

# **Capitolo 12**

## **Ereditarieta'**

# Obiettivi

---

- **Comprendere l'ereditarietà**
- **Comprendere come ereditare e sovrascrivere i metodi delle superclassi**
- **Imparare ad invocare i costruttori delle superclassi**
- **Imparare il significato dell'accesso `protected` e `package`**
- **Comprendere il ruolo della superclasse comune `Object` ed imparare a sovrascrivere i metodi `toString` e `equals`**

# Introduzione all'ereditarieta'

---

- **Ereditarieta'**: estendere le funzionalita' di una classe aggiungendo metodi e campi
- **Esempio: Savings account = bank account con un campo interest**

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

*Continua...*

# Introduzione all'ereditarieta'

---

- **SavingsAccount** automaticamente eredita tutti i metodi ed i campi di istanza di **BankAccount**

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

- Classe Estesa = *superclass* (**BankAccount**),  
Classe che estende = *subclass* (**Savings**)

*Continua...*

# Introduzione all'ereditarietà

---

- **Ereditare da una classe  $\neq$  implementare interface: la sottoclasse eredita comportamento e stato**
- **Un vantaggio dell'ereditarietà è il riuso del codice**

# Diagramma di ereditarieta'

- Ogni classe estende la classe Object direttamente o indirettamente

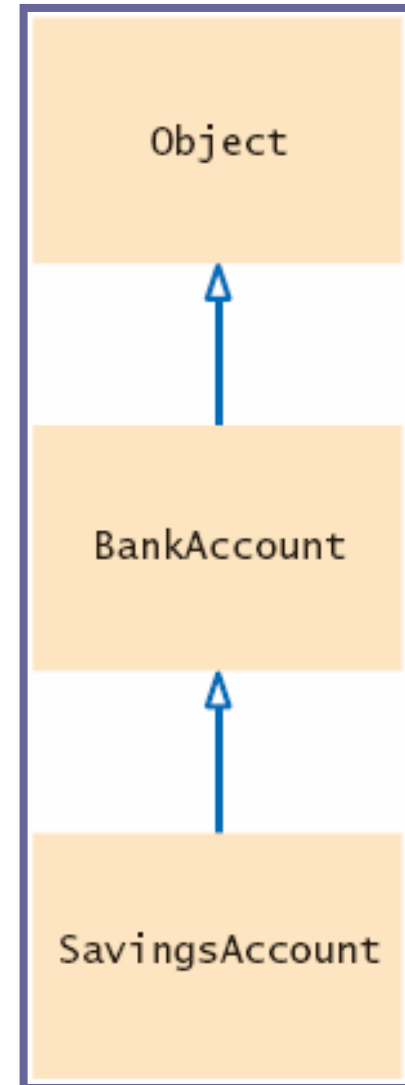


Figura 1:  
Un diagramma di ereditarieta'

# Introduzione all'ereditarietà

- In una sottoclasse, si specificano solo i campi istanza ed i metodi aggiunti, oppure 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);
    }

    private double interestRate;
}
```

# Introduzione all'ereditarietà

---

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

# Layout di un oggetto della sottoclasse

- **SavingsAccount** eredita il campo di istanza **balance** da **BankAccount**, e guadagna un campo di istanza aggiuntivo: **interestRate**

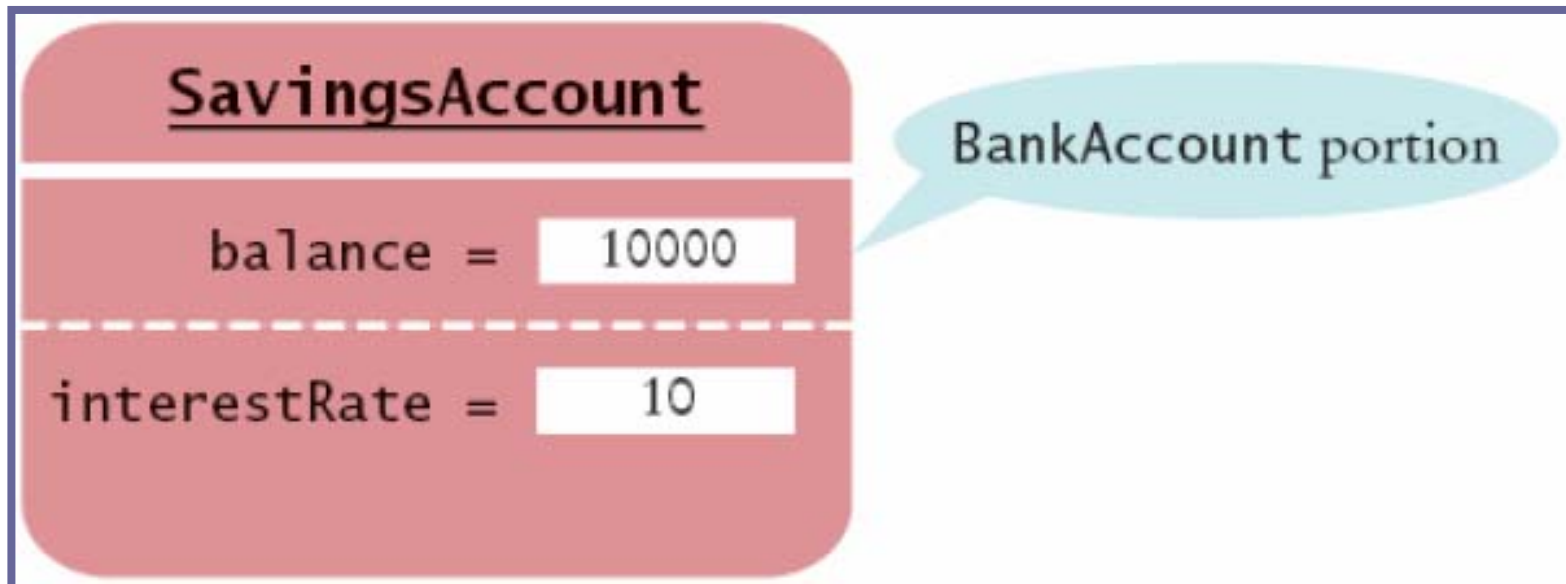


Figura 2:  
Layout di un oggetto della sottoclasse

# Sintassi 12.1: Ereditarietà

---

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

**Continua...**

# Sintassi 12.1: Ereditarieta'

## Esempio:

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

    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 nuovi metodi e campi di istanza aggiunti nella nuova classe

# Self Check

---

1. Quali campi di istanza possiede un oggetto della classe **SavingsAccount**?
2. Citare quattro metodi che si possono applicare agli oggetti di classe **SavingsAccount**
3. Se la classe **Manager** estende la classe **Employee**, quale classe e' la superclasse e quale e' la sottoclasse?

# Risposte

---

1. Due: **balance e interestRate.**
2. **deposit, withdraw, getBalance, e addInterest.**
3. **Manager e' la sottoclasse; Employee e' la superclasse.**

# Gerarchie di ereditarieta'

- Insiemi di classi possono formare complesse gerarchie di ereditarieta'
- Example:

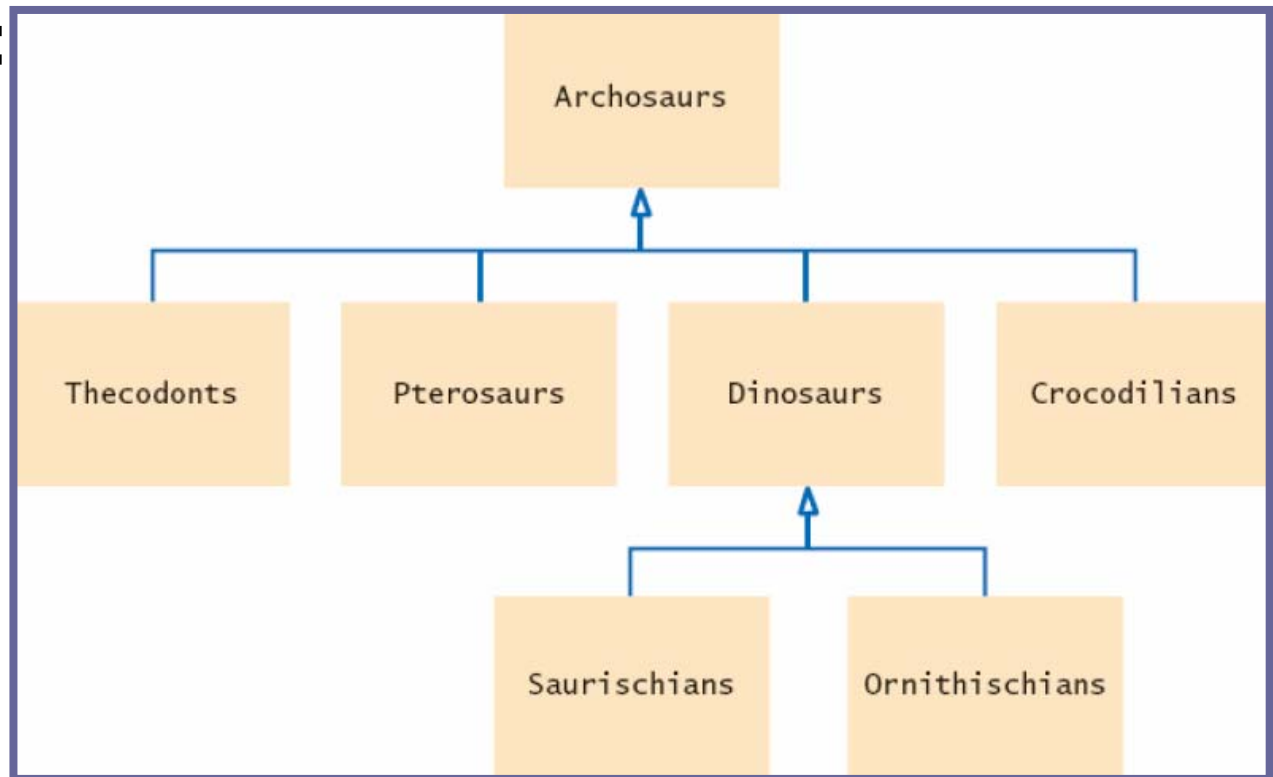


Figura 3:  
Una parte della gerarchia dei rettili preistorici

# Una gerarchia semplice: la gerarchia dei Bank Accounts

---

- **Considerate una banca che offre ai suoi clienti I seguenti tipi di conto:**
  1. Checking account: niente interessi; piccolo numero di transazioni gratuite per mese, transazioni aggiuntive con tariffa economica
  2. Savings account: genera un interesse che si somma mensilmente

*Continua...*

# Una gerarchia semplice: la gerarchia dei Bank Accounts

- Gerarchia di ereditarieta'

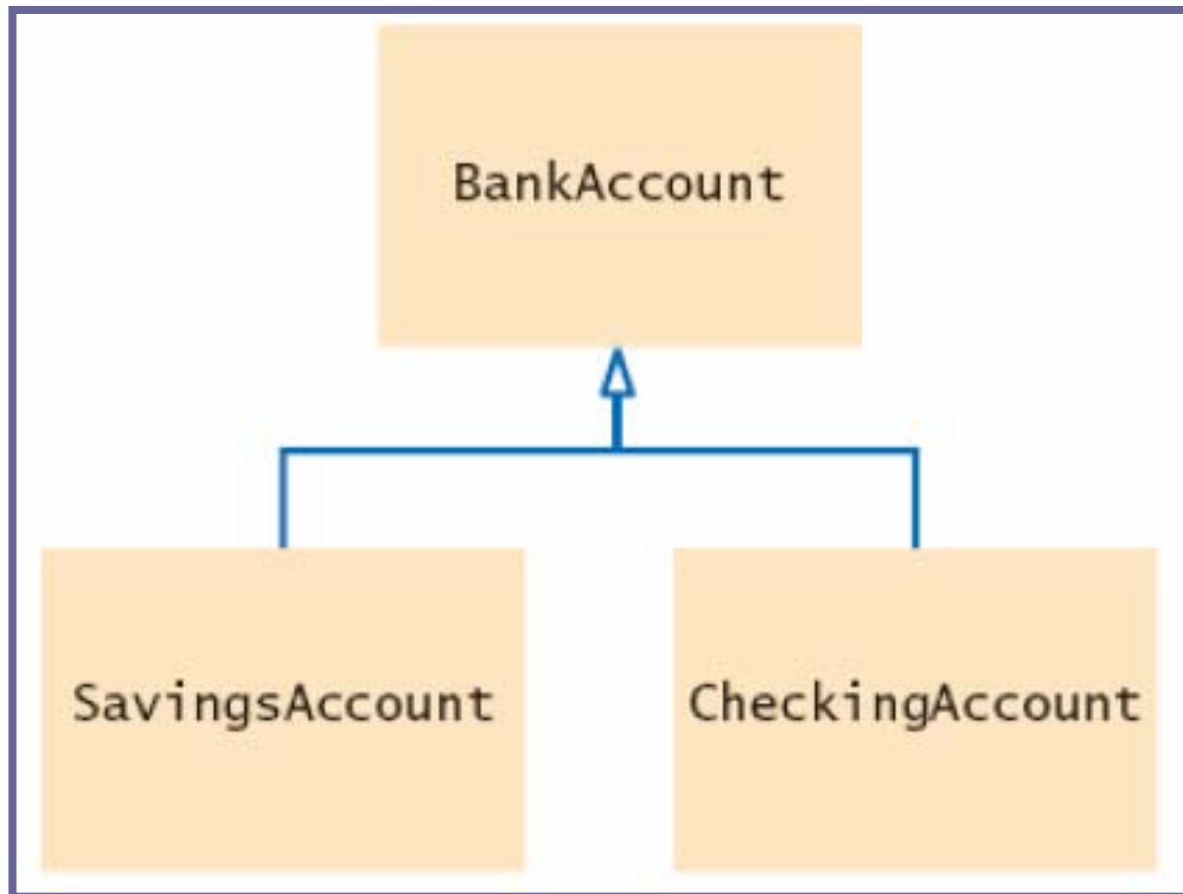


Figura 5:  
Gerarchia di ereditarieta' per le classi Bank Account

*Continua...*

# Una gerarchia semplice: la gerarchia dei Bank Accounts

---

- Tutti i bank accounts supportano il metodo `getBalance`
- Tutti i bank accounts supportano i metodi `deposit` e `withdraw`, ma l'implementazione e' diversa
- Il Checking account necessita di un metodo `deductFees`; il savings account ha bisogno di un metodo `addInterest`

# Self Check

---

4. Quale campo di istanza bisogna aggiungere alla classe `CheckingAccount`?

# Risposte

---

- 4. E' necessario un contatore per contare il numero di prelievi e depositi.**

# Ereditare i metodi

---

- **Override :**
  - Fornire una differente implementazione di un metodo che esiste nella superclasse
  - Devono avere la stessa firma (stesso nome e stesso tipo di parametri)
  - Se il metodo e' applicato ad un oggetto della sottoclasse, viene eseguita la versione modificata
- **Ereditarieta':**
  - Non fornire una nuov implementazione di unmetodo che esiste nella superclasse
  - Il metodo della superclasse puo' essere applicato agli oggetti della sottoclasse

*Continua...*

# Ereditare i Metodi

---

- **Aggiungere un metodo:**
  - Fornire un nuovo metodo che non esiste nella superclasse
  - Il nuovo metodo puo' essere applicato solo agli oggetti della sottoclasse

# Ereditare i campi di istanza

---

- **Non si puo' fare l'override dei campi**
- **Ereditare i campi: tutti i campi della superclasse sono ereditati automaticamente**
- **Aggiungere un campo: fornire un nuovo campo che non esiste nella superclasse**

*Continua...*

# Ereditare i campi di istanza

---

- **Cosa succede se si definisce un nuovo campo con lo stesso nome di quello presente nella superclasse?**
  - Ogni oggetto possiede due campi di istanza con lo stesso nome
  - I campi possono contenere valori diversi
  - Lecito ma molto pericoloso

# Implementare la classe CheckingAccount

- Override di **deposit** e **withdraw** per incrementare il conteggio delle 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
}
```

*Continua...*

# Implementare la classe

## CheckingAccount

---

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

*Continua...*

# Implementare la classe

## CheckingAccount

---

- Si possono applicare quattro metodi agli oggetti di classe **CheckingAccount** :
  - **getBalance()** (ereditato da **BankAccount**)
  - **deposit(double amount)** (override del metodo di **BankAccount**)
  - **withdraw(double amount)** (override del metodo di **BankAccount**)
  - **deductFees()** (nuovo di **CheckingAccount**)

# I campi ereditati sono Private

---

- Considerate il metodo **deposit** di **CheckingAccount**

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

- Non puo' semplicemente sommare **amount** a **balance**
- **balance** e' un campo *private* della superclasse

*Continua...*

# I campi ereditati sono Private

---

- **Una sottoclasse non ha accesso ai campi private della sua superclasse**
- **Le sottoclassi devono usare l'interfaccia pubblica**

# Invocare un metodo della SuperClasse

---

- Non si puo' semplicemente chiamare `deposit ( amount )` nel metodo `deposit` di `CheckingAccount`
- Sarebbe lo stesso di:  
`this.deposit ( amount )`
- Chiamare lo stesso metodo genera una ricorsione infinita
- Invece, invocare il metodo della *superclass*  
`super.deposit ( amount )`

*Continua...*

# Invocare un metodo della SuperClasse

---

- Chiamare il metodo **deposit** della classe **BankAccount**
- Metodo Completo:

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

# Sintassi 12.2: 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 del metodo della classe corrente

# Implementare gli altri Metodi

---

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

***Continua...***

# Implementare gli altri Metodi

```
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

---

6. Perché il metodo `withdraw` della classe `CheckingAccount` chiama `super.withdraw`?
7. Perché il metodo `deductFees` setta a zero il conteggio delle transazioni?

# Risposte

---

6. Ha bisogno di decrementare il balance, e non puo' accedere al campo **balance** direttamente.
7. Perche' il conteggio possa contenere il numero delle transazioni per il mese successivo.

# Errore comune: nascondere i campi di istanza

- Una sottoclasse non ha accesso ai campi di istanza private della superclasse
- Errore : “risolvere” il problema aggiungendo un altro campo 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: nascondere i campi di istanza

- Così il metodo deposit compila, ma non aggiorna il balance corretto!

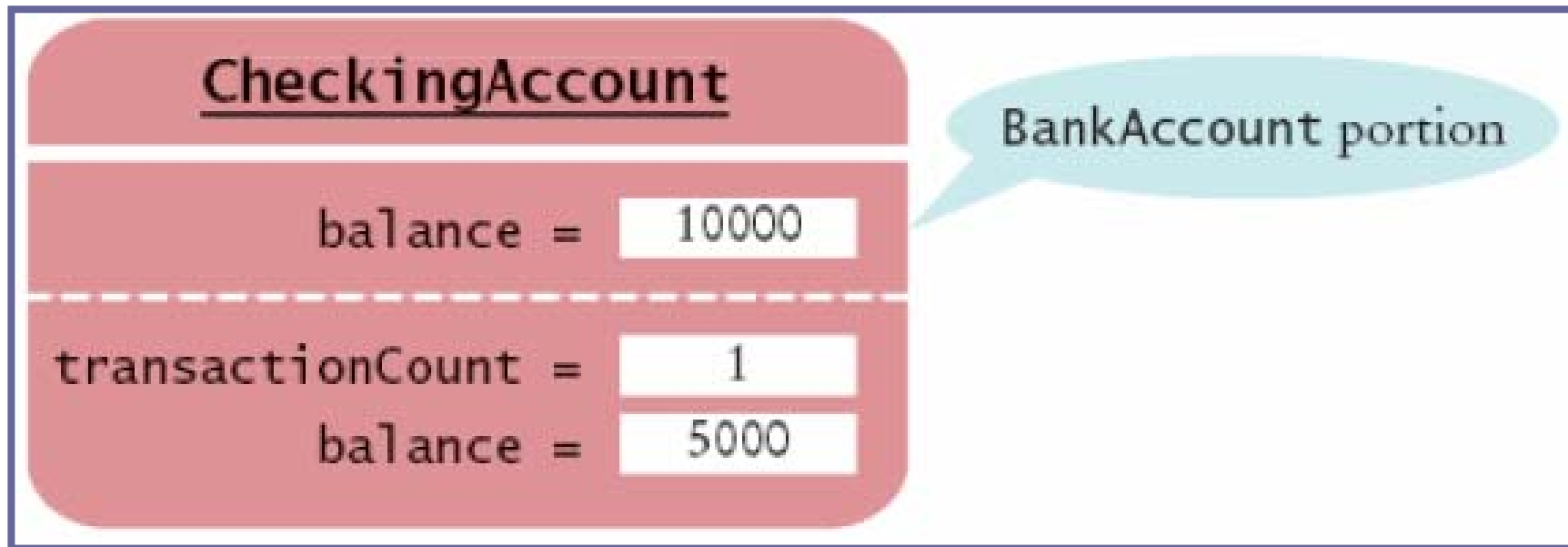


Figura 6:  
Nascondere i campi di istanza

# Costruzione degli oggetti della sottoclasse

- **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;
    }
    . . .
}
```

*Continua...*

# Costruzione degli oggetti della sottoclasse

---

- **Deve essere la prima istruzione nel costruttore della sottoclasse**
- **Se il costruttore della sottoclasse non chiama il costruttore della superclasse esplicitamente, viene comunque chiamato il costruttore di default della superclasse**
  - Il costruttore di Default : quello senza parametri
  - Se tutti i costruttori della superclasse richiedono parametri, il compilatore segnala un errore

# Sintassi 12.3: Chiamare il costruttore della Superclasse

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

## Esempio:

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

## Scopo:

Invocare un costruttore della superclasse. Questa istruzione deve essere la prima del costruttore della sottoclasse.

# Self Check

---

8. Perché il costruttore di **SavingsAccount** nella sezione 12.1 non chiama il costruttore della sua superclasse?
9. Quando si invoca un metodo della superclasse con la parola chiave **super**, la chiamata deve essere la prima istruzione del metodo della sottoclasse?

# Risposte

---

8. Bastava il costruttore di default della superclasse, che setta il balance a zero.
9. No – questo e' un requisito solo per i costruttori. Per esempio, il metodo `SavingsAccount.deposit` prima incrementa il conteggio delle transazioni, poi chiama il metodo della superclasse.

# Convertire fra i tipi Sottoclasse e Superclasse

---

- E' Ok convertire un riferimento di tipo sottoclasse ad uno di tipo superclasse

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

# Convertire fra i tipi Sottoclasse e Superclasse

- I tre riferimenti ad oggetti **collegeFund**, **anAccount**, e **anObject** si riferiscono tutti allo stesso oggetto di tipo **SavingsAccount**

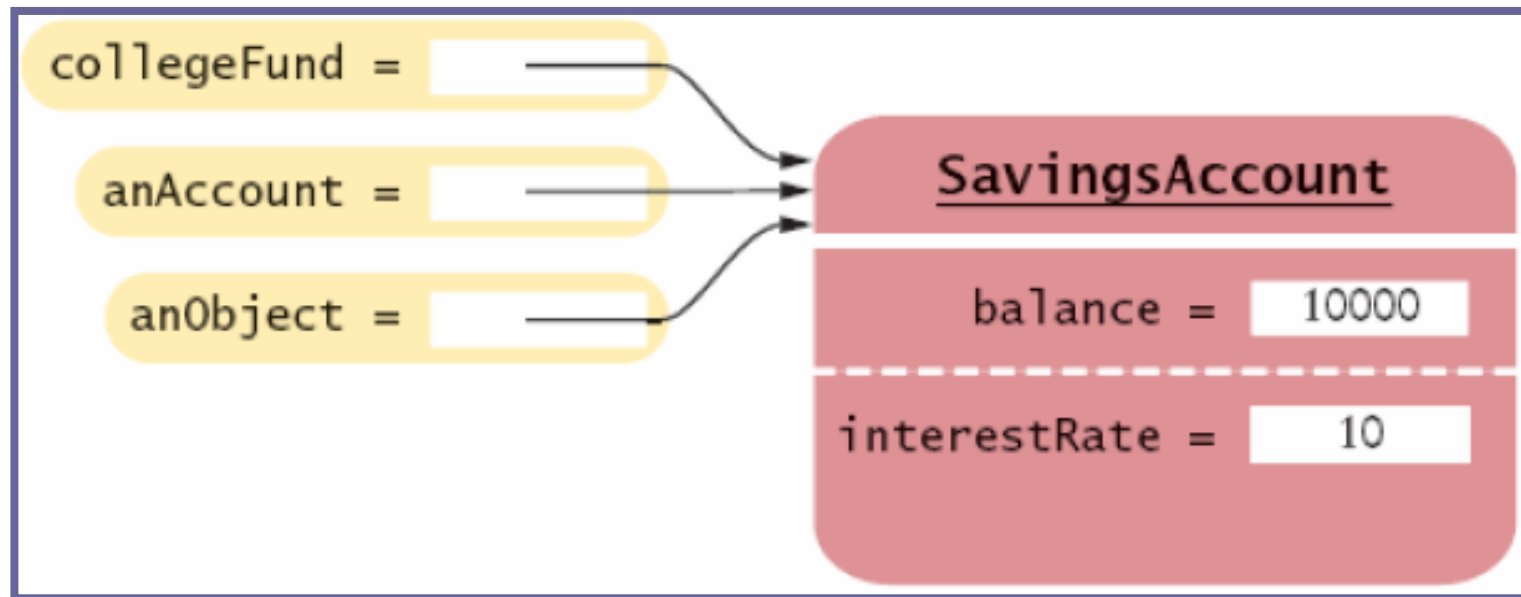


Figura 7:  
Variabili di Tipi Differenti si  
riferiscono allo stesso oggetto

# Convertire fra i tipi Sottoclasse e Superclasse

- I riferimenti di tipo Superclasse non conoscono l'intera storia:

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

- Quando si converte un oggetto della sottoclasse al suo tipo superclasse:
  - Il valore del riferimento rimane invariato – e' la locazione in memoria dell'oggetto
  - Pero': si sanno meno informazioni sull'oggetto in questione

*Continua...*

# Convertire fra i tipi Sottoclasse e Superclasse

- **Perche' si dovrebbe voler sapere di meno riguardo ad un oggetto?**
  - Riutilizzare codice adatto alla superclasse ma non alla sottoclasse

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

- **Puo' essere usato per trasferire denaro da ogni tipo di **BankAccount****

# Convertire fra i tipi Sottoclasse e Superclasse

---

- **Puo' essere necessario convertire un riferimento alla superclasse in un riferimento alla sottoclasse**

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

- **Questo cast e' pericoloso: se e' sbagliato, viene generata un'eccezione**

*Continua...*

# Convertire fra i tipi Sottoclasse e Superclasse

- Soluzione: usare l'operatore **instanceof**
- **instanceof**: verifica se un oggetto appartiene ad un tipo particolare

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

# Sintassi 12.4: l'operatore **InstanceOf**

```
object instanceof TypeName
```

## **Esempio:**

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

## **Scopo:**

Restituisce `true` se *object* è un'istanza di *TypeName* (o di uno dei suoi sottotipi), e `false` altrimenti

# Self Test

---

10. Perché il secondo parametro del metodo `transfer` deve essere di tipo `BankAccount` e non, per esempio, `SavingsAccount`?
11. Perché non si può usare come secondo parametro del metodo `transfer` un semplice `Object`?

# Risposte

---

10. Vogliamo usare il metodo con ogni tipo di bank accounts. Usando un parametro di tipo **SavingsAccount**, non avremo potuto chiamare il metodo con un oggetto **CheckingAccount**.
11. Non possiamo invocare il metodo **deposit** su una variabile di tipo **Object**.

# Polimorfismo

---

- In Java, il tipo di una variabile non determina completamente il tipo dell'oggetto a cui si riferisce

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

- Le chiamate dei metodi vengono determinate dal tipo dell'oggetto attuale, non dal tipo del riferimento

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

*Continua...*

# Polimorfismo

---

- **Il Compilatore deve evrificare che solo metodi leciti siano invocati**

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

# Polimorfismo

---

- **Polimorfismo: capacita' di riferirsi ad oggetti di tipi diversi con diversi comportamenti**
- **Polimorfismo all'opera:**

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

- **In dipendenza dai tipi di amount e other, differenti versioni di **withdraw** e **deposit** vengono chiamate**

# 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:
```

**Continua...**

# 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:     */
```

**Continua...**

# 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: */
```

**Continua...**

# 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: */
```

**Continua...**

# 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:
```

**Continua...**

# 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: */
```

**Continua...**

# 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:     */
```

**Continua...**

# File SavingsAccount.java

---

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

**Continua...**

# File SavingsAccount.java

---

## Output:

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

# Self Check

---

12. Se una variabile di tipo `BankAccount` contiene un riferimento valido, cosa si sa dell'oggetto a cui si riferisce?
13. Se `a` si riferisce ad un checking account, qual'è l'effetto di chiamare `a.transfer(1000, a)`?

# Risposte

---

12. L'oggetto e' un'istanza di **BankAccount** o di una delle sue sottoclassi.
13. Il balance di a e' invariato, ed il conteggio delle transazioni e' incrementato due volte.

# Controllo di accesso

---

- **Java ha quattro livelli di controllo di accesso a campi metodi e classi:**
  - **public**
    - Può essere acceduto da tutti i metodi di tutte le classi
  - **private**
    - Può essere acceduto solo dai metodi della sua classe
  - **protected**
    - See Advanced Topic 13.3

**Continua...**

# Controllo di accesso

---

- **Java ha quattro livelli di controllo di accesso a campi metodi e classi:**
  - package
    - il default, quando non viene specificato niente
    - Può essere acceduto da tutte le classi dello stesso package
    - Buono per le classi, pessimo per i campi di istanza

# Livelli di accesso raccomandati

---

- **Campi di istanza e statici: sempre `private`.**

## **Eccezioni:**

- Le costanti `public static final`
- Alcuni oggetti, come `System.out`, devono essere accessibili a tutti i programmi (`public`)
- Occasionalmente, le classi in un package devono cooperare molto strettamente (alcuni campi possono avere accesso package); le classi interne sono di solito una soluzione migliore

***Continua...***

# Livelli di accesso raccomandati

---

- Metodi: `public` o `private`
- Classi e interfaces: `public` o `package`
- Attenzione ad accessi `package` accidentali (per dimenticanza di `public` o `private`)

# Self Check

---

- 14. Qual'e' una ragione comune per definire campi di istanza con visibilita' di tipo package?**
- 15. Se una classe con un costruttore public ha accesso package, chi puo' costruirne gli oggetti?**

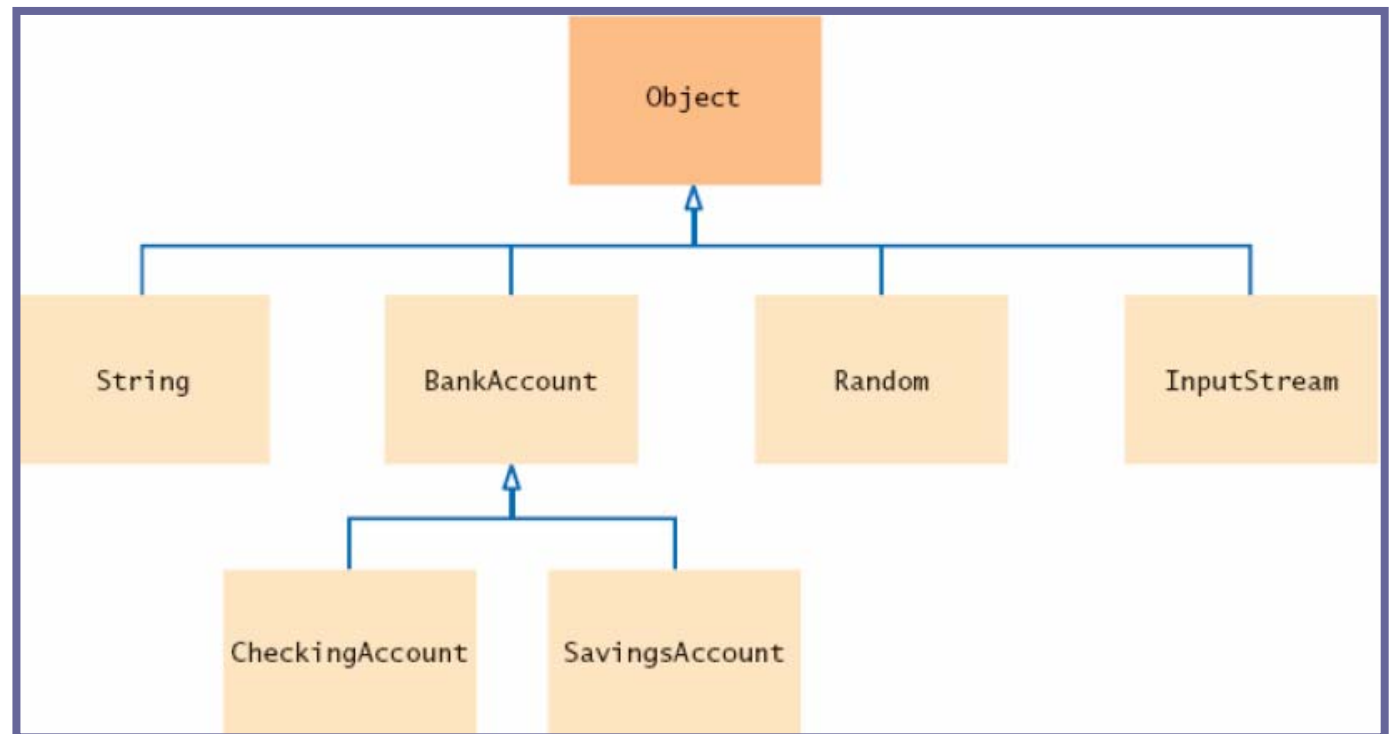
# Risposte

---

14. Dimenticare **private**.
15. Qualunque metodo delle classi nello stesso package.

# Object: la Superclasse cosmica

- Tutte le classi definite senza un'esplicita clausola **extends** estendono automaticamente **Object**



**Figura 8:**  
La classe **Object** e' la Superclasse di ogni classe Java

# Object: la Superclasse Cosmica

---

- **Metodi piu' utili:**
  - `String toString()`
  - `boolean equals(Object otherObject)`
  - `Object clone()`
- **Buona idea fare l'override di questi metodi nelle proprie classi**

# Override di `toString`

- Restituisce una rappresentazione formato 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]"
```

***Continua...***

# Override di `toString`

- `toString` e' chiamato ogni volta che una stringa viene concatenata con un oggetto:

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

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

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

# Override di `toString`

- Per fornire una rappresentazione migliore di un oggetto, fare l'override di `toString`:

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

- Così' va meglio:

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

# Override di `equals`

- `equals` verifica l'uguaglianza dei *contenuti*

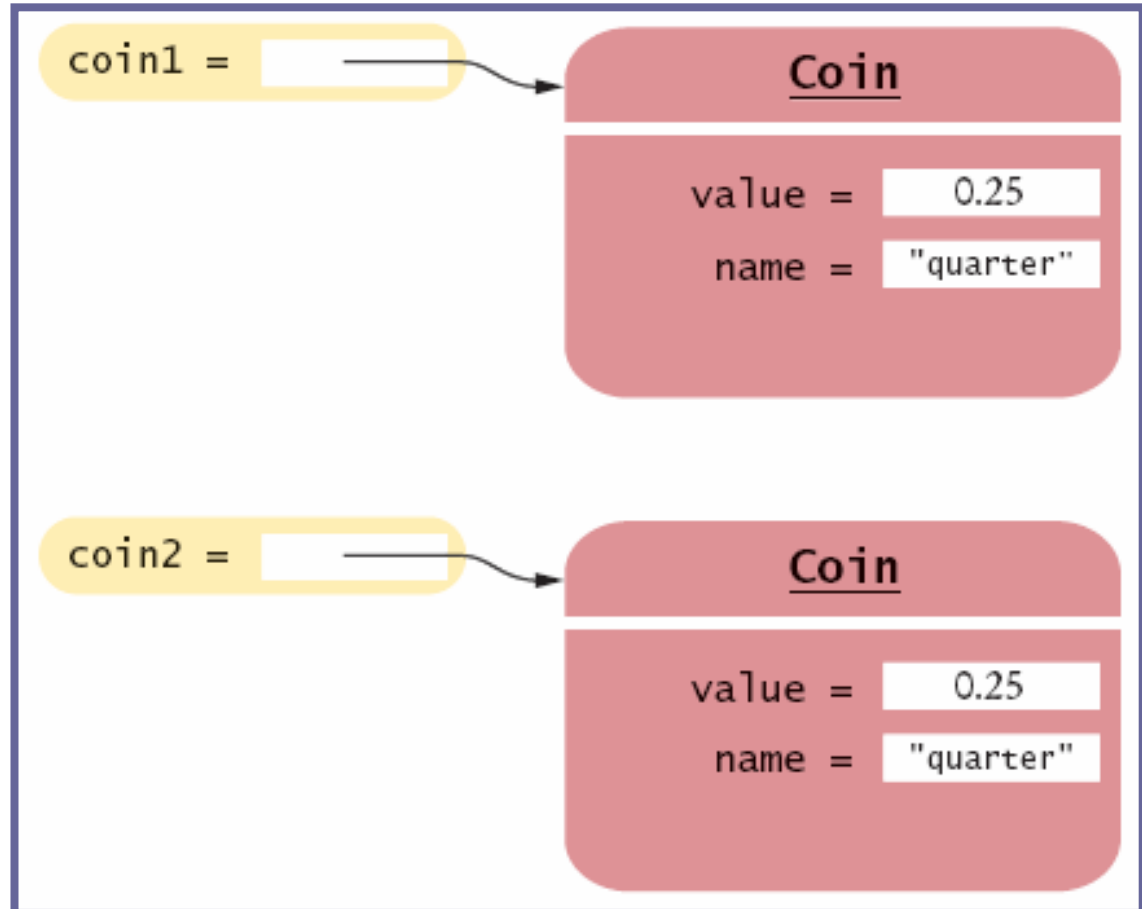


Figura 9:  
due riferimenti ad  
oggetti uguali

*Continued...*

# Override di `equals`

- `==` verifica l'uguaglianza della *locazione*

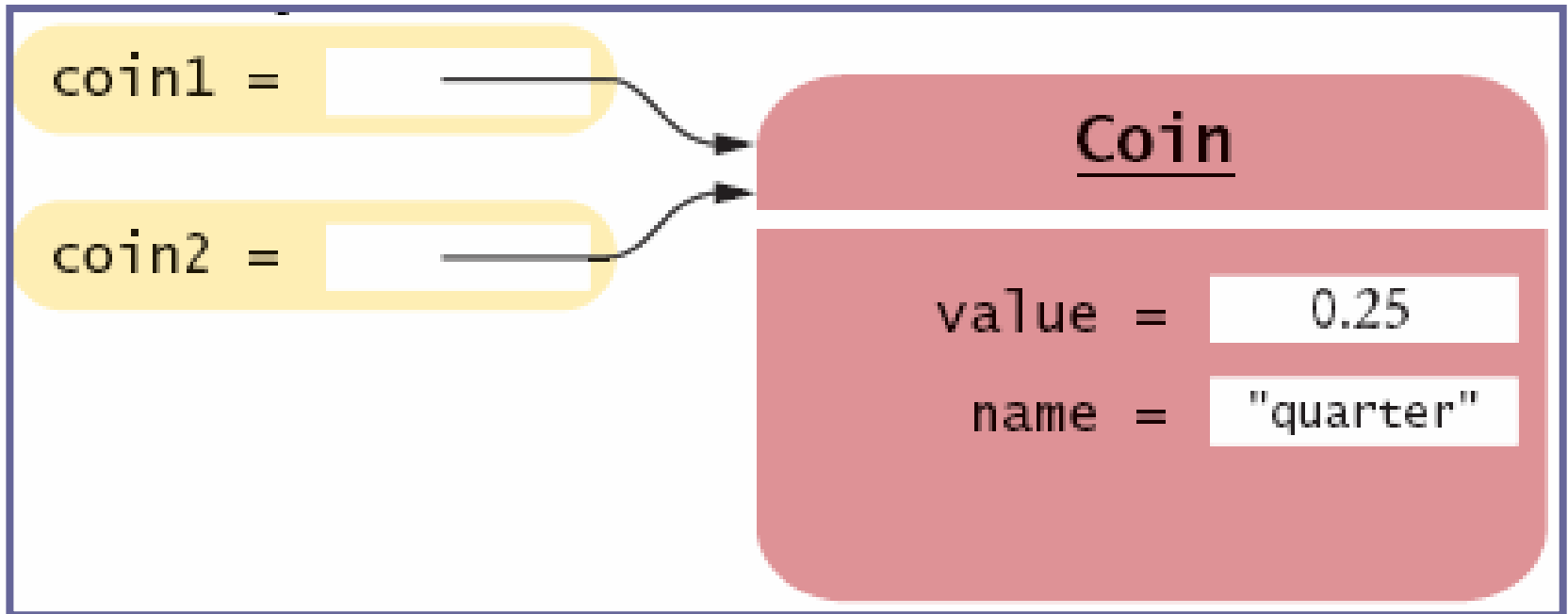


Figura 10:  
due riferimenti allo stesso oggetto

# Override di `equals`

- Definire il metodo `equals` per verificare se due oggetti hanno il medesimo stato
- Nel ridefinire `equals`, non si puo' cambiarne la firma come stabilita dalla classe `Object`; usare un *cast* :

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

# Override di `equals`

---

- Bisognerebbe anche ridefinire il metodo `hashCode` in modo che due oggetti uguali abbiano lo stesso hash code

# Self Check

---

16. La chiamata `x.equals(x)` deve sempre restituire `true`?
17. Si puo' implementare `equals` in termini di `toString`? Si dovrebbe?

# Risposte

---

16. Certamente, a meno che **x** sia **null**.
17. Se **toString** restituisce una stringa che descrive tutti i campi di istanza, si puo' semplicemente chiamare **toString** sui parametri implicito ed esplicito, e confrontare i risultati. Tuttavia, confrontare i campi e' pu' efficiente che convertirli in stringhe.

# Override di `clone`

---

- Copiare il riferimento ad un `object` produce due riferimenti allo stesso oggetto.

```
BankAccount account2 = account;
```

*Continua...*

# Override di `clone`

- A volte e' necessario fare la copia di un oggetto

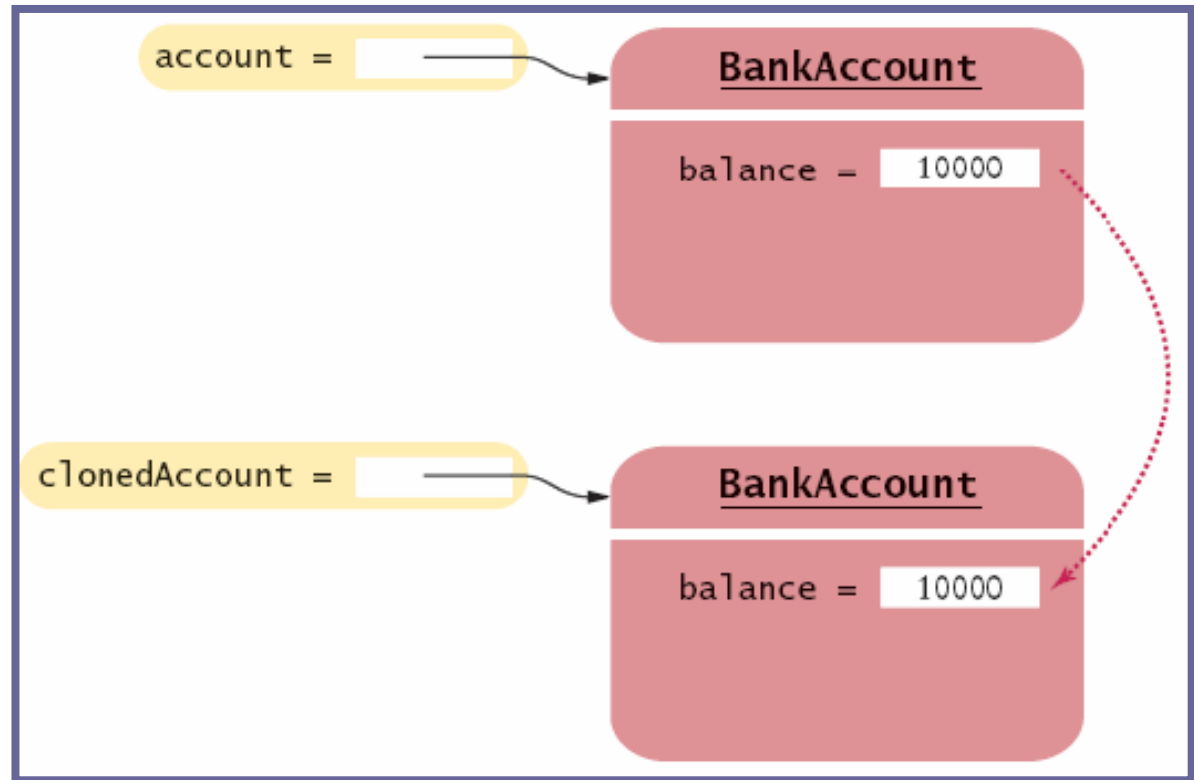


Figura 11:  
Clonare Oggetti

*Continua...*

# Override di `clone`

---

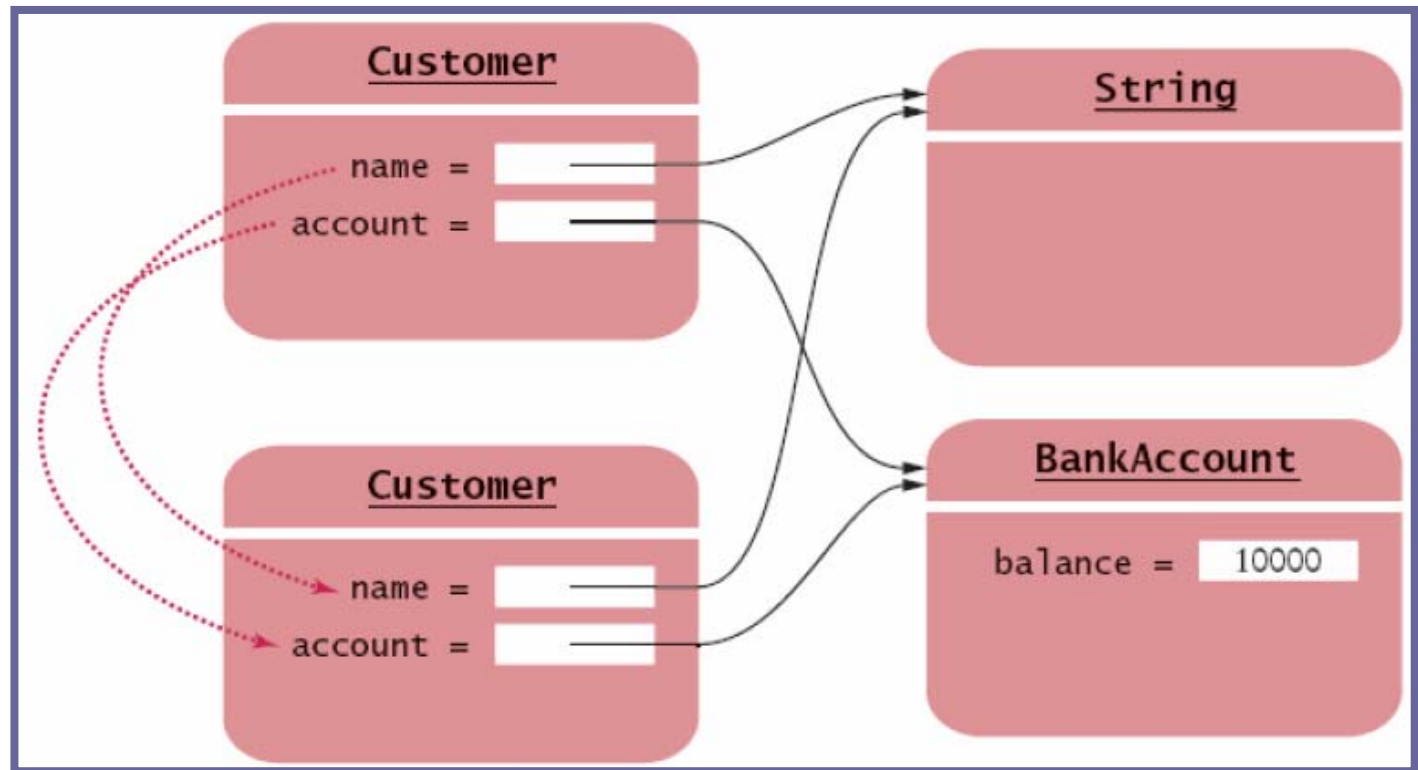
- Definire `clone` in modo da creare un nuovo oggetto (see Advanced Topic 12.6)
- Usare `clone`:

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

- Bisogna fare poi un cast sul valore di ritorno perche' il tipo ritornato e' `Object`

# Il metodo `Object.clone`

- Crea copie *semplici*



**Figura 12:**  
Il metodo `Object.clone` crea una copia semplice

# Il metodo `Object.clone`

---

- Non clona gli oggetti contenuti
- Must be used with caution
- E' dichiarato come `protected`; questo evita la chiamata accidentale di `x.clone()` se la classe a cui appartiene `x` non ha ridefinito `clone` come `public`
- Va riscritto con attenzione `clone` (see Advanced Topic 12.6)