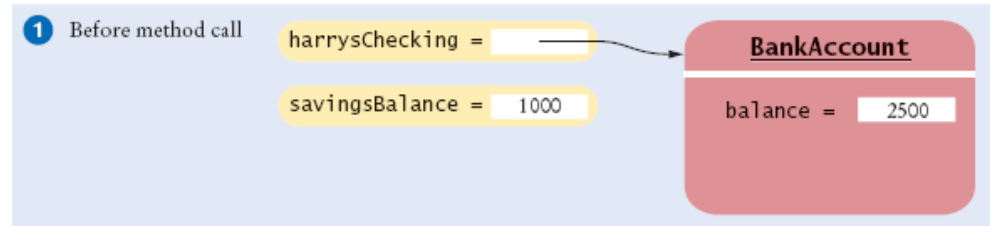
- Scenario:

```
double savingsBalance = 1000;
harrysChecking.transfer(500, savingsBalance);
System.out.println(savingsBalance);
```

- In Java, a method can never change parameters of primitive type

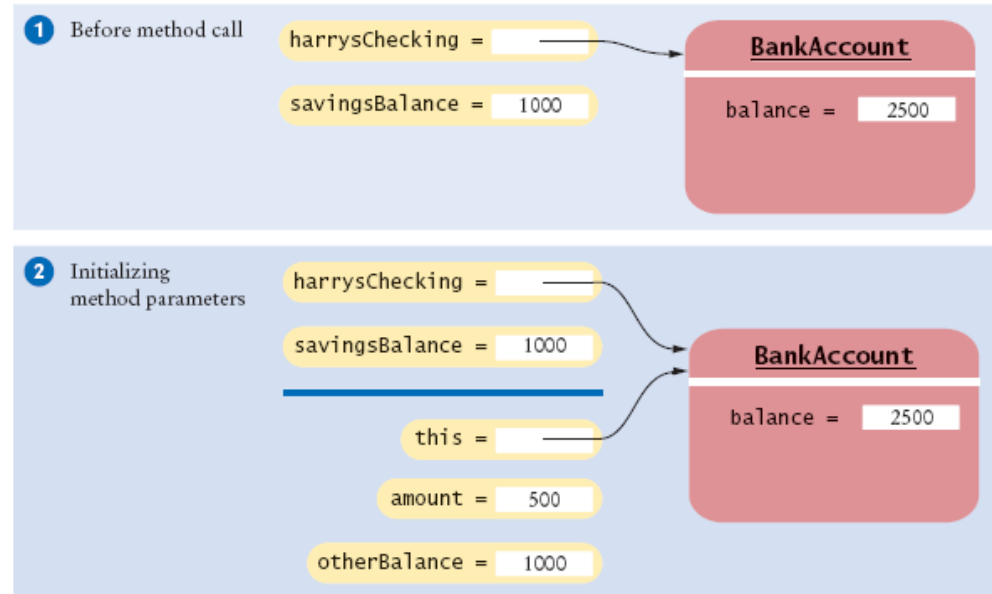
# Common Error: Trying to Modify Primitive Type Parameters

```
double savingsBalance = 1000;
harrysChecking.transfer(500, savingsBalance); ❶
System.out.println(savingsBalance);
...
void transfer(double amount, double otherBalance)
{
 balance = balance - amount;
 otherBalance = otherBalance + amount;
}
```



# Common Error: Trying to Modify Primitive Type Parameters

```
double savingsBalance = 1000;
harrysChecking.transfer(500, savingsBalance); ❶
System.out.println(savingsBalance);
...
void transfer(double amount, double otherBalance) ❷
{
 balance = balance - amount;
 otherBalance = otherBalance + amount;
}
```

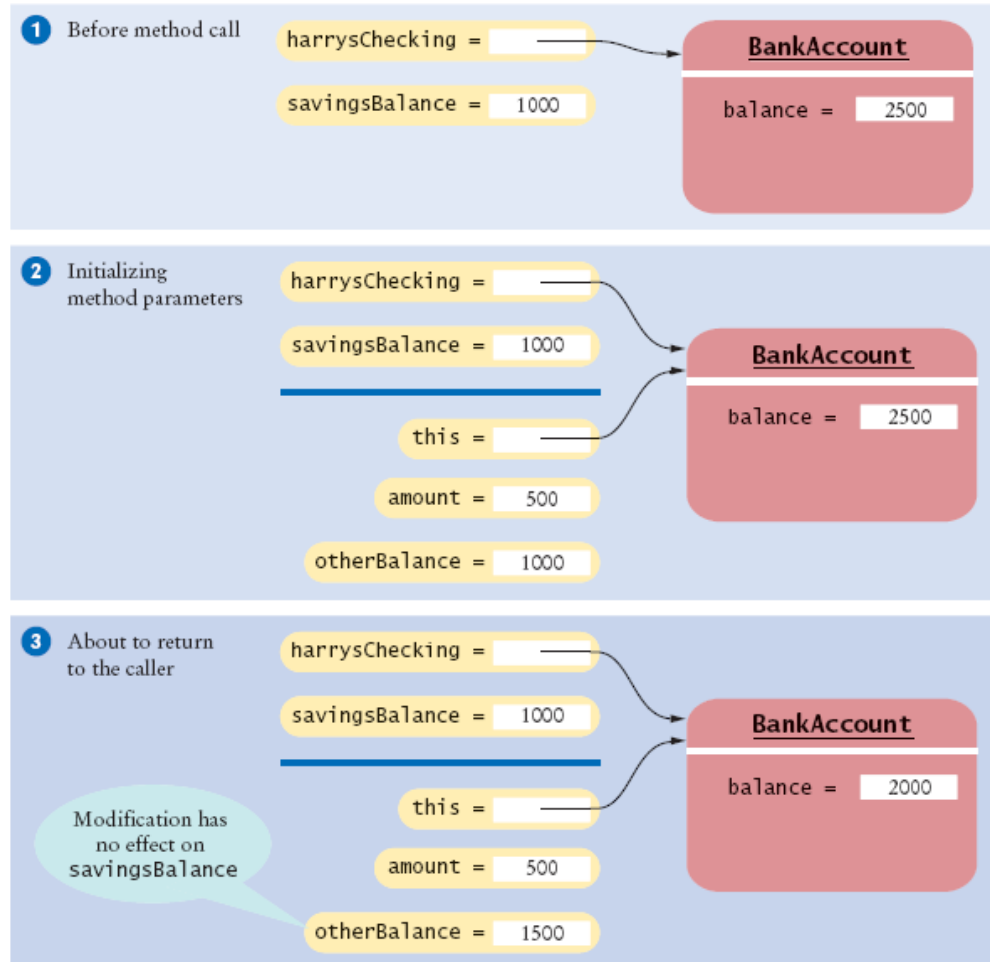


# Common Error: Trying to Modify Primitive Type Parameters

```
double savingsBalance = 1000;
harrysChecking.transfer(500, savingsBalance); ❶
System.out.println(savingsBalance);
...
void transfer(double amount, double otherBalance) ❷
{
 balance = balance - amount;
 otherBalance = otherBalance + amount;
} ❸
```

***Continued***

# Common Error: Trying to Modify Primitive Type Parameters

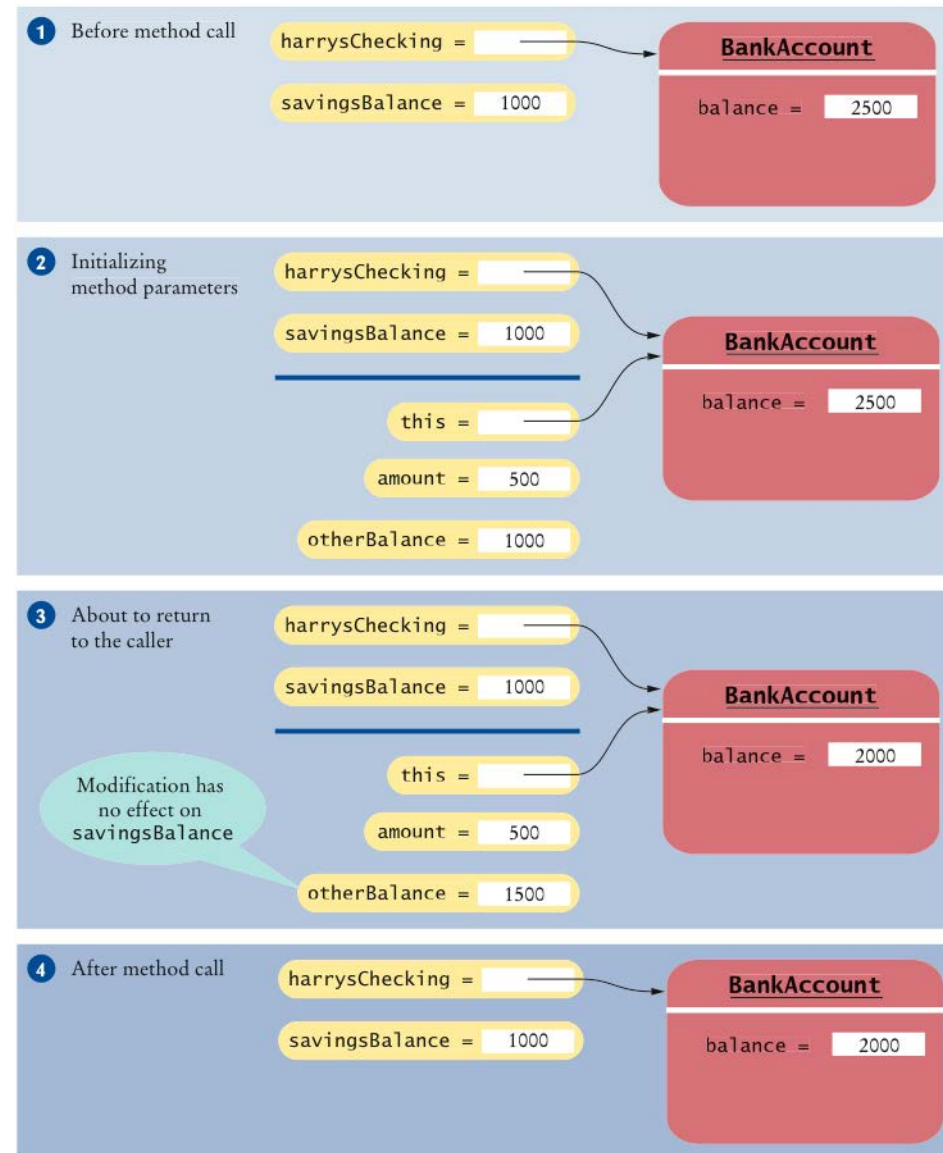


# Common Error: Trying to Modify Primitive Type Parameters

```
double savingsBalance = 1000;
harrysChecking.transfer(500, savingsBalance); ❶
System.out.println(savingsBalance); ❷
...
void transfer(double amount, double otherBalance) ❸
{
 balance = balance - amount;
 otherBalance = otherBalance + amount;
} ❹
```

***Continued***

# Common Error: Trying to Modify Primitive Type Parameters



**Figure 3** Modifying a Numeric Parameter Has No Effect on Caller

# Call by Value and Call by Reference

---

- **Call by value:** Method parameters are copied into the parameter variables when a method starts
- **Call by reference:** Methods can modify parameters
- Java has call by value
- A method can change state of object reference parameters, but cannot replace an object reference with another

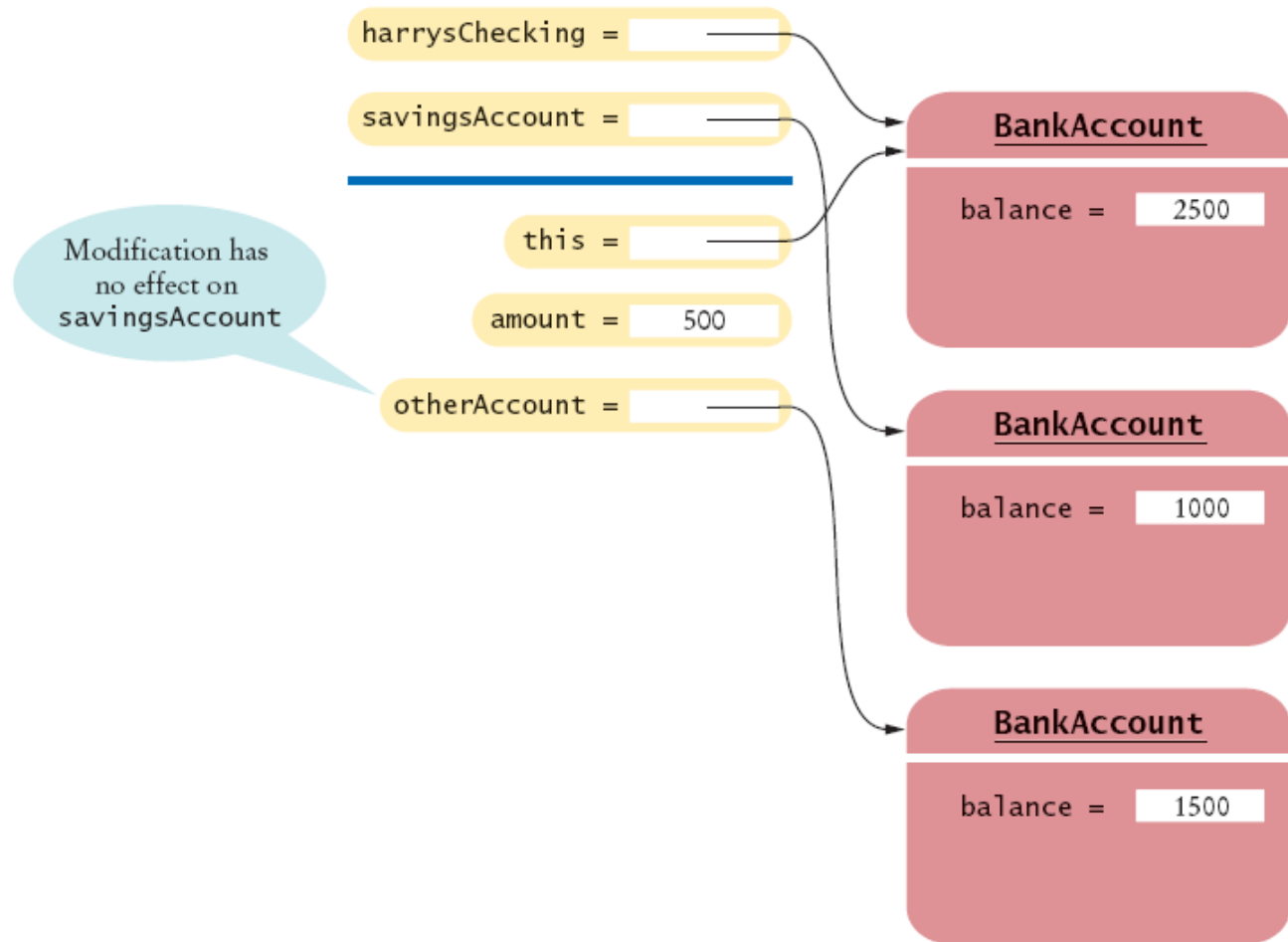


# Call by Value and Call by Reference

```
public class BankAccount
{
 public void transfer(double amount, BankAccount
 otherAccount)
 {
 balance = balance - amount;
 double newBalance = otherAccount.balance + amount;
 otherAccount = new BankAccount(newBalance);
 // Won't work
 }
}
```

# Call by Value Example

```
harrysChecking.transfer(500, savingsAccount);
```



Modifying an Object Reference Parameter Has No Effect on the Caller

# Passing Objects to Methods

- ❑ Passing by value for primitive type value (the value is passed to the parameter)
- ❑ Passing by value for reference type value (the value is the reference to the object)



TestPassObject

Run

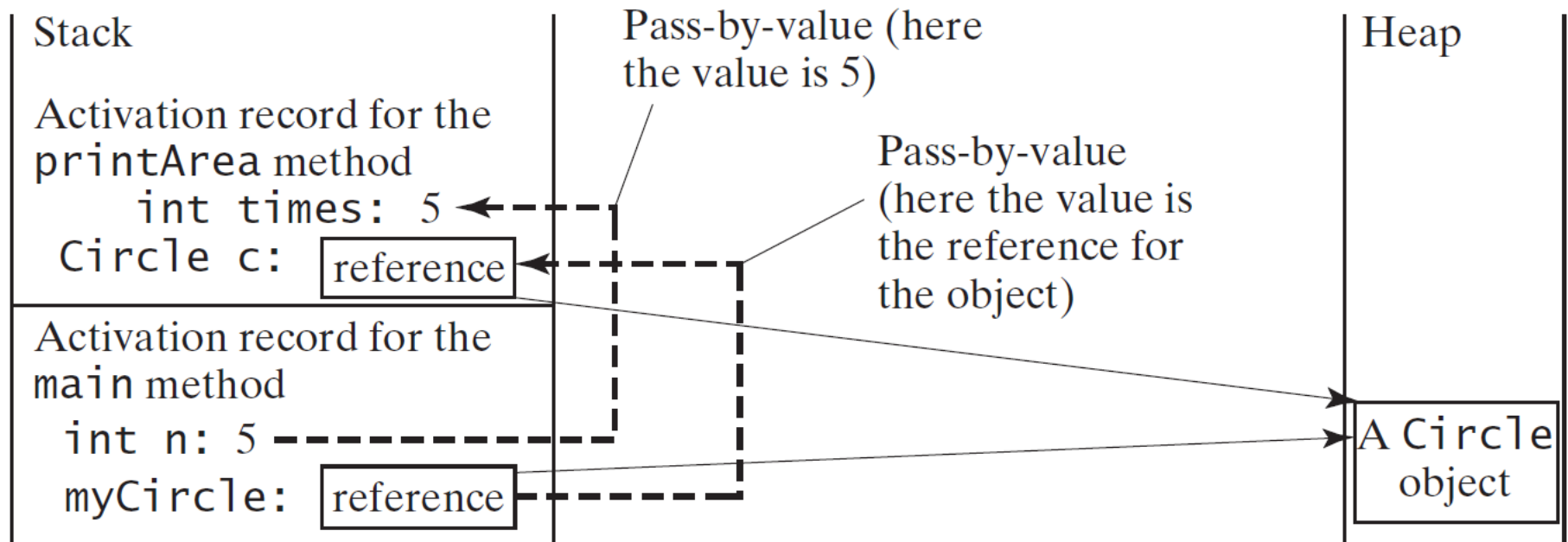
```
public class TestPassObject {
 /** Main method */
 public static void main(String[] args) {
 // Create a Circle object with radius 1
 CircleWithPrivateDataFields myCircle =
 new CircleWithPrivateDataFields(1);

 // Print areas for radius 1, 2, 3, 4, and 5.
 int n = 5;
 printAreas(myCircle, n);

 // See myCircle.radius and times
 System.out.println("\n" + "Radius is " + myCircle.getRadius());
 System.out.println("n is " + n);
 }

 /** Print a table of areas for radius */
 public static void printAreas(
 CircleWithPrivateDataFields c, int times) {
 System.out.println("Radius \t\tArea");
 while (times >= 1) {
 System.out.println(c.getRadius() + "\t\t" + c.getArea());
 c.setRadius(c.getRadius() + 1);
 times--;
 }
 }
}
```

# Passing Objects to Methods, cont.



# Static Methods

---

- Every method must be in a class
- A static method is not invoked on an object
- Why write a method that does not operate on an object
- Common reason: encapsulate some computation that involves only numbers.
  - *Numbers aren't objects, you can't invoke methods on them. E.g. `x.sqrt()` can never be legal in Java*

# Static Methods

- Example:

```
public class Financial
{
 public static double percentOf(double p, double a)
 {
 return (p / 100) * a;
 }
 // More financial methods can be added here.
}
```

- Call with class name instead of object:

```
double tax = Financial.percentOf(taxRate, total);
```

# Static Methods

- If a method manipulates a class that you do not own, you cannot add it to that class
- A static method solves this problem:

```
public class Geometry
{
 public static double area(Rectangle rect)
 {
 return rect.getWidth() * rect.getHeight();
 }
 // More geometry methods can be added here.
}
```

- `main` is static — there aren't any objects yet



# Static Variables

- A static variable belongs to the class, not to any object of the class:

```
public class BankAccount
{
 ...
 private double balance;
 private int accountNumber;
 private static int lastAssignedNumber = 1000;
}
```

- If `lastAssignedNumber` was not `static`, each instance of `BankAccount` would have its own value of `lastAssignedNumber`

# Static Variables

---

- ```
public BankAccount()  
{  
    // Generates next account number to be assigned  
    lastAssignedNumber++; // Updates the static variable  
    accountNumber = lastAssignedNumber;  
    // Sets the instance variable  
}
```

A Static Variable and Instance Variables

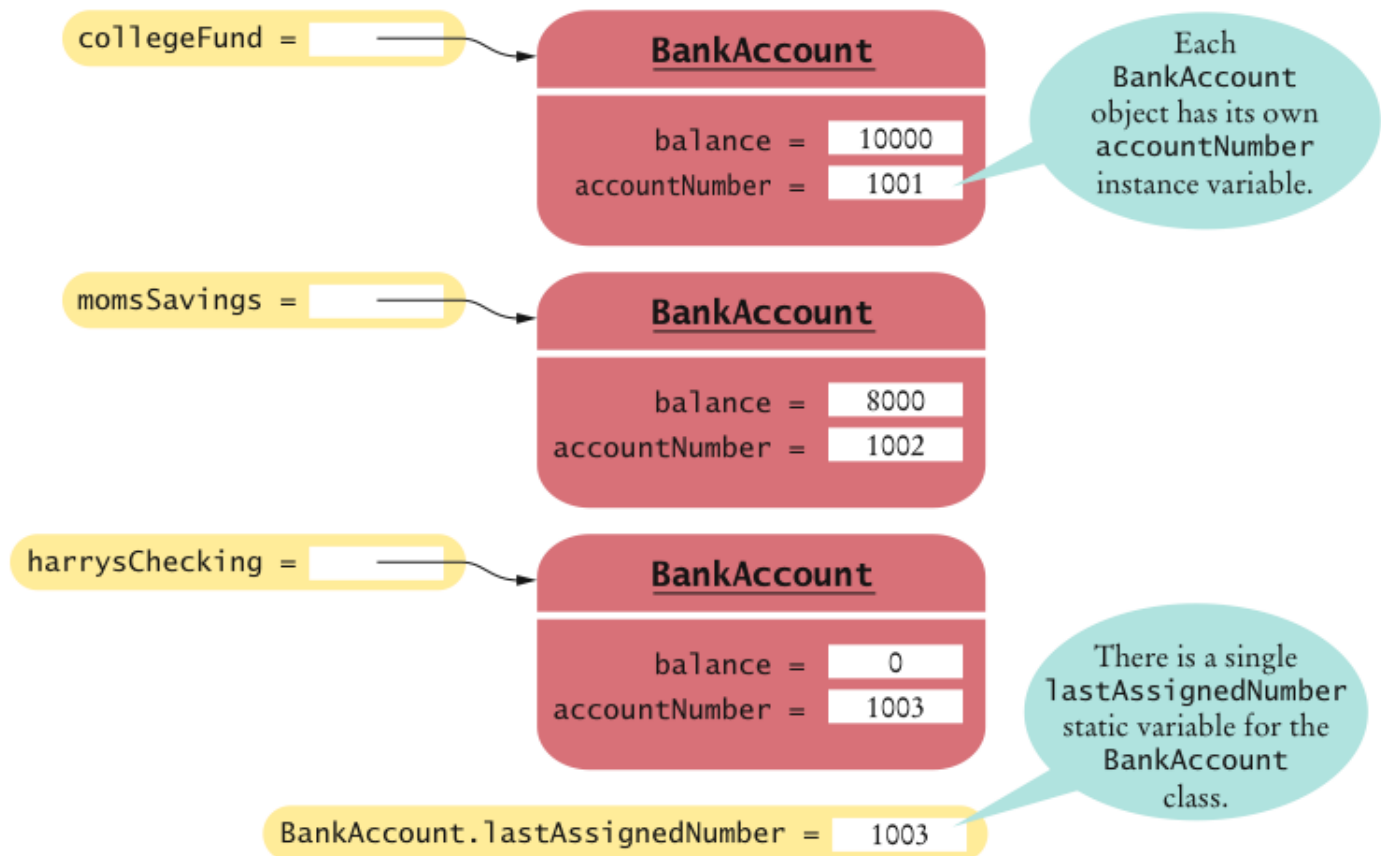


Figure 4
A Static Variable
and Instance
Variables

Static Variables

- Three ways to initialize:
 1. *Do nothing. variable is initialized with 0 (for numbers), false (for boolean values), or null (for objects)*
 2. *Use an explicit initializer, such as*

```
public class BankAccount
{
    ...
    private static int lastAssignedNumber = 1000;
    // Executed once,
}
```
 3. *Use a static initialization block*
- Static variables should always be declared as `private`

Static Variables

- Exception: Static constants, which may be either private or public:

```
public class BankAccount
{
    ...
    public static final double OVERDRAFT_FEE = 5;
    // Refer to it as BankAccount.OVERDRAFT_FEE
}
```

- Minimize the use of static variables (static final variables are ok)

Scope of Local Variables

- **Scope of variable:** Region of program in which the variable can be accessed
- Scope of a local variable extends from its declaration to end of the block that encloses it

Scope of Local Variables

- Sometimes the same variable name is used in two methods:

```
public class RectangleTester
{
    public static double area(Rectangle rect)
    {
        double r = rect.getWidth() * rect.getHeight();
        return r;
    }
    public static void main(String[] args)
    {
        Rectangle r = new Rectangle(5, 10, 20, 30);
        double a = area(r);
        System.out.println(r);
    }
}
```

- These variables are independent from each other; their scopes are disjoint

Scope of Local Variables

- Scope of a local variable cannot contain the definition of another variable with the same name:

```
Rectangle r = new Rectangle(5, 10, 20, 30);  
if (x >= 0)  
{  
    double r = Math.sqrt(x);  
    // Error - can't declare another variable  
    // called r here  
    ...  
}
```


Scope of Local Variables

- However, can have local variables with identical names if scopes do not overlap:

```
if (x >= 0)
{
    double r = Math.sqrt(x);
    ...
} // Scope of r ends here
else
{
    Rectangle r = new Rectangle(5, 10, 20, 30);
    // OK - it is legal to declare another r here
    ...
}
```

Overlapping Scope

- A local variable can *shadow* a variable with the same name
- Local scope wins over class scope:

```
public class Coin
{
    ...
    public double getExchangeValue(double exchangeRate)
    {
        double value; // Local variable
        ...
        return value;
    }
    private String name;
    private double value; // variable with the same name
}
```

Overlapping Scope

- Access shadowed variables by qualifying them with the `this` reference:

```
value = this.value * exchangeRate;
```

Overlapping Scope

- Generally, shadowing an instance variable is poor code — error-prone, hard to read
- Exception: when implementing constructors or setter methods, it can be awkward to come up with different names for instance variables and parameters
- OK:

```
public Coin(double value, String name)
{
    this.value = value;
    this.name = name;
}
```

Visibility Modifiers and Accessor/Mutator Methods

By default, the class, variable, or method can be accessed by any class in the same package.

- ❑ `public`

The class, data, or method is visible to any class in any package.

- ❑ `private`

The data or methods can be accessed only by the declaring class.

The get and set methods are used to read and modify private properties.

```

package p1;

public class C1 {
    public int x;
    int y;
    private int z;

    public void m1() {
    }
    void m2() {
    }
    private void m3() {
    }
}

```

```

package p1;

public class C2 {
    void aMethod() {
        C1 o = new C1();
        can access o.x;
        can access o.y;
        cannot access o.z;

        can invoke o.m1();
        can invoke o.m2();
        cannot invoke o.m3();
    }
}

```

```

package p2;

public class C3 {
    void aMethod() {
        C1 o = new C1();
        can access o.x;
        cannot access o.y;
        cannot access o.z;

        can invoke o.m1();
        cannot invoke o.m2();
        cannot invoke o.m3();
    }
}

```

```

package p1;

class C1 {
    ...
}

```

```

package p1;

public class C2 {
    can access C1
}

```

```

package p2;

public class C3 {
    cannot access C1;
    can access C2;
}

```

The private modifier restricts access to within a class, the default modifier restricts access to within a package, and the public modifier enables unrestricted access.

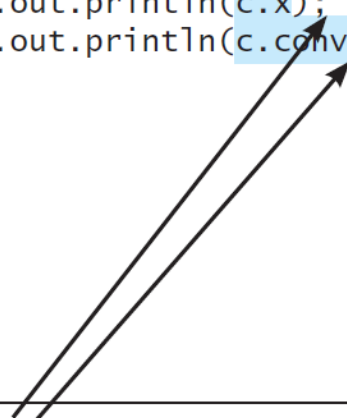
NOTE

An object cannot access its private members, as shown in (b). It is OK, however, if the object is declared in its own class, as shown in (a).

```
public class C {  
    private boolean x;  
  
    public static void main(String[] args) {  
        C c = new C();  
        System.out.println(c.x);  
        System.out.println(c.convert());  
    }  
  
    private int convert() {  
        return x ? 1 : -1;  
    }  
}
```

(a) This is okay because object `c` is used inside the class `C`.

```
public class Test {  
    public static void main(String[] args) {  
        C c = new C();  
        System.out.println(c.x);  
        System.out.println(c.convert());  
    }  
}
```



(b) This is wrong because `x` and `convert` are private in class `C`.

Association

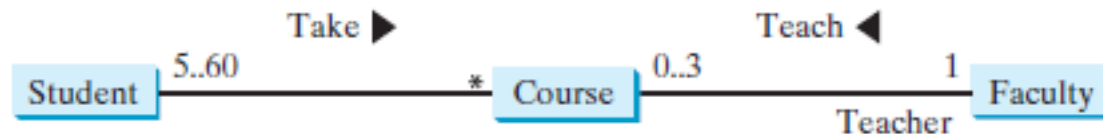


FIGURE 10.4 This UML diagram shows that a student may take any number of courses, a faculty member may teach at most three courses, a course may have from five to sixty students, and a course is taught by only one faculty member.

```
public class Student {
    private Course[]
        courseList;

    public void addCourse(
        Course s) { ... }
}
```

```
public class Course {
    private Student[]
        classList;
    private Faculty faculty;

    public void addStudent(
        Student s) { ... }

    public void setFaculty(
        Faculty faculty) { ... }
}
```

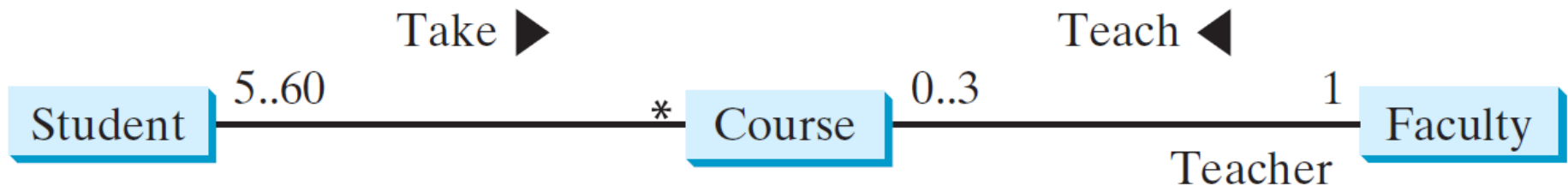
```
public class Faculty {
    private Course[]
        courseList;

    public void addCourse(
        Course c) { ... }
}
```

FIGURE 10.5 The association relations are implemented using data fields and methods in classes.

Object Composition

Composition is actually a special case of the aggregation relationship. Aggregation models *has-a* relationships and represents an ownership relationship between two objects. The owner object is called an *aggregating object* and its class an *aggregating class*. The subject object is called an *aggregated object* and its class an *aggregated class*.



Class Representation

An aggregation relationship is usually represented as a data field in the aggregating class. For example, the relationship in Figure 10.6 can be represented as follows:

```
public class Name {  
    ...  
}
```

Aggregated class

```
public class Student {  
    private Name name;  
    private Address address;  
  
    ...  
}
```

Aggregating class

```
public class Address {  
    ...  
}
```

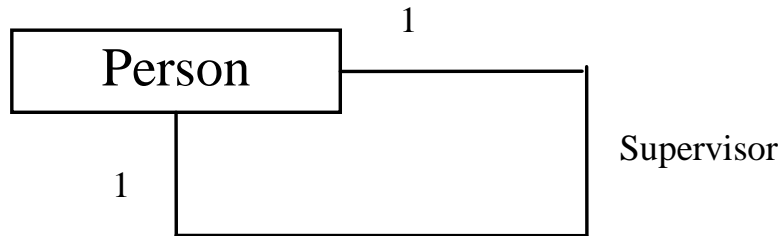
Aggregated class

Aggregation or Composition

Since aggregation and composition relationships are represented using classes in similar ways, many texts don't differentiate them and call both compositions.

Aggregation Between Same Class

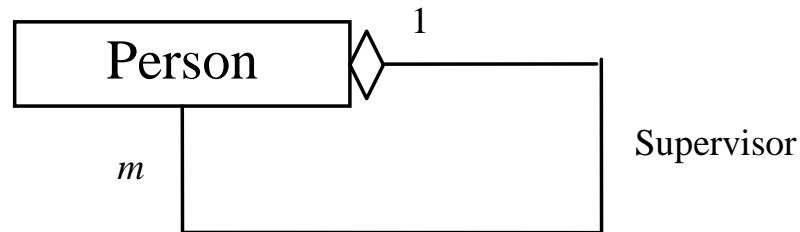
Aggregation may exist between objects of the same class. For example, a person may have a supervisor.



```
public class Person {  
    // The type for the data is the class itself  
    private Person supervisor;  
    ...  
}
```

Aggregation Between Same Class

What happens if a person has several supervisors?



```
public class Person {  
    ...  
    private Person[] supervisors;  
}
```

Example: The Course Class

Course

-courseName: String
-students: String[]
-numberOfStudents: int

+Course(courseName: String)
+getCourseName(): String
+addStudent(student: String): void
+dropStudent(student: String): void
+getStudents(): String[]
+getNumberOfStudents(): int

The name of the course.

An array to store the students for the course.

The number of students (default: 0).

Creates a course with the specified name.

Returns the course name.

Adds a new student to the course.

Drops a student from the course.

Returns the students in the course.

Returns the number of students in the course.



Course



TestCourse

Run

```
public class Course {
    private String courseName;
    private String[] students = new String[4];
    private int numberOfStudents;

    public Course(String courseName) {
        this.courseName = courseName;
    }

    public void addStudent(String student) {
        students[numberOfStudents] = student;
        numberOfStudents++;
    }

    public String[] getStudents() {
        return students;
    }

    public int getNumberOfStudents() {
        return numberOfStudents;
    }

    public String getCourseName() {
        return courseName;
    }

    public void dropStudent(String student) {
        // Left as an exercise in Exercise 10.9
    }
}
```

```
public class TestCourse {
    public static void main(String[] args) {
        Course course1 = new Course("Data Structures");
        Course course2 = new Course("Database Systems");

        course1.addStudent("Peter Jones");
        course1.addStudent("Brian Smith");
        course1.addStudent("Anne Kennedy");

        course2.addStudent("Peter Jones");
        course2.addStudent("Steve Smith");

        System.out.println("Number of students in course1: "
            + course1.getNumberOfStudents());
        String[] students = course1.getStudents();
        for (int i = 0; i < course1.getNumberOfStudents(); i++)
            System.out.print(students[i] + ", ");

        System.out.println();
        System.out.print("Number of students in course2: "
            + course2.getNumberOfStudents());
    }
}
```


Packages

- **Package:** Set of related classes
- Important packages in the Java library:

Package	Purpose	Sample Class
<code>java.lang</code>	Language support	<code>Math</code>
<code>java.util</code>	Utilities	<code>Random</code>
<code>java.io</code>	Input and output	<code>PrintStream</code>
<code>java.awt</code>	Abstract Windowing Toolkit	<code>Color</code>
<code>java.applet</code>	Applets	<code>Applet</code>
<code>java.net</code>	Networking	<code>Socket</code>
<code>java.sql</code>	Database Access	<code>ResultSet</code>
<code>javax.swing</code>	Swing user interface	<code>JButton</code>
<code>org.w3c.dom</code>	Document Object Model for XML documents	<code>Document</code>

Organizing Related Classes into Packages

- To put classes in a package, you must place a line

```
package packageName;
```

as the first instruction in the source file containing the classes

- Package name consists of one or more identifiers separated by periods

Organizing Related Classes into Packages

- For example, to put the `Financial` class introduced into a package named `com.horstmann.bigjava`, the `Financial.java` file must start as follows:

```
package com.horstmann.bigjava;
```

```
public class Financial  
{  
    ...  
}
```

- Default package has no name, no `package` statement

Syntax 8.2 Package Specification

Syntax `package` *packageName*;

Example

The classes in this file
belong to this package.

`package` com.horstmann.bigjava;

A good choice for a package name
is a domain name in reverse.

Importing Packages

- Can always use class without importing:

```
java.util.Scanner in = new java.util.Scanner(System.in);
```

- Tedious to use fully qualified name
- Import lets you use shorter class name:

```
import java.util.Scanner;  
...  
Scanner in = new Scanner(System.in)
```

- Can import all classes in a package:

```
import java.util.*;
```

- Never need to import `java.lang`
- You don't need to import other classes in the same package

Package Names

- Use packages to avoid name clashes

`java.util.Timer`

vs.

`javax.swing.Timer`

- Package names should be unambiguous
- Recommendation: start with reversed domain name:

`com.horstmann.bigjava`

- `edu.sjsu.cs.walters`: for Britney Walters' classes
(`walters@cs.sjsu.edu`)

- Path name should match package name:

`com/horstmann/bigjava/Financial.java`

Package and Source Files

- **Base directory:** holds your program's Files
- Path name, relative to base directory, must match package name:

```
com/horstmann/bigjava/Financial.java
```

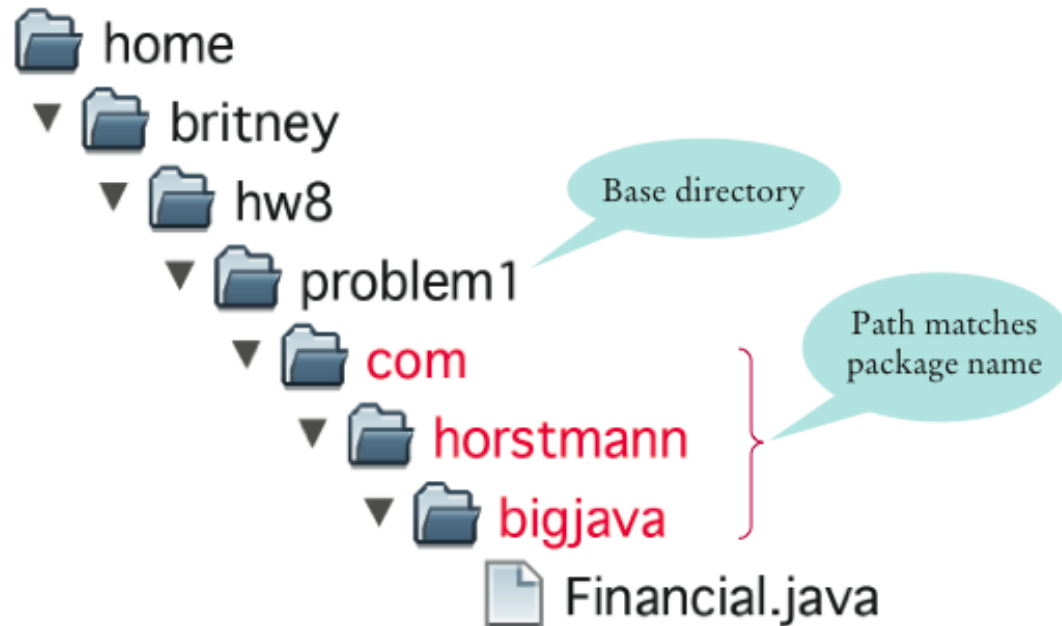


Figure 5
Base Directories
and Subdirectories
for Packages