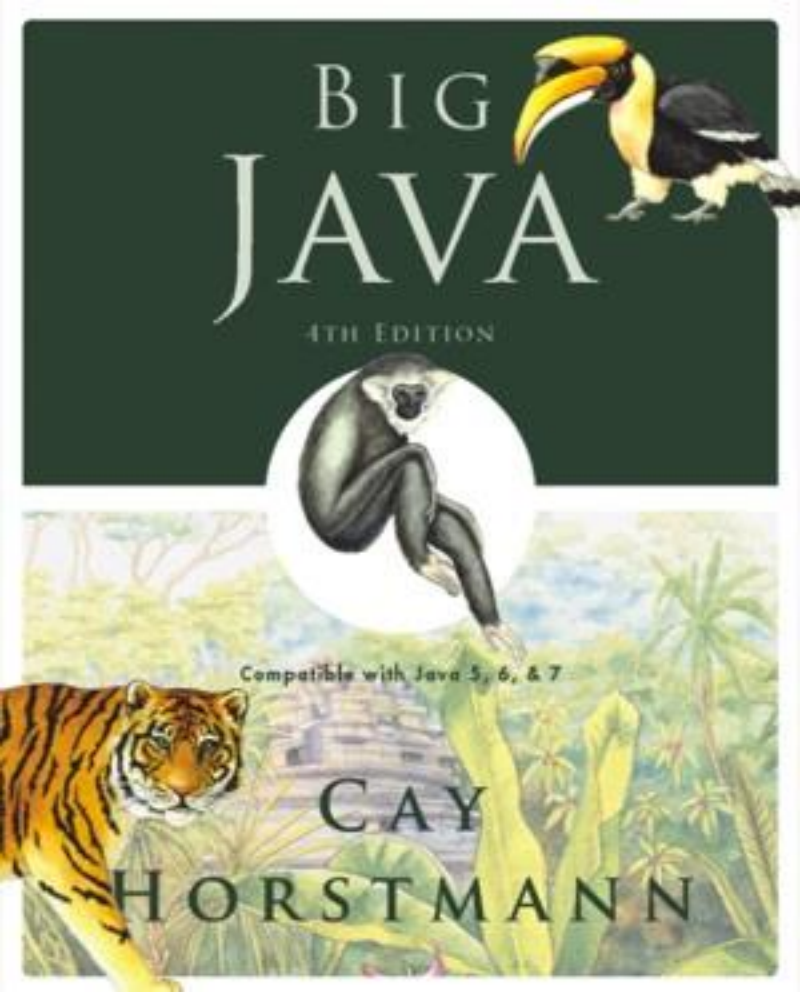


Chapter 10 – Inheritance

Big Java by Cay Horstmann
Copyright © 2009 by John Wiley & Sons. All rights reserved.

Chapter Goals

- To learn about inheritance
- To understand how to inherit and override superclass methods
- To be able to invoke superclass constructors
- To learn about `protected` and package access control
- To understand the common superclass `Object` and to override its `toString` and `equals` methods

G To use inheritance for customizing user interfaces

Big Java by Cay Horstmann
Copyright © 2009 by John Wiley & Sons. All rights reserved.

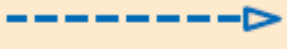
Object-Oriented Design

1. Discover classes
2. Determine responsibilities of each class
3. Describe relationships between the classes

Relationships Between Classes

- Inheritance
- Aggregation
- Dependency

UML Relationship Symbols

Relationship	Symbol	Line Style	Arrow Tip
Inheritance		Solid	Triangle
Interface Implementation		Dotted	Triangle
Aggregation		Solid	Diamond
Dependency		Dotted	Open

Inheritance

- *Is-a* relationship
- Relationship between a more general class (superclass) and a more specialized class (subclass)
- Every savings account is a bank account
- Car is a vehicle

- Benefits of inheritance
- Software reuse
  - Inherit from existing classes
    - Include additional data members and methods
    - Redefine superclass methods
    - No direct access to superclass's source code
      - Link to object code

Aggregation

- *Has-a* relationship
- Objects of one class contain references to objects of another class
- Use an instance variable
  - *A tire has a circle as its boundary:*

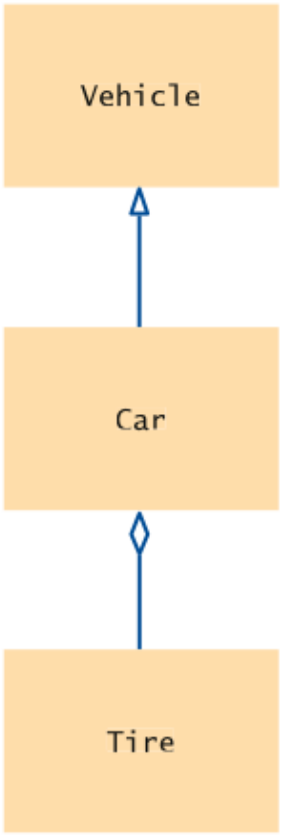
```
class Tire
{
    ...
    private String rating;
    private Circle boundary;
}
```

- Every car has a tire (in fact, it has four)

Example

```
class Car extends Vehicle
{
    ...
    private Tire[] tires;
}
```

Figure 6
UML Notation for
Inheritance and Aggregation



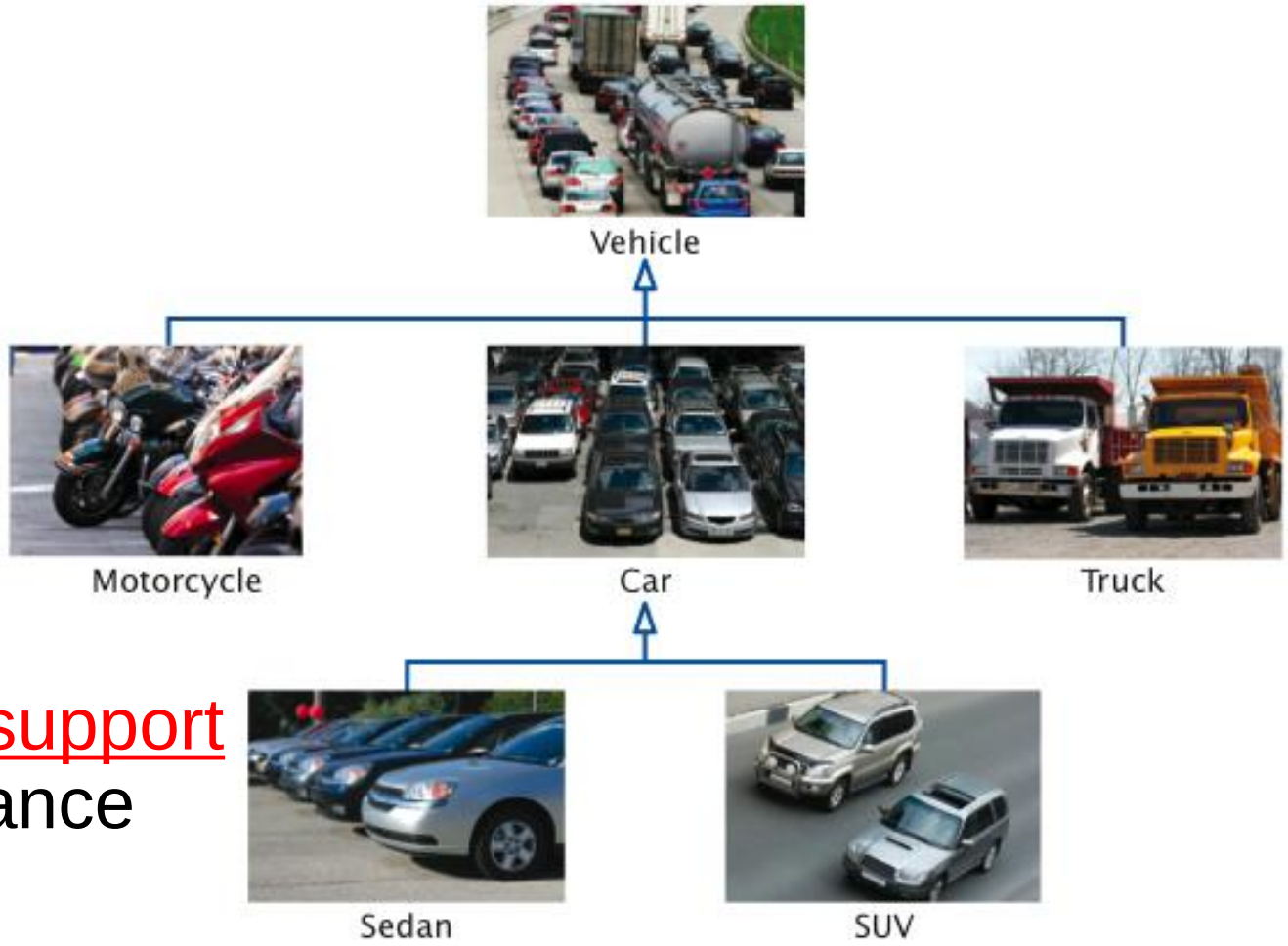
Dependency

- *Uses* relationship
- Example: Many of our applications depend on the `Scanner` class to read input
- Aggregation is a stronger form of dependency
- Use aggregation to remember another object between method calls

Inheritance Hierarchies

- Often categorize concepts into *hierarchies*:

Figure 1
A Hierarchy of Vehicle Types



Java does not support multiple inheritance (use interface)

Big Java by Cay Horstmann
Copyright © 2009 by John Wiley & Sons. All rights reserved.

Inheritance Hierarchies

- Set of classes can form an *inheritance hierarchy*
  - *Classes representing the most general concepts are near the root, more specialized classes towards the branches:*

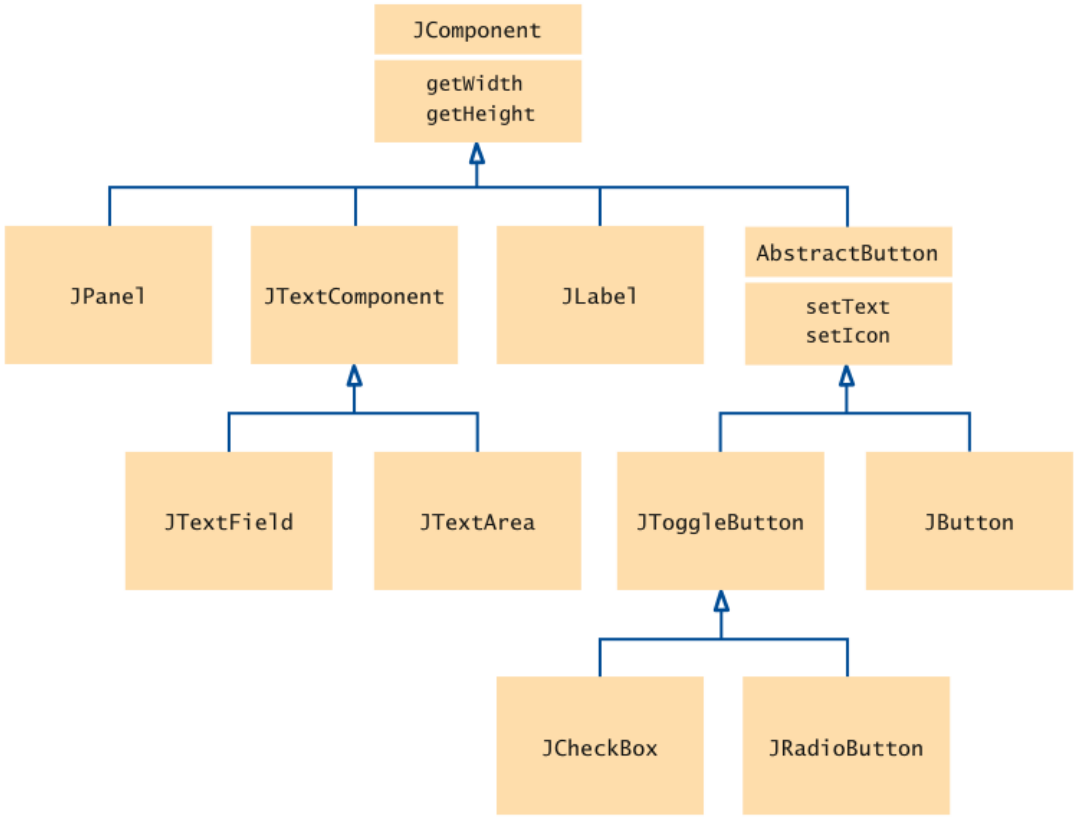


Figure 2 A Part of the Hierarchy of Swing User Interface Components

Big Java by Cay Horstmann
Copyright © 2009 by John Wiley & Sons. All rights reserved.

Inheritance Hierarchies

- **Superclass:** more general class
- **Subclass:** more specialized class that inherits from the superclass
  - *Example:* `JPanel` *is a subclass of* `JComponent`

- Every class extends the Object class
- A subclass inherits all the *non-private members* (fields/instance variables, methods) from its superclass.
- Constructors are not members, so they are not inherited by subclasses, but the constructor of the superclass can be invoked from the subclass.

Big Java by Cay Horstmann
Copyright © 2009 by John Wiley & Sons. All rights reserved.

Self Check 10.1

What is the purpose of the `JTextComponent` class in Figure 2?

Answer: To express the common behavior of text variables and text components.

Inheritance Hierarchies

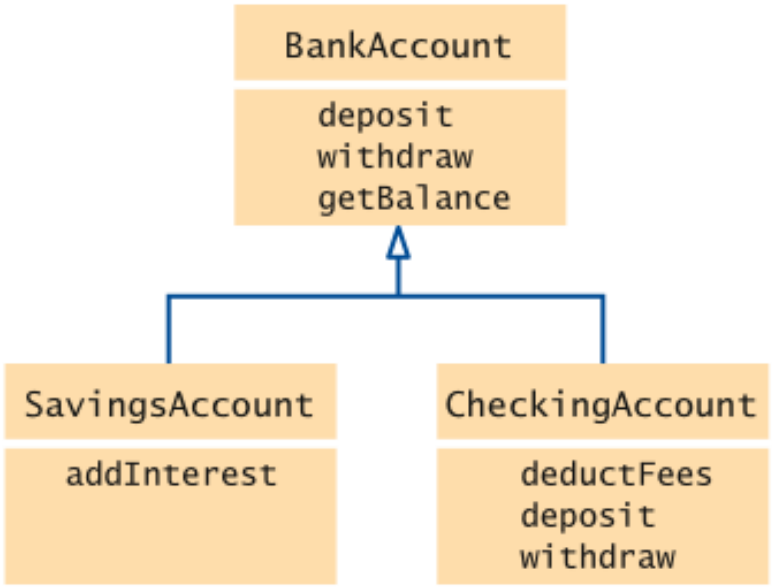


Figure 3 Inheritance Hierarchy for Bank Account Classes

Big Java by Cay Horstmann
Copyright © 2009 by John Wiley & Sons. All rights reserved.

Inheritance Hierarchies

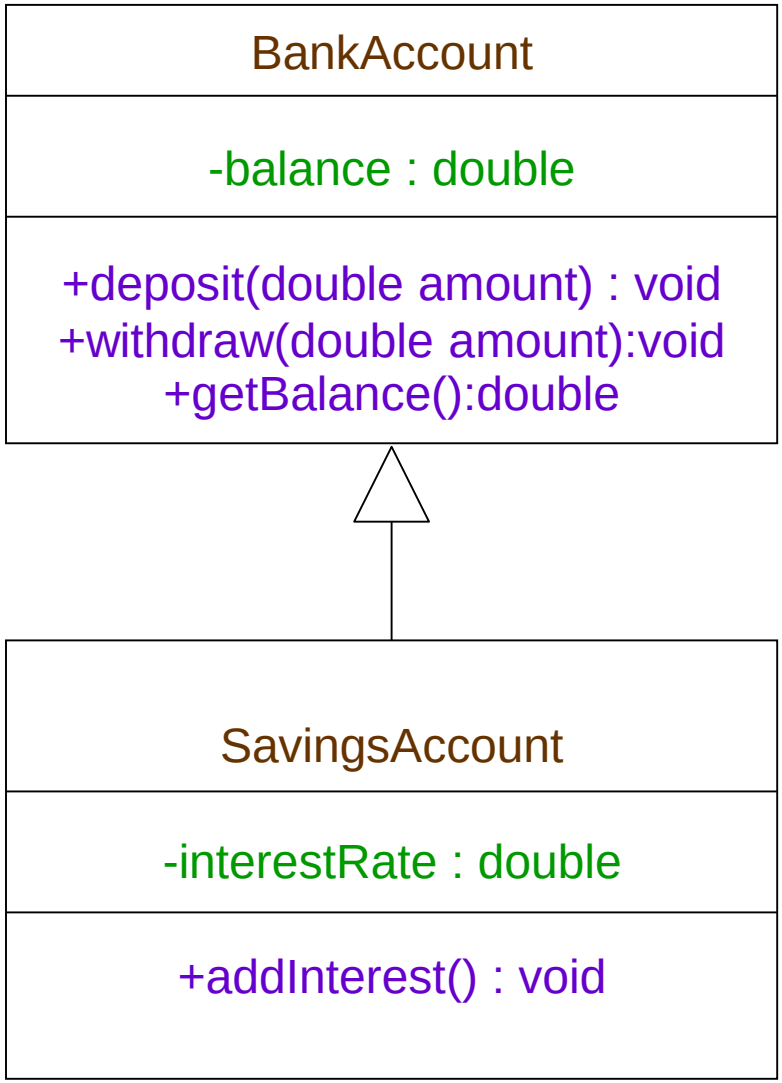
- **Example:** Different account types:
  1. *Checking account:*
     - *No interest*
     - *Small number of free transactions per month*
     - *Charges transaction fee for additional transactions*
  2. *Savings account:*
     - *Earns interest that compounds monthly*
- Superclass: `BankAccount`
- Subclasses: `CheckingAccount` & `SavingsAccount`

Big Java by Cay Horstmann
Copyright © 2009 by John Wiley & Sons. All rights reserved.

```
//BankAccount.java
public class BankAccount
{
   private double balance;

   public BankAccount()
   {
      // by default the balance value = 0
      // balance = 0 ;
   }

   public BankAccount(double initialBalance)
   {
      balance = initialBalance;
   }
```

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

   public void withdraw(double amount)
   {
      balance = balance - amount;
   }

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

UML class hierarchy



Mark	Visibility type
+	Public
#	Protected
-	Private
~	Package

Self Check 10.2

Why don't we place the `addInterest` method in the `BankAccount` class?

Big Java by Cay Horstmann
Copyright © 2009 by John Wiley & Sons. All rights reserved.

Inheritance Hierarchies

- Inheritance is a mechanism for extending existing classes by adding instance variables and methods:

```
class SavingsAccount extends BankAccount
{
   added instance variables
   new methods
}
```

Purpose:

- To define a subclass that inherits from an existing class (superclass)
- define constructors, methods and instance variables that are added in the subclass
- redefine (override) methods of the superclass

- A subclass inherits the methods of its superclass:

```
SavingsAccount collegeFund = new SavingsAccount(10);
// Savings account with 10% interest
collegeFund.deposit(500);
// OK to use BankAccount method with SavingsAccount object
```

Big Java by Cay Horstmann
Copyright © 2009 by John Wiley & Sons. All rights reserved.

Inheritance Hierarchies

- In subclass, specify added instance variables, added methods, and changed or overridden methods:

```
public class SavingsAccount extends BankAccount
{
   private double interestRate;

   public SavingsAccount(double rate)
   {
      Constructor implementation
   }

   public void addInterest()
   {
      Method implementation
   }
}
```

Big Java by Cay Horstmann
Copyright © 2009 by John Wiley & Sons. All rights reserved.

Inheritance Hierarchies

- Instance variables declared in the superclass are present in subclass objects
- `SavingsAccount` object inherits the balance instance variable from `BankAccount`, and gains one additional instance variable, `interestRate`:

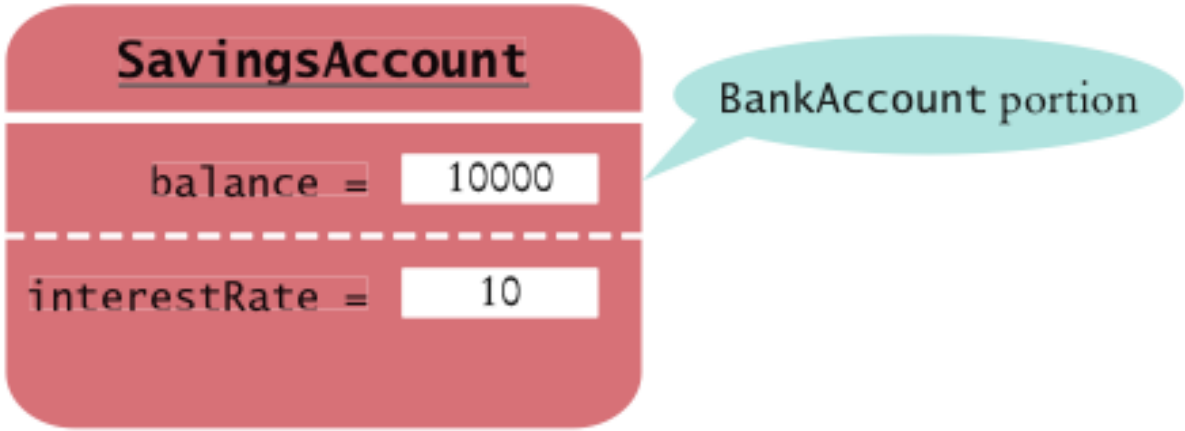


Figure 4
Layout of a Subclass Object

SavingsAccount cannot access the balance directly, but can use public methods of BankAccount to get or set value of balance because it is declared : private

Big Java by Cay Horstmann
Copyright © 2009 by John Wiley & Sons. All rights reserved.

Inheritance Hierarchies

- Implement the new `addInterest` method:

```
public class SavingsAccount extends BankAccount
{
   private double interestRate;
   public SavingsAccount(double rate)
   {
      interestRate = rate;
   }
   public void addInterest()
   {
      double interest = getBalance() * interestRate / 100;
      deposit(interest);
   }
}
```

Big Java by Cay Horstmann
Copyright © 2009 by John Wiley & Sons. All rights reserved.

Inheritance Hierarchies

- A subclass has no access to private instance variables of its superclass
- **Encapsulation:** `addInterest` calls `getBalance` rather than updating the `balance` variable of the superclass (variable is `private`)
- Note that `addInterest` calls `getBalance` without specifying an implicit parameter (the calls apply to the same object)

Big Java by Cay Horstmann
Copyright © 2009 by John Wiley & Sons. All rights reserved.

ch10/accounts/SavingsAccount.java

```
1  /**
2     An account that earns interest at a fixed rate.
3  */
4  public class SavingsAccount extends BankAccount
5  {
6     private double interestRate;
7
8     /**
9        Constructs a bank account with a given interest rate.
10       @param rate the interest rate
11    */
12    public SavingsAccount(double rate)
13    {
14       interestRate = rate;
15    }
16
```

Continued

Big Java by Cay Horstmann
Copyright © 2009 by John Wiley & Sons. All rights reserved.

ch10/accounts/SavingsAccount.java (cont.)

```
17      /**
18         Adds the earned interest to the account balance.
19      */
20      public void addInterest()
21      {
22         double interest = getBalance() * interestRate / 100;
23         deposit(interest);
24      }
25   }
```

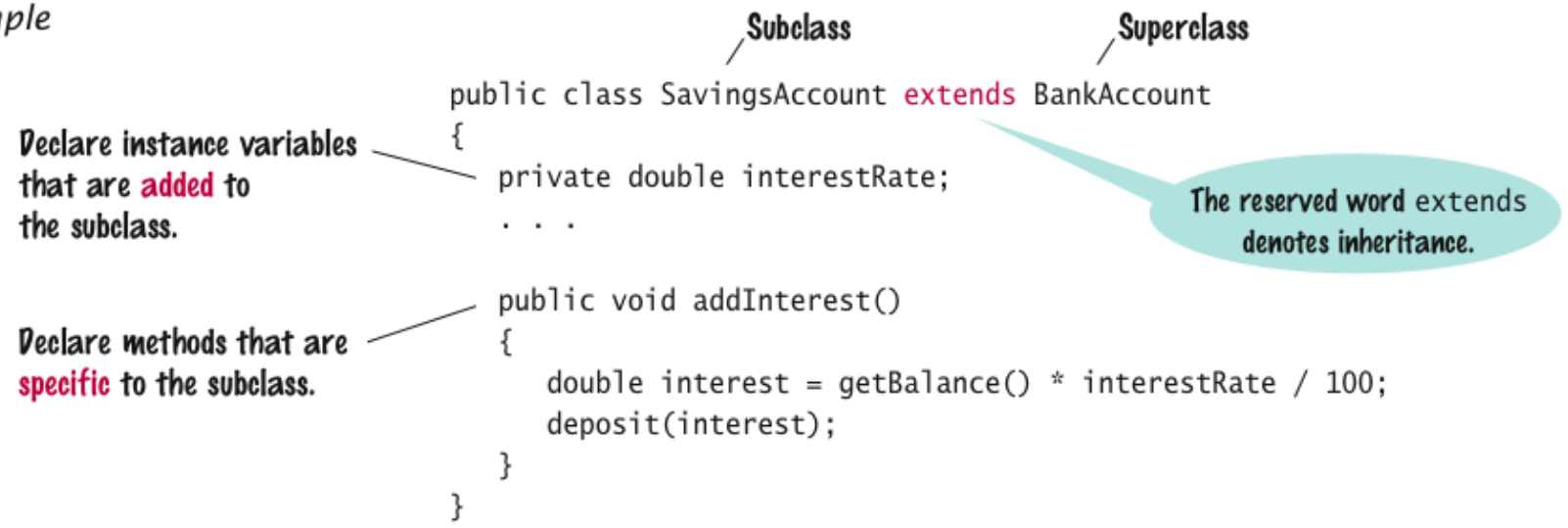
Big Java by Cay Horstmann
Copyright © 2009 by John Wiley & Sons. All rights reserved.

Syntax 10.1 Inheritance

Syntax

```
class SubclassName extends SuperclassName
{
   instance variables
   methods
}
```

Example



Big Java by Cay Horstmann
Copyright © 2009 by John Wiley & Sons. All rights reserved.

Self Check 10.3

Which instance variables does an object of class `SavingsAccount` have?

Big Java by Cay Horstmann
Copyright © 2009 by John Wiley & Sons. All rights reserved.

Self Check 10.4

Name four methods that you can apply to `SavingsAccount` objects.

Big Java by Cay Horstmann
Copyright © 2009 by John Wiley & Sons. All rights reserved.

Self Check 10.5

If the class `Manager` extends the class `Employee`, which class is the superclass and which is the subclass?

Copyright © 2009 by John Wiley & Sons. All rights reserved.

Common Error: Shadowing Instance Variables

- A subclass has no access to the private instance variables of the superclass:

```
public class SavingsAccount extends BankAccount
{
   public void addInterest()
   {
      double interest = getBalance() * interestRate / 100;
      balance = balance + interest; // Error
   }
   . . .
}
```

Big Java by Cay Horstmann
Copyright © 2009 by John Wiley & Sons. All rights reserved.

Common Error: Shadowing Instance Variables

- Beginner's error: "solve" this problem by adding another instance variable with same name:

```
public class SavingsAccount extends BankAccount
{
   private double balance; // Don't
   public void addInterest()
   {
      double interest = getBalance() * interestRate / 100;
      balance = balance + interest; // Compiles but doesn't
         // update the correct balance
   }
   . . .
}
```

Big Java by Cay Horstmann
Copyright © 2009 by John Wiley & Sons. All rights reserved.

Common Error: Shadowing Instance Variables

- Now the addInterest method compiles, but it doesn't update the correct balance!

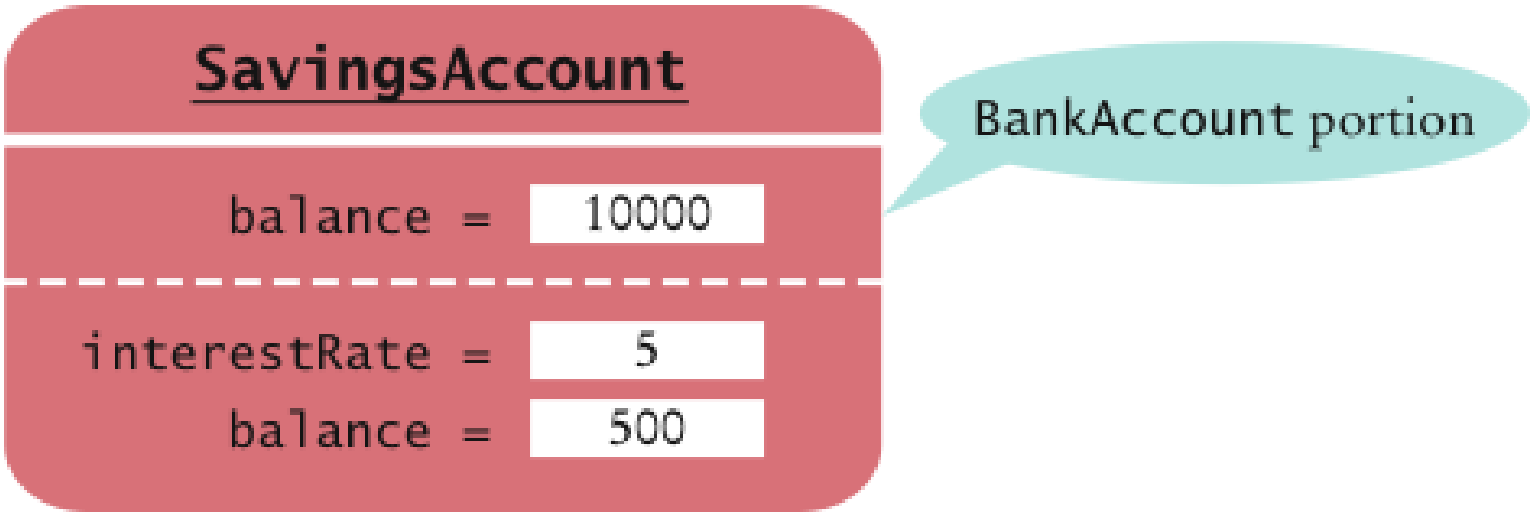
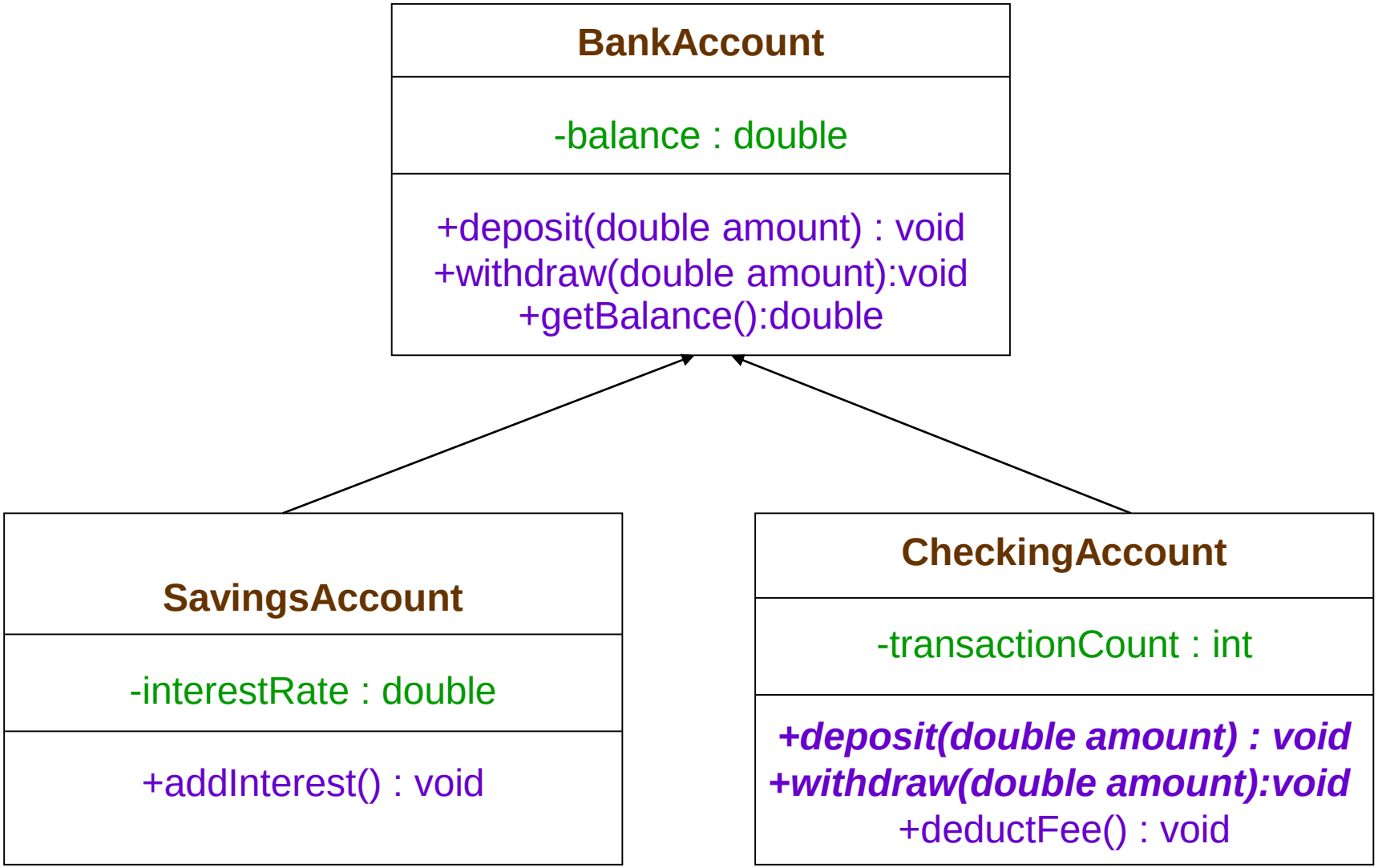


Figure 5 Shadowing Instance Variables

Big Java by Cay Horstmann
Copyright © 2009 by John Wiley & Sons. All rights reserved.

UML class hierarchy



Inheritance Hierarchies

- Behavior of account classes:
  - *All support* `getBalance` *method*
  - *Also support* `deposit` *and* `withdraw` *methods, but implementation details differ*
  - *Checking account needs a method* `deductFees` *to deduct the monthly fees and to reset the transaction counter*
  - *Checking account must override* `deposit` *and* `withdraw` *methods to count the transactions*

Big Java by Cay Horstmann
Copyright © 2009 by John Wiley & Sons. All rights reserved.

Example : CheckingAccount extends BankAccount

- First three transactions are free
- Charge $2 for every additional transaction
- Must override `deposit, withdraw` to increment transaction count
- `deductFees` method deducts accumulated fees, resets transaction count

Overriding Methods

- A subclass method **overrides** a superclass method if it has the same name and parameter types as a superclass method
  - *When such a method is applied to a subclass object, the overriding method is executed*

Big Java by Cay Horstmann
Copyright © 2009 by John Wiley & Sons. All rights reserved.

Overriding vs. Overloading

```
public class Test {
  public static void main(String[] args) {
    A a = new A();
    a.p(10);
    a.p(10.0);
  }
}

class B {
  public void p(double i) {
    System.out.println(i * 2);
  }
}

class A extends B {
  // This method overrides the method in B
  public void p(double i) {
    System.out.println(i);
  }
}
```

```
public class Test {
  public static void main(String[] args) {
    A a = new A();
    a.p(10);
    a.p(10.0);
  }
}

class B {
  public void p(double i) {
    System.out.println(i * 2);
  }
}

class A extends B {
  // This method overloads the method in B
  public void p(int i) {
    System.out.println(i);
  }
}
```

The example above show the differences between overriding and overloading.
In (a), the method p(double i) in class A overrides the same method in class B.
In (b), the class A has two overloaded methods: p(double i) and p(int i).
The method p(double i) is inherited from B.

Overriding Methods

- Example: `deposit` and `withdraw` methods of the `CheckingAccount` class override the `deposit` and `withdraw` methods of the `BankAccount` class to handle transaction fees:

```
public class BankAccount
{
   . . .
   public void deposit(double amount) { . . . }
   public void withdraw(double amount) { . . . }
   public double getBalance() { . . . }
}
public class CheckingAccount extends BankAccount
{
   . . .
   public void deposit(double amount) { . . . }
   public void withdraw(double amount) { . . . }
   public void deductFees() { . . . }
}
```

Big Java by Cay Horstmann
Copyright © 2009 by John Wiley & Sons. All rights reserved.

Overriding Methods

- Problem: Overriding method `deposit` can't simply add `amount` to `balance`:

```
public class CheckingAccount extends BankAccount
{
   . . .
   public void deposit(double amount)
   {
      transactionCount++;
      // Now add amount to balance
      balance = balance + amount; // Error
   }
}
```

- If you want to modify a private superclass instance variable, you must use a public method of the superclass
- `deposit` method of `CheckingAccount` must invoke the `deposit` method of `BankAccount`

Big Java by Cay Horstmann
Copyright © 2009 by John Wiley & Sons. All rights reserved.

Overriding Methods

- Idea:

```
public class CheckingAccount extends BankAccount
{
   public void deposit(double amount)
   {
      transactionCount++;
      // Now add amount to balance
      deposit; // Not complete
   }
}
```

- Won't work because compiler interprets

```
deposit(amount);
```

as

```
this.deposit(amount);
```

which calls the method we are currently writing ⇒ infinite recursion

Big Java by Cay Horstmann
Copyright © 2009 by John Wiley & Sons. All rights reserved.

Overriding Methods

- Use the `super` reserved word to call a method of the superclass:

```
public class CheckingAccount extends BankAccount
{
   public void deposit(double amount)
   {
      transactionCount++;
      // Now add amount to balance
      super.deposit
   }
}
```

Big Java by Cay Horstmann
Copyright © 2009 by John Wiley & Sons. All rights reserved.

Overriding Methods

- Remaining methods of `CheckingAccount` also invoke a superclass method:

```
public class CheckingAccount extends BankAccount
{
   private static final int FREE_TRANSACTIONS = 3;
   private static final double TRANSACTION_FEE = 2.0;
   private int transactionCount;
   . . .
   public void withdraw(double amount
   {
      transactionCount++;
      // Now subtract amount from balance
      super.withdraw(amount);
   }
```

Continued

Big Java by Cay Horstmann
Copyright © 2009 by John Wiley & Sons. All rights reserved.

Overriding Methods (cont.)

```
   public void deductFees()
   {
      if (transactionCount > FREE_TRANSACTIONS)
      {
         double fees = TRANSACTION_FEE *
            (transactionCount - FREE_TRANSACTIONS);
         super.withdraw(fees);
      }
      transactionCount = 0;
   }
   . . .
}
```

Big Java by Cay Horstmann
Copyright © 2009 by John Wiley & Sons. All rights reserved.

Syntax 10.2 Calling a Superclass Method

Syntax `super.methodName(parameters);`

Example

```
public void deposit(double amount)
{
   transactionCount++;
   super.deposit(amount);
}
```

Calls the method of the superclass instead of the method of the current class.

If you omit `super`, this method calls itself.

Big Java by Cay Horstmann
Copyright © 2009 by John Wiley & Sons. All rights reserved.

Self Check 10.6

Categorize the methods of the `SavingsAccount` class as inherited, new, and overridden.

Big Java by Cay Horstmann
Copyright © 2009 by John Wiley & Sons. All rights reserved.

Self Check 10.7

Why does the `withdraw` method of the `CheckingAccount` class call `super.withdraw`?

Copyright © 2009 by John Wiley & Sons. All rights reserved.

Self Check 10.8

Why does the `deductFees` method set the transaction count to zero?

Big Java by Cay Horstmann
Copyright © 2009 by John Wiley & Sons. All rights reserved.

Subclass Construction

- To call the superclass constructor, use the `super` reserved word in the first statement of the subclass constructor:

```
public class CheckingAccount extends BankAccount
{
   public CheckingAccount(double initialBalance)
   {
      // Construct superclass
      super(initialBalance);
      // Initialize transaction count
      transactionCount = 0;
   }
   ...
}
```

Big Java by Cay Horstmann
Copyright © 2009 by John Wiley & Sons. All rights reserved.

Subclass Construction

- When subclass constructor doesn't call superclass constructor, the superclass must have a constructor with no parameters
  - *If, however, all constructors of the superclass require parameters, then the compiler reports an error*

```
public class SavingsAccount extends BankAccount {
        ...
        public SavingsAccount(double rate) {
        interestRate = rate;
}
        ...
}
/* compiler automatically call default constructor of
superclass because subclass doesn’t explicitly call superclass
constructor */
```

// the superclass must have a constructor with no parameters

Big Java by Cay Horstmann
Copyright © 2009 by John Wiley & Sons. All rights reserved.

ch10/accounts/CheckingAccount.java

```
1  /**
2     A checking account that charges transaction fees.
3  */
4  public class CheckingAccount extends BankAccount
5  {
6     private static final int FREE_TRANSACTIONS = 3;
7     private static final double TRANSACTION_FEE = 2.0;
8
9     private int transactionCount;
10
11    /**
12       Constructs a checking account with a given balance.
13       @param initialBalance the initial balance
14    */
15    public CheckingAccount(double initialBalance)
16    {
17       // Construct superclass
18       super(initialBalance);
19
20       // Initialize transaction count
21       transactionCount = 0;
22    }
23
```

Continued

Big Java by Cay Horstmann
Copyright © 2009 by John Wiley & Sons. All rights reserved.

ch10/accounts/CheckingAccount.java (cont.)

```
24      public void deposit(double amount)
25      {
26         transactionCount++;
27         // Now add amount to balance
28         super.deposit(amount);
29      }
30
31      public void withdraw(double amount)
32      {
33         transactionCount++;
34         // Now subtract amount from balance
35         super.withdraw(amount);
36      }
37
```

Continued

Big Java by Cay Horstmann
Copyright © 2009 by John Wiley & Sons. All rights reserved.

ch10/accounts/CheckingAccount.java (cont.)

```
38      /**
39         Deducts the accumulated fees and resets the
40         transaction count.
41      */
42      public void deductFees()
43      {
44         if (transactionCount > FREE_TRANSACTIONS)
45         {
46            double fees = TRANSACTION_FEE *
47                  (transactionCount - FREE_TRANSACTIONS);
48            super.withdraw(fees);
49         }
50         transactionCount = 0;
51      }
52   }
```

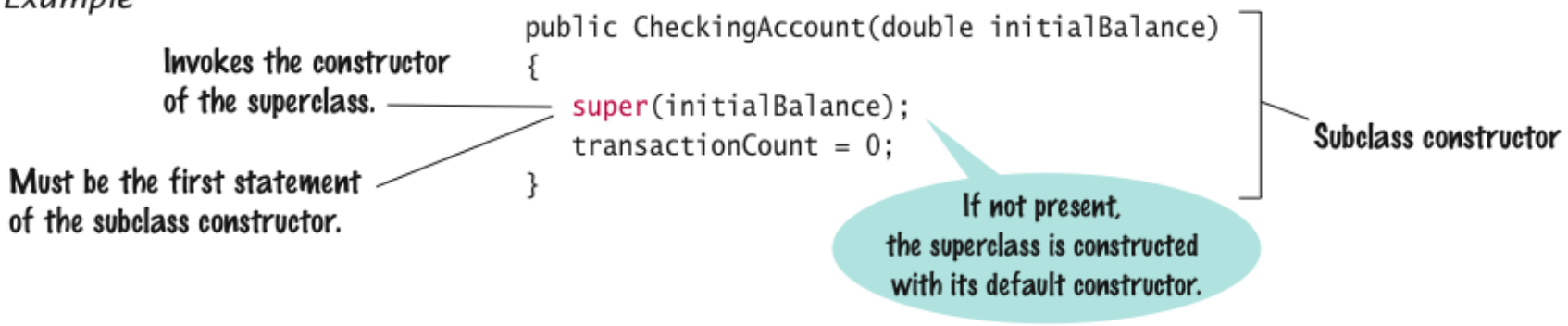
Big Java by Cay Horstmann
Copyright © 2009 by John Wiley & Sons. All rights reserved.

Syntax 10.3 Calling a Superclass Constructor

Syntax

```
accessSpecifier ClassName(parameterType parameterName, . . .)
{
   super(parameters);
   . . .
}
```

Example



Big Java by Cay Horstmann
Copyright © 2009 by John Wiley & Sons. All rights reserved.

Self Check 10.9

Why didn't the `SavingsAccount` constructor in Section 10.2 call its superclass constructor?

Copyright © 2009 by John Wiley & Sons. All rights reserved.

Self Check 10.10

When you invoke a superclass method with the `super` keyword, does the call have to be the first statement of the subclass method?

Big Java by Cay Horstmann
Copyright © 2009 by John Wiley & Sons. All rights reserved.

แบบฝึกหัดในห้อง

- จงสร้าง superclass Employee และ subclass Salesman และ Secretary
- Employee มีข้อมูลคือ ชื่อ (String) ปีที่เริ่มงาน (int) เงินเดือน (double) และมีเมธอดคือ getName() ซึ่งคืนค่าชื่อและสกุลออกมา getStartYear() ซึ่งคืนค่าปีที่เริ่มงานออกมา และ getSalary() ซึ่งคืนค่าเงินเดือนออกมา
- Salesman มีข้อมูลเพิ่มเติมคือ ยอดขาย (sale->double) อัตราคอมมิชชัน (commRate->double) และมีเมธอดคือ getSalary() ซึ่งคืนค่าเงินเดือนรวมกับค่าคอมมิชชันที่ได้

Big Java by Cay Horstmann
Copyright © 2009 by John Wiley & Sons. All rights reserved.

แบบฝึกหัดในห้อง

- Secretary มีข้อมูลเพิ่มเติมคือ ความเร็วในการพิมพ์ดีด (typing ->int) และมีเมธอดคือ getTyping() ซึ่งคืนค่าความเร็วในการพิมพ์ดีดออกมา
- ให้นิสิตสร้างคลาสเหล่านี้โดยกำหนด constructor ให้เหมาะสม
- สร้าง Test class เพื่อทดสอบคลาสที่สร้างขึ้น โดยสร้างอ็อบเจกต์จากคลาสดังกล่าว คลาสละ 1 อ็อบเจกต์ และลองเรียกใช้เมธอดต่าง ๆ เพื่อแสดงข้อมูลทั้งหมดของแต่ละอ็อบเจกต์

Copyright © 2009 by John Wiley & Sons. All rights reserved.

Exercise : Constructor Chaining

Constructing an instance of a class invokes all the **superclasses' constructors** along the inheritance chain. This is known as ***constructor chaining***.

```
public class Faculty extends Employee {
  public static void main(String[] args) {
    new Faculty();
  }

  public Faculty() {
    System.out.println("(4) Faculty's no-arg constructor is invoked");
  }
}

class Employee extends Person {
  public Employee() {
    this("(2) Invoke Employee's overloaded constructor");
    System.out.println("(3) Employee's no-arg constructor is invoked");
  }

  public Employee(String s) {
    System.out.println(s);
  }
}

class Person {
  public Person() {
    System.out.println("(1) Person's no-arg constructor is invoked");
  }
}
```

Protected Access

- Protected features can be accessed by all subclasses and by all classes in the same package
- Solves the problem that `CheckingAccount` methods need access to the `balance` instance variable of the superclass `BankAccount`:

```
public class BankAccount
{
   . . .
   protected double balance;
}
```

Big Java by Cay Horstmann
Copyright © 2009 by John Wiley & Sons. All rights reserved.

Protected Access

- The designer of the superclass has no control over the authors of subclasses:
  - *Any of the subclass methods can corrupt the superclass data*
  - *Classes with protected instance variables are hard to modify — the protected variables cannot be changed, because someone somewhere out there might have written a subclass whose code depends on them*
- Protected data can be accessed by all methods of classes in the same package
- It is best to leave all data private and provide accessor methods for the data

Big Java by Cay Horstmann
Copyright © 2009 by John Wiley & Sons. All rights reserved.

Access control

- From the CheckingAccount example
- What will happen??
  - If we use a shadowing variable : balance in CheckingAccount class
  - If we use a protected variable : balance in BankAccount class

BankAccount example

using shadowing instance field : balance in CheckingAccount class

```
public class CheckingAccount extends BankAccount  {
   private int transactionCount;
   private int balance;
   private static final int FREE_TRANSACTIONS = 3;
   private static final double TRANSACTION_FEE = 2.0;
   public CheckingAccount (int initialBalance)   {
      // construct superclass
      super (initialBalance);
      transactionCount = 0;
   }
   public void deposit (double amount)   {
      transactionCount++;
      //now add amount to balance
      balance += amount;
   }
```

```
public void withdraw(double amount)   {
    transactionCount++;
    // now subtract amount from balance
    balance -= amount;
  }

  /*Deducts the accumulated fees and resets the
   transaction count.*/
  public void deductFees()   {
    if (transactionCount > FREE_TRANSACTIONS)  {
      double fees = TRANSACTION_FEE *
        (transactionCount - FREE_TRANSACTIONS);
      balance -= fees;
    }
    transactionCount = 0;
  }
}
```

AccountTest class

- With the same AccountTest class
- The output is
  - Mom's savings balance = $10500.0
  - Harry's checking balance = $10000.0
- The balance that harrysChecking used in deposit, withdraw, deductFees is the balance in CheckingAccount class, not in BankAccount class. But when printing the value out, it called the getBalance() of BankAccount. So the value is the balance of BankAccount class which is still $10000.0

Access Control Level

Modifier on members in a class	Accessed from the same class	Accessed from the same package	Accessed from a subclass	Accessed from a different package
public	✓	✓	✓	✓
protected	✓	✓	✓	–
default	✓	✓	–	–
private	✓	–	–	–

BankAccount example

using protected instance field : balance in BankAccount class

```
//BankAccount.java
public class BankAccount
{
   protected double balance;

   public BankAccount()
   {
      balance = 0;
   }
    …
}
```

```
public class SavingsAccount extends BankAccount
{
    private double interestRate;
    public SavingsAccount(double rate)
    {
        interestRate = rate;
    }
    public void addInterest()
    {
        double interest = balance * interestRate / 100;
        balance += interest;
    }
}
```

```
//CheckingAccount.java
public class CheckingAccount extends BankAccount  {
   private int transactionCount;
   private static final int FREE_TRANSACTIONS = 3;
   private static final double TRANSACTION_FEE = 2.0;
   public CheckingAccount(int initialBalance)   {
      ///construct superclass
      super(initialBalance);
      transactionCount = 0;
   }
   public void deposit(double amount)   {
      transactionCount++;
      // now add amount to balance
      balance -= amount;
   }
```

```
    public void withdraw (double amount)   {
        transactionCount++;
        // now subtract amount from balance
        balance -= amount;
    }

    /*Deducts the accumulated fees and resets the
     transaction count.*/
    public void deductFees()   {
        if (transactionCount > FREE_TRANSACTIONS)  {
            double fees = TRANSACTION_FEE *
                (transactionCount - FREE_TRANSACTIONS);
            balance -= fees;
        }
        transactionCount = 0;
    }
}
```

AccountTest class

- With the same AccountTest class
- The output is

  Mom's savings balance = $10500.0

  Harry's checking balance = $8098.0
- The balance that harrysChecking used in deposit, withdraw, deductFees is the balance that is inherited from BankAccount class.

Using `protected` instance variables

- Advantages
  - subclasses can modify values directly
  - Slight increase in performance
    - Avoid set/get function call overhead
- Disadvantages
  - No validity checking
    - subclass can assign illegal value **ex** assign negative value

Using `protected` instance variables

- Implementation dependent
  - subclass methods more likely dependent on superclass implementation (normally, subclass should depend only on the superclass service -> non-private method)
  - superclass implementation changes may result in subclass modifications **ex** name of instance variable has been changed, we must correct all subclass which uses that instance variable
    - **Fragile (brittle) software because a small change in superclass can break subclass implementation**

Recommended Access Levels

- Encapsulation or Information hiding
  - Fields: Always *`private`*
  - Exception: *`public static final`* constants
  - Methods: *`public`* or *`private`*`(see purpose)`
  - Classes: *`public`* or *package*
  - Don't use `protected`
  - Beware of accidental package access (forgetting `public` or `private`)

From the CheckingAccount example : conclusion

- What will happen??
  - If we use a shadowing variable : balance in CheckingAccount class
    - ***Error in the value of the balance***
  - If we use a protected variable : balance in BankAccount class
    - ***OK but not encapsulation***

A Subclass Cannot Weaken the Accessibility

A subclass may override a protected method in its superclass and change its visibility to public. However, a subclass cannot weaken the accessibility of a method defined in the superclass. For example, if a method is defined as public in the superclass, it must be defined as public in the subclass.

The `final` Modifier

- The `final` class cannot be extended:

```
final class Math {
  ...
}
```

- The `final` variable is a constant:

```
final static double PI = 3.14159;
```

- The `final` method cannot be overridden by its subclasses.

Exercise

กำหนด class Animal มี source code ดังนี้

```
public class Animal {
    protected String name;
    public Animal(String name) {
        this.name = name;
    }
    public void eat(){
        System.out.println(name + "(Animal) eat");
    }
    public void walk(){
        System.out.println(name + "(Animal) walk");
    }
}
```

Big Java by Cay Horstmann
Copyright © 2009 by John Wiley & Sons. All rights reserved.

กำหนด class Human มี source code ดังนี้

```
public class Human extends Animal{
    public Human(String name) {
        super(name);
    }
}
```

Big Java by Cay Horstmann
Copyright © 2009 by John Wiley & Sons. All rights reserved.

กำหนด class Tester มี source code ดังนี้

```
public class Tester {
    public static void main(String[] args) {
        Animal a = new Animal("Ricky"); // สร้าง สัตว์ชื่อ Ricky
        a.walk(); // สั่งให้ Ricky เดิน
        a.eat(); // สั่งให้ Ricky กิน

        Human h = new Human("Somsak"); // สร้าง คนชื่อ สมศักดิ์
        h.walk(); // สั่งให้ สมศักดิ์เดิน
        h.eat(); // สั่งให้ สมศักดิ์กิน
    }
}
```

ดูผลลัพธ์ และทำความเข้าใจการสืบทอดและผลที่ได้รับจากการสืบทอด ของ Human จาก Animal และตอบคำถามต่อไปนี้ให้ได้

Big Java by Cay Horstmann
Copyright © 2009 by John Wiley & Sons. All rights reserved.

- keyword super ใน constructor Human คืออะไร
- protected แตกต่างจาก public และ private อย่างไร
- Human ได้รับ attribute และ method ใดบ้างที่สามารถเรียกใช้ใน Human ได้
- เหตุใดเมื่อใช้คำสั่ง

```
h.walk(); // สั่งให้ สมศักดิ์เดิน
h.eat(); // สั่งให้ สมศักดิ์กิน
```

ผลลัพธ์ที่ได้ จึงเป็น

Somsak(Animal) walk

Somsak(Animal) eat

และหากต้องการเปลี่ยนจาก (Animal) เป็น (Human) จะต้องทำอย่างไร จง แก้ไข class Human

Big Java by Cay Horstmann
Copyright © 2009 by John Wiley & Sons. All rights reserved.

•หากต้องการให้ Human มี method think() เพิ่มเติมจะต้องทำอย่างไร และให้เรียกใช้ method think() ในคลาส Tester ด้วย
โดยเมื่อแก้ไขทั้งหมดแล้วจะได้ผลลัพธ์ดังต่อไปนี้

```
Ricky(Animal) walk
Ricky(Animal) eat
Somsak(Human) walk
Somsak(Human) eat
Somsak(Human) think
```

เป็นได้หรือไม่ที่ Ricky จะ think หากดูจาก class ที่แก้ไขเสร็จแล้ว

Big Java by Cay Horstmann
Copyright © 2009 by John Wiley & Sons. All rights reserved.

Converting Between Subclass and Superclass Types

- OK to convert subclass reference to superclass reference:

```
SavingsAccount collegeFund = new SavingsAccount(10);
BankAccount anAccount = collegeFund;
Object anObject = collegeFund;
```

- The three object references stored in `collegeFund`, `anAccount`, and `anObject` all refer to the same object of type `SavingsAccount`

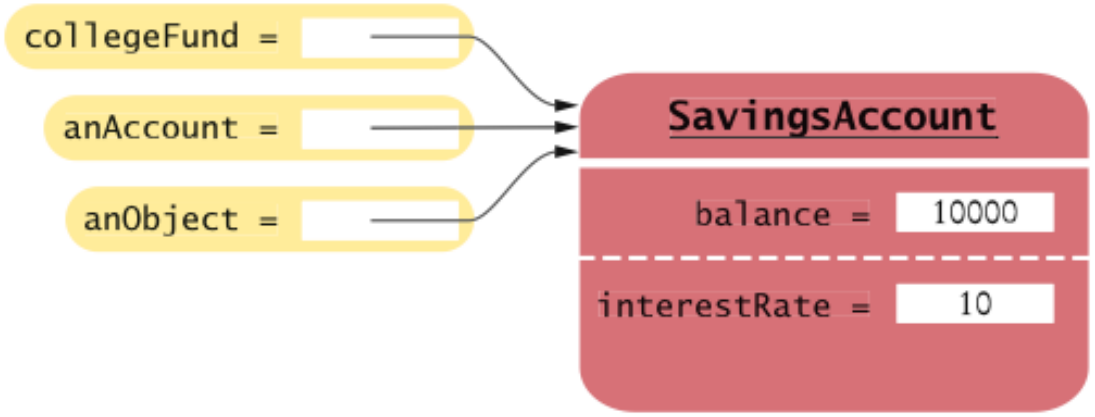


Figure 6
Variables of Different Types Can Refer to the Same Object

Big Java by Cay Horstmann
Copyright © 2009 by John Wiley & Sons. All rights reserved.

Converting Between Subclass and Superclass Types

- Superclass references don't know the full story:

```
anAccount.deposit(1000); // OK
anAccount.addInterest();
// No--not a method of the class to which anAccount
// belongs
```

- Why would anyone want to know *less* about an object?
  - *Reuse code that knows about the superclass but not the subclass:*

```
public void transfer(double amount, BankAccount other)
{
   withdraw(amount);
   other.deposit(amount);
}
```

Can be used to transfer money from any type of `BankAccount`

Big Java by Cay Horstmann
Copyright © 2009 by John Wiley & Sons. All rights reserved.

Converting Between Subclass and Superclass Types

- Occasionally you need to convert from a superclass reference to a subclass reference:

```
BankAccount anAccount = (BankAccount) anObject;
```

- This cast is dangerous: If you are wrong, an exception is thrown
- Solution: Use the `instanceof` operator
- `instanceof`: Tests whether an object belongs to a particular type (return true or false):

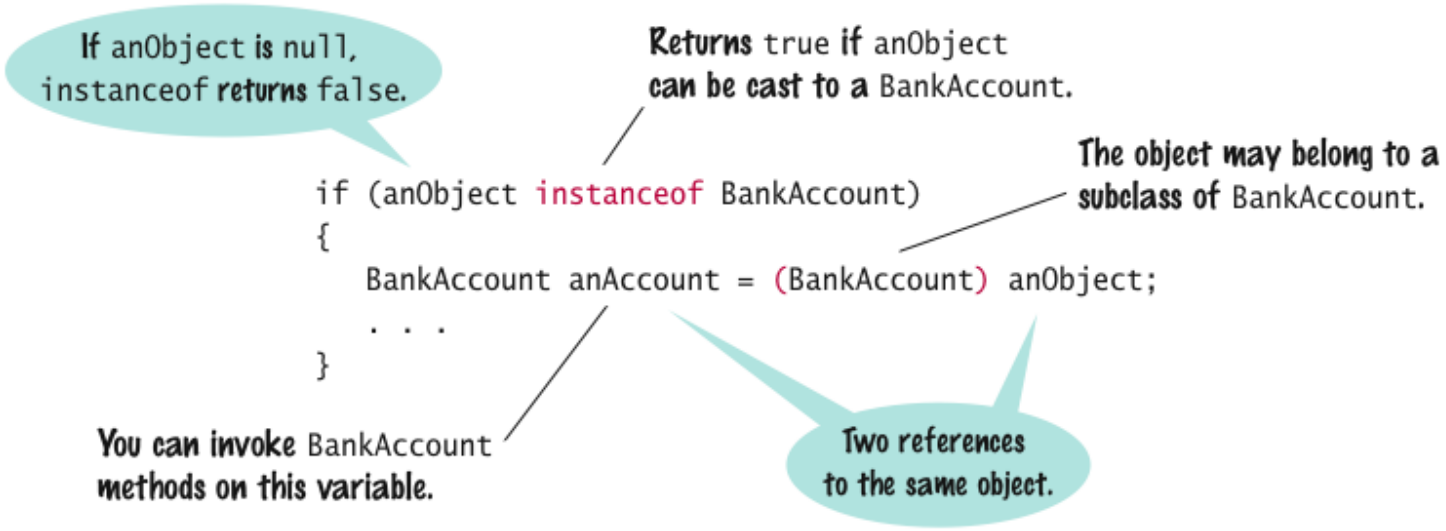
```
if (anObject instanceof BankAccount)
{
   BankAccount anAccount = (BankAccount) anObject;
   ...
}
```

Big Java by Cay Horstmann
Copyright © 2009 by John Wiley & Sons. All rights reserved.

Syntax 10.4 The `instanceof` Operator

Syntax *object* `instanceof` *TypeName*

Example



Big Java by Cay Horstmann
Copyright © 2009 by John Wiley & Sons. All rights reserved.

Self Check 10.11

Why did the second parameter of the `transfer` method have to be of type `BankAccount` and not, for example, `SavingsAccount`?

Big Java by Cay Horstmann
Copyright © 2009 by John Wiley & Sons. All rights reserved.

Self Check 10.12

Why can't we change the second parameter of the `transfer` method to the type `Object`?

Big Java by Cay Horstmann
Copyright © 2009 by John Wiley & Sons. All rights reserved.

From in class exercise

superclass Employee and subclass Salesman, Secretary

- See the test class
- มีข้อผิดพลาดหรือไม่
- ถ้ามี จงบอกว่าผิดเพราะเหตุใด และจะแก้ให้ถูกต้องได้อย่างไร
- ถ้าไม่มี จงแสดงผลลัพธ์

Test class

```
Employee e = new Employee(“Sasipa Pant”, 2000, 25000);
Salesman s = new Salesman(“Somying Meejai”, 2005,
                12500, 150000, 0.05);
Secretary c = new Secretary(“Somjai Deejing”, 2008,
                20000, 60);
Employee ec = c;
System.out.println(“call getName() from ec = ” +
                ec.getName());
System.out.println (“call getTyping() from ec = ” +
                ec.getTyping());
Salesman se = e;
System.out.println (“call getName() from se = ” +
                se.getName());
System.out.println (“call getSalary() from se = ” +
                se.getSalary());
```

Correct??

```
Employee eess = s;
Salesman se =(Salesman) eess;
System.out.println("call getName() from eee= " + se.getName());
System.out.println("call getSalary() from eee= " +
se.getSalary());
```

สรุป assign reference

Converting Between Subclass and Superclass Types

- Assign ref ระดับ subclass ให้ superclass ref ได้
- ชนิดsuperclass ชื่อsuperclassref = subclassref

```
SavingsAccount collegeFund = new SavingsAccount(10);
BankAccount anAccount = collegeFund;
```

- แต่เรียกเมธอดของ subclass ผ่าน superclassref ไม่ได้ เพราะตอนเราเขียนโปรแกรมระดับ superclass มันยังไม่รู้จักเมธอดเหล่านั้น

สรุป assign reference

- แต่ในทางกลับกันจะ assign superclassref ให้กับตัวแปรชนิด subclass ไม่ได้ เพราะไม่ถูกต้องตามหลักการถ่ายทอดคุณสมบัติ จะผิด syntax ของภาษา
- SavingsAccount savings;
- savings = anAccount;
- แต่ถ้าเรารู้ว่า ก้อนออบเจกต์จริงมันเป็นชนิดของ subclass เราสามารถทำได้โดยการทำ downcasting
- savings = (savingsAccount)anAccount;

สรุป assign reference

- การ assign reference และต้องทำ casting นี้ จะเกิดเมื่อเราจะต้องใช้เมธอดที่สร้างโดย superclass ซึ่งตอนนั้น คนเขียนโปรแกรมระดับ superclass ยังไม่รู้จักลูก ๆ ตัวเองมาก่อน ก็เขียนโดยใช้ออบเจ็กต์ในระดับของตัวเอง
- พอเกิดลูกหลาน
  - ลูกหลานจะเรียกใช้เมธอดที่รับมาจากพ่อ ซึ่งของพ่อยังใช้แต่ตัวแปรในระดับ superclass แต่เราส่งตัวแปรระดับ subclass ไปให้พ่อ ดังตัวอย่าง transfer() ใช้ superclass reference แต่ก้อนจริงเป็นของ subclass
  - ลูกหลานจะมาเอาของพ่อมาปรับปรุง แต่ยังต้องใช้หัวเมธอดที่เหมือนของพ่อ (เพราะเป็นการทำ override) ลูกหลานก็ต้องทำการ assign reference กันไปมา และ casting ด้วยดังตัวอย่าง equals() และ clone() ที่จะเรียนต่อไป

Polymorphism and Inheritance

- In object-oriented programming, **polymorphism** refers to a programming language's **ability to process objects differently depending on their data type or class**. More specifically, it is the ability to redefine methods for derived classes. (**A single interface to entities of different types)**
- Type of a variable doesn't completely determine type of object to which it refers:

```
BankAccount aBankAccount = new SavingsAccount(1000);
// aBankAccount holds a reference to a SavingsAccount
```

-
```
BankAccount anAccount = new CheckingAccount();
anAccount.deposit(1000);
```

Which deposit method is called?

- *Dynamic method lookup:* When the virtual machine calls an instance method, it locates the method of the implicit parameter's class

Big Java by Cay Horstmann
Copyright © 2009 by John Wiley & Sons. All rights reserved.

Polymorphism and Inheritance

- Example:

```
public void transfer(double amount, BankAccount other)
{
   withdraw(amount);
   other.deposit(amount);
}
```

- When you call

```
anAccount.transfer(1000, anotherAccount);
```

two method calls result:

```
anAccount.withdraw(1000);
anotherAccount.deposit(1000);
```

Big Java by Cay Horstmann
Copyright © 2009 by John Wiley & Sons. All rights reserved.

Polymorphism and Inheritance

- *Polymorphism:* Ability to treat objects with differences in behavior in a uniform way
- The first method call

  ```
  withdraw(amount);
  ```

  is a shortcut for

  ```
  this.withdraw(amount);
  ```

- `this` can refer to a `BankAccount` or a subclass object

Big Java by Cay Horstmann
Copyright © 2009 by John Wiley & Sons. All rights reserved.

ch10/accounts/AccountTester.java

```
1  /**
2     This program tests the BankAccount class and
3     its subclasses.
4  */
5  public class AccountTester
6  {
7     public static void main(String[] args)
8     {
9        SavingsAccount momsSavings = new SavingsAccount(0.5);
10
11       CheckingAccount harrysChecking = new CheckingAccount(100);
12
13       momsSavings.deposit(10000);
14
15       momsSavings.transfer(2000, harrysChecking);
16       harrysChecking.withdraw(1500);
17       harrysChecking.withdraw(80);
18
19       momsSavings.transfer(1000, harrysChecking);
20       harrysChecking.withdraw(400);
21
```

Continued

Big Java by Cay Horstmann
Copyright © 2009 by John Wiley & Sons. All rights reserved.

ch10/accounts/AccountTester.java (cont.)

```
22          // Simulate end of month
23          momsSavings.addInterest();
24          harrysChecking.deductFees();
25
26          System.out.println("Mom's savings balance: "
27                + momsSavings.getBalance());
28          System.out.println("Expected: 7035");
29
30          System.out.println("Harry's checking balance: "
31                + harrysChecking.getBalance());
32          System.out.println("Expected: 1116");
33       }
34    }
```

Program Run:

```
Mom's savings balance: 7035.0
Expected: 7035
Harry's checking balance: 1116.0
Expected: 1116
```

Big Java by Cay Horstmann
Copyright © 2009 by John Wiley & Sons. All rights reserved.

Self Check 10.13

If `a` is a variable of type `BankAccount` that holds a non-`null` reference, what do you know about the object to which `a` refers?

Big Java by Cay Horstmann
Copyright © 2009 by John Wiley & Sons. All rights reserved.

Self Check 10.14

If `a` refers to a checking account, what is the effect of calling `a.transfer(1000, a)`?

Answer: The balance of `a` is unchanged, and the transaction count is incremented twice.

Big Java by Cay Horstmann
Copyright © 2009 by John Wiley & Sons. All rights reserved.

`Object`: The Cosmic Superclass

- All classes defined without an explicit `extends` clause automatically extend `Object`:

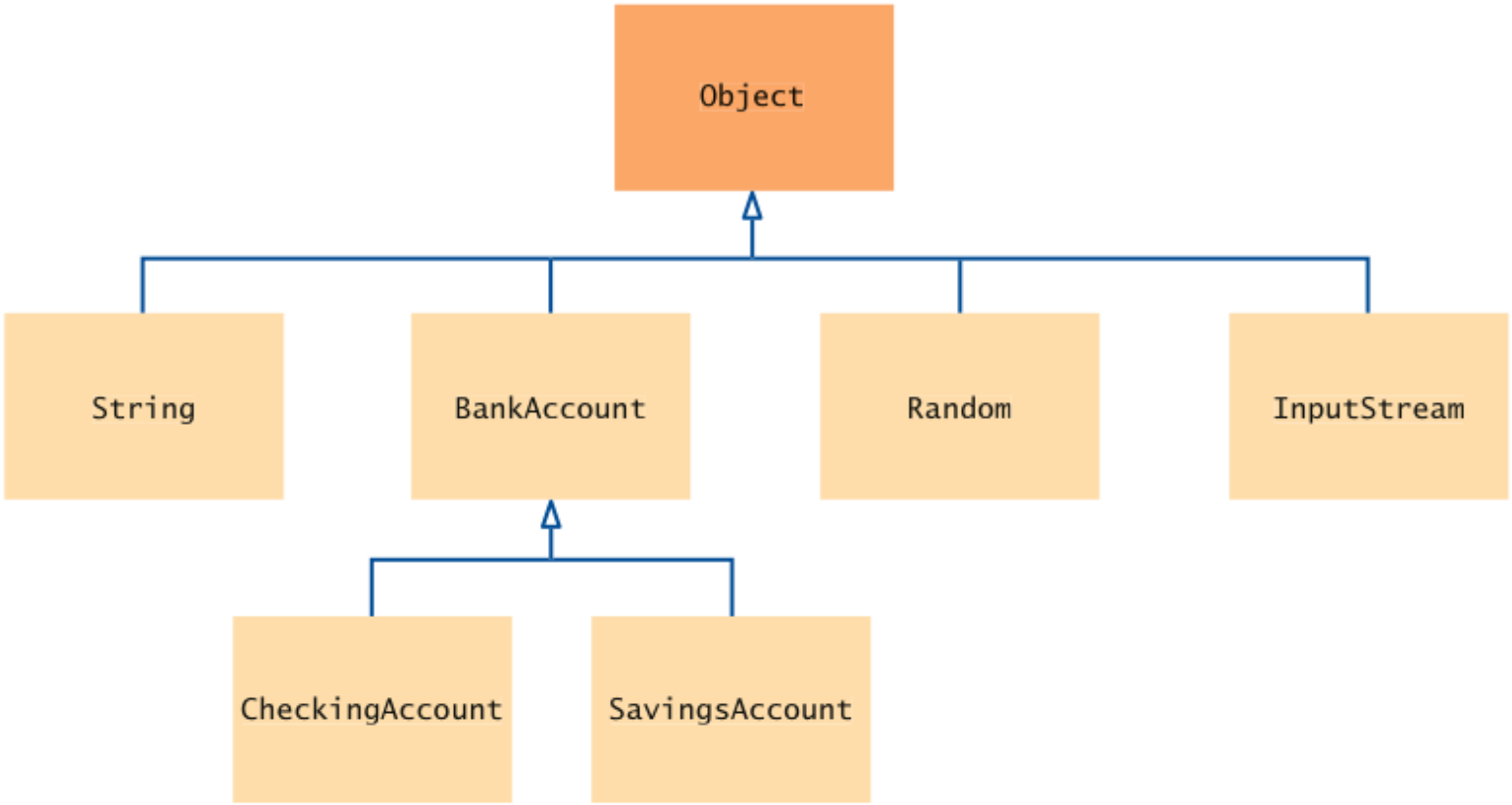


Figure 7 The `Object` Class Is the Superclass of Every Java Class

Big Java by Cay Horstmann
Copyright © 2009 by John Wiley & Sons. All rights reserved.

`Object`: The Cosmic Superclass

- Most useful methods:
  - `String toString()`
  - `boolean equals(Object otherObject)`
  - `Object clone()`
- Good idea to override these methods in your classes

Big Java by Cay Horstmann
Copyright © 2009 by John Wiley & Sons. All rights reserved.

Overriding the `toString` Method

- Returns a string representation of the object
- Useful for debugging:

```
Rectangle box = new Rectangle(5, 10, 20, 30);
String s = box.toString();
// Sets s to "java.awt.Rectangle[x=5,y=10,width=20,
// height=30]"
```

- `toString` is called whenever you concatenate a string with an object:

```
"box=" + box;
// Result: "box=java.awt.Rectangle[x=5,y=10,width=20,
// height=30]"
```

Big Java by Cay Horstmann
Copyright © 2009 by John Wiley & Sons. All rights reserved.

Overriding the `toString` Method

- `Object.toString` prints class name and the *hash code* of the object:

```
BankAccount momsSavings = new BankAccount(5000);
String s = momsSavings.toString();
// Sets s to something like "BankAccount@d24606bf"
```

Big Java by Cay Horstmann
Copyright © 2009 by John Wiley & Sons. All rights reserved.

Overriding the `toString` Method

- To provide a nicer representation of an object, override `toString`:

```
public String toString()
{
   return "BankAccount[balance=" + balance + "]";
}
```

- This works better:

```
BankAccount momsSavings = new BankAccount(5000);
String s = momsSavings.toString();
// Sets s to "BankAccount[balance=5000]"
```

Big Java by Cay Horstmann
Copyright © 2009 by John Wiley & Sons. All rights reserved.

- In SavingsAccount class, override method

```
public String toString()
{
    return super.toString() +
        “\n[interestRate = ” +
        interestRate + “]”;
}
```

- In CheckingAccount class, override method

```
public String toString()
{
    return super.toString() +
            "\n[transactionCount = " +
            transactionCount + "]";
}
```

In AccountTest class add

```
System.out.println(momsSavings);
System.out.println();
System.out.println(harrysChecking);
System.out.println();
```

Or we can write
System.out.println(momsSavings.toString());

- After harrysChecking transactions
- Before //simulate end of month

```
//AccountTest.java
public class AccountTest
{
   public static void main(String[] args)
   {
      SavingsAccount momsSavings
         = new SavingsAccount(5);

      CheckingAccount harrysChecking
         = new CheckingAccount(10000);

      momsSavings.deposit(10000);

      harrysChecking.withdraw(1500);
      harrysChecking.deposit(1500);
      harrysChecking.withdraw(1500);
      harrysChecking.withdraw(400);
```

```
        System.out.println(momsSavings);
        System.out.println();
        System.out.println(harrysChecking);
        System.out.println();

        // simulate end of month
        momsSavings.addInterest();
        harrysChecking.deductFees();

        System.out.println("Mom's savings balance = $"
           + momsSavings.getBalance());

        System.out.println("Harry's checking balance = $"
           + harrysChecking.getBalance());
    }
}
```

Output

BankAccount[balance = 10000.0]
[interestRate = 5.0]

BankAccount[balance = 8100.0]
[transactionCount = 4]

Mom's savings balance = $10500.0
Harry's checking balance = $8098.0

Improving the toString method

- instead of hard-coding the classname in the toString method that you override, you should call the getClass method to obtain a class object and then, invoke the getName method to get the name of the class
- getClass() is in Object class

  (return an object of class Class)
- getName() is in Class class

  (return name of class)

Study more -> getSimpleName()

Edit toString() method in BankAccount class

- In BankAccount class change

```
public String toString()
  {
     return getClass().getName() +
                "[balance = " + balance + "]";
  }
```

New output

SavingsAccount[balance = 10000.0]
[interestRate = 5.0]

CheckingAccount[balance = 8100.0]
[transactionCount = 4]

Mom's savings balance = $10500.0
Harry's checking balance = $8098.0

Polymorphism, Dynamic Binding and Generic Programming

```
public class PolymorphismDemo {
  public static void main(String[] args) {
    m(new GraduateStudent());
    m(new Student());
    m(new Person());
    m(new Object());
  }

  public static void m(Object x) {
    System.out.println(x.toString());
  }
}

class GraduateStudent extends Student {
}

class Student extends Person {
  public String toString() {
    return "Student";
  }
}

class Person extends Object {
  public String toString() {
    return "Person";
  }
}
```

Method m takes a parameter of the Object type. You can invoke it with any object.

An object of a subtype can be used wherever its supertype value is required. This feature is known as *polymorphism*.

What is the output??

When the method m(Object x) is executed, the argument x's toString method is invoked. x may be an instance of GraduateStudent, Student, Person, or Object. Classes GraduateStudent, Student, Person, and Object have their own implementation of the toString method. Which implementation is used will be determined dynamically by the Java Virtual Machine at runtime. This capability is known as *dynamic binding*.

Overriding the `equals` Method

- `equals` tests for same *contents*:

```
if (coin1.equals(coin2)) . . .
// Contents are the same
```

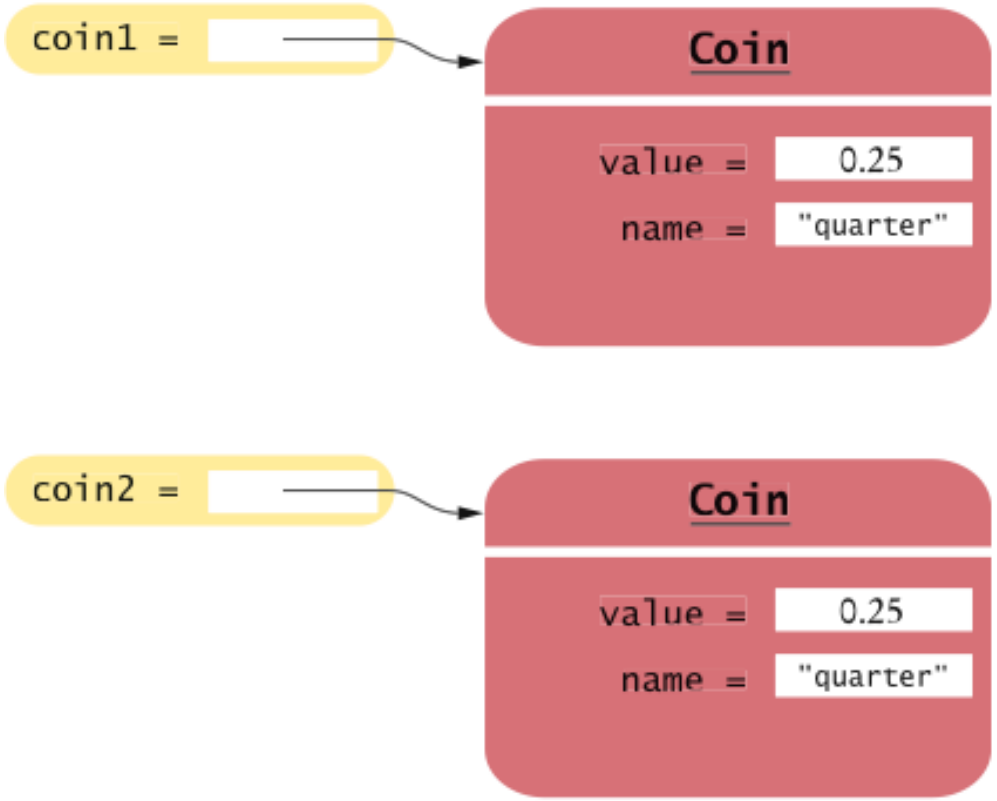


Figure 8 Two References to Equal Objects

Big Java by Cay Horstmann
Copyright © 2009 by John Wiley & Sons. All rights reserved.

Overriding the `equals` Method

- `==` tests for references to the same object:

```
if (coin1 == (coin2)) . . .
// Objects are the same
```

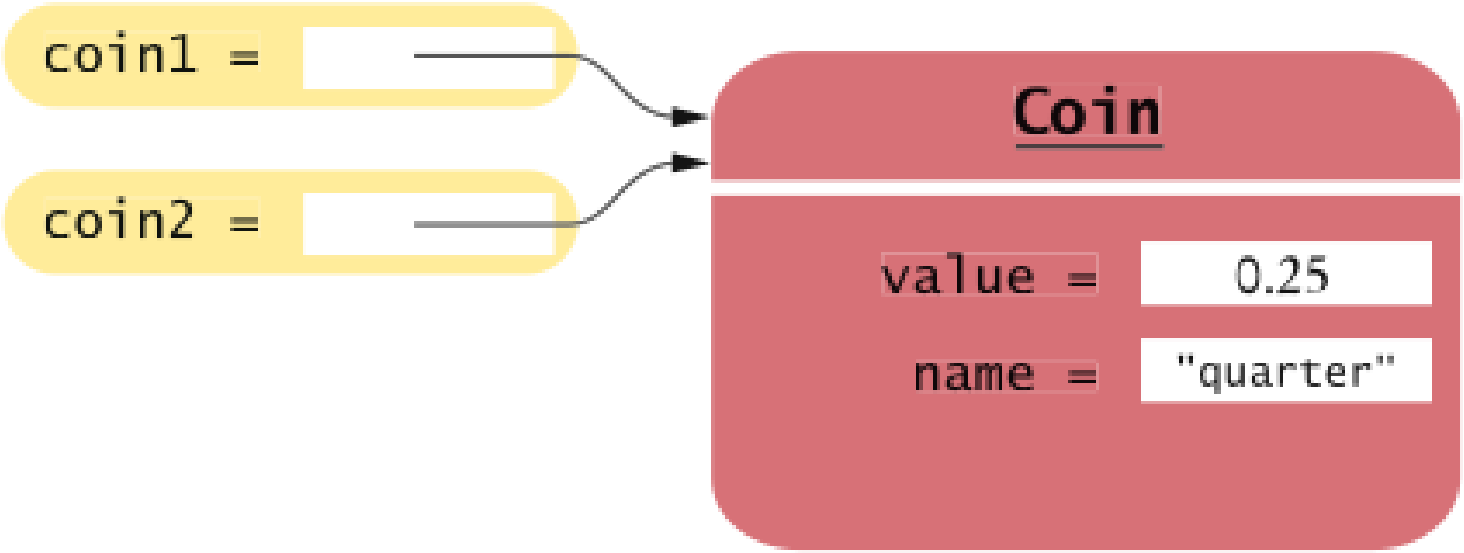


Figure 9 Two References to the Same Object

Big Java by Cay Horstmann
Copyright © 2009 by John Wiley & Sons. All rights reserved.

Overriding the `equals` Method

- Need to override the `equals` method of the `Object` class:

```
public class Coin
{
   ...
   public boolean equals(Object otherObject)
   {
      ...
   }
   ...
}
```

Big Java by Cay Horstmann
Copyright © 2009 by John Wiley & Sons. All rights reserved.

Overriding the `equals` Method

- Cannot change parameter type; use a *cast* instead:

```
public class Coin
{
   ...
   public boolean equals(Object otherObject)
   {
      Coin other = (Coin) otherObject;
      return name.equals(other.name) && value ==
         other.value;
   }
   ...
}
```

```
public class BankAccount {
 public boolean equals(Object otherObject) {
  BankAccount other = (BankAccount)otherObject;
  return balance == other.balance;
 }
}
```

Big Java by Cay Horstmann
Copyright © 2009 by John Wiley & Sons. All rights reserved.

File Coin.java
override equals()

```
public class Coin
{
   private double value;
   private String name;

   public Coin(double aValue, String aName)
   {
      value = aValue;
      name = aName;
   }
   public double getValue()
   {
      return value;
   }
```

```
   public String getName()
   {
      return name;
   }

   public boolean equals(Object otherObject)
   {
      Coin other = (Coin)otherObject;
      return name.equals(other.name) && value==other.value;
   }
}
```

```
public class Test {
    public static void main(String[] args) {
        Coin c1 = new Coin(10, "Gold");
        Coin c2 = new Coin(10, "Gold");
        Coin c3 = new Coin(5, "Silver");
        if (c1.equals(c2))
            System.out.println(c1.getName() + " equals " +
                                              c2.getName());
        else
            System.out.println("Unequal");
        if (c2.equals(c3))
            System.out.println(c2.getName() + " equals " +
                                              c3.getName());
        else
            System.out.println("Unequal");
    }
}
```

Improving the equals method

```
public boolean equals(Object otherObject)
      {
        if (otherObject == null)
             return false;
        if (getClass()!=otherObject.getClass())
             return false;
         Coin other = (Coin)otherObject;
         return name.equals(other.name)
                   && value == other.value;
      }
```

The `clone` Method

- Copying an object reference gives two references to same object:

```
BankAccount account = newBankAccount(1000);
BankAccount account2 = account;
account2.deposit(500); // Now both account and account2
   // refer to a bank account with a balance of 1500
```

- Sometimes, need to make a copy of the object:

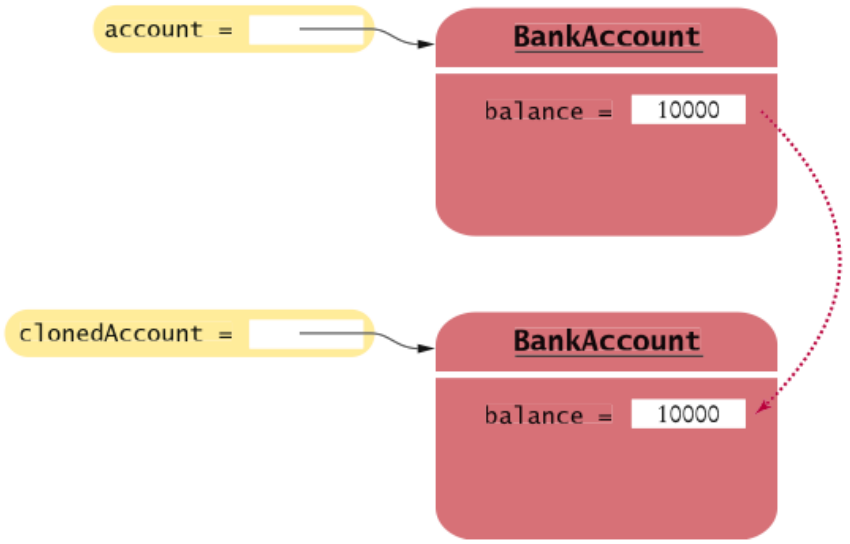


Figure 10
Cloning Objects

Big Java by Cay Horstmann
Copyright © 2009 by John Wiley & Sons. All rights reserved.

The `clone` Method

- Implement `clone` method to make a new object with the same state as an existing object
- Use `clone`:

```
BankAccount clonedAccount =
   (BankAccount) account.clone();
```

- Must cast return value because return type is `Object`

Big Java by Cay Horstmann
Copyright © 2009 by John Wiley & Sons. All rights reserved.

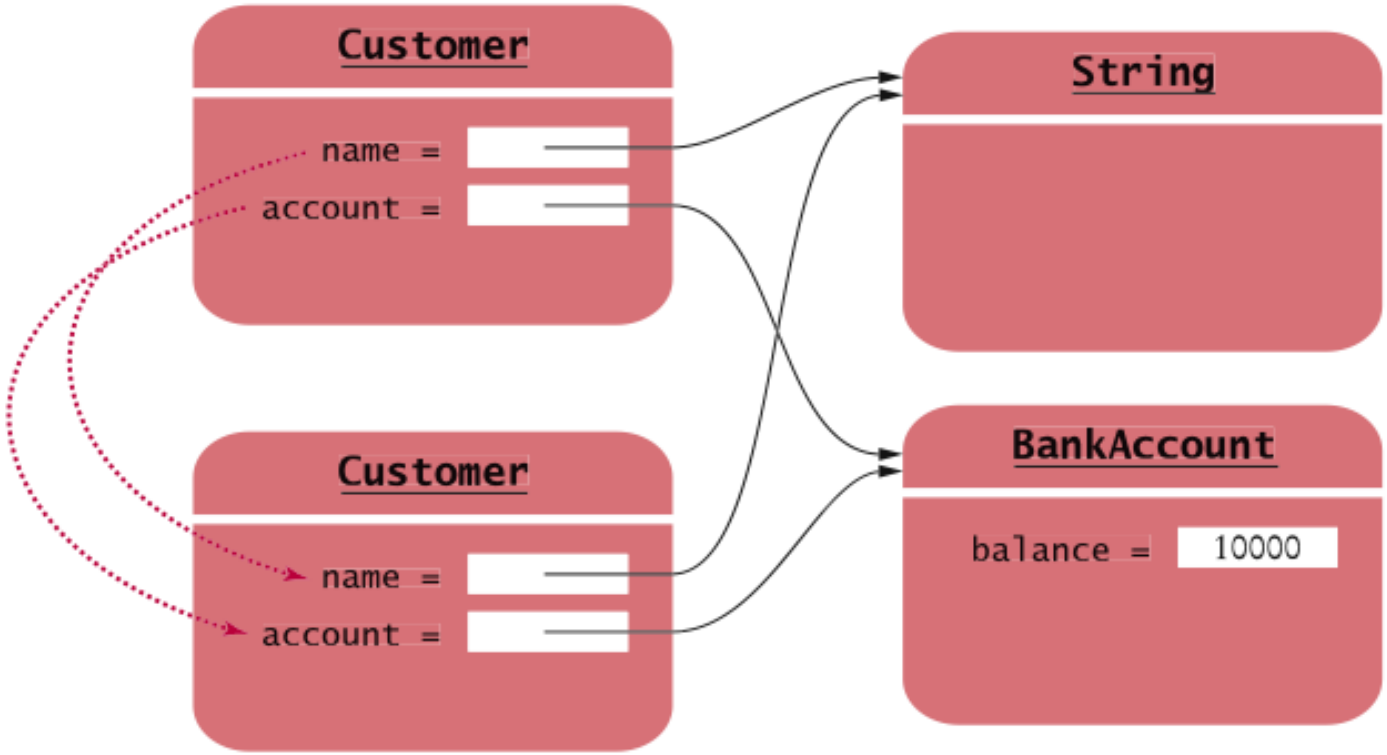
```
public class BankAccount implements Cloneable  {
    private double balance;
    public BankAccount()    {
      // by default the balance value = 0
                // balance = 0 ;
    }
    public BankAccount(double initialBalance)    {
      balance = initialBalance;
    }
    public void deposit(double amount)     {
      balance = balance + amount;
    }
    public void withdraw(double amount)     {
      balance = balance - amount;
    }
    public double getBalance()    {
      return balance;
    }
    public Object clone()     {
        BankAccount a = new BankAccount();
        a.balance = this.balance;
        return a;
    }
}
```

```
package bankacctclone;
public class BankAcctClone {
public static void main(String[] args) {
        BankAccount sasipa = new BankAccount(15000);
        BankAccount clonedSasipa = (BankAccount)sasipa.clone();
        System.out.println(clonedSasipa.getBalance());
        System.out.println(sasipa);
        System.out.println(clonedSasipa);
    }

}
```

```
run:
15000.0
bankacctclone.BankAccount@15db9742
bankacctclone.BankAccount@6d06d69c
```

The `Object.clone` Method

- Creates *shallow copies*:



The `Object.clone` Method Makes a Shallow Copy

Big Java by Cay Horstmann
Copyright © 2009 by John Wiley & Sons. All rights reserved.

The `Object.clone` Method

- Does not systematically clone all subobjects
- Must be used with caution
- It is declared as `protected`; prevents from accidentally calling `x.clone()` if the class to which `x` belongs hasn't redefined `clone` to be `public`
- You should override the `clone` method with care (see Special Topic 10.6 -> implements Cloneable and try-catch statement)
- Study by yourself

Big Java by Cay Horstmann
Copyright © 2009 by John Wiley & Sons. All rights reserved.

Self Check 10.15

Should the call `x.equals(x)` always return `true`?

Big Java by Cay Horstmann
Copyright © 2009 by John Wiley & Sons. All rights reserved.

Self Check 10.16

Can you implement `equals` in terms of `toString`? Should you?

Big Java by Cay Horstmann
Copyright © 2009 by John Wiley & Sons. All rights reserved.

Using Inheritance to Customize Frames

- Use inheritance for complex frames to make programs easier to understand
- Design a subclass of `JFrame`
- Store the components as instance variables
- Initialize them in the constructor of your subclass
- If initialization code gets complex, simply add some helper methods

Big Java by Cay Horstmann
Copyright © 2009 by John Wiley & Sons. All rights reserved.

ch10/frame/InvestmentFrame.java

```
 1  import java.awt.event.ActionEvent;
 2  import java.awt.event.ActionListener;
 3  import javax.swing.JButton;
 4  import javax.swing.JFrame;
 5  import javax.swing.JLabel;
 6  import javax.swing.JPanel;
 7  import javax.swing.JTextField;
 8
 9  public class InvestmentFrame extends JFrame
10  {
11     private JButton button;
12     private JLabel label;
13     private JPanel panel;
14     private BankAccount account;
15
16     private static final int FRAME_WIDTH = 400;
17     private static final int FRAME_HEIGHT = 100;
18
19     private static final double INTEREST_RATE = 10;
20     private static final double INITIAL_BALANCE = 1000;
21
```

Continued

Big Java by Cay Horstmann
Copyright © 2009 by John Wiley & Sons. All rights reserved.

ch10/frame/InvestmentFrame.java

```
22      public InvestmentFrame()
23      {
24         account = new BankAccount(INITIAL_BALANCE);
25
26         // Use instance variables for components
27         label = new JLabel("balance: " + account.getBalance());
28
29         // Use helper methods
30         createButton();
31         createPanel();
32
33         setSize(FRAME_WIDTH, FRAME_HEIGHT);
34      }
35
36      private void createButton()
37      {
38         button = new JButton("Add Interest");
39         ActionListener listener = new AddInterestListener();
40         button.addActionListener(listener);
41      }
42
```

Continued

Big Java by Cay Horstmann
Copyright © 2009 by John Wiley & Sons. All rights reserved.

Example: Investment Viewer Program (cont.)

```
43      private void createPanel()
44      {
45         panel = new JPanel();
46         panel.add(button);
47         panel.add(label);
48         add(panel);
49      }
50
51      class AddInterestListener implements ActionListener
52      {
53         public void actionPerformed(ActionEvent event)
54         {
55            double interest = account.getBalance() * INTEREST_RATE / 100;
56            account.deposit(interest);
57            label.setText("balance: " + account.getBalance());
58         }
59      }
60   }
```

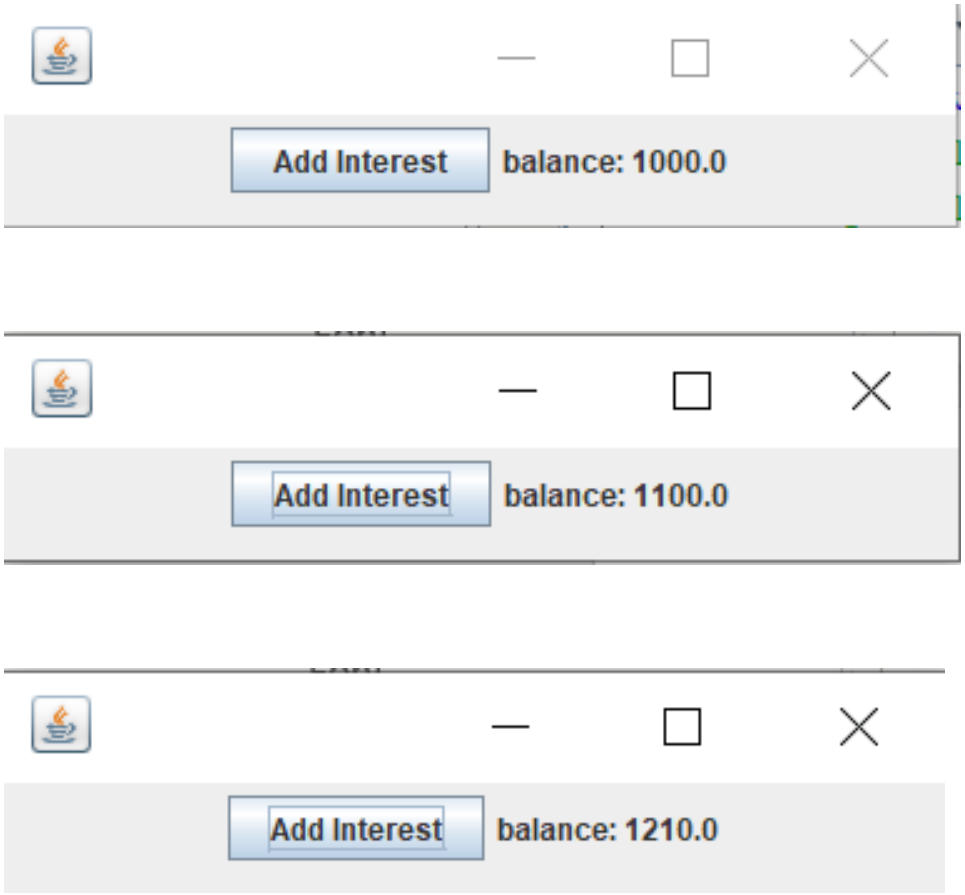
Big Java by Cay Horstmann
Copyright © 2009 by John Wiley & Sons. All rights reserved.

Example: Investment Viewer Program

Of course, we still need a class with a `main` method:

```
 1  import javax.swing.JFrame;
 2
 3  /**
 4     This program displays the growth of an investment.
 5  */
 6  public class InvestmentFrameViewer
 7  {
 8     public static void main(String[] args)
 9     {
10        JFrame frame = new InvestmentFrame();
11        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
12        frame.setVisible(true);
13     }
14  }
```

Big Java by Cay Horstmann
Copyright © 2009 by John Wiley & Sons. All rights reserved.



Big Java by Cay Horstmann
Copyright © 2009 by John Wiley & Sons. All rights reserved.

Self Check 10.17

How many Java source files are required by the investment viewer application when we use inheritance to define the frame class?

Answer: Three: `InvestmentFrameViewer`, `InvestmentFrame`, and `BankAccount`.

Big Java by Cay Horstmann
Copyright © 2009 by John Wiley & Sons. All rights reserved.

Self Check 10.18

Why does the `InvestmentFrame` constructor call `setSize(FRAME_WIDTH, FRAME_HEIGHT)`, whereas the `main` method of the investment viewer class in Chapter 9 called `frame.setSize(FRAME_WIDTH, FRAME_HEIGHT)`?

Answer: The `InvestmentFrame` constructor adds the panel to *itself*.

Big Java by Cay Horstmann
Copyright © 2009 by John Wiley & Sons. All rights reserved.

Exercise

Add a TimeDepositAccount class to the bank account hierarchy. The time deposit account is just like a savings account, but you promise to leave the money in the account for a particular number of months, and there is a $20 penalty for early withdrawal. Construct the account with the interest rate and the number of months to maturity. In the addInterest method, decrement the count of months. If the count is positive during a withdrawal, charge the withdrawal penalty.

Big Java by Cay Horstmann
Copyright © 2009 by John Wiley & Sons. All rights reserved.