Programmazione II

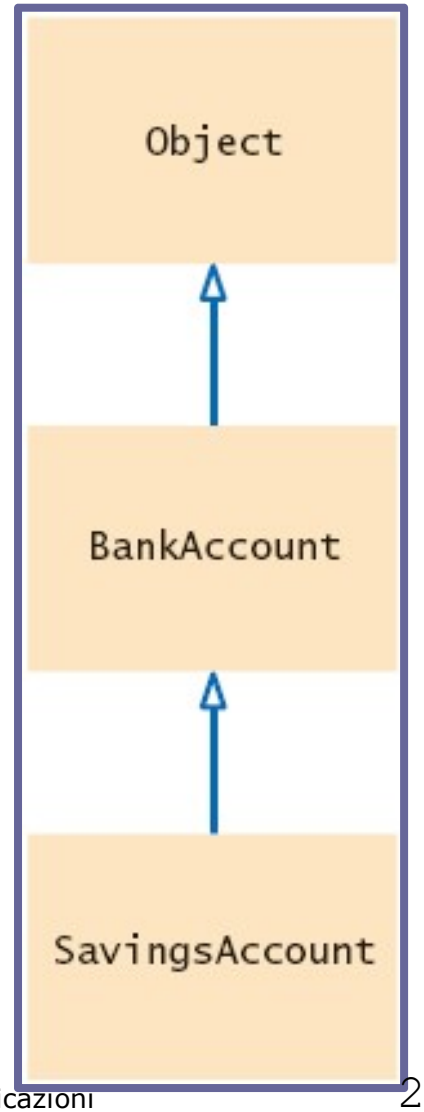
A.A. 2022-23
Prof. Maria Tortorella

Inheritance (Ereditarietà)

- La gerarchia ereditaria
- `protected` and `package` access control
- La superclasse `Object`
- L'ereditarietà come meccanismo di refactoring
- Classi astratte

Abbiamo già visto …

- La gerarchia ereditaria ha origine nella classe Object
- Oltre ad aggiungere metodi, una classe derivata può cambiare (override) i metodi della superclasse
- Le variabili ed i metodi privati di una superclasse rimangono tali anche per le sottoclassi



Nota sintattica – un esempio

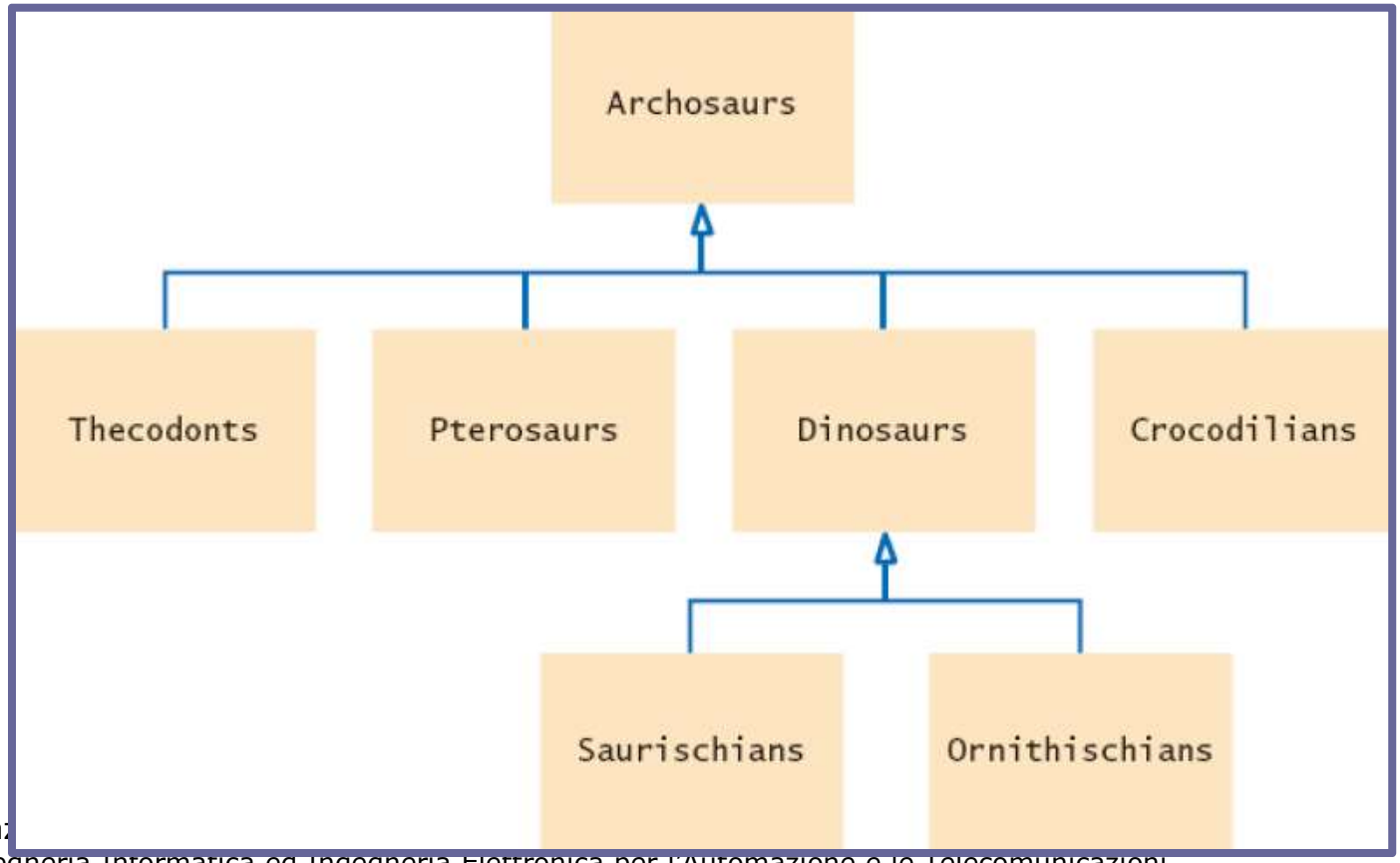
```
class SubclassName extends SuperclassName {
      methods
      instance fields
}
```

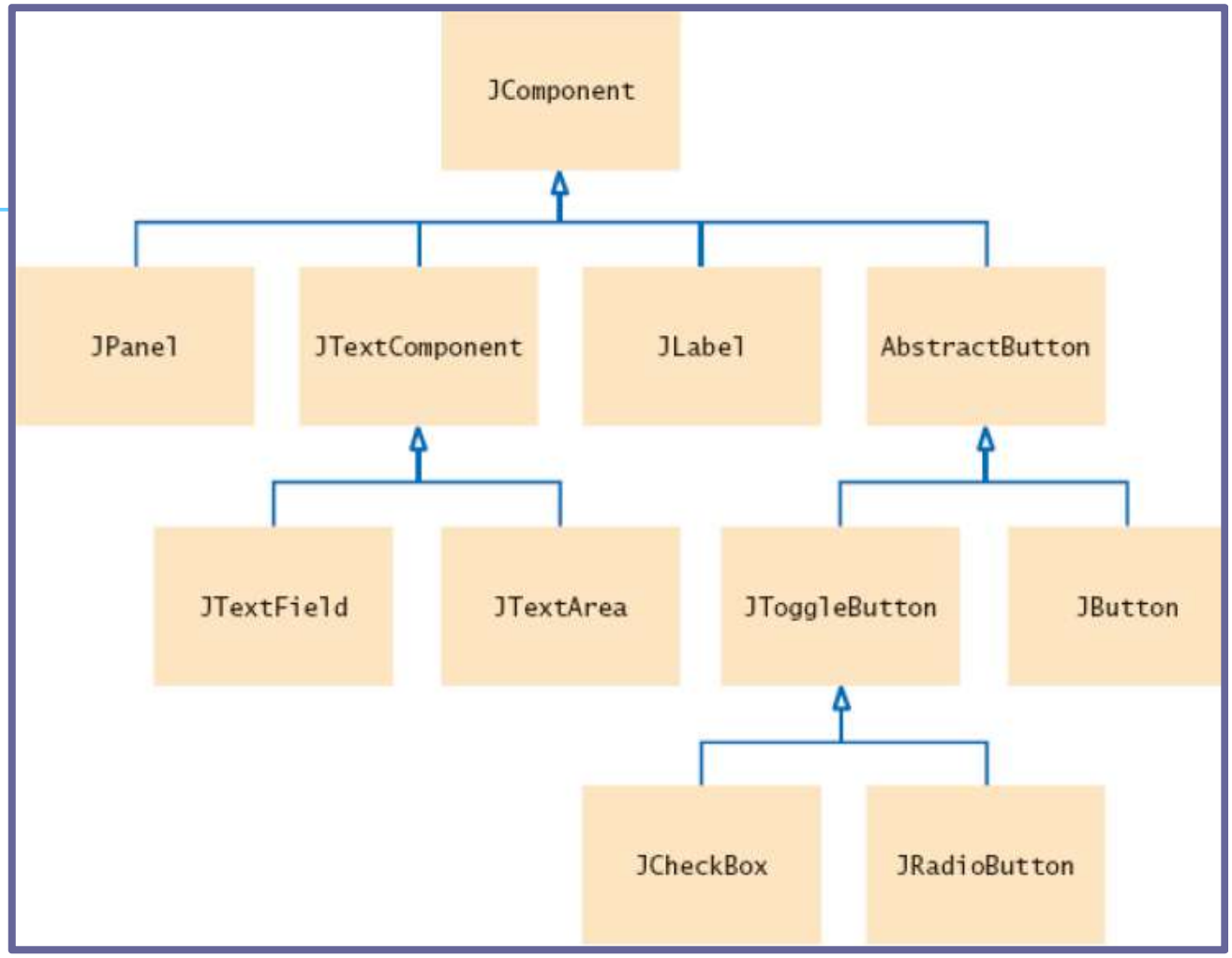
```
public class SavingsAccount extends BankAccount
{
   public SavingsAccount(double rate){
      super();
      interestRate = rate;
   }

   public void addInterest(){
      double interest=getBalance()*interestRate/100;
      deposit(interest);
   }
   private double interestRate;
}
```

Inheritance Hierarchies

- Un insieme di classi può dar vita ad una gerarchia anche molto articolata



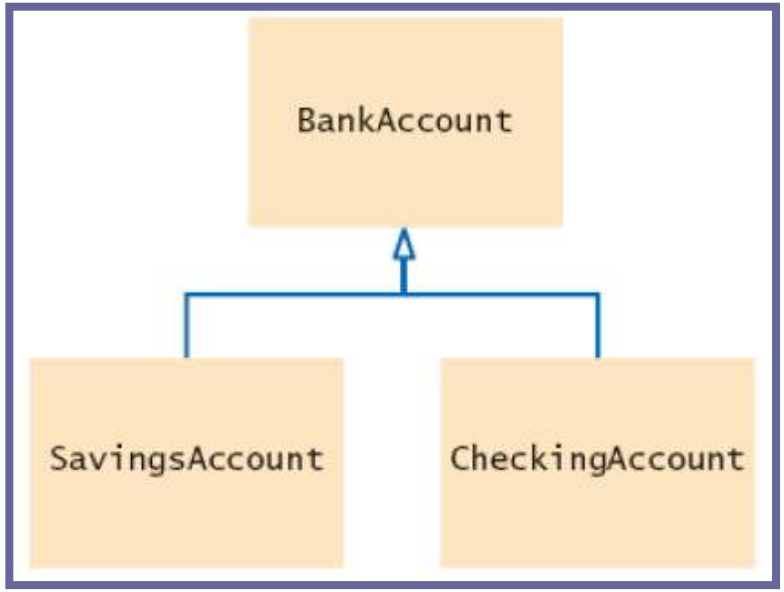


- **La superclasse `JComponent` definisce i metodi `getWidth`, `getHeight`**
- **La classe `AbstractButton` aggiunge i metodi set/get per text e icon**

Esempio

Il Bank account può essere estesa in altre classi:

1. Checking account: nessun interesse; poche transazioni senza costo al mese; transazioni aggiuntive hanno un costo
2. Savings account: offre un interesse che si somma mensilmente



Esempio

- La classe BankAccount è estesa per gestire un numero massimo di transazioni free, a cui fa seguito la deduzione di un dato costo per transazione

```
public class CheckingAccount extends BankAccount {
   public void deposit(double amount) {. . .}
   public void withdraw(double amount) {. . .}
   public void deductFees() {. . .} // new method
   private int transactionCount;  //new instance field
}
```

Il metodo deposit è un overriding del metodo deposit della supeclasse

I campi ereditati sono privati

```
public void deposit(double amount)
{
   transactionCount++;
   // Now add amount to balance
   super.deposit(amount);
}
```

- Una sottoclasse non ha accesso ai campi privati della superclasse
  - Accesso attraverso l'interfaccia pubblica
- super, per l'invocazione di metodi (e costruttore) della superclasse `super.deposit(amount)`

Nota sintatica

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

Esempio:

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

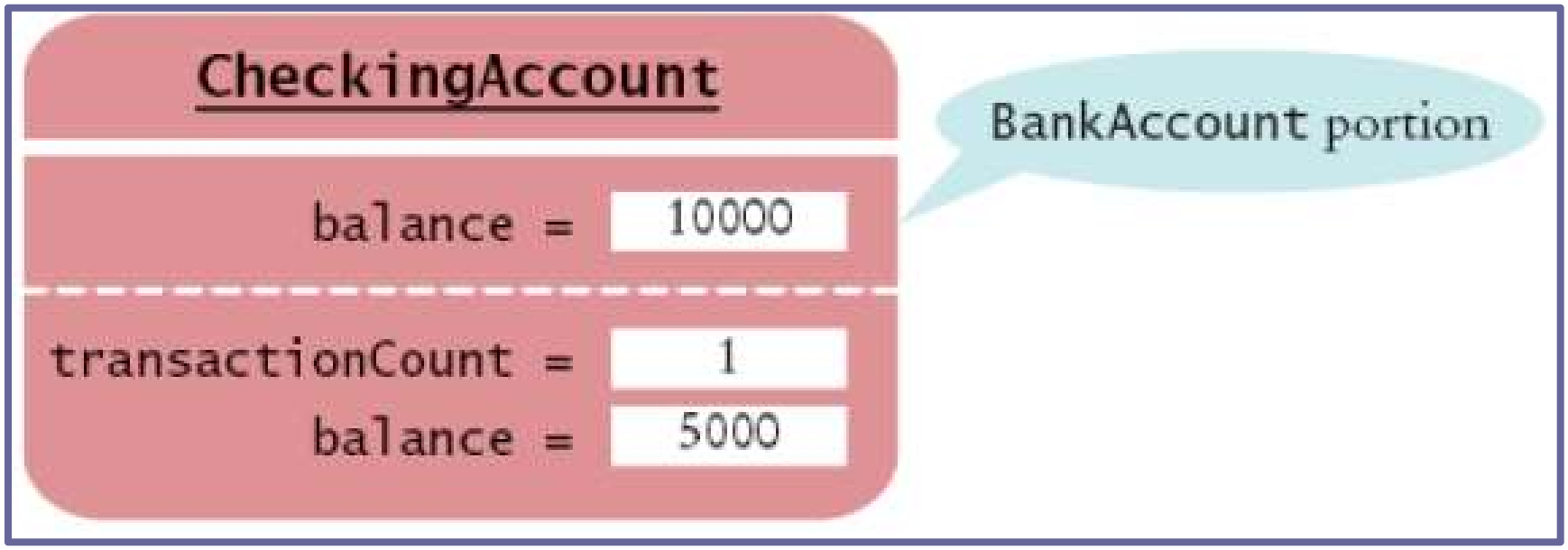
Un errore tipico . . . .

- Poiché una sottoclasse non vede le variabili di stato della superclassse . . . .
- . . . Le aggiungiamo anche alla sottoclasse:

```
public class CheckingAccount extends BankAccount
{
   public void deposit(double amount)
   {
      transactionCount++;
      balance = balance + amount;
   }
   . . .
   private double balance; // Don't
}
```

Shadowing

- Compila correttamente, ma i valori di balance NON vengono aggiornati



Istanziazione di una sottoclasse

- `super` senza il nome di un metodo invoca il costruttore della superclasse

```
public class CheckingAccount extends BankAccount {
   public CheckingAccount(double initialBalance){
      // Construct superclass
      super(initialBalance);
      // Initialize transaction count
      transactionCount = 0;
   }
   . . .
}
```

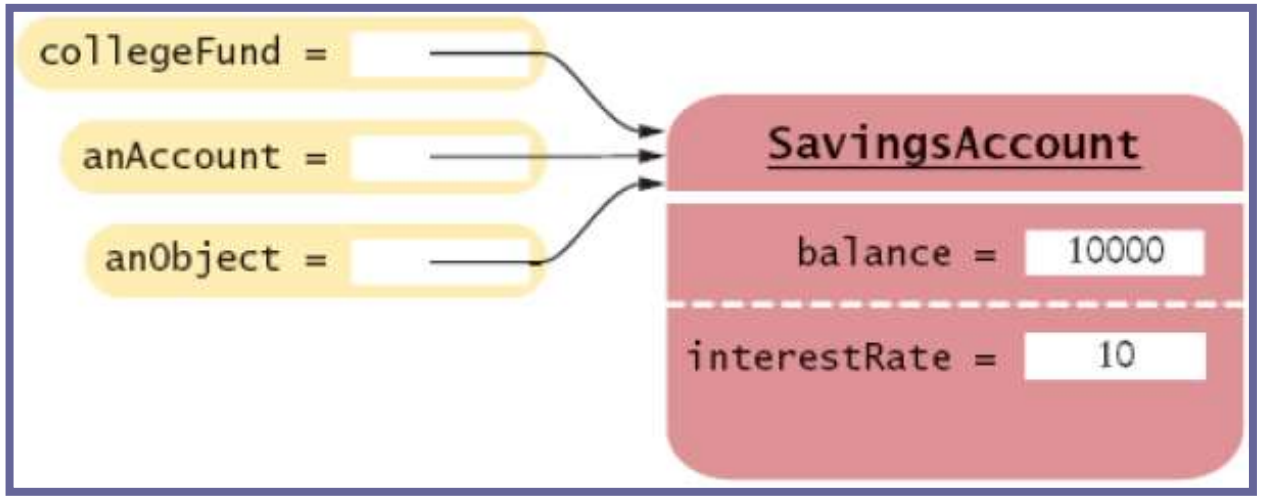
Istanziazione di un oggetto della sottoclasse

- `super` senza il nome di un metodo deve essere il primo statement nel corpo del costruttore della sottoclasse
- Se assente
  - viene invocato il default constructor della superclasse se presente (i.e. no parameters)
  - altrimenti si ottiene un compiler error

Equivalenze di tipo e conversioni

- E' sempre lecito trattare un oggetto di una sottoclasse come oggetto della superclasse

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



Equivalenze di tipo

- Se visto come oggetto della superclasse, non tutti i metodi sono disponibili

```
BankAccount anAccount = collegeFund;
...
anAccount.deposit(1000); // OK
anAccount.addInterest();
/* No--not a method of the class
to which anAccount belongs */
```

Conversioni

- A volte può essere necessario convertire un oggetto della superclasse in uno di una sottoclasse, attraverso l'operazione di cast

```
BankAccount anAccount = new SavingAccount(...);
...
SavingAccount aSavingAccount=(SavingAccount)anAccount;
aSavingAccount.addInterest();
```

- Questo cast è in generale molto pericoloso !

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

Conversioni

- Per alcuni casi particolari si può usare l'operatore **`instanceof`** per verificare l'effettiva classe di appartenenza di un oggetto

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

Ancora sul polimorfismo

Abbiamo visto che:

- In Java, il tipo di una variabile non determina completamente il tipo dell'oggetto referenziato

```
BankAccount aBankAccount = new SavingsAccount(1000);
// aBankAccount holds a reference to a SavingsAccount
```

Come visto con l'uso delle interface:

- Le invocazioni ad un metodo sono determinate guardando al tipo dell'oggetto referenziato, non al tipo della variabile riferimento

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

   // Calls "deposit" from CheckingAccount
```

Polimorfismo

- Il compilatore deve comunque poter verificare che l'invocazione sia lecita. . .
- Questo avviene attraverso i metodi che sono dichiarati nella classe che rappresenta il tipo della variabile di riferimento

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

Access Control

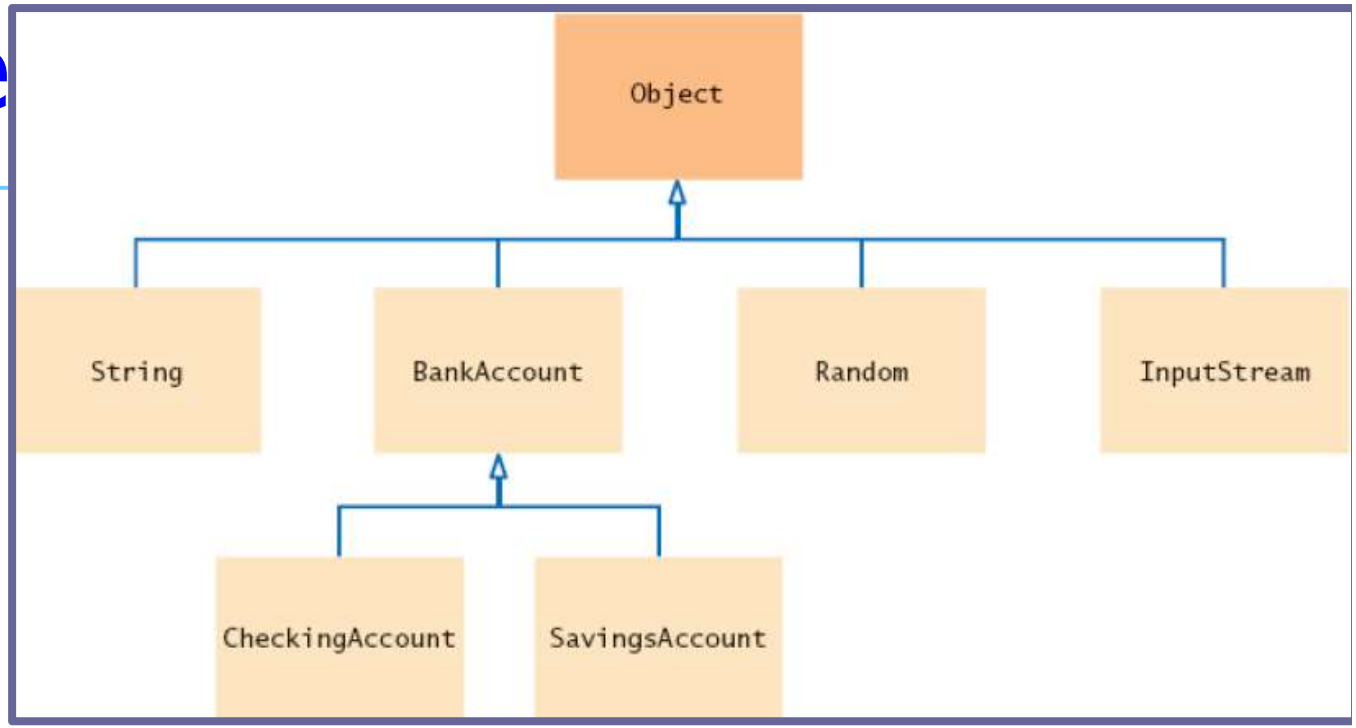
Aumenta il numero di controllo della visibilità

- In Java, ci sono tre livelli di controllo della visibilità per le variabili di istanza ed i metodi di una classe:
  - `public` access
    - I campi possono essere usati dai metodi di tutte le classi
  - `private` access
    - Possono essere usati solo dalla classe che li dichiara
  - `protected` access
    - Sono visibili solo alla classe ed alle sue sottoclassi
- In realtà esiste un quarto livello:
  - package access
    - Se non indicate niente, la visibilità è limitata al package di appartenenza

Raccomandazioni !!!

- Variabili istanza e static: Sempre `private` (o `protected`)
- Con la sola eccezione delle costanti
  - `public static` final constants
- Metodi: `public` o `private`
- Classi e interfacce: `public` o package

La classe Object



La gerarchia spiega perché alcuni metodi della classe Object devono essere riscritti nella definizione di una nuova classe :

- String toString()
- boolean equals(Object otherObject)
- Object clone()

Essi devono essere riscritti per adattarli alle proprie esigenze

Factoring

- L'ereditarietà può essere anche usata per mettere a fattor comune un comportamento condiviso a due o più classi in una singola superclasse
  - Non si incontra spesso in programmi piccoli
- Un esempio: un sistema di inventario
  - Obiettivi (Lens)
  - Borsa per fotocamera (PhotoBag)
  - Macchine fotografiche (Cameras)

Proprietà

- Lens
  - Focal length
  - Zoom/ fixed lens
- PhotoBag
  - Width
  - Heigth
  - Bag colour
- Camera
  - Lens included?
  - Maximum shutter speed
  - Body color

Proprietà comuni a tutti gli item

- Description
- Inventory ID
- Quantity on hand
- Price

Classe Lens

```
class Lens {
      Lens(...) {...} // Constructor
      String getDescription() {return description;};
      int getQuantityOnHand() {return quantityOnHand;}
      int getPrice() {return price;}
      ...
      // Methods specific to Lens class
      ...
      String description;
      int inventoryNumber;
      int quantityOnHand;
      int price;
      boolean isZoom;
      double focalLength;
}
```

Classe PhotoBag

```
class PhotoBag {
      PhotoBag(...) {...} // Constructor
      String getDescription() {return description;};
      int getQuantityOnHand() {return quantityOnHand;}
      int getPrice() {return price;}
      ...
      // Methods specific to Film class
      ...
      String description;
      int inventoryNumber;
      int quantityOnHand;
      int price;
      double width;
      double height;
      double colour;
}
```

Classe Camera

```
class Camera {
      Camera(...) {...} // Constructor
      String getDescription() {return description;};
      int getQuantityOnHand() {return quantityOnHand;}
      int getPrice() {return price;}
      ...
      // Methods specific to Camera class
      ...
      String description;
      int inventoryNumber;
      int quantityOnHand;
      int price;
      boolean hasLens;
      int maxShutterSpeed;
      String bodyColor;
}
```

Fattorizzazione

- Ridondanza nelle tre classi precedenti
- Ogni classe in realtà sta modellando due entità
  - Un generico inventory item
  - Uno specifico item - lens, film, camera

- OOP è responsibility-driven programming!!
- Dividere le responsabilità

La superclasse

```
class InventoryItem {
  InventoryItem(…) {…}
  String getDescription() {…}
  int inventoryID() {…}
  int getQtyOnHand() {…}
  int getPrice() {…}

  String description;
  int inventoryNumber;
  int qtyOnHand;
  int price;
}
```

Le tre sottoclassi

- Il codice della classe Lens

```
class Lens extends InventoryItem {
      Lens(...) {...}
       ...
      // Methods specific to Lens class
       ...
      boolean isZoom;
      double focalLength;
   }
```

Gerarchia di classi

```
InventoryItem [] invarr = new InventoryItem[ 3];
invarr[ 0] = new Lens(…);
invarr[ 1] = new BagPhoto(…);
invarr[ 2] = new Camera(…);
for (int i = 0; i < invarr.length; i++)
    System.out.println(invarr[i].getDescription() + ": " +
                       invarr[i].getQtyOnHand() +  "available");
```

Lavorare con la gerarchia di classi

```
InventoryItem [] invarr = new InventoryItem[3];
invarr[ 0] = new Lens(…);
invarr[ 1] = new BagPhoto(…);
invarr[ 2] = new Camera(…);
for (int i = 0; i < invarr. length; i++)
     invarr[i].print();
```

```
class InventoryItem { ...
     public void print(){
          System.out.println(description);
          System.out.println(inventoryNumber);
          System.out.println(qtyOnHand);
          System.out.println(price);
     ...
}
```

Accedere ai dati delle sottoclassi

- Un metodo print nella superclasse è capace di visualizzare solo gli elementi comuni della superclasse
- Come visualizzare i dati dei singoli oggetti ?
  - *focal length* e *zoom* per lens
  - *width*, *heigth* e *colour* per film
- InventoryItem non conosce queste proprietà!!

Usare il polimorfismo

- Aggiugere un metodo print a ogni sottoclasse

```
class Lens extends InventoryItem {
...
   void print() {
   // prints Lens- specific data
   }
...
}
```

- Allo stesso modo per Film e Camera …

Il polimorfismo ora è disponibile

- I metodi delle sottoclassi sovrascrivono il metodo della superclasse e possono essere invocati polimorficamente attraverso la superclasse!!
- Per esempio, il codice:

  InventoryItem inv = new Lens(...);

  inv. print();

  invoca il metodo print della classe Lens.

Rianalizziamo la superclasse

- InventoryItem non modella un vero oggetto, piuttosto un concetto generico
- Non dovrebbe essere permesso creare un'istanza di questa classe
- Il metodo print di InventoryItem esiste solo per essere sovrascritto
- Quindi potremmo non volerlo implementare nella superclasse
- Dovrebbe però essere specificato nella superclasse in modo tale da poter essere invocato
- Infine, il metodo deve essere sovrascritto dalla sottoclasse

Metodi astratti

- Se un metodo dichiarato nella superclasse non è implementato in questa, lo si dichiara come abstract

```
abstract void print();
```

- La keword abstract nell'intestazione di un metodo
  - specifica che il metodo è dichiarato ma non è implementato nella classe
- Non viene fornita l'implementazione (body) del metodo
- Il metodo deve essere implementato nelle classi che estendono la classe

Classi astratte

- Qualsiasi classe che contiene almeno un metodo astratto
- Deve essere indicato con la keyword abstract nell'intestazione della classe
- InventoryItem così diventa una classe astratta

```
abstract class InventoryItem {
   ...
   abstract void print();
   ...
}
```

Classi astratte

- Non si possono creare istanze di una classe astratta
- Per esempio, il codice

```
InventoryItem inv = new InventoryItem(…);
```

causa un compiler error

- Le variabili reference possono ancora essere dichiarate e ad esse possono ancora essere assegnate istanze di una sottoclasse

```
InventoryItem inv = new Lens(...);
```

Classi astratte vs interfacce

- Talvolta è utile specificare l'esistenza di comportamenti, ma nessuna implementazione
  - Questo può essere compiuto definendo una classe astratta che contiene metodi astratti
  - Oppure una interfaccia
- Le classi astratte forzano il comportamento in una gerarchia
- Le interfacce no, in quanto realizzano una diversa relazione

Esercizio

Una azienda ha organizzato l'archivio anagrafico dei propri dipendenti in un file: le informazioni memorizzate sono differenti a seconda che si tratti di dirigente, o di impiegati ed operai. In particolare, per tutte e due le tipologie di dipendenti vengono memorizzati:

- Codice fiscale
- Nome
- Cognome

Inoltre, per i dirigenti si memorizzano:

- Area di responsabilità (es. Personale, Contabilità, "Relazioni Esterne" …)
- Paga oraria

Per il resto del personale vengono memorizzate:

- Funzione (es. Segretario, Usciere, "Operaio specializzato" …)
- Livello (es. I, .. VI)
- Paga Oraria

Esempio

DIR CFSRD63M07F912T Pasquale Rossi Contabilità 75

IMOP CJKRD33M07G912T Antonio Bianchi Operaio II 23

Esercizio

Scrivere un programma di gestione dell'archivio dipendenti

****

Ogni mese il centro contabilità riceve un file, presenze.dat, che contiene le ore lavorate per ogni dipendente. In particolare, file contiene

- Codice fiscale
- Ore lavorate

Scrivere un programma che calcoli e stampi la paga di tutti i dipendenti, tenendo conto che:

- Per gli impiegati le prime 165 ore sono da considerarsi ordinarie (pagate secondo la paga oraria), ed ogni eventuale ora successiva va considerata straordinario (pagate secondo la paga oraria incrementata del 30%)
- Per i dirigenti NON esiste lo straordinario, e tutte le ore sono pagate secondo la paga oraria.