Ereditarietà

- **Concetti principali**
- **Ereditarietà e (overriding) di metodi**
  - Dynamic dispatch e polimorfismo
- **Ereditarietà e costruttori**
- **Livelli di accesso `protected` e `package`**
- **La classe `Object`**
  - metodi **`toString`**, **`equals`** e **`clone`**

Ereditarietà – introduzione

- Un meccanismo per estendere classi con nuovi campi e nuovi metodi
- Esempio: dato un conto bancario, vogliamo definire un conto con interessi
- **`SavingsAccount`** (~ libretto di risparmio)

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

Continua…

Ereditarietà – introduzione

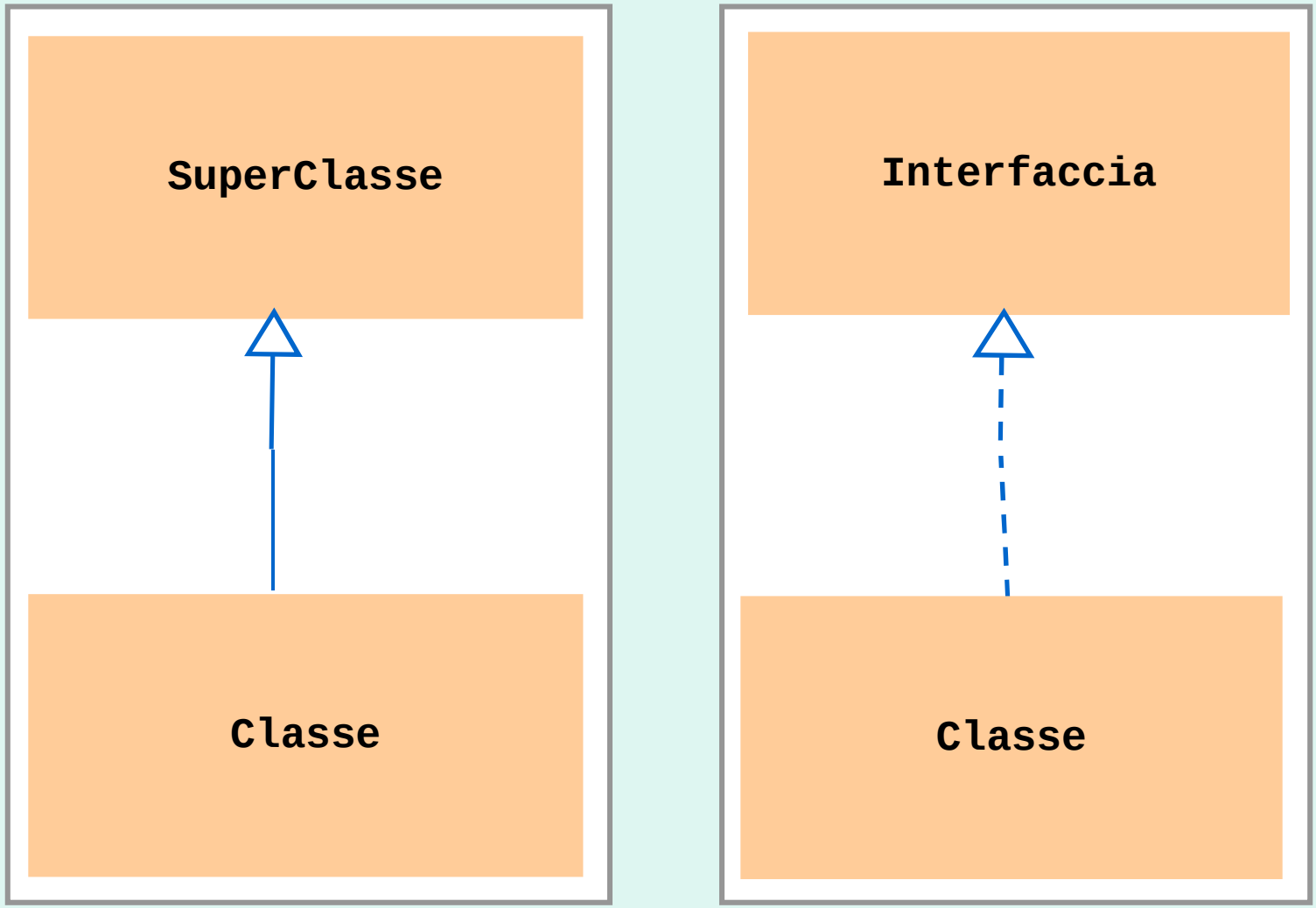
- **`SavingsAccount` eredita metodi e campi di `BankAccount`**

```
SavingsAccount collegeFund = new SavingsAccount(10);
// Conto con il 10% tasso di interesso
collegeFund.deposit(500);
// Possiamo usare un metodo di BankAccount
// anche su oggetti di tipo SavingsAccount
```

- **Terminologia**
  - *superclasse* = classe che viene estesa (**`BankAccount`**)
  - *sottoclasse* = classe che estende (**`Savings`**)

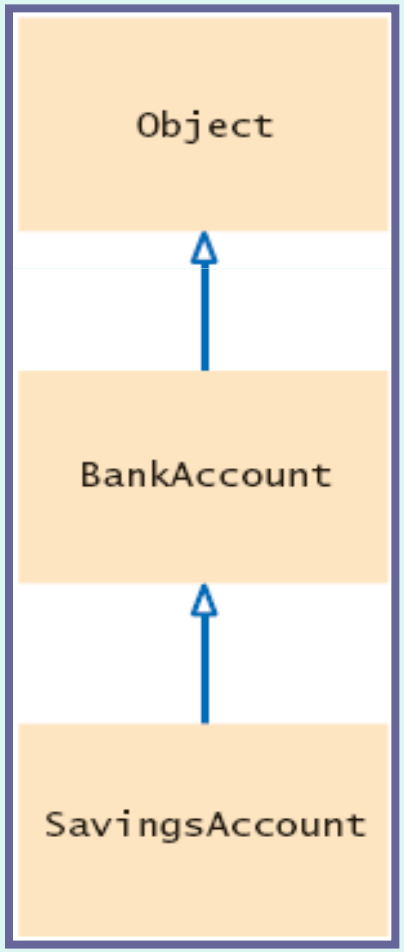
Continua…

Diagrammi UML



Diagrammi UML

- La relazione di sottoclasse è transitiva
- Ogni class estende la classe **`Object`**, direttamente o indirettamente



Superclassi vs interfacce

- **Estendere una classe è diverso da implementare una interfaccia**
- **Entrambi i meccanismi generano sottotipi**
  - Interfacce rappresentano le classi che le implementano
  - Classi rappresentano le proprie sottoclassi
- **Ma …**
  - Una sottoclasse eredita l'implementazione dei metodi ed i campi della superclasse
  - riuso di codice

Ereditarietà – sottoclassi

- **Nell'implementazione di una sottoclasse definiamo solo ciò che cambia rispetto alla superclasse**

Ereditarietà – sottoclassi

- **`SavingsAccount` implementa le differenze rispetto a `BankAccount`**

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

   private double interestRate;
}
```

Domanda

- Quali metodi possono essere invocati su oggetti di tipo `SavingsAccount`?

Risposta

- Tutti i metodi ereditati:

  `deposit()`, `withdraw()`, `getBalance()`

- Il metodo `addInterest()` definito nella sottoclasse

Ereditarietà – `this`

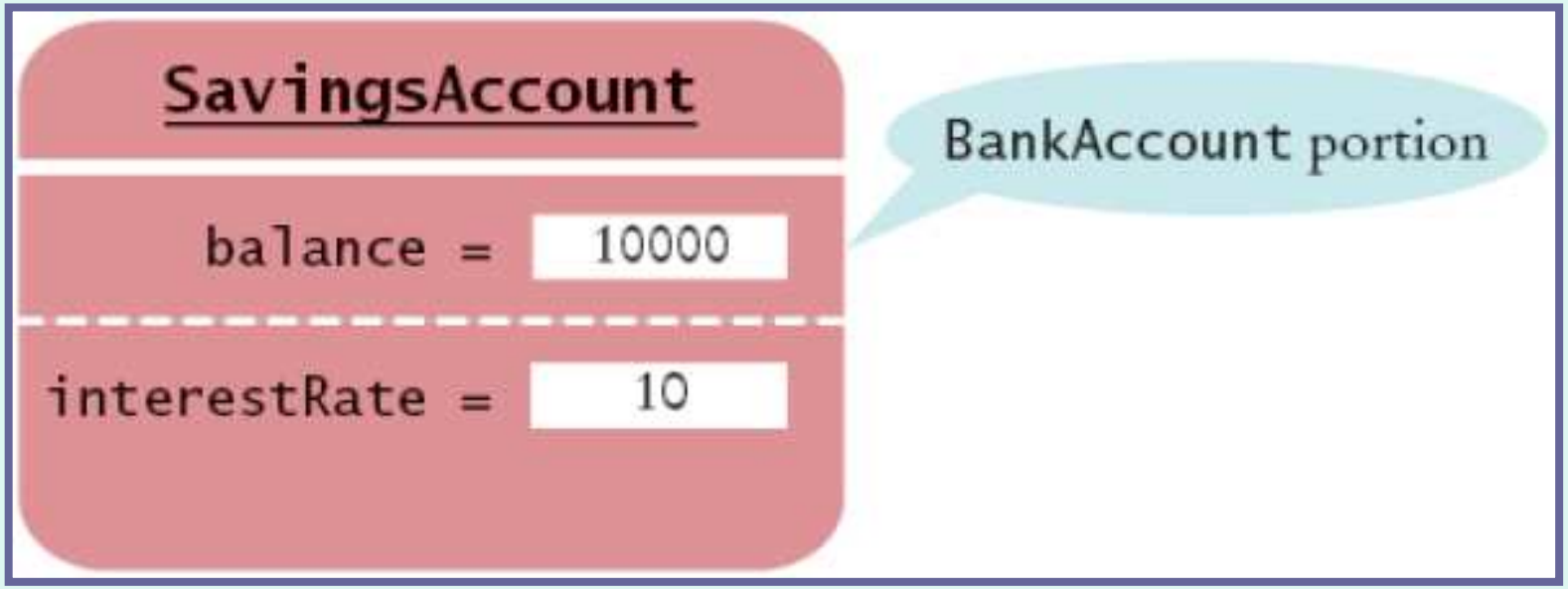
- **`addInterest()` invoca `deposit()` senza specificare un oggetto**
- **l'invocazione riguarda `this`, l'istanza della sottoclasse che eredita i metodi della superclasse**

Ereditarietà – encapsulation

- **Le sottoclassi non hanno accesso ai campi privati della superclasse**
- **`SavingsAccount` non può riferire direttamente il campo `balance`**
- **Può usare i metodi ereditati dalla superclasse**
  - **`addInterest()`** accede a **`balance`** con i metodi **`getBalance()`** e **`deposit()`**

Istanze di sottoclasse

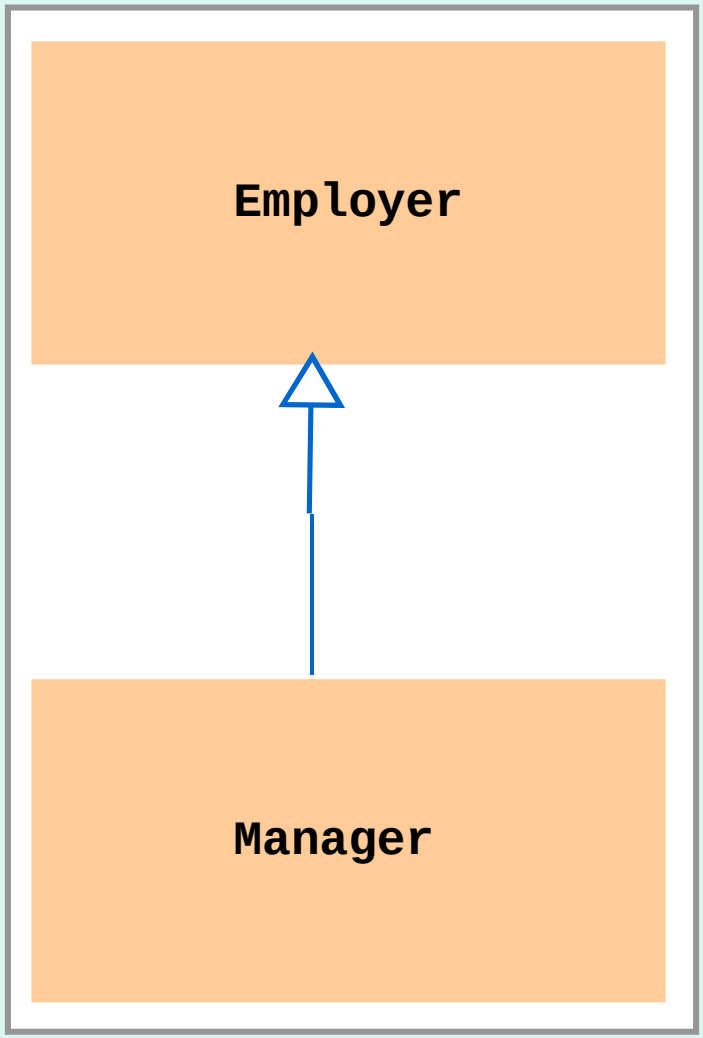
- **Un oggetto di tipo `SavingsAccount` eredita il campo `balance` da `BankAccount`, ed ha un campo aggiuntivo, `interestRate`:**



Domanda

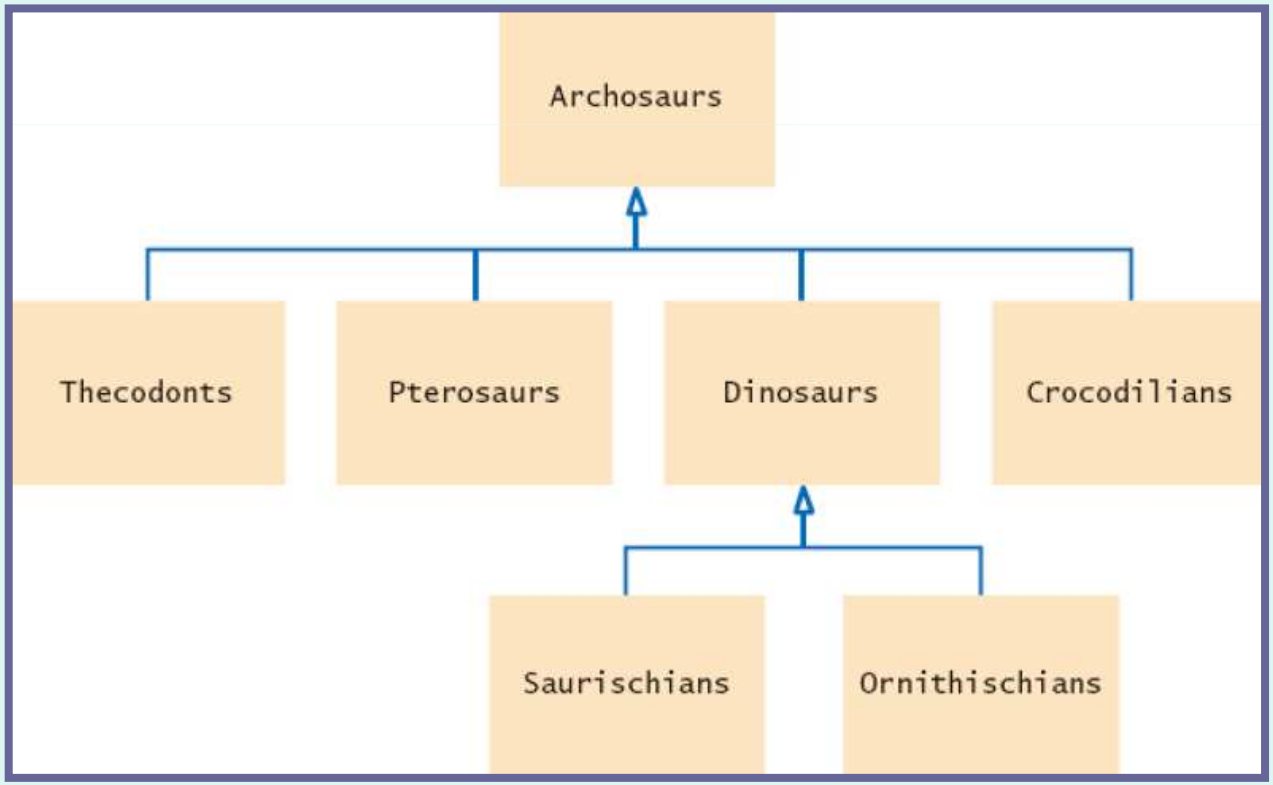
- **Date due classi `Manager` e `Employee`, quale delle due definireste come superclasse e quale come sottoclasse?**

Risposta

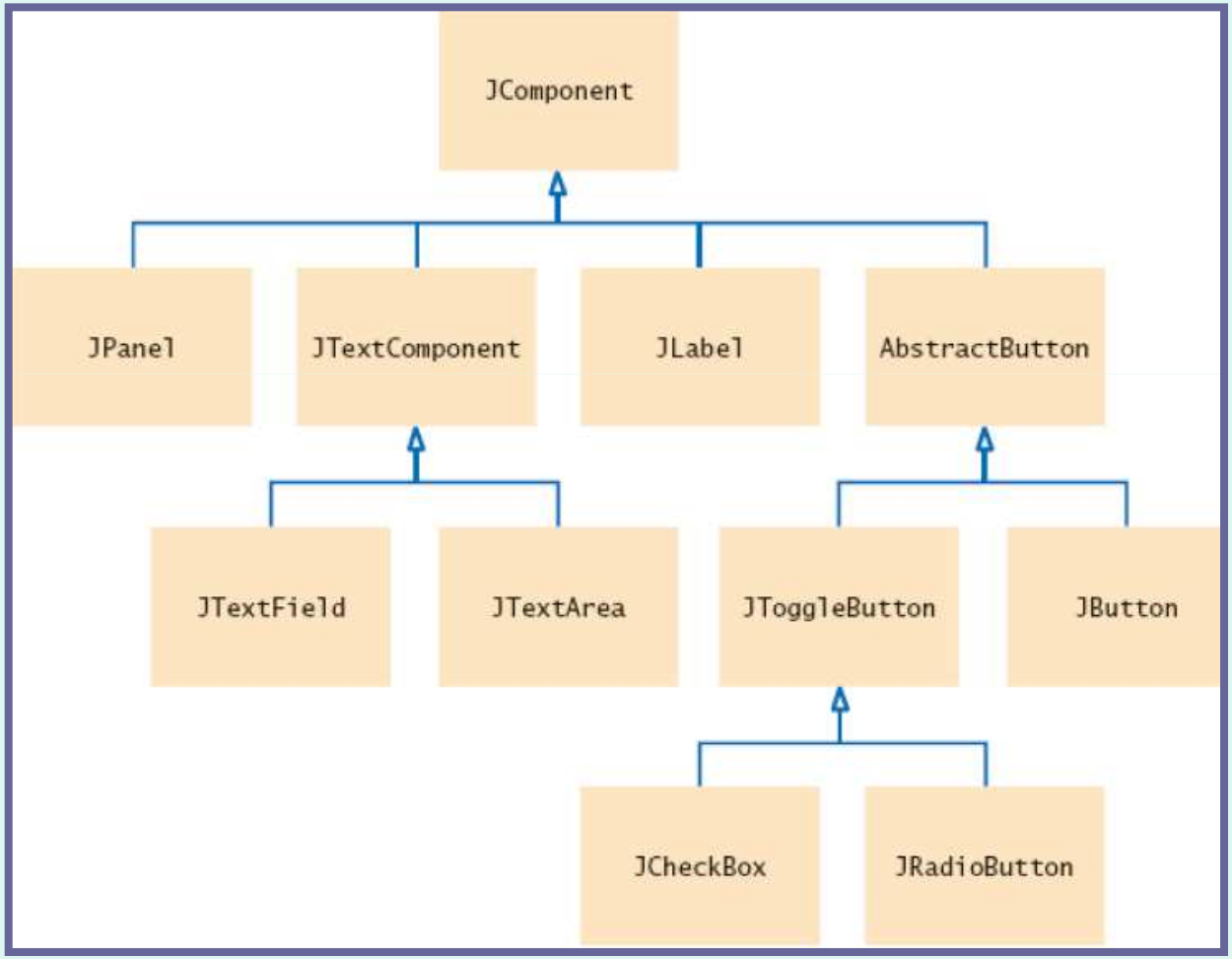


Gerarchie di classi

- **Le applicazioni sono spesso formate come gerarchie complesse di classi in relazione di ereditarietà**



Gerarchie di classi – Swing



Continua…

Gerarchie di classi – Swing

- La superclasse `JComponent` definisce due metodi `getWidth`, `getHeight` ereditati da tutte le classi nella gerarchia
- La classe `AbstractButton` ha metodi set/get per definire e accedere alla etichetta o icona associata al pulsante

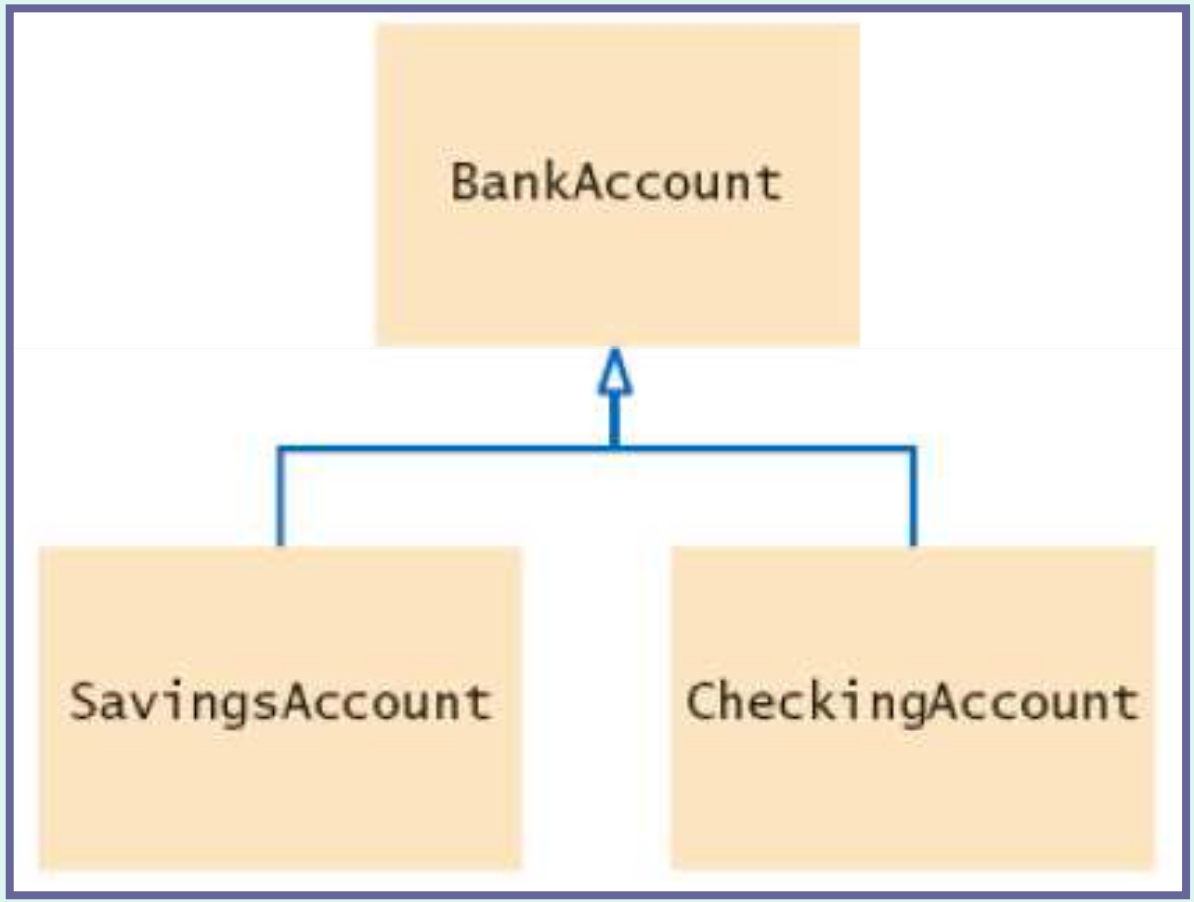
Domanda

- Quale è lo scopo della classe `JTextComponent` nella gerarchia Swing?

Risposta

- Implementa gli aspetti comuni del comportamento delle classi `JTextField` e `JTextArea`

Una gerarchia di conti bancari



Continua…

Una gerarchia di conti bancari

- **`BankAccount`** realizza le funzionalità di base di un conto bancario
- **`CheckingAccount`**: non offre interessi, ma permette un numero di transazioni gratuite e per le altre commissioni basse
- **`SavingsAccount`** : interessi calcolati su base mensile e transazioni con commissione

Continua…

Una gerarchia di conti bancari

- Tutti i tipi di conto offrono il metodo `getBalance()`
- Tutti i tipi di conto offrono `deposit()` e `withdraw()`,
  - Implementazione diversa nei diversi tipi
- `CheckingAccount` include un metodo `deductFees()` per dedurre le commissioni
- `SavingsAccount` include un metodo `addInterest()` per il calcolo degli interessi

Ereditarietà – metodi

- **Tre scelte possibili nel progetto di una sottoclasse**
  - ereditare i metodi della superclasse
  - modificare i metodi della superclasse (*overriding*)
  - definire nuovi metodi / nuove versioni dei metodi ereditati

Continua…

Ereditarietà – metodi ereditati

- **Metodi della superclasse visibili nella sottoclasse**
- **Non ridefiniti nella sottoclasse**
- **I metodi della superclasse possono essere sempre invocati su oggetti della sottoclasse**
  - Quando invocati su oggetti della sottoclasse **`this`** punta all'oggetto della sottoclasse
  - Il tipo di this cambia, a seconda dell'oggetto a cui **`this`** viene legato

Continua…

Ereditarietà – metodi aggiuntivi

- **Le sottoclassi possono definire nuovi metodi, non definiti nella superclasse**
- **I nuovi metodi sono invocabili solo su oggetti della sottoclasse**

Ereditarietà – overriding di metodi

- **Forniscono una implementazione diversa di un metodo che esiste nella superclasse**
- **Il metodo nella sottoclasse deve avere la stessa firma del metodo nella superclasse**
  - stesso nome/tipo risultato, stessi tipi dei parametri,
  - livello di accesso maggiore o uguale

Continua…

Ereditarietà – variabili di istanza

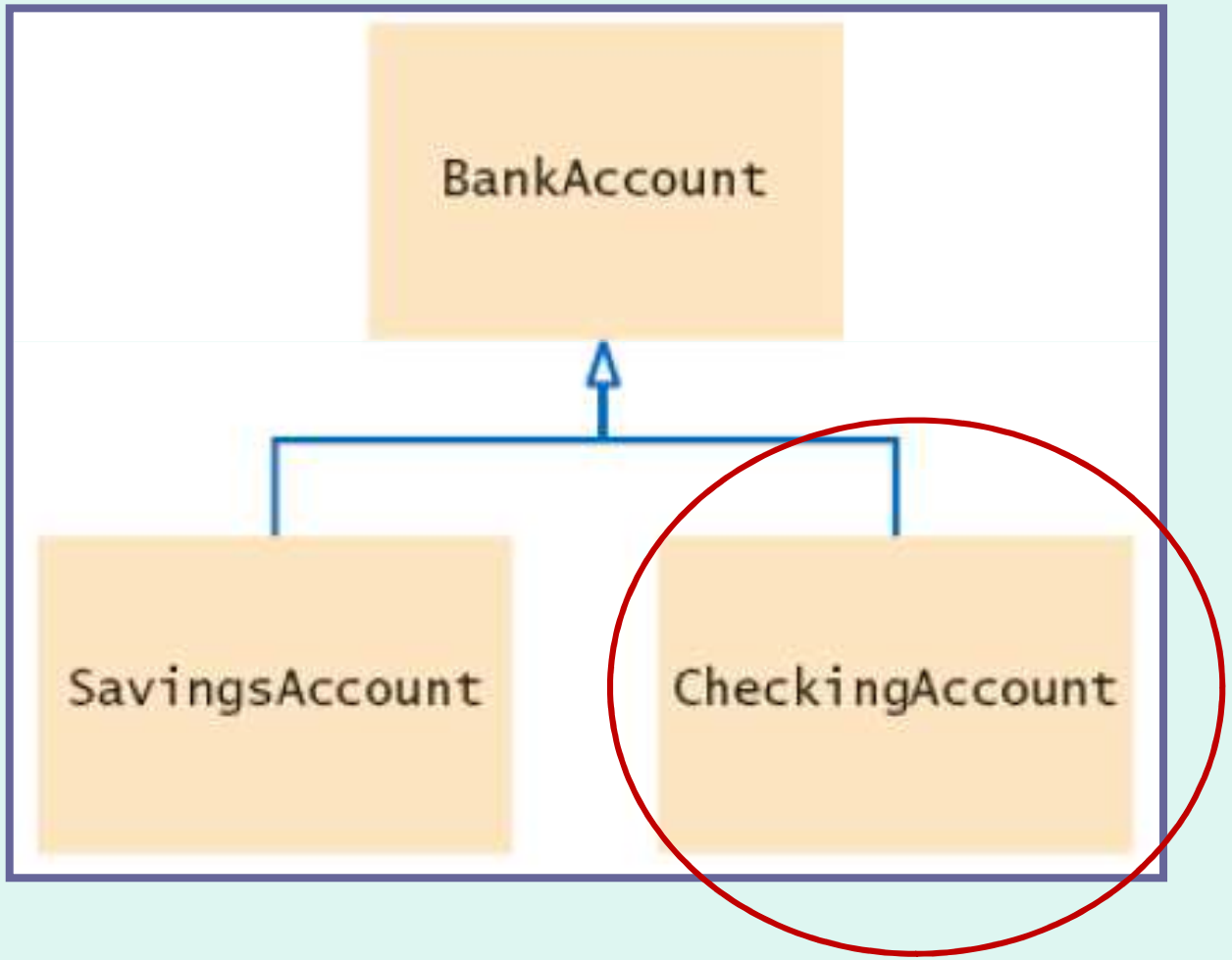
- **Tutti i campi di una superclasse sono automanticamente ereditati nella sottoclasse**
  - Ma, se privati, non possono essere acceduti direttamente dalla sottoclasse!
- **La sottoclasse può definire campi che non esistono nella superclasse**

Continua…

Ereditarietà – variabili di istanza

- **Nel caso di collisione di nomi …**
  - La definizione del campo della sottoclasse maschera quella del campo nella superclasse
- **Collisioni di nomi legali ma decisamente inopportune (… vedremo)**

Una gerarchia di conti bancari



Continua…

La classe `CheckingAccount`

- **Sovrascrive (*overrides*) i metodi `deposit()` e `withdraw()` per incrementare il contatore delle operazioni:**

```
public class CheckingAccount extends BankAccount
{
   public void deposit(double amount) {. . .}
   public void withdraw(double amount) {. . .}
   public void deductFees() {. . .} // nuovo metodo
   private int transactionCount;    // nuovo campo
}
```

Continua…

La classe `CheckingAccount`

- **due variabili di istanza:**
  - **`Balance`**, ereditato da **`BankAccount`**
  - **`transactionCount`**, nuovo

Continua…

La classe `CheckingAccount`

- **Quattro metodi**
  - **`getBalance()`**
    - ereditato da **`BankAccount`**
  - **`deposit(double amount)`**
    - sovrascrive il metodo corrispondente di **`BankAccount`**
  - **`withdraw(double amount)`**
    - sovrascrive il metodo corrispondente di **`BankAccount`**
  - **`deductFees()`**
    - nuovo

Accesso ai campi della superclasse

- Consideriamo il metodo **`deposit()`** della classe **`CheckingAccount`**

```
public void deposit(double amount)
{
   transactionCount++;
   // aggiungi amount a balance
   . . .
}
```

Continua…

Accesso ai campi della superclasse

- **Consideriamo il metodo `deposit()` della classe `CheckingAccount`**

```
public void deposit(double amount)
{
   transactionCount++;
   // aggiungi amount a balance
   balance += amount;// ERRORE!
}
```

- **Non possiamo modificare direttamente `balance`**
  - **`balance`** è un campo privato della superclasse

Continua…

Accesso ai campi della superclasse

- Una sottoclasse non ha accesso ai campi / metodi `private` della superclasse
- Deve utilizzare l'interfaccia `public` (o `protected` vedremo …) della superclasse
- Nel caso in questione, `CheckingAccount` deve invocare il metodo `deposit()` della superclasse

Accesso ai campi della superclasse

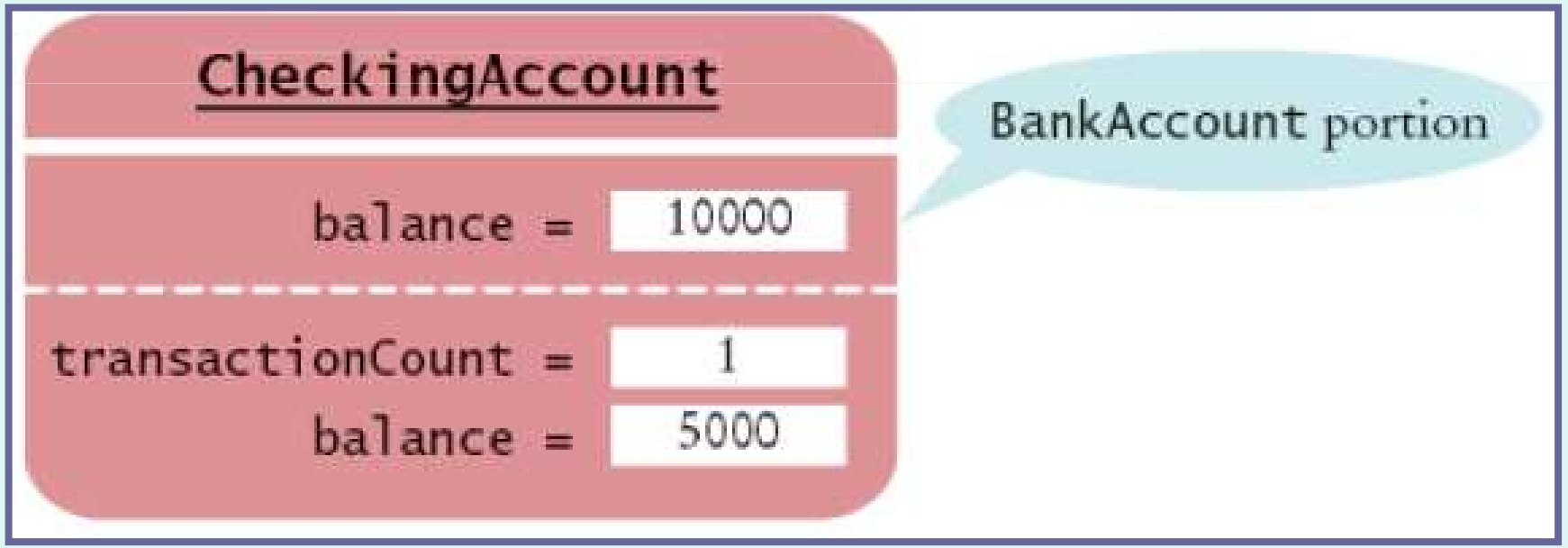
- **Errore tipico: aggiungiamo un nuovo campo con lo stesso nome:**

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

Continua…

Accesso ai campi della superclasse

- **Ora il metodo deposit compila correttamente ma non modifica il saldo corretto!**



Continua…

Accesso ai campi della superclasse

- **Soluzione, accediamo i campi della superclasse via i metodi della stessa superclasse**

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

Continua…

Chiamata di un metodo *overridden*

- **Non possiamo invocare direttamente `deposit(amount)` ....**

```
class CheckingAccount public {
   . . .
   void deposit(double amount)
   {
    transactionCount++;
    balance = balance + amount;
    deposit(amount); // questo è un loop!
   }
   . . .
}
```

- **È come invocare `this.deposit(amount)`**

Continua...

Chiamata di un metodo *overridden*

- Dobbiamo invece invocare il metodo **`deposit()`** della superclasse
- Invochiamo via **`super`**

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

Sintassi: chiamata via `super`

```
super.methodName(parameters)
```

Esempio:

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

Scopo:

Invocare un metodo overridden della superclasse

La classe CheckingAccount

```
public class CheckingAccount extends BankAccount
{
   . . .
   public void withdraw(double amount)
   {
      transactionCount++;
      // sottrai amount da balance

      super.withdraw(amount);
   }
```

Continua…

La classe `CheckingAccount`

- **Definizione dei metodi aggiuntivi**

```
   public void deductFees()
   {
      if (transactionCount > FREE_TRANSACTIONS)
      {
         double fees = TRANSACTION_FEE
            * (transactionCount - FREE_TRANSACTIONS);
         super.withdraw(fees);
      }
      transactionCount = 0;
   }
   . . .
   private static final int FREE_TRANSACTIONS = 3;
   private static final double TRANSACTION_FEE = 2.0;
}
```

Domanda

- Perchè il metodo `withdraw()` nella classe `CheckingAccount` invoca `super.withdraw()`?
- È possibile invocare un metodo della supeclasse senza utilizzare il riferimento `super`?
- Perchè il metodo `deductFees()` nella classe `CheckingAccount` invoca `super.withdraw()`?

Risposte

- Perché deve modificare la variabile `balance` e non può farlo direttamente in quanto è privata della superclasse n
- Si se la sottoclasse non ridefinisce il metodo
- Per evitare che il prelievo dell'importo delle commissioni venga contato come operazione.