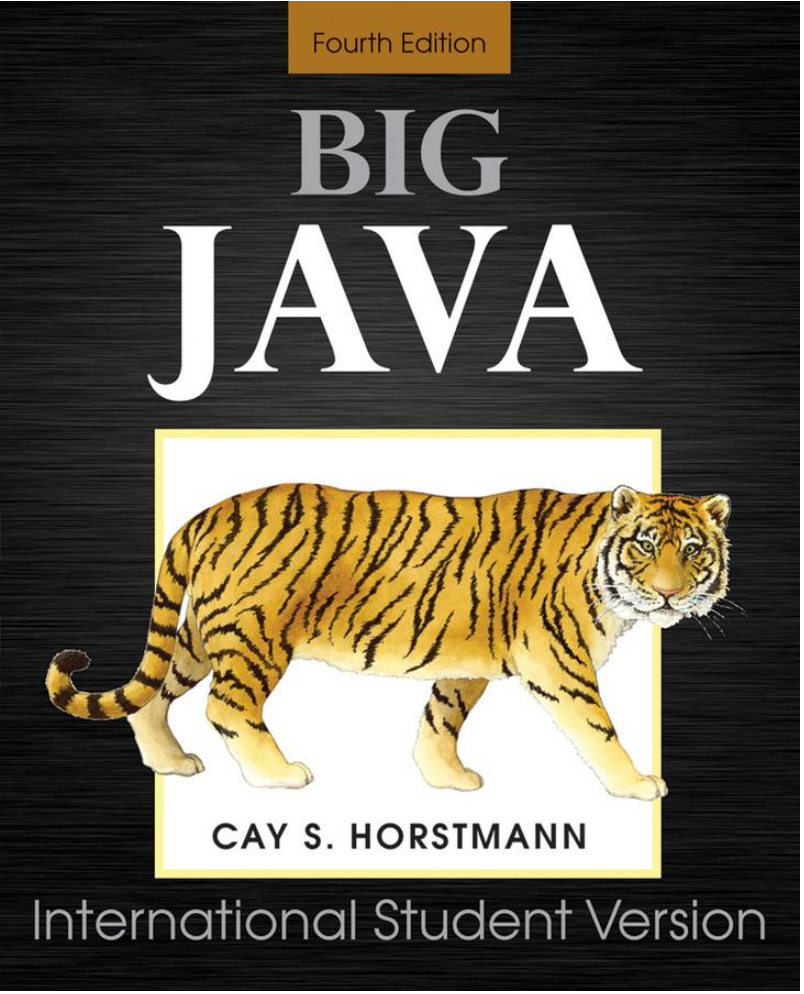


Chapter 9 – Inheritance

Chapter Goals

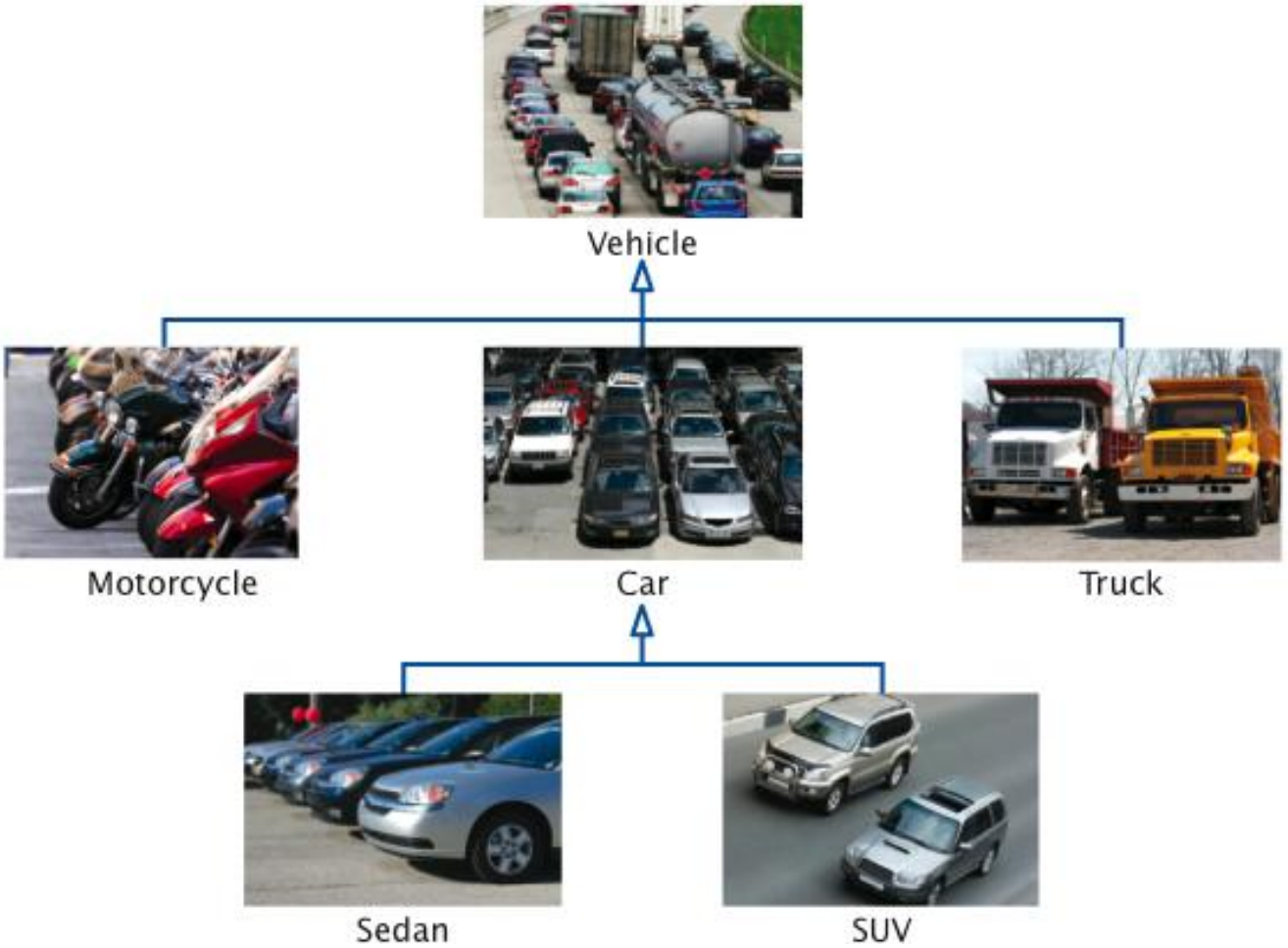
- To learn about inheritance
- To understand how to inherit and override superclass methods
- To be able to invoke superclass constructors
- To learn about `protected` and package access control
- To understand the common superclass `Object` and to override its `toString` and `equals` methods

G To use inheritance for customizing user interfaces

Inheritance Hierarchies

- Often categorize concepts into *hierarchies*:

Figure 1
A Hierarchy of Vehicle Types



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:*

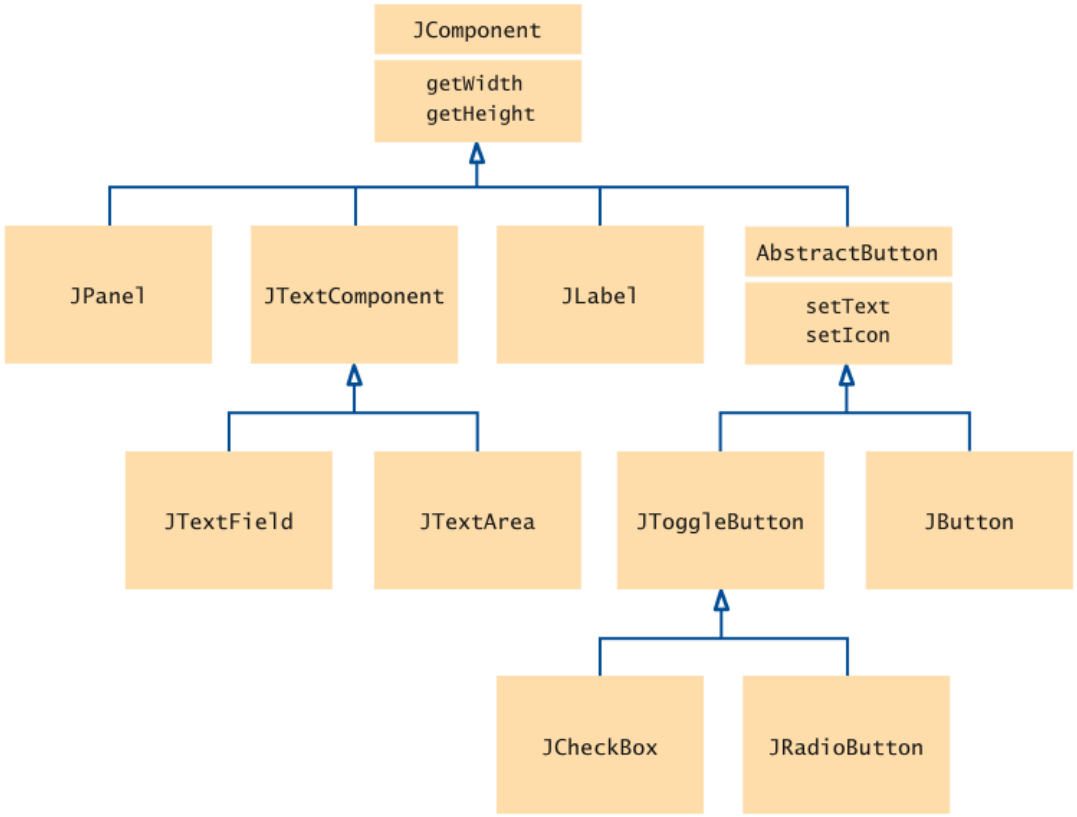


Figure 2 A Part of the Hierarchy of Swing User Interface Components

Inheritance Hierarchies

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

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`

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*

Inheritance Hierarchies

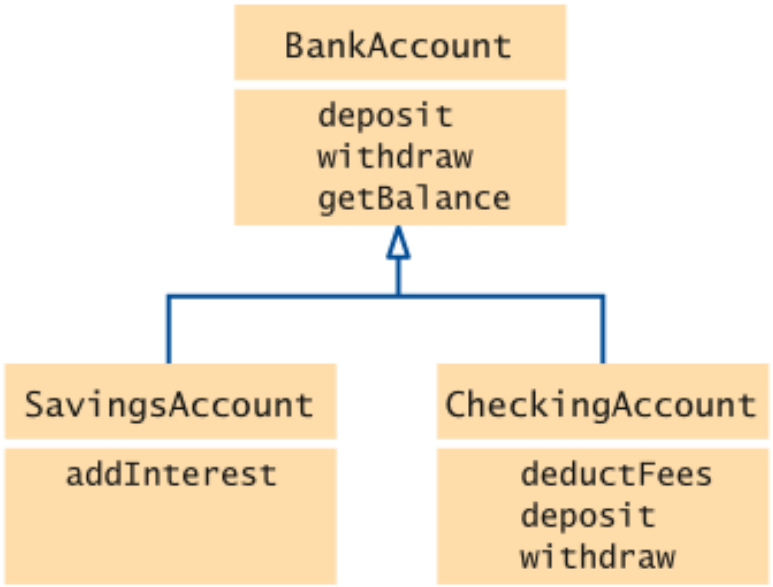


Figure 3 Inheritance Hierarchy for Bank Account Classes

Self Check 9.1

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

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

Self Check 9.2

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

Answer: Not all bank accounts earn interest.

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
}
```

- 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
```

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
   }
}
```

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`:

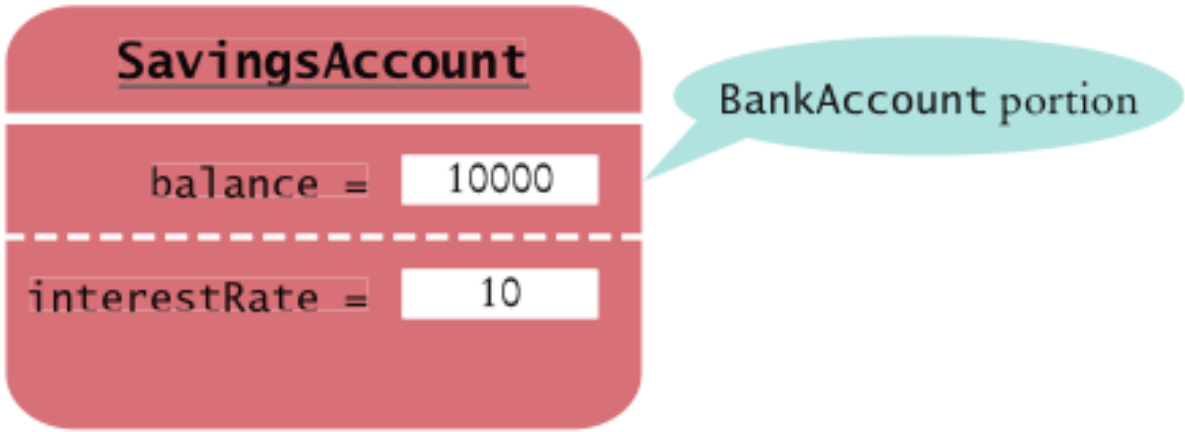


Figure 4
Layout of a Subclass Object

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);
   }
}
```

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)
- Inheriting from a class differs from implementing an interface: the subclass inherits behavior from the superclass

ch09/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

ch09/accounts/SavingsAccount.java (cont.)

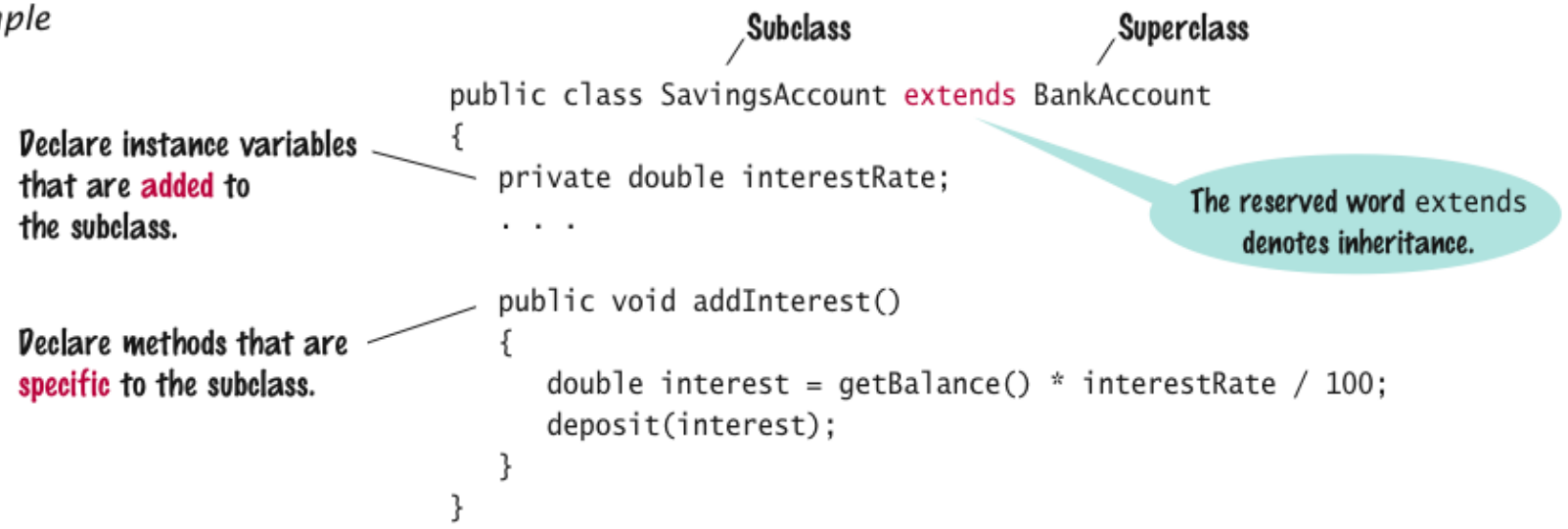
```
17      /**
18         Adds the earned interest to the account balance.
19      */
20      public void addInterest()
21      {
22         double interest = getBalance() * interestRate / 100;
23         deposit(interest);
24      }
25   }
```

Syntax 9.1 Inheritance

Syntax

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

Example



Self Check 9.3

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

Answer: Two instance variables: `balance` and `interestRate`.

Self Check 9.4

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

Answer: `deposit`, `withdraw`, `getBalance`, and `addInterest`.

Self Check 9.5

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

Answer: `Manager` is the subclass; `Employee` is the superclass.

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
   }
   . . .
}
```

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
   }
   . . .
}
```

Common Error: Shadowing Instance Variables

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

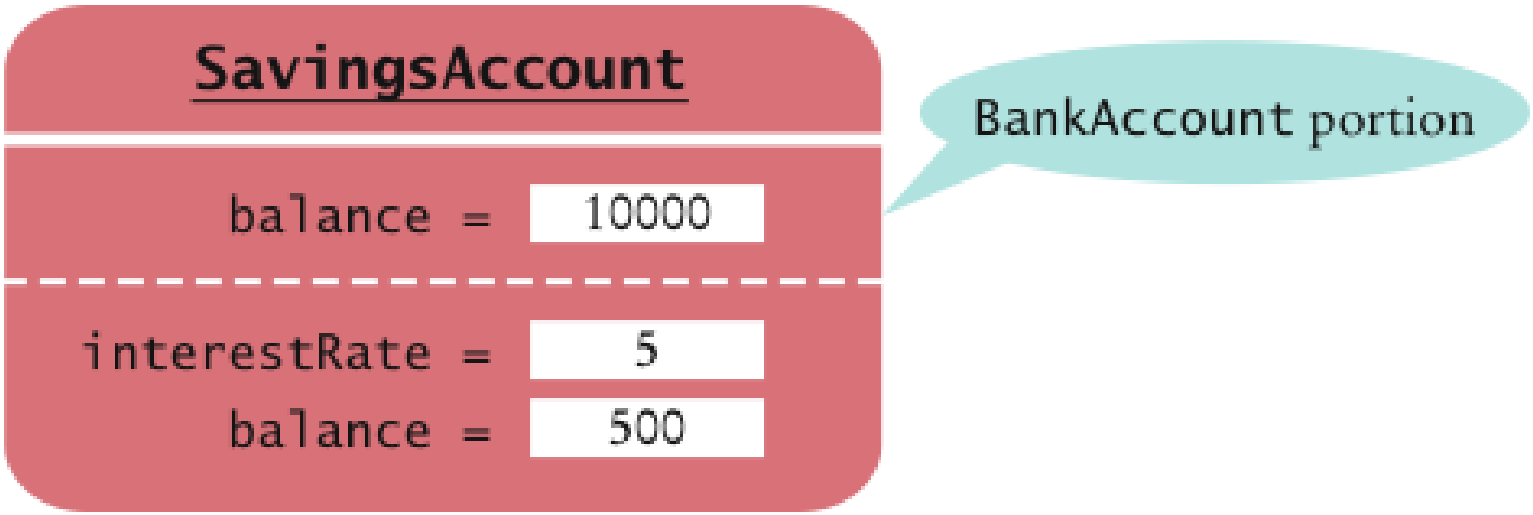


Figure 5 Shadowing Instance Variables

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*

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() { . . . }
}
```

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`

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

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
   }
}
```

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

Overriding Methods (cont.)

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

Syntax 9.2 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.

Animation 9.1: Inheritance

Self Check 9.6

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

Answer: The `SavingsAccount` class inherits the `deposit`, `withdraw`, and `getBalance` methods. The `addInterest` method is new. No methods override superclass methods.

Self Check 9.7

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

Answer: It needs to reduce the balance, and it cannot access the `balance` variable directly.

Self Check 9.8

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

Answer: So that the count can reflect the number of transactions for the following month.

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;
   }
   ...
}
```

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*

ch09/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

ch09/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

ch09/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   }
```

Syntax 9.3 Calling a Superclass Constructor

Syntax

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

Example

```
public CheckingAccount(double initialBalance)
{
   super(initialBalance);
   transactionCount = 0;
}
```

Invokes the constructor of the superclass.

Must be the first statement of the subclass constructor.

If not present, the superclass is constructed with its default constructor.

Subclass constructor

Self Check 9.9

Why didn't the `SavingsAccount` constructor in Section 9.2 call its superclass constructor?

Answer: It was content to use the default constructor of the superclass, which sets the balance to zero.

Self Check 9.10

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

Answer: No — this is a requirement only for constructors. For example, the `SavingsAccount.deposit` method first increments the transaction count, then calls the superclass method.

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`

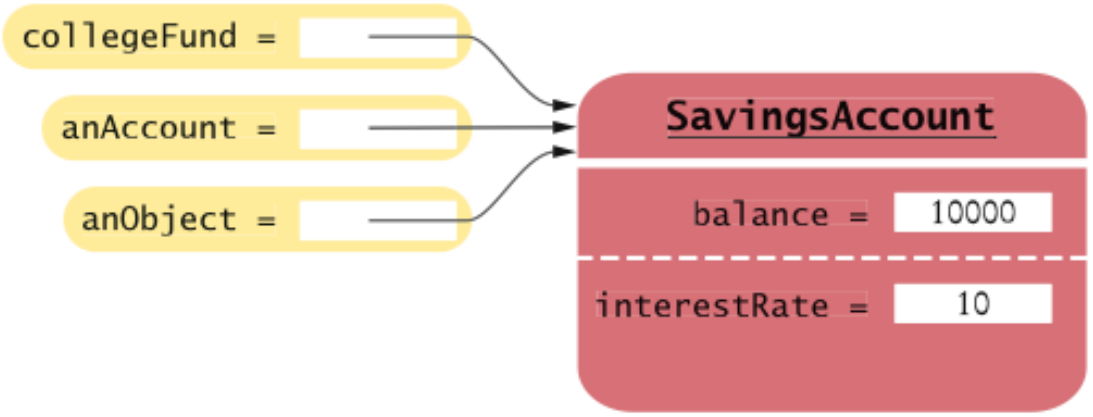


Figure 6
Variables of Different Types Can Refer to the Same Object

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`

Converting Between Subclass and Superclass Types

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

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

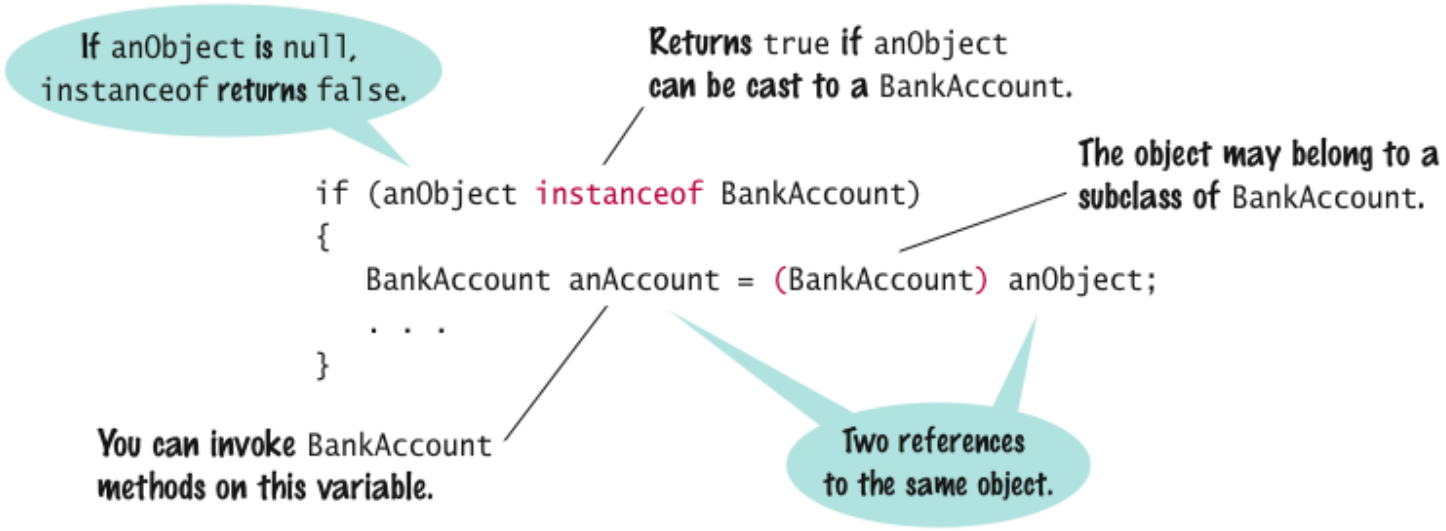
- This cast is dangerous: If you are wrong, an exception is thrown
- Solution: Use the `instanceof` operator
- `instanceof`: Tests whether an object belongs to a particular type:

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

Syntax 9.4 The `instanceof` Operator

Syntax *object* `instanceof` *TypeName*

Example



Self Check 9.11

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

Answer: We want to use the method for all kinds of bank accounts. Had we used a parameter of type `SavingsAccount`, we couldn't have called the method with a `CheckingAccount` object.

Self Check 9.12

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

Answer: We cannot invoke the `deposit` method on a variable of type `Object`.

Polymorphism and Inheritance

- 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

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);
```

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

ch09/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

ch09/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
```

Self Check 9.13

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?

Answer: The object is an instance of `BankAccount` or one of its subclasses.

Self Check 9.14

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.

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;
}
```

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

`Object`: The Cosmic Superclass

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

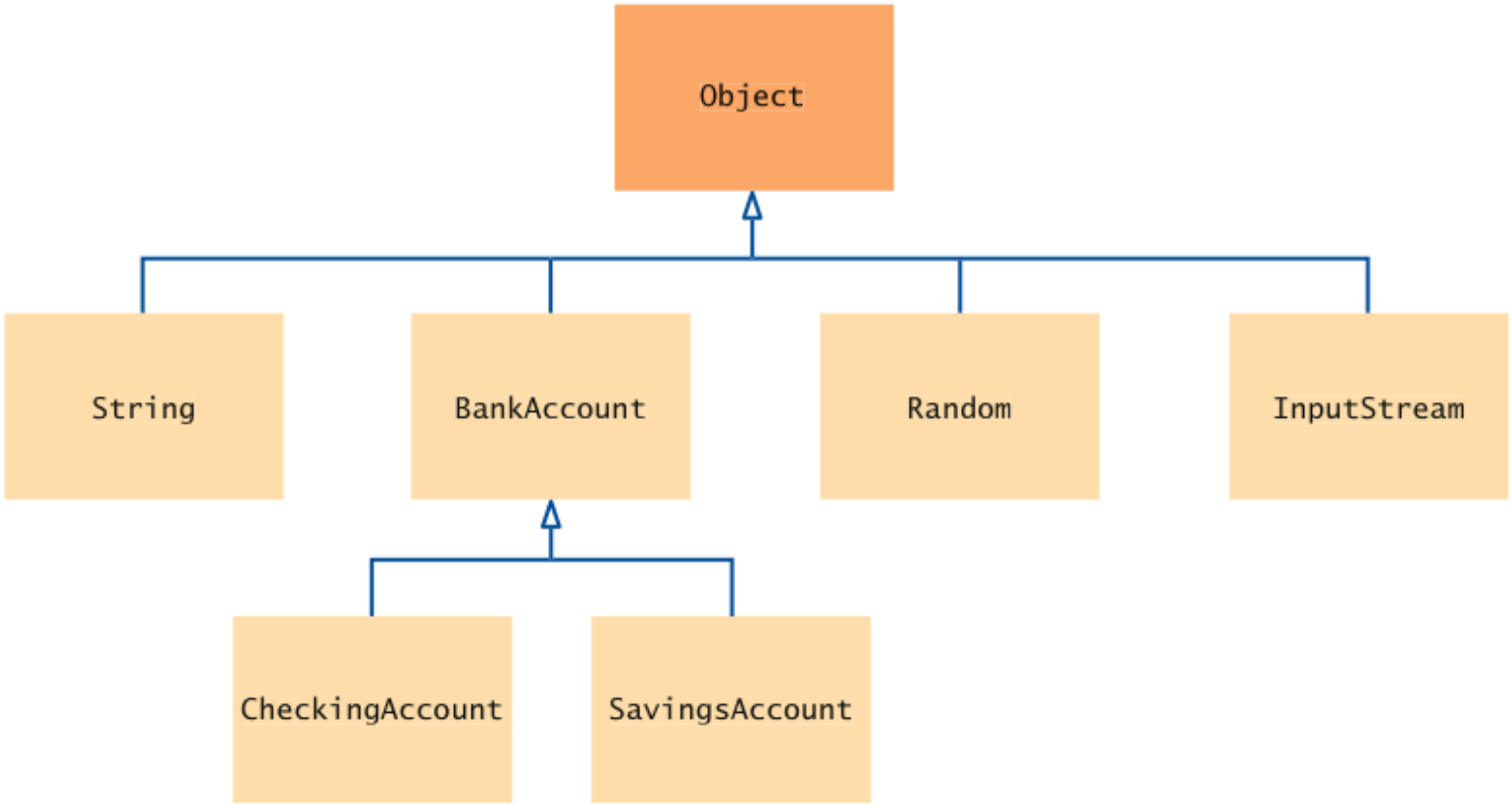


Figure 7 The `Object` Class Is the Superclass of Every Java Class

`Object`: The Cosmic Superclass

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

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]"
```

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"
```

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]"
```

Overriding the `equals` Method

- `equals` tests for same *contents*:

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

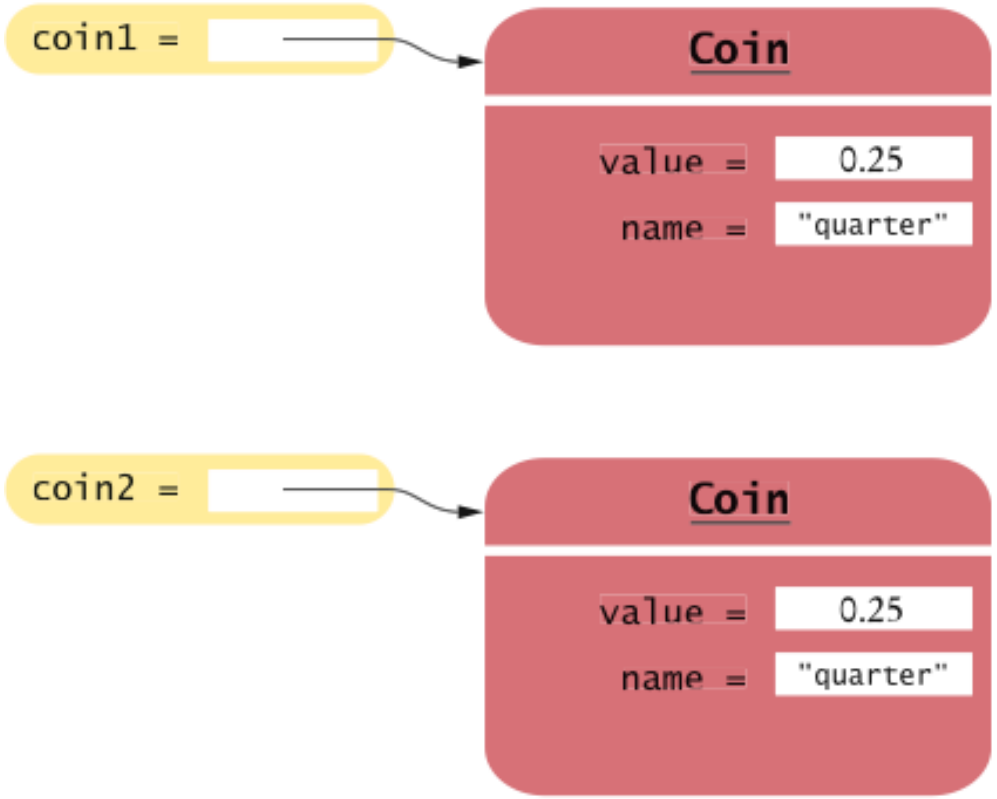


Figure 8 Two References to Equal Objects

Overriding the `equals` Method

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

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

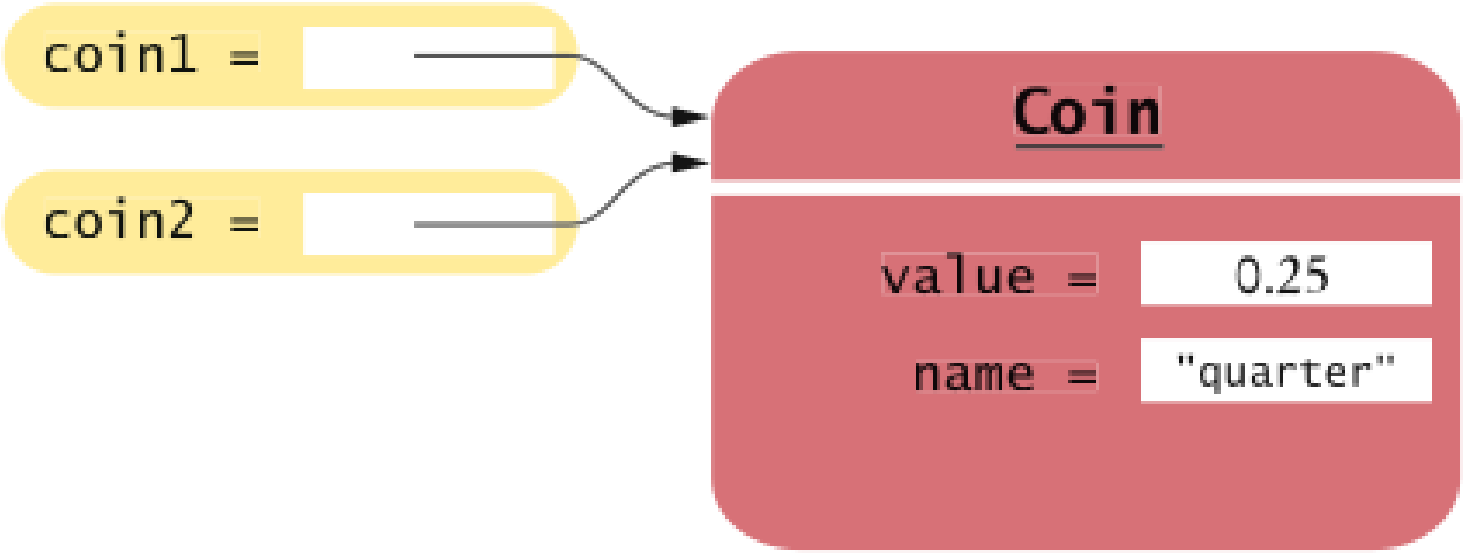


Figure 9 Two References to the Same Object

Overriding the `equals` Method

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

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

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;
   }
   ...
}
```

- You should also override the `hashCode` method so that equal objects have the same hash code

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:

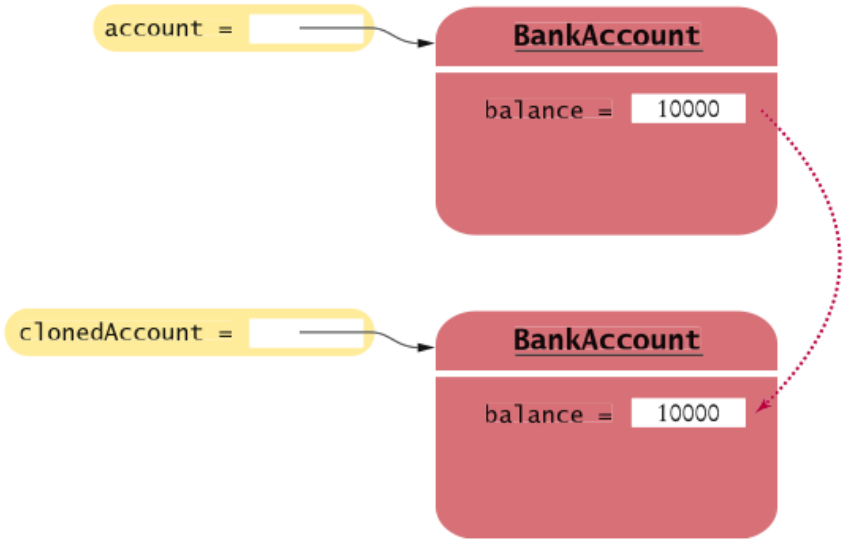


Figure 10
Cloning Objects

The `clone` Method

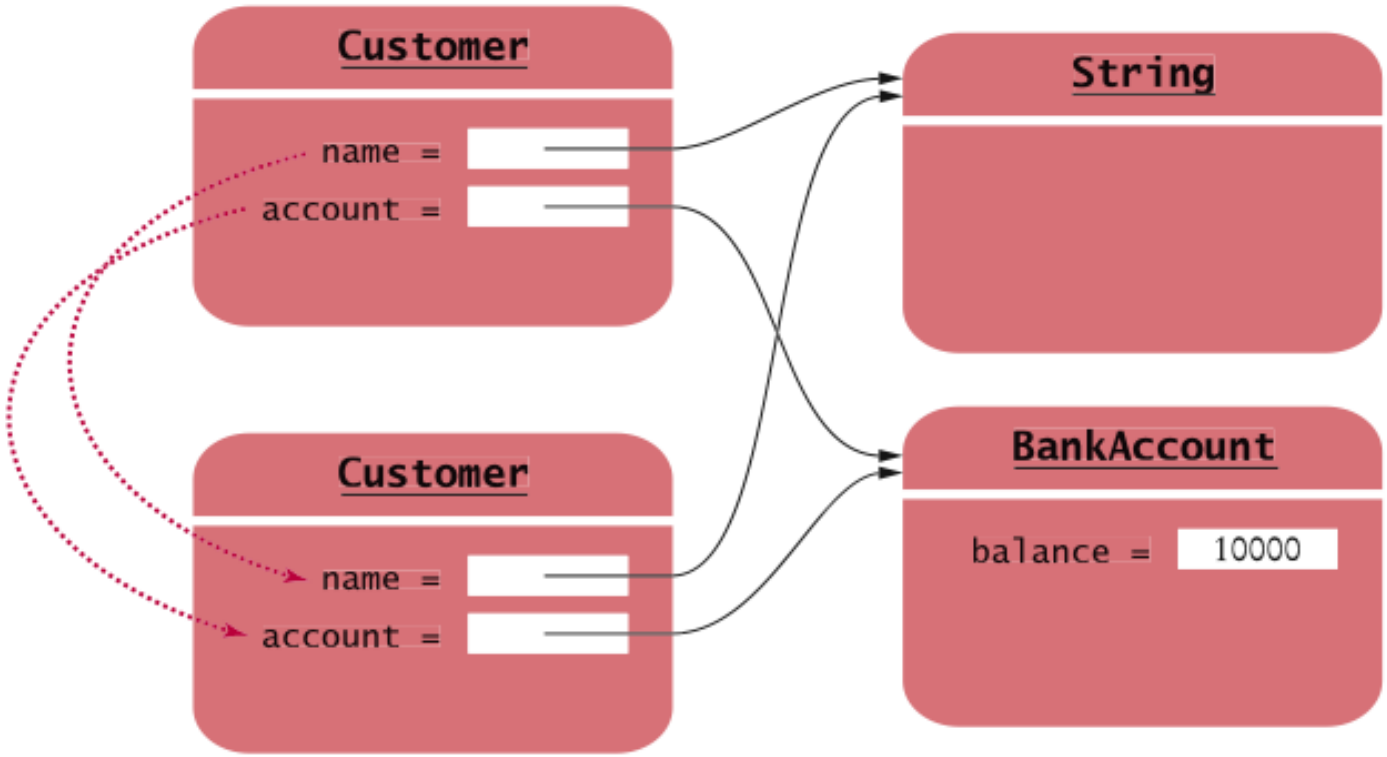
- Implement `clone` method to make a new object with the same state as an existing object
- Use `clone`:

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

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

The `Object.clone` Method

- Creates *shallow copies*:



The `Object.clone` Method Makes a Shallow Copy

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 9.6)

Self Check 9.15

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

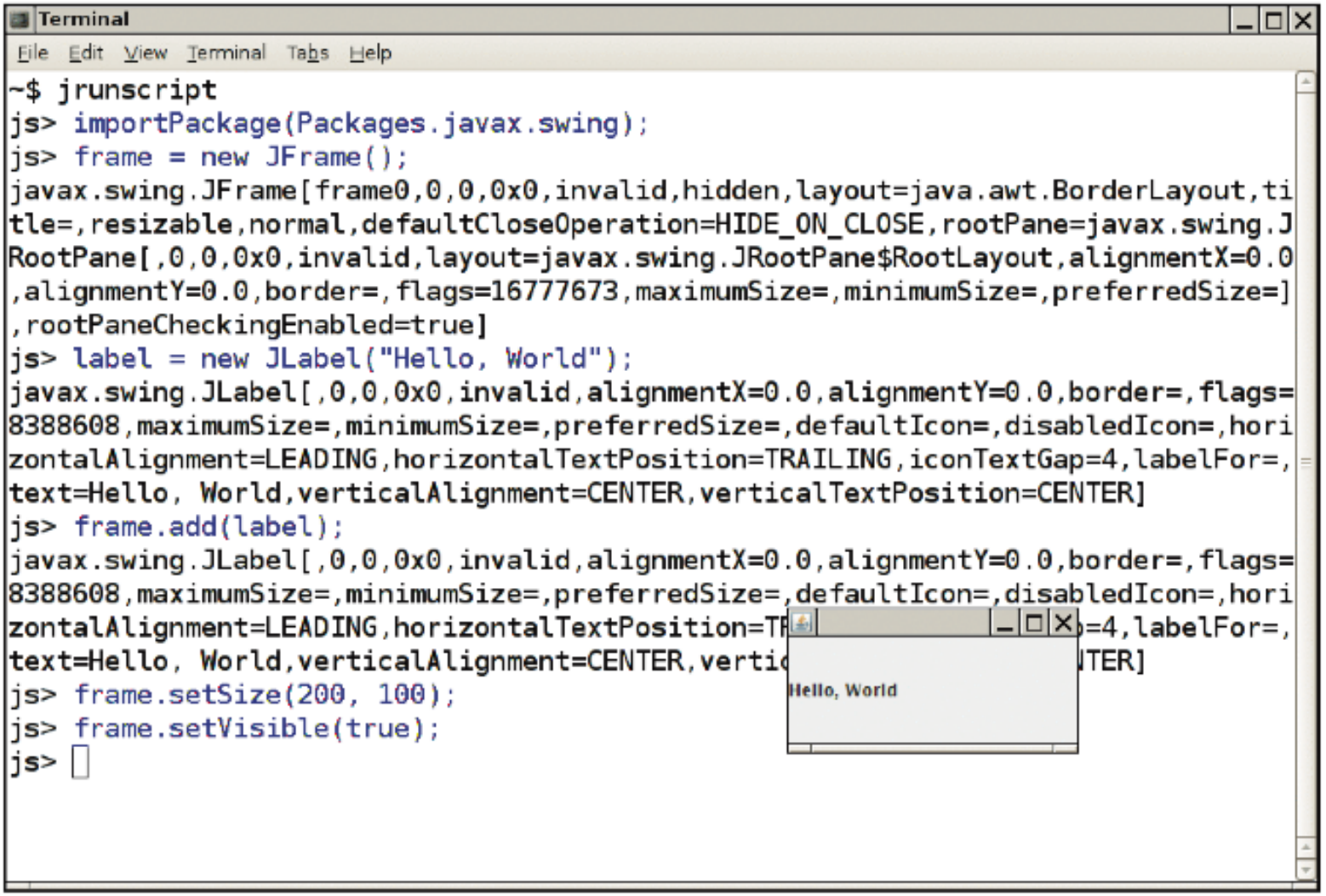
Answer: It certainly should — unless, of course, `x` is `null`.

Self Check 9.16

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

Answer: If `toString` returns a string that describes all instance variables, you can simply call `toString` on the implicit and explicit parameters, and compare the results. However, comparing the variables is more efficient than converting them into strings.

Scripting Languages



Scripting Java Classes with JavaScript

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

ch09/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

ch09/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

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   }
```

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 InvestmentViewer2
 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  }
```

Self Check 9.17

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`.

Self Check 9.18

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*.