

Capitolo 10

Ereditarietà

Cay S. Horstmann
Concetti di informatica e fondamenti di Java
quarta edizione

Obiettivi del capitolo

- Imparare l'ereditarietà
- Capire come ereditare e come sovrascrivere metodi di superclasse
- Saper invocare costruttori della superclasse
- Conoscere il controllo d'accesso protetto e di pacchetto
- Capire il concetto di superclasse comune, `Object`, e saper sovrascrivere i suoi metodi `toString`, `equals` e `clone`
- Usare l'ereditarietà per personalizzare le interfacce utente

Introduzione all'ereditarietà

- L'ereditarietà è un metodo per estendere classi esistenti aggiungendovi metodi e campi.
- Per esempio, immaginate di dover definire una classe `SavingsAccount` per descrivere un conto bancario che garantisce un tasso di interesse fisso sui depositi

```
class SavingsAccount extends BankAccount
{
    nuovi metodi
    nuovi campi di esemplare
}
```

Segue...

Introduzione all'ereditarietà

- Tutti i metodi e tutti i campi di esemplare della classe `BankAccount` vengono *ereditati automaticamente* dalla classe `SavingsAccount`

```
SavingsAccount collegeFund = new SavingsAccount(10);  
// un conto di risparmio con tasso d'interesse del 10%  
collegeFund.deposit(500);  
//é corretto usare un metodo di BankAccount  
// con l'oggetto di tipo SavingsAccount
```

- La classe più generica, che costituisce le basi dell'ereditarietà, viene chiamata *superclasse* (`BankAccount`), mentre la classe più specifica, che eredita dalla superclasse, è detta *sottoclasse* (`SavingsAccount`).

Segue...

Introduzione all'ereditarietà

- Ereditare da una classe è diverso da realizzare un'interfaccia: la sottoclasse eredita il comportamento e lo stato della propria superclasse.
- Uno dei vantaggi dell'ereditarietà è il riutilizzo del codice.

Un diagramma di ereditarietà

- Tutte le classi estendono direttamente o indirettamente la classe `Object`
- N.B. Ciascuna classe può estendere (direttamente ed esplicitamente) una sola altra classe

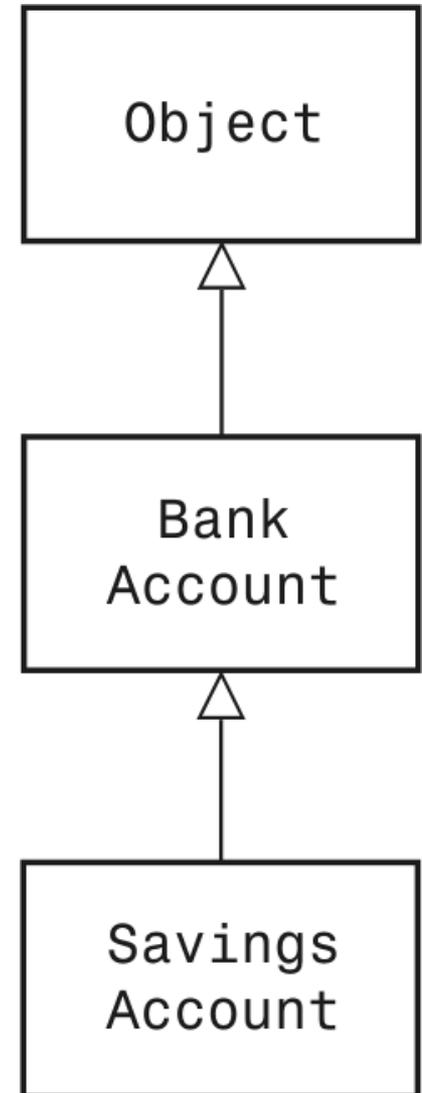


Figura 1

Introduzione all'ereditarietà

- Quando definite una sottoclasse, ne potete specificare campi di esemplare e metodi aggiuntivi, oltre a metodi modificati 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: il campo `balance` era stato definito `private` nella classe `BankAccount` e il metodo `addInterest` è definito nella classe `SavingsAccount`, per cui non ha accesso al campo privato di un'altra classe
- Notate come il metodo `addInterest` chiami i metodi `getBalance` e `deposit` della superclasse senza specificare un parametro implicito

La struttura di un esemplare di sottoclasse

- La Figura 2 mostra la struttura interna di un oggetto di tipo `SavingsAccount`, che eredita il campo di esemplare `balance` dalla superclasse `BankAccount` e acquisisce un campo di esemplare aggiuntivo, `interestRate`.

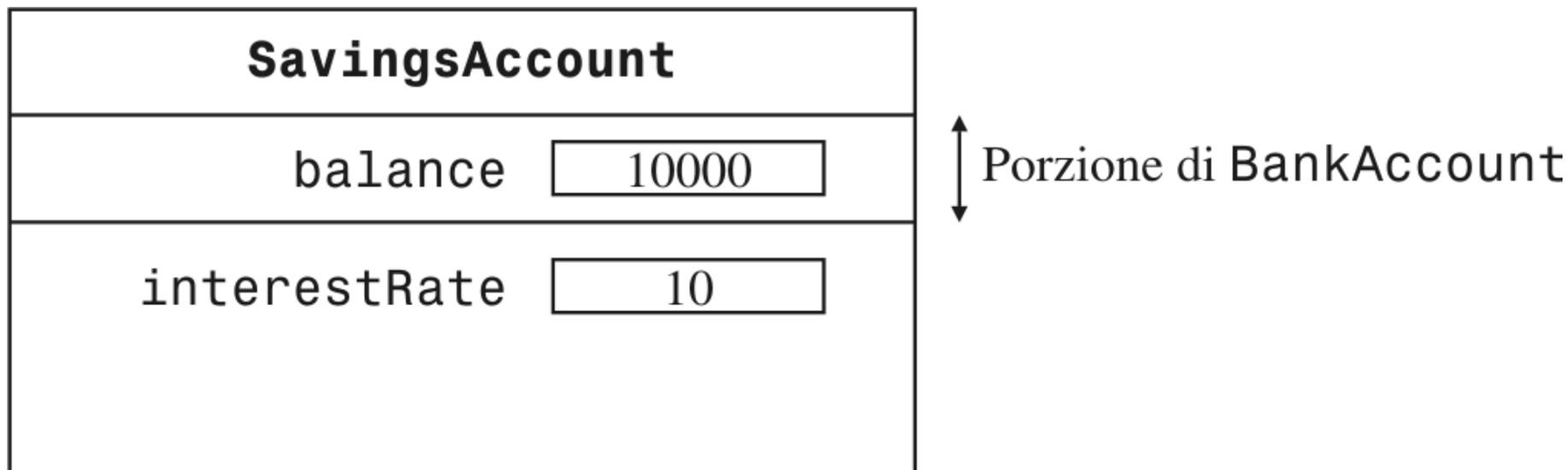


Figura 2

Sintassi di Java 10.1 Ereditarietà

```
class NomeSottoClasse extends NomeSuperclasse
{
    metodi
    campi di esemplare
}
```

Segue...

Sintassi di Java 10.1 Ereditarietà

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;
}
```

Obiettivo:

Definire una nuova classe che eredita da una classe esistente e definire i metodi e i campi di esemplare che si aggiungono alla nuova classe.

Gerarchie di ereditarietà

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

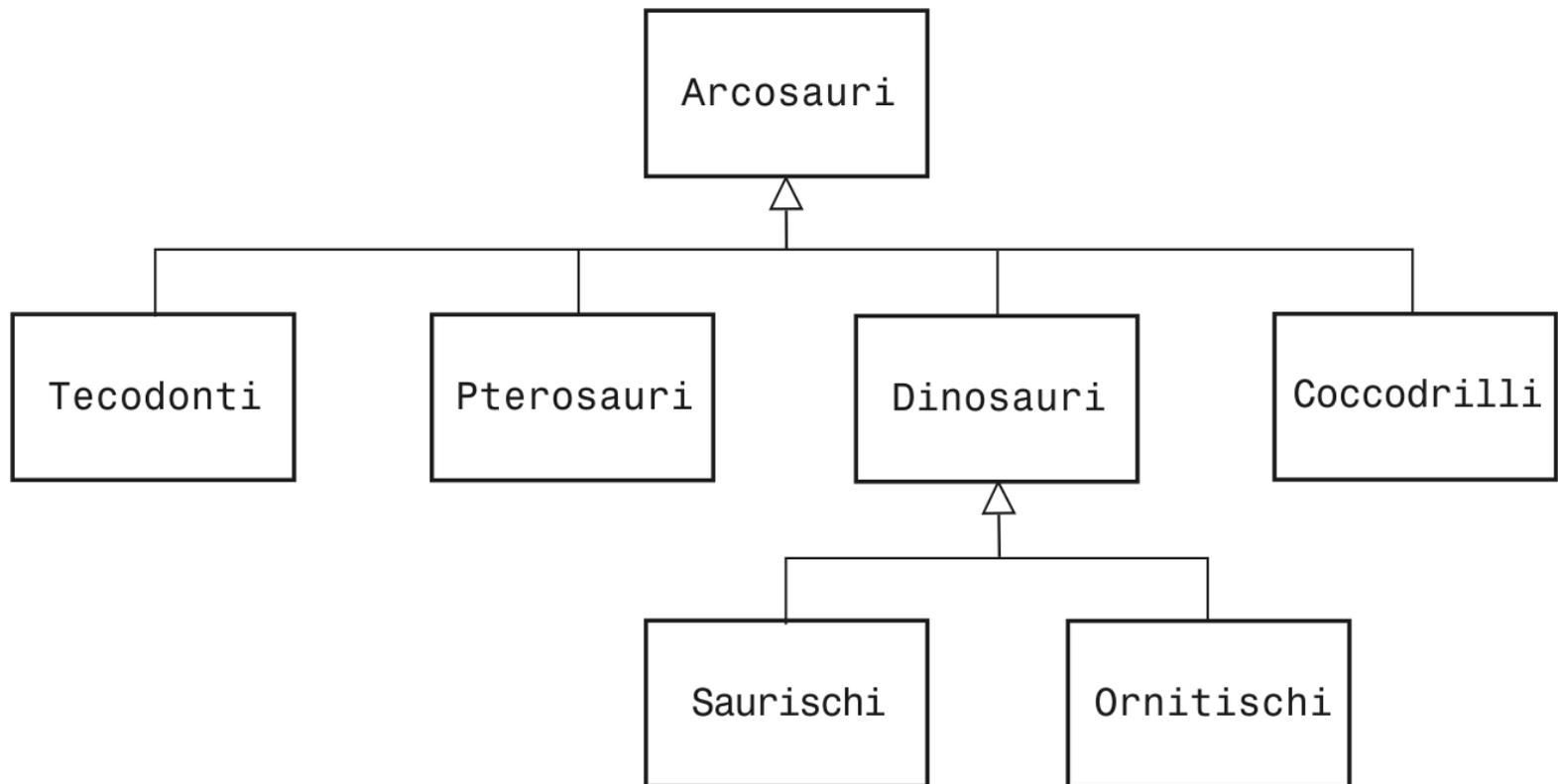


Figura 3:
Una parte della gerarchia dei rettili antichi

Una parte della gerarchia dei componenti per l'interfaccia utente Swing

- La superclasse `JComponent` ha i metodi `getWidth`, `getHeight`
- La classe `AbstractButton` ha i metodi per impostare e per ottenere testo e icone per i pulsanti

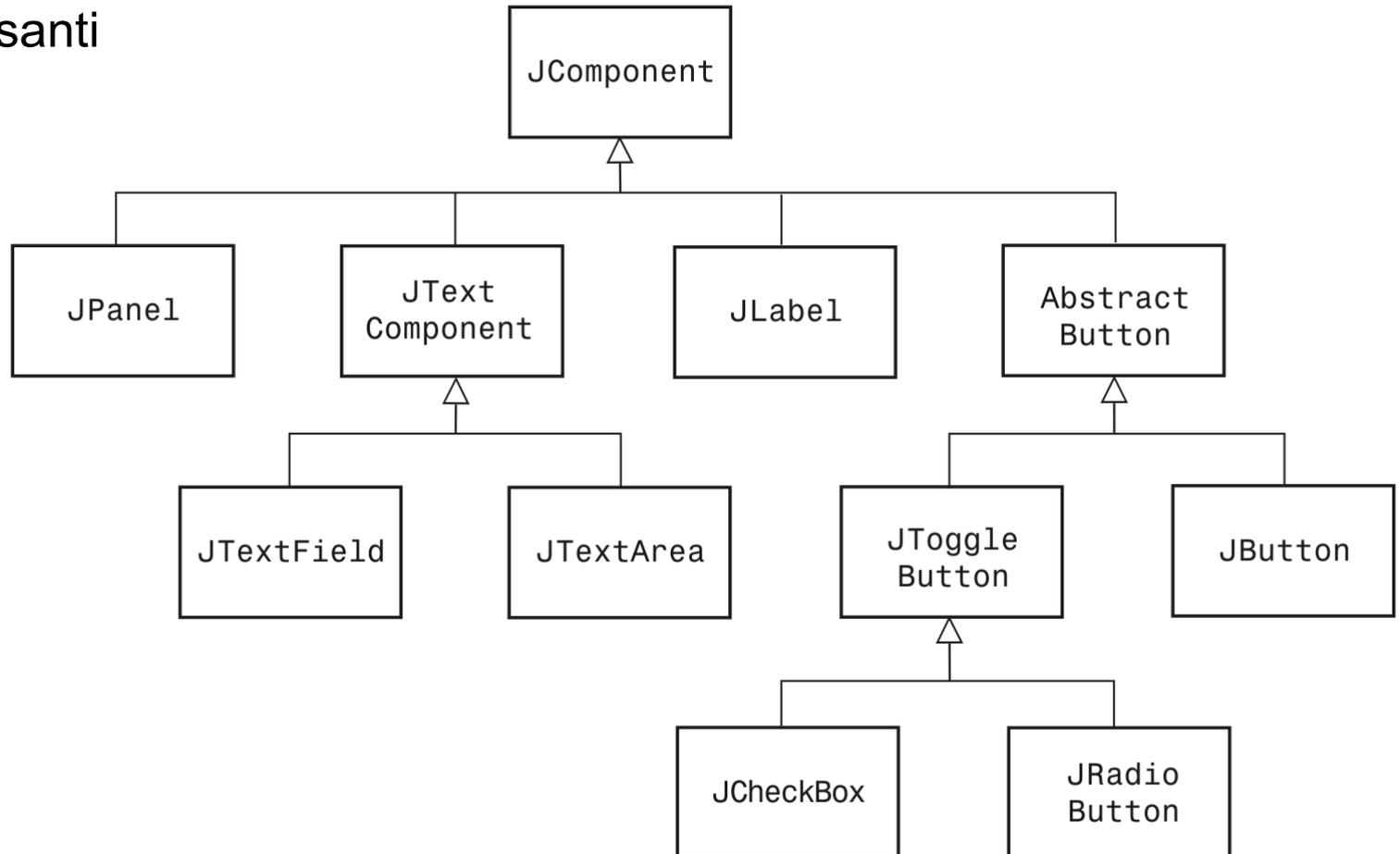


Figura 4

Un esempio più semplice: Gerarchia di ereditarietà per le classi dei conti bancari

- Considerate un banca che offre ai suoi clienti due tipi di conto:
 1. Il conto corrente (*checking account*) non offre interessi, concede un numero limitato di transazioni mensili gratuite e addebita una commissione per ciascuna transazione aggiuntiva.
 2. Il conto di risparmio (*savings account*) frutta interessi mensili.
- Tutti i conti bancari mettono a disposizione il metodo `getBalance`
- Tutti forniscono i metodi `deposit` e `withdraw`, sia pure con dettagli diversi nella realizzazione
- Il conto corrente ha bisogno di un metodo `deductFees` mentre i conti di risparmio hanno bisogno di un metodo `addInterest`

Un esempio più semplice: Gerarchia di ereditarietà per le classi dei conti bancari

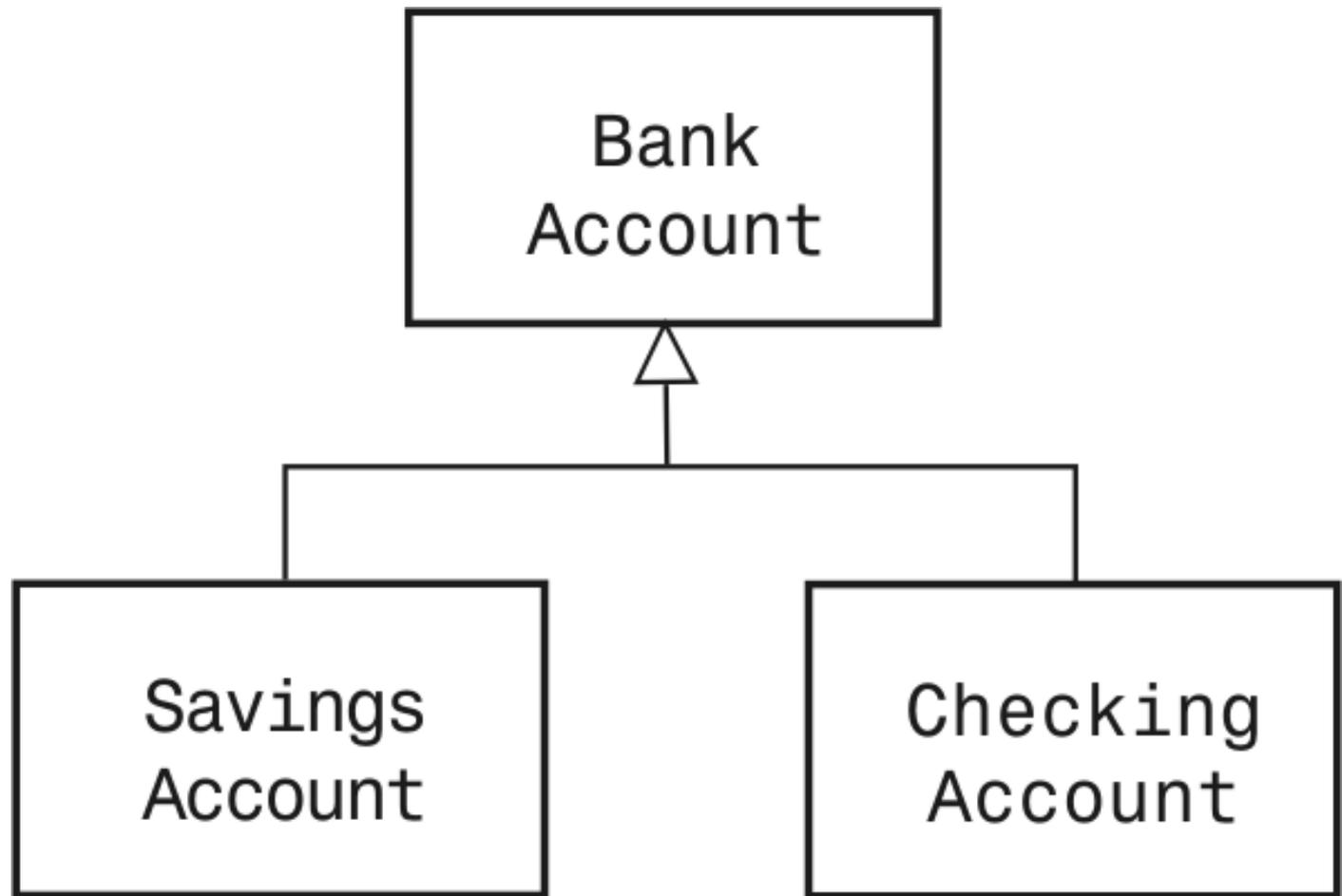


Figura 5

Ereditare metodi e campi di esemplare

- Definire nuovi metodi:
 - Potete *sovrascrivere* (o ridefinire, “override”) metodi della superclasse. Se specificate un metodo con la stessa *firma*, il metodo prevale su quello avente lo stesso nome e definito nella superclasse, ovvero la sovrascrive. Ogni volta che si applica il metodo a un oggetto della sottoclasse, viene eseguito il metodo ridefinito, invece di quello originale.
 - Potete *ereditare* metodi dalla superclasse. Se non sovrascrivete esplicitamente un metodo della superclasse, lo ereditate automaticamente e lo potete applicare agli oggetti della sottoclasse.
 - Potete *definire* nuovi metodi. Se si definisce un metodo che non esiste nella superclasse, il nuovo metodo si può applicare solo agli oggetti della sottoclasse.

Ereditare metodi e campi di esemplare

- La situazione dei *campi di esemplare* è piuttosto diversa: non potete sovrascrivere campi di esemplare e, per i campi di una sottoclasse, esistono solo due possibilità:
 1. tutti i campi di esemplare della superclasse sono ereditati automaticamente dalla sottoclasse.
 2. qualsiasi nuovo campo di esemplare che definite nella sottoclasse esiste solo negli oggetti della sottoclasse.
- Che cosa succede se definite un nuovo campo con lo stesso nome di un campo della superclasse?
 - Ciascun oggetto avrà *due* campi di esemplare con lo stesso nome
 - Tali campi possono contenere valori diversi
 - Tutto ciò genererà molto probabilmente confusione

Implementare la classe CheckingAccount

- Sovrascrive i metodi `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() { . . . } // nuovo metodo

    private int transactionCount; // nuovo campo di esemplare
}
```

- Ciascun oggetto della classe `CheckingAccount` ha due campi di esemplare:
 1. `balance` (ereditato da `BankAccount`)
 2. `transactionCount` (nuovo, definito in `CheckingAccount`)

Segue...

Implementare la classe `CheckingAccount`

- Agli oggetti di tipo `CheckingAccount` potete applicare quattro metodi:
 - `getBalance()` (ereditato da `BankAccount`)
 - `deposit(double amount)` (sovrascrive il metodo di `BankAccount`)
 - `withdraw(double amount)` (sovrascrive il metodo di `BankAccount`)
 - `deductFees()` (nuovo, definito in `CheckingAccount`)

Campi privati

- Consideriamo il metodo `deposit` di `CheckingAccount`

```
public void deposit(double amount)
{
    transactionCount++;
    // ora aggiungi amount al saldo
    . . .
}
```

- Non possiamo semplicemente sommare `amount` a `balance`
- `balance` è un campo privato della superclasse
- Una sottoclasse non ha accesso ai campi privati della sua superclasse
- Una sottoclasse deve usare un metodo pubblico della superclasse stessa

Invocare un metodo della superclasse

- Questo codice non funziona completamente. Il compilatore interpreta l'enunciato:
`deposit (amount)`
come se fosse:
`this.deposit (amount)`
- Quindi, il metodo chiamerà se stesso ripetutamente, facendo cadere il programma in una ricorsione infinita
- Dobbiamo quindi essere più precisi e specificare che intendiamo chiamare il metodo `deposit` della superclasse.

`super.deposit (amount)`

Segue...

Invocare un metodo della superclasse

- Si chiami ora il metodo `deposit` della classe `BankAccount`.
- I metodi rimanenti ora sono semplici:

```
public void deposit(double amount)
{
    transactionCount++;
    // ora aggiungi amount al saldo
    super.deposit (amount);
}
```

Segue...

Invocare un metodo della superclasse

```
public class CheckingAccount extends BankAccount
{
    . . .
    public void withdraw(double amount)
    {
        transactionCount++;
        // ora sottrai amount dal saldo
        super.withdraw(amount);
    }
}
```

Segue...

Invocare un metodo della superclasse

```
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;
}
```

Sintassi di Java 10.2

Invocare un metodo della superclasse

```
super.nomeMetodo (parametri);
```

Esempio:

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

Obiettivo:

Invocare un metodo della superclasse, anziché della classe attuale

Mettere in ombra variabili di esemplare

- Una sottoclasse non ha accesso alle variabili di esemplare private della superclasse.
- È un comune errore da principianti “risolvere” questo problema aggiungendo *un'altra* variabile di esemplare con lo stesso nome.

```
public class CheckingAccount extends BankAccount
{
    public void deposit(double amount)
    {
        transactionCount++;
        balance = balance + amount;
    }
    . . .
    private double balance; // NON FATELO
}
```

Segue...

Mettere in ombra variabili di esemplare

- Ora il metodo deposit si può certamente compilare, ma non aggiornerà il saldo giusto!

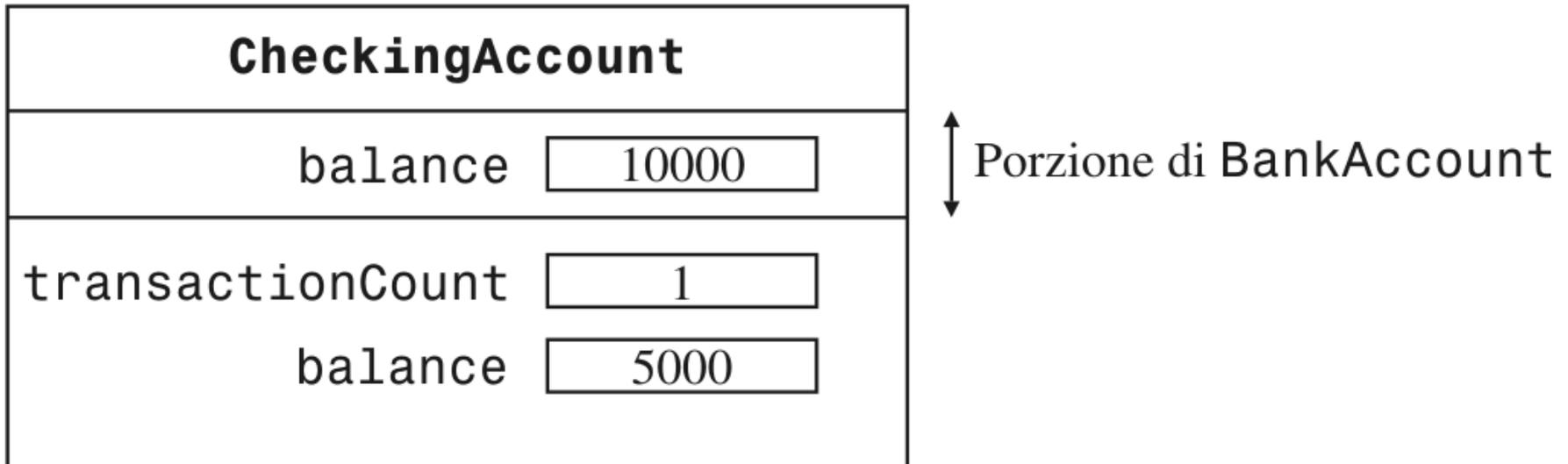


Figura 6: Mettere in ombra campi di esemplare

Costruzione di sottoclassi

- La parola chiave `super`, seguita da parametri di costruzione fra parentesi, indica una chiamata al costruttore della superclasse.
 - Facilita il riutilizzo dei costruttori

```
public class CheckingAccount extends BankAccount
{
    public CheckingAccount(double initialBalance)
    {
        // costruisci la superclasse
        super(initialBalance);
        // Inizializza il contatore delle transazioni
        transactionCount = 0;
    }
    . . .
}
```

Segue...

Costruzione di sottoclassi

- Si usa la parola chiave `super` nel primo enunciato del costruttore della sottoclasse
- Se un costruttore di una sottoclasse non chiama il costruttore della superclasse, la superclasse viene costruita mediante il suo costruttore predefinito (cioè il suo costruttore privo di parametri)
- Se, invece, tutti i costruttori della superclasse richiedono parametri, il compilatore genera un errore.

Sintassi di Java 10.3 Invocare un costruttore di superclasse

```
specificatoreDiAccesso NomeClasse (TipoDiParametro nomeParametro)  
{  
    super (parametri) ;  
    . . .  
}
```

Esempio:

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

Obiettivo:

Invocare il costruttore della superclasse. Ricordate che questo enunciato deve essere il primo enunciato del costruttore della sottoclasse.

Conversione di tipo fra sottoclasse e superclasse

- Spesso si ha la necessità di convertire un tipo di sottoclasse nel tipo della sua superclasse

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

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

Conversione di tipo fra sottoclasse e superclasse

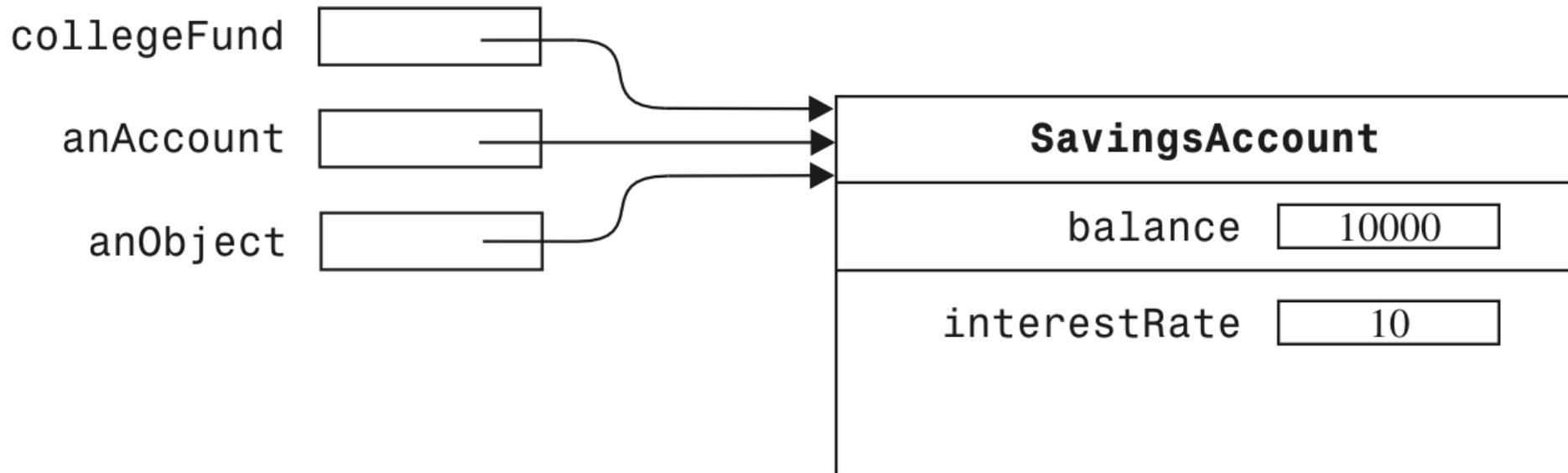


Figura 7:

Variabili di tipi diversi si riferiscono allo stesso oggetto

Conversione di tipo fra sottoclasse e superclasse

- La variabile di riferimento non conosce tutta la storia dell'oggetto a cui si riferisce. Esempio:

```
anAccount.deposit(1000); // Va bene
anAccount.addInterest();
//No, non é un metodo della classe a cui appartiene anAccount
```

- Quando convertite la sottoclasse nella sua superclasse:
 - Il valore del riferimento stesso non cambia: si tratta dell'indirizzo di memoria che contiene l'oggetto.
 - Ma dopo la conversione si hanno minori informazioni sull'oggetto

Segue...

Conversione di tipo fra sottoclasse e superclasse

- Perché mai qualcuno vorrebbe *intenzionalmente* avere meno informazioni a proposito di un oggetto e memorizzarne un riferimento in una variabile del tipo della sua superclasse?
 - Se volete *riutilizzare codice* che può elaborare oggetti della superclasse ma non oggetti della sottoclasse:

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

- Potete usare questo metodo per trasferire denaro da un conto bancario a un altro

Conversione di tipo fra sottoclasse e superclasse

- Più raramente c'è bisogno di effettuare la conversione opposta, da un riferimento di superclasse a un riferimento di sottoclasse

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

- Questo cast, però, è piuttosto pericoloso: se vi sbagliate viene lanciata un'eccezione.
- Come protezione contro i cast errati, potete usare l'operatore `instanceof`
- L'operatore `instanceof` verifica se un oggetto è di un particolare tipo.

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

Sintassi di Java 10.4 L'operatore `instanceof`

oggetto instanceof NomeTipo

Esempio:

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

Obiettivo:

Restituisce `true` se *oggetto* é un esemplare di *nomeTipo* (o di uno dei suoi sottotipi); altrimenti restituisce `false`.

Polimorfismo

- In Java il tipo di una variabile non determina completamente il tipo dell'oggetto a cui essa fa riferimento. In particolare, abbiamo visto che il tipo dell'oggetto può essere uno dei sottotipi del tipo della variabile. Altro caso interessante è quello delle interfacce.
- Le invocazioni di metodi *sono sempre decise in base al tipo reale dell'oggetto*, non in base al tipo del riferimento all'oggetto

```
BankAccount anAccount = new CheckingAccount();  
anAccount.deposit(1000);
```

Domanda: quale versione del metodo deposit() viene eseguita in seguito all'invocazione mostrata sopra?

Segue...

Polimorfismo

- il compilatore deve poter verificare che vengano invocati solamente metodi leciti:

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

- Questa verifica viene fatta staticamente, a tempo di compilazione, NON durante l'esecuzione.

Polimorfismo

- *Polimorfismo*: capacità di comportarsi in modo diverso con oggetti di tipo diverso.
- Come funziona il polimorfismo:

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

- In relazione ai **reali tipi** degli oggetti a cui fanno riferimento `anAccount` e `anotherAccount`, vengono invocate versioni diverse dei metodi `withdraw` e `deposit`

File AccountTester.java

```
01: /**
02:     Questo programma collauda la classe BankAccount
           e le sue sottoclassi.
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:
```

Segue...

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:     // simulazione della fine del mese
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:     System.out.println("Harry's checking balance: “
32:         + harrysChecking.getBalance());
33:     System.out.println("Expected: 1116");
34: }
35: }
```

File BankAccount.java

```
01: /**
02:     Un conto bancario ha un saldo che può essere modificato
03:     con versamenti e prelievi.
04: */
05: public class BankAccount
06: {
07:     /**
08:         Costruisce un conto bancario con saldo uguale a zero.
09:     */
10:     public BankAccount()
11:     {
12:         balance = 0;
13:     }
14:
15:     /**
16:         Costruisce un conot bancario con saldo assegnato.
17:         @param initialBalance il saldo iniziale
18:     */
```

Segue...

File BankAccount.java

```
19: public BankAccount(double initialBalance)
20: {
21:     balance = initialBalance;
22: }
23:
24: /**
25:     Versa denaro nel conto bancario.
26:     @param amount la somma da versare
27: */
28: public void deposit(double amount)
29: {
30:     balance = balance + amount;
31: }
32:
33: /**
34:     Preleva denaro dal conto bancario.
35:     @param amount la somma da prelevare
36: */
```

Segue...

File BankAccount.java

```
37: public void withdraw(double amount)
38: {
39:     balance = balance - amount;
40: }
41:
42: /**
43:     Restituisce il valore del saldo del conto bancario.
44:     @return il saldo attuale
45: */
46: public double getBalance()
47: {
48:     return balance;
49: }
50:
51: /**
52:     Trasferisce il denaro dal conto a un altro conto.
53:     @param amount la somma da trasferire
54:     @param other l'altro conto
55: */
```

Segue...

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:     Un conto corrente con commissioni per le transazioni
03: */
04: public class CheckingAccount extends BankAccount
05: {
06:     /**
07:         Costruisce un conto corrente con saldo assegnato.
08:         @param initialBalance il saldo iniziale
09:     */
10:     public CheckingAccount(double initialBalance)
11:     {
12:         // Costruisci la superclasse
13:         super(initialBalance);
14:
15:         // Azzera il conteggio delle transazioni
16:         transactionCount = 0;
17:     }
18:
```

Segue...

File CheckingAccount.java

```
19: public void deposit(double amount)
20: {
21:     transactionCount++;
22:     // Ora aggiungi amount al saldo
23:     super.deposit(amount);
24: }
25:
26: public void withdraw(double amount)
27: {
28:     transactionCount++;
29:     // Ora sottrai amount dal saldo
30:     super.withdraw(amount);
31: }
32:
33: /**
34:     Sottrae le commissioni accumulate
35:     e azzera il relativo conteggio.
36: */
```

Segue...

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:     Un conto bancario che matura interessi a un tasso fisso.
03: */
04: public class SavingsAccount extends BankAccount
05: {
06:     /**
07:         Costruisce un conto con tasso di interesse assegnato
08:         @param rate il tasso di interesse
09:     */
10:     public SavingsAccount(double rate)
11:     {
12:         interestRate = rate;
13:     }
14:
15:     /**
16:         Aggiunge al saldo del conto gli interessi maturati.
17:     */
```

Segue...

File SavingsAccount.java

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

Visualizza:

```
Mom's savings balance: 7035.0  
Expected: 7035  
Harry's checking balance: 1116.0  
Expected: 1116
```

Riuso dei costruttori della stessa classe

- Abbiamo visto che è possibile richiamare i costruttori della superclasse tramite `super(..)`
- Analogamente, è possibile riusare i costruttori della stessa classe tramite `this(...)`

```
BankAccount(double amount) {  
    balance = amount;  
}  
BankAccount() {  
    this(0.0);  
}
```

- La chiamata a `this(...)` deve comparire come prima istruzione all'interno del costruttore. Quindi non si possono usare sia `this()` che `super()` nello stesso costruttore (entrambe le chiamate devono essere le prime!)

Ereditarietà ed interfacce

- Le interfacce possono estendere altre interfacce. Per le interfacce è possibile usare ereditarietà multipla.

```
public interface A { void a(); }
public interface B { void b(); }
public interface C extends A, B { void c(); }
class Z implements C {
    void a() { ... }
    void b() { ... }
    void c() { ... }
}
```

- In `java.util` le interfacce `List` e `Set` estendono `Collection`, e `OrderedSet` estende `Set`.
 - `Collection`: collezione di oggetti
 - `List`: collezione ordinata
 - `Set`: collezione senza duplicati
 - `OrderedSet`: collezione ordinata senza duplicati

Metodi astratti

- Metodo astratto: metodo senza implementazione
- Esempio
 - `public abstract void deductFees();`
 - N.B. senza corpo, come nelle interfacce
- Utile quando non esiste un'implementazione di default accettabile per ogni sottoclasse e si vuole forzare il programmatore a scriverne una.
- Se si estende una classe con metodi astratti bisogna implementarli tutti.

Classi astratte

- Classe astratta: classe per cui non si possono creare oggetti
 - `public abstract class BankAccount {...}`
- Compromesso tra interfaccia e classi normali (concrete)
 - Diversamente dalle interfacce può avere attributi d'esemplare, costruttori e metodi concreti.
- Una classe che contiene un metodo abstract o che estende una classe che ne contiene ma non li ridefinisce, deve essere abstract.
- Una classe può essere abstract anche se non ha metodi abstract
- Per utilizzare una classe astratta i programmatori devono realizzarne delle sottoclassi e possono così creare oggetti ed usare i metodi e gli attributi ereditati dalla classe astratta.

Metodi e Classi astratte

- Non si possono costruire oggetti istanze di classi astratte ma si possono usare riferimenti ad oggetto del tipo di una classe astratta

```
BankAccount anAccount; // OK
anAccount = new BankAccount(); // Error-BankAccount is abstract
anAccount = new SavingsAccount(); // OK
anAccount = null; // OK
```

Metodi e classi final

- Attributi final sono costanti
- Classe final non può essere estesa da altre classi (es: classe String)
- Metodo final non può essere ridefinito in sottoclassi
- `public final boolean checkPassword(String password){...}`
 - In questa maniera non è possibile sovrascrivere il metodo con un altro che restituisce sempre true.

Controllo di accesso

- Un campo di una classe (dato, funzione o classe) puo` essere specificato con uno fra quattro livelli di accesso.
- Public: accesso incondizionato a (tutti i metodi di) tutte le classi
- Protected: accesso da tutte le sottoclassi, indipendentemente dal package in cui si trovano.
- Non qualificato (default): accesso da tutte le classi dello stesso pacchetto, anche se non sono sottoclassi
- Private: soltanto dalla classe stessa.
 - N.B. Ai membri privati di un oggetto è sempre possibile accedere da altri oggetti della stessa classe.

Segue...

Controllo di accesso ed ereditarietà

Access Modifiers	<i>Same Class</i>	<i>Same Package</i>	<i>Subclass</i>	<i>Other packages</i>
public	Y	Y	Y	Y
protected	Y	Y	Y	N
no access modifier	Y	Y	N	N
private	Y	N	N	N

Controllo di accesso

- Nota bene: appartenere allo stesso package da maggiori privilegi (non qualificato) rispetto ad essere una sottoclasse (protected).
- L'appartenenza ad un package non è vincolata: posso mettere una mia classe in un package qualunque.
- Posso invece impedire che altre classi estendano una mia classe dichiarandola come final.
- Non si può ridefinire un metodo con un modificatore più restrittivo! La restrizione sarebbe una falsa illusione: è sempre possibile accedere al metodo della superclasse tramite super.

Controllo di accesso

- I campi di esemplare e le variabili statiche delle classi dovrebbero essere sempre private. Ecco un ristretto numero di eccezioni:
 - Le costanti pubbliche (variabili `public static final`) sono utili e sicure.
 - È necessario che alcuni oggetti, come `System.out`, siano accessibili a tutti i programmi, pertanto dovrebbero essere pubblici.
 - In rare occasioni alcune classi in un pacchetto devono cooperare molto strettamente: può essere utile fornire ad alcune variabili l'accesso di pacchetto. Le classi interne sono una soluzione migliore.

Segue...

Controllo di accesso

- I metodi dovrebbero essere `public` o `private`.
- L'accesso a classi e a interfacce può essere *pubblico* o di *pacchetto*. Non è possibile dichiarare `private` o `protected` una classe.
 - Le classi che si usano per motivi limitati all'implementazione dovrebbero avere accesso di *pacchetto*; le potete nascondere ancora meglio spostandole in classi interne
 - in generale, le classi interne dovrebbero essere `private`; esistono alcuni rari esempi di classi interne pubbliche, come la classe `Ellipse2D.Double`
- L'accesso di *pacchetto* è utile molto di rado; la maggior parte delle variabili usa questo accesso per caso, perché il programmatore si è dimenticato di inserire la parola chiave `private`.

La superclasse universale `Object`

- Ciascuna classe che non estende un'altra classe estende automaticamente la classe `Object`

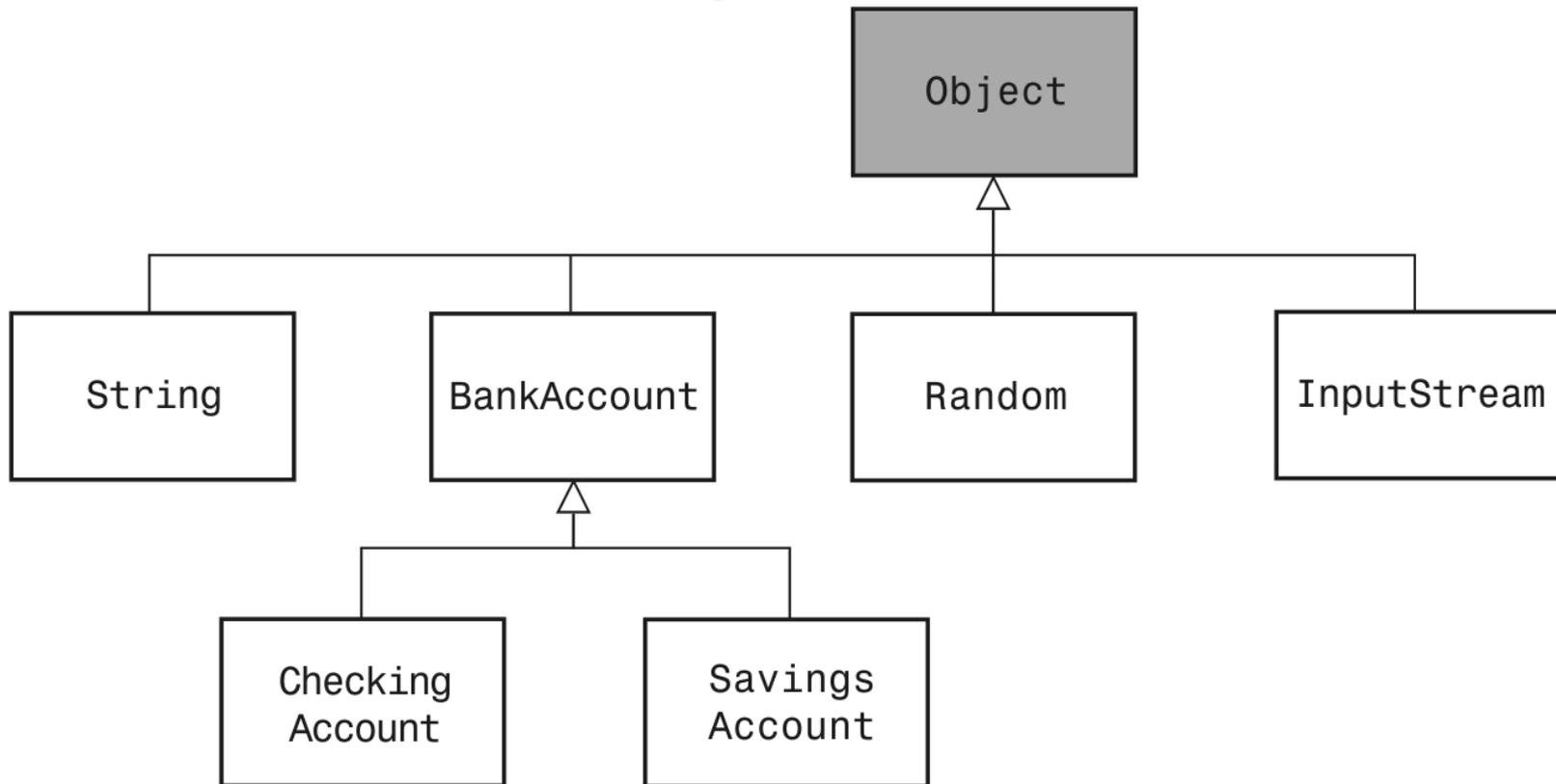


Figura 8: La classe `Object` é la superclasse di tutte le classi Java

La superclasse universale `Object`

- I metodi della classe `Object` sono assai generici. Ecco quelli più utili:
 - `String toString()`
 - `boolean equals(Object otherObject)`
 - `Object clone()`
- Nelle vostre classi è opportuno sovrascrivere questi metodi.

Sovrascrivere il metodo `toString`

- Il metodo `toString` restituisce una rappresentazione in forma di stringa per ciascun oggetto
- E' utile per scovare errori logici

```
Rectangle box = new Rectangle(5, 10, 20, 30);  
String s = box.toString();  
// s si riferisce alla stringa  
  
// "java.awt.Rectangle[x=5,y=10,width=20,height=30]"
```

- `Object.toString`: ciò che viene stampato è il nome della classe seguito dal *codice hash* dell'oggetto

```
BankAccount momsSavings = new BankAccount(5000);  
String s = momsSavings.toString();  
//s si riferisce a qualcosa di simile a "BankAccount@d24606bf"
```

Sovrascrivere il metodo `toString`

- Se si vuole sapere cosa si trova all'interno dell'oggetto, dobbiamo sovrascrivere il metodo: il metodo `toString` della classe `Object` non sa cosa si trovi all'interno della nostra classe `BankAccount`

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

- Questo sistema funziona meglio:

```
BankAccount momsSavings = new BankAccount(5000);  
String s = momsSavings.toString();  
// s si riferisce alla stringa "BankAccount[balance=5000]"
```

Sovrascrivere il metodo `equals`

- Il metodo `equals` viene invocato quando volete confrontare il contenuto di due oggetti

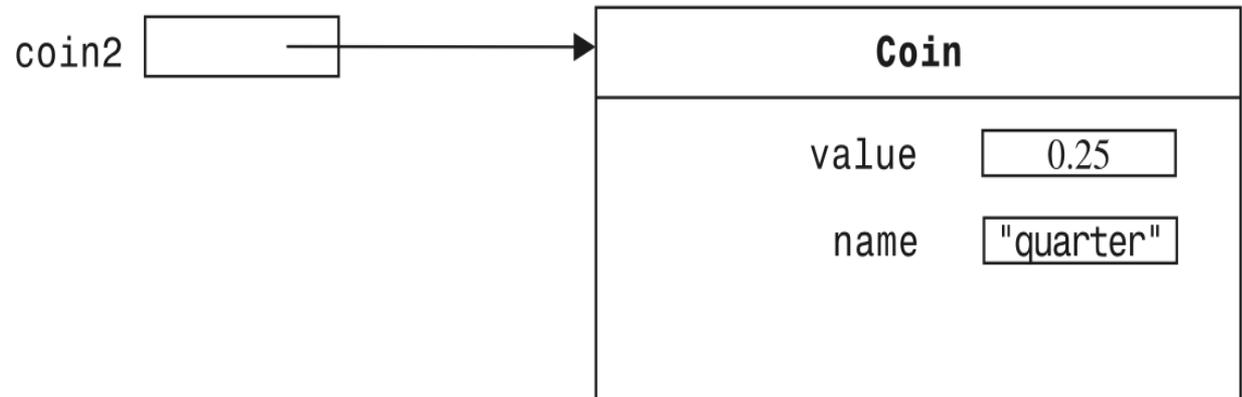
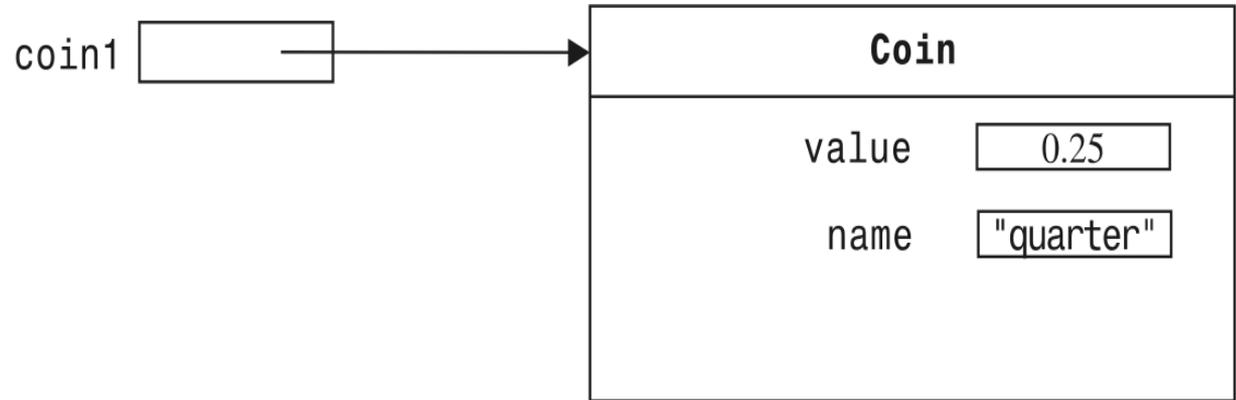


Figura 9:
Due riferimenti a
oggetti uguali

Sovrascrivere il metodo `equals`

- Esiste una differenza rispetto al confronto mediante l'operatore `==`, che verifica se due riferimenti puntano allo stesso oggetto

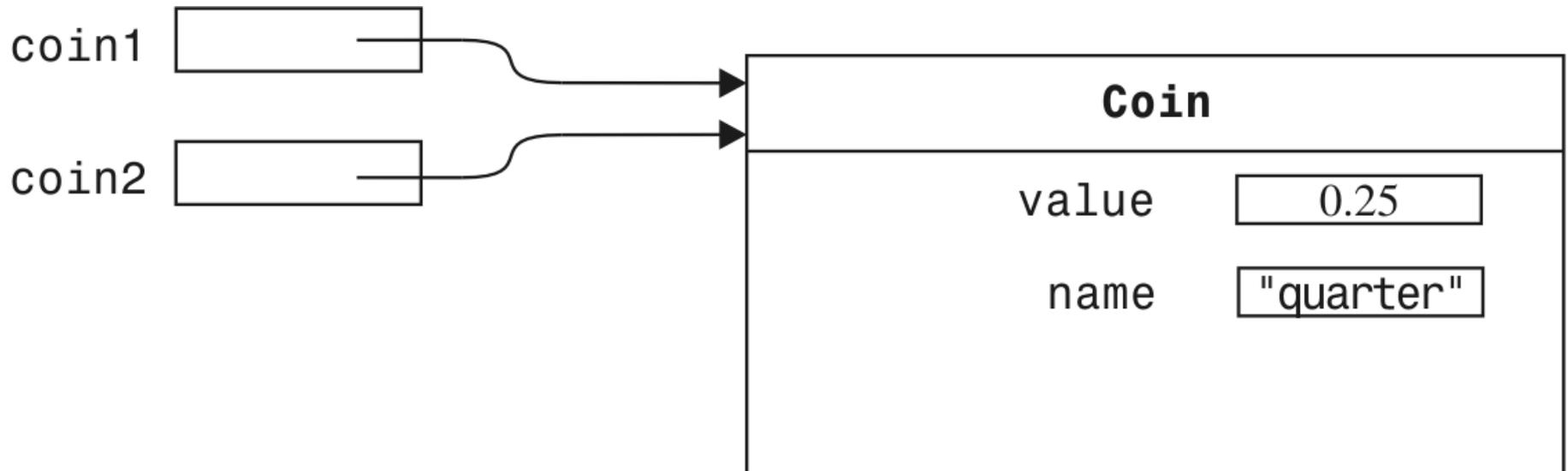


Figura 10:
Due riferimenti allo stesso oggetto

Sovrascrivere il metodo `equals`

- Definite il metodo `equals` in modo che verifichi se due oggetti hanno identiche informazioni di stato.
- Quando ridefinite il metodo, non vi è permesso modificare la firma del metodo stesso, dovete invece eseguire un *cast* sul parametro

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

- Dovreste sovrascrivere anche il metodo `hashCode`, in modo che oggetti uguali abbiano lo stesso codice hash

Sovrascrivere il metodo `equals`

```
public boolean equals(Object obj) {  
    if (this == obj) return true;  
    if (obj == null) return false;  
    if (getClass() != obj.getClass()) return false;  
    final BankAccount other = (BankAccount) obj;  
    if (balance != other.balance) return false;  
    return true;  
}
```

equals

- The equals method implements an equivalence relation on non-null object references:
- reflexive: for any non-null reference value `x`, `x.equals(x)` should return true.
- symmetric: for any non-null reference values `x` and `y`, `x.equals(y)` should return true if and only if `y.equals(x)` returns true.
- transitive: for any non-null reference values `x`, `y`, and `z`, if `x.equals(y)` returns true and `y.equals(z)` returns true, then `x.equals(z)` should return true.
- consistent: for any non-null reference values `x` and `y`, multiple invocations of `x.equals(y)` consistently return true or consistently return false, provided no information used in equals comparisons on the objects is modified.

equals

- For any non-null reference value `x`, `x.equals(null)` should return `false`.
- The `equals` method for class `Object` implements the most discriminating possible equivalence relation on objects; that is, for any non-null reference values `x` and `y`, this method returns `true` if and only if `x` and `y` refer to the same object (`x == y` has the value `true`).

Sovrascrivere il metodo `clone`

- Copiare un riferimento ad un oggetto fornisce semplicemente due riferimenti allo stesso oggetto:

```
BankAccount account = new BankAccount(1000);  
BankAccount account2 = account;
```

Segue...

Sovrascrivere il metodo `clone`

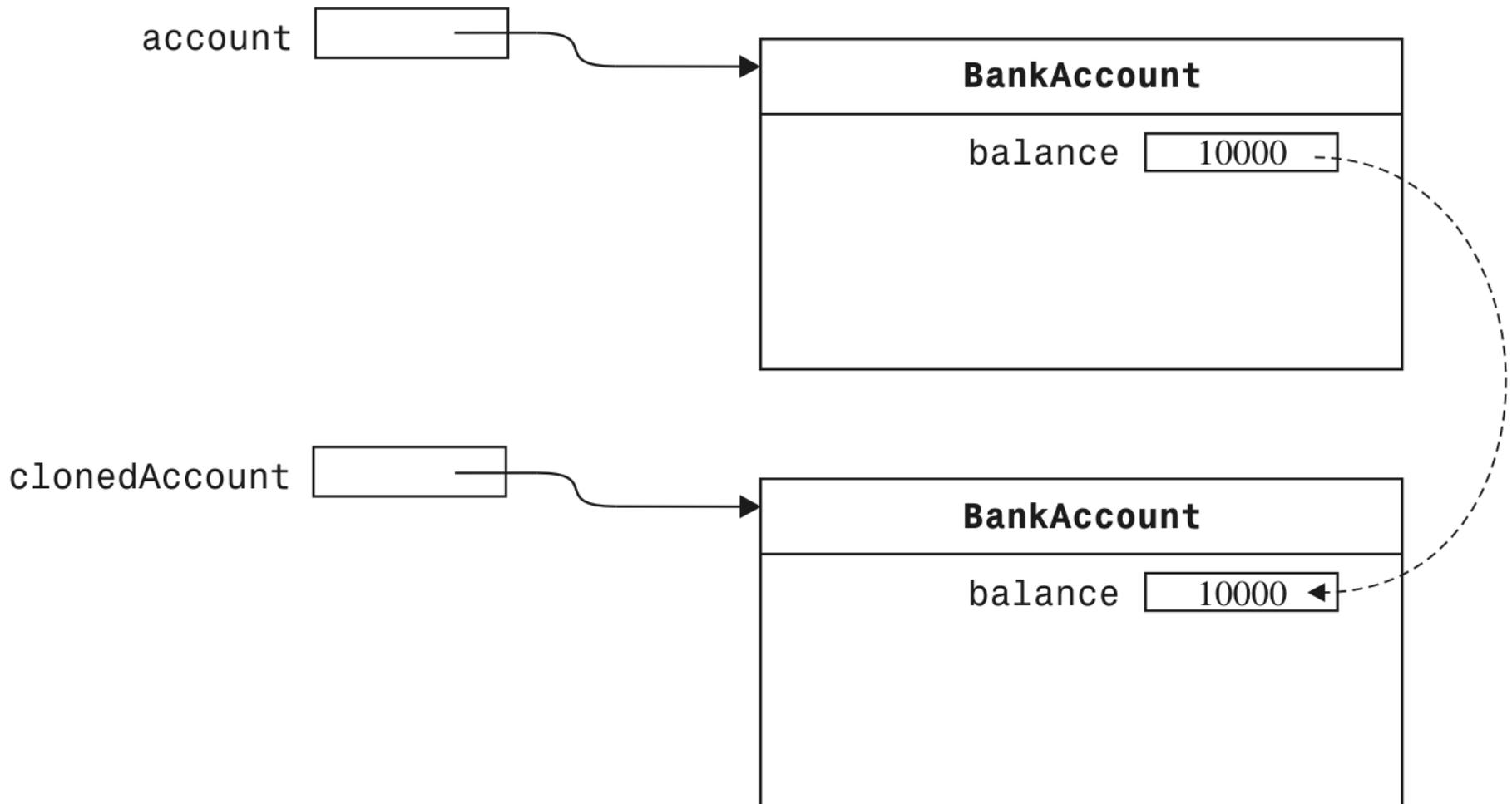


Figura 11: Clonazione di un oggetto

Sovrascrivere il metodo `clone`

- La realizzazione del metodo `clone` è un po' più difficile della realizzazione dei metodi `toString` e `equals`: consultate gli Argomenti avanzati 10.6 per maggiori dettagli.
- Supponiamo che qualcuno abbia realizzato il metodo `clone` per la classe `BankAccount`. Ecco come invocarlo:

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

- Il tipo di dato che viene restituito dal metodo `clone` è `Object`. Quando invocate il metodo, dovete usare un cast

Realizzare il metodo `clone`

- Crea una copia detta *copia superficiale*.

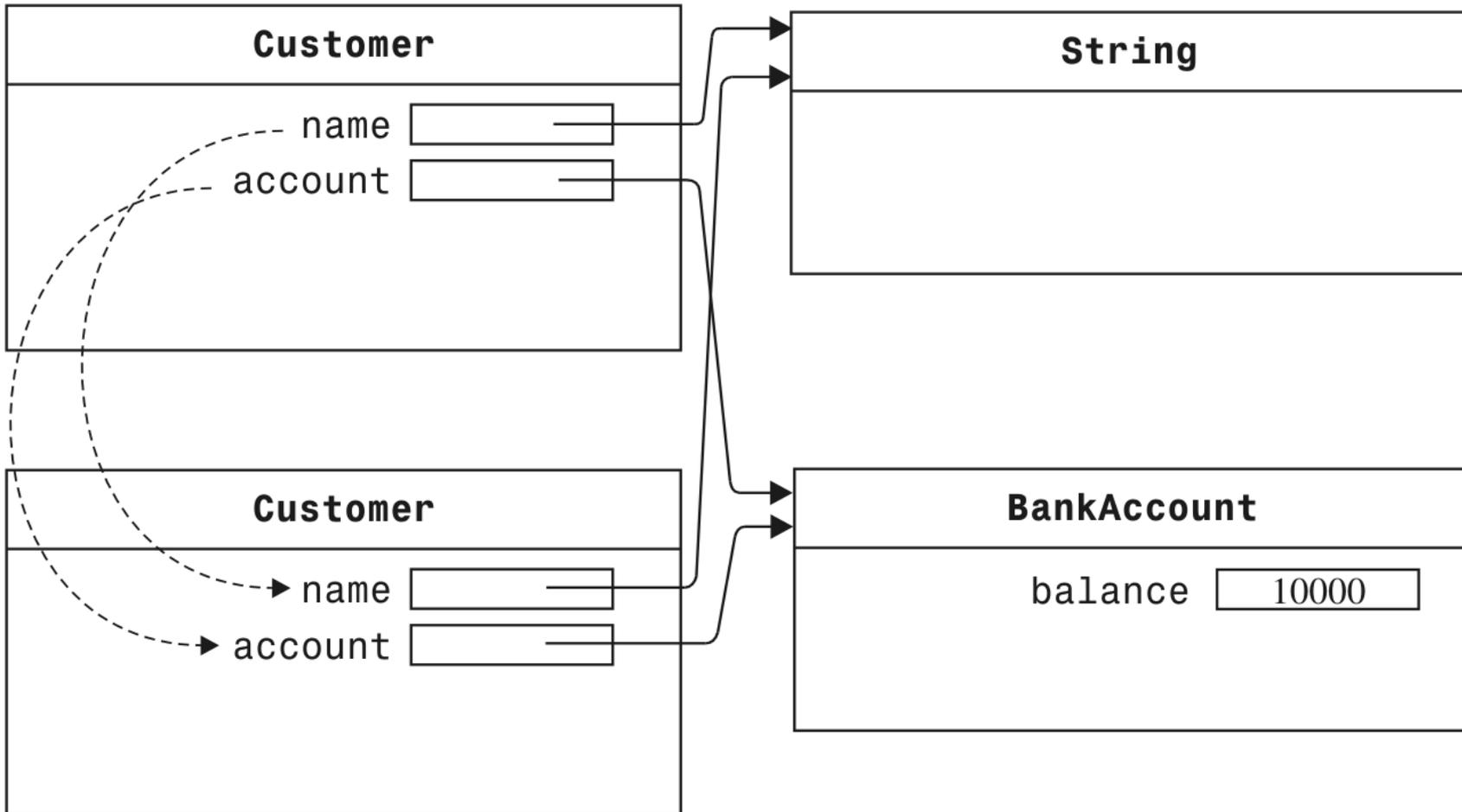


Figure 12: Il metodo `clone` crea una copia superficiale

Realizzare il metodo `clone`

- Il metodo `Object.clone` non clona sistematicamente tutti i sotto-oggetti
- Bisogna usarlo con cautela. Cosa può accadere se effettuiamo una copia profonda incondizionatamente?
- Il metodo viene dichiarato `protected`: questo impedisce di chiamare `x.clone()` per sbaglio, se la classe a cui appartiene `x` non ha ridefinito `clone` come metodo pubblico.
- `Object.clone` controlla che l'oggetto da clonare realizzi l'interfaccia `Cloneable`; in caso contrario, lancia un'eccezione `CloneNotSupportedException`.
- Queste protezioni implicano che i legittimi invocanti di `Object.clone()` paghino un prezzo: intercettare l'eccezione *anche se la loro classe realizza* `Cloneable`.

Esempio di `clone` superficiale

- La copia superficiale va bene quando nella classe ci sono solo attributi di tipo primitivo o oggetti immutabili.
- ```
public class CloneSuperficiale implements Cloneable {
 private int contatore;
 private String descrizione;
 protected Object clone() throws CloneNotSupportedException {
 try {
 CloneSuperficiale o = (CloneSuperficiale) super.clone();
 return o;
 } catch(CloneNotSupportedException e) { throw e;}
 ..}
```

# Esempio di clone profonda

- La copia profonda diventa necessaria nella classe ci sono oggetti non immutabili: se clonassi il solo riferimento potrei comunque modificare l'oggetto
- ```
public class CloneProfonda implements Cloneable {  
    private BankAccount conto;  
    protected Object clone() throws CloneNotSupportedException {  
        try {  
            CloneProfonda o = (CloneProfonda) super.clone();  
            o.conto = (BankAccount) conto.clone();  
            return o;  
        } catch(CloneNotSupportedException e) { throw e;}  
        ..}  
}
```

HashCode, equals, clone

- Se `a.equals(b)` allora deve essere vero che `a.hashCode() == b.hashCode()`
- Se `! a.equals(b)` allora dovrebbe essere vero (performance) che `a.hashCode() != b.hashCode()`
- Deve valere `x.clone() != x`
- `clone` non deve utilizzare `new` per costruire il clone ma deve utilizzare `super.clone()`
- Tipicamente
 - `x.clone().getClass() == x.getClass()`
 - True if the returned object is obtained by calling `super.clone()`
 - `x.clone().equals(x)`

L'ereditarietà per personalizzare i frame

- Usate l'ereditarietà per rendere i frame complessi più semplici da capire
- Definite una sottoclasse di `JFrame`
- Memorizzate i componenti in campi di esemplare
- Inizializzatevi nel costruttore della vostra sottoclasse
- Se il codice del costruttore diventa troppo complesso, aggiungete semplicemente metodi ausiliari

Elaborare testo in ingresso

- Usate `JTextField` per fornire all'utente uno spazio per digitare dati in ingresso

```
final int FIELD_WIDTH = 10;
```

```
final JTextField rateField = new JTextField(FIELD_WIDTH);
```

- Posizionate un esemplare di `JLabel` accanto a ogni campo di testo

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

- Fornite un pulsante che l'utente possa premere per segnalare che i dati sono pronti per essere elaborati



Figura 14:
Un'applicazione con
un campo di testo

Elaborare testo in ingresso

- Quando viene premuto tale pulsante, il suo metodo `actionPerformed` legge i dati introdotti dall'utente nei campi di testo (usando `getText`)

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

File InvestmentViewer3.java

```
01: import javax.swing.JFrame;
02:
03: /**
04:     Questo programma visualizza la crescita di un investimento.
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: }
```

File 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:     Questo frame visualizza la crescita di un investimento con
11:     interesse variabile.
12: */
13: public class InvestmentFrame extends JFrame
14: {
15:     public InvestmentFrame()
16:     {
17:         account = new BankAccount(INITIAL_BALANCE);
18:         // Usiamo campi di esemplare per i componenti
19:         resultLabel = new JLabel("balance: " + account.getBalance());
20:
```

File InvestmentFrame.java

```
21:         // usiamo metodi ausiliari
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:
```

File InvestmentFrame.java

```
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()
```

File InvestmentFrame.java

```
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: }
```

Aree di testo

- Usate un oggetto di tipo `JTextArea` per visualizzare più linee di testo

- Potete specificare il numero di righe e di colonne

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

- Usate `setText` per impostare il testo di un campo di testo o di un'area di testo

- Usate `append` per aggiungere testo alla fine di un'area di testo

- Usate caratteri di “andata a capo” per separare le linee

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

- Se volete usare un campo di testo o un'area di testo soltanto per la visualizzazione, potete usare il metodo `setEditable`

```
textArea.setEditable(false);
```

Aree di testo

- Usate `JScrollPane` per aggiungere barre di scorrimento a un'area per il testo

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

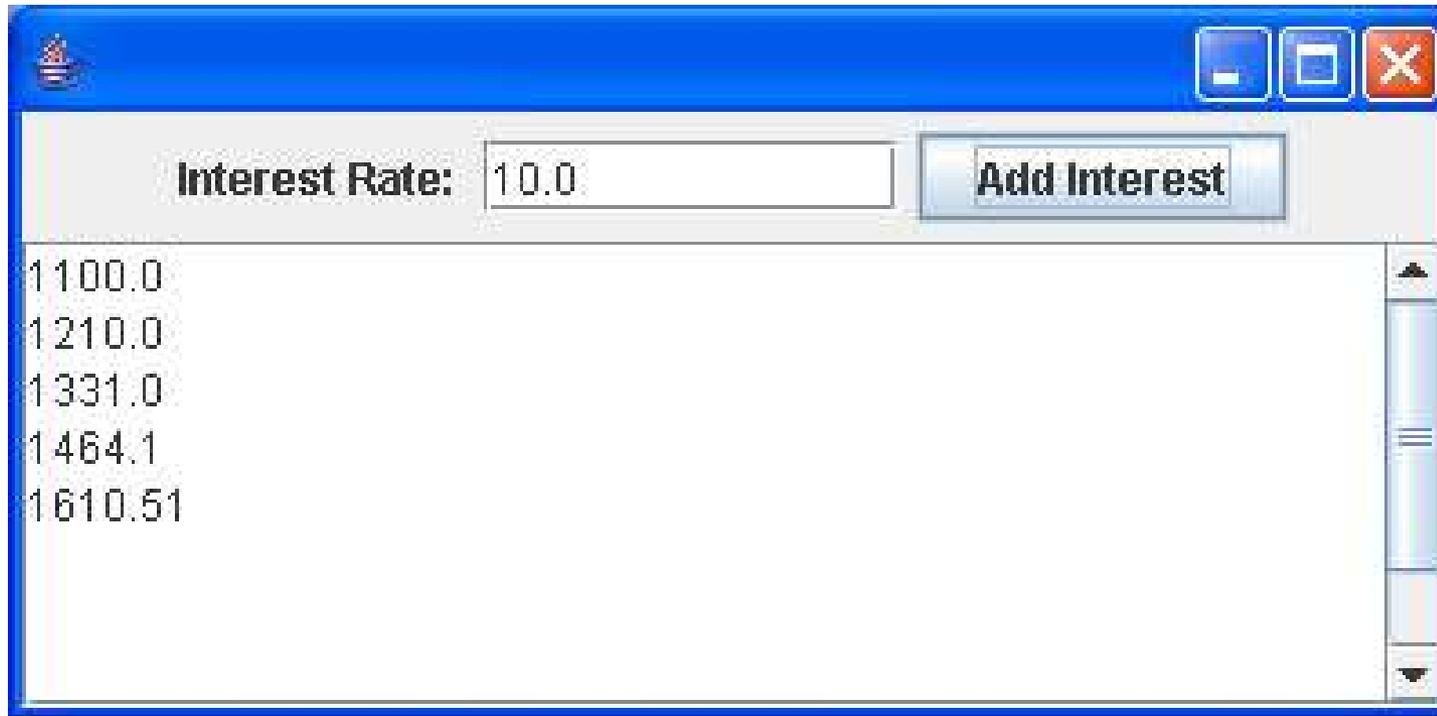


Figura 15: L'applicazione di visualizzazione di investimenti con un campo di testo

File 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:     Questo frame visualizza la crescita di un investimento con
    interesse variabile.
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:
```

Segue...

File InvestmentFrame.java

```
22:         // Usiamo metodi ausiliari
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:
```

Segue...

File InvestmentFrame.java

```
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);
```

Segue...

File InvestmentFrame.java

```
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: }
```