Ereditarietà

Ereditarietà

- E' un meccanismo per estendere classi esistenti, aggiungendo altri metodi e campi.

```
class SavingsAccount extends BankAccount
{   nuovi metodi
    nuove variabili d'istanza
}
```

- Consente il riutilizzo del codice:
  - tutti i metodi e le variabili d'istanza della classe BankAccount vengono ereditati automaticamente

Classe: SavingsAccount

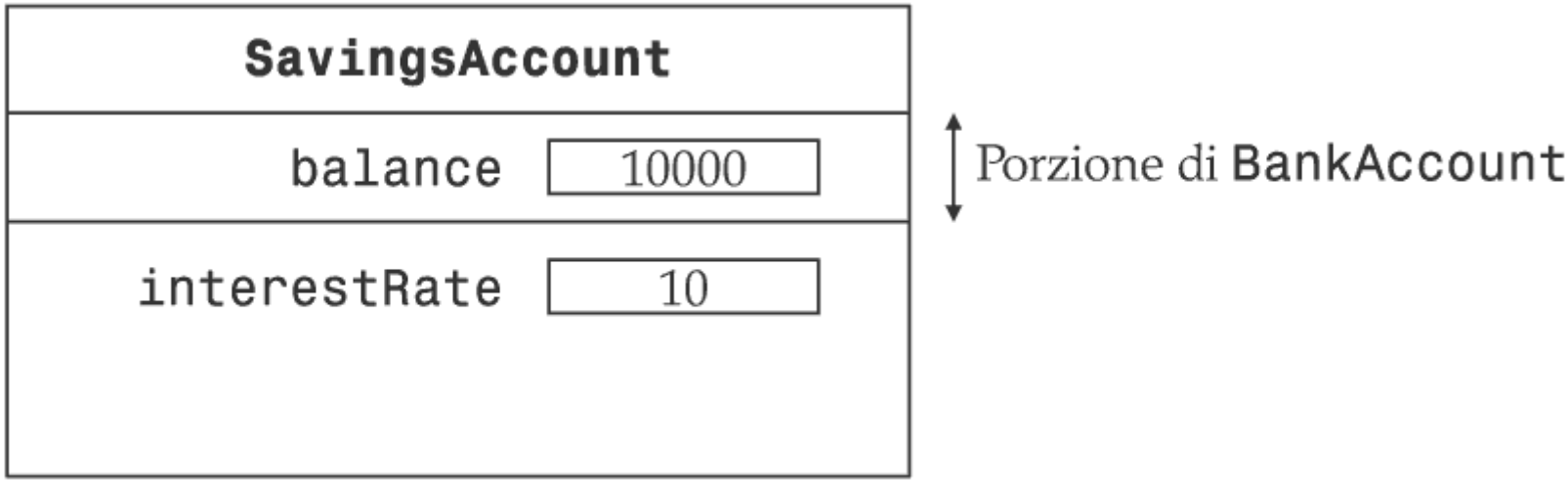
```
public class SavingsAccount extends BankAccount
   {
    public SavingsAccount(double rate){
       interestRate = rate;
    }
    public void addInterest(){
       double interest = getBalance()*interestRate/100;
       deposit(interest);
    }
    private double interestRate;
   }
```

- estende BankAccount aggiungendo variabile di istanza interestRate e metodo addInterest()

Riutilizzo di codice

- La classe `SavingsAccount` eredita i metodi della classe `BankAccount`:
  - `withdraw`
  - `deposit`
  - `getBalance`
- Inoltre, `SavingsAccount` ha un metodo che calcola gli interessi maturati e li versa sul conto
  - `addInterest`

Stati di oggetti SavingsAccount



`SavingsAccount` eredita la variabile di istanza balance da `BankAccount` e ha una variabile di istanza in più: interestRate

SavingsAccount: metodo addInterest()

- Il metodo `addInterest` chiama i metodi `getBalance` e `deposit` della superclasse
  - Non viene specificato alcun oggetto per le invocazioni di tali metodi
  - Viene usato il parametro implicito di `addInterest`

```
double interest = this.getBalance()
                  * this.interestRate / 100;
this.deposit(interest);
```

  - Non si può usare direttamente balance
    - è dichiarato private in BankAccount

Ereditarietà

- La classe preesistente (più generale) è detta SUPERCLASSE e la nuova classe (più specifica) è detta SOTTOCLASSE
  - **`BankAccount`**: superclasse
  - **`SavingsAccount`**: sottoclasse

- Gli oggetti della sottoclasse sono anche del tipo della superclasse

Ereditarietà vs Interfacce

- un'interfaccia non è una classe:
  - non ha uno stato, né un comportamento
  - è un elenco di metodi da implementare

- una superclasse è una classe:
  - ha uno stato e un comportamento che sono ereditati dalla sottoclasse

Classe Object

- La classe `Object` è la superclasse di tutte le classi.
  - Ogni classe è una sottoclasse di `Object`

- Ha un piccolo numero di metodi, tra cui
  - `String toString()`
  - `boolean equals(Object otherObject)`
  - `Object clone()`

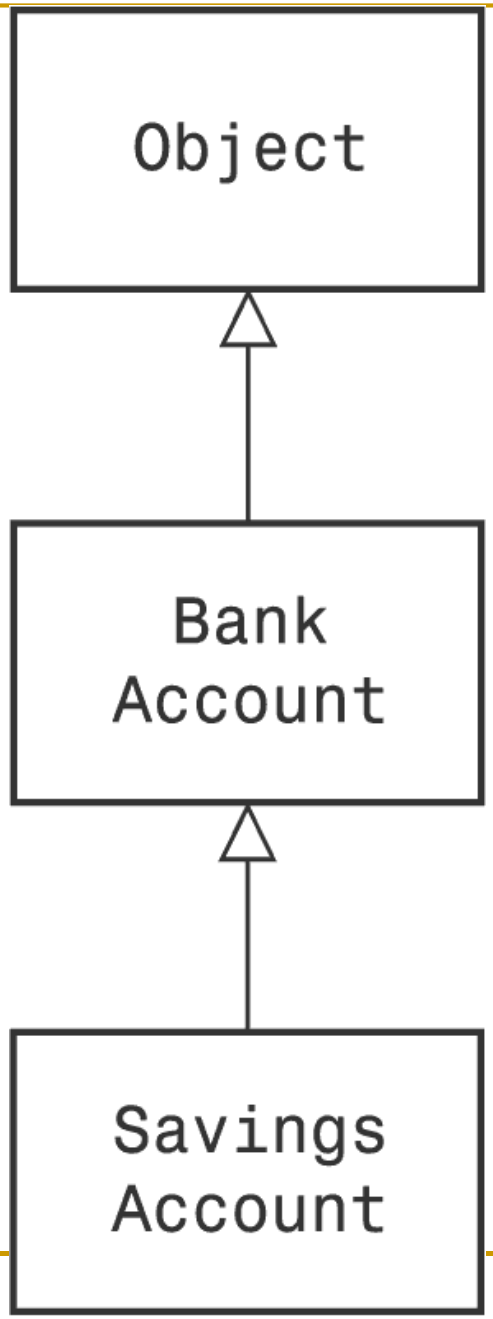
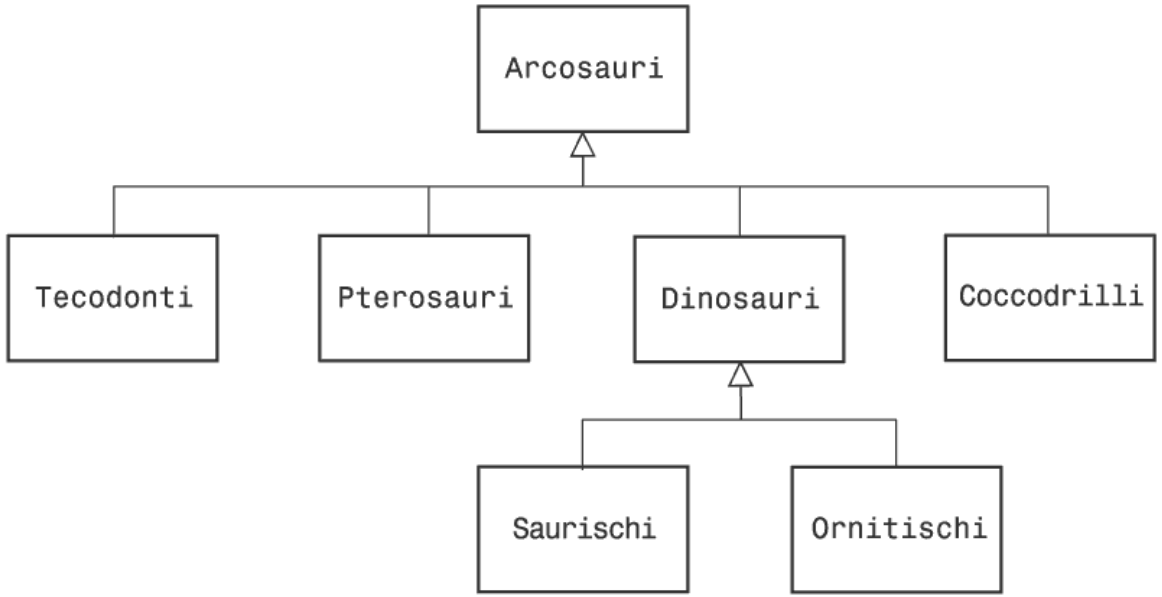


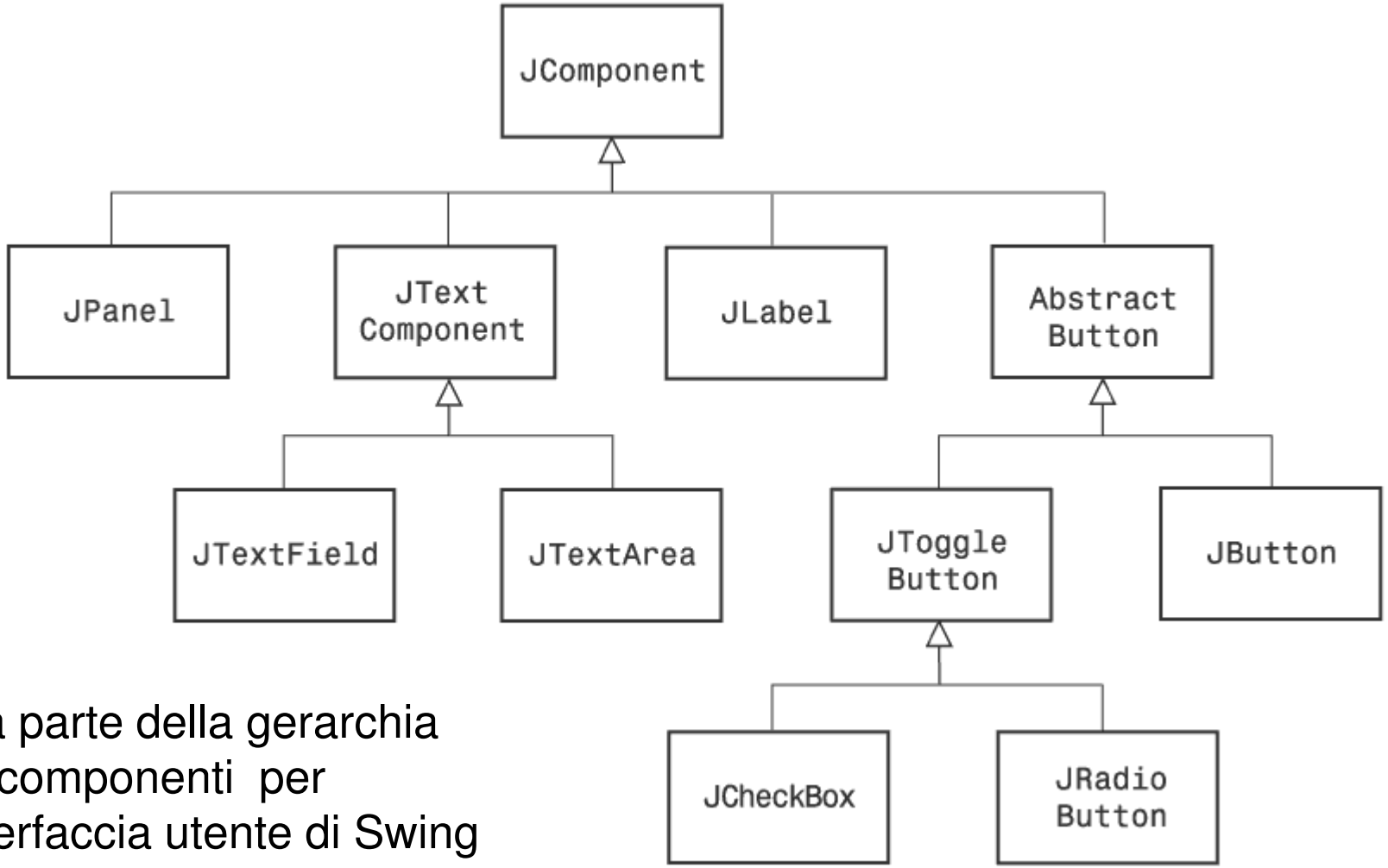
Figura 1
Diagramma UML che mostra ereditarietà (relazione “è un”)

Gerarchia per ereditarietà

- In Java le classi sono organizzate in maniera gerarchica attraverso l'ereditarietà
  - Le classi che rappresentano concetti più generali sono più vicine alla radice
  - Le classi più specializzate sono alla fine delle diramazioni



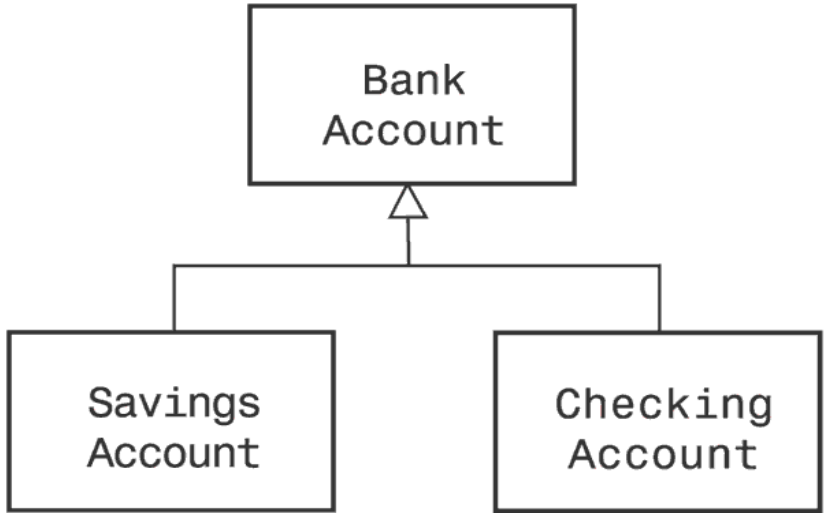
Gerarchia per ereditarietà



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

Gerarchia per ereditarietà

- Consideriamo una banca che offre due tipi di conto:
  - Checking account, che non offre interessi, concede un certo numero di operazioni mensili gratuite e addebita una commissione per ogni operazione aggiuntiva
  - Savings account, che frutta interessi mensili



Gerarchia di BankAccount

- Determiniamo i comportamenti:
  - Tutti i conti forniscono i metodi
    - `getBalance, deposit e withdraw`
  - Per CheckingAccount bisogna contare le transazioni
  - Per CheckingAccount è necessario un metodo per addebitare le commissioni mensili
    - `deductFees`
  - `SavingsAccount` ha un metodo per sommare gli interessi
    - `addInterest`

Metodi di una sottoclasse

Tre possibilità per definirli:

- Sovrascrivere i metodi della superclasse
  - la sottoclasse ridefinisce un metodo con la stessa firma del metodo della superclasse
  - vale il metodo della sottoclasse
- Ereditare metodi dalla superclasse
  - la sottoclasse non ridefinisce un metodo della superclasse
- Definire nuovi metodi
  - la sottoclasse definisce un metodo che non esiste nella superclasse

Variabili di istanza di sottoclassi

Due possibilità:

- Ereditare variabili di istanza
  - Le sottoclassi ereditano tutte le variabili di istanza della superclasse
- Definire nuove variabili di istanza
  - Esistono solo negli oggetti della sottoclasse
  - Possono avere lo stesso nome di quelle nella superclasse, ma non le sovrascrivono
    - Quelle della sottoclasse mettono in ombra quelle della superclasse

La nuova classe: CheckingAccount

```
public class BankAccount
   {  public double getBalance() {…}
      public void deposit(double d) {…}
      public void withdraw(double d) {…}
      private double balance;
   }

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

CheckingAccount

- Ciascun oggetto di tipo `CheckingAccount` ha due variabili stanza
  - `balance` (ereditata da `BankAccount`)
  - `transactionCount` (nuova)
- E' possibile applicare quattro metodi
  - `getBalance()` (ereditato da `BankAccount`)
  - `deposit(double)` (sovrascritto)
  - `withdraw(double)` (sovrascritto)
  - `deductFees()` (nuovo)

CheckingAccount: metodo deposit

```
public void deposit(double amount)
   {
     transactionCount++; // NUOVA VBL IST

     //aggiungi amount al saldo
     balance = balance + amount; //ERRORE
   }
```

- `CheckingAccount` ha una variabile balance, ma è una variabile privata della superclasse!
- I metodi della sottoclasse non possono accedere alle variabili private della superclasse

CheckingAccount: metodo deposit

- Possiamo invocare il metodo `deposit` della classe `BankAccount`…
- Ma se scriviamo

```
deposit(amount)
```

viene interpretato come

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

cioè viene chiamato il metodo che stiamo scrivendo!

- Dobbiamo chiamare il metodo `deposit` della superclasse:

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

CheckingAccount: metodo deposit

```
public void deposit(double amount)
  {

     transactionCount++; // NUOVA VBL IST.


     //aggiungi amount al saldo
     super.deposit(amount);
  }
```

CheckingAccount: metodo withdraw

```
public void withdraw(double amount)
   {
      transactionCount++; // NUOVA VBL IST.

      //sottrai amount al saldo
      super.withdraw(amount);
   }
```

CheckingAccount: metodo deductFees

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

Regole da rispettare in sovrascrittura

- A partire da Java 5.0, è stato introdotto il *covariant* return type:
  - si può sovrascrivere un metodo usando la stessa firma ma cambiando il tipo di return con un sottotipo
  - Prima di Java 5.0, nel sovrascrivere un metodo sia i parametri espliciti che il tipo di return dovevano coincidere

Esempio

- Sia B un sottotipo di A (classe B estende A)
- Quanto segue è consentito:

```
public class Super {
        public A getObject() {
                return new A();
        }
}

public class Sub extends Super {
        public B getObject() {
                return new B();
        }
}
```

Esempio

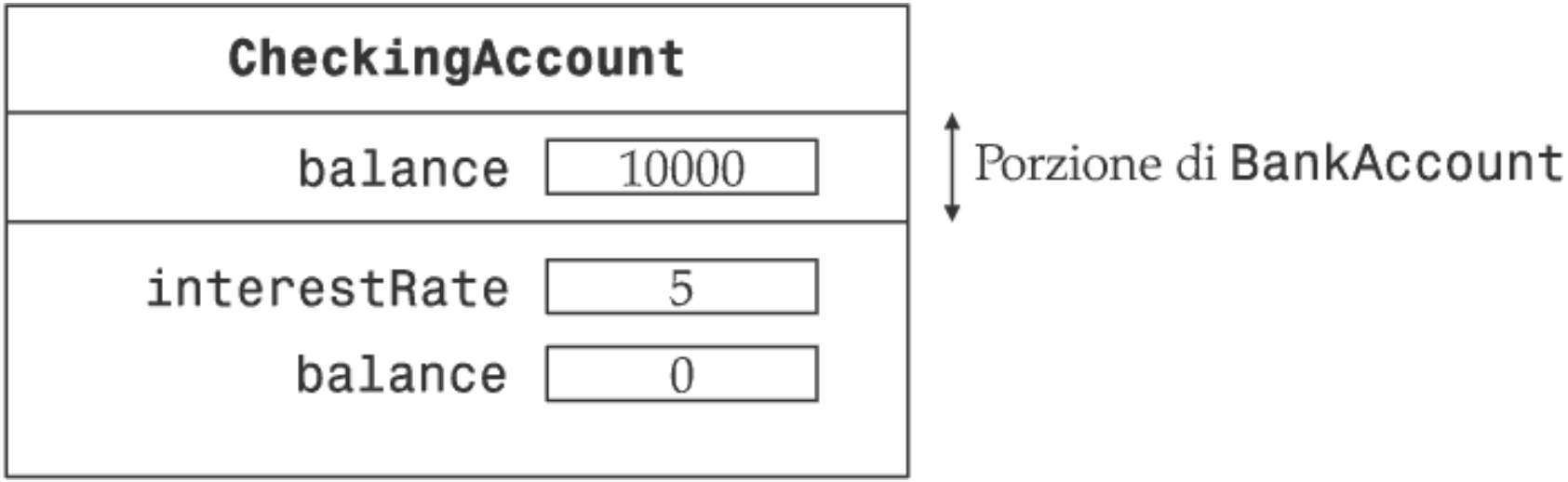
- Quanto segue non è consentito, ad esempio se String non è un sottotipo di A:

```
public class Super {
        public A getObject() {
                return new A();
        }
}

public class Sub extends Super {
        public String getObject() {
                return “prova”;
        }
}
```

Mettere in ombra variabili istanza

- Una sottoclasse non ha accesso alle variabili private della superclasse
- E' un errore comune risolvere il problema creando un'altra variabile di istanza con lo stesso nome
- La variabile della sottoclasse mette in ombra quella della superclasse



Costruttore in sottoclassi

- Per invocare il costruttore della superclasse dal costruttore di una sottoclasse uso la parola chiave **`super`** seguita dai parametri del costruttore
  - Deve essere il primo comando del costruttore della sottoclasse

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

Costruttore in sottoclassi

- Se il costruttore della sottoclasse non chiama il costruttore della superclasse,

  viene invocato il costruttore predefinito

  della superclasse

- Ad es., se il costruttore di `CheckingAccount` non invoca il costruttore di `BankAccount`,

  viene impostato il saldo iniziale a zero

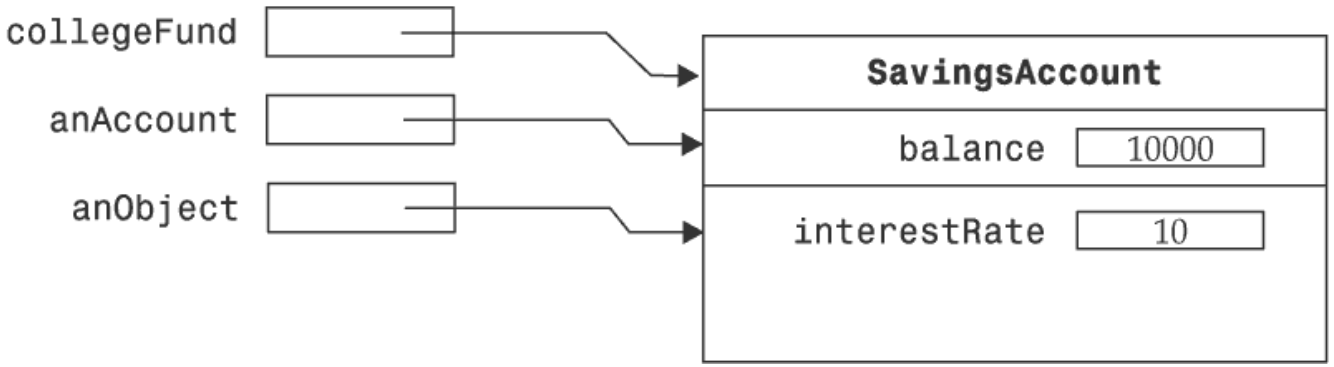
Conversione da Sottoclasse a Superclasse

- Si può salvare un riferimento ad una sottoclasse in una variabile del tipo di una superclasse:

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

- Il riferimento a qualsiasi oggetto può essere memorizzato in una variabile di tipo `Object`

```
Object anObject = collegeFund;
```



Conversione da Sottoclasse a Superclasse

- Se si usa una variabile del tipo di una superclasse per un riferimento ad un oggetto della sottoclasse

  i metodi della sottoclasse non sono visibili:

```
anAccount.deposit(1000); //Va bene
//deposit è un metodo della classe BankAccount

anAccount.addInterest(); // Errore
//addInterest non è un metodo della classe BankAccount

anObject.deposit(); // Errore
//deposit non è un metodo della classe Object
```

Polimorfismo

- Metodo **`transfer`**:

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

- Possiamo usarlo con paremetri di un qualsiasi tipo di **`BankAccount`**

Polimorfismo

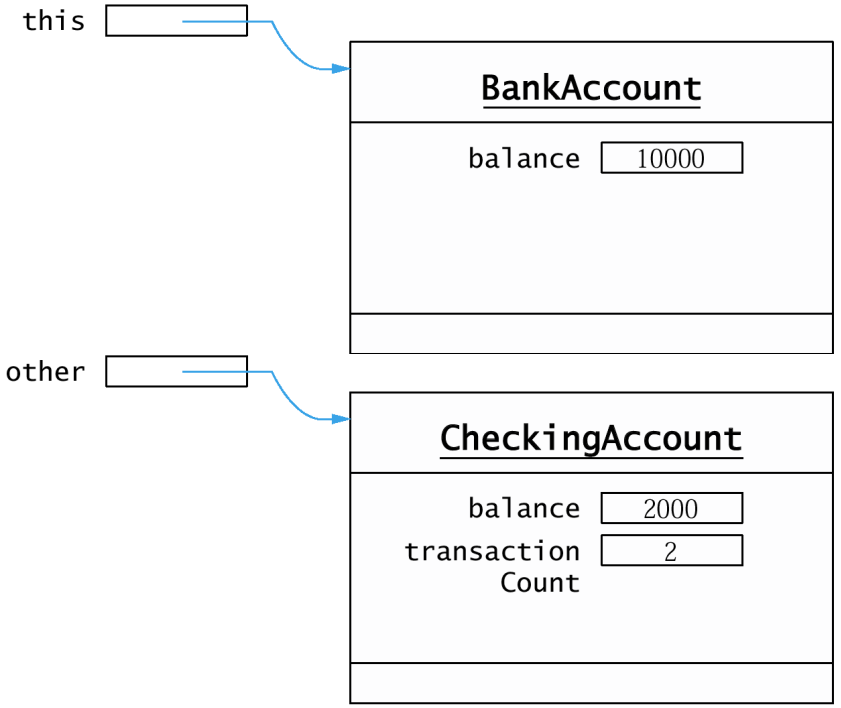
- E' lecito passare un riferimento di tipo **`CheckingAccount`** a un metodo che si aspetta un riferimento di tipo **`BankAccount`**

```
BankAccount momsAccount = . . .;
CheckingAccount harrysChecking = . . .;
momsAccount.transfer(harrysChecking, 1000);
```

- Il compilatore copia il riferimento all'oggetto harrisChecking (di tipo CheckingAccount) nella variabile **`other`** del tipo della superclasse BankAccount

Polimorfismo

- a tempo di compilazione non è possibile stabilire il tipo effettivo della variabile other di transfer
  - non è possibile stabilire ad esempio che si riferisce a un oggetto di tipo **`CheckingAccount`**
- l'unica informazione è che other è di tipo **`BankAccount`**



Polimorfismo

- il metodo transfer invoca il metodo `deposit`.
  - Quale?
- la decisione avviene a runtime (late binding)
  - dallo spazio dell'oggetto si segue il link al codice da eseguire
- su un oggetto di tipo `CheckingAccount` viene invocato `deposit` di `CheckingAccount`
  - vale il tipo effettivo dell'oggetto non il tipo della variabile

File BankAccount.java

```
/**
   Un conto bancario ha un saldo che può essere modificato
   con versamenti e prelievi.
*/
public class BankAccount
{
   /**
      Costruisce un conto bancario con saldo zero.
   */
   public BankAccount()
   {
      balance = 0;
   }

   /**
      Costruisce un conto bancario con un saldo assegnato.
      @param initialBalance il saldo iniziale
   */
   public BankAccount(double initialBalance)
   {
      balance = initialBalance;
   }
```

```
/**
      Versa denaro nel conto bancario.
      @param amount la somma da versare
   */
   public void deposit(double amount)
   {
      balance += amount;
   }

   /**
      Preleva denaro dal conto bancario.
      @param amount la somma da prelevare
   */
   public void withdraw(double amount)
   {
      balance -= amount;
   }

   /**
      Restituisce il valore del saldo del conto bancario.
      @return il saldo attuale
```

```
   */
   public double getBalance()
   {
      return balance;
   }

   /**
      Trasferisce denaro dal conto ad un altro conto.
      @param amount la somma da trasferire
      @param other l'altro conto
   */
   public void transfer(double amount, BankAccount other)
   {
      withdraw(amount);
      other.deposit(amount);
   }

   private double balance;
}
```

File CheckingAccount.java

```
/**
   Un conto corrente che addebita commissioni per ogni
transazione.
*/
public class CheckingAccount extends BankAccount
{
   /**
      Costruisce un conto corrente con un saldo assegnato.
      @param initialBalance il saldo iniziale
   */
   public CheckingAccount(double initialBalance)
   {
      // chiama il costruttore della superclasse
      super(initialBalance);

      // inizializza il conteggio delle transazioni
      transactionCount = 0;
   }
```

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

//metodo sovrascritto
 public void withdraw(double amount){
      transactionCount++;
      // ora sottrai amount dal saldo
      super.withdraw(amount);
 }
```

```
//metodo nuovo
public void deductFees(){
      if (transactionCount > FREE_TRANSACTIONS){
         double fees = TRANSACTION_FEE *
               (transactionCount - FREE_TRANSACTIONS);
         super.withdraw(fees);
      }
      transactionCount = 0;
  }


  private int transactionCount;

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

File SavingsAccount.java

```
/**
   Un conto bancario che matura interessi ad un tasso
fisso.
*/
public class SavingsAccount extends BankAccount{
   /**
      Costruisce un conto bancario con un tasso di
      interesse assegnato.
      @param rate il tasso di interesse
   */
   public SavingsAccount(double rate){

      interestRate = rate;
   }
```

```
   /**
      Aggiunge al saldo del conto gli interessi maturati.
   */
      public void addInterest()
      {
            double interest = getBalance()
                  * interestRate / 100;
            deposit(interest);
      }

      private double interestRate;
}
```

File AccountTest.java

```
/**
   Questo programma collauda la classe BankAccount
   e le sue sottoclassi.
*/
public class AccountTest{
   public static void main(String[] args){
      BankAccount momsSavings
            = new SavingsAccount(0.5);

      BankAccount harrysChecking
            = new CheckingAccount(100);

      momsSavings.deposit(10000);
      momsSavings.transfer(2000, harrysChecking);
      harrysChecking.withdraw(1500);
      harrysChecking.withdraw(80);
```

```
        momsSavings.transfer(1000, harrysChecking);
        harrysChecking.withdraw(400);

        // simulazione della fine del mese
        ((SavingsAccount) momsSavings).addInterest();
        ((CheckingAccoount) harrysChecking).deductFees();

        System.out.println("Mom's savings balance = $"
              + momsSavings.getBalance());

        System.out.println("Harry's checking balance = $"
              + harrysChecking.getBalance());
    }
}
```

Osservazioni su ereditarietà: precondition

- Situazione:
  - B è una sottoclasse di A e sovrascrive un metodo foo: int foo (int x)
  - foo in A ha precondition x>0
- Possiamo cambiare questa precondition in B?
  - Si, ma la nuova condizione non può rafforzare la vecchia condizione:

  la precondition di A deve implicare la precondition di B (es. x >= 0)
  - Un oggetto di B può essere assegnato ad una variabile di tipo A e quindi può essere utilizzato in un pezzo di codice che rispetta le precondition di A

Osservazioni su ereditarietà: postcondition

- Situazione:
  - B è una sottoclasse di A e sovrascrive un metodo foo: int foo (int x)
  - foo in A ha postcondition return value >= 0
- Possiamo cambiare questa postcondition in B?
  - Si, ma la nuova condizione non può rilassare la vecchia condizione:

  la postcondition di B deve implicare la postcondition di A (es. x > 0)
  - Un oggetto di B può essere assegnato ad una variabile di tipo A e quindi può essere utilizzato in un pezzo di codice che assume le postcondition di A

Estensione di Interfacce

- il meccanismo delle estensioni si può applicare anche alle interfacce
  - effetto incrementale sulla richiesta di metodi da implementare
- un'interfaccia può estendere più di una interfaccia, così come una classe può implementare più interfacce
- Es.

```
public interface DataIO extends
                    DataInput, DataOutput {……}
```

(DataInput e DataOutput sono interfacce)

Classi astratte

- Un ibrido tra classe ed interfaccia
- Può avere metodi normalmente implementati ed altri *astratti*
  - Un metodo astratto non ha implementazione:

```
public abstract void deductFees();
```

- Le classi non astratte sono dette concrete
- Le classi concrete che estendono una classe astratta sono OBBLIGATE ad implementarne i metodi astratti
  - nelle sottoclassi di classi concrete non si è obbligati ad implementare i metodi della superclasse

Classi astratte

- non si possono creare oggetti di classi astratte
  - ...ci sono metodi non implementati
    (come nelle interfacce!)

```
public abstract class BankAccount{
   public abstract void deductFees();
   ...
}
```

Classi astratte

- E’ possibile dichiarare astratta una classe priva di metodi astratti
  - Effetto: non possono essere istanziati oggetti con il suo costruttore

- Le classi astratte forzano la realizzazione di sottoclassi

- Un metodo astratto ha il vantaggio di non dover scrivere un metodo fittizio che può essere ereditato dalle sottoclassi
  - indica chiaramente l'obbligo di dover "sovrascrivere" il metodo

Confronto Classi Astratte - Interfacce

- Le classi astratte e le interfacce sono simili:
  - catturano astrazioni che possono raggruppare aspetti comuni di concetti differenti
- Le classi in generale forniscono un'implementazione
  - Anche le classi astratte solitamente implementano alcuni metodi
- L'ereditarietà è usata per raggruppare concetti strettamente correlati
  - un'estensione può essere vista come un raffinamento del concetto espresso dalla superclasse (e quindi si può estendere una sola classe)
- Le interfacce possono essere usate per enfatizzare un aspetto comune di concetti eterogenei
  - Ad es., l'uso di Measurable, Measurer, etc.
  - Una classe può implementare più interfacce, ciascuna corrispondente ad un aspetto del concetto espresso dalla classe

Riepilogando….

- Un'interfaccia indica solo dei metodi da implementare
  - consente l'uso di "altre classi" per l'elaborazione di dati
  - può facilmente essere integrata in un progetto sviluppato indipendentemente
  - è consentito implementare più interfacce con la stessa classe
- Una classe astratta fornisce più struttura
  - definisce alcune implementazioni di default
  - permette di definire delle variabili di istanza/statiche/final

- Non è errato usare entrambe in un progetto:
  - l'interfaccia definisce un supertipo per un aspetto comune a diversi concetti (e consente di scrivere codice comune per concetti eterogenei, ad es. DataSet)
  - una classe astratta fornisce una base comune all'implementazione di concetti specifici correlati (una classe può estendere una sola superclasse)

Osservazioni su uso ereditarietà

- Raggruppare comportamenti comuni tra classi diverse in una classe astratta
  - Es. Personale con sottoclassi Dipendente, Quadro, Manager.
- Fornire implementazioni alternative per uno stesso metodo (polimorfismo)
  - Es. calcolo paga per lavoratore a ore e lavoratore a stipendio fisso
- Raffinare l'implementazione di un metodo
  - Es. deposit di BankAccount e CheckingAccount
- Estendere i comportamenti di una classe esistente
  - Es. CollectibleCoin, MeasurableBankAccount.

Estensione e composizione

- Due modi fondamentali per costruire classi e riutilizzare codice
- Estensione:
  - Usa ereditarietà
  - Esprime una relazione di appartenenza della sottoclasse alla superclasse
- Composizione:
  - Aggrega oggetti di altre classi (definiti come variabili di istanza nella nuova classe)
  - Esprime una relazione di possesso (la classe composta ha istanze delle classi componenti)

Metodi e classi final

- Per impedire al programmatore di creare sottoclassi o di sovrascrivere certi metodi, si usa la parola chiave **`final`**
  - **`public final class String`**
    - questa classe non si può estendere
  - **`public final void mioMetodo(...)`**
    - questo metodo non si può sovrascrivere

Controllo di accesso a variabili, metodi e classi (specificatori di accesso)

Accessibile da	`public`	`protected`	`package`	`private`
Stessa Classe	Si	Si	Si	Si
Altra Classe (stesso package)	Si	Si	Si	No
Sottoclasse (altro package)	Si	Si	No	No
Altra Classe non sottoclasse (altro package)	Si	No	No	No

Accesso protetto: variabili d'istanza

- Nell'implementazione del metodo deposit in **`CheckingAccount`** dobbiamo accedere alla variabile balance della superclasse
- Possiamo dichiarare la variabile balance protetta

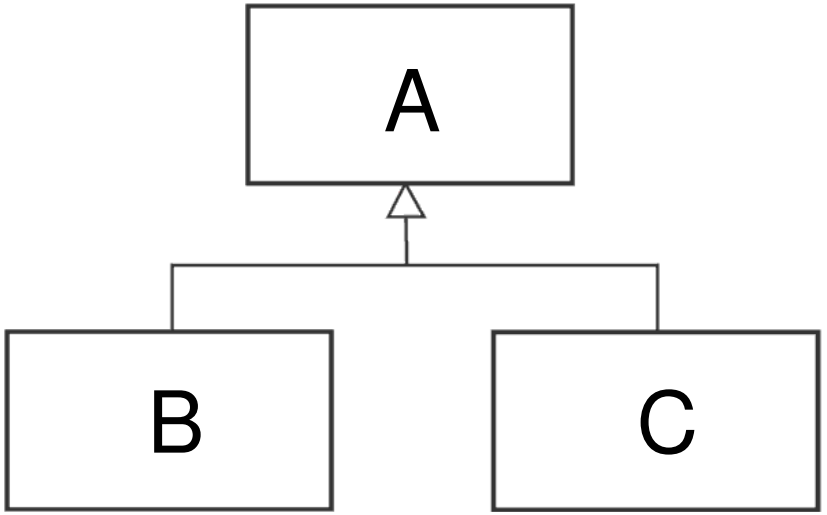
```
public class BankAccount{
    …
    protected double balance;
}
```

- Ai dati protected di un oggetto si può accedere dai metodi della classe e di tutte le sottoclassi
  - **`CheckingAccount`** è sottoclasse di **`BankAccount`** e può accedere a balance
  - Problema: la sottoclasse può avere metodi aggiuntivi che alterano i dati della superclasse

Accesso protetto: metodi

- Protected può essere usato per forzare l'uso di alcuni metodi solo da oggetti della stessa classe o di una sottoclasse
  - Si usa in genere come protezione per evitare un uso scorretto di alcuni metodi
  - Ad es. metodi che dipendono dalla conoscenza di dettagli di implementazione, come clone di Object (che vedremo in seguito)

Significato di protected



- foo è un metodo di A ereditato in B con specificatore protected
- C usa un'istanza di B e non è nello stesso pacchetto di B
- Possiamo invocare foo su questa istanza di B?
- NO, il metodo foo di B è visibile solo nelle sottoclassi di B e nel pacchetto di B

Ereditarietà e specificatori di accesso

- Quando si sovrascrivono i metodi di una superclasse non se ne può restringere la visibilità
  - Ad esempio: un metodo dichiarato protected può essere sovrascritto in una sottoclasse assegnando specificatore d'accesso protected o public ma non package o private.

`Object`: La classe universale

- Ogni classe che non estende un'altra classe, estende per default la classe **`Object`**
- Metodi della classe **`Object`**
  - **`String toString()`**
    - Restituisce una rappresentazione dell'oggetto in forma di stringa
  - **`boolean equals(Object otherObject)`**
    - Verifica se l'oggetto è uguale a un altro
  - **`Object clone()`**
    - Crea una copia dell'oggetto
- E' opportuno sovrascrivere questi metodi nelle nostre classi

Object: la classe universale

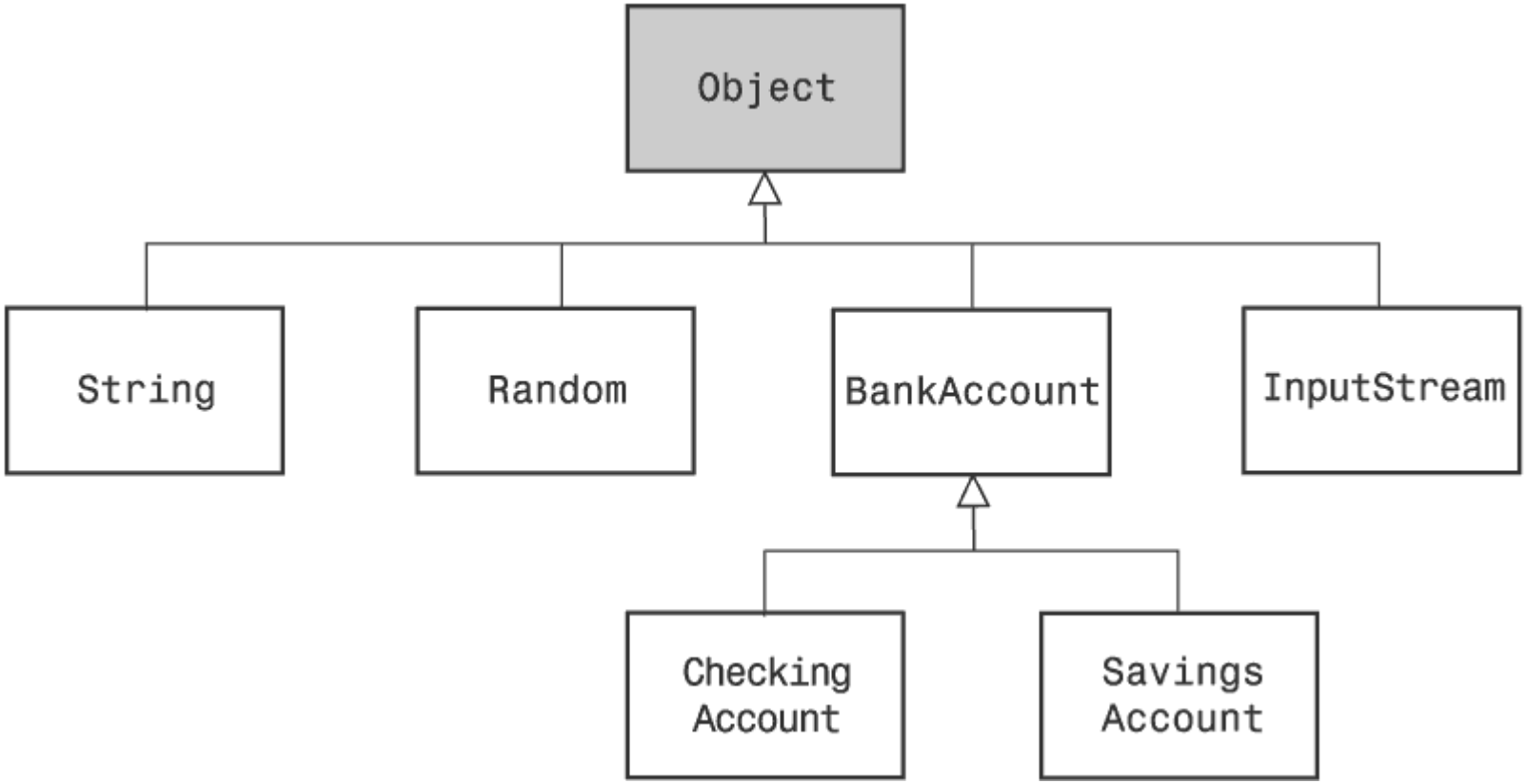


Figura 8 La classe Object è la superclasse di tutte le classi Java

Il metodo `toString`

- Restituisce una stringa contenente lo stato dell'oggetto in un formato specifico

```
Rectangle cerealBox = new Rectangle(5,10,20,30);
String s = cerealBox.toString();
```

s si riferisce alla stringa
"java.awt.Rectangle[x=5,y=10,width=20,height=30]"

- Il formato da rispettare è:

nomeClasse lista_variabili_istanza_e_valori

dove la lista ha il formato

[instVarName$_1$=val$_1$, …,instVarName$_n$=val$_n$]

- In presenza di ereditarietà

nomeClasse listeSuperclassi listaClasse

Conversione di oggetti a String

- toString può essere usato per convertire il parametro implicito in un oggetto String
- viene invocato automaticamente quando si concatena un oggetto di qualsiasi tipo con un oggetto di tipo String:

```
"cerealBox=" +cerealBox
```

viene valutata:

```
"cerealBox=java.awt.Rectangle[x=5,y=10,width=20,height=30]"
```

Conversione di oggetti a String

- toString() può essere invocato su qualsiasi oggetto in quanto ogni classe estende la classe Object

- la conversione automatica funziona solo se uno dei due oggetti è già di tipo String
  - se nessuno dei due oggetti è di tipo String il compilatore genera un errore

Sovrascrivere `toString`

- Proviamo a usare il metodo **`toString()`** nella classe **`BankAccount`**:

```
BankAccount momsSavings = new BankAccount(5000);
String s  = momsSavings.toString();
```

`s si riferisce a "BankAccount@d24606bf"`

- Viene stampato il nome della classe seguito dall'indirizzo in memoria dell'oggetto
- Siamo interessati al dato contenuto nell'oggetto!

Sovrascrivere `toString`

- Dobbiamo sovrascrivere il metodo nella classe **`BankAccount`**:

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

- In tal modo:

```
BankAccount momsSavings = new BankAccount(5000);
String s  = momsSavings.toString();
```

s si riferisce a "BankAccount[balance=5000]"

Sovrascrivere `toString`

- E' importante fornire il metodo **`toString()`** in tutte le classi!
  - Ci consente di controllare lo stato di un oggetto
  - Se x è un oggetto e abbiamo sovrascritto **`toString()`**, possiamo invocare **`System.out.println(x)`**
    - Il metodo **`println`** della classe **`PrintStream`** invoca **`x.toString()`**

Sovrascrivere `toString`

- E' preferibile non inserire il nome della classe, ma determinarlo dinamicamente con **`getClass().getName()`**
  - Il metodo **`getClass()`** (classe **`Object`**)

    consente di sapere il tipo esatto dell'oggetto a cui punta un riferimento.
  - restituisce un oggetto di tipo **`Class,`** da cui possiamo ottenere informazioni relative alla classe
    - **`Class c = e.getClass();`**
  - il metodo **`getName()`** della classe **`Class`** restituisce la stringa contenente il nome della classe

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

Sovrascrivere `toString`

- Ora possiamo invocare **`toString()`** anche su un oggetto della sottoclasse

```
SavingsAccount sa = new SavingsAccount(10);
System.out.println(sa);
```

```
stampa “SavingsAccount[balance=1000]”;
non stampa il contenuto di interestRate!
```

Sottoclassi: sovrascrivere `toString` riutilizzando codice superclasse

- Nella sottoclasse dobbiamo sovrascrivere **`toString()`** e aggiungere i valori delle vbl di istanza della sottoclasse

```
public class SavingsAccount extends BankAccount{
   public String toString(){
      return super.toString() +
                "[interestRate=" + interestRate + "]";
   }
   ……
}
```

Sottoclassi: sovrascrivere `toString`

- Vediamo la chiamata su un oggetto di tipo **`SavingsAccount`**:

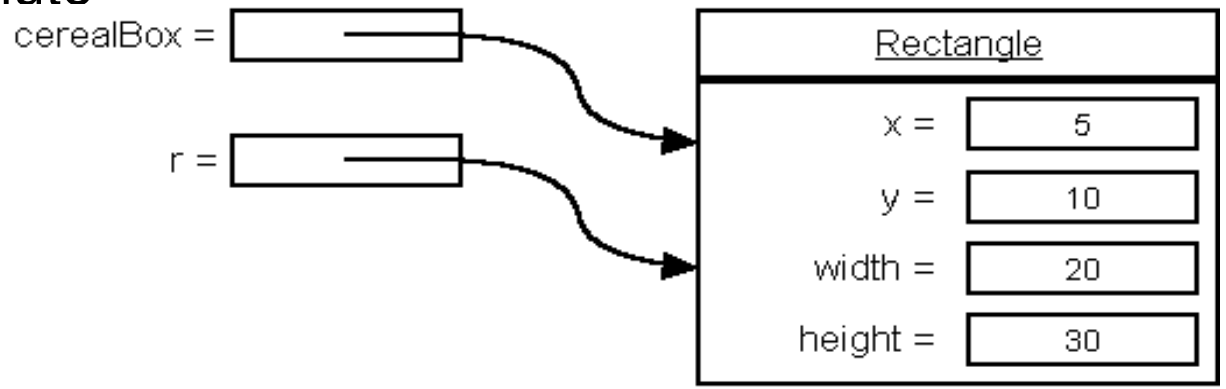
```
SavingsAccount sa = new SavingsAccount(10);
System.out.println(sa);
```

stampa

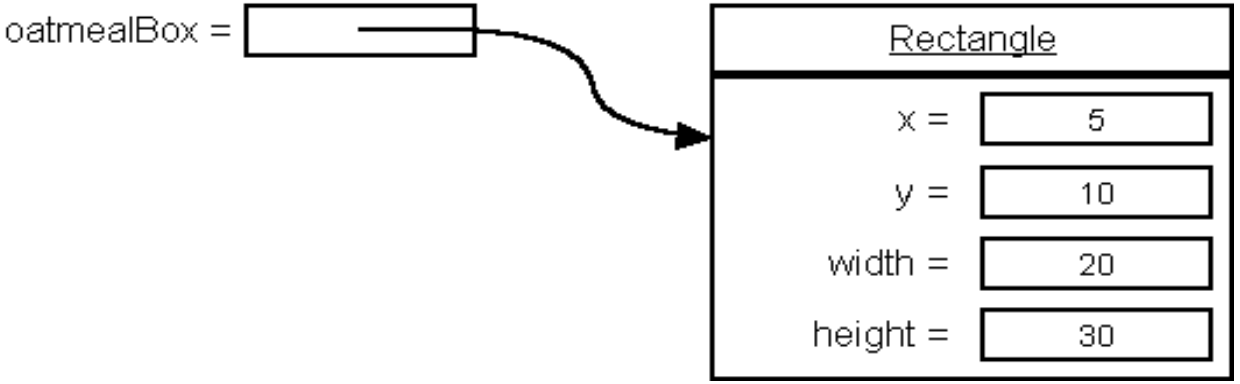
"SavingsAccount[balance=1000][interestRate=10]";

Sovrascrivere `equals`

- Il metodo equals verifica se due oggetti hanno lo stesso contenuto

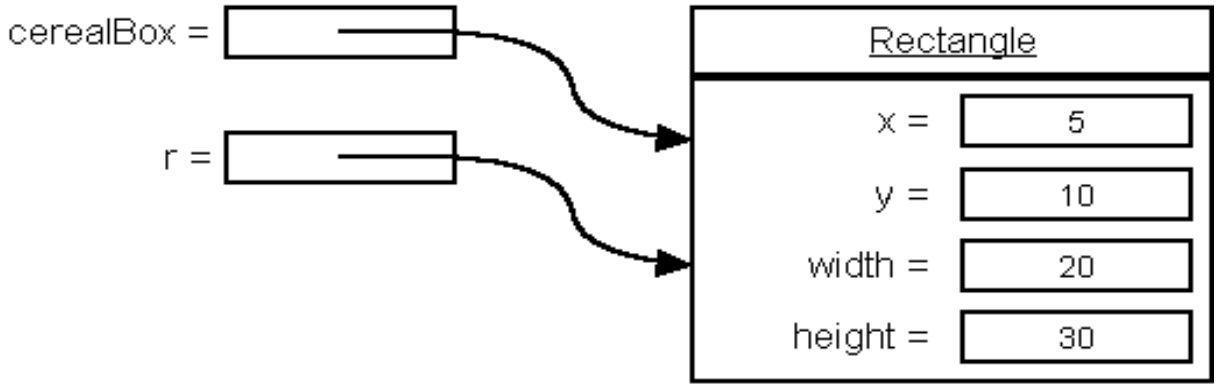



```
if (cerealBox.equals(oatmealBox))…
    restituisce true
```

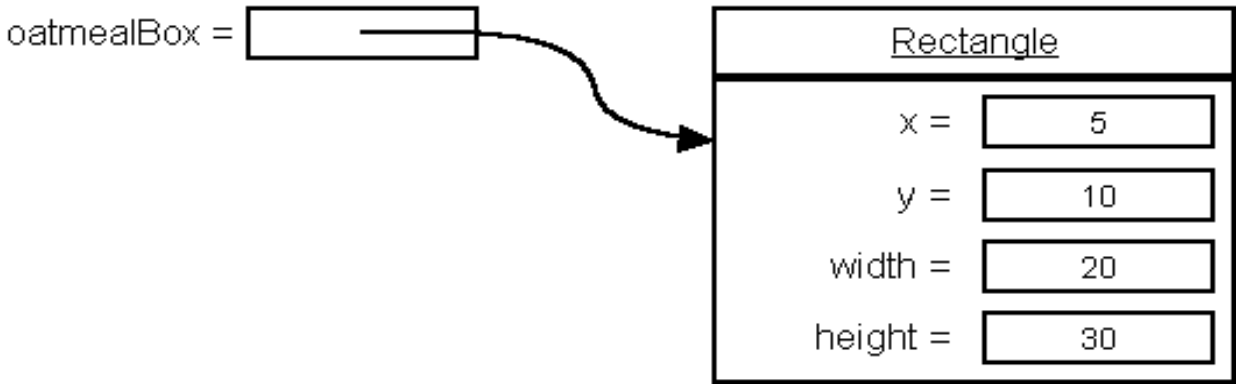


Sovrascrivere `equals`

- L'operatore == verifica se due riferimenti indicano lo stesso oggetto




```
if (cerealBox == oatmealBox)…
   viene valutato false
```



Sovrascrivere `equals`

```
boolean equals(Object otherObject){
        ..........
   }
```

- Sovrascriviamo il metodo equals nella classe **`Coin`**
  - il parametro otherObject è di tipo **`Object`** e non **`Coin`**
  - quando riscriviamo il metodo non possiamo variare la firma, ma dobbiamo eseguire un cast sul parametro

```
Coin other = (Coin)otherObject;
```

Sovrascrivere `equals`

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

- Controlla se hanno lo stesso nome e lo stesso valore
  - Per confrontare name e other.name usiamo equals perché si tratta di riferimenti a stringhe
  - Per confrontare value e other.value usiamo == perché si tratta di variabili numeriche

Sovrascrivere `equals`

- Se invochiamo **`coin1.equals(x)`** e **`x`** non è di tipo **`Coin`**?
  - Viene generata un'eccezione
- Possiamo usare **`instanceOf`** per controllare se **`x`** è di tipo **`Coin`**

```
public boolean equals(Object otherObject){
    if (otherObject instanceof Coin){
       Coin other = (Coin)otherObject;
       return name.equals(other.name)
       && value == other.value;
   }
   else return false;
}
```

Sovrascrivere `equals`

- Se si usa **`instanceOf`** per controllare se un oggetto è di un certo tipo, la risposta sarà **`true`** anche se l'oggetto appartiene a qualche sottoclasse…
- Dovrei verificare se i due oggetti appartengono alla stessa classe:

```
if (getClass() != otherObject.getClass()) return false;
```

- Infine, **`equals`** dovrebbe restituire false se otherObject è **`null`**

Classe Coin: Sovrascrivere `equals` in maniera robusta e riutilizzabile

```
public boolean equals(Object otherObject){
   if (otherObject == null) return false;
   if (getClass() != otherObject.getClass())
       return false;
   Coin other = (Coin)otherObject;
   return name.equals(other.name)
     && value == other.value;
}
```

Sottoclassi: Sovrascrivere `equals`

- Creiamo una sottoclasse di **`Coin`**: **`CollectibleCoin`**
  - Una moneta da collezione è caratterizzata dall'anno di emissione (vbl. istanza aggiuntiva)

```
public CollectibleCoin extends Coin{
 …

    private int year;
}
```

- Due monete da collezione sono uguali se hanno uguali nomi, valori e anni di emissione
  - Ma name e value sono variabili private della superclasse!
  - Il metodo equals della sottoclasse non può accedervi

Sottoclassi: Sovrascrivere `equals`

- Soluzione: il metodo equals della sottoclasse invoca il metodo omonimo della superclasse
  - Se il confronto ha successo, procede confrontando le altre vbl aggiuntive

```
public boolean equals(Object otherObject){
 if (!super.equals(otherObject)) return false;

 CollectibleCoin other =
                      (CollectibleCoin)otherObject;
 return year == other.year;
 }
```

Sovrascrivere `clone`

- Il metodo clone della classe **`Object`** crea un nuovo oggetto con lo stesso stato di un oggetto esistente (clone)

```
protected Object clone()
```

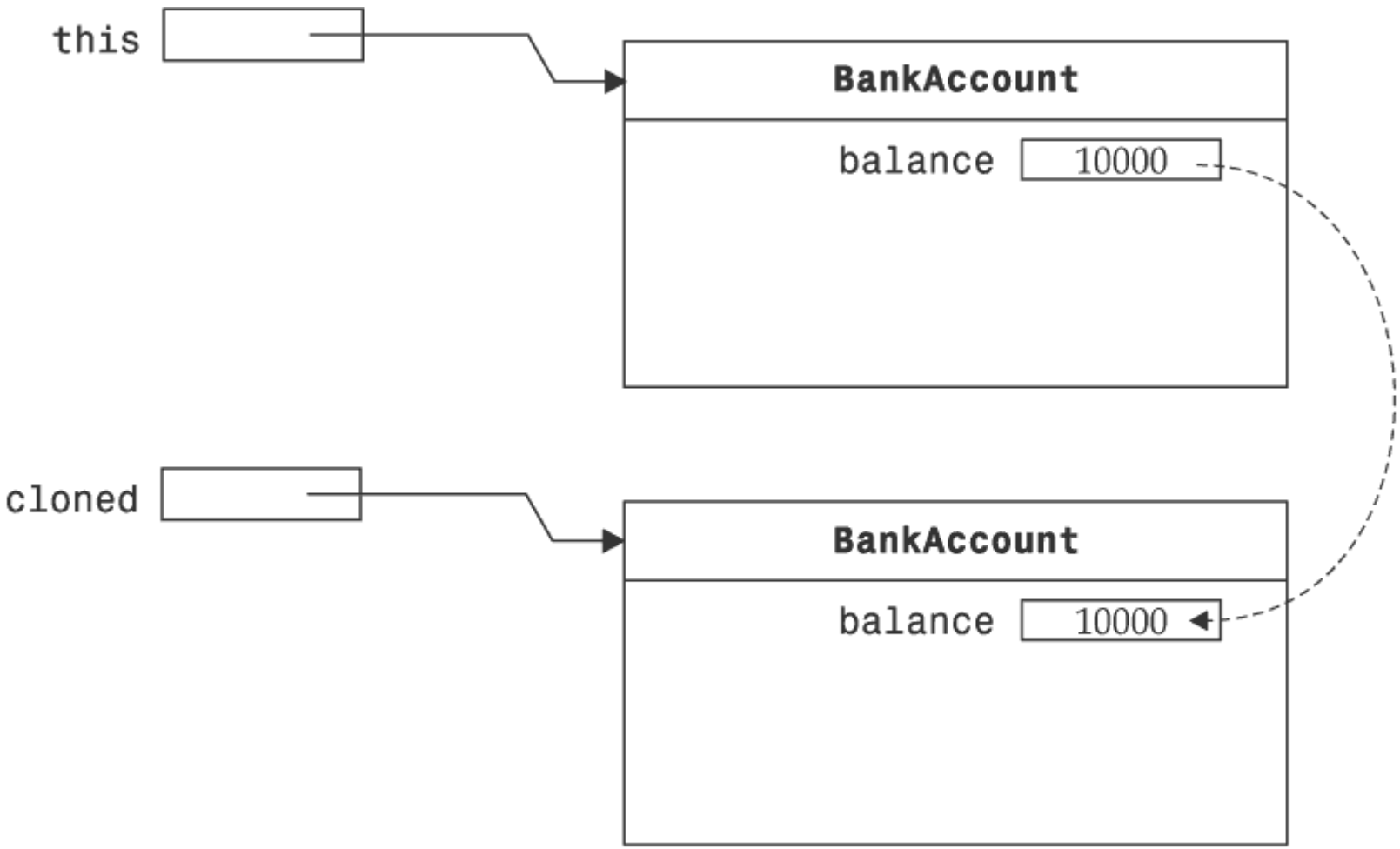
- Se x è l'oggetto che vogliamo clonare, allora
  - **`x.clone()`** e **`x`** sono oggetti con diversa identità
  - **`x.clone()`** e **`x`** hanno lo stesso contenuto
  - **`x.clone()`** e **`x`** sono istanze della stessa classe

Sovrascrivere `clone`

- Clonare un conto corrente

```
public BankAccount clone()
{
  BankAccount cloned= new BankAccount();
  cloned.balance = balance;
  return cloned;
 }
```

Nota: BankAccount è un sottotipo di Object quindi questa riscrittura è consentita (il tipo di return doveva essere Object prima di Java 5.0)


this
BankAccount
balance
10000
cloned
BankAccount
balance
10000

L'ereditarietà e il metodo clone

- Abbiamo visto come clonare un oggetto **BankAccount**

```
public BankAccount clone(){
   BankAccount cloned= new BankAccount();
   cloned.balance = balance;
   return cloned; }
```

- Problema: questo metodo non funziona per le sottoclassi!

```
SavingsAccount s= new SavingsAccount(0.5);
SavingsAccount clonedAccount = s.clone();
        //NON VA BENE
```

L'ereditarietà e il metodo clone

- Viene costruito un BankAccount e non un SavingsAccount!

- Possiamo invocare il metodo clone della classe `Object`
  - Crea un nuovo oggetto dello stesso tipo dell'oggetto originario
  - Copia il contenuto delle variabili di istanza dall'oggetto originario a quello clonato

L'ereditarietà e il metodo clone

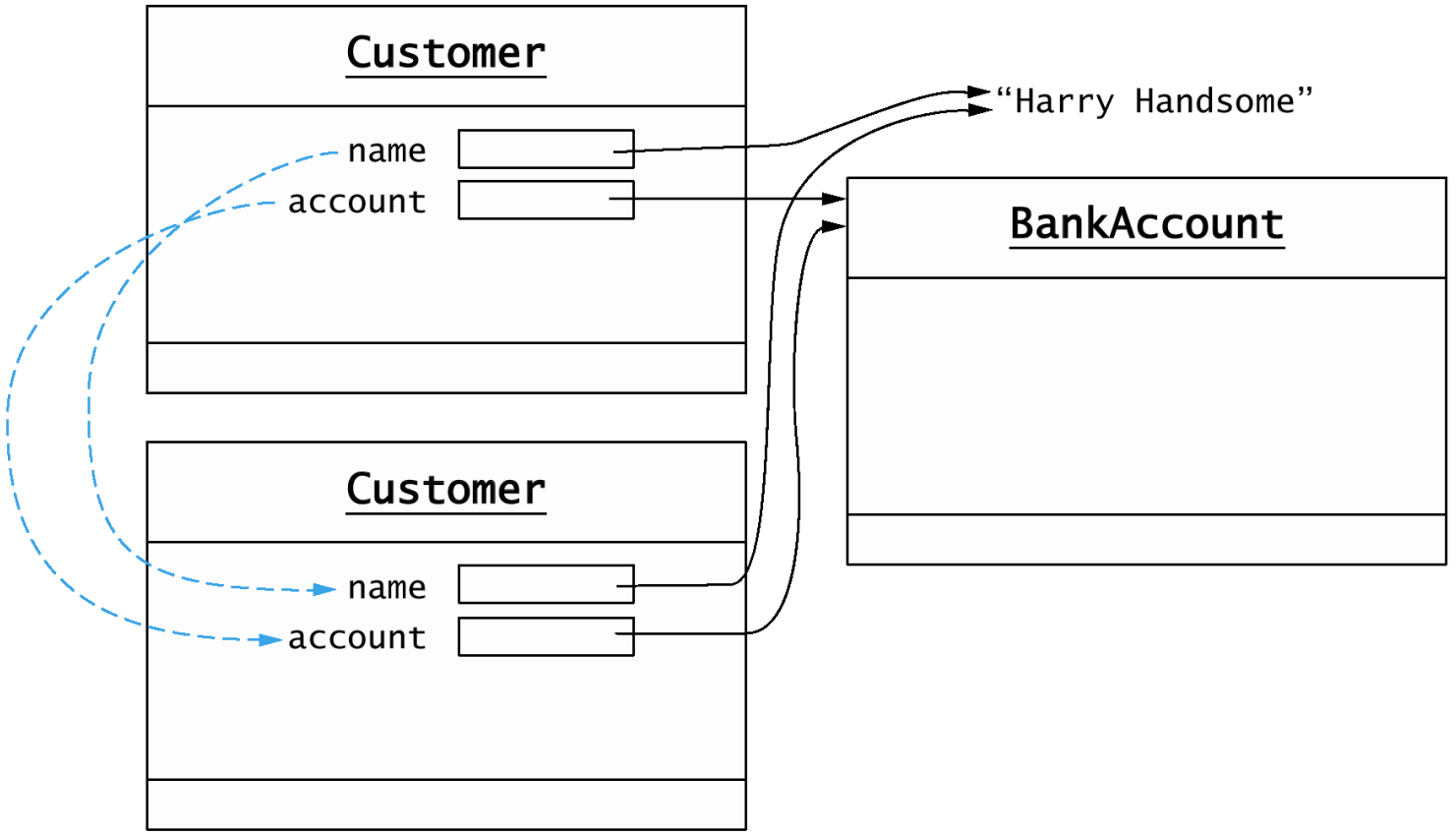
```
public class BankAccount{
…
  public BankAccount clone(){
  …
  //invoca il metodo Object.clone()
     BankAccount cloned =
                  (BankAccount) super.clone();
     return cloned;
  }
}
```

L'ereditarietà e il metodo clone

- Consideriamo una classe **`Customer`**
  - Un cliente è caratterizzato da un nome e un conto bancario
  - L'oggetto originale e il clone condividono un oggetto di tipo **`String`** e uno di tipo **`BankAccount`**
    - Nessun problema per il tipo **`String`** (oggetto immutabile)
    - Ma l'oggetto di tipo **`BankAccount`** potrebbe essere modificato da qualche metodo di **`Customer`**!
    - Andrebbe clonato anch'esso

L'ereditarietà e il metodo clone

- Problema: viene creata una copia superficiale
  - Se un oggetto contiene un riferimento ad un altro oggetto, viene creata una copia di riferimento all'oggetto, non un clone!



L'ereditarietà e il metodo clone

- Il metodo `Object.clone` si comporta bene se un oggetto contiene
  - Numeri, valori booleani, stringhe (e in generale oggetti immutabili)
- Bisogna però usarlo con cautela se l'oggetto contiene riferimenti ad altri oggetti
  - Quindi è inadeguato per la maggior parte delle classi!

L'ereditarietà e il metodo clone

- Precauzioni dei progettisti di Java:
  - Il metodo `Object.clone` è stato dichiarato protected
    - Non possiamo invocare `x.clone()` se non all'interno della classe dell'oggetto `x`, di una sua sottoclasse o del pachetto che la contiene
  - Una classe che voglia consentire di clonare i suoi oggetti deve implementare l'interfaccia `Cloneable`
    - In caso contrario viene lanciata un'eccezione di tipo `CloneNotSupportedException`
    - Tale eccezione va catturata anche se la classe implementa `Cloneable`
- In genere, quando sovrascriviamo clone lo ridefiniamo public così è possibile usarlo dovunque.

Interfacce di contrassegno

```
public interface Cloneable{
}
```

- Interfaccia di contrassegno
  - Non ha metodi
  - Usata solo per controlli come nel caso di Cloneable con clone

Clonare un BankAccount

```
public class BankAccount implements Cloneable
{
   …
   public BankAccount clone()
   {

       try
       {
          return (BankAccount) super.clone();
       }
       catch (CloneNotSupportedException e)
       {
          //non succede mai perchè implementiamo Cloneable
          return null;
       }
    }
}
```

Clonare CheckingAccount e SavingsAccount

- Basta ereditare metodo clone di BankAccount
  - interestRate e transactionCount sono di tipo primitivo
  - eventalmente si sovrascrive solo per fare il casting

- Non c'è bisogno di usare **`implements`** Cloneable

```
…… class SavingsAccount extends BankAccount {…}
…… class Checking Account extends BankAccount {…}
```

  - BankAccount già implementa Cloneable, e quindi anche le sue sottoclassi

Clonare un Customer

```
public class Customer implements Cloneable
{
   …
   public Customer clone()
   {
        try
        {
            Customer cloned = (Customer) super.clone();
            cloned.account = account.clone();
            return cloned;
        }
        catch (CloneNotSupportedException e)
        {
           //non succede mai perchè implementiamo Cloneable
            return null;
        }
    }
   private String name;
   private BankAccount account;
}
```

Clonare un PremiumCustomer

```
public class PremiumCustomer Extends Customer
{

//Customer che ha diritto a consegne a domicilio gratis


   ……
    public PremiumCustomer clone()
    {
           PremiumCustomer cloned =
                              (PremiumCustomer)super.clone();
           cloned.indirizzoConsegne =
                              indirizzoConsegne.clone();
           return cloned;
    }

    private Indirizzo indirizzoConsegne;
}
```

Violazione incapsulamento

- se il dato memorizzato attraverso una variabile di istanza diventa modificabile (in altre classi) senza utilizzare i metodi dell'interfaccia pubblica l'incapsulamento è violato
- per le variabili di istanza non primitive questo può avvenire anche se dichiarate private
  - assegnando direttamente una variabile di istanza con un parametro esplicito
  - restituendo il riferimento contenuto in una variabile di istanza
- si può rimediare utilizzando
  - oggetti immutabili
  - oppure la clonazione

Clonazione e incapsulamento

metodi della classe Customer

- ```
  public void setAccount(BankAccount anAccount){
      account = anAccount;   //violazione incapsulamento
  }
  ```
  - si può incapsulare il BankAccount memorizzato in account assegnando un clone di anAccount

    ```
    account = anAccount.clone();
    ```
- ```
  public BankAccount getAccount(){
      return account;    //violazione incapsulamento
  }
  ```
  - si può incapsulare il BankAccount memorizzato in account restituendo un suo clone

    ```
    return account.clone();
    ```

Clonazione e oggetti immutabili

- le variabili di istanza contenenti riferimenti ad oggetti immutabili non necessitano di essere clonate

- rispetto alla clonazione e ai problemi di violazione dell'incapsulamento visti

  gli oggetti immutabili sono assimilabili ai tipi primitivi

Nota su ArrayList

- ArrayList sovrascrive i metodi toString, equals e clone
- Il metodo clone esegue una copia superficiale
  - per clonare occorre forzare clonazione di ogni elemento dell'arrayList
- toString e equals funzionano come ci si aspetta