

# ICOM 4015: Advanced Programming

## Lecture 10

### Chapter Ten: Inheritance



## Chapter Ten: Inheritance

*Big Java* by Cay Horstmann  
Copyright © 2008 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
- To use inheritance for customizing user interfaces

# An Introduction to Inheritance

---

- Inheritance: extend classes by adding methods and fields
- Example: Savings account = bank account with interest

```
class SavingsAccount extends BankAccount
{
    new methods
    new instance fields
}
```

- SavingsAccount **automatically inherits all methods and instance fields of** BankAccount

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

***Continued***

## An Introduction to Inheritance (cont.)

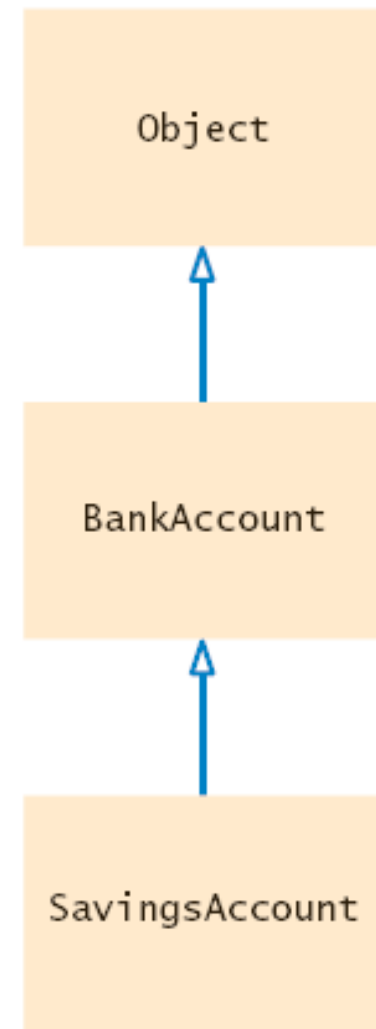
---

- Extended class = *superclass* (`BankAccount`), extending class = *subclass* (`Savings`)
- Inheriting from class  $\neq$  implementing interface: subclass inherits behavior and state
- One advantage of inheritance is code reuse

## An Inheritance Diagram

---

Every class extends the `Object` class either directly or indirectly



**Figure 1**  
An Inheritance Diagram

## An Introduction to Inheritance

---

- In `subclass`, specify added instance fields, added methods, and changed or overridden methods

```
public class SavingsAccount extends BankAccount
{
    public SavingsAccount(double rate)
    {
        interestRate = rate;
    }

    public void addInterest()
    {
        double interest = getBalance() * interestRate /
            100;
        deposit(interest);
    }
}
```

***Continued***

## An Introduction to Inheritance

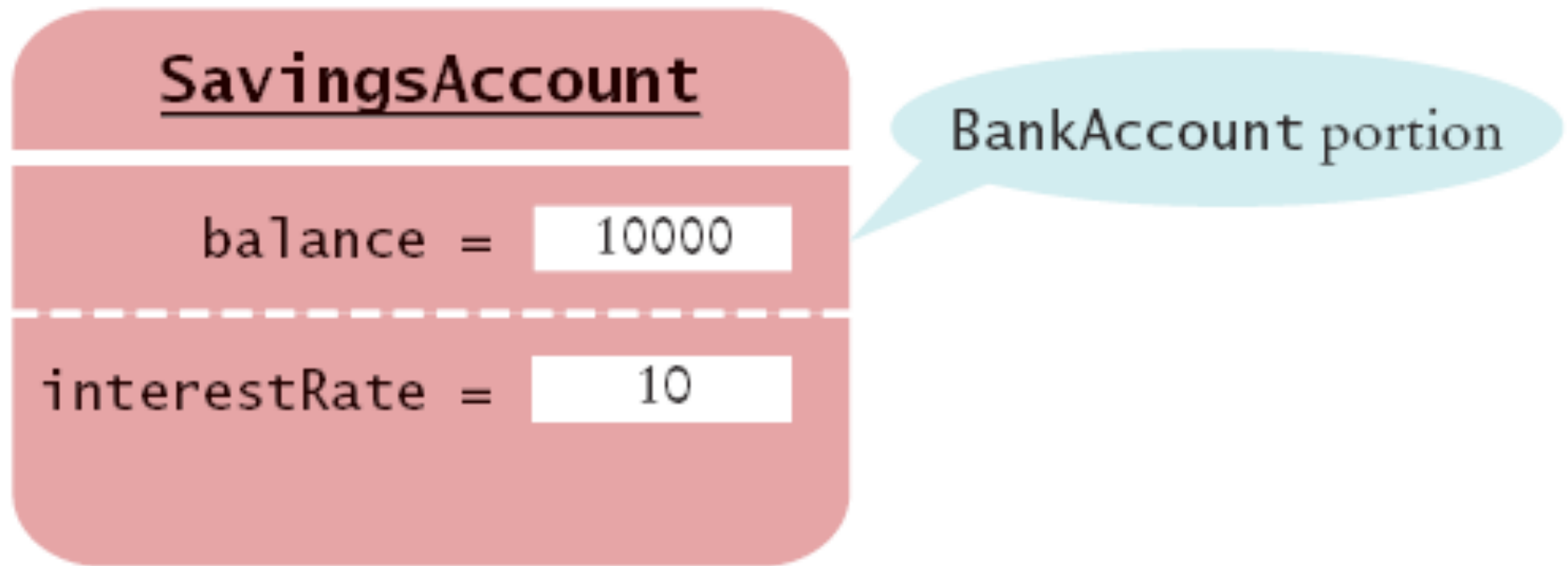
---

```
    private double interestRate;  
}
```

- **Encapsulation:** `addInterest` **calls** `getBalance` rather than updating the `balance` field of the superclass (field is `private`)
- **Note that** `addInterest` **calls** `getBalance` without specifying an implicit parameter (the calls apply to the same object)

## Layout of a Subclass Object

`SavingsAccount` object inherits the `balance` instance field from `BankAccount`, and gains one additional instance field: `interestRate`:



**Figure 2** Layout of a Subclass Object

## Syntax 10.1 Inheritance

---

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

### Example:

```
public class SavingsAccount extends BankAccount
{
    public SavingsAccount(double rate)
    {
        interestRate = rate;
    }
}
```

***Continued***

## Syntax 10.1 Inheritance

```
public void addInterest()  
{  
    double interest = getBalance() * interestRate / 100;  
    deposit(interest);  
}  
  
private double interestRate;  
}
```

### Purpose:

To define a new class that inherits from an existing class, and define the methods and instance fields that are added in the new class.

## Self Check 10.1

---

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

**Answer:** Two instance fields: `balance` and `interestRate`.

## Self Check 10.2

---

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

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

## Self Check 10.3

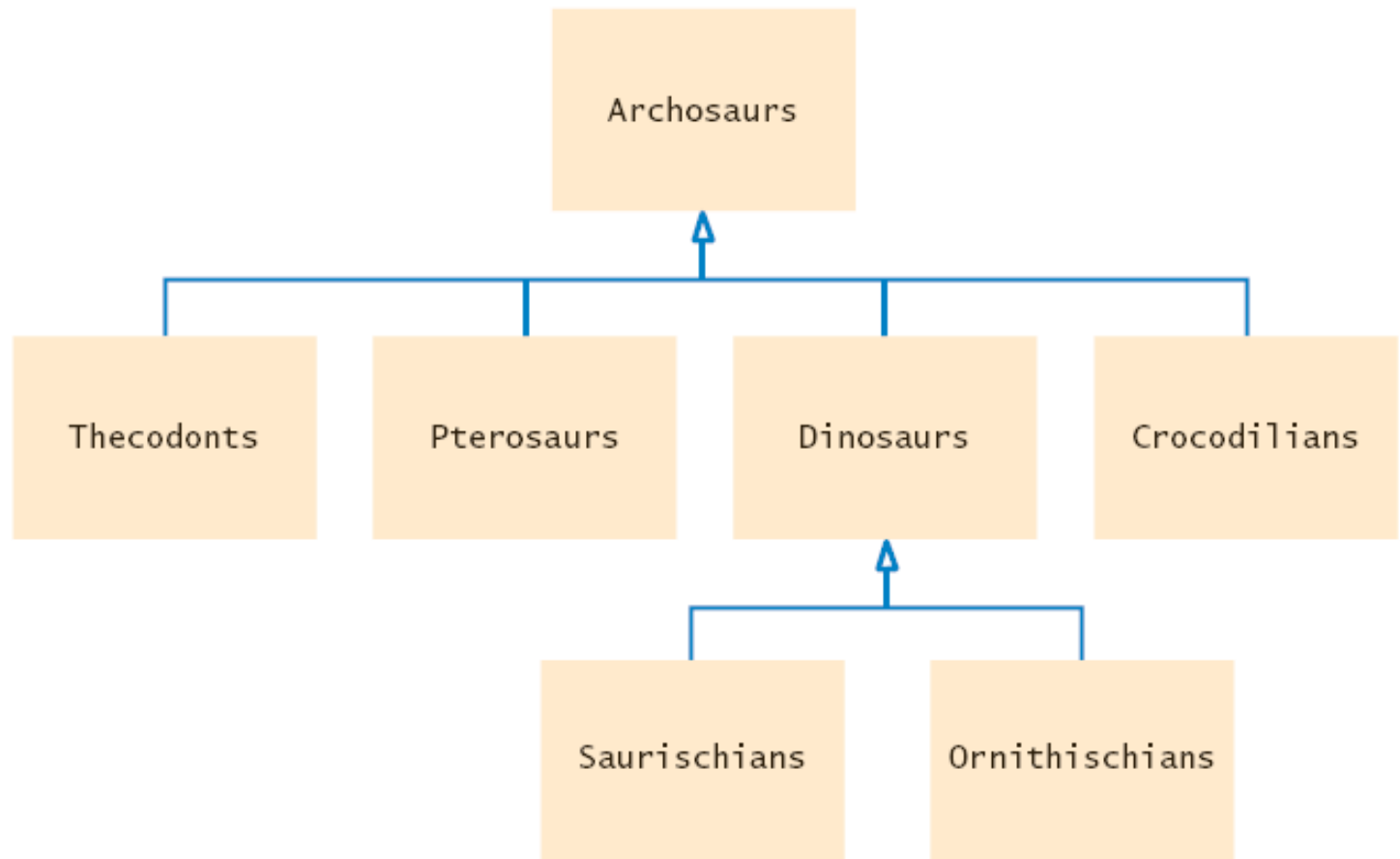
---

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.

## Inheritance Hierarchies

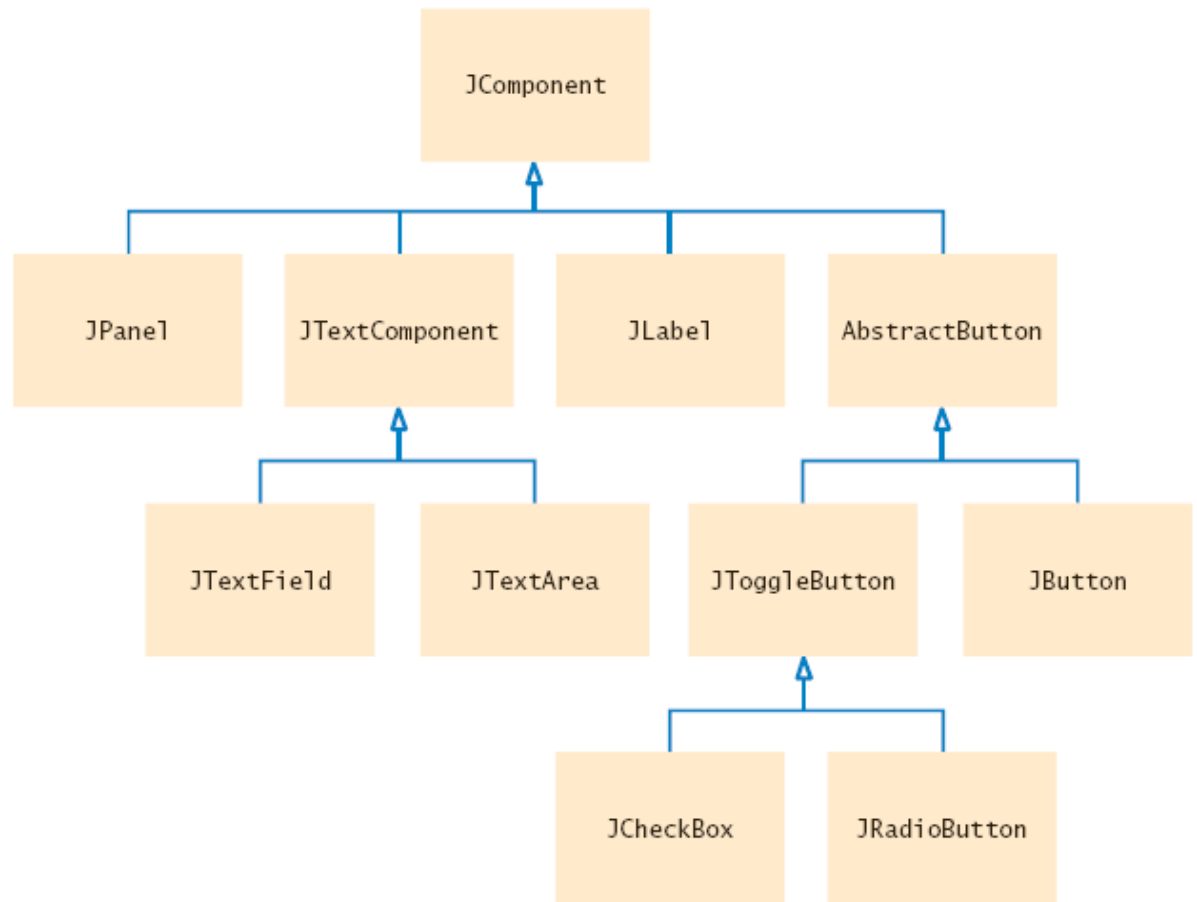
- Sets of classes can form complex inheritance hierarchies
- Example:



**Figure 3** A Part of the Hierarchy of Ancient Reptiles

## Inheritance Hierarchies Example: Swing Hierarchy

- Superclass `JComponent` has methods `getWidth`, `getHeight`
- `AbstractButton` class has methods to set/get button text and icon



**Figure 4** A Part of the Hierarchy of Swing User Interface Components

## A Simpler Example: Hierarchy of Bank Accounts

- Consider a bank that offers its customers the following account types:

1. *Checking account: no interest; small number of free transactions per month, additional transactions are charged a small fee*

2. *Savings account: earns interest that compounds monthly*

- Inheritance hierarchy:

- All bank accounts support the `getBalance` method

- All bank accounts support the `deposit` and `withdraw` methods, but the implementations differ

- Checking account needs a method `deductFees`; savings account needs a method `addInterest`

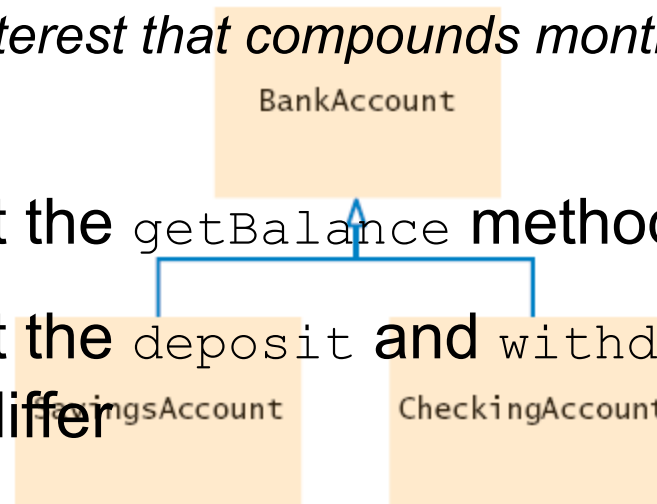


Figure 5

Inheritance Hierarchy for Bank Account Classes

## Self Check 10.4

---

What is the purpose of the `JTextComponent` class in Figure 4?

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

## Self Check 10.5

---

Which instance field will we need to add to the `CheckingAccount` class?

**Answer:** We need a counter that counts the number of withdrawals and deposits.

# Inheriting Methods

---

- **Override method:**
  - *Supply a different implementation of a method that exists in the `superclass`*
  - *Must have same signature (same name and same parameter types)*
  - *If method is applied to an object of the `subclass` type, the overriding method is executed*
- **Inherit method:**
  - *Don't supply a new implementation of a method that exists in `superclass`*
  - *`Superclass` method can be applied to the `subclass` objects*
- **Add method:**
  - *Supply a new method that doesn't exist in the `superclass`*
  - *New method can be applied only to `subclass` objects*

## Inheriting Instance Fields

---

- Can't override fields
- Inherit field: All fields from the superclass are automatically inherited
- Add field: Supply a new field that doesn't exist in the superclass
- What if you define a new field with the same name as a superclass field?
  - *Each object would have two instance fields of the same name*
  - *Fields can hold different values*
  - *Legal but extremely undesirable*

## Implementing the `CheckingAccount` Class

---

- Overrides `deposit` and `withdraw` to increment the transaction count:

```
public class CheckingAccount extends BankAccount
{
    public void deposit(double amount) { . . . }
    public void withdraw(double amount) { . . . }
    public void deductFees() { . . . }
    // new method private int transactionCount; // new
    instance field }
```

- Each `CheckingAccount` object has two instance fields:
  - *balance* (*inherited from BankAccount*)
  - *transactionCount* (*new to CheckingAccount*)

***Continued***

## Implementing the `CheckingAccount` Class (cont.)

---

- You can apply four methods to `CheckingAccount` objects:
  - `getBalance()` (*inherited from `BankAccount`*)
  - `deposit(double amount)` (**overrides `BankAccount` method**)
  - `withdraw(double amount)` (**overrides `BankAccount` method**)
  - `deductFees()` (**new to `CheckingAccount`**)

## Inherited Fields are Private

---

- Consider `deposit` method of `CheckingAccount`

```
public void deposit(double amount)
{
    transactionCount++;
    // now add amount to balance
    . . .
}
```
- Can't just add `amount` to `balance`
- `balance` is a *private* field of the superclass
- A subclass has no access to private fields of its superclass
- Subclass must use public interface

## Invoking a Superclass Method

---

- Can't just call

```
deposit (amount)
```

```
in deposit method of CheckingAccount
```

- That is the same as

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

- Calls the same method (infinite recursion)

- Instead, invoke *superclass method*

```
super.deposit (amount)
```

- Now calls `deposit` method of `BankAccount` class

***Continued***

## Invoking a Superclass Method (cont.)

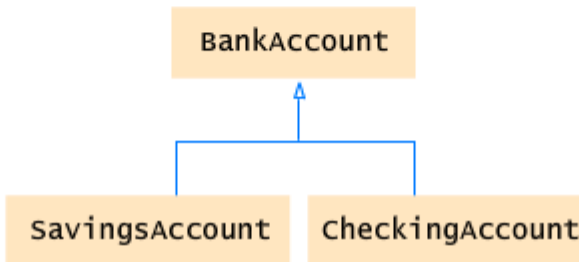
---

- Complete method:

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

## Animation 10.1 –

```
public class BankAccount
{
    public BankAccount() { . . . }
    public BankAccount(double initialBalance) { . . . }
    public void deposit(double amount) { . . . }
    public void withdraw(double amount) { . . . }
    public double getBalance() { . . . }
    private double balance;
}
```



```
public class SavingsAccount extends BankAccount
{
    public SavingsAccount(double rate) { . . . }
    public void addInterest() { . . . }
    private double interestRate;
}
```

```
public class CheckingAccount extends BankAccount
{
    public CheckingAccount() { . . . }
    public void deposit(double amount) { . . . }
    public void withdraw(double amount) { . . . }
    public void deductFees() { . . . }
    private int transactionCount;
}
```

This animation demonstrates the process of inheritance.



## Syntax 10.2 Calling a Superclass Method

---

```
super.methodName(parameters)
```

### **Example:**

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

### **Purpose:**

To call a method of the superclass instead of the method of the current class.

## Implementing Remaining Methods

---

```
public class CheckingAccount extends BankAccount
{
    . . .
    public void withdraw(double amount)
    {
        transactionCount++;
        // Now subtract amount from balance
        super.withdraw(amount);
    }

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

**Continued**

## Implementing Remaining Methods (cont.)

---

```
        transactionCount = 0;
    }
    . . .
    private static final
    int FREE_TRANSACTIONS = 3;
    private static final double TRANSACTION_FEE = 2.0;
}
```

## Self Check 10.6

---

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 field directly.

## Self Check 10.7

---

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.

## Common Error: Shadowing Instance Fields

---

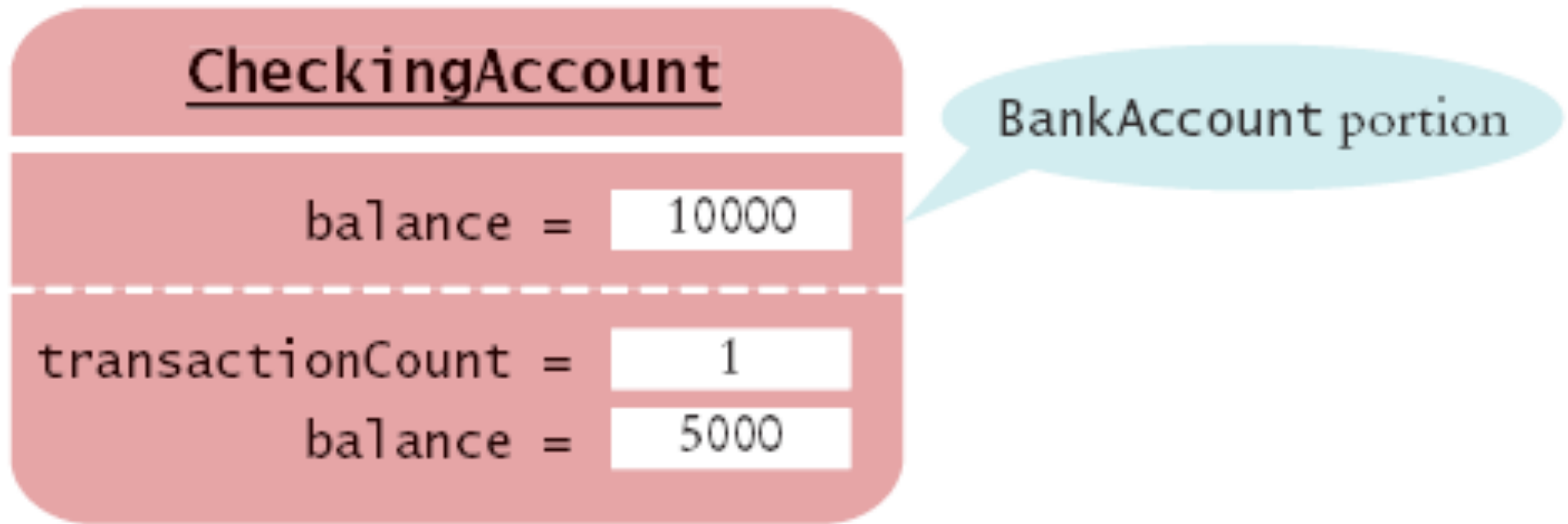
- A subclass has no access to the private instance fields of the superclass
- Beginner's error: "solve" this problem by adding another instance field with same name:

```
public class CheckingAccount extends BankAccount
{
    public void deposit(double amount)
    {
        transactionCount++;
        balance = balance + amount;
    }
    . . .
    private double balance; // Don't
}
```

***Continued***

## Common Error: Shadowing Instance Fields (cont.)

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



**Figure 6** Shadowing Instance Fields

## Subclass Construction

---

- `super` followed by a parenthesis indicates a call to the superclass constructor

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

- Must be the *first* statement in subclass constructor

***Continued***

## Subclass Construction (cont.)

---

- If subclass constructor doesn't call superclass constructor, default superclass constructor is used
  - *Default constructor: constructor with no parameters*
  - *If all constructors of the superclass require parameters, then the compiler reports an error*

## Syntax 10.3 Calling a Superclass Constructor

```
ClassName (parameters)  
{  
    super (parameters) ;  
    . . .  
}
```

### Example:

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

### Purpose:

To invoke a constructor of the superclass. Note that this statement must be the first statement of the subclass constructor.

## Self Check 10.8

---

Why didn't the `SavingsAccount` constructor in Section 10.1 call its superclass constructor?

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

## Self Check 10.9

---

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

---

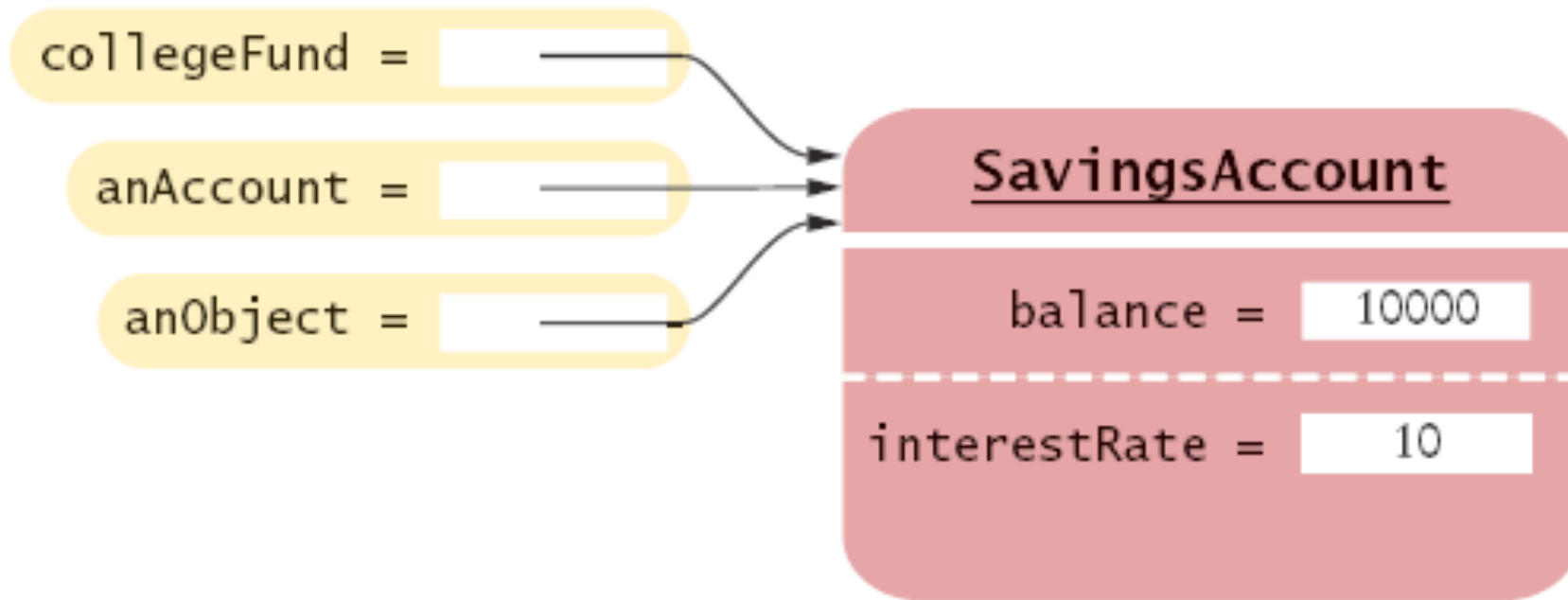
- Ok to convert subclass reference to superclass reference

```
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`

***Continued***

## Converting Between Subclass and Superclass Types (cont.)



**Figure 7** Variables of Different Types 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
```

- When you convert between a subclass object to its superclass type:
  - *The value of the reference stays the same – it is the memory location of the object*
  - *But, less information is known about the object*

***Continued***

## Converting Between Subclass and Superclass Types (cont.)

---

- 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 10.4 The instanceof Operator

---

*object* instanceof *TypeName*

### Example:

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

### Purpose:

To return `true` if the *object* is an instance of *TypeName* (or one of its subtypes), and `false` otherwise.

## Self Check 10.10

---

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 10.11

---

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

---

- In Java, 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
```

- Method calls are determined by type of actual object, not type of object reference

```
BankAccount anAccount = new CheckingAccount();  
anAccount.deposit(1000); // Calls "deposit" from  
CheckingAccount
```

- Compiler needs to check that only legal methods are invoked

```
Object anObject = new BankAccount();  
anObject.deposit(1000); // Wrong!
```

# Polymorphism

---

- Polymorphism: ability to refer to objects of multiple types with varying behavior

- Polymorphism at work:

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

- Depending on types of `amount` and `other`, different versions of `withdraw` and `deposit` are called

## ch10/accounts/AccountTester.java

---

```
01: /**
02:     This program tests the BankAccount class and
03:     its subclasses.
04: */
05: public class AccountTester
06: {
07:     public static void main(String[] args)
08:     {
09:         SavingsAccount momsSavings
10:             = new SavingsAccount(0.5);
11:
12:         CheckingAccount harrysChecking
13:             = new CheckingAccount(100);
14:
15:         momsSavings.deposit(10000);
16:
17:         momsSavings.transfer(2000, harrysChecking);
18:         harrysChecking.withdraw(1500);
19:         harrysChecking.withdraw(80);
20:
```

***Continued***

## ch10/accounts/AccountTester.java (cont.)

---

```
21:      momsSavings.transfer(1000, harrysChecking);
22:      harrysChecking.withdraw(400);
23:
24:      // Simulate end of month
25:      momsSavings.addInterest();
26:      harrysChecking.deductFees();
27:
28:      System.out.println("Mom's savings balance: "
29:          + momsSavings.getBalance());
30:      System.out.println("Expected: 7035");
31:
32:      System.out.println("Harry's checking balance: "
33:          + harrysChecking.getBalance());
34:      System.out.println("Expected: 1116");
35:  }
36: }
```

## ch10/accounts/CheckingAccount.java

---

```
01: /**
02:     A checking account that charges transaction fees.
03: */
04: public class CheckingAccount extends BankAccount
05: {
06:     /**
07:         Constructs a checking account with a given balance.
08:         @param initialBalance the initial balance
09:     */
10:     public CheckingAccount(double initialBalance)
11:     {
12:         // Construct superclass
13:         super(initialBalance);
14:
15:         // Initialize transaction count
16:         transactionCount = 0;
17:     }
18:
19:     public void deposit(double amount)
20:     {
21:         transactionCount++;
```

***Continued***

## ch10/accounts/CheckingAccount.java (cont.)

---

```
22:         // Now add amount to balance
23:         super.deposit(amount);
24:     }
25:
26:     public void withdraw(double amount)
27:     {
28:         transactionCount++;
29:         // Now subtract amount from balance
30:         super.withdraw(amount);
31:     }
32:
33:     /**
34:      * Deducts the accumulated fees and resets the
35:      * transaction count.
36:      */
37:     public void deductFees()
38:     {
39:         if (transactionCount > FREE_TRANSACTIONS)
40:         {
41:             double fees = TRANSACTION_FEE *
42:                 (transactionCount - FREE_TRANSACTIONS);
43:             super.withdraw(fees);
44:         }
```

**Continued**

*Big Java* by Cay Horstmann

Copyright © 2008 by John Wiley & Sons. All rights reserved.

## ch10/accounts/CheckingAccount.java (cont.)

---

```
45:         transactionCount = 0;
46:     }
47:
48:     private int transactionCount;
49:
50:     private static final int FREE_TRANSACTIONS = 3;
51:     private static final double TRANSACTION_FEE = 2.0;
52: }
```

## ch10/accounts/BankAccount.java

---

```
01: /**
02:     A bank account has a balance that can be changed by
03:     deposits and withdrawals.
04: */
05: public class BankAccount
06: {
07:     /**
08:         Constructs a bank account with a zero balance.
09:     */
10:     public BankAccount()
11:     {
12:         balance = 0;
13:     }
14:
15:     /**
16:         Constructs a bank account with a given balance.
17:         @param initialBalance the initial balance
18:     */
19:     public BankAccount(double initialBalance)
20:     {
21:         balance = initialBalance;
22:     }
23:
```

***Continued***

## ch10/accounts/BankAccount.java (cont.)

---

```
24:    /**
25:        Deposits money into the bank account.
26:        @param amount the amount to deposit
27:    */
28:    public void deposit(double amount)
29:    {
30:        balance = balance + amount;
31:    }
32:
33:    /**
34:        Withdraws money from the bank account.
35:        @param amount the amount to withdraw
36:    */
37:    public void withdraw(double amount)
38:    {
39:        balance = balance - amount;
40:    }
41:
42:    /**
43:        Gets the current balance of the bank account.
44:        @return the current balance
45:    */
```

***Continued***

## ch10/accounts/BankAccount.java (cont.)

---

```
46:     public double getBalance()
47:     {
48:         return balance;
49:     }
50:
51:     /**
52:      * Transfers money from the bank account to another account
53:      * @param amount the amount to transfer
54:      * @param other the other account
55:      */
56:     public void transfer(double amount, BankAccount other)
57:     {
58:         withdraw(amount);
59:         other.deposit(amount);
60:     }
61:
62:     private double balance;
63: }
```

## ch10/accounts/SavingsAccount.java

---

```
01: /**
02:     An account that earns interest at a fixed rate.
03: */
04: public class SavingsAccount extends BankAccount
05: {
06:     /**
07:         Constructs a bank account with a given interest rate.
08:         @param rate the interest rate
09:     */
10:     public SavingsAccount(double rate)
11:     {
12:         interestRate = rate;
13:     }
14:
15:     /**
16:         Adds the earned interest to the account balance.
17:     */
```

***Continued***

## ch10/accounts/SavingsAccount.java (cont.)

---

```
18:     public void addInterest()  
19:     {  
20:         double interest = getBalance() * interestRate / 100;  
21:         deposit(interest);  
22:     }  
23:  
24:     private double interestRate;  
25: }
```

### Output:

Mom's savings balance: 7035.0

Expected: 7035

Harry's checking balance: 1116.0

Expected: 1116

## Self Check 10.12

---

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 10.13

---

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.

# Access Control

---

- Java has four levels of controlling access to fields, methods, and classes:
  - *public access*
    - *Can be accessed by methods of all classes*
  - *private access*
    - *Can be accessed only by the methods of their own class*
  - *protected access*
    - *See Advanced Topic 10.3*
  - *package access*
    - *The default, when no access modifier is given*
    - *Can be accessed by all classes in the same package*
    - *Good default for classes, but extremely unfortunate for fields*

## Recommended Access Levels

---

- Instance and static fields: Always private. Exceptions:
  - *public static final constants are useful and safe*
  - *Some objects, such as `System.out`, need to be accessible to all programs (`public`)*
  - *Occasionally, classes in a package must collaborate very closely (give some fields package access); inner classes are usually better*
- Methods: `public` or `private`
- Classes and interfaces: `public` or `package`
  - Better alternative to package access: inner classes
    - In general, inner classes should not be `public` (some exceptions exist, e.g., `Ellipse2D.Double`)
- Beware of accidental package access (forgetting `public` or `private`)

## Self Check 10.14

---

What is a common reason for defining package-visible instance fields?

**Answer:** Accidentally forgetting the `private` modifier.

## Self Check 10.15

---

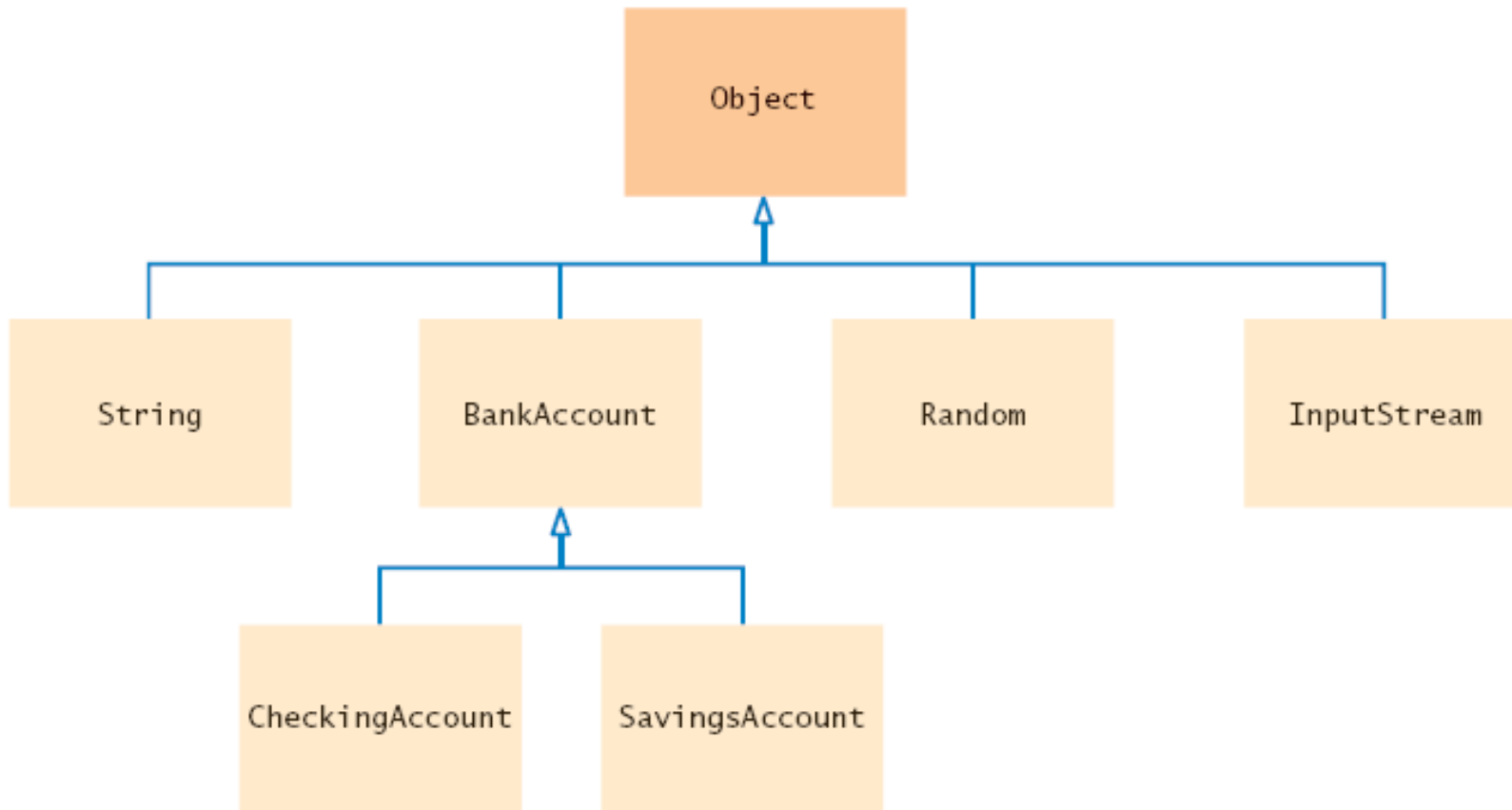
If a class with a public constructor has package access, who can construct objects of it?

**Answer:** Any methods of classes in the same package.

# Object: The Cosmic Superclass

---

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



**Figure 8** The `Object` Class Is the Superclass of Every Java Class

## Object: The Cosmic Superclass

---

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

***Continued***

## Overriding the `toString` Method (cont.)

---

- `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`:

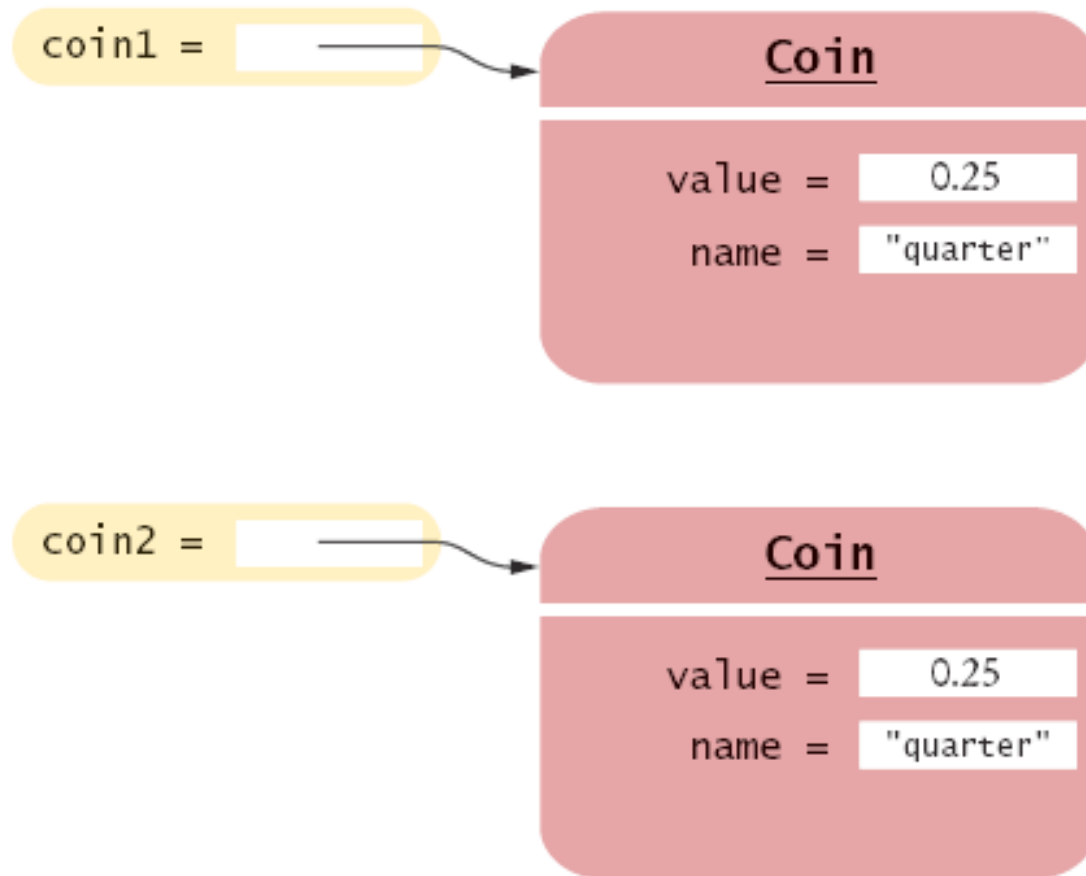
```
public String toString()  
{  
    return "BankAccount[balance=" + balance + "];"  
}
```

- This works better:

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

## Overriding the `equals` Method

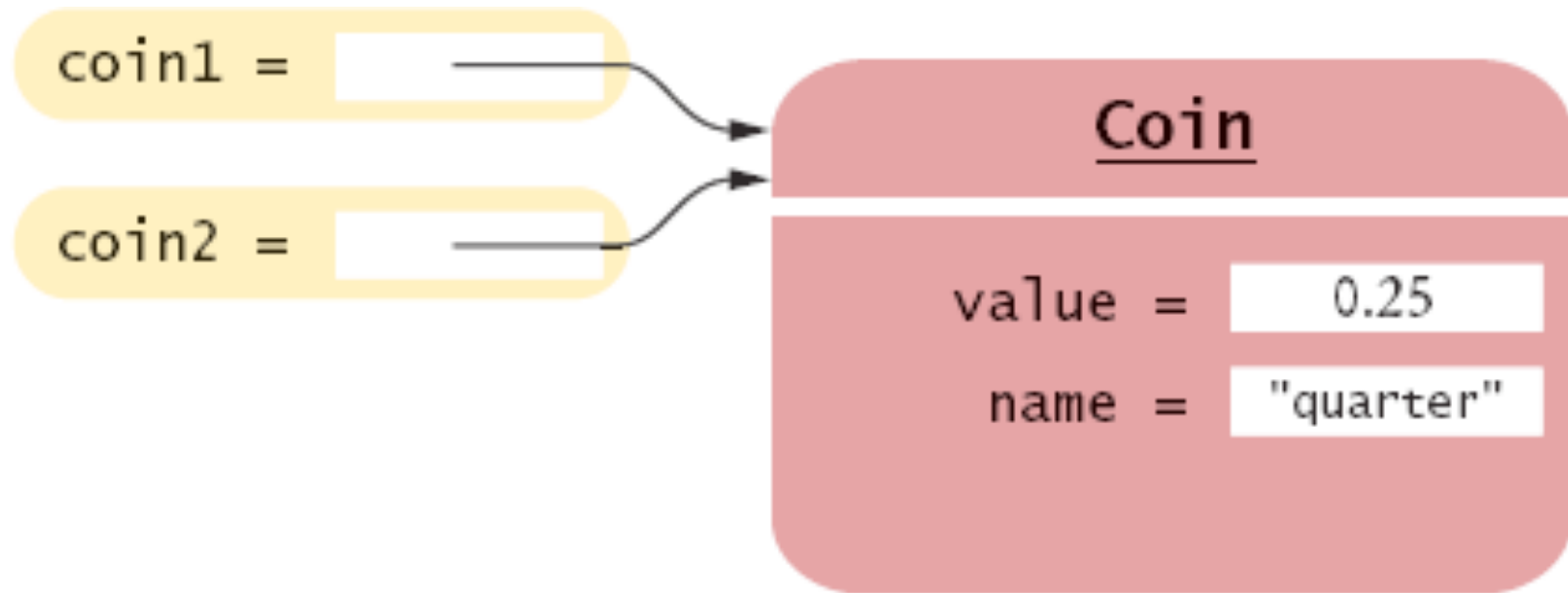
- `Equals` tests for equal *contents*



**Figure 9** Two References to Equal Objects

***Continued***

## Overriding the `equals` Method (cont.)



**Figure 10** Two References to the Same Object

## Overriding the `equals` Method

---

- Define the `equals` method to test whether two objects have equal state
- When redefining `equals` method, you cannot change object signature; use a *cast* instead:

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

***Continued***

## Overriding the `equals` Method (cont.)

---

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

## Self Check 10.16

---

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

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

## Self Check 10.17

---

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

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

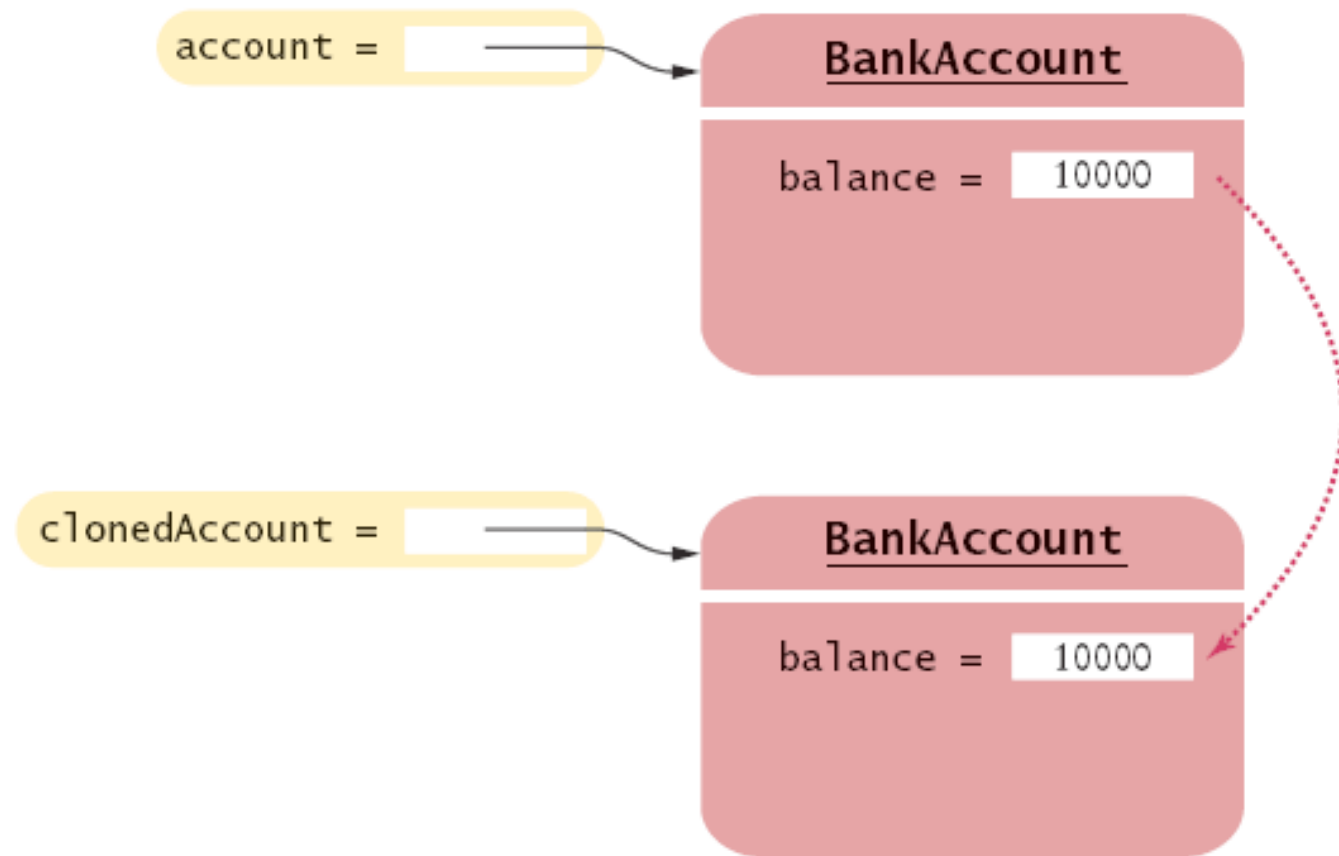
## Overriding the `clone` Method

---

- Copying an object reference gives two references to same object  
`BankAccount account2 = account;`
- Sometimes, need to make a copy of the object

***Continued***

## Overriding the clone Method (cont.)



**Figure 11**  
Cloning Objects

***Continued***

## Overriding the `clone` Method (cont.)

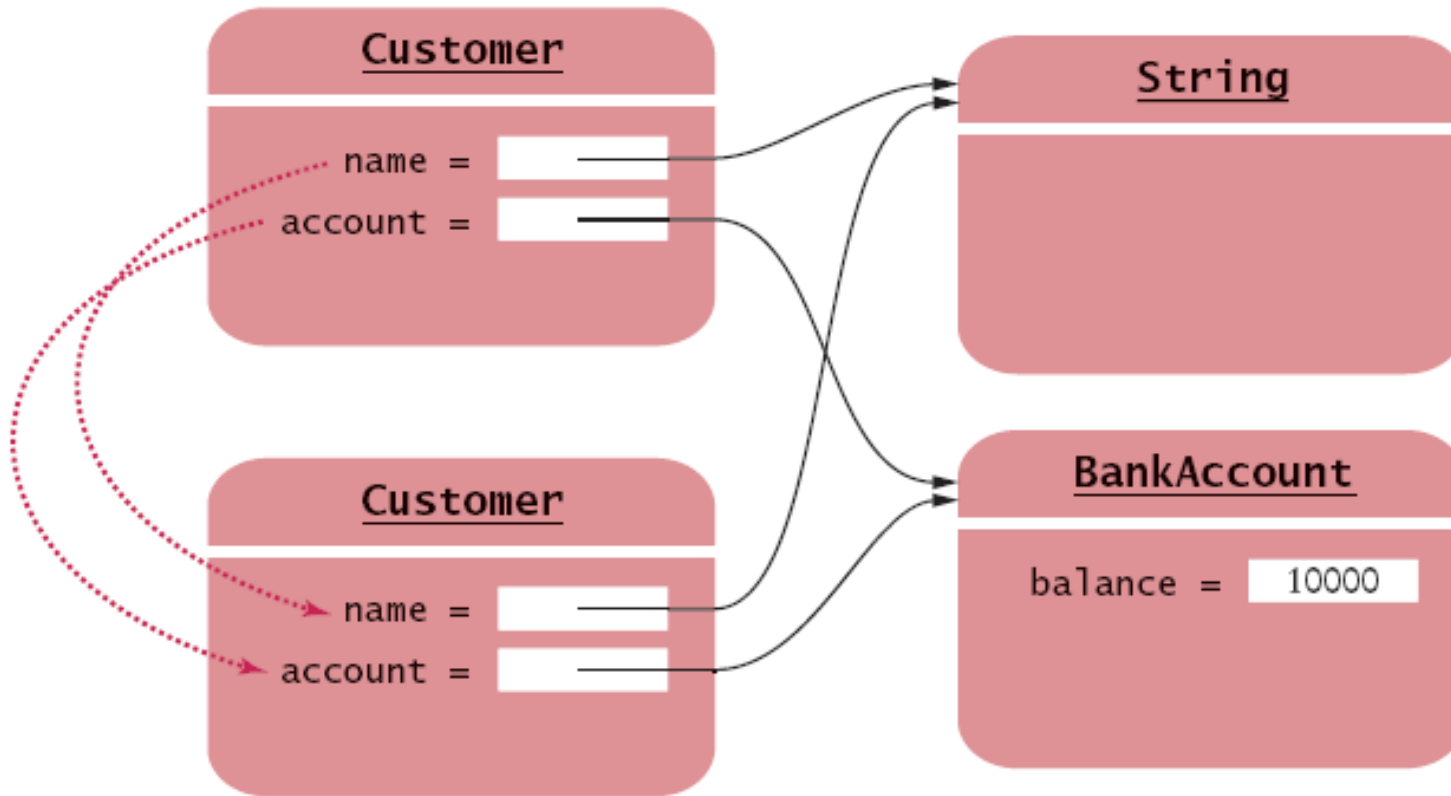
---

- Define `clone` method to make new object (see Advanced Topic 10.6)
- 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

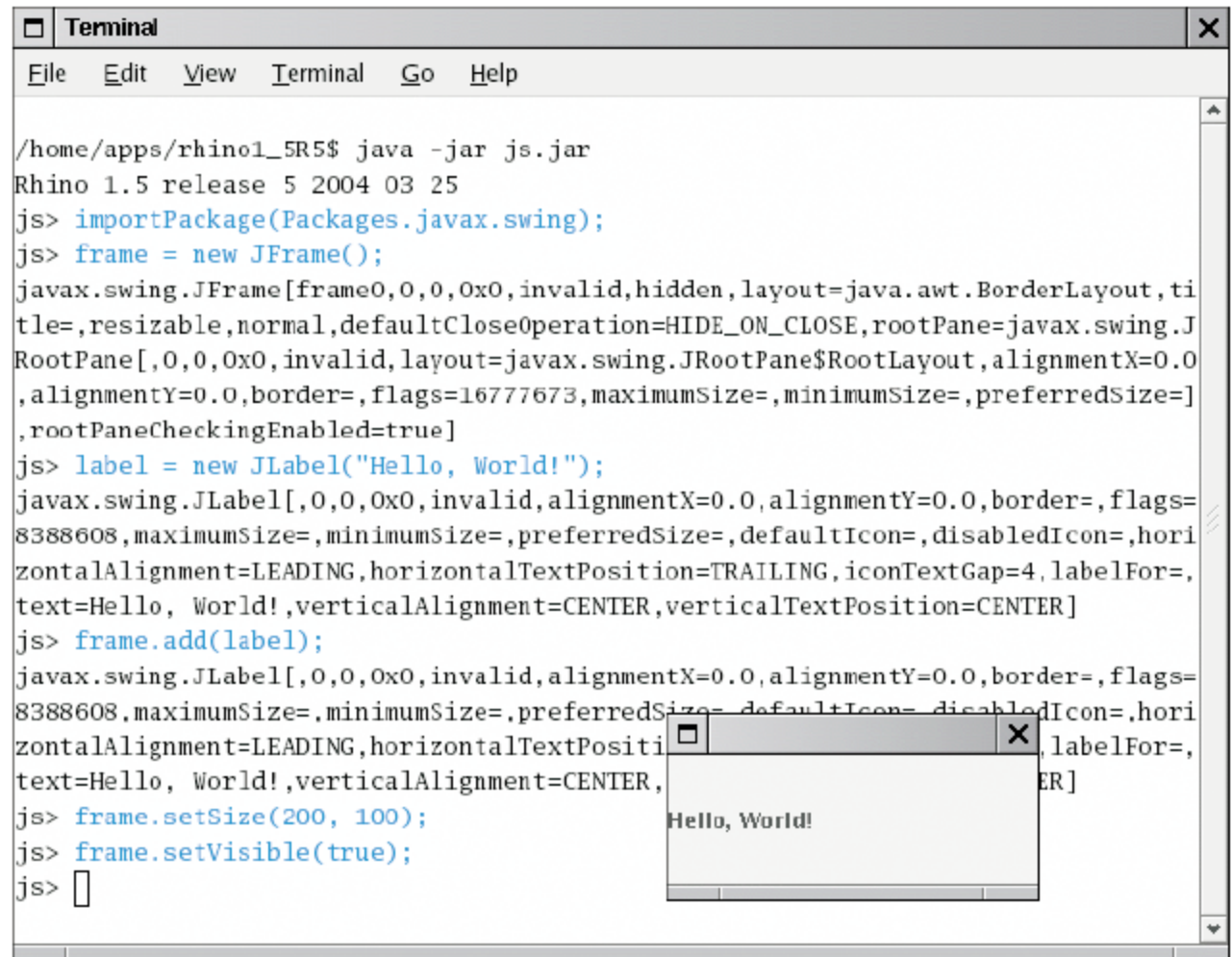
**Continued**

## The `Object.clone` method (cont.)

---

- 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 Advanced Topic 10.6)

# Scripting Languages



The screenshot shows a terminal window titled "Terminal" with a menu bar (File, Edit, View, Terminal, Go, Help). The terminal output shows the execution of a Java script using the Rhino interpreter. The script creates a JFrame window titled "Hello, World!" with a JLabel containing the text "Hello, World!". The window is 200x100 pixels and is visible. The terminal output is as follows:

```
/home/apps/rhino1_5R5$ java -jar js.jar
Rhino 1.5 release 5 2004 03 25
js> importPackage(Packages.java.swing);
js> frame = new JFrame();
javax.swing.JFrame[frame0,0,0,0x0,invalid,hidden,layout=java.awt.BorderLayout,tit
tle=,resizable,normal,defaultCloseOperation=HIDE_ON_CLOSE,rootPane=javax.swing.J
RootPane[,0,0,0x0,invalid,layout=javax.swing.JRootPane$RootLayout,alignmentX=0.0
,alignmentY=0.0,border=,flags=16777673,maximumSize=,minimumSize=,preferredSize=]
,rootPaneCheckingEnabled=true]
js> label = new JLabel("Hello, World!");
javax.swing.JLabel[,0,0,0x0,invalid,alignmentX=0.0,alignmentY=0.0,border=,flags=
8388608,maximumSize=,minimumSize=,preferredSize=,defaultIcon=,disabledIcon=,hori
zontalAlignment=LEADING,horizontalTextPosition=TRAILING,iconTextGap=4,labelFor=,
text=Hello, World!,verticalAlignment=CENTER,verticalTextPosition=CENTER]
js> frame.add(label);
javax.swing.JLabel[,0,0,0x0,invalid,alignmentX=0.0,alignmentY=0.0,border=,flags=
8388608,maximumSize=,minimumSize=,preferredSize=,defaultIcon=,disabledIcon=,hori
zontalAlignment=LEADING,horizontalTextPosition=TRAILING,iconTextGap=4,labelFor=,
text=Hello, World!,verticalAlignment=CENTER,verticalTextPosition=CENTER]
js> frame.setSize(200, 100);
js> frame.setVisible(true);
js> 
```

Overlaid on the terminal window is a small Java Swing window titled "Hello, World!". The window contains a single line of text "Hello, World!" and has a standard Mac OS X-style title bar with minimize, maximize, and close buttons.

Writing a Rhino Script

## 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 fields
- Initialize them in the constructor of your subclass
- If initialization code gets complex, simply add some helper methods

## Example: Investment Viewer Program

---

Lauren – I'm not sure what is supposed to go here.

## Example: Investment Viewer Program

---

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

Lauren – I'm not sure what is supposed to go here.

## Example: Investment Viewer Program (cont.)

---

Lauren – I'm not sure what is supposed to go here.

## Self Check 10.18

---

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 10.19

---

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*.

## Processing Text Input

---

- Use `JTextField` components to provide space for user input

```
final int FIELD_WIDTH = 10; // In characters
final JTextField rateField = new
    JTextField(FIELD_WIDTH);
```

- Place a `JLabel` next to each text field

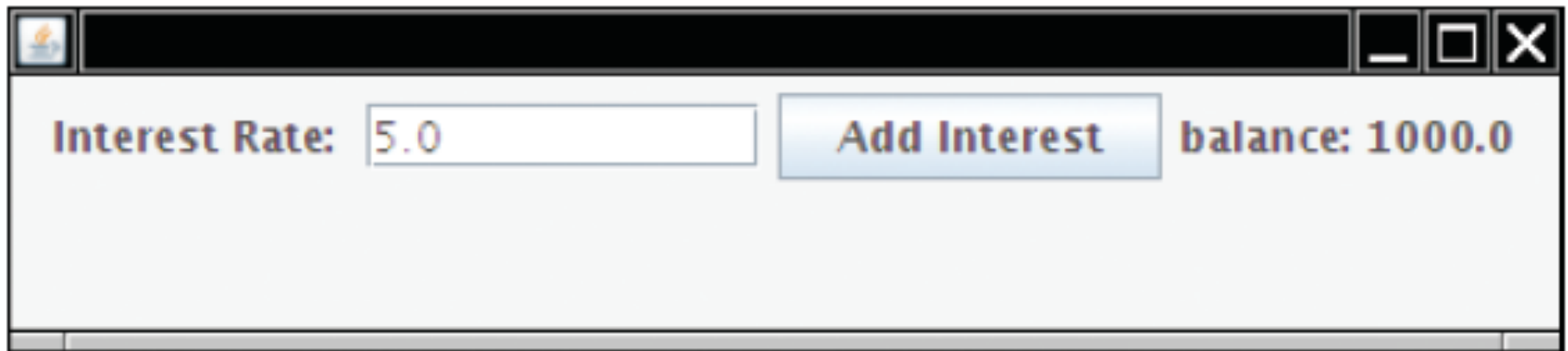
```
JLabel rateLabel = new JLabel("Interest Rate: ");
```

***Continued***

## Processing Text Input

---

- Supply a button that the user can press to indicate that the input is ready for processing



**Figure 12** An Application with a Text Field

***Continued***

## Processing Text Input (cont.)

---

- The button's `actionPerformed` method reads the user input from the text fields (use `getText`)

```
Class AddInterestListener implements ActionListener
{
    public void actionPerformed(ActionEvent event)
    {
        double rate =
            Double.parseDouble(rateField.getText());
        . . .
    }
}
```

## ch10/textfield/InvestmentViewer3.java

---

```
01: import javax.swing.JFrame;
02:
03: /**
04:     This program displays the growth of an investment.
05: */
06: public class InvestmentViewer3
07: {
08:     public static void main(String[] args)
09:     {
10:         JFrame frame = new InvestmentFrame();
11:         frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
12:         frame.setVisible(true);
13:     }
14: }
```

## ch10/textfield/InvestmentFrame.java

---

```
01: import java.awt.event.ActionEvent;
02: import java.awt.event.ActionListener;
03: import javax.swing.JButton;
04: import javax.swing.JFrame;
05: import javax.swing.JLabel;
06: import javax.swing.JPanel;
07: import javax.swing.JTextField;
08:
09: /**
10:     A frame that shows the growth of an investment with variable
11:     interest.
12: */
13: public class InvestmentFrame extends JFrame
14: {
15:     public InvestmentFrame()
16:     {
17:         account = new BankAccount(INITIAL_BALANCE);
18:
19:         // Use instance fields for components
20:         resultLabel = new JLabel("balance: " + account.getBalance());
```

**Continued**

*Big Java* by Cay Horstmann

Copyright © 2008 by John Wiley & Sons. All rights reserved.

## ch10/textfield/InvestmentFrame.java (cont.)

---

```
21:         // Use helper methods
22:         createTextField();
23:         createButton();
24:         createPanel();
25:
26:         setSize(FRAME_WIDTH, FRAME_HEIGHT);
27:     }
28:
29:     private void createTextField()
30:     {
31:         rateLabel = new JLabel("Interest Rate: ");
32:
33:         final int FIELD_WIDTH = 10;
34:         rateField = new JTextField(FIELD_WIDTH);
35:         rateField.setText("" + DEFAULT_RATE);
36:     }
37:
38:     private void createButton()
39:     {
40:         button = new JButton("Add Interest");
41:
```

**Continued**

*Big Java* by Cay Horstmann

Copyright © 2008 by John Wiley & Sons. All rights reserved.

## ch10/textfield/InvestmentFrame.java (cont.)

---

```
42:         class AddInterestListener implements ActionListener
43:         {
44:             public void actionPerformed(ActionEvent event)
45:             {
46:                 double rate = Double.parseDouble(
47:                     rateField.getText());
48:                 double interest = account.getBalance()
49:                     * rate / 100;
50:                 account.deposit(interest);
51:                 resultLabel.setText(
52:                     "balance: " + account.getBalance());
53:             }
54:         }
55:
56:         ActionListener listener = new AddInterestListener();
57:         button.addActionListener(listener);
58:     }
59:
60:     private void createPanel()
```

**Continued**

*Big Java* by Cay Horstmann

Copyright © 2008 by John Wiley & Sons. All rights reserved.

## ch10/textfield/InvestmentFrame.java (cont.)

---

```
61:     {
62:         panel = new JPanel();
63:         panel.add(rateLabel);
64:         panel.add(rateField);
65:         panel.add(button);
66:         panel.add(resultLabel);
67:         add(panel);
68:     }
69:
70:     private JLabel rateLabel;
71:     private JTextField rateField;
72:     private JButton button;
73:     private JLabel resultLabel;
74:     private JPanel panel;
75:     private BankAccount account;
76:
77:     private static final int FRAME_WIDTH = 450;
78:     private static final int FRAME_HEIGHT = 100;
79:
80:     private static final double DEFAULT_RATE = 5;
81:     private static final double INITIAL_BALANCE = 1000;
82: }
```

## Self Check 10.20

---

What happens if you omit the first `JLabel` object?

**Answer:** Then the text field is not labeled, and the user will not know its purpose.

## Self Check 10.21

---

If a text field holds an integer, what expression do you use to read its contents?

**Answer:** `Integer.parseInt(textField.getText())`

## Text Areas

---

- Use a `JTextArea` to show multiple lines of text
- You can specify the number of rows and columns:

```
final int ROWS = 10;  
final int COLUMNS = 30;  
JTextArea textArea = new JTextArea(ROWS, COLUMNS);
```

- `setText`: to set the text of a text field or text area
- `append`: to add text to the end of a text area
- Use `newline` characters to separate lines:

```
textArea.append(account.getBalance() + "\n");
```

- To use for display purposes only:

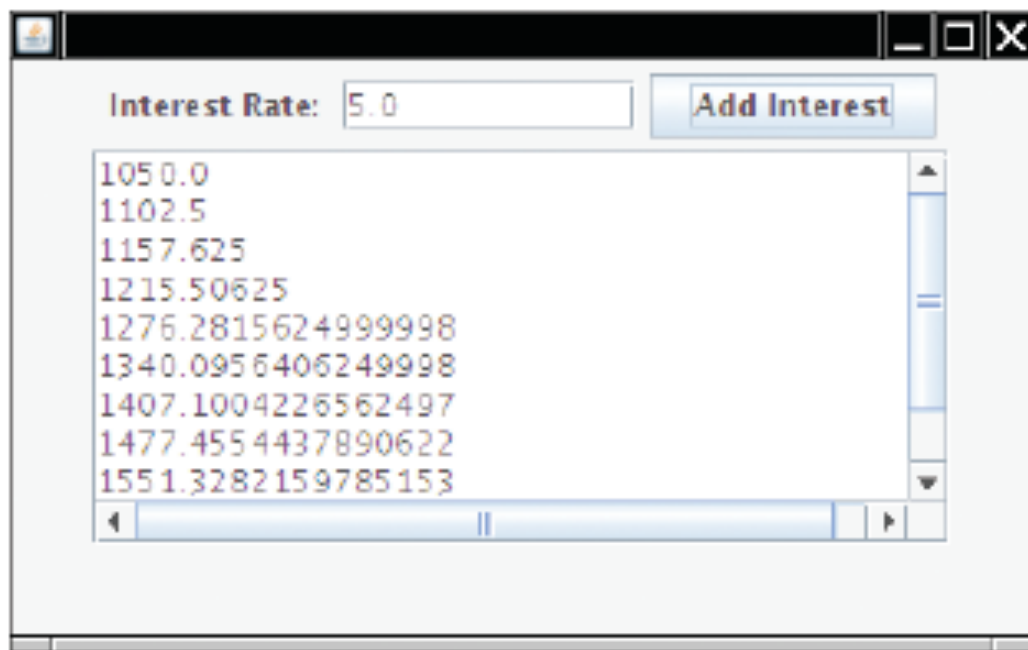
```
textArea.setEditable(false); // program can call setText  
and append to change it
```

## Text Areas

---

- To add scroll bars to a text area:

```
JTextArea textArea = new JTextArea(ROWS, COLUMNS);  
JScrollPane scrollPane = new JScrollPane(textArea);
```



**Figure 13** The Investment Application with a Text Area

## ch10/textarea/InvestmentFrame.java

---

```
01: import java.awt.event.ActionEvent;
02: import java.awt.event.ActionListener;
03: import javax.swing.JButton;
04: import javax.swing.JFrame;
05: import javax.swing.JLabel;
06: import javax.swing.JPanel;
07: import javax.swing.JScrollPane;
08: import javax.swing.JTextArea;
09: import javax.swing.JTextField;
10:
11: /**
12:     A frame that shows the growth of an investment with variable
13:     interest.
14: */
15: public class InvestmentFrame extends JFrame
16: {
17:     public InvestmentFrame()
18:     {
19:         account = new BankAccount(INITIAL_BALANCE);
20:         resultArea = new JTextArea(AREA_ROWS, AREA_COLUMNS);
21:         resultArea.setEditable(false);
22:     }
23: }
```

**Continued**

*Big Java* by Cay Horstmann

Copyright © 2008 by John Wiley & Sons. All rights reserved.

## ch10/textarea/InvestmentFrame.java (cont.)

---

```
22:         // Use helper methods
23:         createTextField();
24:         createButton();
25:         createPanel();
26:
27:         setSize(FRAME_WIDTH, FRAME_HEIGHT);
28:     }
29:
30:     private void createTextField()
31:     {
32:         rateLabel = new JLabel("Interest Rate: ");
33:
34:         final int FIELD_WIDTH = 10;
35:         rateField = new JTextField(FIELD_WIDTH);
36:         rateField.setText("" + DEFAULT_RATE);
37:     }
38:
39:     private void createButton()
40:     {
41:         button = new JButton("Add Interest");
42:
```

**Continued**

*Big Java* by Cay Horstmann

Copyright © 2008 by John Wiley & Sons. All rights reserved.

## ch10/textarea/InvestmentFrame.java (cont.)

---

```
43:         class AddInterestListener implements ActionListener
44:         {
45:             public void actionPerformed(ActionEvent event)
46:             {
47:                 double rate = Double.parseDouble(
48:                     rateField.getText());
49:                 double interest = account.getBalance()
50:                     * rate / 100;
51:                 account.deposit(interest);
52:                 resultArea.append(account.getBalance() + "\n");
53:             }
54:         }
55:
56:         ActionListener listener = new AddInterestListener();
57:         button.addActionListener(listener);
58:     }
59:
60:     private void createPanel()
61:     {
62:         panel = new JPanel();
63:         panel.add(rateLabel);
64:         panel.add(rateField);
65:         panel.add(button);
```

**Continued**

*Big Java* by Cay Horstmann

Copyright © 2008 by John Wiley & Sons. All rights reserved.

## ch10/textarea/InvestmentFrame.java (cont.)

---

```
66:         JScrollPane scrollPane = new JScrollPane(resultArea);
67:         panel.add(scrollPane);
68:         add(panel);
69:     }
70:
71:     private JLabel rateLabel;
72:     private JTextField rateField;
73:     private JButton button;
74:     private JTextArea resultArea;
75:     private JPanel panel;
76:     private BankAccount account;
77:
78:     private static final int FRAME_WIDTH = 400;
79:     private static final int FRAME_HEIGHT = 250;
80:
81:     private static final int AREA_ROWS = 10;
82:     private static final int AREA_COLUMNS = 30;
83:
84:     private static final double DEFAULT_RATE = 5;
85:     private static final double INITIAL_BALANCE = 1000;
86: }
```

## Self Check 10.22

---

What is the difference between a text field and a text area?

**Answer:** A text field holds a single line of text; a text area holds multiple lines.

## Self Check 10.23

---

Why did the `InvestmentFrame` program call `resultArea.setEditable(false)`?

**Answer:** The text area is intended to display the program output. It does not collect user input.

## Self Check 10.24

---

How would you modify the `InvestmentFrame` program if you didn't want to use scroll bars?

**Answer:** Don't construct a `JScrollPane` and add the `resultArea` object directly to the frame.