Un'introduzione all'ereditarietà

- Ereditarietà: estende le classi aggiungendo metodi e variabili
- Esempio: Conto risparmio = conto bancario con interesse

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

- `SavingsAccount` automaticamente eredita tutti i metodi e le variabili di `BankAccount`

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

Continua

Un’introduzione all’ereditarietà (continua)

- Classe estesa = *superclass* (`BankAccount`), classe che estende = *subclass* (`Savings`)
- Ereditare da una classe ≠ implementare un’interfaccia: la sottoclasse eredita comportamento e stato
- Un vantaggio dell’ereditarietà è il riutilizzo del codice

Diagramma di ereditarietà

Ogni classe estende la classe `Object` direttamente o indirettamente

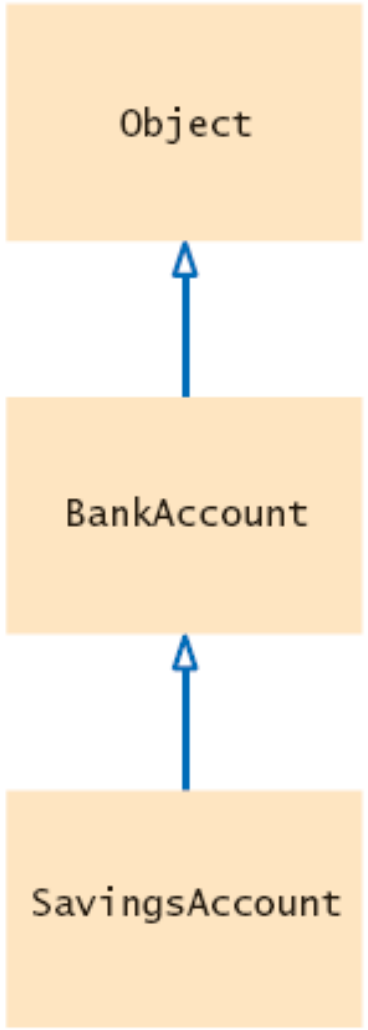


Figure 1
An Inheritance Diagram

Un'introduzione all'ereditarietà

- Nella `sottoclasse`, si specifichino le variabili di istanza, i metodi aggiunti ed i metodi cambiati o sovrascritti

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

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

Continua

Un’introduzione all’ereditarietà

```
   private double interestRate;
}
```

- Incapsulamento: `addInterest` chiama `getBalance` invece di aggiornare la variabile `balance` della superclasse (la variabile è `private`)
- Si noti che il metodo `addInterest` chiama `getBalance` senza specificare un parametro implicito (la chiamata si applica allo stesso oggetto)

Layout di un oggetto sottoclasse

L'oggetto `SavingsAccount` eredita la variabile di istanza `balance` da `BankAccount`, e guadagna un'ulteriore variabile di istanza `interestRate`:

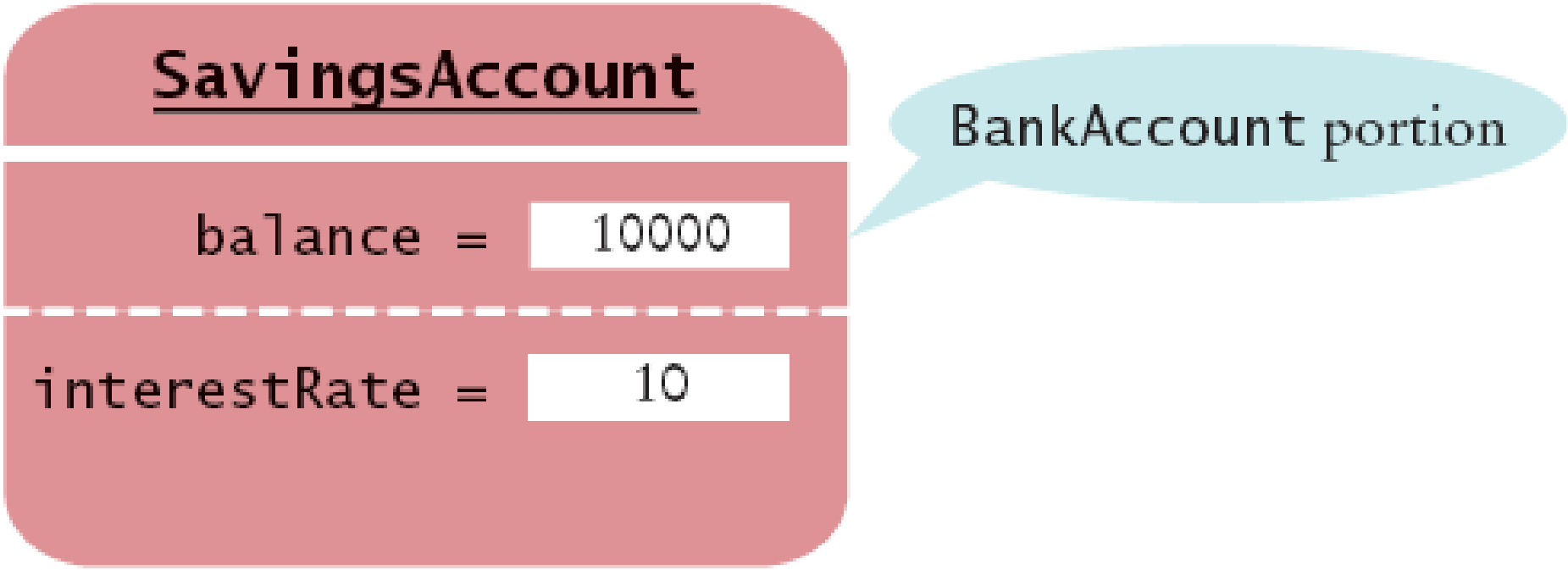


Figure 2 Layout of a Subclass Object

Sintassi 10.1 Ereditarietà

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

Esempio:

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

Continua

Sintassi 10.1 Ereditarietà

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

   private double interestRate;
}
```

Scopo:

Definire una nuova classe che eredita da una classe esistente, e definire i metodi e le variabili di istanza che sono aggiunti nella nuova classe.

Gerarchie di ereditarietà

- Insiemi di classi possono formare complesse gerarchie di ereditarietà
- Esempio:

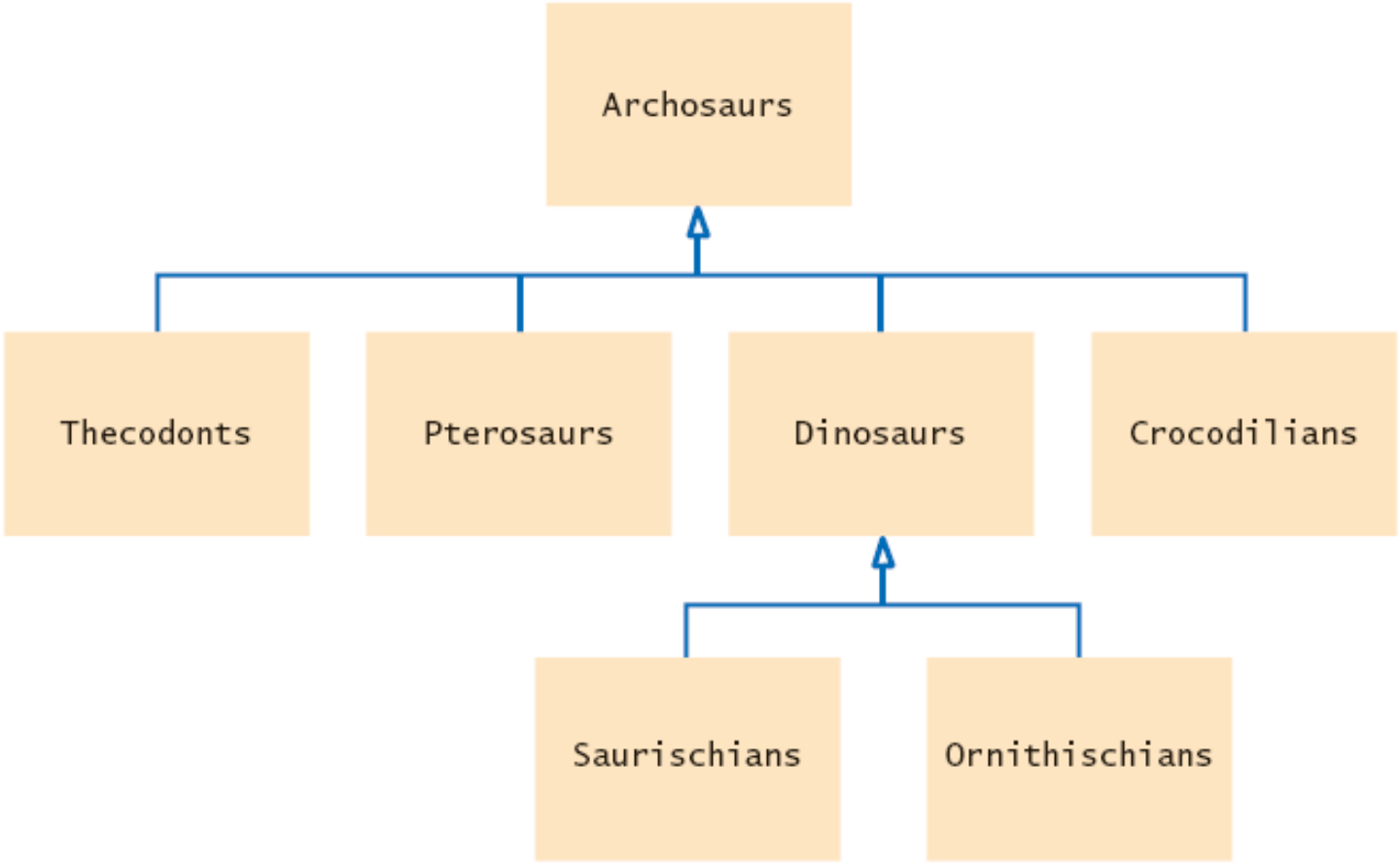


Figure 3 A Part of the Hierarchy of Ancient Reptiles

Esempio di gerarchie di ereditarietà: gerarchia Swing

- La superclasse `JComponent` ha i metodi `getWidth, getHeight`
- La classe `AbstractButton` ha i metodi per posizionare e rilevare il testo ed il disegno del pulsante

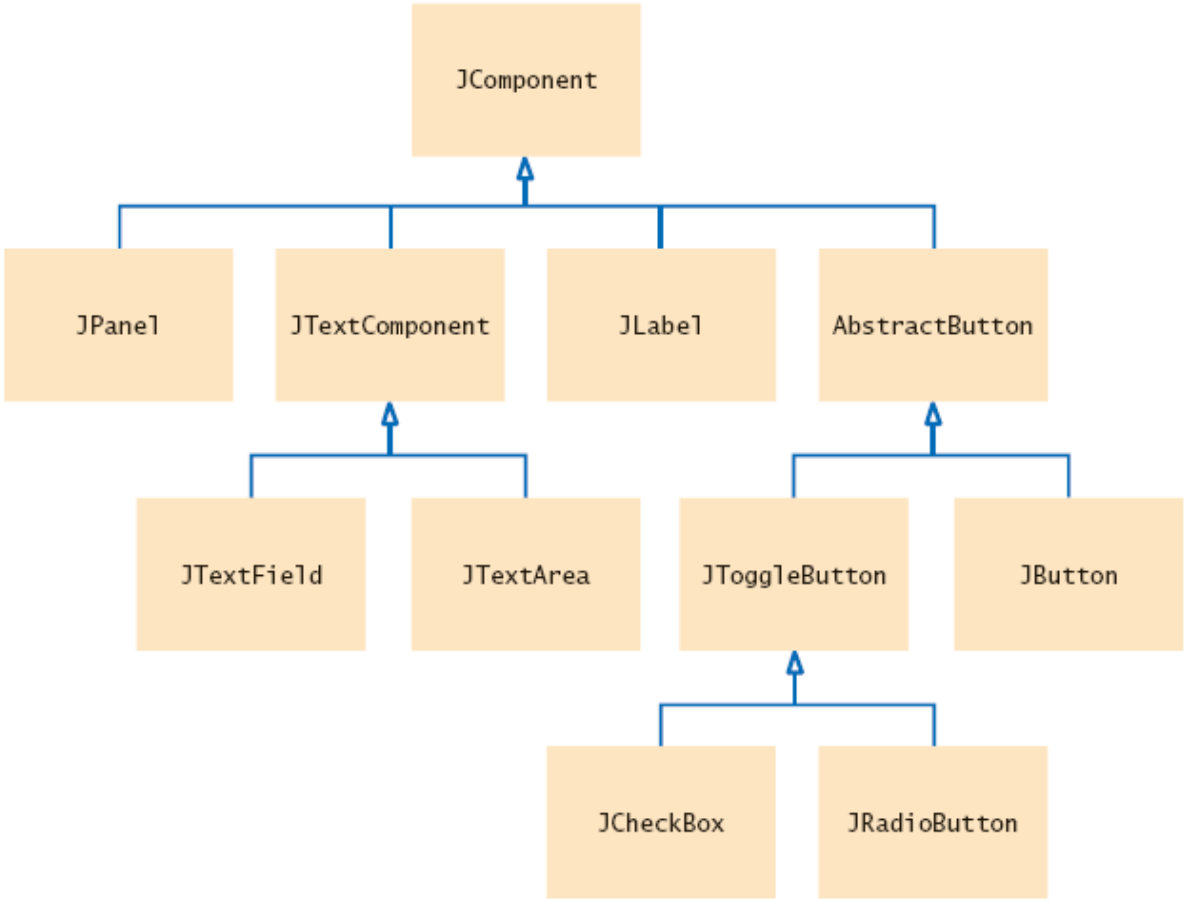


Figure 4 A Part of the Hierarchy of Swing User Interface Components

Un semplice esempio: gerarchia di conti correnti

- Si consideri una banca che offre ai suoi clienti i seguenti tipi di conti:
  1. *Conto corrente: nessun interesse, piccolo numero di operazioni gratuite al mese; per ulteriori operazioni addebito di una piccola commissione*
  2. *Conto di risparmio: matura un interesse che viene accreditato ogni mese*
- Gerarchia di ereditarietà:

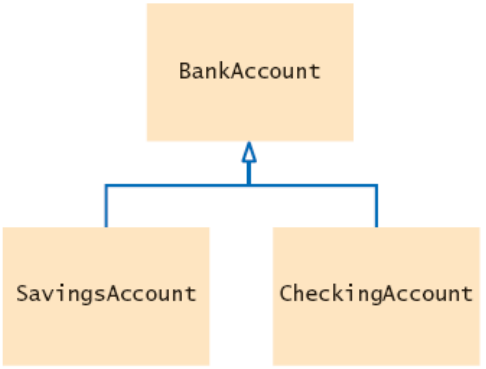


Figure 5 Inheritance Hierarchy for Bank Account Classes

- Entrambi i tipi di conti hanno il metodo *getBalance*
- Entrambi i tipi di conti hanno i metodi *deposit* e *withdraw*, ma con diversa struttura
- Il conto corrente necessita di un metodo *deductFees*, quello a risparmio di un metodo *addInterest*

Ereditare metodi

- Sovrascrivere un metodo:
  - *Fornire una diversa implementazione di un metodo che esiste nella* `superclasse`
  - *Deve avere la stessa firma (stesso nome e stessi tipi di parametri)*
  - *Se il metodo è applicato ad un oggetto del tipo della* `sottoclasse`*, viene eseguito il metodo che sovrascrive e non quello sovrascritto*
- Ereditare un metodo:
  - *Non fornire una nuova implementazione di un metodo che esiste nella* `superclasse`
  - *Il metodo della* `superclasse` *può essere applicato agli oggetti della sottoclasse*
- Aggiungere un metodo:
  - *Fornire un nuovo metodo che non esiste nella* `superclasse`
  - *Il nuovo metodo può essere applicato soltanto ad oggetti della* `sottoclasse`

Ereditare variabili di istanza

- Non si possono sovrascrivere variabili
- Ereditarietà delle variabili: tutte le variabili di istanza della superclasse sono ereditate automaticamente
- Aggiungere una variabile: fornire una nuova variabile che non esiste nella superclasse
- Cosa succede se si definisce una nuova variabile con lo stesso nome di una variabile nella superclasse?
  - *Ogni oggetto avrebbe due variabili di istanza con lo stesso nome*
  - *Le variabili possono contenere valori differenti*
  - *Legale ma estremamente indesiderabile*

Implementare la classe `CheckingAccount`

- Sovrascrive `deposit` e `withdraw` per incrementare il conteggio della transazioni:

```
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 }
```

- Ogni oggetto `CheckingAccount` ha due variabili di istanza:
  - *`balance`* *(ereditato da* *`BankAccount`*`)`
  - *`transactionCount`* *(nuovo per* *`CheckingAccount`*`)`

Continua

Implementare la classe `CheckingAccount` (continua)

- Si possono applicare agli oggeti `CheckingAccount` 4 metodi:
  - *`getBalance()` (ereditato da `BankAccount`)*
  - *`deposit(double amount)` (sovrascrive il metodo di `BankAccount`)*
  - *`withdraw(double amount)` (sovrascrive il metodo di `BankAccount`)*
  - *`deductFees()` (nuovo per `CheckingAccount`)*

Le variabili ereditate sono private

- Si consideri il metodo `deposit` di `CheckingAccount`

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

- Non si può semplicemente aggiungere `amount` a `balance`
- `balance` è una variabile *private* della superclasse
- Una sottoclasse non ha accesso alle variabili private della superclasse
- Le sottoclassi devono usare l'interfaccia pubblica

Richiamare un metodo della superclasse

- Non si può semplicemente chiamare

  `deposit(amount)`

  nel metodo `deposit` di `CheckingAccount`
- Equivale infatti a

  `this.deposit(amount`)
- Chiama il metodo stesso (ricorsione infinita)
- Invece, si chiami il metodo della *superclasse*

  `super.deposit(amount)`
- In questo modo si chiama il metodo `deposit` della classe `BankAccount`

Continua

Richiamare un metodo della superclasse (continua)

- Il metodo completo:

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

Sintassi 10.2 Chiamare un metodo della superclasse

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

Esempio:

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

Scopo:

Chiamare un metodo della superclasse invece di un metodo della classe corrente

Implementazione dei metodi rimanenti

```
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);
      }
```

Continua

Implementazione dei metodi rimanenti (continua)

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

Errore comune: mettere in ombra variabili di istanza

- Una sottoclasse non ha accesso alle variabili di istanza private della superclasse
- Errore comune: “risolvere" questo problema aggiungendo un’altra variabile di istanza con lo stesso nome:

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

Continua

Errore comune: mettere in ombra variabili di istanza (cont.)

- Ora il metodo *deposit* viene compilato, ma esso non aggiorna la variabile *balance* corretta!

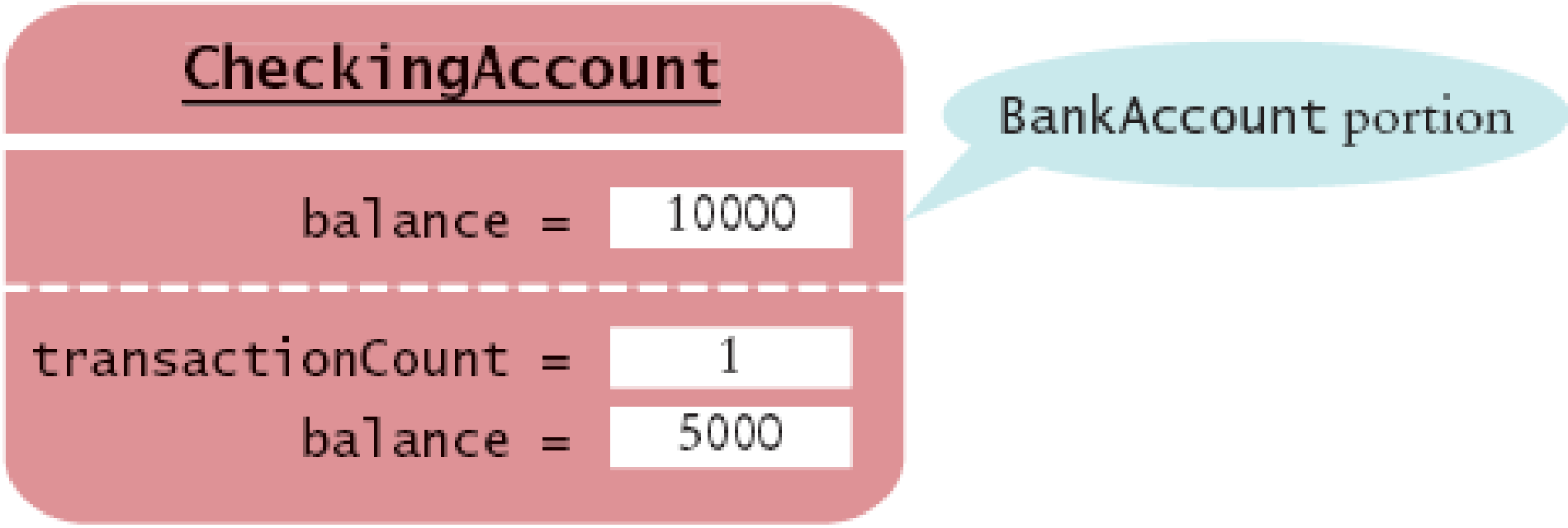


Figure 6 Shadowing Instance Fields

Costruzione di sottoclassi

- `super` seguito da parentesi indica una chiamata al costruttore della `superclasse`

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

- Deve essere il *primo* enunciato nel costruttore della sottoclasse

Continua

Costruzione di sottoclassi (continua)

- Se il costruttore della sottoclasse non chiama il costruttore della superclasse, viene impiegato il costruttore di default della superclasse
  - *Costruttore di default: costruttore senza parametri*
  - *Se tutti i costruttori della superclasse richiedono parametri, allora il compilatore segnala un errore*

Sintassi 10.3 Chiamare un costruttore di superclasse

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

Esempio:

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

Scopo:

Chiamare il costruttore della superclasse. Si noti che questo enunciato deve essere il primo del costruttore della sottoclasse.

Conversione tra tipi sottoclasse e superclasse

- Ok per convertire riferimenti sottoclasse a riferimenti superclasse

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

- I tre riferimenti memorizzati in `collegeFund`, `anAccount`, e `anObject` si riferiscono tutti allo stesso oggetto di tipo `SavingsAccount`

Continua

Conversione tra tipi sottoclasse e superclasse (continua)

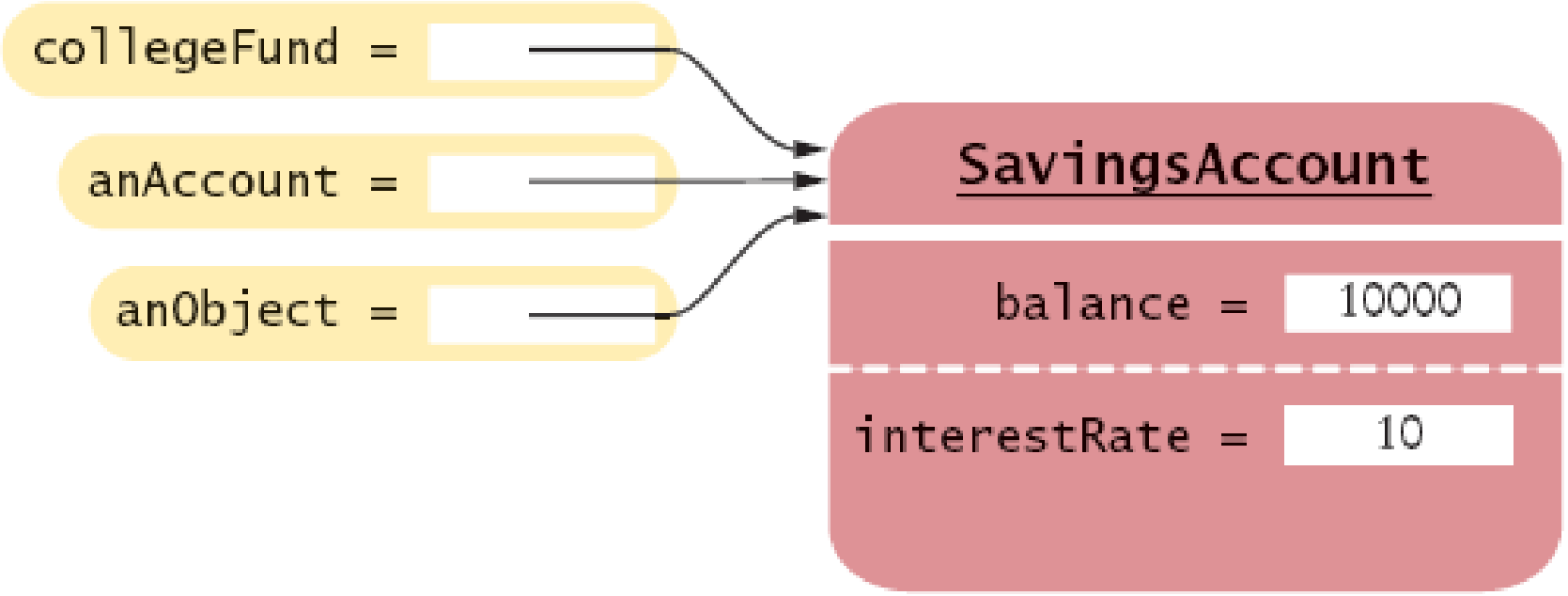


Figure 7 Variables of Different Types Refer to the Same Object

Conversione tra tipi sottoclasse e superclasse (continua)

- Riferimenti di tipo superclasse non conoscono la storia completa:

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

- Quando si converte un riferimento di un oggetto sottoclasse al suo tipo `superclasse`:
  - *Il valore del riferimento resta lo stesso – esso è la locazione di memoria dell'oggetto*
  - *Tuttavia, si conosce meno informazione rispetto all'oggetto*

Continua

Conversione tra tipi sottoclasse e superclasse

- Perché si dovrebbe voler conoscere *meno* su un oggetto?
  - *Riutilizzo del codice che conosce la* `superclasse` *ma non la* `sottoclasse`*:*

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

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

Conversione tra tipi sottoclasse e superclasse

- Talvolta può essere necessario convertire un riferimento di superclasse ad un riferimento sottoclasse

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

- Si tratta di una conversione pericolosa: se sbagliata, viene lanciata un'eccezione
- Soluzione: usare l'operatore `instanceof`
- `instanceof`: verifica se un oggetto appartiene ad uno specifico tipo

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

Sintassi 10.4 L'operatore instanceof

object `instanceof` *TypeName*

Esempio:

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

Scopo:

Restituire `true` se l'*oggetto* è istanza di *TypeName* (o uno dei suoi sottotipi), e `false` altrimenti.

Polimorfismo

- In Java, il tipo di una variabile non determina completamente il tipo dell'oggetto al quale essa si riferisce

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

- Le chiamate di metodo sono determinate dal tipo dell'oggetto reale, non dal tipo di riferimento dell'oggetto

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

- Il compilatore ha bisogno di verificare che siano chiamati soltanto metodi legali

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

Polimorfismo

- Polimorfismo: capacità di riferirsi ad oggetti di tipo multiplo con comportamento variabile
- Polimorfismo al lavoro:

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

- A seconda del tipo di `amount` e `other`, sono chiamate versioni differenti di `withdraw` e `deposit`

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:
```

Continua

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++;
```

Continua

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:          }
```

Continua

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:
```

Continua

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:    */
```

Continua

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:    */
```

Continua

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
```

Controlli di accesso

- Java ha quattro livelli di controlli di accesso a variabili, metodi e classi:
  - `public`
    - *L'accesso è consentito ai metodi di tutte le classi*
  - `private`
    - *L'accesso è consentito solo ai metodi della classe stessa*
  - `protected`
    - *L'accesso è consentito ai metodi della classe stessa, a quelli delle sottoclassi ed ai metodi delle classi che stanno nello stesso package*
  - *package access* (accesso di pacchetto)
    - *Il default, quando non viene indicato nessun modificatore di accesso*
    - *L'accesso è consentito da tutte le classi nello stesso package*
    - *Buon default per le classi, ma estremamente infelice per le variabili*

Livelli di accesso consigliati

- Variabili `istanza` e `statiche` : sempre private. Eccezioni:
  - *`public static final` per le costanti è utile e sicuro*
  - *Alcuni oggetti, come `System.out`, necessitano di essere accessibili a tutti i programmi (`public`)*
  - *Occasionalmente, le classi in un package devono collaborare molto strettamente (si dia ad alcune variabili accesso di pacchetto); classi interne sono di norma una scelta migliore*
- Methodi: `public` o `private`
- Classi ed interfacce: `public` o di pacchetto
  - Alternativa migliore all'accesso di pacchetto: classi interne
    - In generale, classi interne non dovrebbero essere `public` (ci sono alcune eccezioni, per esempio, `Ellipse2D.Double`)
- Attenzione ad accessi di pacchetto accidentali (dimenticanza di `public` o `private`)

Object: la superclasse universale

- Tutte le classi definite senza un esplicito `extends` automaticamente estendono `Object`

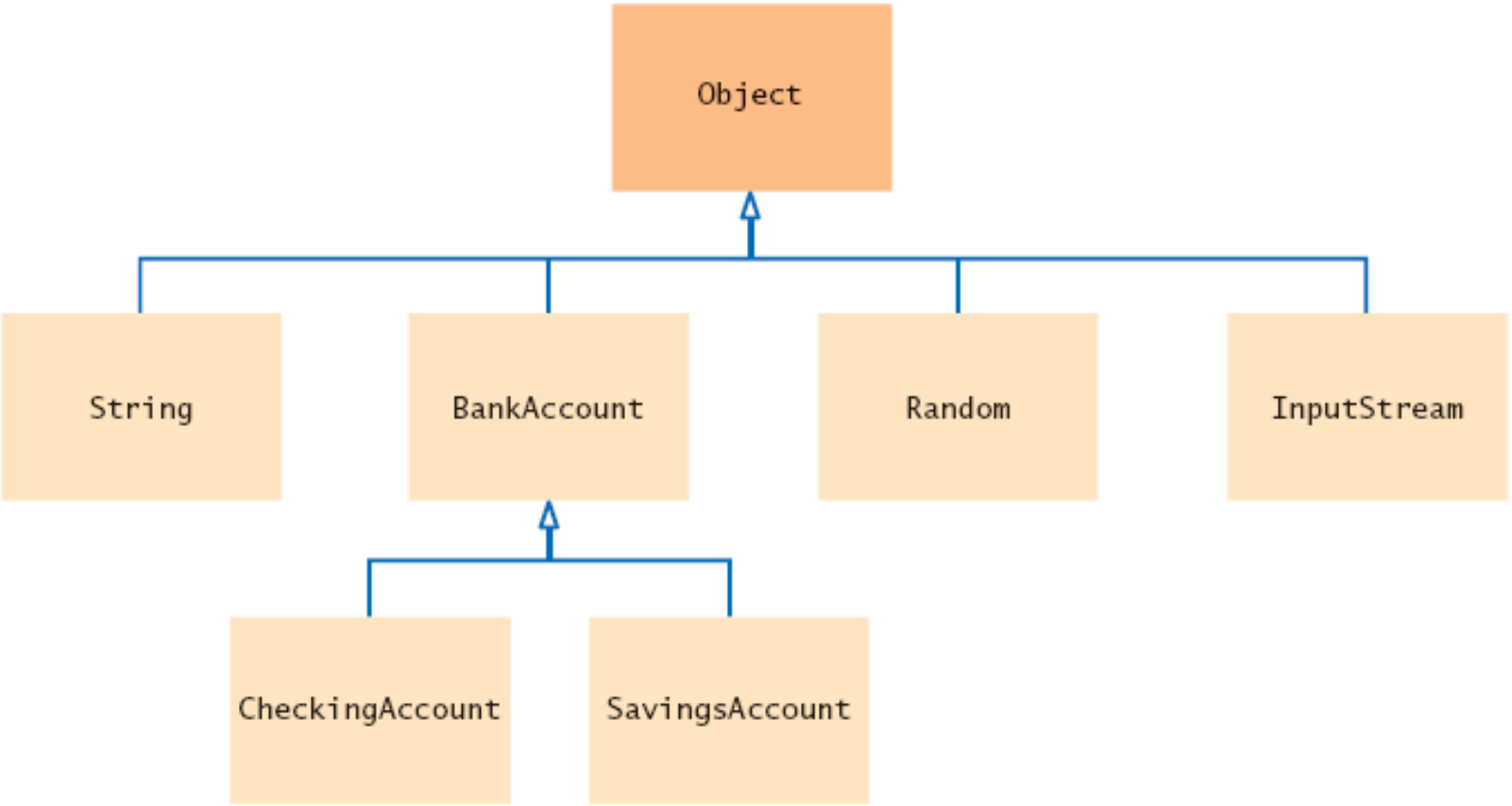


Figure 8 The Object Class Is the Superclass of Every Java Class

Object: la superclasse universale

- Tutte le classi definite senza un esplicito `extends` automaticamente extendono `Object`
- Contiene molti utili metodi:
  - *`String toString()`*
  - *`boolean equals(Object otherObject)`*
  - *`Object clone()`*
- Buona idea sovrascrivere questi metodi nelle proprie classi

Sovrascrivere il metodo `toString`

- Restituisce una rappresentazione stringa dell'oggetto
- Utile per il 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` viene chiamato ogni volta che si concatena una stringa con un `object`:

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

Continua

Sovrascrivere il metodo `toString` (continua)

- `Object.toString` produce il nome della classe e lo *hash code* dell'oggetto

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

Sovrascrivere il metodo `toString`

- Per fornire una migliore rappresentazione di un oggetto, si sovrascriva `toString`:

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

- In questo modo funziona meglio:

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

Sovrascrivere il metodo `equals`

- `Equals` verifica uguaglianza di *contenuto*

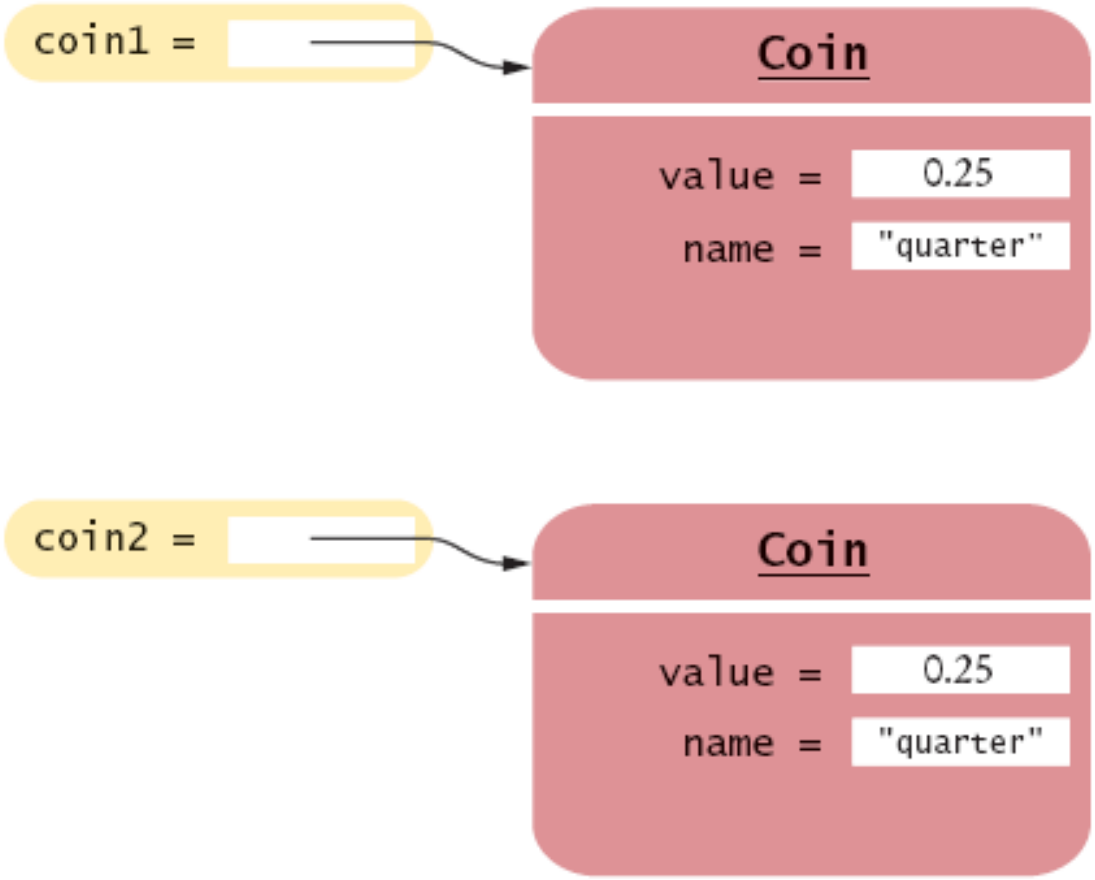


Figure 9 Two References to Equal Objects

Continua

Sovrascrivere il metodo `equals` (continua)

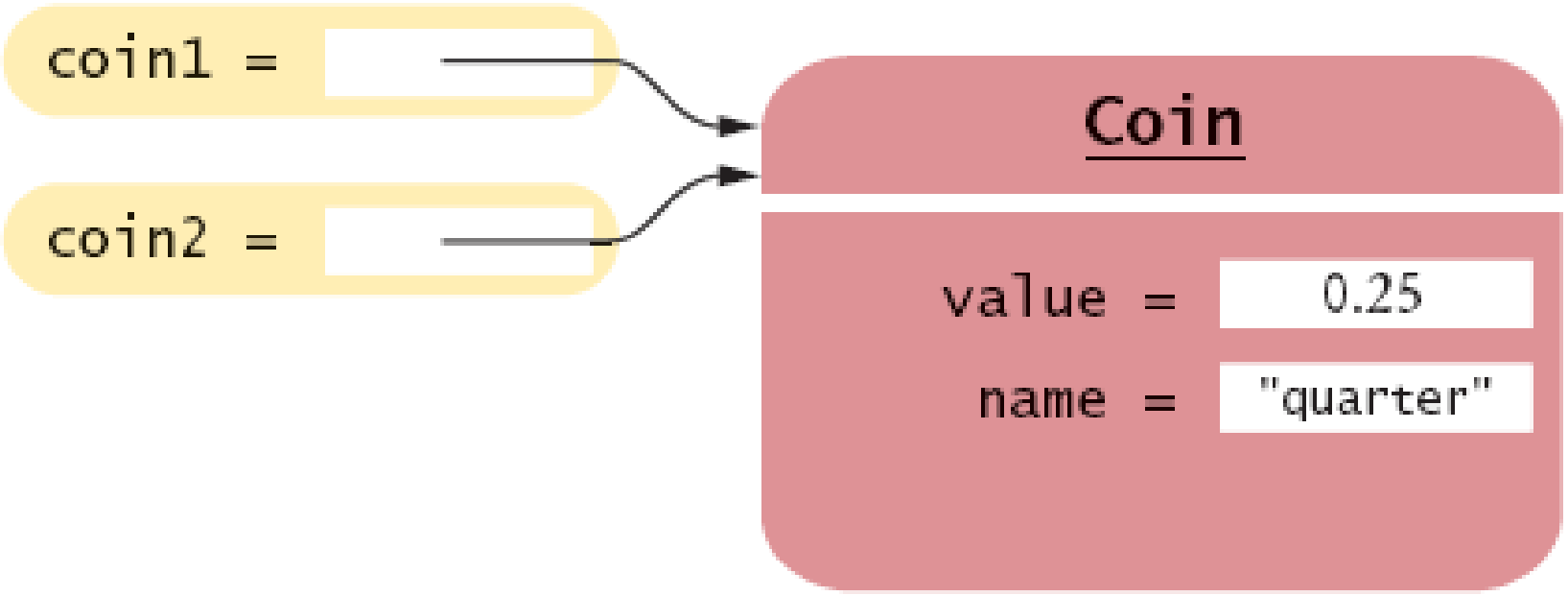


Figure 10 Two References to the Same Object

Sovrascrivere il metodo `equals`

- Si definisca il metodo `equals` per verificare se due oggetti hanno uguale stato
- Nel ridefinire il metodo `equals`, non si può cambiare la firma dell'oggetto; si usi invece un *cast* (conversione):

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

Continued

Sovrascrivere il metodo `equals` (continua)

- Si dovrebbe anche sovrascrivere il metodo `hashCode` in modo che oggetti `uguali` abbiano lo stesso hash code

Self Check 10.16

La chiamata `x.equals(x)` restituisce sempre `true`?

Risposta: Certamente – a meno che, naturalmente, `x` sia `null`.

Sovrascrivere il metodo clone

- Copiare un riferimento ad oggetto produce due riferimento allo stesso oggetto
  BankAccount account2 = account;
- Talvolo vi è bisogno di creare una copia dell'oggetto

Continua

Sovrascrivere il metodo clone (continua)

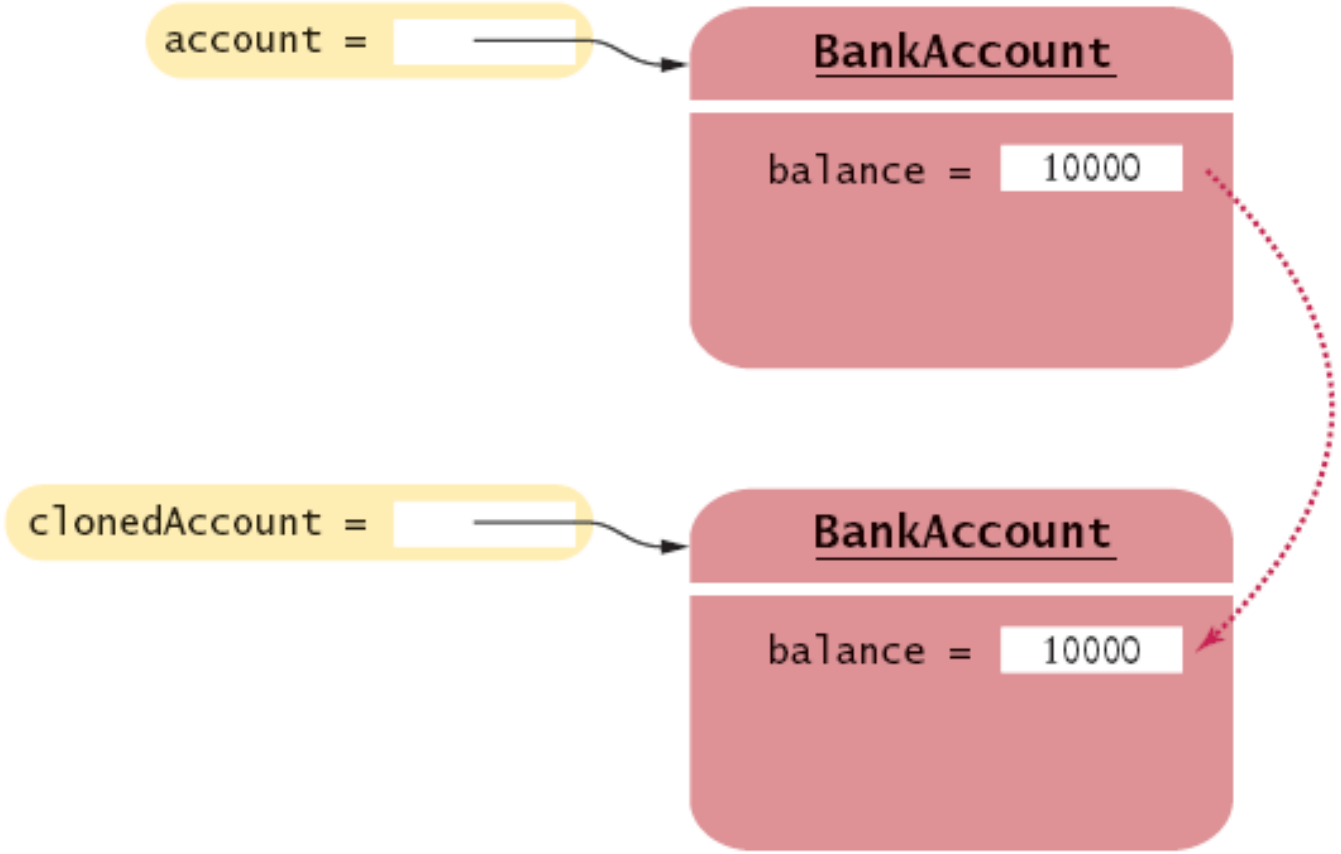


Figure 11
Cloning Objects

Continua

Sovrascrivere il metodo clone (continua)

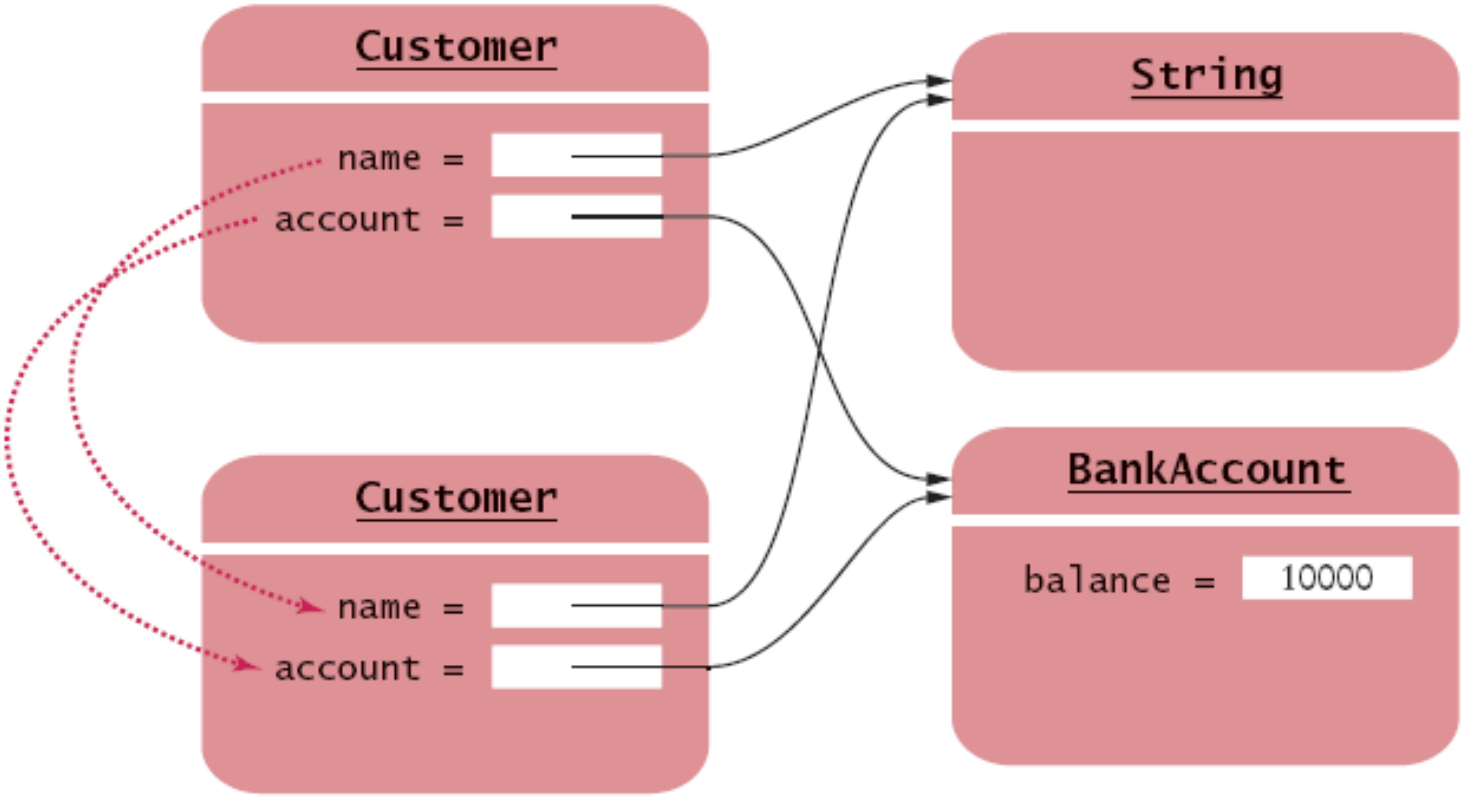
- Si definisca il metodo `clone` per creare un nuovo oggetto (si vedano gli Argomenti Avanzati 10.6)
- Si usi `clone`:

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

- Si deve effettuare un cast sul valore restituito perché questo è di tipo `Object`

Il metodo `Object.clone`

- Crea copie *superficiali*



The `Object.clone` Method Makes a Shallow Copy

Continua

Il metodo `Object.clone` (continua)

- Non clona sistematicamente tutti i sottooggetti
- Va usato con cautela
- E' dichiarato `protected`; impedisce chiamate accidentali del tipo `x.clone()` se la classe alla quale `x` appartiene non ha ridefinito `clone` come `public`
- Si dovrebbe sovrascrivere il metodo `clone` con attenzione (si vedano gli Argomenti Avanzati 10.6)

Aree di testo

- Si usi una `JTextArea` per mostrare più linee di testo
- Si può specificare il numero di righe e di colonne:

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

- `setText`: per stabilire il testo di una casella di testo o di un'area di testo
- `append`: per aggiungere testo alla fine di un'area di testo
- Si usino caratteri `newline` per separare linee:

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

- Se si vuole usare soltanto per la visualizzazione:

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

Aree di testo

- Per aggiungere barre di scorrimento ad un'area di testo:

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

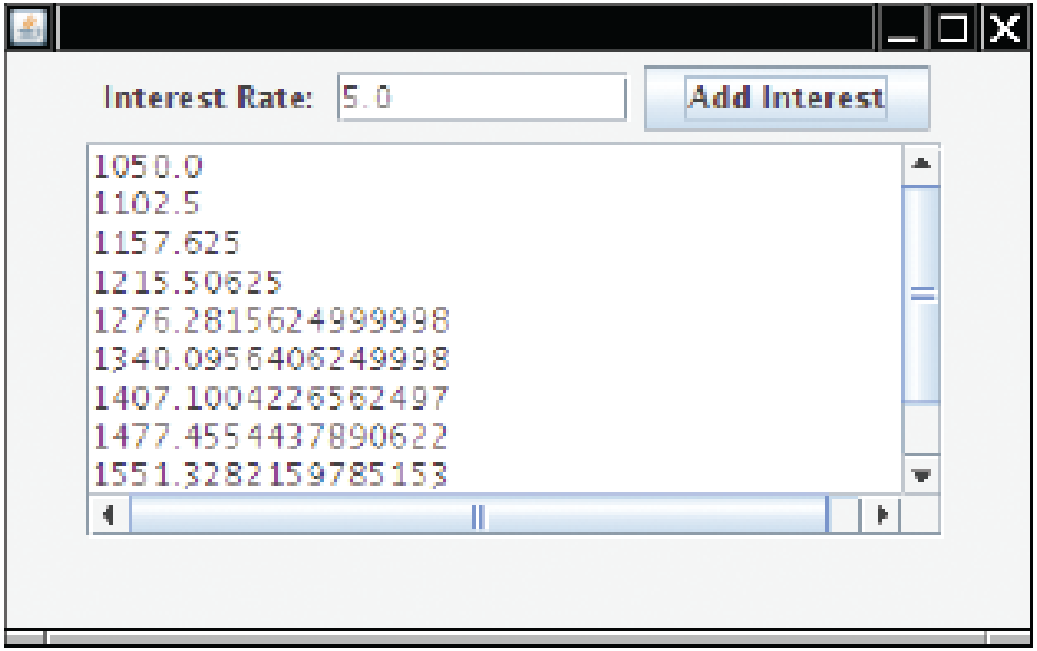


Figure 13 The Investment Application with a Text Area

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
interest.
13: */
14: public class InvestmentFrame extends JFrame
15: {
16:    public InvestmentFrame()
17:    {
18:       account = new BankAccount(INITIAL_BALANCE);
19:       resultArea = new JTextArea(AREA_ROWS, AREA_COLUMNS);
20:       resultArea.setEditable(false);
21:
```

Continua

ch10/textarea/InvestmentFrame.java (continua)

```
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:
```

Continua

ch10/textarea/InvestmentFrame.java (continua)

```
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);
```

Continua

ch10/textarea/InvestmentFrame.java (continua)

```
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: }
```