Ereditarietà

L'ereditarietà

- L'ereditarietà è lo strumento concettuale che supporta l'obiettivo di ***riusabilità del codice nell'OOP***
  - si usa quando si deve realizzare una classe ed è già disponibile un'altra classe che rappresenta ***un concetto più generale***
  - La nuova classe da scrivere è più **specializzata**, **eredita** comportamenti (metodi) della classe più generale e ne **aggiunge di nuovi**

L'ereditarietà

- Supponiamo di voler realizzare una classe **SavingsAccount** per rappresentare un ***conto bancario di risparmio***
  - un tasso di interesse annuo determinato al momento dell'apertura
  - un metodo **addInterest** per accreditare gli interessi sul conto

- un ***conto bancario di risparmio*** ha ***tutte le stesse*** caratteristiche di un ***conto bancario***, ***più alcune altre*** caratteristiche che gli sono peculiari
  - riutilizziamo il codice già scritto per la classe **BankAccount**

Ricordiamo BankAccount

```
public class BankAccount
{  private double balance;

   public BankAccount()
   {  balance = 0;
   }
   public BankAccount(double initialBalance)
   {  balance = initialBalance;
   }
   public void deposit(double amount)
   {  balance = balance + amount;
   }
   // versione semplificata, senza controllo
   public void withdraw(double amount)
   {  balance = balance - amount;
   }
   public double getBalance()
   {  return balance;
   }
}
```

Progettiamo SavingsAccount da zero

```
public class SavingsAccount{

   private double balance; // in rosso il codice copiato

   public SavingsAccount(double rate){
      balance = 0; interestRate = rate;
   }
   // per brevità, non mettiamo altri costruttori
   public void addInterest(){
      balance += balance * interestRate / 100;
   }
   private double interestRate; // nuova variabile

   public void deposit(double amount)
   {  balance = balance + amount;   }

   public void withdraw(double amount)
   {  balance = balance - amount;   }

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

Riutilizzo del codice

- Come previsto, abbiamo potuto ***copiare*** nella definizione della classe **SavingsAccount** buona parte del codice scritto per **BankAccount**
- Inoltre ci sono tre cambiamenti:
  - abbiamo aggiunto una variabile di esemplare
    - **interestRate**
  - abbiamo modificato il costruttore
    - ***ovviamente il costruttore ha anche cambiato nome***
  - abbiamo aggiunto un metodo
    - **addInterest**

Riutilizzo del codice

- Cosa succede se modifichiamo **BankAccount**?
  - ad esempio, modifichiamo **withdraw** in modo da impedire che il saldo diventi negativo
- Dobbiamo modificare di conseguenza anche **SavingsAccount**
  - molto scomodo… e se avessimo creato decine di tipi di conti bancari?
  - fonte di errori…
- **L'ereditarietà risolve questi problemi**

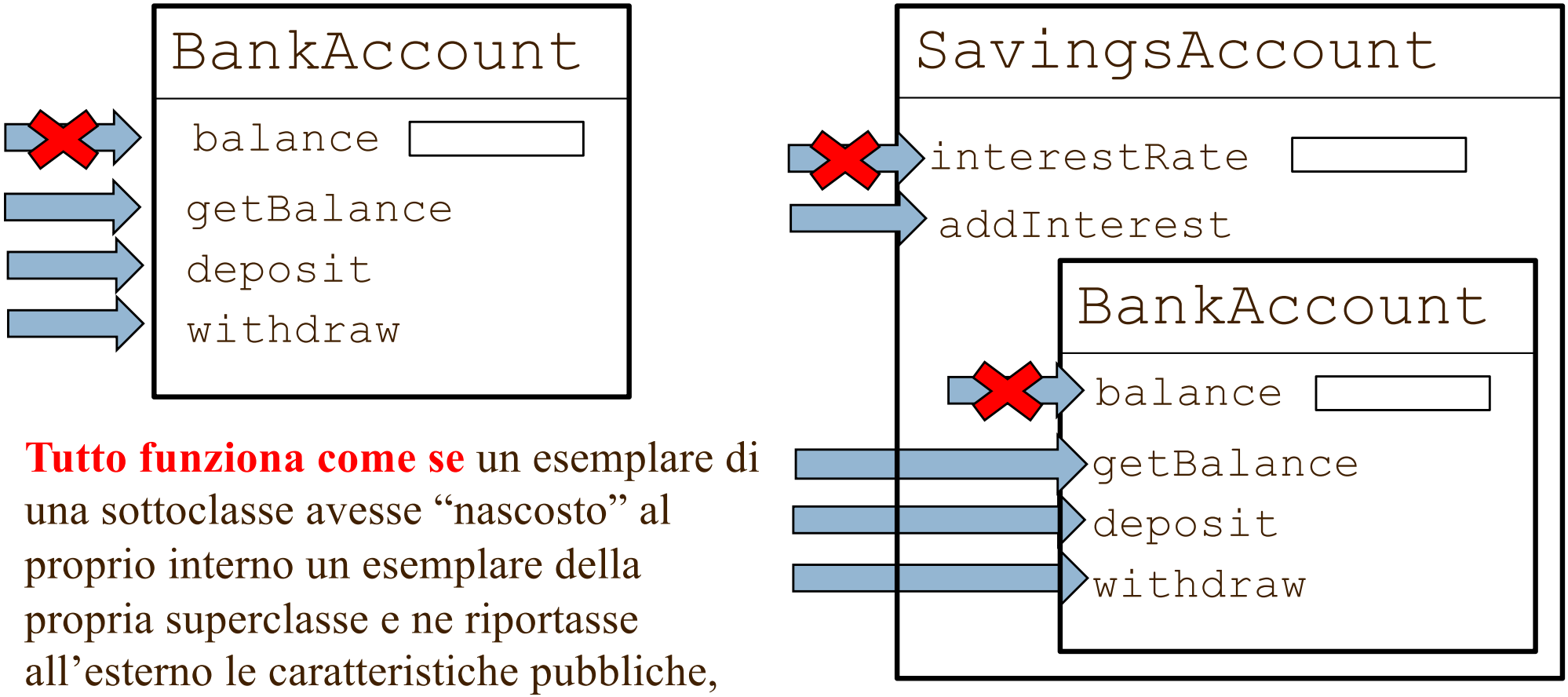
SavingsAccount

DIPARTIMENTO

```
public class SavingsAccount extends BankAccount
{
    private double interestRate;

    public SavingsAccount(double rate)
    {  interestRate = rate;
    } // balance viene implicitamente inizializzata a 0
      // si vedrà meglio più avanti...
   public void addInterest()
    {  deposit(getBalance() * interestRate / 100);
    } // qui non potrei usare direttamente balance, è privata...

}
```

- Con la clausola **extends** dichiariamo che **SavingsAccount** è una classe ***derivata*** da **BankAccount**, nel senso che ne ***estende*** le funzionalità
  - ne **eredita** tutte le caratteristiche
  - specifichiamo soltanto le **peculiarità**

SavingsAccount



Tutto funziona come se un esemplare di una sottoclasse avesse "nascosto" al proprio interno un esemplare della propria superclasse e ne riportasse all'esterno le caratteristiche pubbliche, oltre a poterle utilizzare

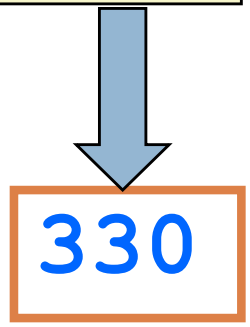
= accesso pubblico

= accesso privato

Come usare la classe derivata

- Esemplari della classe derivata **SavingsAccount** si usano **come se** fossero esemplari di **BankAccount**, con ***qualche proprietà in più***

```
SavingsAccount account = new SavingsAccount(10);
account.deposit(500);
account.withdraw(200);
account.addInterest();
System.out.println(account.getBalance());
```



- La classe derivata si chiama
  - ***sottoclasse***
- La classe da cui questa deriva si chiama
  - ***superclasse***

La superclasse universale Object

- In Java, ogni classe che non deriva ***esplicitamente*** (cioè mediante la clausola **`extends`**) da un'altra classe deriva ***implicitamente*** dalla ***superclasse universale***: **Object**
  - Massimo dell'astrazione
- Quindi, **SavingsAccount** deriva da **BankAccount**, che a sua volta deriva da **Object**
- **Object** ha (pochissimi) metodi (tra cui toString) che, quindi, sono ereditati da tutte le classi in Java
  - **l'ereditarietà avviene anche su più livelli**, quindi **SavingsAccount** eredita anche le proprietà di **Object**

Ereditarietà

- Sintassi:

```
class NomeSottoclasse extends NomeSuperclasse
{   costruttori
    nuovi metodi
    nuove variabili
}
```

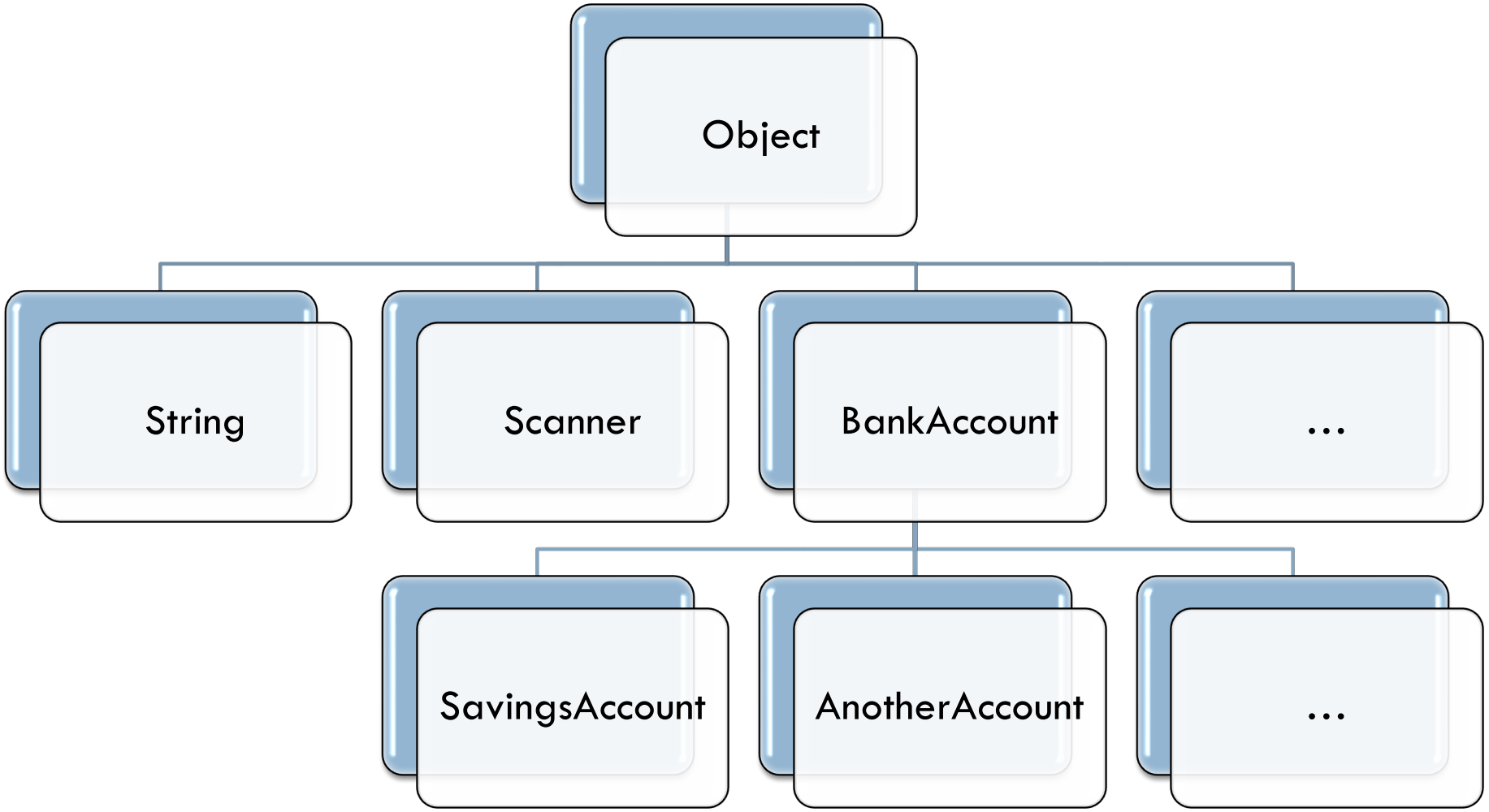
- Scopo: definire la classe ***NomeSottoclasse*** che deriva dalla classe ***NomeSuperclasse***, definendo ***nuovi metodi*** e/o ***nuove variabili***, oltre ai suoi ***costruttori***
- Nota: se la superclasse non è indicata esplicitamente, il compilatore usa implicitamente `java.lang.Object`

Gerarchia di ereditarietà

- **In Java, ciascuna classe ne estende un'altra e soltanto una**
  - tranne **Object**, che non estende nessuna classe

- Di conseguenza, in Java la struttura gerarchica dell'ereditarietà è un ***albero***, la cui ***radice*** è la superclasse universale **Object**

Gerarchia di ereditarietà



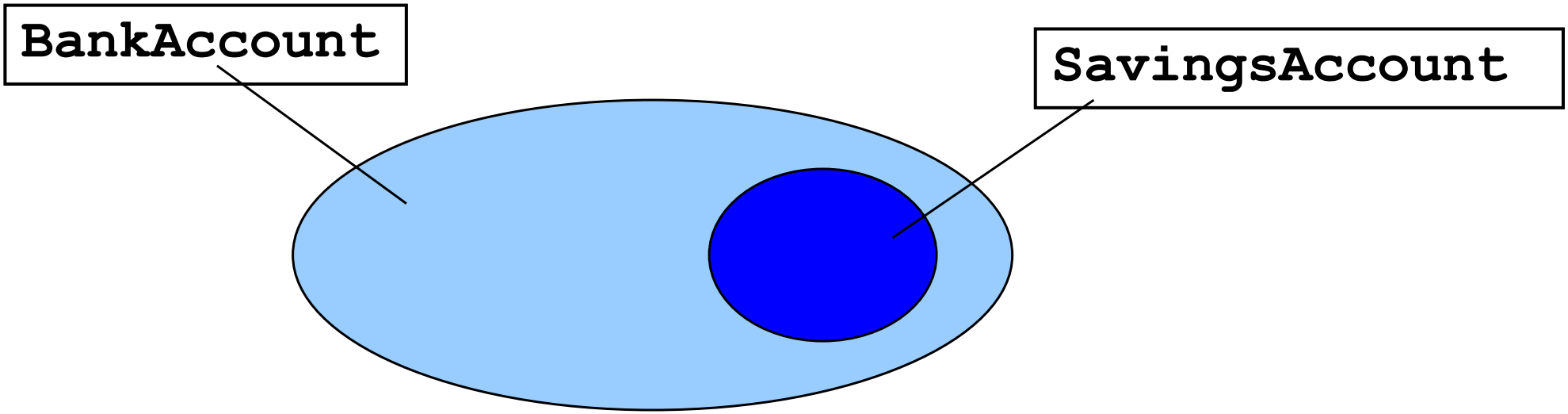
Terminologia e notazione

Confondere superclassi e sottoclassi

- Dato che oggetti di tipo **SavingsAccount** sono
  - un'***estensione*** di oggetti di tipo **BankAccount**
  - più "grandi" di oggetti di tipo **BankAccount**, perché hanno una variabile di esemplare in più
  - più "abili" di oggetti di tipo **BankAccount**, perché hanno un metodo in più
- perché mai **SavingsAccount** si chiama ***sottoclasse*** e non ***superclasse***?
  - è facile fare confusione
  - verrebbe forse spontaneo usare i nomi al contrario...

Confondere superclassi e sottoclassi

- I termini superclasse e sottoclasse derivano dalla ***teoria degli insiemi***
  - I conti bancari di risparmio (gli oggetti di tipo **SavingsAccount**) costituiscono un ***sottoinsieme*** dell'insieme di tutti i conti bancari (gli oggetti di tipo **BankAccount**) e quest'ultimo insieme è, quindi, un ***superinsieme*** di tale sottoinsieme



I metodi della sottoclasse

Metodi di una sottoclasse

- Quando si definisce una sottoclasse, per quanto riguarda i suoi metodi possono succedere tre cose:
  - nella sottoclasse viene definito un metodo che nella superclasse non esisteva (**addInterest**)
  - un metodo della superclasse viene ereditato dalla sottoclasse (**deposit, withdraw, getBalance**)
  - ***un metodo della superclasse viene sovrascritto nella sottoclasse***

Sovrascrivere un metodo

- La possibilità di ***sovrascrivere*** (***override***) un metodo della superclasse, **modificandone il comportamento (soltanto) quando è usato per la sottoclasse**, è una delle caratteristiche più potenti di OOP
- Per sovrascrivere un metodo bisogna definire nella sottoclasse un metodo ***con la stessa firma*** (cioè stesso nome e stesso elenco di tipi dei parametri) di quello definito nella superclasse
  - tale metodo ***prevale*** (***overrides***) su quello ereditato dalla superclasse quando viene invocato con un oggetto della sottoclasse
  - da non confondere con metodi **sovraccarichi**, che hanno lo stesso nome ma parametri diversi!

Sovrascrivere un metodo: Esempio

- Vogliamo modificare la classe **SavingsAccount** introducendo un contatore delle sole operazioni di versamento (e metodi che consentano di ispezionare e azzerare tale contatore)

```
public class SavingsAccount extends BankAccount
{  ...
   private int count = 0; // azzerato qui per brevità
   public int getCount() { return count; }
   public void resetCount() { count = 0; }
}
```

- I versamenti nei conti di tipo **SavingsAccount** si fanno però invocando il metodo **deposit** di **BankAccount**, sul quale non abbiamo controllo

Sovrascrivere un metodo: Esempio

- Possiamo però ***sovrascrivere*** **deposit**, ***ridefinendolo*** in **SavingsAccount**

```
public class SavingsAccount extends BankAccount
{   ...
    public void deposit(double amount)
    {   count++;
        ... // aggiungi amount a balance
    }
}
```

- In questo modo, quando viene invocato **deposit** con un oggetto di tipo **SavingsAccount**, viene effettivamente invocato il metodo **deposit** definito nella classe **SavingsAccount** e non quello definito in **BankAccount**
  - Ovviamente ***nulla cambia*** per oggetti di tipo **BankAccount**

Sovrascrivere un metodo: Esempio

- Proviamo a completare il metodo
  - dobbiamo versare **amount** nel conto, cioè sommarlo a **balance**
  - non possiamo modificare direttamente **balance**, che è una variabile privata in **BankAccount**
  - sappiamo anche che l'unico modo per aggiungere una somma di denaro a **balance** è l'invocazione del metodo **deposit**

```
public class SavingsAccount extends BankAccount
{   ...
    public void deposit(double amount)
    {   count++;
        deposit(amount); // NON FUNZIONA
    }
}
```

- Così però non funziona, perché il metodo invoca se stesso, cioè diventa ricorsivo (con ricorsione infinita!!!)

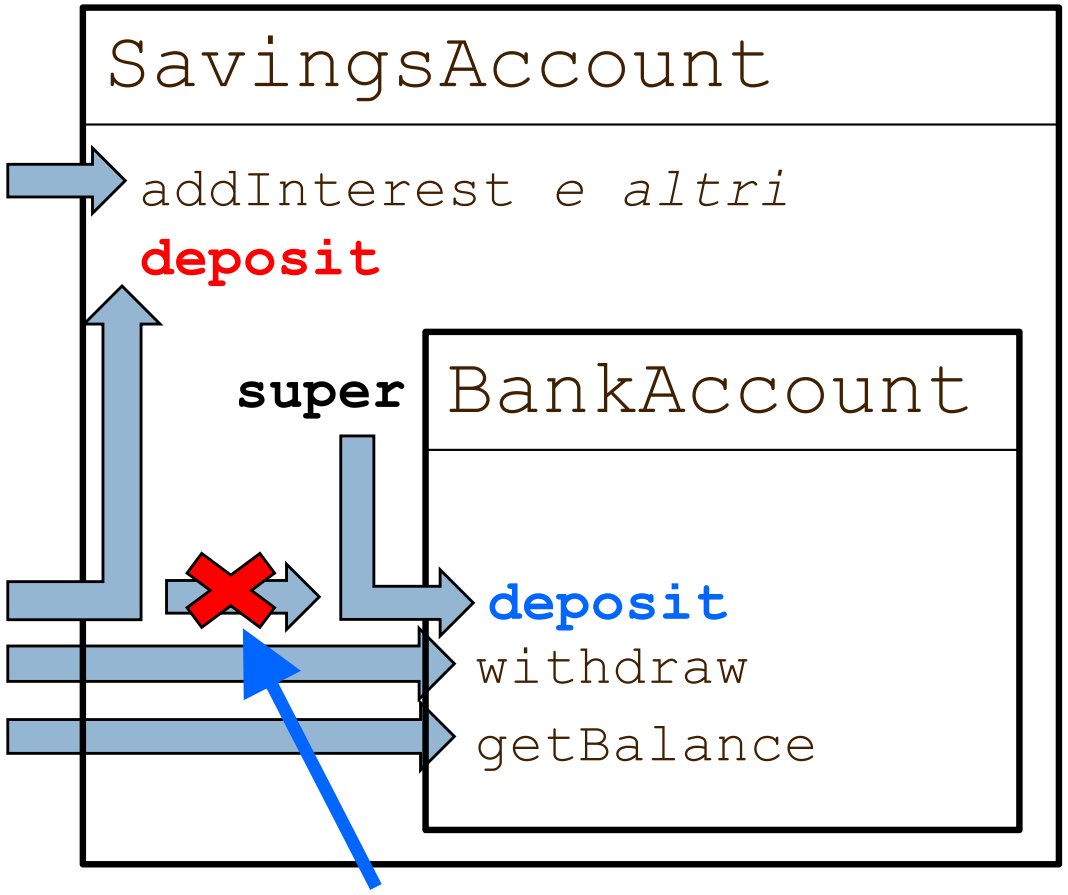
Sovrascrivere un metodo: Esempio

- Ciò che dobbiamo fare è ***invocare il metodo* deposit *di* BankAccount**, quello che stiamo sovrascrivendo!
- Questo si può fare usando il riferimento implicito **super**, gestito automaticamente dal compilatore per accedere agli elementi ereditati dalla superclasse

```
public class SavingsAccount extends BankAccount
{   ...
    public void deposit(double amount)
    {   count++;
        // invoca deposit della superclasse
        super.deposit(amount);
    }
}
```

Sovrascrivere un metodo

La figura rappresenta, dal punto di vista funzionale ciò che succede per effetto della sovrascrittura (o ridefinizione) del metodo **deposit** nella classe **SavingsAccount**



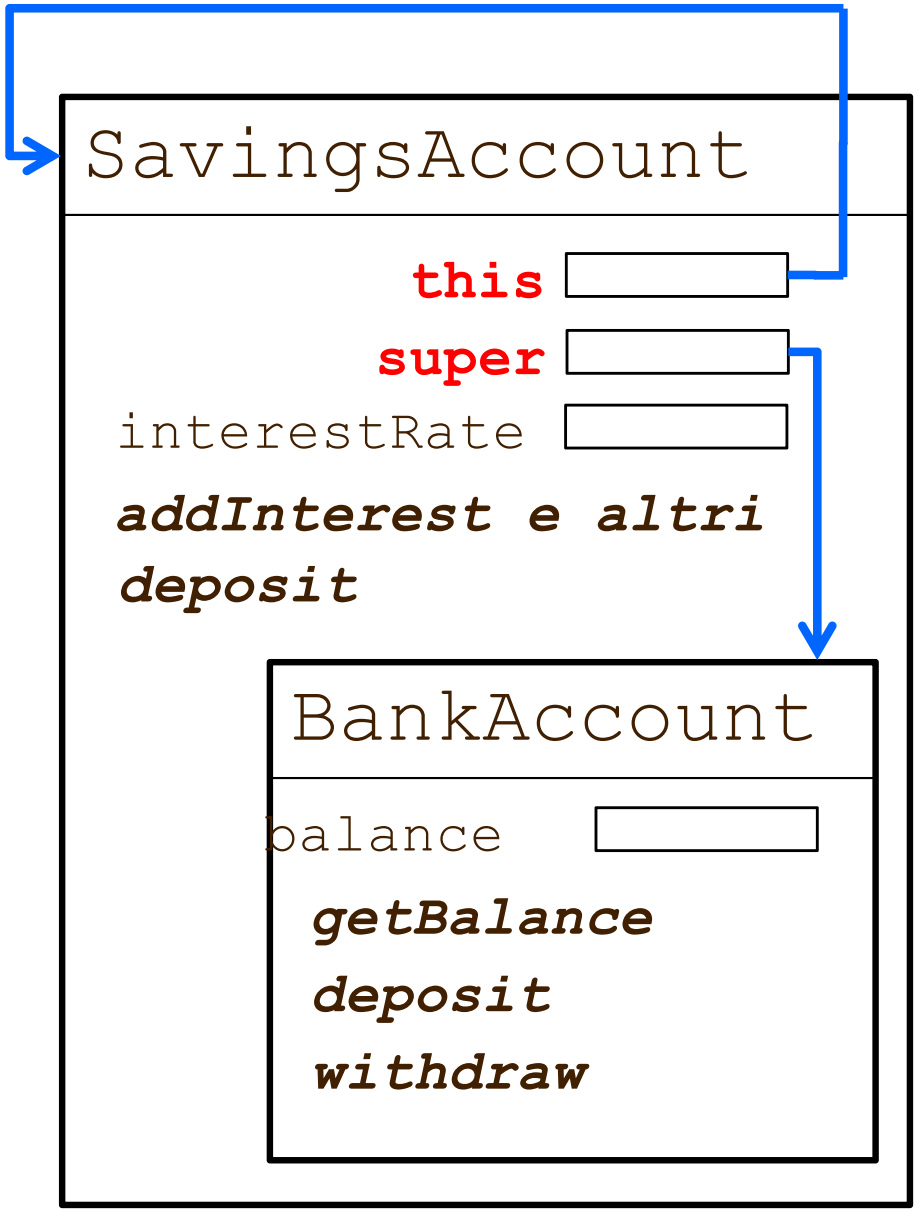
Il metodo **deposit** ereditato rimane invocabile soltanto da metodi della classe **SavingsAccount** tramite il riferimento **super**: non è più accessibile dall'esterno

Accesso diretto inibito dalla sovrascrittura di **deposit** nella sottoclasse

Il riferimento implicito `super`

All'interno di un oggetto, **ogni metodo di esemplare**, oltre alle (eventuali) variabili di esemplare e alle (eventuali) variabili statiche, ha a disposizione i due riferimenti **`this`** e **`super`**

(anche negli oggetti di tipo **`BankAccount`**, come in ogni altro oggetto, esistono i due riferimenti **`this`** e **`super`**, anche se qui non sono stati raffigurati)



Invocare un metodo della superclasse

- Sintassi:

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

- Scopo: invocare il metodo ***nomeMetodo*** della superclasse anziché il metodo con lo stesso nome (sovrascritto) della classe corrente
- Si potrebbe usare **super** per ogni invocazione di un metodo della superclasse
  - In pratica lo si usa **SOLO** quando c'è ambiguità, cioè quando il metodo da invocare è sovrascritto

Costruttori della sottoclasse

Costruttori della sottoclasse

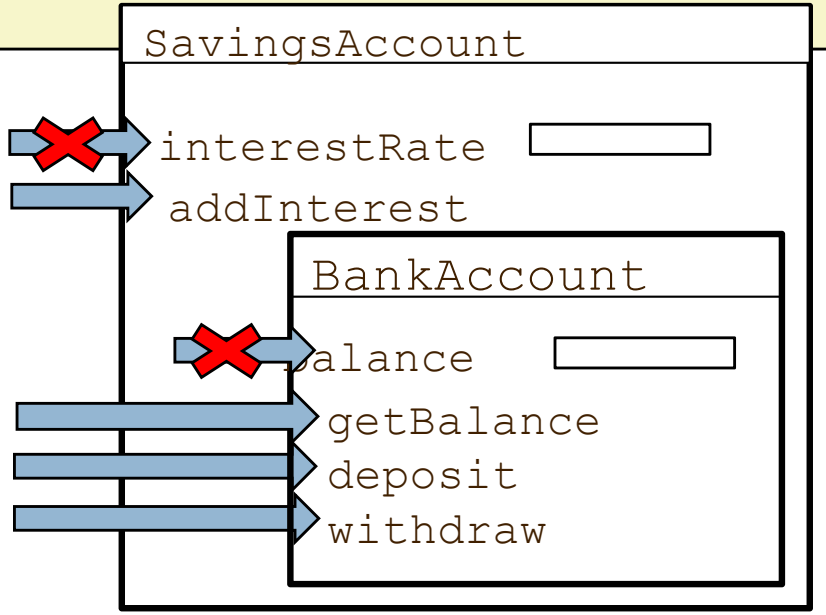
```
public class SavingsAccount extends BankAccount
{
   private double interestRate;

   public SavingsAccount(double rate)
   {  interestRate = rate;
   } // balance implicitamente inizializzata a 0

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

- Proviamo a definire un secondo costruttore in **SavingsAccount**, per ***aprire un conto corrente di risparmio con saldo iniziale diverso da zero***
  - analogamente a **BankAccount**, che ha due costruttori

Costruttori della sottoclasse

```
public class SavingsAccount extends BankAccount
{  public SavingsAccount(double rate)
   {  interestRate = rate;
   }
   public SavingsAccount(double rate,double initialAmount)
   {  interestRate = rate;
      balance = initialAmount; // NON FUNZIONA
   }
   ...
}
```

- Sappiamo già che questo approccio non può funzionare, perché **balance** è una variabile **private** di **BankAccount**
  - Viene segnalato un errore in fase di compilazione



Costruttori della sottoclasse

```
public SavingsAccount(double rate, double initialAmount)
{  interestRate = rate;
   deposit(initialAmount); // POCO ELEGANTE
}
```

- Si potrebbe risolvere il problema simulando un primo versamento, invocando **deposit**
- Questo è lecito, ma non è una soluzione molto buona, perché introduce potenziali effetti collaterali
  - ad esempio, le operazioni di versamento potrebbero avere un costo, dedotto automaticamente dal saldo, mentre il "versamento di apertura conto" potrebbe essere gratuito

Costruttori della sottoclasse

- Dato che la variabile **balance** è gestita dalla classe **BankAccount**, è molto meglio ***delegare*** a tale classe il compito di inizializzarla
- La classe **BankAccount** ha già un costruttore creato appositamente per gestire l'apertura di un conto con saldo diverso da zero
  - invochiamo tale costruttore dall'interno del costruttore di **SavingsAccount**, usando la sintassi **super(...)**

```
public SavingsAccount(double rate, double initialAmount)
{   super(initialAmount);
    interestRate = rate;
}
```

Costruttori della sottoclasse

- In realtà, un'invocazione di **super** viene **sempre** eseguita quando la JVM costruisce un esemplare di una sottoclasse
  - se la chiamata a **super(…)** non è indicata esplicitamente dal programmatore, il compilatore esegue automaticamente l'invocazione di **super( )** ***senza parametri***
    - questo avviene, ad esempio, nel caso del primo costruttore definito in **SavingsAccount**

```
public SavingsAccount(double rate)
{  // invocazione implicita di super();
   // che inizializza balance a zero
   interestRate = rate;
}
```

Invocare un costruttore di superclasse

- Sintassi:

```
NomeSottoclasse(parametri)
{   super(eventualiParametri);
    ...
}
```

- Scopo: invocare il costruttore della superclasse di ***NomeSottoclasse*** passando ***eventualiParametri*** (che possono essere anche diversi dai ***parametri*** del costruttore della sottoclasse)
- Nota: per motivi "tecnici", **se esplicitamente presente, deve essere il primo enunciato del costruttore**
- Nota: se non è indicato esplicitamente, viene invocato implicitamente **super( )** senza parametri

Attenzione al costruttore predefinito

- Il costruttore predefinito (senza argomenti) esiste implicitamente SOLTANTO se nella classe non vengono definiti dei costruttori
  - Altrimenti deve essere stato scritto esplicitamente

```
public class BankAccount2
{  public BankAccount2(double initialBalance)
   {...}
   ... // nessun altro costruttore
} // non esiste il costruttore predefinito!!
```

```
public class SavingsAccount extends BankAccount2
{  /* senza costruttori: c'è il costruttore
      predefinito, che invoca implicitamente
      super(), ma non esiste un costruttore senza
      parametri in BankAccount2!!
   */
   ...
}
```

Il compilatore segnala errore

Attenzione al costruttore predefinito

- Il costruttore predefinito (senza argomenti) esiste implicitamente SOLTANTO se nella classe non vengono definiti dei costruttori
  - Altrimenti deve essere stato scritto esplicitamente

```
public class BankAccount2
{  public BankAccount2(double initialBalance)
   {...}
   ... // nessun altro costruttore
} // non esiste il costruttore predefinito!!
```

```
public class SavingsAccount extends BankAccount2
{  public SavingsAccount(double rate)
   {  interestRate = rate;
   } // invoca implicitamente super()
     // ma non esiste un costruttore senza
     // parametri in BankAccount2!!
   ...
}
```

Il compilatore segnala errore

Attenzione al costruttore predefinito

```
public class BankAccount2
{  public BankAccount2(double initialBalance)
   {...}
   ... // nessun altro costruttore
} // non esiste il costruttore predefinito!!
```

- In tutte le classi derivate da questa
  - ci deve essere almeno un costruttore esplicito
  - ogni costruttore deve iniziare con
    - ***super(espressioneDiTipoDouble)***

Conversione fra riferimenti di superclasse e sottoclasse (up-casting)

Conversione fra riferimenti

- Un oggetto di tipo **SavingsAccount** è un ***caso speciale*** di oggetti di tipo **BankAccount**
- Questa proprietà si riflette in **una proprietà sintattica** del linguaggio Java
  - ***un riferimento a un oggetto di una classe derivata può essere assegnato a una variabile oggetto del tipo di una sua superclasse***, diretta o indiretta

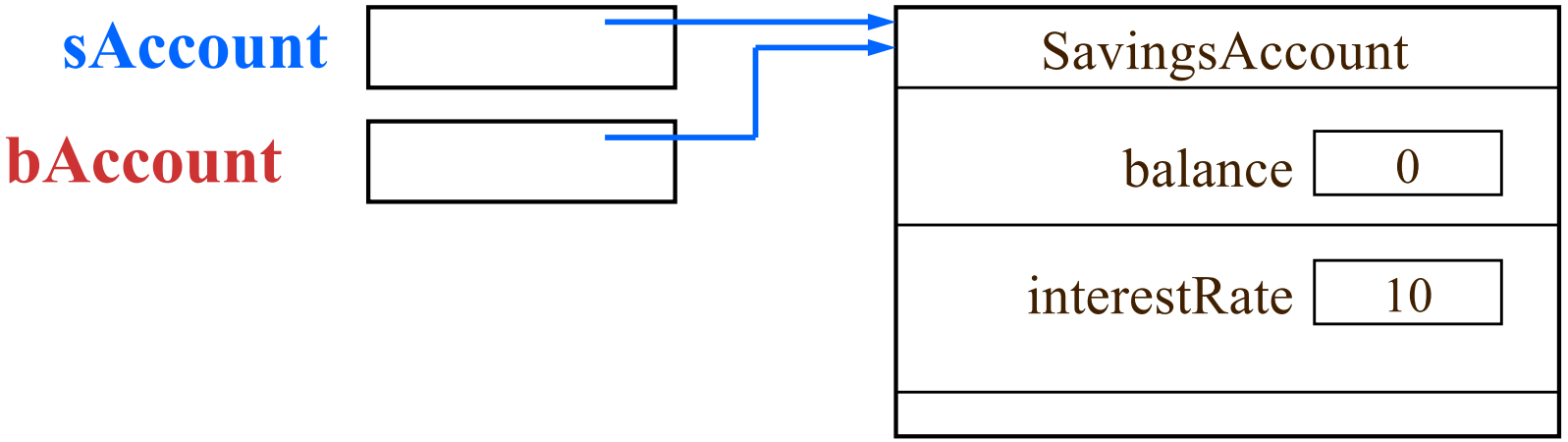
```
SavingsAccount sAccount = new SavingsAccount(10);
BankAccount bAccount = sAccount;
// si potrebbe anche fare Object obj = sAccount;
```

  - **non c'è nessuna "conversione" effettiva,** cioè non vengono modificati i dati, che sono indirizzi… e rimangono indirizzi!

Conversione fra riferimenti

```
SavingsAccount sAccount = new SavingsAccount(10);
BankAccount bAccount = sAccount;
```

- Le due variabili, ***di tipi diversi***, puntano ora ***allo stesso oggetto***



Conversione fra riferimenti

```
SavingsAccount sAccount = new SavingsAccount(10);
BankAccount bAccount = sAccount;
bAccount.deposit(500); // OK
```

- Tramite la variabile **bAccount** si può usare l'oggetto di tipo **SavingsAccount** come se fosse di tipo **BankAccount**, senza però poter accedere alle proprietà specifiche di **SavingsAccount**

```
bAccount.addInterest();
```

```
cannot resolve symbol
symbol  : method addInterest()
location: class BankAccount
      bAcct.addInterest();
            ^
1 error
```

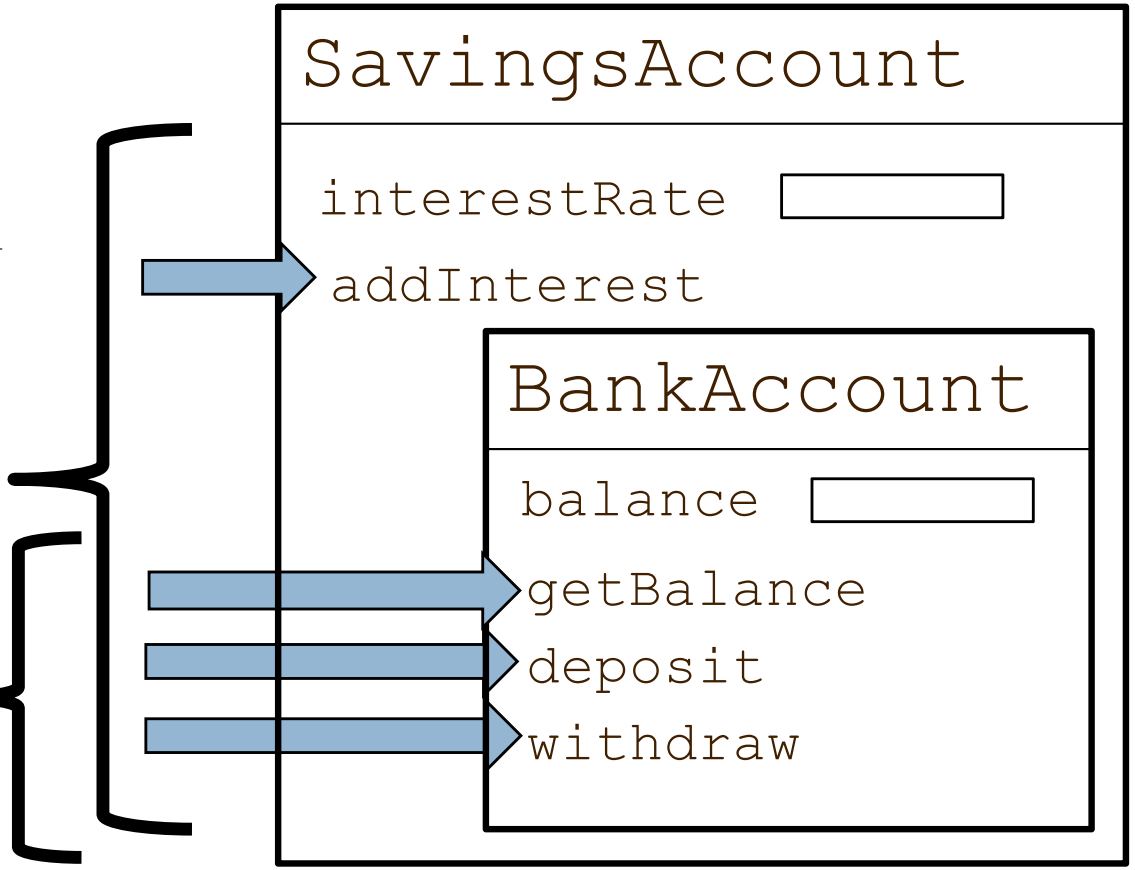
Conversione fra riferimenti

```
SavingsAccount sAccount = ...
BankAccount bAccount = sAccount;
```

È come se si vedesse l'oggetto attraverso una finestra più piccola… alcuni metodi non si vedono più!!

sAccount

bAccount



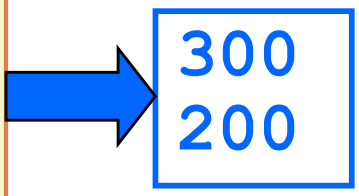
Quale scopo può avere questo tipo di conversione?

BankAccount: transferTo

- Aggiungiamo un metodo a **BankAccount**

```
public class BankAccount
{   ...
    public void transferTo(BankAccount other,double amount)
    {  withdraw(amount); // this.withdraw(...)
       other.deposit(amount);
    } // effettua un bonifico...
}
```

```
BankAccount account1 = new BankAccount(500);
BankAccount account2 = new BankAccount();
account1.transferTo(account2, 200);
System.out.println(account1.getBalance());
System.out.println(account2.getBalance());
```



Conversione fra riferimenti

Per quanto visto finora, **other** può anche riferirsi ad un oggetto di tipo **SavingsAccount**

```
BankAccount fromAccount = new BankAccount(1000);
SavingsAccount sAccount = new SavingsAccount(10);
BankAccount bAccount = sAccount;
fromAccount.transferTo(bAccount, 500);
```

- Il metodo **transferTo** di **BankAccount** richiede un parametro di tipo **BankAccount** e non conosce (né usa) i metodi specifici di **SavingsAccount**, ma è comodo poterlo invocare senza doverlo modificare!

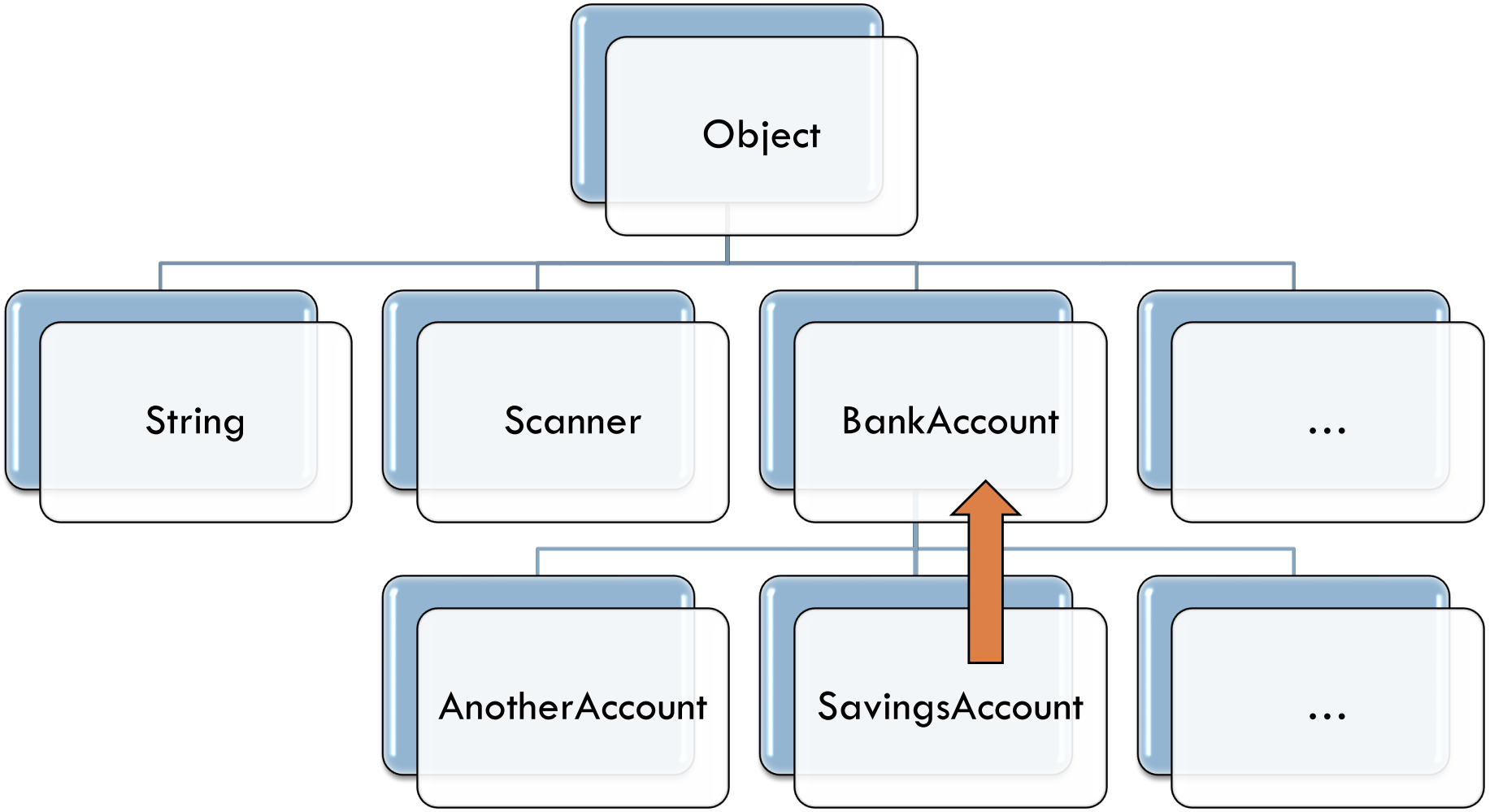
```
public class BankAccount
{   ...
    public void transferTo(BankAccount other,
                           double amount)
    {   withdraw(amount); //
        this.withdraw(...)
        other.deposit(amount);
    } // effettua un bonifico…
}
```

Conversione fra riferimenti

- Si tratta di una proprietà **ESTREMAMENTE** utile
  - Chi ha progettato la classe **`BankAccount`** l'ha dotata della possibilità di "fare un bonifico"
  - Questa funzionalità può operare anche se uno dei due conti bancari coinvolti (o anche entrambi!) è di un tipo ***derivato*** da **`BankAccount`**, tipo di cui il progettista di **`BankAccount`** (e, quindi, del metodo **`transferTo`**) non poteva nemmeno prevedere l'esistenza!!!
- Vengono "ridotte" le potenzialità dell'oggetto
  - In realtà, si usa **un riferimento che consente minori funzionalità**, l'oggetto non subisce modifiche

Conversione fra riferimenti

- Si parla di conversione mediante **up-casting**, perché si "risale" nell'albero della gerarchia di ereditarietà



Conversione fra riferimenti

- La conversione tra ***riferimento a sottoclasse*** e ***riferimento a superclasse*** può avvenire anche ***implicitamente*** (come tra **int** e **double**)

```
BankAccount fromAccount = new BankAccount(1000);
SavingsAccount sAccount = new SavingsAccount(10);
fromAccount.transferTo(sAccount, 500);
```

- Il compilatore sa che il metodo **transferTo** richiede un riferimento di tipo **BankAccount**, quindi
  - controlla che il tipo della variabile **sAccount** sia **BankAccount oppure** quello di una sua sottoclasse
  - effettua la "conversione" automaticamente
    - In realtà non c'è niente da convertire perché è un riferimento, ma si intende conversione come visto finora…

E senza up-casting?

- Se in Java non esistesse up-casting, sarebbe piuttosto complicato fare bonifici...

Servirebbero tanti metodi sovraccarichi, uno per ogni sottoclasse di BankAccount, ma tutti uguali!!! Pessimo...

```
public class BankAccount
{   ...
    public void transferTo(BankAccount other, double x)
    {   withdraw(x);
        other.deposit(x);
    }
    public void transferTo(SavingsAccount other, double x)
    {   withdraw(x);
        other.deposit(x);
    }
    public void transferTo(XyzkwjAccount other, double x)
    {   withdraw(x);
        other.deposit(x);
    }
}
```

Inoltre, dopo aver progettato una nuova sottoclasse, sarebbe necessario inserire un nuovo metodo in BankAccount !! Pessimo...

Conversione di oggetti?

- **Attenzione! Ribadiamo:**
  La "conversione" tra riferimenti non ha alcun effetto sull'oggetto che si trova all'indirizzo "convertito"
- Infatti, dopo un up-casting, mediante il riferimento originario si può ancora accedere all'oggetto avente tutte le sue potenzialità

```
SavingsAccount sAccount = new SavingsAccount(10);
BankAccount bAccount = sAccount; // up-casting
...
// bAccount.addInterest() non si può fare
sAccount.addInterest(); // funziona regolarmente
```

- **Non si convertono gli oggetti!!!**

Polimorfismo ed ereditarietà

Polimorfismo

- Sappiamo già che un oggetto di una sottoclasse può essere usato come se fosse un oggetto della superclasse

```
BankAccount account = new SavingsAccount(10);
account.deposit(500);
account.withdraw(200);
```

- Abbiamo sovrascritto/ridefinito **deposit** in **SavingsAccount**, in modo che prelevi una commissione ad ogni versamento.
- Quando si invoca **deposit** con la variabile **account**, che è una variabile di tipo **BankAccount** che punta a un oggetto di tipo **SavingsAccount**, quale metodo viene invocato?
  - il metodo **deposit** definito in **BankAccount**?
  - il metodo **deposit** ***ridefinito*** in **SavingsAccount**?

Polimorfismo

```
BankAccount account = new SavingsAccount(10);
account.deposit(500);
```

- Si potrebbe pensare
  - **account** è una variabile dichiarata di tipo **BankAccount,** quindi viene invocato il metodo **deposit** di **BankAccount** (cioè **NON** vengono addebitate le spese per l'operazione di versamento…)
- Ma **account** contiene un riferimento a un oggetto che, ***in realtà***, è di tipo **SavingsAccount**!
  E l'interprete Java lo sa perché lo vede nella memoria… (il compilatore no…)
  - secondo la semantica di Java viene invocato il metodo **deposit** di **SavingsAccount**

Polimorfismo

- Questa semantica si chiama ***polimorfismo*** ed è caratteristica dei linguaggi OOP

  l'invocazione di un metodo è SEMPRE determinato dal tipo dell'oggetto usato come parametro implicito e NON dal tipo della variabile oggetto

- Si parla di polimorfismo (dal greco, "molte forme") perché ***si compie la stessa elaborazione*** (**deposit**) ***in modi diversi***, in relazione all'oggetto usato

Polimorfismo

- Abbiamo già visto il polimorfismo a proposito dei **metodi sovraccarichi**
  - l'invocazione del metodo **println** si traduce nell'invocazione di un metodo scelto fra alcuni metodi diversi, in relazione **al tipo del parametro esplicito**
  - **il compilatore** decide quale metodo invocare

Polimorfismo

- Nel caso dei metodi sovrascritti la situazione è molto diversa, perché **la decisione non può essere presa dal compilatore, ma deve essere presa dall'ambiente runtime (l'interprete)**
  - si parla di selezione posticipata (late binding)
  - selezione anticipata (early binding) nel caso di metodi sovraccarichi

Ereditarieta': figli e nipoti

La classe "nonno"

- La classe C deriva dalla classe B che deriva dalla classe A
- La classe A ha un metodo
  - public boolean metodo()
  - La classe B sovrascrive metodo()
- La classe C può :
  - Sovrascrivere a sua volta metodo()
  - Invocare metodo di B con super.metodo()
- La classe C può invocare metodo di A?
  - No! super.super.metodo() non funziona!

La classe "nonno"

- Perché la classe C non può accedere al metodo del nonno sovrascritto dal genitore?
  - Quando avete progettato la classe C avete deciso di farla derivare da B e non da A
    - Vuol dire che vi andava bene come B si era specializzata rispetto ad A
    - Se vi serviva qualche metodo di A così com'era e non come sovrascritto da B, avete sbagliato a livello progettuale! C doveva derivare da A!

- Si può aggirare questo vincolo?
  - Sì, ma non si fa! La mamma ha sempre ragione!

L'operatore *instanceof*

- E' possibile stabilire se un oggetto è un'istanza del tipo specificato (classe o interfaccia) attraverso **l'operatore instance of**

```
variabileOggetto instanceof NomeClasse
```

- Restituisce un valore booleano
- Dipende dal tipo di oggetto in esecuzione, non dal tipo dichiarato del riferimento

L'operatore *instanceof*

- Posso utilizzarlo per fare casting in modo sicuro

```
BankAccount account = new SavingAccount();
SavingAccount saccount;
if (account instanceof SavingAccount){
    saccount = (SavingAccount)account;
    saccount.addInterest(); //non potevo invocarlo da account
}
```

- Senza instanceof, se la variabile **non** contiene un riferimento a SavingAccount verrà lanciata ClassCastException

```
BankAccount account = new BankAccount();
((SavingAccount)account).addInterest(); //ClassCastException
```

Ereditarietà e controllo di accesso

Controllo di accesso

- Java fornisce **quattro livelli** per il controllo di accesso a metodi, campi, e classi
  - **public** ✓
  - **private** ✓
  - **protected**
  - **package**

Accesso package

- Un membro di classe (o una classe) senza specificatore di accesso ha **di default** un'impostazione di accesso **package**
  - I metodi di classi nello **stesso pacchetto** vi hanno accesso
  - Può essere una buona impostazione per le classi, **non lo è** per le **variabili**, perché si viola l'incapsulamento
  - **Errore comune**: dimenticare lo specificatore di accesso per una variabile di esemplare

```
public class Window extends Container {
   String warningString; // è accessibile da altre
   …                     // classi nello stesso pacchetto!
}
```

- È un **rischio** per la sicurezza: un altro programmatore può realizzare una classe nello stesso pacchetto e **ottenere l'accesso** al campo di esemplare

Accesso protected

- Il progettista della superclasse decide se rendere accessibile in modo **protected** lo stato della classe (o una sua parte…)
  - Variabili e metodi definiti protected sono accessibili da **tutte** le sottoclassi!
- È una parziale violazione dell'incapsulamento, ma avviene in modo consapevole ed esplicito
- Anche i metodi possono essere definiti **protected**
  - possono essere invocati soltanto all'interno della classe in cui sono definiti (come i metodi **private**) **e all'interno delle classi derivate da essa**

Ereditarietà e modificatore final

Modificatore di accesso final

- Abbiamo incontrato la parola chiave **final** nella definizione di una variabile per impedire successive assegnazioni di valori
- Può essere utilizzata anche per
  - Metodi: non possono essere sovrascritti da sottoclassi
  - Classi: non posso derivare sottoclassi

```
public final class BankAccount {
        …
}
```

```
public class SavingAccount extends BankAccount{ //ERRORE
        …
}
```

La superclasse universale Object

La classe Object

- Sappiamo già che ogni classe di Java (in maniera diretta o indiretta) è **sottoclasse di Object**
- Quindi ogni classe di Java **eredita tutti i metodi** della classe **Object**
- Alcuni dei più utili metodi di **Object** sono i seguenti

Metodo	Obiettivo
`String toString()`	Restituisce una stringa che descrive l'oggetto
`boolean equals(Object otherObject)`	Verifica se l'oggetto è uguale a un altro
`Object clone()`	Crea una copia completa dell'oggetto

- Ma perché siano davvero utili, nella maggior parte dei casi bisogna **sovrascriverli**.

Sovrascrivere il metodo toString

Sovrascrivere il metodo toString

- Perché sovrascrivere toString?

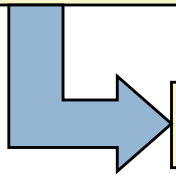
- L'invocazione di questo metodo con un oggetto (qualsiasi) ne restituisce la cosiddetta ***descrizione testuale standard***

```
BankAccount@111f71
```

  - La descrizione testuale standard è *il nome della classe* seguito dal carattere @ e da un numero che dipende dall'indirizzo dell'oggetto in memoria

Sovrascrivere il metodo toString

```
BankAccount account = new BankAccount();
System.out.println(account.toString());
```



```
BankAccount@111f71
```

- In generale la descrizione testuale standard non è molto utile, anche se ha un'interessante proprietà
  - **Oggetti distinti hanno descrizione testuale standard diversa** (perché si trovano in posizioni diverse nella memoria)

Sovrascrivere il metodo toString

- E' molto più comodo, ad esempio per verificare il corretto funzionamento del programma durante il collaudo, ottenere una descrizione testuale di **BankAccount** contenente il valore del saldo
  - questa funzionalità non può essere svolta dal metodo **toString** di **Object**, perché chi ha definito **Object** nella libreria standard non aveva alcuna conoscenza della struttura di **BankAccount**
- Bisogna sovrascrivere **toString** in **BankAccount** (già visto!)

```
public String toString()
{  return "BankAccount[balance=" + balance + "]"
          + super.toString();
} // stile convenzionale per ridefinire toString
```

```
BankAccount[balance=1500]BankAccount@111eac
```

Sovrascrivere sempre toString

- Sovrascrivere il metodo **toString** in tutte le classi che si definiscono è considerato un ottimo stile di programmazione
- Il metodo **toString** di una classe dovrebbe produrre una stringa contenente tutte le informazioni di stato dell'oggetto, cioè
  - il valore di tutte le sue variabili di esemplare
  - il valore di eventuali variabili statiche della classe che non siano costanti
- **Questo stile di programmazione è molto utile per il debugging ed è spesso usato nella libreria standard**

Sovrascrivere sempre toString

- Nelle classi derivate, il metodo **toString** dovrebbe sempre invocare il metodo **toString** della superclasse, concatenando le informazioni specifiche della sottoclasse

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

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

```
SavingsAccount[interestRate=10]BankAccount[balance=1500]
SavingsAccount@116ef5
```

Sovrascrivere il metodo toString

- Il metodo **toString** di una classe viene invocato implicitamente anche quando si concatena un oggetto della classe con una stringa

```
BankAccount account = new BankAccount();
String s = "Conto " + account;
```

- Questa concatenazione è sintatticamente corretta, come per i tipi di dati numerici, e viene interpretata dal compilatore così

```
BankAccount account = new BankAccount();
String s = "Conto " + account.toString();
```

Sovrascrivere il metodo equals

```
public boolean equals(Object otherObject)
```

- Perché sovrascrivere equals?
  - L'invocazione di questo metodo restituisce true se e solo se l'oggetto su cui viene invocato coincide con l'oggetto passato come parametro:
  - *this == otherObject*
  - Non molto utile… sarebbe meglio confrontare gli oggetti in base alle variabili che ne descrivono lo stato!

Esempio: bankAccount

```
public class BankAccount{
    …
    public boolean equals(Object otherObject){
       BankAccount otherAccount = (BankAccount)otherObject;
       return balance == otherAccount.balance;
    }
}
```

- Problema: nella firma di equals (che devo lasciare invariata per sovrascrivere) ho Object e non BankAccount
  - Devo fare casting esplicito per poter accedere alle variabili di esemplare dell'oggetto otherObject che in realtà sarà un riferimento di tipo BankAccount
  - ATTENZIONE: se scrivo un metodo
    - public boolean equals(BankAccount account) non sovrascrivo il metodo equals di Object!!!

Metodi "magici"

- A questo punto siamo in grado di capire come è possibile definire metodi che, come **println**, sono in grado di ricevere come parametro un oggetto di qualsiasi tipo
  - un riferimento a un oggetto di qualsiasi tipo può sempre essere convertito automaticamente (up-casting) in un riferimento di tipo **Object**, con cui si invoca **toString()**

```
public void println(Object obj)
{  println(obj.toString()); // ricorsione?? NO
}

public void println(String s)
{  ... // questo viene invocato per le stringhe
       // e ne stampa i singoli caratteri
}
```

```
public void println(Object obj)
{  println(obj.toString());
}
public void println(String s)
{  ... // questo viene invocato per le stringhe
}
```

- Ma un esemplare di **String** è anche un esemplare di **Object**…
  - come viene scelto il giusto metodo, tra quelli sovraccarichi, quando si invoca **println** con una stringa, come avviene all'interno del primo metodo?
  - Nell'up-casting usato implicitamente per i parametri, durante l'invocazione di metodi, il compilatore cerca sempre di "**fare il minor numero di conversioni**"
    - **viene usato, quindi, il metodo "più specifico"**
    - **la presenza della seconda versione del metodo fa in modo che l'invocazione presente nel primo metodo NON sia una ricorsione**

Metodi "magici"

- Poi, il metodo **println** ha una forma sovraccarica anche per ogni tipo di dato primitivo, perché **variabili di tipi primitivi non sono variabili riferimento** e non possono essere assegnate, mediante up-casting, a un riferimento di tipo **Object**

```
public void println(Object obj)
{  println(obj.toString()); }
public void println(String s)
{  ... // per ogni carattere... }

public void println(char c)
{  ... // l'unico che interagisce con il flusso!! }

public void println(int x)
{  println(Integer.toString(x));
   // così invoca il metodo che stampa una stringa
}
public void println(double x)
{  println(Double.toString(x)); }
...
```

Errori comuni: mettere in ombra le variabili esemplare

Mettere in ombra le variabili ereditate

```
Public class SavingsAccount extends BankAccount
{   …
    public void addInterest()
    {  deposit(getBalance() * interestRate/100);
    }
}
```

- Abbiamo visto che una sottoclasse **eredita tutte** le variabili di esemplare definite nella superclasse
  - Ogni oggetto della sottoclasse ha una propria copia di tali variabili, come ogni oggetto della superclasse, ma se sono **private non vi si può accedere dai metodi della sottoclasse!**

Mettere in ombra le variabili ereditate

```
Public class SavingsAccount extends BankAccount
{   …
    public void addInterest()
    {   deposit(balance * interestRate/100); //ERRORE
    }
}
```

- Il compilatore segnala l'errore
  - **balance has private access in BankAccount**
- Per accedere alle variabili private ereditate, bisogna utilizzare metodi pubblici messi a disposizione dalla superclasse (se ce ne sono!)

Mettere in ombra le variabili ereditate

- A volte si cade nell'errore di **definire la variabile anche nella sottoclasse**

```
Public class SavingsAccount extends BankAccount
{  private double balance;
   public void addInterest()
   {  deposit(balance* interestRate/100);
      // sto usando balance di SavingsAccount!
   }
   …
   }
```

- Il compilatore non segnala alcun errore, ma ora **SavingsAccount** ha due variabili che si chiamano **balance, una propria e una ereditata, tra le quali non c'e' alcuna relazione!**

Mettere in ombra le variabili ereditate

- L'esistenza di due distinte variabili balance è fonte di errori difficili da diagnosticare
  - **Modifiche a balance fatte da metodi di SavingsAccount non sono visibili a metodi di BankAccount, e viceversa**
  - Ad esempio, dopo un'invocazione a withdraw(), la variabile balance definita in SavingsAccount non viene modificata
    - Il calcolo degli interessi sarà sbagliato