

Chapter 13

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

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
}
```

Continued...

An Introduction to Inheritance

- **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
```

- **Extended class = superclass (BankAccount),
extending class = subclass (Savings)**

Continued...

An Introduction to Inheritance

- **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**

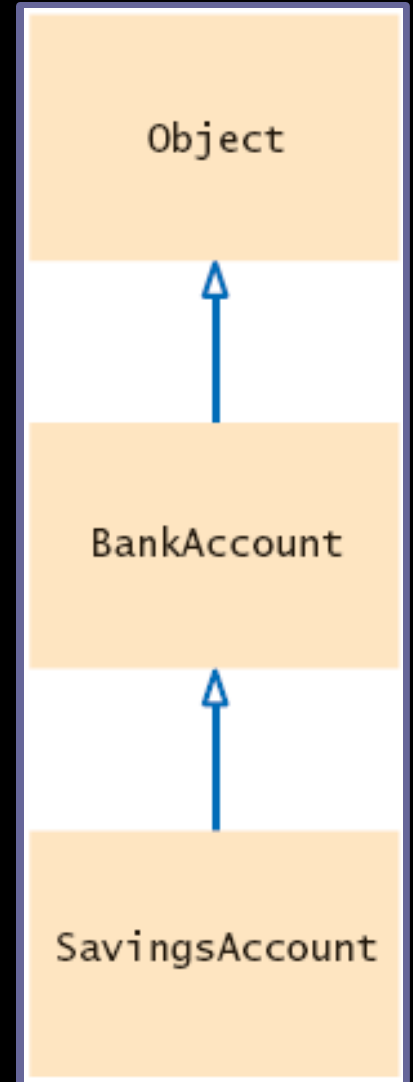


Figure 1:
An Inheritance Diagram

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);
    }

    private double interestRate;
}
```

An Introduction to Inheritance

- **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:

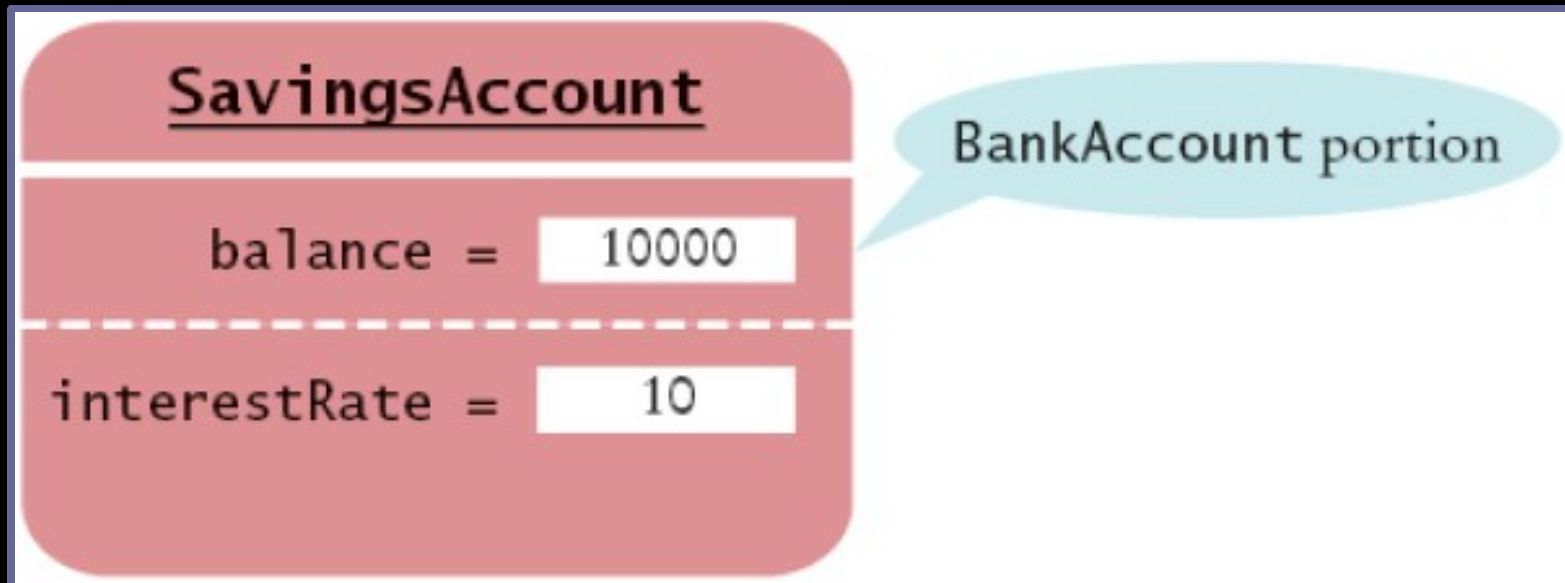


Figure 2:
Layout of a Subclass Object

Syntax 13.1: Inheritance

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

Continued...

Syntax 13.1: Inheritance

Example:

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

    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

1. Which instance fields does an object of class `SavingsAccount` have?
2. Name four methods that you can apply to `SavingsAccount` objects
3. If the class `Manager` extends the class `Employee`, which class is the superclass and which is the subclass?

Answers

1. **Two instance fields: balance and interestRate.**
2. **deposit, withdraw, getBalance, and addInterest.**
3. **Manager is the subclass; Employee is the superclass.**

Inheritance Hierarchies

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

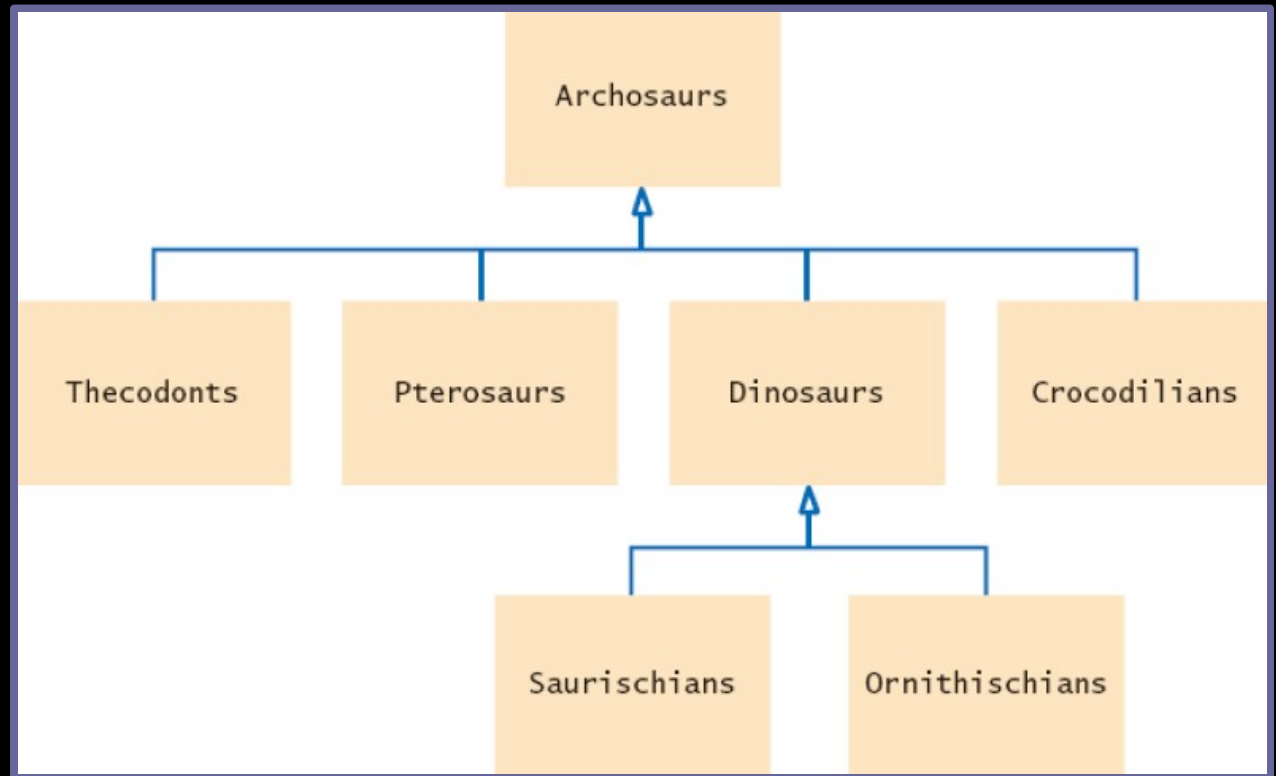


Figure 3:
A Part of the Hierarchy of Ancient Reptiles

Inheritance Hierarchies Example: Swing hierarchy

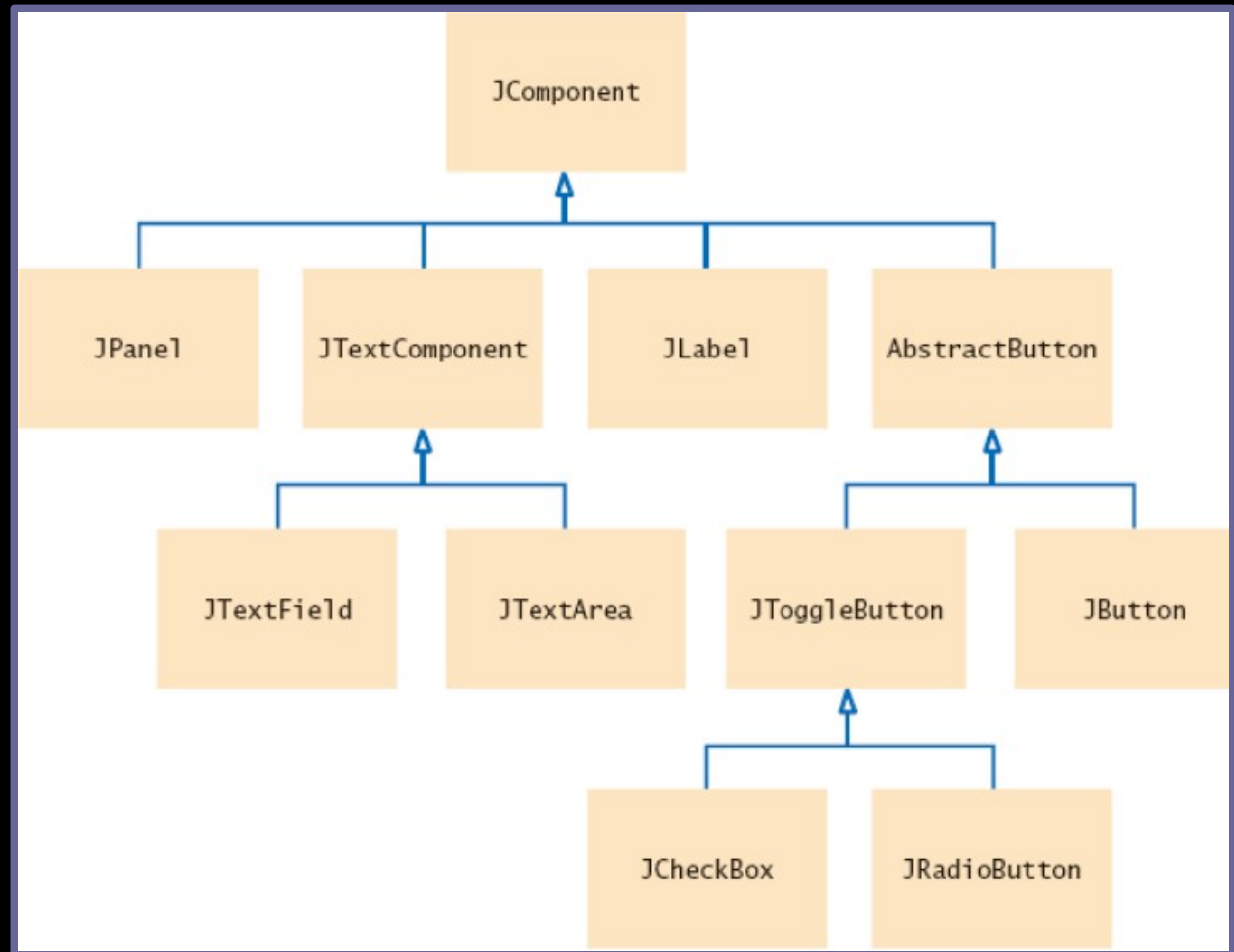


Figure 4:
A Part of the Hierarchy
of Swing User
Interface Components

Continued...

Inheritance Hierarchies Example: Swing hierarchy

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

A Simpler Hierarchy: 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

Continued...

A Simpler Hierarchy: Hierarchy of Bank Accounts

- **Inheritance hierarchy:**

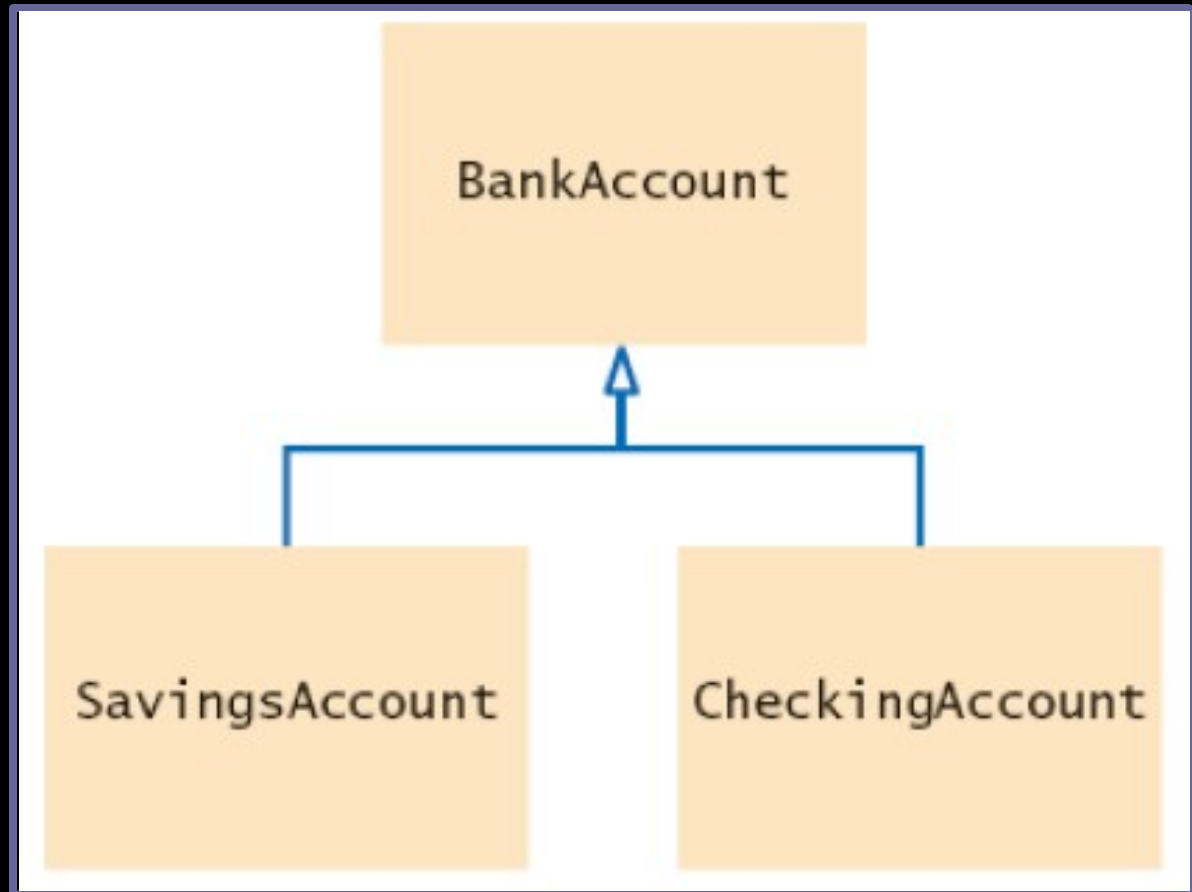


Figure 5:
Inheritance Hierarchy for Bank Account Classes

Continued...

A Simpler Hierarchy: Hierarchy of Bank Accounts

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

A Simpler Hierarchy:

Hierarchy of Bank Accounts

- 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`

Self Check

1. What is the purpose of the `JTextComponent` class in Figure 4?
2. Which instance field will we need to add to the `CheckingAccount` class?

Answers

- 1. To express the common behavior of text fields and text components.**
- 2. 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

Continued...

Inheriting Methods

- **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**

Continued...

Inheriting Instance Fields

- **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
}
```

Continued...

Implementing the CheckingAccount Class

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

Continued...

Implementing the CheckingAccount Class

- **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

Continued...

Inherited Fields Are Private

- **A subclass has no access to private fields of its superclass**
- **Subclass must use public interface**

Invoking a Super Class 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)`

Continued...

Invoking a Super Class Method

- **Now calls deposit method of BankAccount class**
- **Complete method:**

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

Syntax 13.2: 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);
    }
}
```

Continued...

Implementing Remaining Methods

```
public void deductFees()
{
    if (transactionCount > FREE_TRANSACTIONS)
    {
        double fees = TRANSACTION_FEE
            * (transactionCount - FREE_TRANSACTIONS);
        super.withdraw(fees);
    }
    transactionCount = 0;
}
. . .
private static final int FREE_TRANSACTIONS = 3;
private static final double TRANSACTION_FEE = 2.0;
}
```

Self Check

1. Why does the `withdraw` method of the `CheckingAccount` class call `super.withdraw`?
2. Why does the `deductFees` method set the transaction count to zero?

Answers

1. It needs to reduce the balance, and it cannot access the `balance` field directly.
2. 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

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

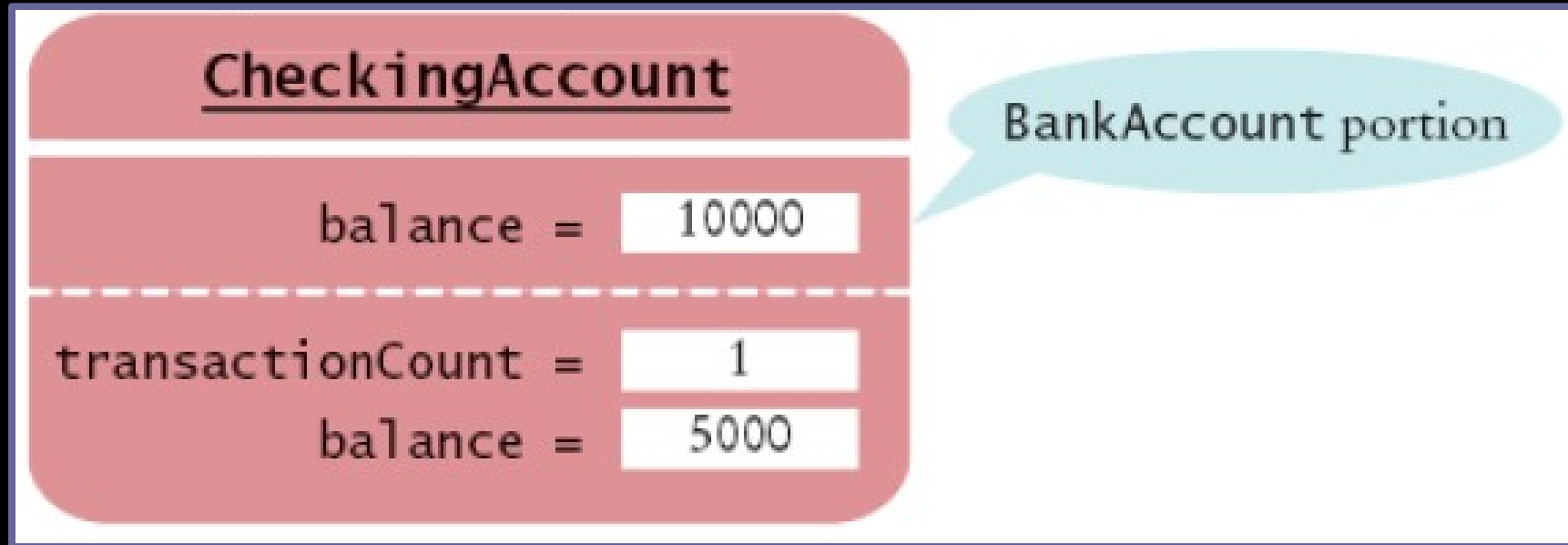


Figure 6:
Shadowing Instance Fields

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;
    }
    . . .
}
```

Continued...

Subclass Construction

- **Must be the *first* statement in subclass constructor**
- **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 13.1: 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

1. Why didn't the `SavingsAccount` constructor in Section 13.1 call its superclass constructor?
2. When you invoke a superclass method with the `super` keyword, does the call have to be the first statement of the subclass method?

Answers

1. It was content to use the default constructor of the superclass, which sets the balance to zero.
2. 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;
```

Converting Between Subclass and Superclass Types

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

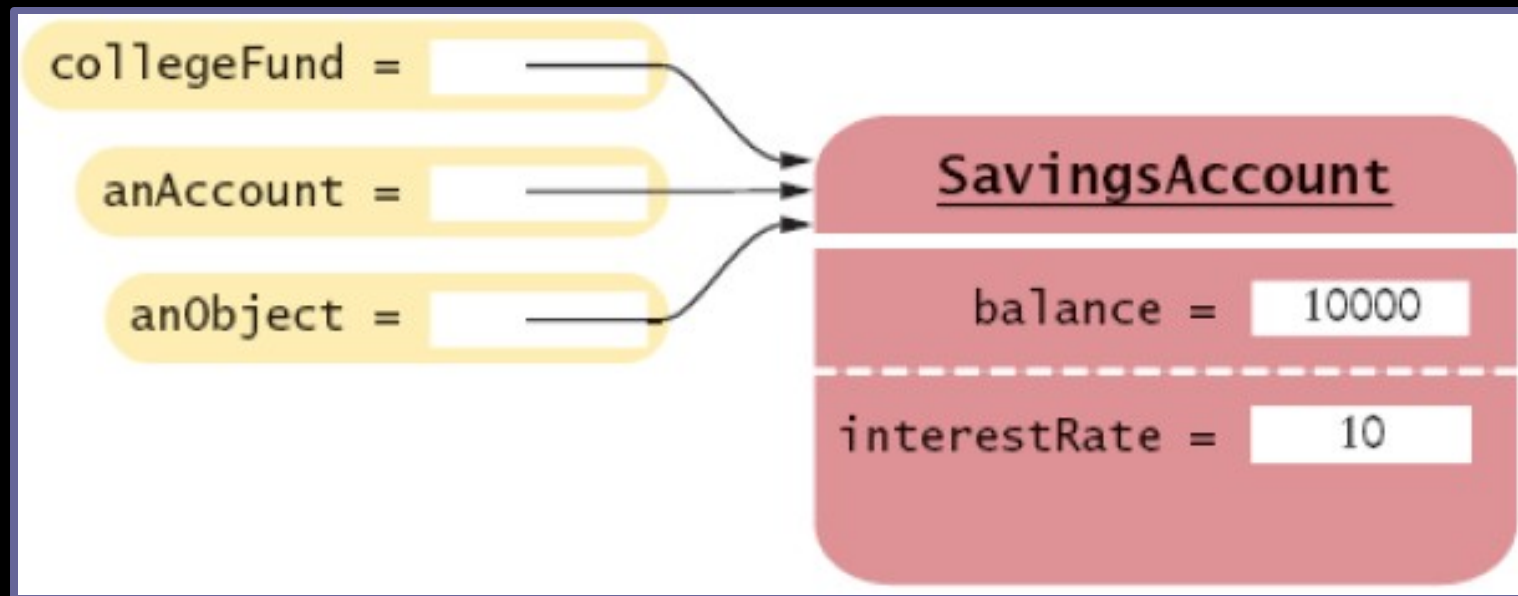


Figure 7:
Variables of Different Types
Refer to the Same Object

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

- **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

Continued...

Converting Between Subclass and Superclass Types

- **Solution: use the `instanceof` operator**
- **`instanceof`: tests whether an object belongs to a particular type**

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

Syntax 13.4: 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 Test

1. Why did the second parameter of the `transfer` method have to be of type `BankAccount` and not, for example, `SavingsAccount`?
2. Why can't we change the second parameter of the `transfer` method to the type `Object`?

Answers

1. 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.
2. 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
```

Continued...

Polymorphism

- **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**

File 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:
```

Continued...

File AccountTester.java

```
17:     momsSavings.transfer(2000, harrysChecking);
18:     harrysChecking.withdraw(1500);
19:     harrysChecking.withdraw(80);
20:
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:
31:     System.out.println("Harry's checking balance = $"
32:         + harrysChecking.getBalance());
33: }
34: }
```

File 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:     */
```

Continued...

File BankAccount.java

```
19: public BankAccount(double initialBalance)
20: {
21:     balance = initialBalance;
22: }
23:
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: */
```

Continued...

File BankAccount.java

```
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: */
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: */
```

Continued...

File BankAccount.java

```
56:     public void transfer(double amount, BankAccount other)
57:     {
58:         withdraw(amount);
59:         other.deposit(amount);
60:     }
61:
62:     private double balance;
63: }
```

File 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:
```

Continued...

File CheckingAccount.java

```
19: public void deposit(double amount)
20: {
21:     transactionCount++;
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: */
```

Continued...

File CheckingAccount.java

```
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:         }
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: }
```

File 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...

File SavingsAccount.java

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

Continued...

File SavingsAccount.java

Output:

```
Mom's savings balance = $7035.0  
Harry's checking balance = $1116.0
```

Self Check

1. 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?
2. If `a` refers to a checking account, what is the effect of calling `a.transfer(1000, a)`?

Answers

1. The object is an instance of `BankAccount` or one of its subclasses.
2. 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 13.3

Continued...

Access Control

- Java has four levels of controlling access to fields, methods, and classes:
 - 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

Continued...

Recommended Access Levels

- **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

1. **What is a common reason for defining package-visible instance fields?**
2. **If a class with a public constructor has package access, who can construct objects of it?**

Answers

1. **Accidentally forgetting the `private` modifier.**
2. **Any methods of classes in the same package.**

Object: The Cosmic Superclass

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

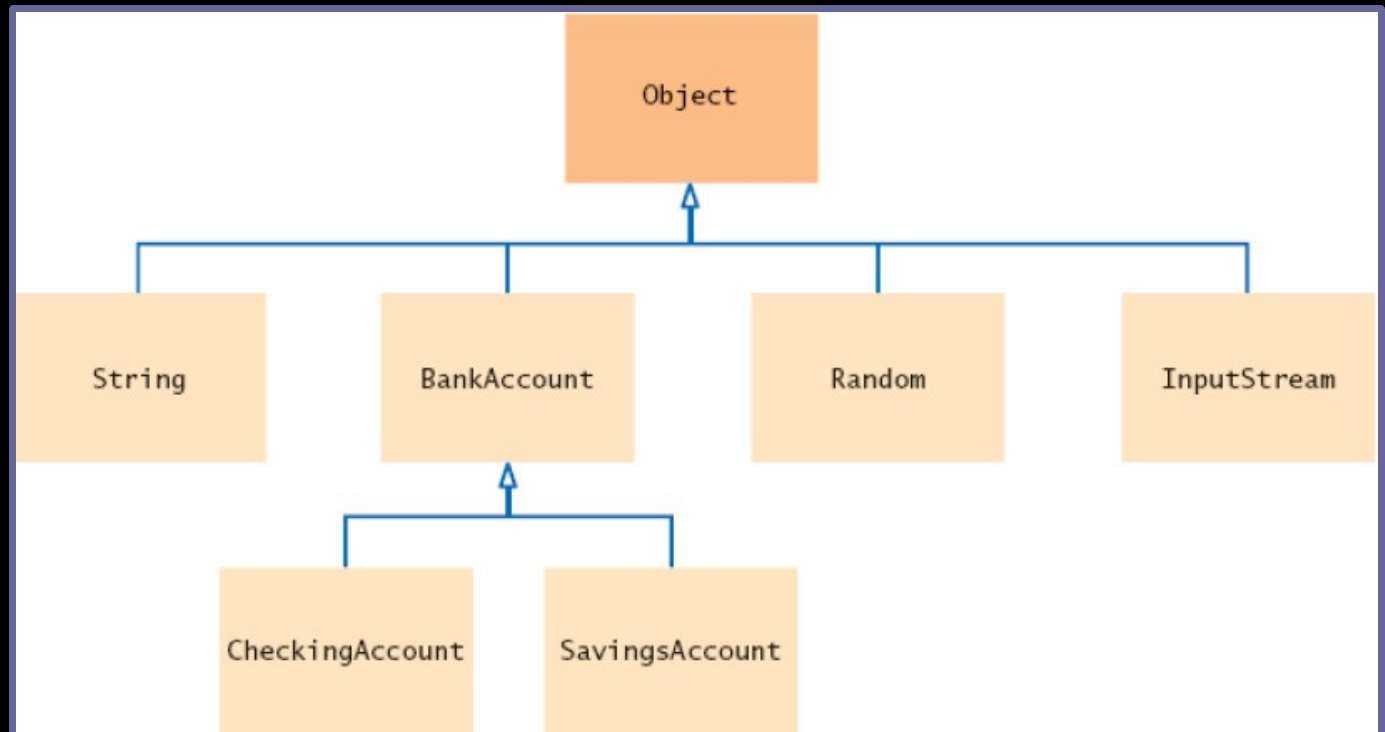


Figure 8:
The `Object` Class is the Superclass of Every Java Class

Object: The Cosmic Superclass

- **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]"
```

Continued...

Overriding the toString Method

- `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]"
```

- `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*

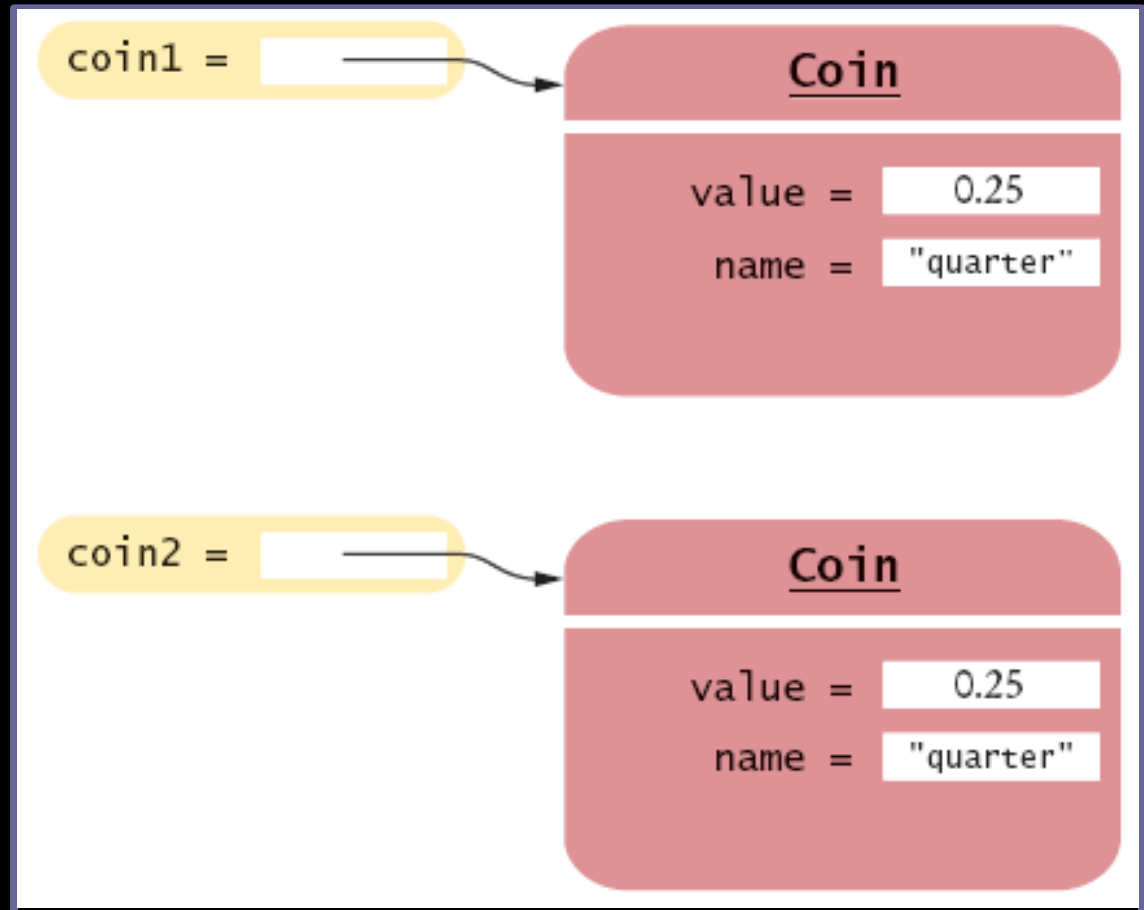


Figure 9:
Two References to
Equal Objects

Continued...

Overriding the equals Method

- `==` tests for equal *location*

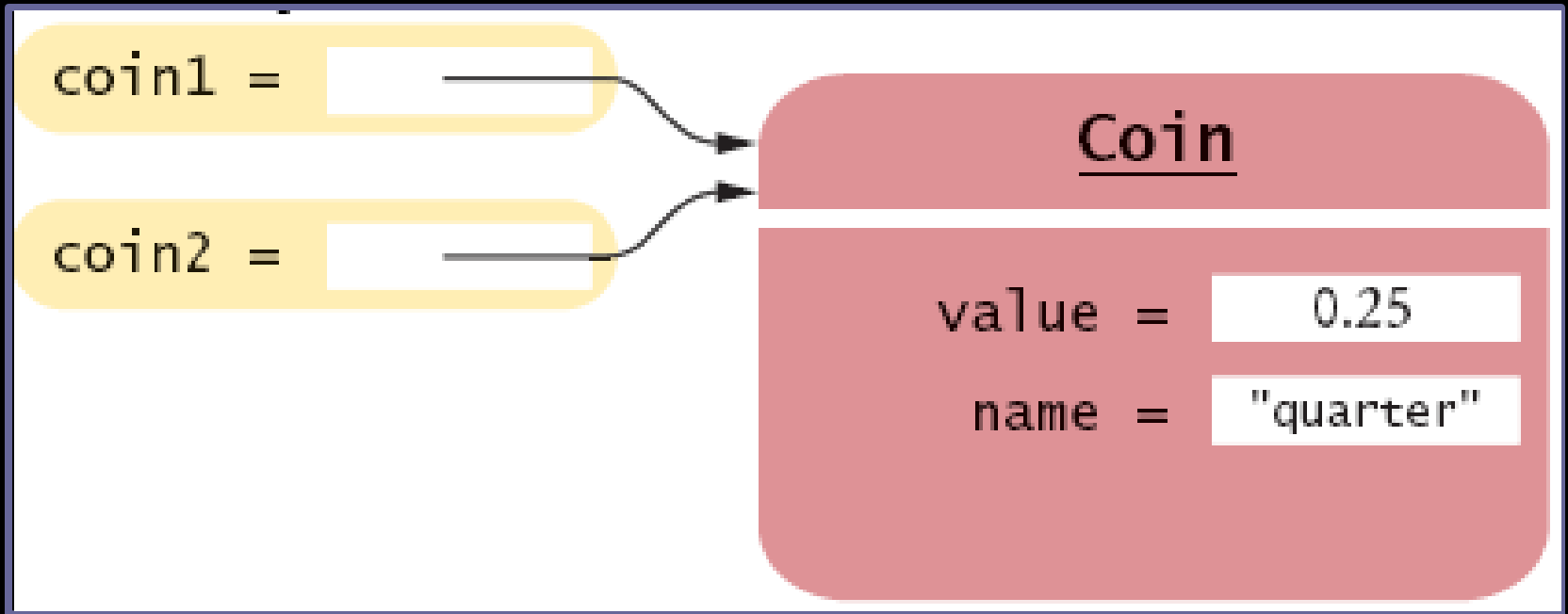


Figure 10:
Two References to the Same Object

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

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

Self Check

1. Should the call `x.equals(x)` always return `true`?
2. Can you implement `equals` in terms of `toString`? Should you?

Answers

1. It certainly should—unless, of course, `x` is `null`.
2. 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

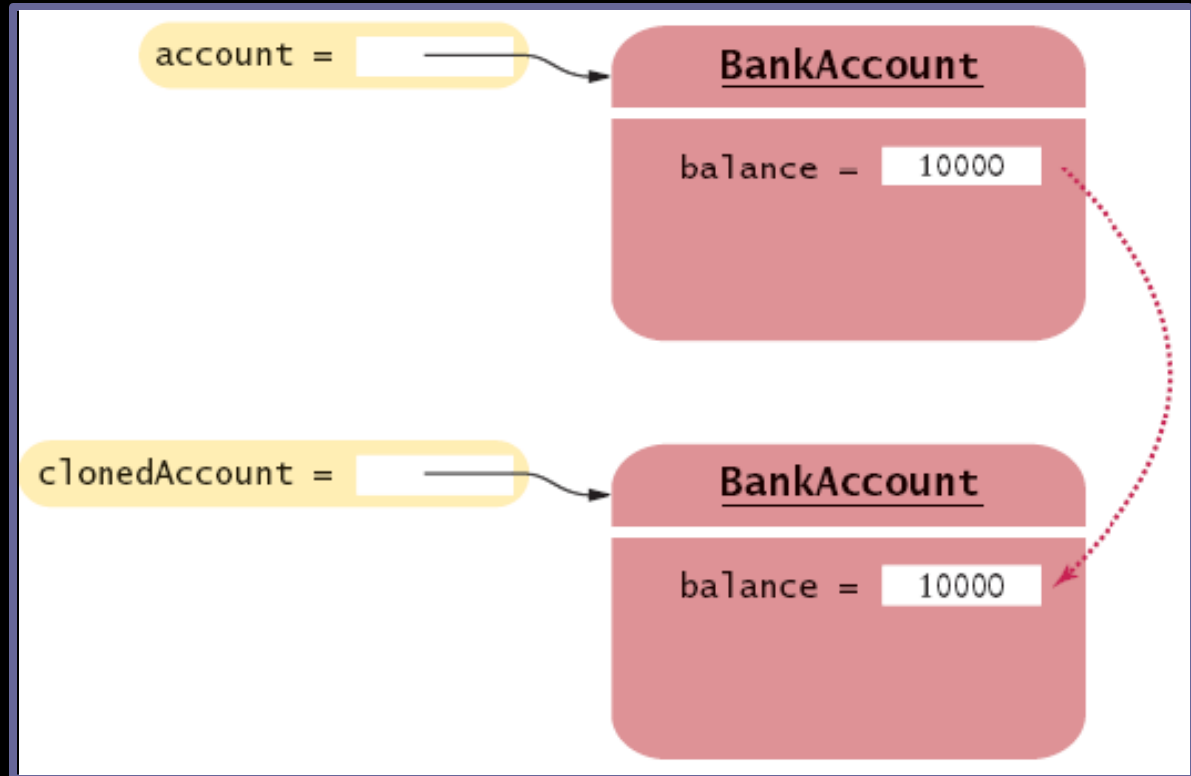
- Copying an object reference gives two references to same object

```
BankAccount account2 = account;
```

Continued...

Overriding the clone Method

- Sometimes, need to make a copy of the object



Object 11:
Cloning Objects

Continued...

Overriding the `clone` Method

- Define `clone` method to make new object (see Advanced Topic 13.6)
- Use `clone`:

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

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

The Object.clone Method

- Creates *shallow copies*

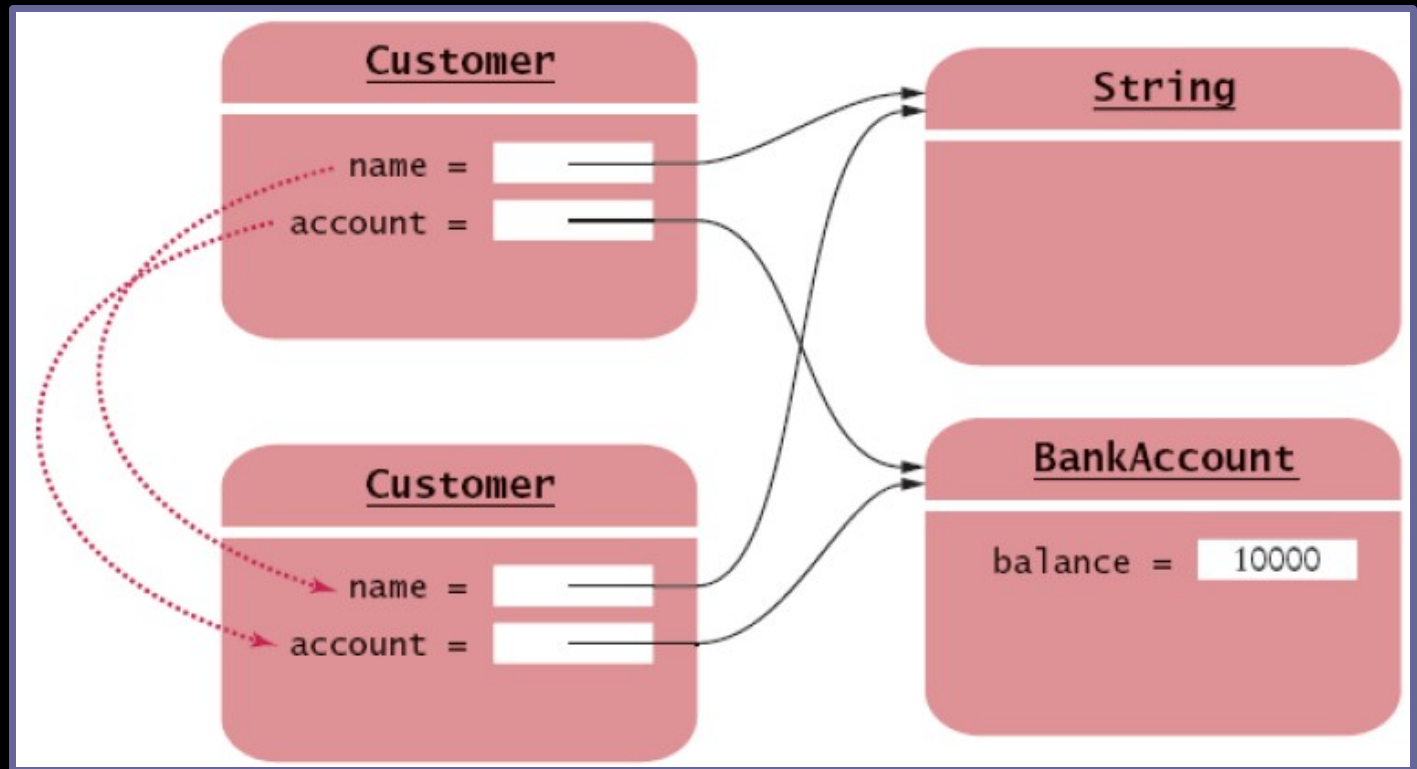


Figure 12:
The `Object.clone` Method Makes a Shallow Copy

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 Advanced Topic 13.6)

Scripting Languages
