

Lezione XXV
Lu 19-Nov-2007

Ereditarietà

L'ereditarietà

- L'*ereditarietà* è uno dei principi basilari della programmazione orientata agli oggetti
 - insieme all'*incapsulamento* e al *polimorfismo*
- L'ereditarietà è il paradigma che consente il *riutilizzo del codice*
 - si usa quando si deve realizzare una classe ed è già disponibile un'altra classe che rappresenta *un concetto più generale*



L'ereditarietà

- Supponiamo di voler realizzare una classe **SavingsAccount** per rappresentare un *conto bancario di risparmio*, con un tasso di interesse annuo determinato al momento dell'apertura e 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**

BankAccount

```
public class BankAccount
{
    public BankAccount()
    {
        balance = 0;
    }
    public void deposit(double amount)
    {
        balance = balance + amount;
    }
    public void withdraw(double amount)
    {
        balance = balance - amount;
    }
    public double getBalance()
    {
        return balance;
    }
    private double balance;
}
```

SavingsAccount

```
public class SavingsAccount
{
    public SavingsAccount(double rate)
    {
        balance = 0;
        interestRate = rate;
    }
    public void addInterest()
    {
        deposit(getBalance() * interestRate / 100);
    }
    private double interestRate;
    public void deposit(double amount)
    {
        balance = balance + amount;
    }
    public void withdraw(double amount)
    {
        balance = balance - amount;
    }
    public double getBalance()
    {
        return balance;
    }
    private double balance;
}
```

Riutilizzo del codice

- Come avevamo previsto, buona parte del codice scritto per **BankAccount** ha potuto essere *copiato* nella definizione della classe **SavingsAccount**
 - 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

- ❑ Copiare il codice è già un risparmio di tempo, però non è sufficiente
- ❑ 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...
 - fonte di errori...
- ❑ Cosa succede se non abbiamo a disposizione il codice sorgente, ma solo il bytecode?
 - non possiamo copiare la classe **BankAccount**...
 - e quindi la riscrittura di **SavingsAccount** deve essere completa
- ❑ L'**ereditarietà** risolve questi problemi

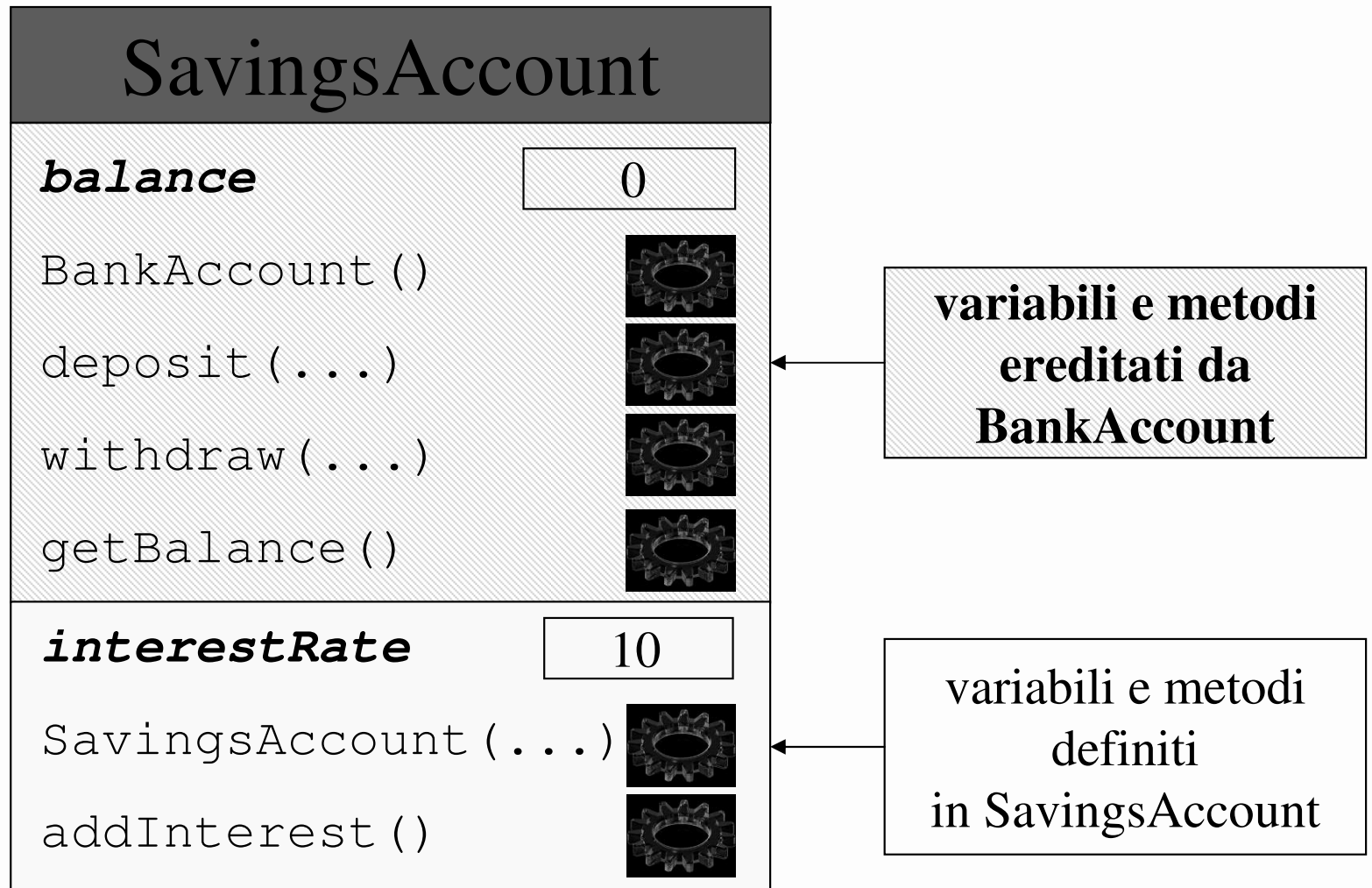
SavingsAccount

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

- ❑ Dichiariamo che **SavingsAccount** è una classe *derivata* da **BankAccount** (**extends**)
 - ne eredita tutte le caratteristiche
 - Variabili di esemplare
 - Variabili statiche
 - Metodi pubblici e privati
 - specifichiamo soltanto le peculiarità

SavingsAccount

```
SavingsAccount sAcct = new SavingsAccount(10);
```

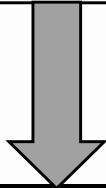


Come usare la classe derivata

- ❑ Oggetti della classe derivata **SavingsAccount** si usano come se fossero oggetti di **BankAccount**, con *qualche proprietà in più*

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

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



330

La superclasse universale **Object**

- ❑ In Java, ogni classe che non deriva da nessun'altra deriva *implicitamente* dalla *superclasse universale del linguaggio*, che si chiama **Object**
- ❑ Quindi, **SavingsAccount** deriva da **BankAccount**, che a sua volta deriva da **Object**
- ❑ **Object** ha alcuni metodi, che vedremo più avanti (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

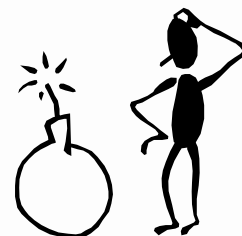


□ Sintassi:

```
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**

Confondere superclassi e sottoclassi

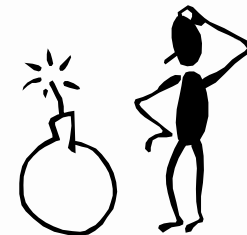


- Dato che oggetti di tipo **SavingsAccount** sono
 - un’*estensione* di oggetti di tipo **BankAccount**
 - più “grandi” di oggetti di tipo **BankAccount**, perché hanno una variabile di esemplare in più
 - più “abili” di oggetti di tipo **BankAccount**, perché hanno un metodo in più

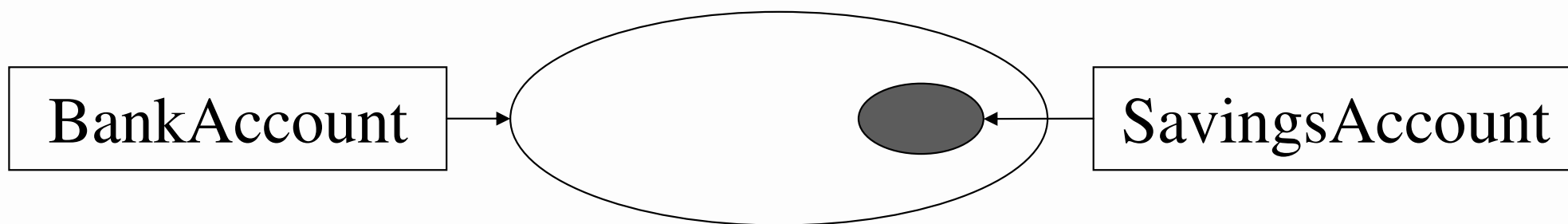
perché mai **SavingsAccount** si chiama *sottoclasse* e non *superclasse*?

- è facile fare confusione
- verrebbe forse spontaneo usare i nomi al contrario...

Confondere superclassi e sottoclassi

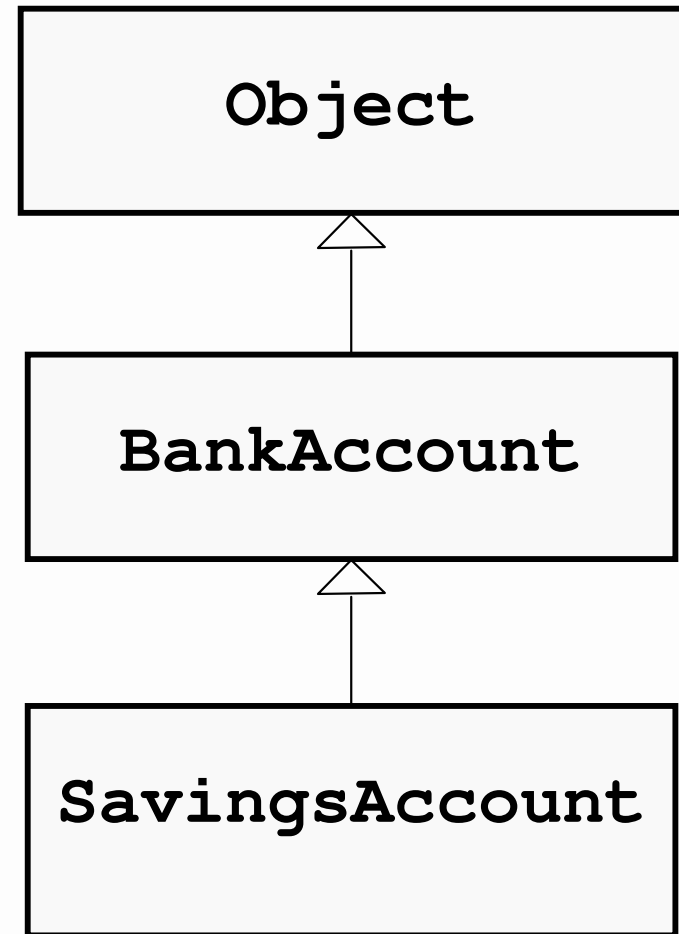


- ❑ I termini superclasse e sottoclasse derivano dalla *teoria degli insiemi*
- ❑ I conti bancari di risparmio (gli oggetti di tipo **SavingsAccount**) costituiscono un *sottoinsieme* dell'insieme di tutti i conti bancari (gli oggetti di tipo **BankAccount**)



Rappresentazione UML diagramma di classe

- ❑ Unified Modeling Language: notazione per l'analisi e la progettazione orientata agli oggetti
- ❑ Una classe è rappresentata con un rettangolo contenente il nome della classe
- ❑ Il rapporto di ereditarietà fra classi è indicato con un segmento terminante con una freccia
- ❑ La freccia a triangolo vuoto punta alla superclasse



Ereditare metodi

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 esiste: **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 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 intestazione* di quello definito nella superclasse
 - tale metodo *prevale* (*overrides*) su quello della superclasse quando viene invocato con un oggetto della sottoclasse

Sovrascrivere un metodo: Esempio

- ❑ Vogliamo modificare la classe **SavingsAccount** in modo che ogni operazione di versamento abbia un costo (fisso) **FEE**, che viene automaticamente addebitato sul conto

```
public class SavingsAccount extends
BankAccount
{
    ...
    private final static double FEE = 2.58;
    // euro
}
```

- ❑ 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`

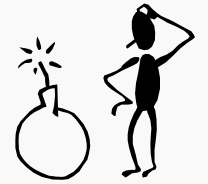
```
public class SavingsAccount
    extends BankAccount
{
    ...
    public void deposit(double amount)
    {
        withdraw(FEE);
        ... // aggiungi amount a balance
    }
}
```

- ❑ 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**
 - *nulla cambia* per oggetti di tipo **BankAccount**, ovviamente

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)
    {
        withdraw(FEE);
        deposit(amount); // NON FUNZIONA
    }
}
```



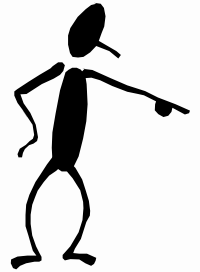
- ❑ Così però non funziona, perché il metodo diventa ricorsivo con 21 ricorsione infinita!!! (manca il caso base...)

Sovrascrivere un metodo: Esempio

- ❑ Ciò che dobbiamo fare è *invocare il metodo deposit() di BankAccount*
- ❑ 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)
    {
        withdraw(FEE);
        // invoca deposit della superclasse
        super.deposit(amount);
    }
}
```

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

**Sovrascrivere il metodo
toString()**

Sovrascrivere il metodo toString()

- Abbiamo già visto che la classe **Object** è la superclasse universale, cioè tutte le classi definite in Java ne ereditano implicitamente i metodi, tra i quali

public String toString()

- L'invocazione di questo metodo per qualsiasi oggetto ne restituisce la cosiddetta *descrizione testuale standard* **BankAccount@111f71**

– *il nome della classe* seguito dal carattere @ e dall'*indirizzo dell'oggetto in memoria*

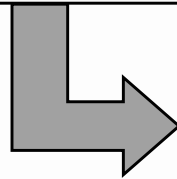
Sovrascrivere il metodo toString()

- ❑ Il fatto che tutte le classi siano derivate (anche indirettamente) da **Object** e che **Object** definisca il metodo **toString()** consente il funzionamento del metodo **println()** di **PrintStream**
 - passando un oggetto di qualsiasi tipo a **System.out.println()** si ottiene la visualizzazione della descrizione testuale standard dell'oggetto
 - come funziona **println()**?
 - **println()** invoca semplicemente **toString()** dell'oggetto, e l'invocazione è possibile perché tutte le classi hanno il metodo **toString()**, eventualmente ereditato da **Object**

Sovrascrivere il metodo toString()

- ❑ Se tutte le classi hanno già il metodo **toString()**, ereditandolo da **Object**, perché lo dovremmo sovrascrivere, ridefinendolo?

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



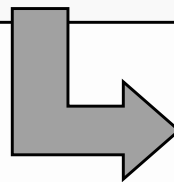
```
BankAccount@111f71
```

- ❑ Perché in generale la descrizione testuale standard non è particolarmente utile

Sovrascrivere il metodo toString()

- ❑ Sarebbe molto più comodo, ad esempio per verificare il corretto funzionamento del programma, 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**

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



BankAccount[balance=1500]

Sovrascrivere il metodo toString()

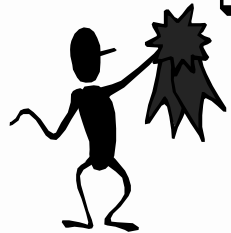
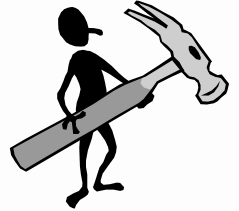
- ❑ Il metodo **toString()** di una classe viene invocato implicitamente anche quando si concatena un oggetto della classe con una stringa

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

- ❑ Questa concatenazione è sintatticamente corretta, come avevamo già visto per i tipi di dati numerici, e viene interpretata dal compilatore come se fosse stata scritta così

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

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 non costanti della classe
- ❑ Questo stile di programmazione è molto utile per il debugging ed è usato nella libreria standard

Costruzione della sottoclasse

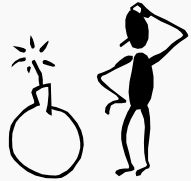
Costruzione della sottoclasse

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

- ❑ 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

Costruzione 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**

Costruzione 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

Costruzione della sottoclasse

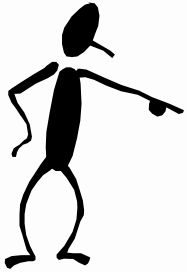
- ❑ Dato che la variabile **balance** è gestita dalla classe **BankAccount**, bisogna *delegare* a tale classe il compito di inicializzarla
- ❑ La classe **BankAccount** ha già un costruttore creato appositamente per gestire l'apertura di un conto con saldo diverso da zero
 - invochiamo tale costruttore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

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

Costruzione della sottoclasse

- ❑ In realtà, un'invocazione di **super()** viene sempre eseguita quando la JVM costruisce una sottoclasse
 - se la chiamata a **super(...)** non è indicata esplicitamente dal programmatore, il compilatore inserisce automaticamente l'invocazione di **super()** *senza parametri*
 - questo avviene, ad esempio, nel caso del primo costruttore di **SavingsAccount**
- ❑ L'invocazione *esplicita* di **super(...)**, se presente, *deve essere il primo enunciato del costruttore*

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: deve essere il primo enunciato del costruttore
- ❑ Nota: se non è indicato esplicitamente, viene invocato implicitamente **super**() senza parametri

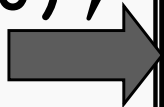
Conversione fra riferimenti

BankAccount: transferTo()

□ Aggiungiamo un metodo a BankAccount

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

```
BankAccount acct1 = new BankAccount(500);
BankAccount acct2 = new BankAccount();
acct1.transferTo(acct2, 200);
System.out.println(acct1.getBalance());
System.out.println(acct2.getBalance());
```

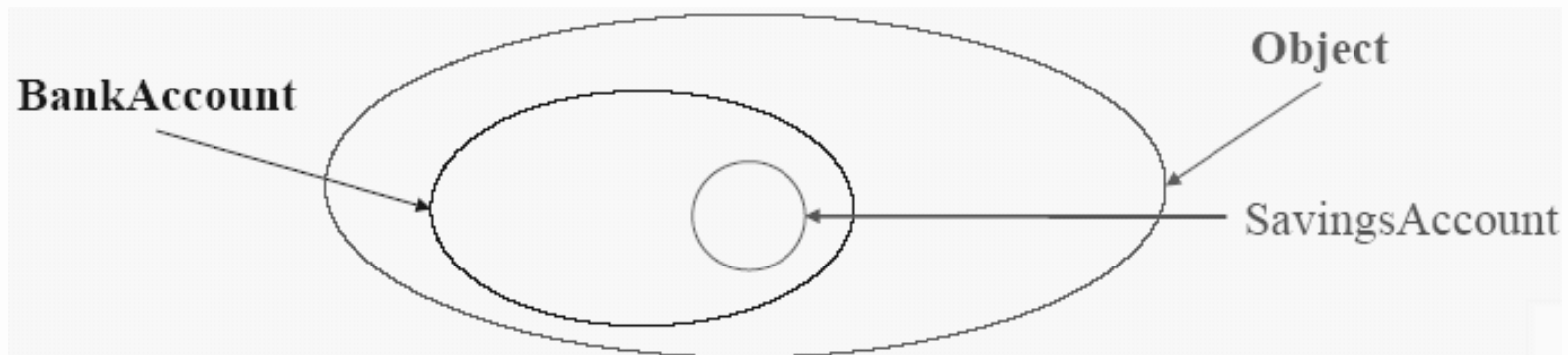


300
200

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
 - *un riferimento a un oggetto di una classe derivata può essere assegnato a una variabile oggetto del tipo di una sua superclasse*

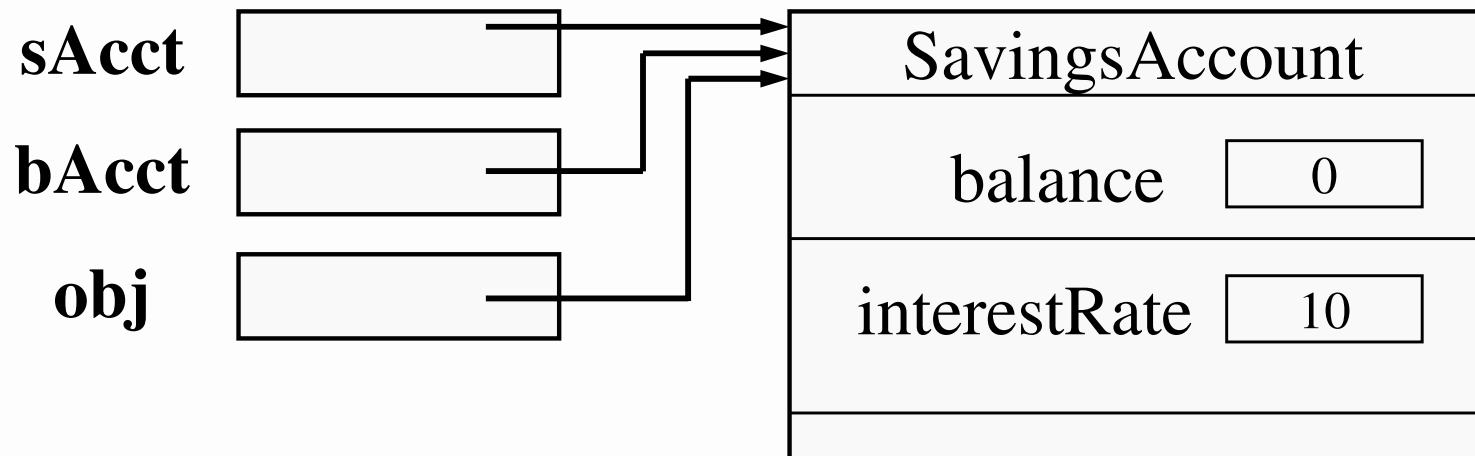
```
SavingsAccount sAcct = new SavingsAccount (10) ;  
BankAccount bAcct = sAcct;  
Object obj = sAcct;
```



Conversione fra riferimenti

```
SavingsAccount sAcct = new SavingsAccount(10);  
BankAccount bAcct = sAcct;  
Object obj = sAcct;
```

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



Conversione fra riferimenti

```
SavingsAccount sAcct = new SavingsAccount(10);  
BankAccount bAcct = sAcct;  
bAcct.deposit(500); // OK
```

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

```
bAcct.addInterest();
```



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

```
1 error
```

Conversione fra riferimenti

- ❑ Quale scopo può avere questo tipo di conversione?
 - per quale motivo vogliamo trattare un oggetto di tipo **SavingsAccount** come se fosse di tipo **BankAccount**?

```
BankAccount other = new BankAccount(1000);  
SavingsAccount sAcct = new SavingsAccount(10);  
BankAccount bAcct = sAcct;  
other.transferTo(bAcct, 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 usare senza doverlo modificare!

```
public void transferTo(BankAccount other,  
                      double amount)
```

Conversione fra riferimenti

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

```
BankAccount other = new BankAccount(1000);  
SavingsAccount sAcct = new SavingsAccount(10);  
other.transferTo(sAcct, 500);
```

- ❑ Il compilatore sa che il metodo **transferTo()** richiede un riferimento di tipo **BankAccount**, quindi
 - controlla che **sAcct** sia un riferimento a una classe derivata da **BankAccount**
 - effettua la conversione automaticamente

Lezione XXVI
Ma 20-Nov-2007

**Conversione fra
riferimenti**

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 in un riferimento di tipo **Object**

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

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

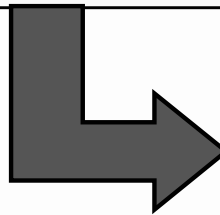
❑ Ma un esemplare di **String** è anche un esemplare di **Object**...

- come viene scelto il giusto metodo, tra quelli sovraccarichi, quando si invoca **println()** con una stringa?
- nella conversione automatica dei riferimenti durante l’invocazione di metodi, il compilatore cerca sempre di “fare il minor numero di conversioni”
- viene usato, quindi, il metodo “più specifico”

Conversione fra riferimenti

- ❑ La conversione inversa non può invece avvenire automaticamente

```
BankAccount bAcct = new BankAccount(1000);  
SavingsAccount sAcct = bAcct;
```



- ❑ *Ma ha senso cercare di effettuare una conversione di questo tipo?*

```
incompatible types  
found    : BankAccount  
required:  
SavingsAccount  
        sAcct = bAcct;  
                ^  
1 error
```


Conversione fra riferimenti

- ❑ La conversione di un riferimento a *superclasse* in un riferimento a *sottoclasse* ha senso soltanto se, per le specifiche dell'algoritmo, siamo sicuri che il riferimento a *superclasse* punta in realtà ad un oggetto della *sottoclasse*
 - richiede un *cast* esplicito

```
SavingsAccount sAcct = new SavingsAccount(1000);  
BankAccount bAcct = sAcct;  
...  
SavingsAccount sAcct2 = (SavingsAccount) bAcct;  
// se in fase di esecuzione bAcct non punta  
// effettivamente a un oggetto SavingsAccount  
// l'interprete lancia ClassCastException
```



Polimorfismo

Polimorfismo

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

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

- ❑ Quando si invoca **deposit()**, quale metodo viene invocato?
 - il metodo **deposit()** definito in **BankAccount**?
 - il metodo **deposit()** *ridefinito* in **SavingsAccount**?

Polimorfismo

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

□ Si potrebbe pensare

- **acct** 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 **acct** contiene un riferimento a un oggetto che, *in realtà*, è di tipo **SavingsAccount**!

E l'interprete Java lo sa (il compilatore no)

- secondo la semantica di Java, viene invocato il metodo **deposit()** di **SavingsAccount**

Polimorfismo

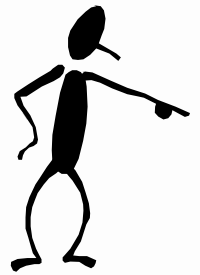
- ❑ Questa semantica si chiama *polimorfismo* ed è caratteristica dei linguaggi OOP

l'invocazione di un metodo in runtime è sempre determinata dal tipo dell'oggetto effettivamente 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*, dipendentemente dall'oggetto usato

Polimorfismo

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

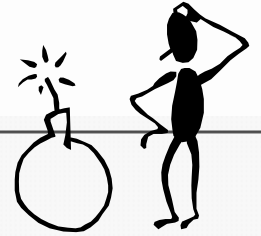


L'operatore instanceof

- ❑ Sintassi: `variabileOggetto instanceof NomeClasse`
- ❑ Scopo: è un operatore booleano che restituisce **true** se e solo se la *variabileOggetto* contiene un riferimento ad un oggetto che è un esemplare della classe *NomeClasse* (o di una sua sottoclasse)
 - in tal caso l'assegnamento di *variabileOggetto* ad una variabile di tipo *NomeClasse* **NON** lancia l'eccezione **ClassCastException**
- ❑ Nota: il risultato non dipende dal tipo dichiarato per la *variabileOggetto*, ma dal tipo dell'oggetto a cui la variabile si riferisce effettivamente al momento dell'esecuzione

Errori tipici con i metodi e con i costruttori

Invocare un metodo di esemplare senza un oggetto



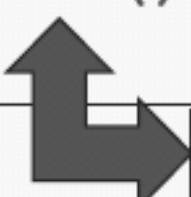
```
public class Program
{
    public static void main(String[] args)
    {
        BankAccount account = new BankAccount();
        deposit(500); // ERRORE
    }
}
```

- ❑ In questo caso il compilatore non è in grado di compilare la classe **Program**, perché non sa a quale oggetto applicare il metodo **deposit**, che non è un metodo statico e quindi richiede un riferimento a un oggetto da usare come parametro implicito

Invocare un metodo di esemplare senza un oggetto

- ❑ Ci sono, però, casi in cui è ammesso (ed è *pratica molto comune!*) invocare metodi non statici senza specificare un riferimento ad oggetto
- ❑ Questo accade quando si invoca *un metodo non statico* di una classe da *un altro metodo non statico della stessa classe*
 - viene usato il parametro implicito **this**

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

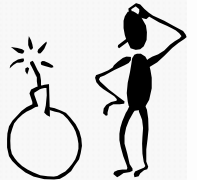


```
this.getBalance()
```

Tentare di ricostruire un oggetto

- ❑ A volte viene la tentazione di *invocare un costruttore su un oggetto già costruito* con l'obiettivo di *riportarlo alle condizioni iniziali*

```
BankAccount account = new BankAccount();  
account.deposit(500);  
account.BankAccount(); // NON FUNZIONA!
```



- ❑ Il messaggio d'errore è un po' strano... il compilatore cerca un *metodo* **BankAccount**, non un *costruttore*, e naturalmente non lo trova!

```
cannot resolve symbol  
symbol   : method BankAccount()  
location : class BankAccount
```

Tentare di ricostruire un oggetto

- ❑ Un costruttore crea un *nuovo oggetto* con prefissate condizioni iniziali
 - un costruttore può essere invocato *soltanto* con l'operatore **new**
- ❑ La soluzione è semplice
 - *assegnare un nuovo oggetto alla variabile oggetto che contiene l'oggetto che si vuole “ricostruire”*

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

Ordinamento e ricerca di oggetti

Ordinamento di oggetti

- ❑ Abbiamo visto algoritmi di ordinamento efficienti e ne abbiamo verificato il funzionamento con array di numeri, ma spesso si pone il problema di *ordinare dati più complessi*
 - ad esempio, ordinare stringhe
 - in generale, ordinare oggetti
- ❑ Vedremo ora che si possono usare gli stessi algoritmi, a patto che gli oggetti da ordinare siano tra loro *confrontabili*

Ordinamento di oggetti

- ❑ Per ordinare numeri è, ovviamente, necessario effettuare confronti tra loro
 - Operatori di confronto: $==$, $!=$, $>$, $<$, ...
- ❑ La stessa affermazione è vera per oggetti
 - *per ordinare oggetti è necessario effettuare confronti tra loro*
- ❑ C'è però una differenza
 - confrontare numeri ha un significato ben definito dalla matematica
 - *confrontare oggetti ha un significato che dipende dal tipo di oggetto*, e a volte può non avere significato alcuno...

Ordinamento di oggetti

- ❑ Confrontare oggetti ha un significato che dipende dal tipo di oggetto
 - quindi *la classe che definisce l'oggetto deve anche definire il significato del confronto*
- ❑ Consideriamo la classe **String**
 - essa definisce il metodo **compareTo()** che attribuisce un significato ben preciso all'ordinamento tra stringhe
 - l'ordinamento lessicografico
- ❑ Possiamo quindi riscrivere, ad esempio, il metodo **selectionSort** per ordinare stringhe invece di ordinare numeri, *senza cambiare l'algoritmo*


```
public class ArrayAlgorithms
{public static void selectionSort(String[] a)
  { for (int i = 0; i < a.length - 1; i++)
    {   int minPos = findMinPos(a, i);
        if (minPos != i)    swap(a, minPos, i);
    }
  }
private static void swap(String[] a,int i,int j)
  { String temp = a[i];
    a[i] = a[j];  a[j] = temp;
  }
private static int findMinPos(String[] a,
  int from)
  {   int pos = from;
    for (int i = from + 1; i < a.length; i++)
      if (a[i].compareTo(a[pos]) < 0) pos = i;
    return pos;
  }
}
```

Ordinamento di oggetti

- ❑ Allo stesso modo si possono riscrivere tutti i metodi di ordinamento e ricerca visti per i numeri interi e usarli per le stringhe
- ❑ Ma come fare per altre classi?
 - possiamo ordinare oggetti di tipo **BankAccount** in ordine di saldo crescente?
 - bisogna definire un metodo **compareTo()** nella classe **BankAccount**
 - bisogna riscrivere i metodi perché accettino come parametro un array di **BankAccount**

```

public class ArrayAlgorithms
{
    public static void selectionSort(BankAccount[] a)
    {
        for (int i = 0; i < a.length - 1; i++)
        {
            int minPos = findMinPos(a, i);
            if (minPos != i)
                swap(a, minPos, i);
        }
    }
    private static void swap(BankAccount[] a, int i, int j)
    {
        BankAccount temp = a[i];
        a[i] = a[j];
        a[j] = temp;
    }
    private static int findMinPos(BankAccount[] a, int from)
    {
        int pos = from;
        for (int i = from + 1; i < a.length; i++)
            if (a[i].compareTo(a[pos]) < 0) pos = i;
        return pos;
    }
    ...
}

```

❑ Non possiamo certo usare questo approccio per qualsiasi classe, *deve esserci un metodo migliore!*

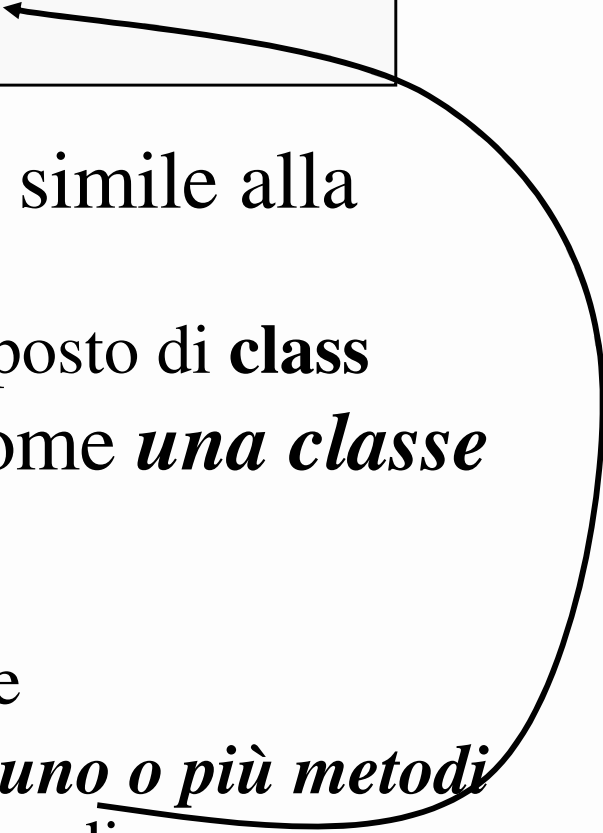
Realizzare una proprietà astratta

- ❑ Quello che deve fare una classe perché i suoi esemplari possano essere ordinati è
 - definire un metodo adatto a confrontare esemplari della classe, con lo stesso comportamento di **compareTo()**
- ❑ Gli oggetti della classe diventano *confrontabili*
 - gli algoritmi di ordinamento e ricerca *non hanno bisogno di conoscere alcun particolare degli oggetti*
 - è sufficiente che gli oggetti siano tra loro *confrontabili*
- ❑ Per *definire un comportamento astratto* si usa in Java la definizione di un' *interfaccia*, che deve essere *realizzata* (“implementata”) da una classe che dichiari di avere tale comportamento

Realizzare una proprietà astratta: interfacce

Realizzare una proprietà astratta

```
public interface Comparable
{
    int compareTo(Object other);
}
```



- ❑ La definizione di un'interfaccia è simile alla definizione di una classe
 - si usa la parola chiave **interface** al posto di **class**
- ❑ Un'interfaccia può essere vista come *una classe ridotta*, perché
 - non può avere costruttori
 - non può avere variabili di esempio
 - *contiene soltanto le intestazioni di uno o più metodi non statici*, ma non può definirne il codice
 - i metodi sono *implicitamente public*

Realizzare una proprietà astratta

- ❑ Se una classe dichiara di realizzare concretamente un comportamento astratto definito in un'interfaccia, deve *implementare* l'interfaccia
 - oltre alla dichiarazione, **DEVE** *definire i metodi specificati nell'interfaccia, con la stessa firma*

```
public class BankAccount implements Comparable
{
    ...
    public int compareTo(Object other)
    {
        // notare che viene sempre ricevuto un
        // Object
        BankAccount acct = (BankAccount) other;
        if (balance < acct.balance) return -1;
        if (balance > acct.balance) return 1;
        return 0;
    }
}
```

Realizzare l'interfaccia Comparable

□ Il metodo *`int compareTo(object obj)`* deve definire una *relazione di ordine totale* caratterizzata dalle seguenti proprietà

– *riflessiva*:

- `x.compareTo(x)` deve restituire zero

– *anti-simmetrica*:

- `x.compareTo(y) <= 0   =>   y.compareTo(x) >= 0`

– *transitiva*:

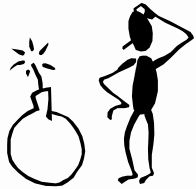
- `x.compareTo(y) <= 0 && y.compareTo(z) <= 0`
- `=> x.compareTo(z) <= 0`

Realizzare l'interfaccia Comparable

□ Il metodo *int compareTo(object obj)* deve:

- ritornare zero se i due oggetti confrontati sono equivalenti nella comparazione
 - `x.compareTo(y)` ritorna zero se `x` e `y` sono “uguali”
 - *if (x.compareTo(y) == 0) ...*
- ritornare un numero intero negativo se il parametro implicito precede nella comparazione il parametro esplicito
 - `x.compareTo(y)` ritorna un numero intero negativo se `x` precede `y`
 - *if (x.compareTo(y) < 0) ...*
- ritornare un numero un numero positivo se il parametro implicito segue nella comparazione il parametro esplicito
 - *if (x.compareTo(y) > 0) ...*

Realizzare una proprietà astratta



- ❑ Non è possibile costruire oggetti da un'interfaccia

```
new Comparable(); // ERRORE DI SINTASSI
```

- ❑ È invece possibile definire riferimenti ad oggetti che realizzano un'interfaccia

```
Comparable c = new BankAccount(10);
```

- ❑ Queste conversioni tra un oggetto e un riferimento a una delle interfacce che sono realizzate dalla sua classe sono automatiche

– *come se l'interfaccia fosse una superclasse*

- ❑ *Una classe estende sempre una sola altra classe, mentre può realizzare più interfacce*

L'interfaccia Comparable

```
public interface Comparable
{
    int compareTo(Object other);
}
```

- ❑ L'interfaccia **Comparable** è definita nel pacchetto **java.lang**, per cui non deve essere importata né deve essere definita
 - la classe **String**, ad esempio, realizza **Comparable**
- ❑ Come può l'interfaccia **Comparable** risolvere il nostro problema di *definire un metodo di ordinamento valido per tutte le classi*?
 - basta definire un metodo di ordinamento che ordini un array di riferimenti ad oggetti che realizzano l'interfaccia **Comparable**, indipendentemente dal tipo

```
public class ArrayAlgorithms
{
    public static void selectionSort(Comparable[] a)
    { for (int i = 0; i < a.length - 1; i++)
        {
            int minPos = findMinPos(a, i);
            if (minPos != i)
                swap(a, minPos, i);
        }
    }
    private static void swap(Comparable[] a, int i, int j)
    {
        Comparable temp = a[i];
        a[i] = a[j];
        a[j] = temp;
    }

    private static int findMinPos(Comparable[] a, int from)
    {
        int pos = from;
        for (int i = from + 1; i < a.length; i++)
            if (a[i].compareTo(a[pos]) < 0)
                pos = i;
        return pos;
    }
}
```

Ordinamento di oggetti

- ❑ Definito un algoritmo per ordinare un array di riferimenti **Comparable**, se vogliamo ordinare un array di oggetti **BankAccount** basta fare

```
BankAccount[] v = new BankAccount[10];  
// creazione dei singoli elementi  
dell'array  
...  
// eventuali modifiche allo stato  
// degli oggetti dell'array  
...  
ArrayAlgorithms.selectionSort(v);
```

- ❑ Dato che **BankAccount** realizza **Comparable**, il vettore di riferimenti viene convertito automaticamente

Ordinamento di oggetti

- ❑ Tutti i metodi di ordinamento e ricerca che abbiamo visto per array di numeri interi possono essere riscritti per array di oggetti **Comparable**, usando le seguenti “traduzioni”

```
if (a < b)
```

```
if (a > b)
```

```
if (a == b)
```

```
if (a != b)
```



```
if (a.compareTo(b) < 0)
```

```
if (a.compareTo(b) > 0)
```

```
if (a.compareTo(b) == 0)
```

```
if (a.compareTo(b) != 0)
```

Lezione XXVII
Me 21-Nov-2007

**Interfaccia Comparabile
parametrica**

Interfaccia Comparable Parametrica

```
public class BankAccount implements Comparable
{ public int compareTo(Object obj)
{
    BankAccount acct = (BankAccount) obj;
    if (balance < acct.balance) return -1;
    if (balance > acct.balance) return 1;
    return 0;
}
}
```

```
BankAccount acct1 = new BankAccount(10);
String s = "ciao";
...
Object acct2 = s; // errore del programmatore!
if (acct1.compareTo(acct2) < 0)
...
```

- ❑ Il compilatore non segnala alcun errore!!!
- ❑ Quando il codice viene eseguito, il metodo `compareTo()` genera **ClassCastException**

Interfaccia Comparable parametrica

- ❑ Sarebbe preferibile che l'errore fosse diagnosticato dal compilatore
- ❑ Che succede se l'eccezione viene generata in un programma che realizza un sistema di controllo? Ad esempio un programma per il controllo del traffico aereo?
- ❑ Java 5.0 fornisce uno strumento comodo l'**interfaccia Comparable parametrica** che permette al compilatore di trovare l'errore
- ❑ Nell'interfaccia Comparable parametrica possiamo
- ❑ definire il tipo di oggetto nel parametro esplicito del metodo compareTo()

Interfaccia Comparable parametrica

```
public interface Comparable<T>
{
    int compareTo(T o);
}
```

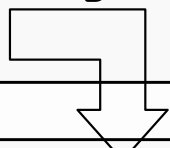
```
public class BankAccount implements
    Comparable<BankAccount>
{
    public int compareTo(BankAccount obj)
    { //non e' necessario il forzamento
        if (balance < obj.balance) return -1;
        if (balance > obj.balance) return 1;
        return 0;
    }
}
```

- ❑ T rappresenta una classe generica
- ❑ Generalmente il tipo generico T e' la classe stessa, come nell'esempio riportato per la classe BankAccount
- ❑ Si puo' evitare di programmare nel metodo compareTo() una conversione forzata

Interfaccia Comparable parametrica

```
public class BankAccount implements
    Comparable<BankAccount>
{
    public int compareTo(BankAccount o)
    { if (balance < o.balance) return -1;
      if (balance > o.balance) return 1;
      return 0;
    }
}
```

```
BankAccount acct1 = new BankAccount(10);
String s = "ciao";
...
Object acct2 = s; // errore del programmatore
if (acct1.compareTo(acct2) < 0)
```



```
compareTo(BankAccount) cannot be applied to
<java.lang.String>
```

❑ L'errore viene intercettato dal compilatore

Ordinamento e ricerche “di libreria”

- ❑ La libreria standard fornisce, per l'ordinamento e la ricerca, alcuni metodi statici in *java.util.Arrays*
 - un metodo **sort()** che ordina array di tutti i tipi fondamentali e array di **Comparable**

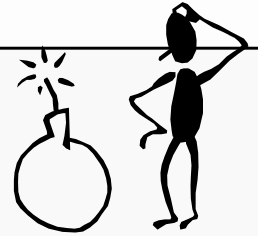
```
String[] ar = new String[10];  
...  
Arrays.sort(ar);
```

- un metodo **binarySearch()** che cerca un elemento in array ordinati di tutti i tipi fondamentali e in array di **Comparable**
 - restituisce la posizione come numero intero
- ❑ *Questi metodi sono da usare nelle soluzioni professionali*
- ❑ *Non potrete usarli nel corso e agli esami!*
- ❑ Dovete dimostrare di conoscere la programmazione degli algoritmi fondamentali

Errori Comuni:

Mettere in ombra le variabili ereditate

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

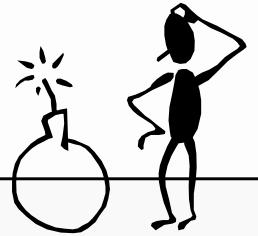


❑ Abbiamo visto che una sottoclasse *eredita* le variabili di esemplare definite nella superclasse

– ogni oggetto della sottoclasse ha una propria copia di tali variabili, come ogni oggetto della superclasse

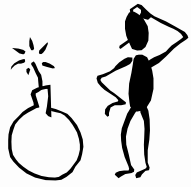
ma se sono **private** *non vi si può accedere dai metodi della sottoclasse!*

Mettere in ombra le variabili ereditate



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

- ❑ Proviamo: il compilatore segnala l'errore
balance has private access in BankAccount
- ❑ Quindi, per accedere alle variabili private ereditate, bisogna utilizzare metodi pubblici messi a disposizione dalla superclasse, se ce ne sono



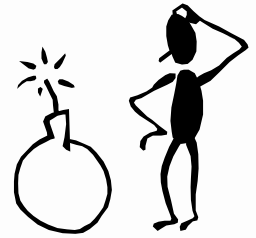
Mettere in ombra le variabili ereditate

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

```
public class SavingsAccount extends
    BankAccount
{
    ...
    public void addInterest()
    {
        deposit(balance * interestRate / 100);
    }
    private double balance;
}
```

- ❑ In questo modo 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'è alcuna relazione!* 87

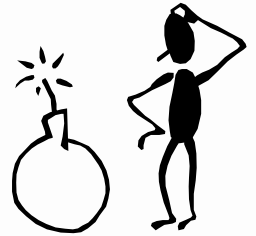
Mettere in ombra le variabili ereditate



```
public class SavingsAccount extends
    BankAccount
{
    ...
    public void addInterest()
    { deposit(balance * interestRate / 100);
    }
    private double balance;
}
```

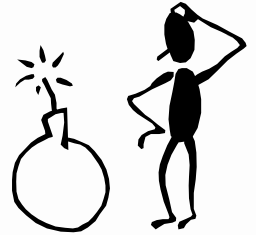
SavingsAccount	
balance	x
interestRate	y
balance	z

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 di **withdraw()**, la variabile **balance** definita in **SavingsAccount** non viene modificata
 - il calcolo degli interessi sarà sbagliato

Mettere in ombra le variabili ereditate



- ❑ Per quale motivo Java presenta questa apparente “debolezza”?
 - secondo le regole di OOP, chi scrive una sottoclasse non deve conoscere niente di quanto dichiarato privato nella superclasse, né scrivere codice che si basa su tali caratteristiche
 - quindi è perfettamente lecito definire e usare una variabile **balance** nella sottoclasse
 - sarebbe strano impedirlo, visto che chi progetta la sottoclasse non deve sapere che esiste una variabile **balance** nella superclasse!
 - ciò che non è lecito è usare una variabile nella sottoclasse con lo stesso nome di quella della superclasse *e sperare che siano la stessa cosa!*

Il metodo equals()

Oggetti distinti

- ❑ Cerchiamo di risolvere il problema di *determinare se in un array di oggetti esistono oggetti ripetuti, cioè* (per essere più precisi) *oggetti con identiche proprietà di stato*

```
public static boolean areUnique(Object[] p)
{
    for (int i = 0; i < p.length; i++)
        for (int j = i+1; j < p.length; j++)
            if (p[i] == p[j])
                return false;
    return true;
}
```

**Non Funziona
Correttamente!**

- ❑ Questo metodo non funziona correttamente
 - *verifica se nell'array esistono riferimenti che puntano allo stesso oggetto*

Oggetti distinti

- ❑ Per verificare, invece, che non esistano riferimenti che puntano a oggetti (eventualmente diversi) con le medesime proprietà di stato, occorre confrontare il *contenuto* (lo stato) degli oggetti stessi, e non solo i loro indirizzi in memoria
- ❑ Lo stato degli oggetti non è generalmente accessibile dall'esterno dell'oggetto stesso
- ❑ Come risolvere il problema?
 - sappiamo che per le classi della libreria standard possiamo invocare il metodo **equals()**

Il metodo equals()

- ❑ Come è possibile che il metodo **equals** sia invocabile con qualsiasi tipo di oggetto?
 - il metodo **equals()** è definito nella classe **Object**

```
public boolean equals(Object obj)
{
    return (this == obj);
}
```

- ❑ Essendo definito in **Object**, **equals()** viene ereditato da tutte le classi Java, quindi può essere invocato con qualsiasi oggetto, come **toString()**
 - il comportamento ereditato, se non sovrascritto, svolge la stessa funzione del confronto tra i riferimenti che abbiamo visto prima

Sovrascrivere il metodo equals()

- ❑ Per consentire il confronto per uguaglianza tra due oggetti di una classe in modo che venga esaminato lo *stato* degli oggetti stessi, occorre sovrascrivere il metodo **equals**

```
public class BankAccount
{
    public boolean equals(Object obj)
    {
        if (!(obj instanceof BankAccount))
            return false;
        BankAccount other = (BankAccount) obj;
        return (balance == other.balance);
    }
    ...
}
```

Oggetti distinti

```
public static boolean areUnique(Object[] p)
{
    for (int i = 0; i < p.length; i++)
        for (int j = i+1; j < p.length; j++)
            if (p[i].equals(p[j]))
                return false;
    return true;
}
```

- ❑ Questo metodo funziona correttamente se gli oggetti appartengono a classi che hanno sovrascritto il metodo **equals()**
 - non ci sono alternative: se una classe non sovrascrive **equals()**, in generale non si possono confrontare i suoi oggetti

Oggetti distinti comparabili

```
{ public boolean equals(Object obj)
{...}
public int compareTo(Object obj)
{...}
}
```

- ❑ Nelle classi che realizzano l'interfaccia Comparable e sovrascrivono il metodo equals(), si raccomanda che **l'ordinamento naturale** definito dal metodo **compareTo** e il **confronto di uguaglianza** fra oggetti definito al metodo **equals** siano consistenti:
$$(x.compareTo(y) == 0) == x.equals(y)$$
- ❑ Se questo non accade, nella documentazione della classe si deve scrivere esplicitamente
 - *"Notare: questa classe ha un ordinamento naturale che non è consistente con equals"*

**Classi non estendibili
e
metodi non sovrascrivibili**

Classi non estendibili

- Talvolta è opportuno impedire l'estensione di una classe
 - Ad esempio la classe *java.lang.String* è una classe *immutabile* ovvero i suoi metodi non possono mai modificare le variabili di esemplare della classe
 - i progettisti della classe hanno anche deciso che non possa mai essere estesa
 - Questo si può ottenere definendo la classe *final*

```
public final class String {...}
```

La classe non può essere estesa

Metodi non sovrascrivibili

- ❑ Talvolta è anche opportuno impedire di sovrascrivere un metodo nelle sottoclassi. Questo si può ottenere definendo il metodo *final* nella superclasse

```
public class BankAccount
{
    ...
    public final void withdraw(double amount)
    {
        if (balance < amount)
            throw new IllegalStateException();
        balance = balance - amount;
    }
}
```

```
public class MyDreamBankAccount extends BankAccount
{
    ...
    public void withdraw(double amount)
    {
        deposit(amount);
    }
}
```



```
MyDreamBankAccount.java:45: withdraw(double) in
MyDreamBankAccount cannot override withdraw(double)
in BankAccount; overridden method is final
    public void withdraw(double amount)
```

Metodi astratti

- ❑ Un metodo astratto non ha implementazione nella classe in cui è definito
- ❑ Nella classe si definisce l'intestazione del metodo con la parola chiave ***abstract***. Anche la classe deve essere definita ***abstract***.

```
public abstract class Poligono
{...
    public abstract double getSurface();
}
```

- ❑ Il metodo deve essere realizzato nelle classi derivate

```
public class Quadrato extends Poligono
{...
    public double getSurface()
    {
        return lato * lato;
    }
}
```

Esempio

- ❑ Si scriva la classe contenitore `ArchivioOrdinato` con la seguente interfaccia pubblica:
 - `void aggiungi (comparable c)`
 - aggiunge un oggetto all'archivio, mantenendolo ordinato. In caso di necessita' ridimensiona l'archivio
 - `Object toglia ()`
 - ritorna il valore massimo nel contenitore - massimo nel senso di `compareTo()`
 - `boolean vuoto ()`
 - ritorna true se il contenitore è vuoto, false altrimenti
- ❑ Notare che la classe contenitore è generica, senza riferimenti specifici alla classe da memorizzare
 - Il parametro del metodo `aggiungi ()` è un `Comparable`
 - Il metodo `toglia ()` ritorna `Object`

Esempio

- ❑ Si usi il contenitore per memorizzare oggetti della classe studente

```
public class Studente implements Comparable
{
    private final String nome;
    private final int matricola;

    public Studente(String n, int m)
    {
        nome = n;
        matricola = m;
    }
    public int matricola() { return matricola; }
    public String nome() { return nome; }
    public String toString()
    {
        return matricola + ":" + nome;
    }
    public int compareTo(Object rhs)
    {
        return matricola - ((Studente)rhs).matricola;
    }
}
```

Esempio

- ❑ Usando l'interfaccia Comparable parametrica

```
public class Studente implements Comparable<Studente>
{
    private final String nome;
    private final int matricola;

    public Studente(String n, int m)
    {
        nome = n;
        matricola = m;
    }
    public int matricola() { return matricola; }
    public String nome() { return nome; }
    public String toString()
    {
        return matricola + ":" + nome;
    }
    public int compareTo(Studente rhs)
    {
        return matricola - rhs.matricola;
    }
}
```


Criteri multipli di ordinamento

Criteri multipli di ordinamento

```
public class StudenteEsteso extends Studente
{ public static final int NOME = 0;
  public static final int MATRICOLA = 1;
  private static int ordinamento = MATRICOLA;

  public StudenteEsteso(String n, int m)
  { super(n, m); }

  public int compareTo(Object rhs)
  { StudenteEsteso s = (StudenteEsteso)rhs;
    if (ordinamento == MATRICOLA)
      return matricola() - s.matricola();
    return nome().compareTo(s.nome());
  }

  public static void ordinaPer(int valore)
  { if (valore != NOME && valore != MATRICOLA)
    throw new IllegalArgumentException(
      "Non so ordinare per : " + valore);
    ordinamento = valore;
  }
}
```

Criteri multipli di ordinamento

```
...
StudenteEsteso[] v = new StudenteEsteso[100];

// inizializzazione elementi dell'array
v[0] = new StudenteEsteso("marco", 12345);

...
//Ordinamento per matricola
StudenteEsteso.ordinaPer(StudenteEsteso.MATRICOLA);
ArrayAlgorithms.selectionSort(v);
System.out.println("*** Per Matricola ***");
for (int i = 0; i < v.length; i++)
    System.out.println(v[i]);

//Ordinamento per nome
StudenteEsteso.ordinaPer(StudenteEsteso.NOME);
ArrayAlgorithms.selectionSort(v);
System.out.println("\n*** Per Nome ***");
for (int i = 0; i < v.length; i++)
    System.out.println(v[i]);

...
```

ArchivioOrdinato

```
/**
 * archivio ordinato realizzato con un array riempito
 * solo in parte.
 * Classe didattica.
 *
 * @author A. Luchetta
 * @version 14-Nov-2006
 */

import java.util.NoSuchElementException;

public class ArchivioOrdinato
{
    // variabili di esemplare
    private Comparable[] array;
    private int arraySize;

    public ArchivioOrdinato()
    {
        array = new Comparable[1];
        arraySize = 0;
    }
}
```

```

/**
    Aggiunge un elemento all'archivio ordinato
    Andamento asintotico  $O(n)$ 
    Prima dell'inserimento l'array e' ordinato!
    @param n il numero intero da aggiungere
*/
public void aggiungi(Comparable c)
{
    // ridimensionamento dell'array
    if (arraySize >= array.length)
    { Comparable[] newArray = new Comparable[2*array.length];
      for (int i = 0; i < arraySize; i++)
        newArray[i] = array[i];
      array = newArray;
    }

    /* inserimento e ordinamento dell'archivio
       (basta il ciclo interno dell'algoritmo di
       ordinamento per inserimento)
    */
    int j;
    for (j = arraySize; j>0 && c.compareTo(array[j-1]) < 0; j--)
        array[j] = array[j - 1];

    array[j] = c;
    arraySize++;
}

```

ArchivioOrdinato

```
/**
    ritorna il valore massimo nell'archivio,
    cancellandolo dall'arrchivio.
    Andamento asintotico O(1)
    @return il valore massimo
    @throws NoSuchElementException se l'array e' vuoto
 */
public Object toglì()
{
    if (vuoto())
        throw new NoSuchElementException();

    Object tmp = array[arraySize - 1];
    array[arraySize - 1] = null; // garbage collector

    arraySize--;

    return tmp;
}
```

ArchivioOrdinato

```
/**
    verifica se l'archivio è vuoto
    Andamento asintotico  $O(1)$ 

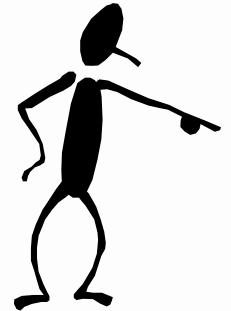
    @return true se l'array e' vuoto, false altrimenti
 */
public boolean vuoto()
{
    return arraySize == 0;
}
}
```

Lezione XXVIII
Gi 22-Nov-2007

Eccezioni

Lanciare Eccezioni (ripasso)

Lanciare eccezioni



❑ Sintassi:

```
throw oggettoEccezione;
```

❑ Scopo: lanciare un'eccezione

❑ Nota: di solito l'*oggettoEccezione* viene creato con **new** *ClasseEccezione*()

```
public void deposit(double amount)
{
    if (amount <= 0)
        throw new IllegalArgumentException();
    balance = balance + amount;
}
```

Gestire le Eccezioni

Le eccezioni in Java

- ❑ Quando un metodo *lancia* un'eccezione
 - l'esecuzione del metodo viene immediatamente interrotta
 - l'eccezione viene “propagata” al metodo chiamante la cui esecuzione viene a sua volta immediatamente interrotta
 - l'eccezione viene via via propagata fino al metodo **main()** la cui interruzione provoca l'arresto anormale del programma con la segnalazione dell'eccezione che è stata la causa di tale terminazione prematura
- ❑ Il lancio di un'eccezione è quindi un modo per terminare un programma in caso di errore
 - *non sempre però gli errori sono così gravi...*

Gestire le eccezioni di input

- ❑ Nell'esempio di conversione di stringhe in numeri supponiamo che la stringa sia stata introdotta dall'utente
 - se la stringa non contiene un numero valido viene generata un'eccezione **NumberFormatException**
 - sarebbe interessante poter *gestire* tale eccezione segnalando l'errore all'utente e chiedendo di inserire nuovamente il dato numerico *anziché terminare* prematuramente il programma
- ❑ Possiamo *intercettare* l'eccezione e gestirla con il costrutto sintattico **try / catch**

Conversione di stringhe in numeri

- ❑ La conversione corretta si ottiene invocando il metodo **statico `parseInt()`** della classe **`Integer`**

```
String ageString = "36";  
int age = Integer.parseInt(ageString);  
// age contiene il numero 36
```

- ❑ La conversione di un *numero in virgola mobile* si ottiene analogamente invocando il metodo **statico `parseDouble()`** della classe **`Double`**

```
String numberString = "34.3";  
double number =  
    Double.parseDouble(numberString);  
// number contiene il numero 34.3
```

Eccezioni nei formati numerici

	<i>done</i>	<i>!done</i>
...		
Scanner in = new Scanner(System.in);	false	true
int n = 0;		
boolean done = false;	true	false
while (!done)		
{		
try		
{		
String line = in.nextLine();		
n = Integer.parseInt(line);		
// l'assegnazione seguente viene		
// eseguita soltanto se NON viene		
// lanciata l'eccezione		
done = true;		
}		
catch (NumberFormatException e)		
{		
System.out.println("Riprova");		
// done rimane false		
}		
}		

Gestire le eccezioni

- ❑ L'enunciato che contiene l'invocazione del metodo che può generare l'eccezione deve essere racchiuso all'interno di una coppia di parentesi graffe precedute dalla parola chiave **try**

```
try
{
    ...
    n = Integer.parseInt(line);
    ...
}
```

- ❑ Bisogna poi sapere *di che tipo* è l'eccezione generata dal metodo
 - *ogni eccezione è un esemplare di una classe specifica, nel nostro caso*
 - `java.lang.NumberFormatException`

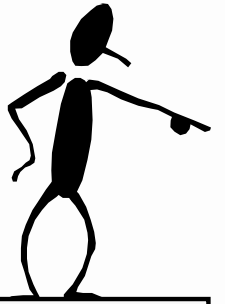
Gestire le eccezioni

- ❑ Il *blocco try* è seguito da una *clausola catch* definita in modo simile a *un metodo che riceve un solo parametro del tipo dell'eccezione* che si vuole gestire

```
catch (NumberFormatException e)
{
    System.out.println("Riprova");
}
```

- ❑ Nel blocco *catch* si trova *il codice che deve essere eseguito nel caso in cui si verifichi l'eccezione*
 - l'esecuzione del blocco **try** viene interrotta nel punto in cui si verifica l'eccezione e non viene più ripresa

Blocchi try / catch



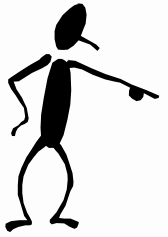
□ Sintassi:

```
try
{
    enunciatiCheGeneranoUnaEccezione
}
catch (ClasseEccezione riferimentoEccezione)
{
    enunciatiEseguitiInCasoDiEccezione
}
```

□ Scopo:

- eseguire enunciati che generano una eccezione
 - *se si verifica l'eccezione* di tipo *ClasseEccezione* eseguire gli enunciati contenuti nella *clausola catch*
 - *altrimenti* ignorare la *clausola catch*

Blocchi try / catch



- ❑ Se gli enunciati nel blocco try generano più di un'eccezione

```
try
{  enunciatiCheGeneranoPiùEccezioni
}
catch (ClasseEccezione1 riferimentoEccezione1)
{  enunciatiEseguitiInCasoDiEccezione1
}
catch (ClasseEccezione2 riferimentoEccezione2)
{  enunciatiEseguitiInCasoDiEccezione2
}
...
```

Catalogazione delle eccezioni

Diversi tipi di eccezioni

- ❑ In Java esistono diversi tipi di eccezioni (cioè diverse **classi** di cui le eccezioni sono esemplari)
 - eccezioni di tipo *Error*
 - eccezioni di tipo *Exception*
 - un sottoinsieme sono di tipo *RuntimeException*
- ❑ *Error* ed *Exception* derivano entrambe dalla superclasse *java.lang.Throwable*
- ❑ Solo oggetti che siano esemplari di questa classe (o di qualsiasi sua sottoclasse) sono “lanciati” dalla Java Virtual Machine o possono essere lanciati dall’enunciato Java *throw*.

Diversi tipi di eccezioni

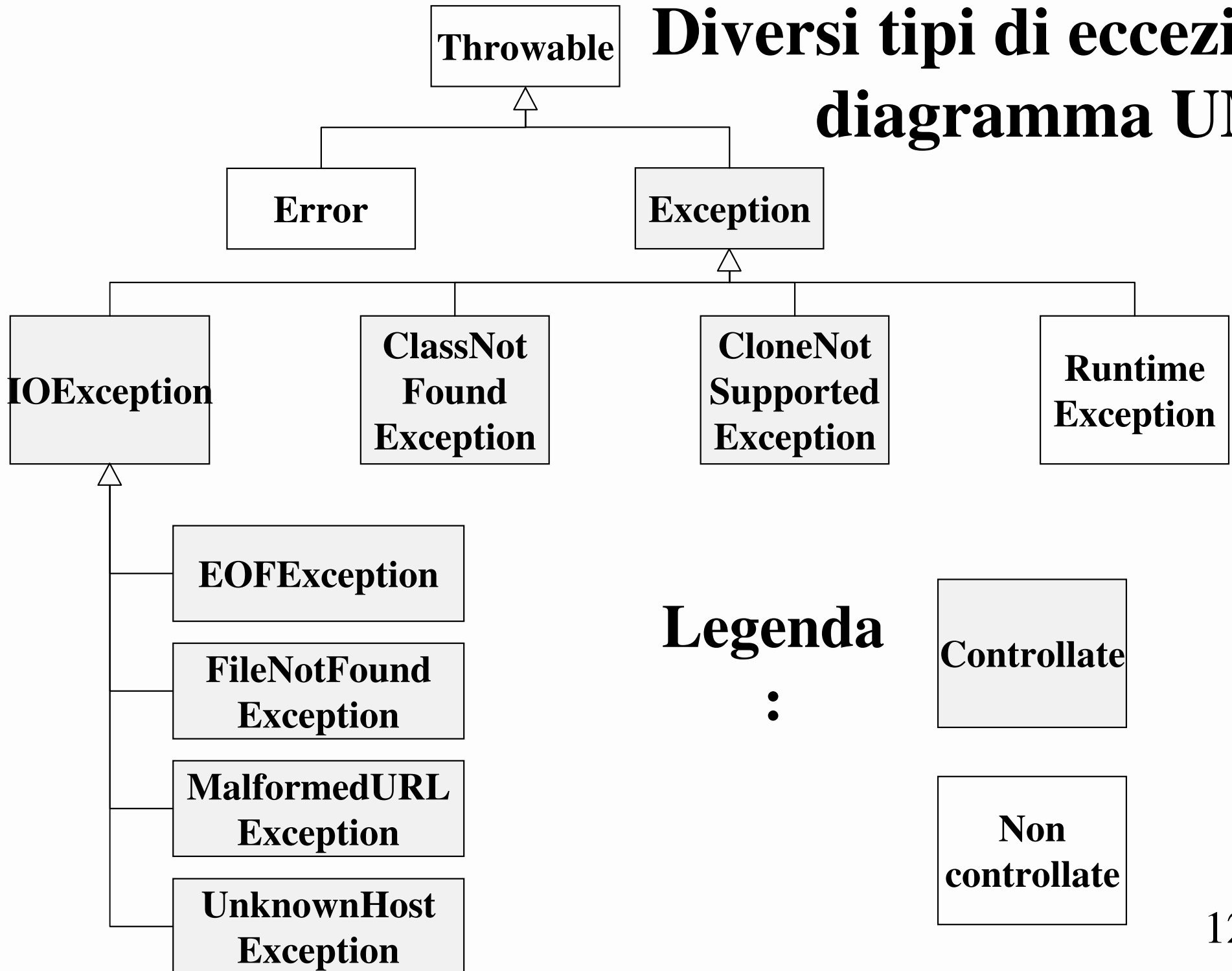
- ❑ **Error**: è una sottoclasse di *java.lang.Throwable* che indica l'occorrenza di seri problemi che un'applicazione “sensata” non dovrebbe tentare (try) di catturare (catch)
 - esempio: *java.lang.StackOverflowError*

- ❑ **Exception**: è una sottoclasse di *java.lang.Throwable* che indica condizioni che un'applicazione “sensata” potrebbe voler catturare
 - esempio: *java.lang.NumberFormatException*

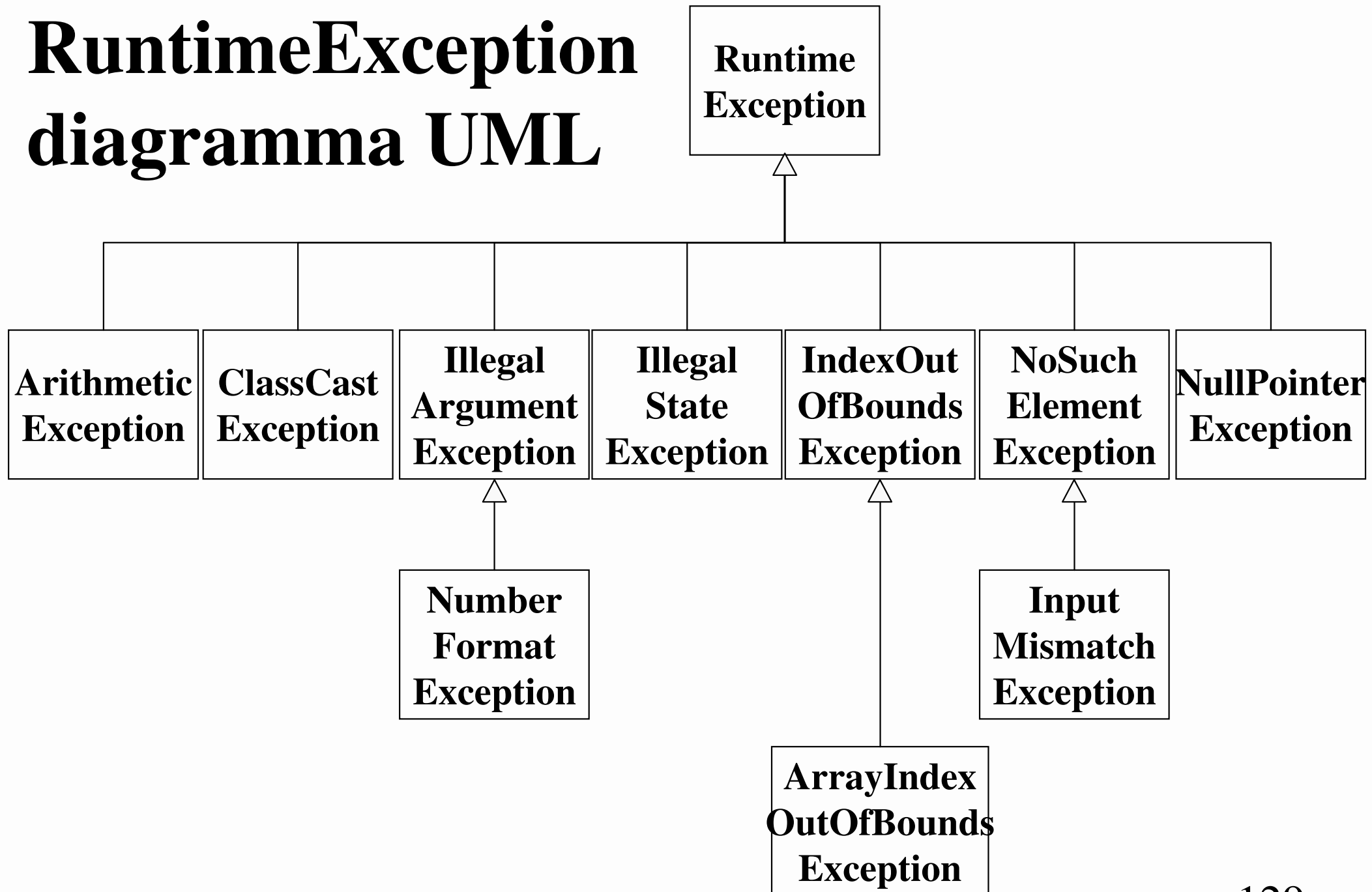
Diversi tipi di eccezioni

- ❑ La gestione delle eccezioni di tipo **Error** e di tipo **RuntimeException** è *facoltativa*
 - se non vengono gestite e vengono lanciate, provocano la terminazione del programma
- ❑ Il fatto che la loro gestione sia *facoltativa* (*non obbligatoria*) si esprime anche dicendo che **Error** e **RuntimeException** sono *eccezioni non controllate*.

Diversi tipi di eccezioni diagramma UML



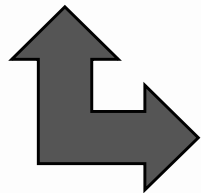
RuntimeException diagramma UML



Diversi tipi di eccezioni

- ❑ Ad esempio non e' obbligatorio gestire l'eccezione **NumberFormatException** che appartiene all'insieme delle **RuntimeException**
- ❑ Viene lanciata fra l'altro dai metodi **Integer.parseInt()** **Double.parseDouble()**

```
...  
int n = Integer.parseInt(line);  
...
```



Il compilatore non segnala errore se l'eccezione non e' gestita

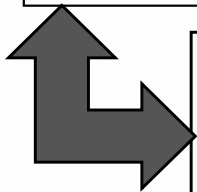
- ❑ Durante l'esecuzione del programma, se viene generata l'eccezione, questa si propaga fino al metodo **main()**, se non gestita, e provoca la terminazione del programma

**Exception in thread main java.lang.NumberFormatException:
at java.lang.Integer.parseInt(...)**

Diversi tipi di eccezioni

- ❑ E' invece obbligatorio gestire le eccezioni che sorgono nella gestione dell'input/output, come ***java.io.IOException*** o ***java.io.FileNotFoundException*** che appartengono all'insieme delle eccezioni **Exception**
- ❑ La classe **Scanner** può essere usata anche per leggere file (lo vedremo prossimamente); in questo caso il parametro esplicito e' un oggetto di classe **FileReader**
- ❑ Il costruttore **FileReader()** se non trova il file lancia l'eccezione controllata (a gestione obbligatoria)
java.io.FileNotFoundException

```
String filename = ...;  
FileReader reader = new FileReader(filename);  
Scanner in = new Scanner(reader);  
...
```



Il compilatore segnala errore se l'eccezione non e' gestita

Eccezioni Controllate

- ❑ Possimo gestire questa situazione in *due modi*
 - *try / catch*

```
try
{
    String filename = ...;
    FileReader reader = new FileReader(filename);
    Scanner in = new Scanner(reader);
}
catch (FileNotFoundException e)
{...}...
```

- ❑ Spesso pero' il metodo potrebbe non avere il contesto per gestire l'eccezione (il programmatore non sa come reagire all'interno del metodo)

Eccezioni Controllate

- ❑ In alternativa alla gestione con l'enunciato *try / catch*, si può avvisare il compilatore che si è consapevoli che l'eventuale insorgenza di un'eccezione causerà la terminazione del metodo
- ❑ Si lascia quindi la gestione al metodo chiamante

```
public void read(String filename)
    throws FileNotFoundException
{
    FileReader reader = new FileReader(filename);
    Scanner in = new Scanner(reader);
    ...
}
```

- ❑ Se il metodo può lanciare più eccezioni, si pongono nella firma dopo la clausola *throws*, separandole con una virgola

```
public void read(String filename)
    throws FileNotFoundException,
        ClassNotFoundException
{ ... }
```

Gestire le eccezioni di input

```
// NON FUNZIONA!  
import java.io.FileNotFoundException;  
import java.io.FileReader;  
import java.util.Scanner;  
  
public class ReadInput  
{ public static void main(String[] args)  
  { String filename = ...;  
    FileReader reader = new FileReader(filename);  
    Scanner in = new Scanner(reader);  
    ...  
  }  
}
```

ReadInput.java:10:
unreported exception java.io.FileNotFoundException;
must be caught or declared to be thrown



Gestire le eccezioni di input

- ❑ Per dichiarare che un metodo dovrebbe essere terminato quando accade un'eccezione controllata al suo interno si contrassegna il metodo con il marcatore **throws**

```
// FUNZIONA!
import java.io.FileNotFoundException;
import java.io.FileReader;
import java.util.Scanner;

public class ReadInput
{ public static void main(String[] args)
    throws FileNotFoundException
    { String filename = ...;
      FileReader reader = new FileReader(filename);
      Scanner in = new Scanner(reader);
      ...
    }
}
```

Avvisare il compilatore: **throws**

- ❑ La clausola **throws** segnala al chiamante del metodo che potrà trovarsi di fronte a un'eccezione del tipo dichiarato nell'intestazione del metodo.
 - ad esempio ad `IOException` nel caso del costruttore `FileReader()`
- ❑ Il metodo chiamante dovrà prendere una decisione:
 - o gestire l'eccezione (**try/catch**)
 - o dichiarare al proprio metodo chiamante che può essere generata un'eccezione (**throws**)
- ❑ Spesso non siamo in grado di gestire un'eccezione in un metodo, perché non possiamo reagire in modo adeguato o non sappiamo che cosa fare (inviare un messaggio allo standard output e terminare il programma potrebbe non essere adeguato)
- ❑ In questi casi è meglio lasciare la gestione dell'eccezione al metodo chiamante mediante la clausola **throws**

Tacitare le eccezioni

- ❑ Talvolta si è tentati di “*mettere a tacere*” le eccezioni controllate, programmando un blocco *catch* vuoto
- ❑ In questo modo il compilatore non segnala più errore...

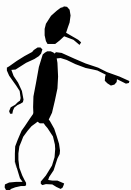
```
// Proibito!
import java.io.FileNotFoundException;
import java.io.FileReader;
import java.util.Scanner;

public class ReadInput
{ public static void main(String[] args)
  { String filename = ...;
    try
    { FileReader reader = new FileReader(filename);
    }
    catch (FileNotFoundException e)
    { // vuoto
    }

    ...
  }
}
```

Tacitare le eccezioni

- ❑ In esecuzione, se viene lanciata, però, l'eccezione questa non causa alcuna azione Infatti, il blocco catch la cattura, impedendo che si propaghi al metodo chiamante
- ❑ Poiché l'insorgenza dell'eccezione è il sintomo di un errore di qualche tipo nel programma, il programma fallirà, probabilmente, eseguendo qualche enunciato successivo
- ❑ Non siamo, però, in grado di diagnosticare correttamente che cosa è avvenuto, perché non abbiamo la segnalazione puntuale di dove è avvenuto effettivamente l'errore.
- ❑ Il meccanismo delle eccezioni è stato pensato per agevolare il programmatore a diagnosticare gli errori runtime e, quindi, a correggere o migliorare il codice.
- ❑ Se mettete a tacere le eccezioni, oltre a non proteggere il vostro codice dagli errori, impedito anche una corretta diagnosi degli stessi!



Proibito mettere a tacere le eccezioni!

Progettare eccezioni

Progettare Eccezioni

❑ Ora sappiamo

- programmare il lancio di eccezioni appartenente alla libreria standard (**throw new ...**)
- catturare eccezioni (**try/catch**)
- lasciare al chiamante la gestione di eccezioni sollevate nei nostri metodi (**throws** nell'intestazione del metodo)

❑ Non sappiamo ancora costruire eccezioni solo nostre

- ad esempio se vogliamo lanciare l'eccezione **LaNostraEccezioneException** come facciamo?)



Progettare Eccezione

- ❑ A volte nessun tipo di eccezione definita nella libreria standard descrive abbastanza precisamente una possibile condizione di errore di un metodo
- ❑ Supponiamo di voler generare un'eccezione di tipo `InsufficientBalanceException` nel metodo `withdraw()` della classe `BankAccount` quando si tenti di prelevare una somma maggiore del saldo del conto

```
...  
if (amount > balance)  
    throw new InsufficientBalanceException(  
        "prelievo di " + amount + " eccede la"  
        + " disponibilita' pari a " + balance);  
...
```

Progettare Eccezione

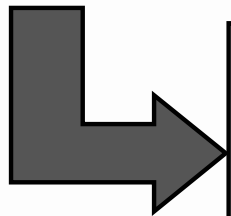
- ❑ Per costruire la nostra eccezione `InsufficientBalanceException` si usa l'ereditarietà, estendendo:
 - La classe *Exception* (o una delle sue sottoclassi), se si vuole creare *un'eccezione controllata*
 - La classe *RuntimeException* (o una delle sue sottoclassi), se si vuole creare *un'eccezione a gestione non obbligatoria*
- ❑ Di una classe eccezione solitamente si forniscono due costruttori, uno senza argomenti e uno che accetta una stringa che descrive il motivo dell'eccezione

```
public class InsufficientBalanceException
    extends RuntimeException
{
    public InsufficientBalanceException() {}
    public InsufficientBalanceException(
        String causa)
    {    super(causa); }
}
```

Progettare Eccezione

- ❑ Notare che **InsuffientBalanceException** è una classe pubblica e come tale va definita in un file a parte di nome **InsuffientBalanceException.java**

```
...  
BankAccount account = new BankAccount (500) ;  
account.withdraw (550) ;  
...
```



```
Exception in thread "main"  
InsuffientBalanceException: prelievo di  
550 eccede la disponibilita' pari a 500  
at ...
```

File e Flussi in Java

Gestione di file in Java

- ❑ Finora abbiamo visto programmi Java che interagiscono con l'utente soltanto tramite i *flussi* standard di ingresso e di uscita (*System.in*, *System.out*)
 - ciascuno di tali flussi può essere collegato a un file con un comando di sistema operativo (*redirezione*)
- ❑ Ci chiediamo: è possibile leggere e scrivere file in un programma Java?
 - con la redirezione di input/output, ad esempio, non possiamo leggere da due file o scrivere su due file...

Gestione di file in Java

- ❑ Limitiamoci inizialmente ad affrontare il problema della *gestione di file di testo*
 - *file di testo* $\equiv$ *file contenenti caratteri*
- ❑ In un *file di testo* i numeri vengono memorizzati come stringhe di caratteri numerici decimali e punti: 123.45 diventa “123.45”
 - esistono anche i *file binari*, che contengono semplicemente configurazioni di bit (byte) che rappresentano qualsiasi tipo di dati. Nei *file binari* i numeri sono memorizzati con la loro *rappresentazione numerica*. Es.: l'intero 32 diventa la sequenza di 4 byte 00000000 00000000 00000000 00100000
- ❑ La gestione dei file avviene interagendo con il sistema operativo mediante classi del pacchetto *java.io* della libreria standard

Gestione di file in Java

- ❑ Per leggere/scrivere *file di testo* si usano esemplari creati dalle classi
 - *java.io.FileReader*
 - *java.io.FileWriter*
- ❑ In generale le classi **reader** e **writer** sono in grado di gestire *flussi di caratteri*
- ❑ Per leggere/scrivere *file binari* si usano oggetti delle classi
 - *java.io.FileInputStream*
 - *java.io.FileOutputStream*
- ❑ In generale le classi con suffisso *Stream* sono in grado di gestire flussi di byte

Lettura di file di testo

- ❑ Prima di *leggere caratteri* da un file (esistente) occorre *aprire* il file in *lettura*
 - questa operazione si traduce in Java nella creazione di un oggetto di tipo **FileReader**

```
FileReader reader = new FileReader("file.txt");
```

- il costruttore necessita del nome del file sotto forma di stringa: "**file.txt**"
- se il file non esiste, viene lanciata l'eccezione **FileNotFoundException** a gestione obbligatoria

Lettura di file di testo

- ❑ Con l'oggetto di tipo *FileReader* si può invocare il metodo *read()* che restituisce *un intero* a ogni invocazione, iniziando dal primo carattere del file e procedendo fino alla fine del file stesso

```
FileReader reader = new FileReader("file.txt");
while(true)
{
    int x = reader.read(); // read restituisce un
    if (x == -1) break;     // intero che vale -1
    char c = (char) x;      // se il file è finito
    // elabora c
} // il metodo lancia IOException, da gestire!
```

- ❑ Non è possibile tornare indietro e rileggere caratteri già letti

- bisogna creare un nuovo oggetto di tipo *FileReader*

Lettura con `java.util.Scanner`

- ❑ In alternativa, si può costruire un'esemplare della classe *`java.util.Scanner`*, con il quale leggere righe di testo dal file

```
FileReader reader = new FileReader("file.txt");
Scanner in = new Scanner(reader);
while(in.hasNextLine())
{
    String line = in.nextLine();
    ... // elabora
}
// FileReader lancia FileNotFoundException
// (IOException), da gestire obbligatoriamente!
```

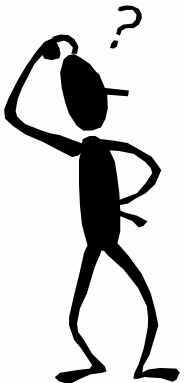
- ❑ Quando il file finisce, il metodo `nextLine()` della classe `Scanner` ritorna **false**!

Lettura di file di testo

- ❑ Al termine della lettura del file (che non necessariamente deve procedere fino alla fine...) occorre *chiudere* il file

```
FileReader reader = new FileReader("file.txt");  
...  
reader.close();
```

- ❑ Anche questo metodo lancia **IOException**, da gestire obbligatoriamente
- ❑ Se il file non viene chiuso non si ha un errore, ma una potenziale situazione di instabilità per il sistema operativo



Scrittura di file di testo

- ❑ Prima di scrivere caratteri in un *file di testo* occorre *aprire* il file in scrittura

Java 5.0

- questa operazione si traduce in Java nella creazione di un oggetto di tipo **PrintWriter**

```
PrintWriter writer = new PrintWriter("file.txt");
```

- il costruttore necessita del nome del file sotto forma di stringa e può lanciare l'eccezione **IOException**, che deve essere gestita
 - *se il file non esiste, viene creato*
 - *se il file esiste, il suo contenuto viene sovrascritto con i nuovi contenuti*
 - *e se vogliamo aggiungere in coda al file senza cancellare il testo già contenuto?*

boolean append

```
FileWriter writer = new  
FileWriter("file.txt", true);
```


Scrittura di file di testo

- ❑ L'oggetto di tipo *FileWriter* non ha i comodi metodi **print()/println()**
 - è comodo creare un oggetto di tipo **PrintWriter** che incapsula l'esemplare di **FileWriter**, aggiungendo la possibilità di invocare **print()/println()** con qualsiasi argomento

```
FileWriter writer = new FileWriter("file.txt");  
PrintWriter pw = new PrintWriter(writer);  
pw.println("Ciao");  
...
```

Scrittura di file di testo

- ❑ Al termine della scrittura del file occorre *chiudere* il file

```
PrintWriter writer = new FileWriter("file.txt");  
...  
writer.close();
```

- ❑ Anche questo metodo lancia **IOException**, da gestire obbligatoriamente
- ❑ *Se non viene invocato non si ha un errore, ma è possibile che la scrittura del file non venga ultimata prima della terminazione del programma, lasciando il file incompleto*

```

/** Copia1.java -- copia un file di testo una riga alla
    volta
    i nomi dei file sono passati come parametri dalla
    riga di comando
*/
import java.io.FileReader;
import java.io.FileWriter;
import java.io.PrintWriter;
import java.io.IOException;
import java.util.Scanner;
public class Copia1
{ public static void main (String[] arg) throws
    IOException
    {
        Scanner in = new Scanner(new FileReader(arg[0]));
        PrintWriter out = new PrintWriter(new
            FileWriter(arg[1]));

        while (in.hasNextLine())
            out.println(in.nextLine());

        out.close(); //chiusura scrittore
        in.close(); //chiusura lettore
    }
}

```

Esempio: numerare le righe

```
/**
 NumeratoreRighe.java crea la copia di un file di testo
 numerandone le righe. I nomi dei file sono passati
 come parametri dalla riga di comando
 */
import java.io.IOException;
import java.io.FileReader;
import java.io.PrintWriter;
import java.util.Scanner;
public class NumerarRighe
{ public static void main (String[] args) throws
    IOException
    {
        FileReader reader = new FileReader(args[0]);
        Scanner in = new Scanner(reader);
        PrintWriter out = new PrintWriter(args[1]);
        int numRiga = 0;
        while (in.hasNextLine())
        { numRiga++;
          out.println(numRiga + " " + in.nextLine());
        }
        out.close(); //chiusura scrittore
        in.close(); //chiusura lettore
    }
}
```

Copiare a caratteri un file di testo

```
/** Copia2.java -- copia un file di testo un carattere
    alla volta
    fare attenzione a non cercare di copiare un file
    binario con questo programma
 */
import java.io.FileReader;
import java.io.FileWriter;
import java.io.IOException;
public class Copia2
{ public static void main (String[] arg) throws
    IOException
    { FileReader in = new FileReader(arg[0]);
      FileWriter out = new FileWriter(arg[1]);
      int c = 0;
      while ((c = in.read()) != -1)
        out.write(c);
      out.close(); //chiusura scrittura
      in.close();  //chiusura lettore
    }
}
```

Lettura di file binari

- ❑ Prima di *leggere byte* da un file (esistente) occorre *aprire* il file in *lettura*
 - questa operazione si traduce in Java nella creazione di un oggetto di tipo **FileInputStream**

```
FileInputStream inputStream = new  
    FileInputStream("file.bin");
```

- il costruttore necessita del nome del file sotto forma di stringa
- se il file non esiste, viene lanciata l'eccezione **FileNotFoundException** che deve essere gestita

Lettura di file binari

- ❑ Con l'oggetto di tipo **FileInputStream** si può invocare il metodo **read()** che restituisce un *intero* (fra 0 e 255 o -1) a ogni invocazione, iniziando dal primo byte del file e procedendo fino alla fine del file stesso

```
FileInputStream in = new
    FileInputStream("file.bin");
while(true)
{   int x = in.read();           // read restituisce un
    if (x == -1) break;          // intero che vale -1
    byte b = (byte) x;           // se il file è finito
    // elabora b
} // il metodo lancia IOException, da gestire
```

- ❑ Non è possibile tornare indietro e rileggere caratteri già letti
 - bisogna creare un nuovo oggetto di tipo **FileInputStream**

Scrittura di file binari

❑ Prima di scrivere byte in un *file binario* occorre *aprire* il file in scrittura

- questa operazione si traduce in Java nella creazione di un oggetto di tipo **FileOutputStream**

```
FileOutputStream outputStream = new  
    FileOutputStream("file.txt");
```

- il costruttore necessita del nome del file sotto forma di stringa e può lanciare l'eccezione **IOException**, che deve essere gestita
 - *se il file non esiste, viene creato*
 - *se il file esiste, il suo contenuto viene sovrascritto con i nuovi contenuti*
 - *e se vogliamo aggiungere in coda al file senza cancellare il testo già contenuto?*

```
FileOutputStream outputStream = new  
    FileOutputStream("file.txt", true);
```

boolean append



Scrittura di file binario

- ❑ L'oggetto di tipo **FileOutputStream** ha il metodo `write()` che scrive un byte alla volta
- ❑ Al termine della scrittura del file occorre *chiudere* il file

```
FileOutputStream outputStream = new  
    FileOutputStream("file.txt");  
...  
outputStream.close();
```

- ❑ Anche questo metodo lancia **IOException**, da gestire obbligatoriamente
- ❑ *Se non viene invocato non si ha un errore, ma è possibile che la scrittura del file non venga ultimata prima della terminazione del programma, lasciando il file incompleto*

Gestione di file in Java

□ Usando le classi

- *FileReader*, *FileWriter* e `PrintWriter` del pacchetto **java.io** è possibile manipolare, all'interno di un programma Java, più *file di testo* in lettura e/o più file in scrittura
- *FileInputStream* e *FileOutputStream* del pacchetto **java.io** è possibile manipolare, all'interno di un programma Java, più *file binari* in lettura e/o più file in scrittura
- I metodi `read()` e `write()` di queste classi sono gli unici metodi basilari di lettura/scrittura su file della libreria standard
- Per elaborare linee di testo (o addirittura interi oggetti in binario) i flussi e i lettori devono essere combinati con altre classi, ad esempio `Scanner` o `PrintWriter`

Copiare un file binario

```
/**
 * Copia3.java - copia un file binario un byte alla volta
 */
import java.io.FileInputStream;
import java.io.FileOutputStream;
import java.io.IOException;
public class Copia3
{
    public static void main (String[] arg) throws
        IOException
    {
        FileInputStream in = new FileInputStream(arg[0]);
        FileOutputStream out = new
            FileOutputStream(arg[1]);
        int c = 0;
        while ((c = in.read()) != -1)
            out.write(c);
        out.close(); //chiusura scrittore
        in.close();  //chiusura lettore
    }
}
```

Fine riga e EOF

Che cosa è il *fineriga*?

- ❑ Un file di testo è un file composto da caratteri (nel nostro caso da caratteri di un byte secondo la codifica ASCII)
- ❑ Il file è diviso in *righe* cioè in sequenze di caratteri terminati dalla *stringa fineriga*
- ❑ Il *fineriga* dipende dal sistema operativo
 - “\r\n” Windows (‘\r’ Carriage Return, ‘\n’ Line Feed)
 - “\n” Unix – Linux
 - “\r” MAC OS
- ❑ L’uso di uno *Scanner* e del metodo *nextLine()* in ingresso, di un *PrintWriter* e del metodo *println()* in uscita, consentono di ignorare il problema della dipendenza del *fineriga* dall’ambiente in cui un’applicazione è utilizzata
- ❑ Se si elabora un file di testo un carattere alla volta è necessario gestire le differenze fra i diversi ambienti, in particolare il fatto che in due casi la stringa fineriga è di un carattere e nell’altro è composta da due caratteri

Che cosa è l'*End Of File* (EOF)

- ❑ La fine di un file (EOF) corrisponde alla fine dei dati che compongono il file
- ❑ L'EOF non è un carattere contenuto nel file!
- ❑ La condizione di EOF viene segnalata quando si cerca di leggere un byte o un carattere dopo la fine fisica del file
- ❑ Per segnalare la fine di un file da tastiera si usa una sequenza di controllo
 - Ctr z Windows
 - Ctr d Unix