# Ereditarietà

# Ereditarietà

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

```
class SavingsAccount extends BankAccount
{  nuovi metodi
   nuove vbl istanza
}
```

- Tutti i metodi e le variabili istanza della classe `BankAccount` sono ereditati automaticamente

# Ereditarietà

- La classe più "vecchia" viene detta SUPERCLASSE e quella più giovane SOTTOCLASSE
  - `BankAccount`: superclasse
  - `SavingsAccount`: sottoclasse



# La madre di tutte le classi

- La classe `Object` è la radice 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()`

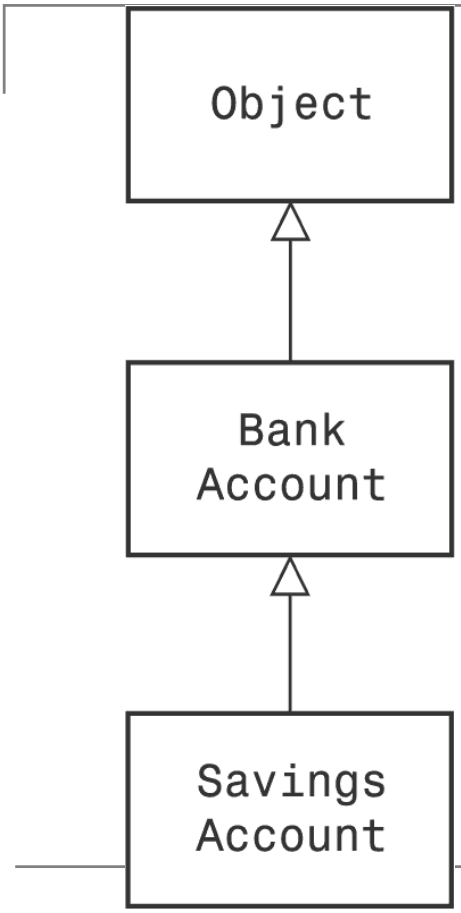



**Figura 1**
Diagramma di ereditarietà



# Ereditarietà vs Interfacce

- Differenza con l'implementazione di una interfaccia:
  - 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

# Riutilizzo di codice

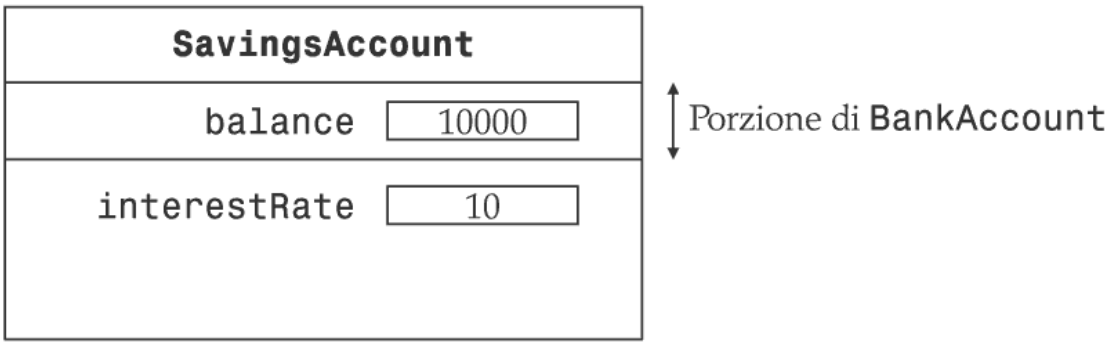
- La classe `SavingsAccount` eredita i metodi della classe `BankAccount`:
  - `withdraw`
  - `deposit`
  - `getBalance`
- Inoltre, `SavingsAccount` ha un metodo che calcola gli interessi maturati e li versa sul conto
  - `addInterest`



# 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;
  }
```

# Classe: SavingsAccount




`SavingsAccount` eredita la vbl istanza balance da `BankAccount` e hauna vbl istanza in più: il tasso di interesse



# Classe: SavingsAccount

- 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);
```

# Classe: SavingsAccount

- **`SavingsAccount sa = new SavingsAccount(10);`**

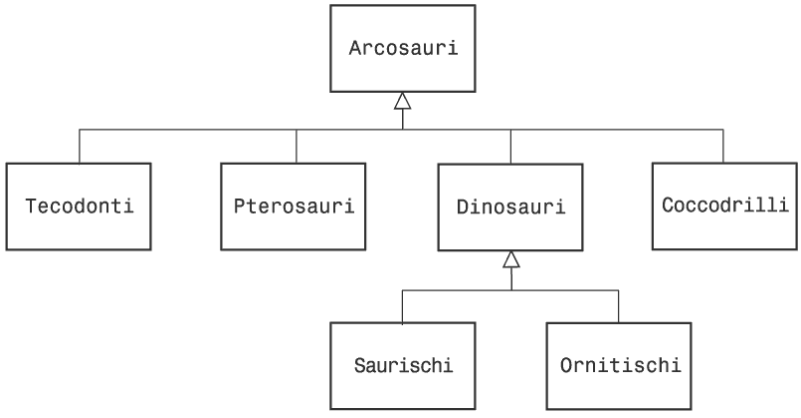
- **`sa.addInterest();`**
  - Viene usato il parametro implicito di `addInterest`

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

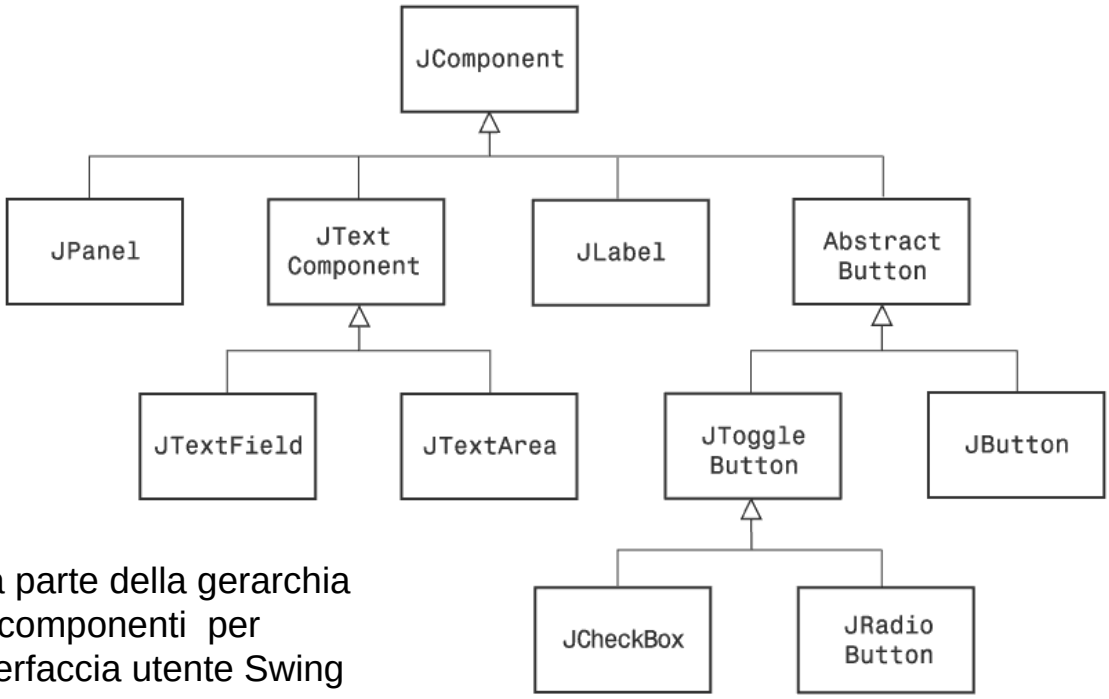


# Gerarchie di ereditarietà

- In Java le classi sono raggruppate in gerarchie di ereditarietà
  - Le classi che rappresentano concetti più generali sono più vicine alla radice
  - Le classi più specializzate sono nelle diramazioni



# Gerarchie di ereditarietà

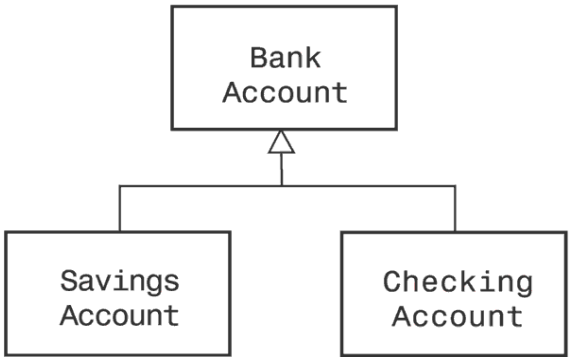



Una parte della gerarchia dei componenti per l'interfaccia utente Swing



# Gerarchie di 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



# Gerarchie di ereditarietà

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



# Metodi di una sottoclasse

Tre possibilità per definirli:

- Sovrascrivere 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 nessun metodo della superclasse
- Definire nuovi metodi
  - la sottoclasse definisce un metodo che non esiste nella superclasse

# Variabili istanza di sottoclassi

Due possibilità:

- Ereditare variabili istanza
  - Le sottoclassi ereditano tutte le vbl istanza della superclasse
- Definire nuove variabili istanza
  - Esistono solo negli oggetti della sottoclasse
  - Possono avere lo stesso nome di quelle nella superclasse, ma non sono sovrascritte
  - 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 vbl balance, ma è una vbl privata della superclasse!
- I metodi della sottoclasse non possono accedere alle vbl 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);
  }
```

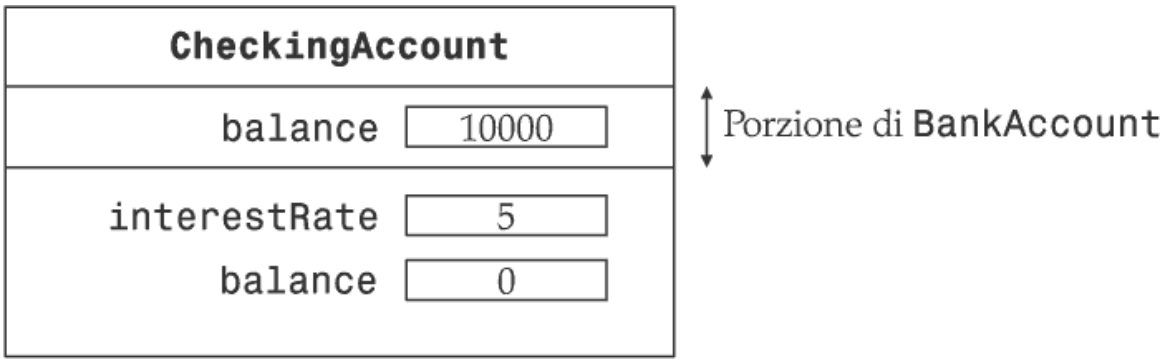


# CheckingAccount: metodo deductFees

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

# 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 vbl istanza con lo stesso nome
- La variabile della sottoclasse mette in ombra quella della superclasse






# Costruzione di 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 enunciato del costruttore della sottoclasse

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

# Costruzione di sottoclassi

- Se il costruttore della sottoclasse non chiama il costruttore della superclasse, viene invocato il costruttore predefinito della superclasse
  - Se il costruttore di `CheckingAccount` non invoca il costruttore di `BankAccount`, viene impostato il saldo iniziale a zero



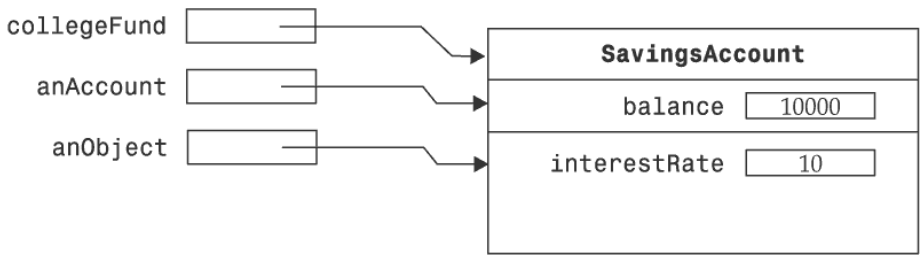
# Conversione da Sottoclasse a Superclasse

- Si può salvare un riferimento ad una sottoclasse in una vbl di riferimento ad 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

- Non si possono applicare metodi della sottoclasse:

```
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

- Vi ricordate il metodo `transfer`:

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

- Gli si può passare qualsiasi tipo di `BankAccount`

# Polimorfismo

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

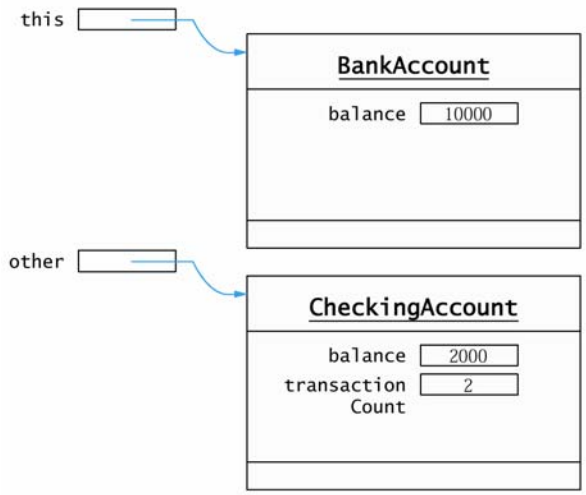
```
BankAccount momsAccount = . . .;
CheckingAccount harrysChecking = . . .;
momsAccount.transfer(harrysChecking, 1000);
```

- Il compilatore copia il riferimento all'oggetto harrisChecking di tipo sottoclasse nel riferimento di superclasse **`other`**



# Polimorfismo

- Il metodo transfer non sa che other si riferisce a un oggetto di tipo **`CheckingAccount`**
- Sa solo che other è un riferimento di tipo **`BankAccount`**



# Polimorfismo

- Il metodo transfer invoca il metodo `deposit`.
  - Quale metodo?
- Dipende dal tipo reale dell'oggetto!
  - Su un oggetto di tipo `CheckingAccount` viene invocato `CheckingAccount.deposit`
- Vediamo un programma che chiama i metodi polimorfici `withdraw` e `deposit`



# File AccountTest.java

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

      CheckingAccount 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
      momsSavings.addInterest();
      harrysChecking.deductFees();

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

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



# 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;
}
```



# Classi astratte

- Un ibrido tra classe ed interfaccia
- Ha alcuni metodi normalmente implementati ed alcuni altri *astratti*
  - Un metodo astratto non ha implementazione

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

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

# Classi astratte

- Attenzione: 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
  - In tal modo evitiamo che possano essere costruiti oggetti di quella classe
- In generale, sono astratte le classi di cui non si possono creare esemplari
- Le classi non astratte sono dette concrete

# 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`**
    - non si può estendere tale classe
  - **`public final void mioMetodo(...)`**
    - non si può sovrascrivere tale metodo



# Controllo di accesso a variabili, metodi e classi

| Accessibile da | `public` | `package` | `private` | `protected` |
|---|---|---|---|---|
| Stessa Classe | Si | Si | Si | Si |
| Altra Classe (stesso package) | Si | Si | No | Si |
| Altra Classe (package diverso) | Si | No | No | No |

# Accesso protetto

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

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

- Ai dati protetti 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



# `Object`: La classe universale

- Ogni classe che non estende un'altra classe, estende per default la classe di libreria `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

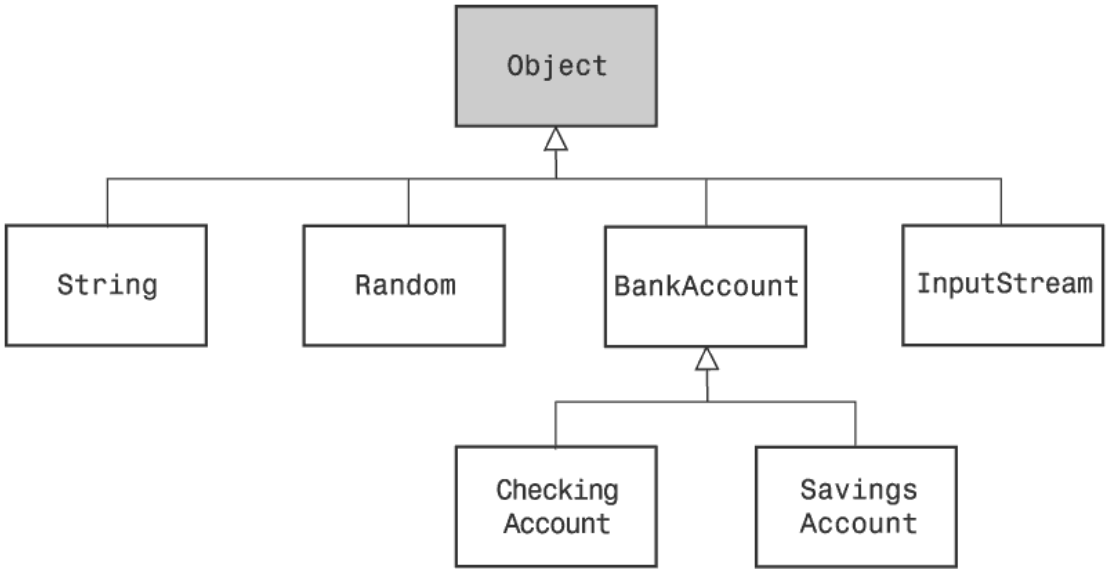



**Figura 8** La classe Object è la superclasse di tutte le classi Java



# Sovrascrivere `toString`

- Restituisce una stringa contenente lo stato dell'oggetto.

```
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]"
```

- Automaticamente invocato quando si concatena una stringa con un oggetto:

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

# Sovrascrivere `toString`

- L'operazione vista prima funziona solo se uno dei due oggetti è già una stringa
  - Il compilatore può invocare `toString()` su qualsiasi oggetto, dato che ogni classe estende la classe `Object`
- Se nessuno dei due oggetti è una stringa 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
- Ma noi volevamo sapere cosa si trova nell'oggetto!
  - Il metodo `toString()` della classe `Object` non può sapere cosa si trova all'interno della classe `BankAccount`

# 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 `getClass().getName()`
  - Il metodo `getClass()` consente di sapere il tipo esatto dell'oggetto a cui punta un riferimento.
  - Restituisce un oggetto do tipo `Class`, da cui possiamo ottenere molte informazioni relative alla classe
    - `Class c = e.getClass()`
  - Ad esempio, il metodo `getName()` della classe `Class` restituisce la stringa contentente 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 anche il contenuto di addInterest!
```

## Sottoclassi: sovrascrivere `toString`

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

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



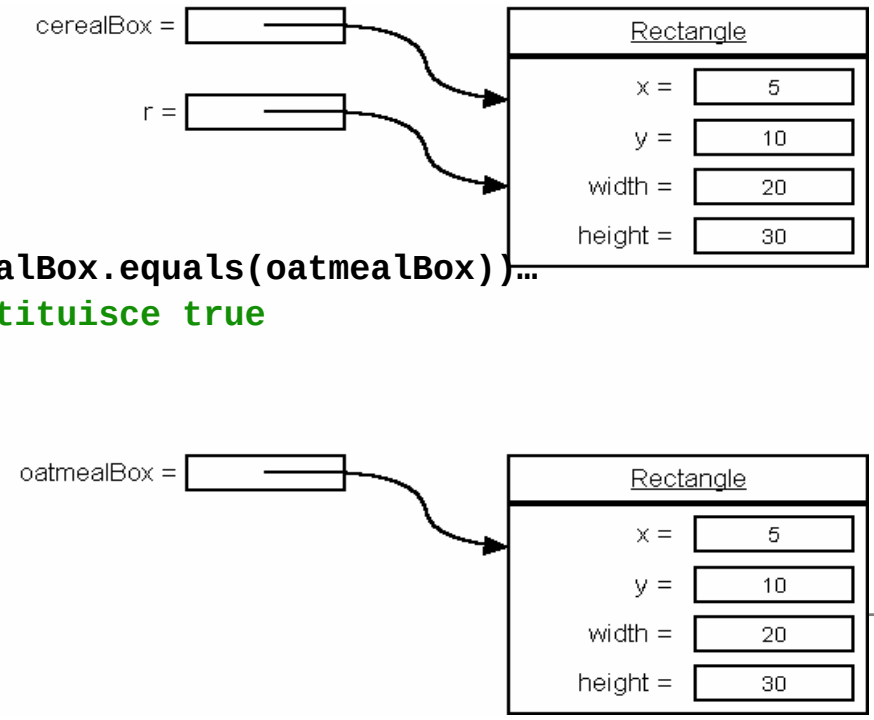
## Sottoclassi: sovrascrivere `toString`

- 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

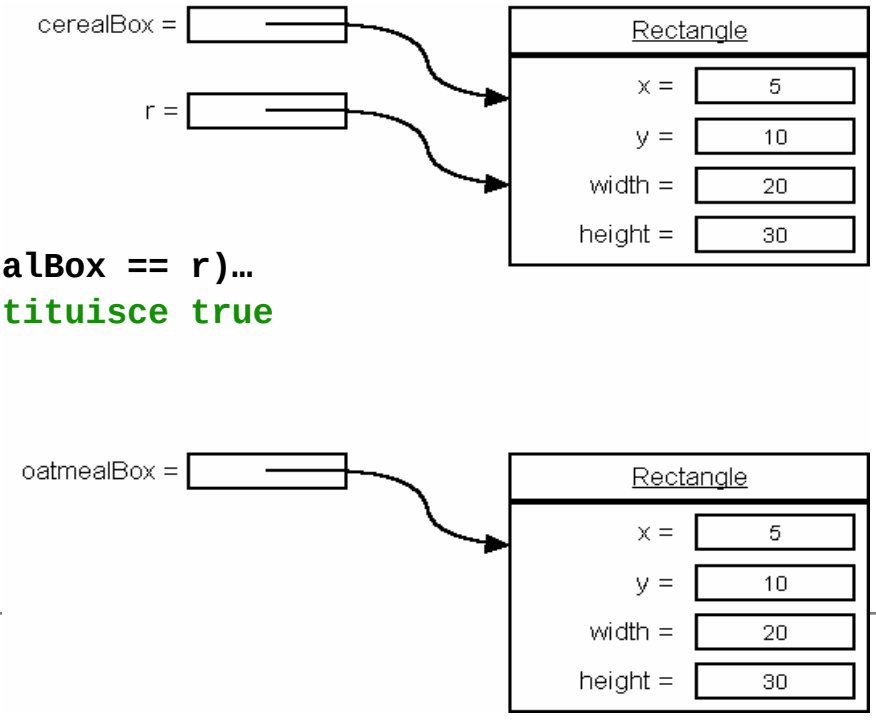



```
if (cerealBox.equals(oatmealBox))…
   //restituisce true
```



# Sovrascrivere equals

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




```
if (cerealBox == r)…
   //restituisce true
```

# Sovrascrivere `equals`

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

- Sovrascriviamo il metodo equals nella classe `Coin`
  - Problema: il parametro otherObject è di tipo `Object` e non `Coin`
  - Se riscriviamo il metodo non possiamo variare la firma, ma dobbiamo eseguire un cast sul parametro
  - `Coin other = (Coin)otherObject;`



# Sovrascrivere `equals`

- Ora possiamo confrontare le monete:

```
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 vbl numeriche

# Sovrascrivere `equals`

- Se invochiamo `coin1.equals(x)` e `x` non è di tipo `Coin`?
  - Il cast errato eseguito in seguito genera 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 uso **`instanceOf`** per controllare se una classe è di un certo tipo, la risposta sarà true anche se l'oggetto appartiene a qualche sottoclasse…
- Dovrei verificare se i due oggetti appartengano alla stessa classe:

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

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

# Classe Coin: Sovrascrivere `equals`

```
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 vbl 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 (copia profonda o clone)
  - `public 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 Object clone()
{
 BankAccount cloned= new BankAccount();
 cloned.balance = balance;
 return cloned;
 }
```



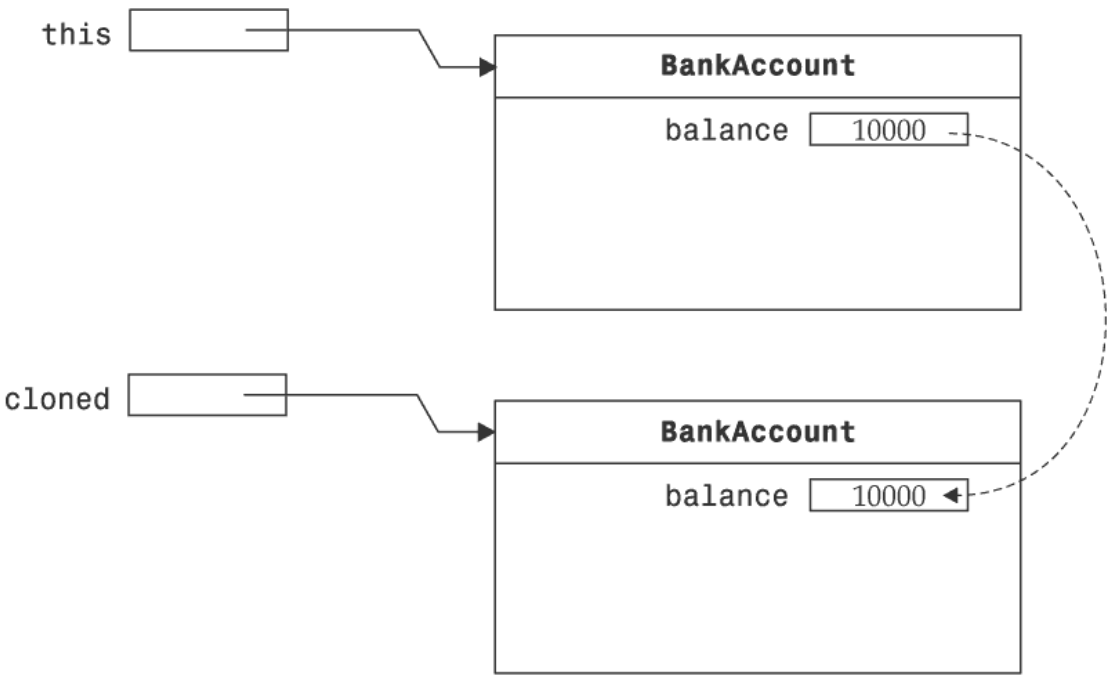



**Figura 11** Clonazione di oggetti

# Sovrascrivere `clone`

- Il tipo restituito dal metodo clone è `Object`
- Se invochiamo il metodo dobbiamo usare un cast per convincere il compilatore che `account1.clone()` ha lo stesso tipo di `account2`:

```
BankAccount account1 = . . .;
BankAccount account2 =
    (BankAccount)account1.clone();
```



# L'ereditarietà e il metodo clone

- Abbiamo visto come clonare un oggetto `BankAccount`

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

- Problema: questo metodo non funziona nelle sottoclassi!

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

# L'ereditarietà e il metodo clone

- Viene costruito un conto bancario e non un conto di risparmio!
  - `SavingsAccount` ha una variabile aggiuntiva, che non viene considerata!
- Possiamo invocare il metodo clone della classe `Object`
  - Crea un nuovo oggetto dello stesso tipo dell'oggetto originario
  - Copia le variabili istanza dall'oggetto originario a quello clonato

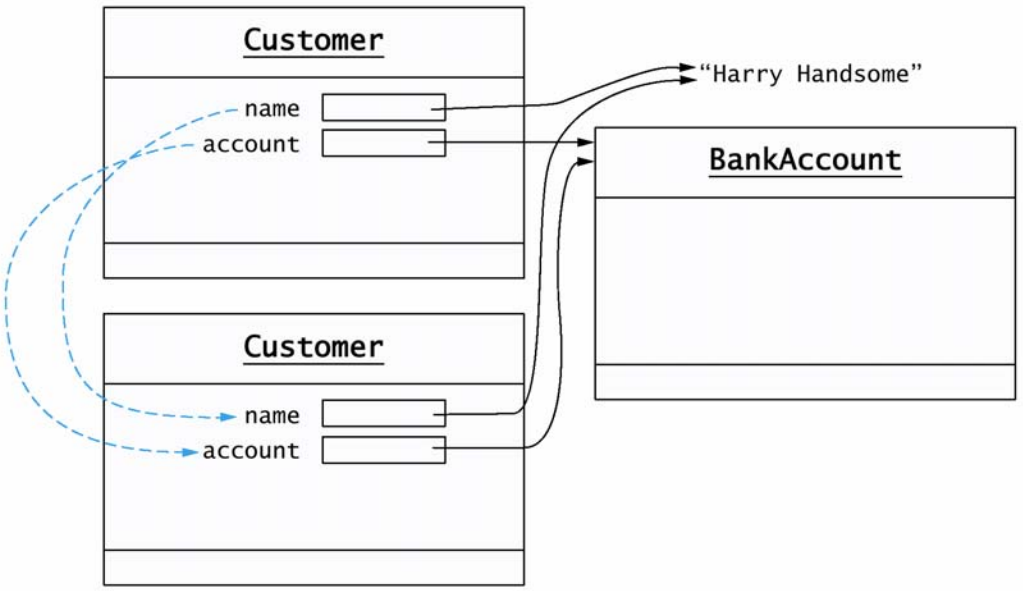


# L'ereditarietà e il metodo clone

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

# 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

- Consideriamo una classe `Customer`
  - Un cliente è caratterizzato da un nome e un conto corrente
  - 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

- Il metodo `Object.clone` si comporta bene se un oggetto contiene
  - Numeri, valori booleani, stringhe
- 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 protetto
    - Non possiamo invocare `x.clone()` se la classe a cui appartiene `x` non ridefinisce clone come metodo pubblico
  - 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`

# L'interfaccia Cloneable

```
public interface Cloneable{
}
```

- Interfaccia contrassegno
  - Non ha metodi
  - Usata solo per verificare se un'altra classe la realizza
  - Se l'oggetto da clonare non è un esemplare di una classe che la realizza viene lanciata l'eccezione



# Clonare un conto corrente

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

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

# Clonare un cliente

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