



# Algoritmi di ricerca

*...And that, in simple terms, is how you increase your ranking on search engines."*

# Ricerca lineare

- Abbiamo già visto un algoritmo da utilizzare per individuare la posizione di un elemento che abbia un particolare valore all'interno di un array i cui elementi **non siano ordinati**
- Dato che bisogna esaminare tutti gli elementi, si parla di **ricerca sequenziale** o **lineare**

```
public class ArrayAlgs
{
    ...
    public static int linearSearch(int[] v, int vSize, int value)
    {
        for (int i = 0; i < vSize; i++)
            if (v[i] == value) return i; // trovato valore
        return -1; // valore non trovato
    }
    ...
}
```



# Ricerca lineare: prestazioni

- Stima di caso peggiore
  - Se il valore cercato non è presente nell'array, sono **sempre** necessari **n** accessi
  - Se il valore cercato è presente nell'array, il numero di accessi necessari per trovarlo dipende dalla sua posizione
    - Nel **caso peggiore** la ricerca inizia dal primo indice dell'array, e il valore cercato si trova nell'ultima posizione: sono necessari **n** accessi
  - In ogni caso, quindi, le prestazioni dell'algoritmo sono

$$T(n) = O(n)$$

- Stima di caso medio
  - Se il valore cercato è presente nell'array, **in media** sono necessari **n/2** accessi, quindi ancora  **$T(n) = O(n)$**

# È tutto chiaro? ...

1. Si immagini di cercare con **linearSearch** un numero telefonico in un array di 1000000 di dati. Quanti dati vanno esaminati mediamente per trovare il numero?

# Ricerca binaria

---

# Ricerca in un array ordinato

- Il problema di individuare la posizione di un elemento all'interno di un array può essere affrontato in modo più efficiente **se l'array è ordinato**
  - Esempio:** Ricerca dell'elemento **12** in questo array

5	9	11	12	17	21
<b>a[0]</b>	<b>a[1]</b>	<b>a[2]</b>	<b>a[3]</b>	<b>a[4]</b>	<b>a[5]</b>

- Confrontiamo **12** con l'elemento che si trova (circa) al **centro** dell'array, **a[2]**, che è **11**
- L'elemento che cerchiamo è **maggiore di 11**
  - se** è presente nell'array, **allora** sarà a **destra** di **11**

# Ricerca in un array ordinato

5	9	11	12	17	21
$a[0]$	$a[1]$	$a[2]$	$a[3]$	$a[4]$	$a[5]$

- A questo punto dobbiamo cercare l'elemento **12** nel solo sotto-array che si trova a destra di  $a[2]$ 
  - Usiamo lo stesso algoritmo, confrontando **12** con l'elemento che si trova al centro,  $a[4]$ , che è **17**
  - L'elemento che cerchiamo è minore di **17**
    - **se** è presente nell'array, **allora** sarà a **sinistra** di **17**

# Ricerca in un array ordinato

5	9	11	12	17	21
$a[0]$	$a[1]$	$a[2]$	$a[3]$	$a[4]$	$a[5]$

- A questo punto dobbiamo cercare l'elemento **12** nel sotto-array composto dal **solo elemento**  $a[3]$ 
  - Usiamo lo stesso algoritmo, confrontando **12** con l'elemento che si trova al centro,  $a[3]$ , che è **12**
  - L'elemento che cerchiamo è uguale a **12**
    - l'elemento che cerchiamo **è presente** nell'array e si trova in **posizione 3**



# Ricerca in un array ordinato

5	9	11	12	17	21
$a[0]$	$a[1]$	$a[2]$	$a[3]$	$a[4]$	$a[5]$

- Se il confronto tra l'elemento da cercare e l'elemento  $a[3]$  avesse dato esito negativo
  - avremmo cercato nel sotto-array **vuoto** a sinistra o a destra
  - concludendo che l'elemento cercato **non è presente** nell'array
- Questo algoritmo si chiama **ricerca binaria**
  - Perché a ogni passo si divide l'array **in due parti**
  - Può essere utilizzato **soltanto se l'array è ordinato**

# Realizzazione di `binarySearch`

```
public class ArrayAlgs
{
    ...
    public static int binarySearch(int[] v, int vSize, int value)
    {
        return binSearch(v, 0, vSize-1, value);
    }

    private static int binSearch(int[] v, int from,
                                int to, int value)
    {
        if (from > to) return -1; // caso base: el. non trovato
        int mid = (from + to) / 2; // mid e` circa in mezzo
        int middle = v[mid];
        if (middle == value)
            return mid; // caso base: elemento trovato
        else if (middle < value) //cerca a destra
            return binSearch(v, mid + 1, to, value);
        else // cerca a sinistra
            return binSearch(v, from, mid - 1, value);
    } //ATTENZIONE: e` un algoritmo con ricorsione SEMPLICE
    ...
}
```

# *Prestazioni di binarySearch*

- Valutiamo le prestazioni dell'algoritmo di ricerca binaria in un array ordinato
  - osserviamo che *l'algoritmo è ricorsivo*
- Per cercare in un array di dimensione **n** bisogna
  - effettuare un confronto (con l'elemento centrale)
  - effettuare una ricerca in un array di dimensione **n/2**
- Quindi

$$T(n) = T(n/2) + 1$$
- Analogamente all'analisi delle prestazioni di **mergeSort**, l'equazione per  **$T(n)$**  che abbiamo trovato è una **equazione di ricorrenza**

# Prestazioni di *binarySearch*

$$T(n) = T(n/2) + 1$$

- Come nel caso di **mergeSort**, una espressione esplicita per  $T(n)$  si trova per **sostituzioni successive**, fino ad arrivare al caso base  $T(1)=1$

$$\begin{aligned} T(n) &= (T(n/4) + 1) + 1 = T(n/4) + 2 = \\ &= \dots \text{(dopo } k \text{ sostituzioni)} \dots = T(n/2^k) + k \end{aligned}$$

- Dal termine  $T(n/2^k)$  si vede che il caso base è raggiunto per  $k = \log_2 n$ , ovvero per  $2^k = n$

$$T(n) = T(1) + \log_2 n = 1 + \log_2 n$$

- Quindi le prestazioni sono

$$T(n) = O(\log n)$$

- E sono **migliori** di quelle della **ricerca lineare**

# È tutto chiaro? ...

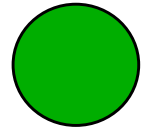
1. Si immagini di cercare con **binarySearch** un numero telefonico in un array *ordinato* di 1000000 di dati. Quanti dati vanno esaminati mediamente per trovare il numero?

# Approfondimento

## Analisi delle prestazioni di *recursiveFib*

---

# Prestazioni di recursiveFib



- Qualche lezione fa abbiamo scritto un metodo **ricorsivo** per calcolare i numeri di Fibonacci
  - Abbiamo verificato sperimentalmente che per  $n > 30$  il calcolo non è più istantaneo
  - E abbiamo visto perché: il metodo ricalcola più volte (inutilmente) valori già calcolati

```
public static long recursiveFib(int n)
{
    if (n < 1) throw new IllegalArgumentException();
    long f;
    if (n < 3) f = 1;
    else f = recursiveFib(n-1) + recursiveFib(n-2);
    return f;
}
```

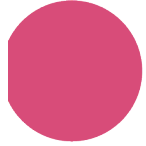
# Prestazioni di recursiveFib

- Stimiamo il tempo  $T(n)$  necessario a calcolare **fib**( $n$ )
  - Poiché l'algoritmo è ricorsivo, proviamo a cercare una **equazione di ricorrenza**
- Per calcolare **fib**( $n$ ) devo calcolare **fib**( $n-1$ ) e **fib**( $n-2$ )
- Quindi

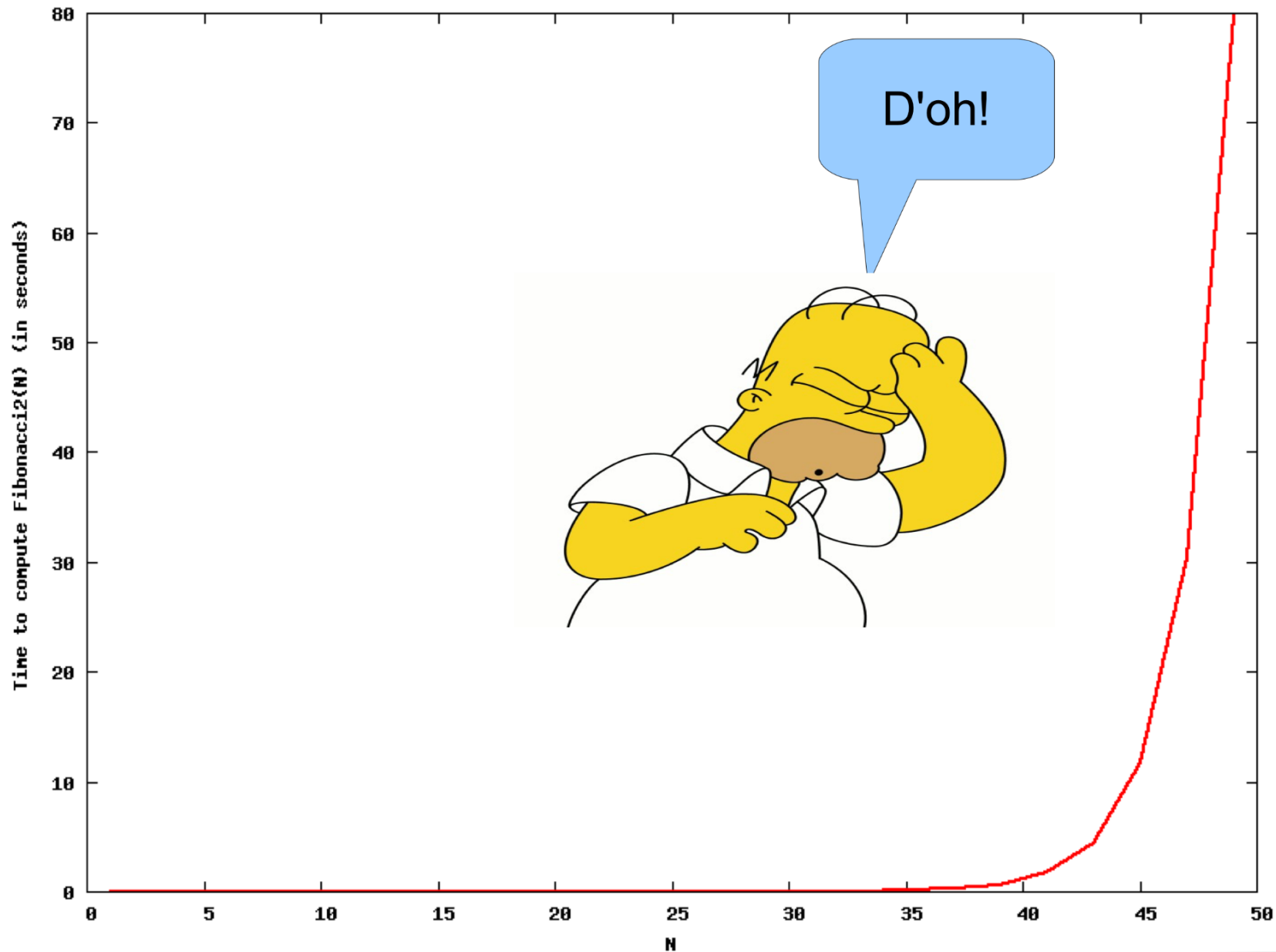
$$\begin{aligned}
 T(n) &= T(n-1) + T(n-2) > 2T(n-2) > 2[2T(n-4)] \\
 &> \dots > 2^k T(n-2k) > 2^{(n-1)/2}
 \end{aligned}$$

- Similmente
 
$$T(n) < 2T(n-1) < \dots < 2^k T(n-k) < 2^{n-1}$$
- Come avevamo anticipato, il tempo di esecuzione dell'algoritmo **fib** è **esponenziale** in  $n$





# Prestazioni di recursiveFib



# **Riflessione: Ordinamento di oggetti**

---

# Ordinamento di oggetti

- Si sono visti algoritmi di ordinamento su array di **numeri**
  - Ma spesso bisogna **ordinare dati più complessi**
  - Per esempio stringhe, ma anche **oggetti** di altro tipo
- Gli algoritmi di ordinamento che abbiamo esaminato effettuano **confronti tra numeri**
  - Si possono usare gli stessi algoritmi per ordinare oggetti, a patto che **questi siano tra loro confrontabili**
- C'è però una differenza
  - confrontare numeri ha un significato matematico ben definito
  - 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, per esempio, il metodo **selectionSort** per ordinare stringhe invece di ordinare numeri, **senza cambiare l'algoritmo**

# SelectionSort per stringhe

```
public class ArrayAlgs{...
    public static void selectionSort(String[] v, int vSize)
    {
        for (int i = 0; i < vSize - 1; i++)
        {
            int minPos = findMinPos(v, i, vSize-1);
            if (minPos != i) swap(v, minPos, i);
        }
    } //abbiamo usato due metodi ausiliari, swap e findMinPos
    private static void swap(String[] v, int i, int j)
    {
        String temp = v[i];
        v[i] = v[j];
        v[j] = temp;
    }
    private static int findMinPos(String[] v, int from, int to)
    {
        int pos = from;
        String min = v[from];
        for (int i = from + 1; i <= to; i++)
            if (v[i].compareTo(min) < 0)
            {
                pos = i;
                min = v[i];
            }
        return pos;
    }
    ...
}
```

# Ordinamento di oggetti

- 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, per esempio, di saldo crescente?
    - Bisogna definire nella classe **BankAccount** un metodo analogo al metodo **compareTo** della classe String
    - Bisogna **riscrivere i metodi** perché accettino come parametro un array di **BankAccount**
- Non possiamo certo usare questo approccio per qualsiasi classe, ***deve esserci un sistema migliore!***
  - In effetti c'è, ma dobbiamo prima studiare l'**ereditarietà**, il **polimorfismo** e l'uso di **interfacce** in Java ...
  - ... poi torneremo sul problema di ordinare oggetti generici

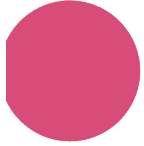
# Ereditarietà (capitolo 9)

---

# *L'ereditarietà*

- L'**ereditarietà** è uno dei principi basilari della programmazione orientata agli oggetti
- L'ereditarietà è un paradigma che supporta l'obiettivo di **riusabilità del codice**
  - Si sfrutta quando si deve realizzare una classe ed è già disponibile un'altra classe che rappresenta **un concetto più generale**
  - In questi casi, la nuova classe da scrivere è una classe più **specializzata**, che **eredita** i comportamenti (metodi) della classe più generale e ne **aggiunge di nuovi**





# L'ereditarietà

- Supponiamo di voler realizzare una classe **SavingsAccount** per rappresentare un **conto bancario di risparmio**,
  - Dovrà avere un **tasso di interesse** annuo determinato al momento dell'apertura
  - Dovrà avere 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
  - Allora possiamo riutilizzare il codice già scritto per la classe **BankAccount**

# *La classe BankAccount*

- È quella che abbiamo scritto tempo fa

```
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;
}
```

# La classe *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 previsto, buona parte del codice di **BankAccount** ha potuto essere **copiato** nella classe **SavingsAccount**
- Inoltre ci sono **tre cambiamenti**
  - Una nuova **variabile di esemplare**: **interestRate**
  - Un **costruttore diverso** (ovviamente il costruttore ha anche cambiato nome)
  - Un **nuovo metodo**: **addInterest**
- Copiare il codice non è una scelta soddisfacente
  - Cosa succede se **BankAccount** viene modificata?
  - Per esempio, modifichiamo **withdraw** in modo da impedire che il saldo diventi negativo => va modificato di conseguenza anche **SavingsAccount**
  - Molto scomodo, e fonte di molti errori...

# SavingsAccount

- Ri-scriviamo la classe **SavingsAccount** usando il meccanismo dell'ereditarietà
- Dichiariamo che **SavingsAccount** è una classe **derivata** da **BankAccount** (**extends**)
  - **eredita** tutte le caratteristiche (campi di esemplare e metodi) di **BankAccount**
  - specifichiamo soltanto le peculiarità di **SavingsAccount**

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

# SavingsAccount

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

sAcct

## SavingsAccount

*balance*

0

BankAccount()

deposit(...)

withdraw(...)

getBalance()

*interestRate*

10

SavingsAccount(...)

addInterest()

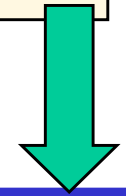
Metodi, costruttori,  
variabili di  
esemplare  
ereditati da  
**BankAccount**

Metodi, costruttori,  
variabili di  
esemplare definiti  
in  
**SavingsAccount**

# 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());
```



330

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

# Osservazioni su SavingsAccount

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

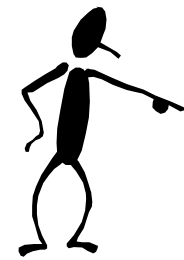
- Il metodo **addInterest** usa i metodi **getBalance** e **deposit** di **BankAccount**
  - **Non** specifica un parametro implicito
  - Cioè il parametro implicito è **this**
- Il metodo **addInterest** invoca **getBalance** e **deposit** invece che usare direttamente il campo **balance**
  - Questo perché il campo **balance** è definito come **private** in **BankAccount**
  - Mentre **addInterest** è definito in **SavingsAccount** e **non** ha accesso a **balance**



# *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:

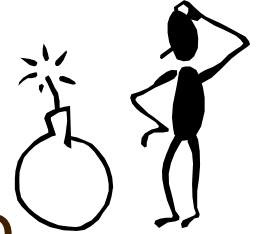
```
class NomeSottoclasse extends NomeSuperclasse
{
    costruttori
    nuovi metodi
    nuove variabili
}
```

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

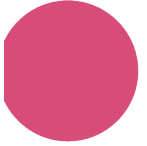
# Terminologia e notazione

---

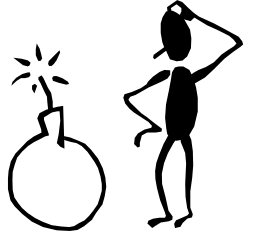
# Confondere superclassi e sottoclassi



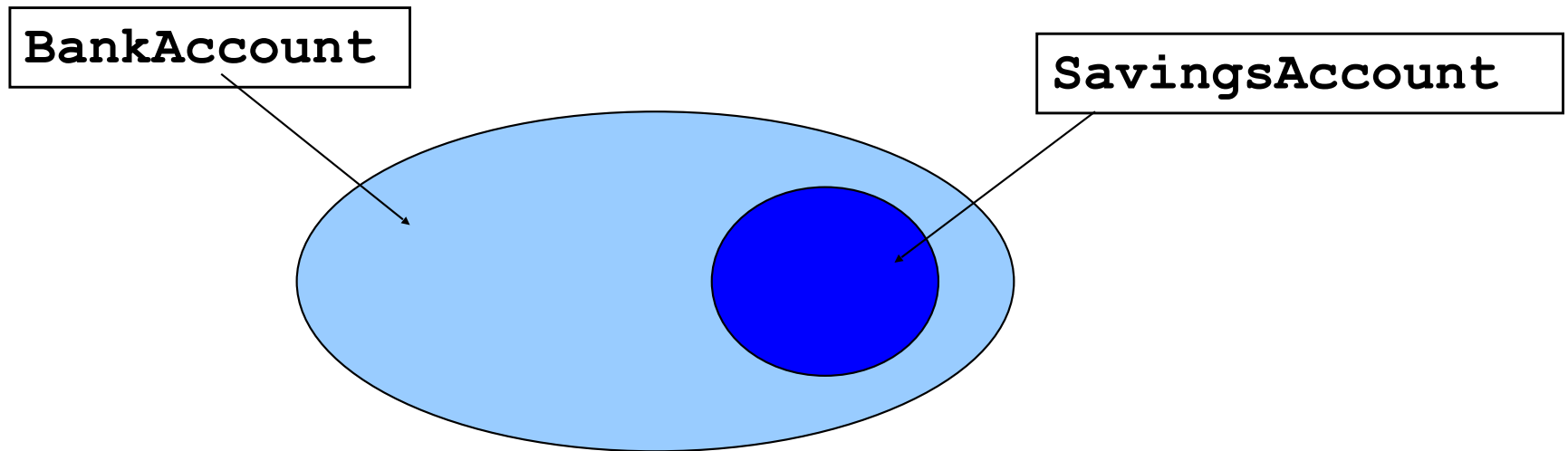
- Dato che oggetti di tipo **SavingsAccount** sono
  - un'**estensione** di oggetti di tipo **BankAccount**
  - più “grandi” di oggetti di tipo **BankAccount**, nel senso che hanno una variabile di esemplare in più
  - più “abili” di oggetti di tipo **BankAccount**, perché hanno un metodo in più
- perché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



# Confondere superclassi e sottoclassi

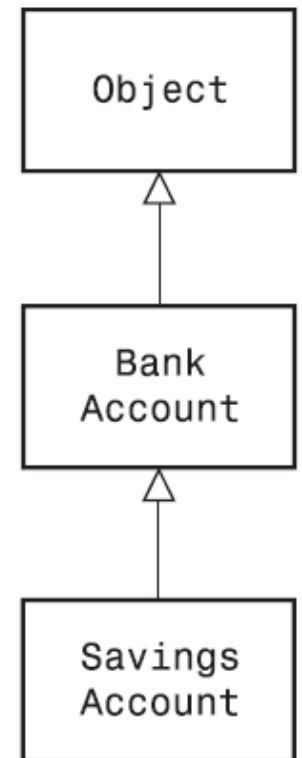


- I termini superclasse e sottoclasse derivano dalla **teoria degli insiemi**
- I conti bancari di risparmio (gli oggetti di tipo **SavingsAccount**) costituiscono un **sottoinsieme** dell'insieme di tutti i conti bancari (gli oggetti di tipo **BankAccount**)

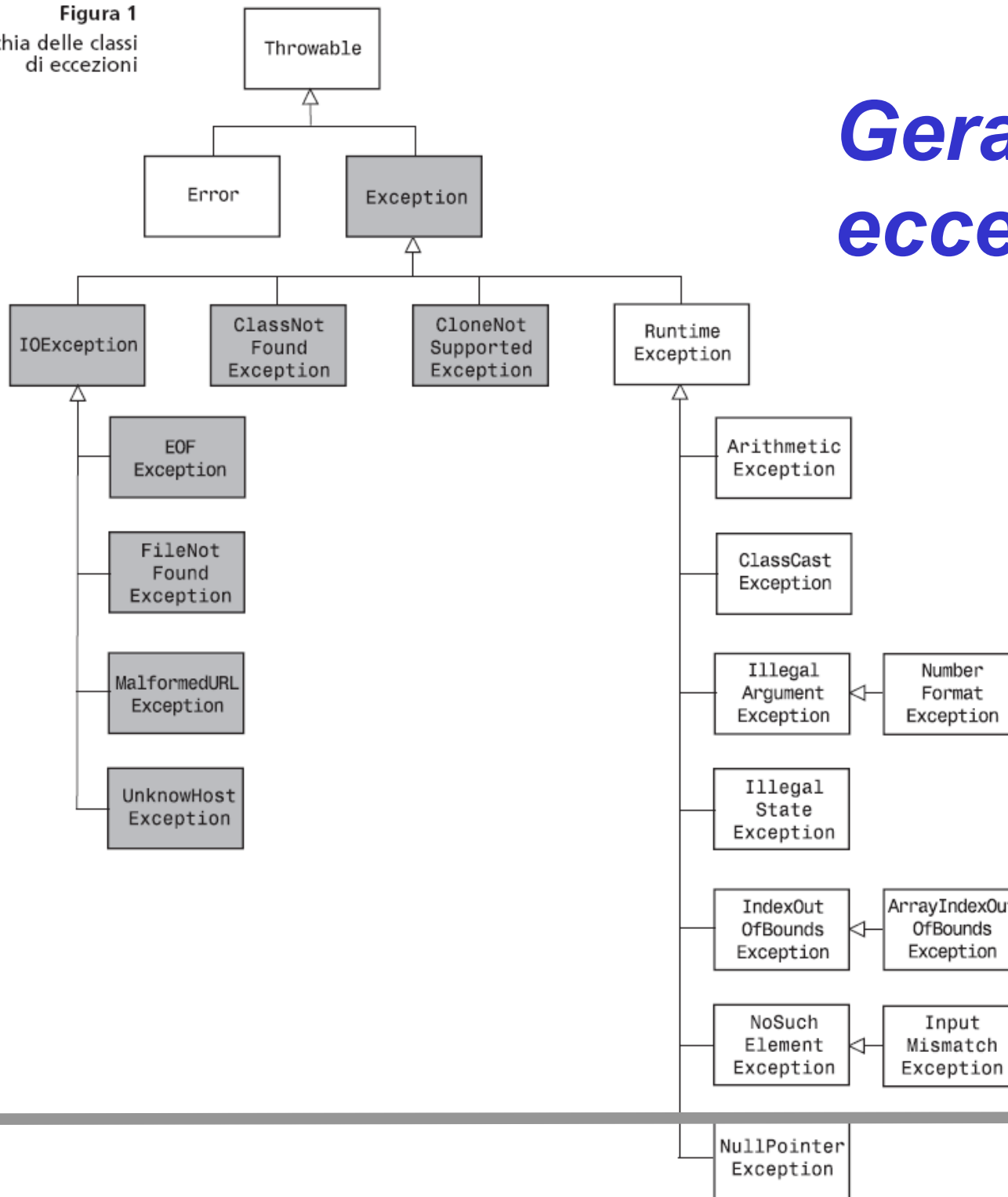


# Diagrammi e gerarchie di ereditarietà

- Abbiamo già visto diagrammi di classi per visualizzare **accoppiamento** tra classi
- In un diagramma di classi **l'ereditarietà** viene rappresentata mediante una freccia con la punta a “triangolo vuoto”, diretta verso la superclasse.
- Abbiamo già visto diagrammi di ereditarietà...
  - La gerarchia delle **classi di eccezioni**
  - Le gerarchie delle **classi del pacchetto io**
    - InputStream, OutputStream, Reader, Writer



**Figura 1**  
chia delle classi  
di eccezioni



# Gerarchia delle eccezioni

# È tutto chiaro? ...

1. Quali sono i campi di esemplare presenti in un oggetto di tipo **SavingsAccount**?
2. Elencare quattro metodi che possono essere invocati con un oggetto di tipo **SavingsAccount**
3. Se la classe **Manager** estende **Employee**, qual è la superclasse e quale la sottoclasse?

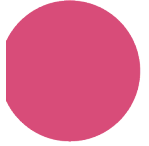


# Metodi di una sottoclasse

---

# Metodi di una sottoclasse

- Quando definiamo una sottoclasse, possono verificarsi **tre diverse situazioni** per quanto riguarda i suoi metodi
  - **Primo caso**: nella sottoclasse **viene definito un metodo** che nella superclasse non esisteva
    - Ad esempio il metodo **addInterest** di **SavingsAccount**
  - **Secondo caso**: un metodo della superclasse **viene ereditato** dalla sottoclasse
    - Ad esempio i metodi **deposit**, **withdraw**, **getBalance** di **SavingsAccount**, ereditati da **BankAccount**
  - **Terzo caso**: un metodo della superclasse **viene sovrascritto** nella sottoclasse
    - Vediamo ora un esempio anche di questo caso



U

I

I

I

# *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 del **OOP**
- Per sovrascrivere un metodo bisogna definire nella sottoclasse un metodo **con la stessa firma** di quello definito nella superclasse
  - tale metodo **prevale** 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
  - I versamenti nei conti di tipo **SavingsAccount** si fanno però invocando il metodo **deposit** di **BankAccount**, sul quale non abbiamo controllo
  - Possiamo però **sovrascrivere deposit**, **ridefinendolo** in **SavingsAccount**
  - In più aggiungiamo una costante di classe **FEE**, che contenga l'importo del costo fisso da addebitare

# Sovrascrivere un metodo: esempio

- Sovrascriviamo il metodo **deposit**

```
public class SavingsAccount extends BankAccount
{
    ...
    public void deposit(double amount)
    {
        ... // aggiungi amount a balance
        withdraw(FEE);
    }
    ...
    private final static double FEE = 2.58; // euro
}
```

- Quando viene invocato **deposit** con un oggetto di tipo **SavingsAccount**, viene invocato il metodo **deposit** **definito in 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** e sommarlo a **balance**
  - non possiamo modificare direttamente **balance**, che è una **variabile privata** in **BankAccount**
  - 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)
    {    deposit(amount); // NON FUNZIONA
        withdraw(FEE) ;
    }
}
```

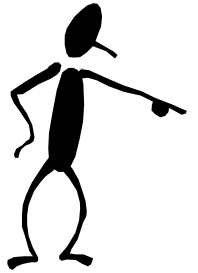
- Così non funziona, perché il metodo diventa **ricorsivo con ricorsione infinita!!!** (manca il caso base...)

# Sovrascrivere un metodo: esempio

- Ciò che dobbiamo fare è **invocare il metodo `deposit` di `BankAccount`**
- Questo si può fare usando la parola riservata **super**, gestita automaticamente dal compilatore per accedere agli elementi ereditati dalla superclasse

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

# Invocare un metodo della superclasse



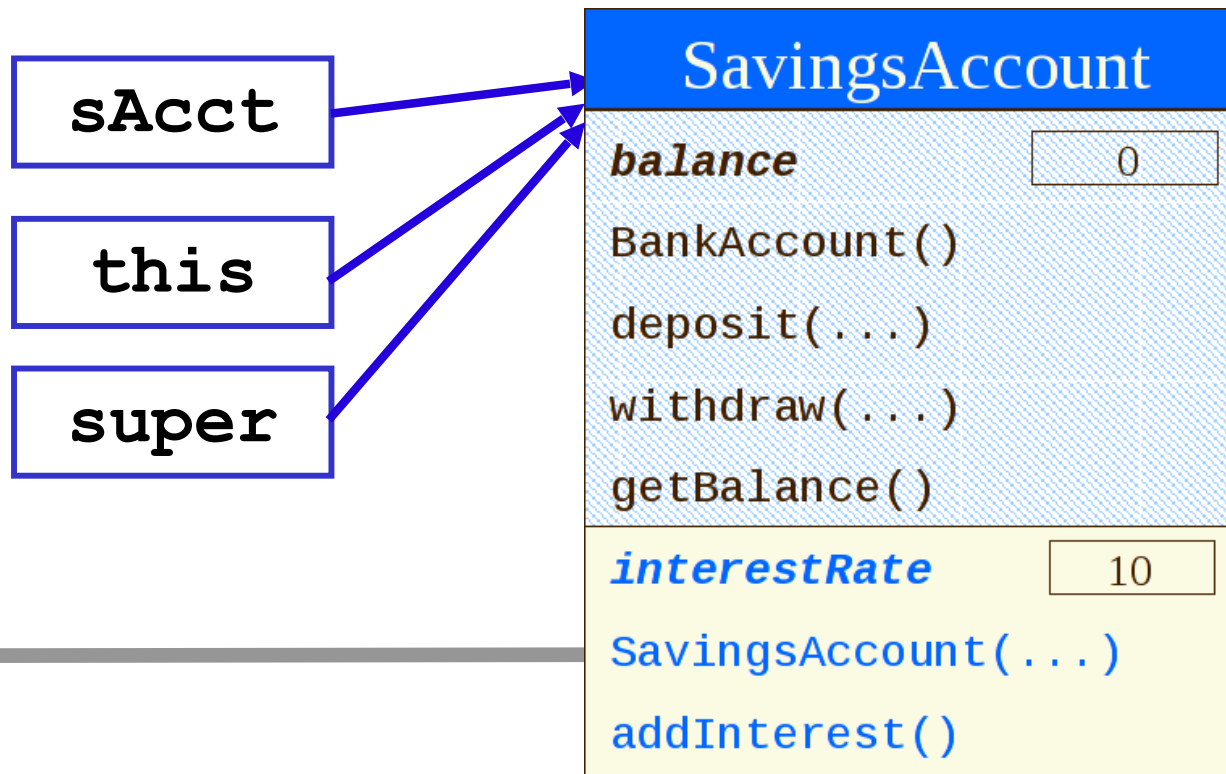
- Sintassi: `super.nomeMetodo(parametri)`
- Scopo: invocare il metodo **nomeMetodo** della superclasse anziché il metodo con lo stesso nome (sovrascritto) della classe corrente
  - **super** tratta l'oggetto a cui si riferisce **come se fosse un esemplare della superclasse**



# Il riferimento super

```
SavingsAccount sAcct = new SavingsAccount(10);  
sAcct.deposit(1000);
```

- Subito dopo l'invocazione di deposit (sovrascritto in **SavingsAccount**), esistono
- i riferimenti **sAcct** e **this** di tipo **SavingsAccount**
- **super** provoca forzatamente l'esecuzione del metodo della superclasse **BankAccount**



# Campi di esemplare di una sottoclasse

---

# *Ereditare campi di esemplare*

- Quando definiamo una sottoclasse, possono verificarsi **solo due diverse situazioni** per quanto riguarda i suoi campi di esemplare
  - **Primo caso**: nella sottoclasse **viene definito un campo di esemplare** che nella superclasse non esisteva
    - Ad esempio il campo **interestRate** di **SavingsAccount**
  - **Secondo caso**: un campo della superclasse **viene ereditato** dalla sottoclasse
    - Ad esempio il campo **balance** di **SavingsAccount**, ereditato da **BankAccount**
- Invece **non è possibile sovrascrivere** un campo della superclasse nella sottoclasse

# Ereditare campi di esemplare

- Cosa succede se nella sottoclasse viene definito un **campo omonimo** di uno della superclasse?
  - È un'operazione lecita, ma molto **sconsigliabile**
  - Si creano **due campi di esemplare** omonimi ma distinti, e in particolare il nuovo campo di esemplare **mette in ombra** il suo omonimo della superclasse

```
public class SavingsAccount extends BankAccount
{
    ...
    public void deposit(double amount)
    {
        withdraw(FEE);
        balance = balance + amount;
    }
    ...
    private double balance;    // Meglio non farlo!
}
//Il metodo deposit aggiorna questa variabile balance,
//mentre i metodi ereditati da BankAccount fanno
//riferimento alla variabile balance ereditata da
//BankAccount: errore logico!
```

# ***Mettere in ombra campi ereditati***

- Perché Java presenta questa apparente “debolezza”?
  - Chi scrive una sottoclasse **non deve conoscere niente** di quanto dichiarato **private** nella superclasse, né scrivere codice che si basa su tali caratteristiche: **incapsulamento!**
  - quindi è **perfettamente lecito** definire e usare una variabile `balance` nella sottoclasse
    - sarebbe strano impedirlo: chi progetta la sottoclasse **non sa** che esiste una variabile `balance` nella superclasse!
  - Ma **non si può** usare nella sottoclasse variabili omonime di quelle della superclasse e sperare che siano la stessa cosa

# Costruttori di una sottoclasse

---

# Costruttori di una sottoclasse

- Sappiamo che i campi di esemplare **privati** della superclasse **non** sono accessibili dalla sottoclasse
  - Allora come si fa a inizializzare questi campi di esemplare da un costruttore della sottoclasse?
- Bisogna invocare un costruttore della superclasse, usando la parola chiave **super** **seguita da parentesi tonde** ed eventuali **parametri espliciti** del costruttore
  - L'invocazione del costruttore della superclasse deve essere il **primo enunciato** del costruttore della sottoclasse

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

# Costruttori di una sottoclasse

- E se non invochiamo un costruttore della superclasse?
  - Allora viene invocato il **costruttore predefinito** della superclasse (quello privo di parametri espliciti)
  - Se questo non esiste (ovvero, se tutti i costruttori definiti nella superclasse richiedono parametri espliciti) viene generato un **errore in compilazione**

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



# È tutto chiaro? ...

1. Perché il costruttore di **SavingsAccount** visto prima non invoca il costruttore della superclasse?
2. Quando si invoca un metodo della superclasse usando l'enunciato **super**, l'invocazione deve essere il primo enunciato del metodo della sottoclasse?

# Esercizio: la classe CheckingAccount

---

# *La classe CheckingAccount*

```
public class CheckingAccount extends BankAccount
{
    public CheckingAccount(double initialBalance)
    {
        super(initialBalance); //costruttore della superclasse
        transactionCount = 0; // azzera conteggio transaz.
    }

    public void deposit(double amount) //SOVRASCRITTO!!
    {
        super.deposit(amount); // aggiungi amount al saldo
        transactionCount++;
    }

    public void withdraw(double amount) //SOVRASCRITTO!!
    {
        super.withdraw(amount); // sottrai amount dal saldo
        transactionCount++;
    }
}
```

// continua

# La classe *CheckingAccount*

```
// continua

public void deductFees() //NUOVO METODO
{
    if (transactionCount > FREE_TRANSACTIONS)
    {
        double fees = TRANSACTION_FEE *
            (transactionCount - FREE_TRANSACTIONS);
        super.withdraw(fees);
    }
    transactionCount = 0;
}

//nuovi campi di esempio
private int transactionCount;
private static final int FREE_TRANSACTIONS = 3;
private static final double TRANSACTION_FEE = 2.0;
}
```



# È tutto chiaro? ...

1. Perché il metodo `withdraw` di `CheckingAccount` invoca `super.withdraw`?
2. Perché il metodo `deductFees` pone a zero il conteggio delle transazioni effettuate?

# Conversioni di tipo tra superclasse e sottoclasse

---

# Conversione fra riferimenti

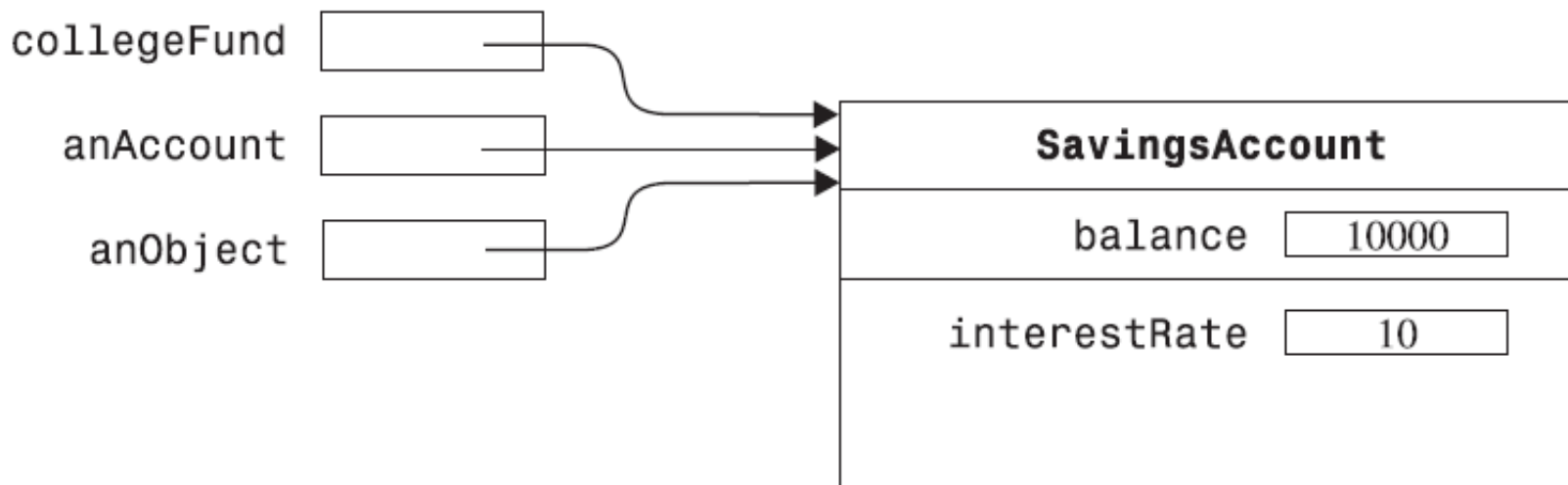
- Un oggetto di tipo **SavingsAccount** è un **caso speciale** di oggetti di tipo **BankAccount**
- Questa proprietà si riflette in una proprietà sintattica
  - una variabile oggetto del tipo di una superclasse **può riferirsi a un oggetto di una classe derivata**
  - **non c'è nessuna “conversione”** effettiva, cioè non vengono modificati i dati

```
SavingsAccount collegeFund = new SavingsAccount(10); //tutto ok
...
BankAccount anAccount = collegeFund;           // questo è lecito!!
Object anObject = collegeFund;                  // anche questo!!
```

# Conversione fra riferimenti

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

```
SavingsAccount collegeFund = new SavingsAccount(10); //tutto ok
...
BankAccount anAccount = collegeFund;           // questo è lecito!!
Object anObject = collegeFund;                  // anche questo!!
```



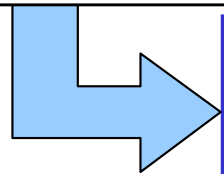


# Conversione fra riferimenti

```
SavingsAccount collegeFund = new SavingsAccount(10);  
BankAccount anAccount = collegeFund;  
anAccount.deposit(10000); //OK: deposit è metodo di BankAccount
```

- Tramite la variabile **anAccount** **si può** usare l'oggetto come se fosse di tipo **BankAccount**,
  - Ma **non si può** accedere alle proprietà specifiche di **SavingsAccount**
- Il tipo della variabile oggetto specifica **cosa si può fare** con un oggetto (cioè quali metodi si possono utilizzare)

```
anAccount.addInterest(); // questo NON è lecito perché  
                        //addInterest NON è metodo di BankAccount
```




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

# Conversione tra riferimenti

- Aggiungiamo il metodo **transfer** a **BankAccount**

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

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



300  
200

- Per quanto detto finora, il parametro **other** può anche riferirsi a un oggetto di tipo **SavingsAccount**

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

# Conversione fra riferimenti

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

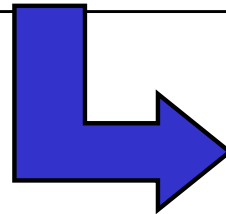
```
BankAccount other = new BankAccount(1000) ;  
SavingsAccount sAcct = new SavingsAccount(10) ;  
other.transfer(500, sAcct) ;
```

- Il compilatore sa che il metodo **transfer** richiede un riferimento di tipo **BankAccount**, quindi
  - controlla che **sAcct** sia un riferimento di tipo **BankAccount o di una sua sottoclasse**
  - effettua la conversione **automaticamente**

# Conversione fra riferimenti

- Vediamo la **conversione inversa** di quella vista finora
  - Ovvero la conversione di un riferimento a **superclasse** in un riferimento a **sottoclasse**
- Questa non può avvenire automaticamente

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



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

1 error

- Ma **ha senso** cercare di effettuarla?
  - In generale **non ha senso**, perché in generale un oggetto della superclasse non è un oggetto della sottoclasse

# Conversione fra riferimenti

- Tale conversione ha senso soltanto se, per le specifiche dell'algoritmo, siamo sicuri che il riferimento a **superclasse** punta in realtà a un oggetto della **sottoclasse**
  - Richiede un **cast** esplicito
  - Richiede attenzione, perché se ci siamo sbagliati verrà lanciata un'**eccezione**

```
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 ed ereditarietà

---

# Polimorfismo

- Sappiamo 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) ;
```

- Ma quale metodo **deposit** viene invocato, quello di **BankAccount** o quello **ridefinito** in **SavingsAccount**?
  - **acct** è una variabile dichiarata di tipo **BankAccount**
  - Ma contiene un riferimento a un oggetto che, **in realtà**, è di tipo **SavingsAccount**!
  - Questa informazione è disponibile solo in esecuzione (all'interprete Java), non in compilazione
  - secondo la semantica di Java, **viene invocato il metodo deposit di SavingsAccount**

# Polimorfismo

- In Java il tipo di una variabile **non determina in modo completo** il tipo dell'oggetto a cui essa si riferisce
- Questa semantica si chiama **polimorfismo** ("molte forme") ed è caratteristica dei linguaggi **Object-Oriented**
  - La stessa operazione (ad es. **deposit**) può essere svolta in modi diversi, a seconda dell'oggetto a cui ci si riferisce
- L'esecuzione di un metodo su un oggetto **è sempre determinata dal tipo dell'oggetto**, e **NON** dal tipo della variabile oggetto
  - Il tipo della variabile oggetto specifica **cosa si può fare** con un oggetto (cioè quali metodi si possono utilizzare)
  - Il tipo dell'oggetto specifica **come farlo**



# Polimorfismo

- Abbiamo già visto una forma di polimorfismo a proposito dei **metodi sovraccarichi**
  - L'invocazione del metodo **println** è in realtà una invocazione di un metodo scelto fra alcuni metodi con lo stesso nome ma **con firme diverse**
  - **Il compilatore** è in grado di capire quale metodo viene invocato, sulla base della firma
- 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**)
  - Mentre nel caso di metodi sovraccarichi si parla di **selezione anticipata** (**early binding**)

# Esercizio: polimorfismo (vers. 1)

```
public class AccountTester1
{
    public static void main(String[] args)
    {
        SavingsAccount momsSavings = new SavingsAccount(0.5);
        CheckingAccount harrysChecking = new CheckingAccount(100);

        // metodi polimorfici di BankAccount e sue sottoclassi
        momsSavings.deposit(10000); //vengono invocati i metodi delle
        momsSavings.transfer(2000, harrysChecking); //sottoclassi
        harrysChecking.withdraw(1500);
        harrysChecking.withdraw(80);
        momsSavings.transfer(1000, harrysChecking);
        harrysChecking.withdraw(400);

        // simulazione della fine del mese
        momsSavings.addInterest(); //metodo solo di SavingsAccount
        harrysChecking.deductFees(); //metodo solo di CheckingAccount
        System.out.println("Mom's savings balance = $"
                           + momsSavings.getBalance());
        System.out.println("Harry's checking balance = $"
                           + harrysChecking.getBalance());
    }
}
```

# Esercizio: polimorfismo (vers. 2)

```
public class AccountTester2
{
    public static void main(String[] args)
    {
        BankAccount momsSavings = new SavingsAccount(0.5);
        BankAccount harrysChecking = new CheckingAccount(100);

        // metodi polimorfici di BankAccount e sue sottoclassi
        momsSavings.deposit(10000); //vengono invocati i metodi delle
        momsSavings.transfer(2000, harrysChecking); //sottoclassi
        harrysChecking.withdraw(1500); //anche se i riferimenti sono
        harrysChecking.withdraw(80); //di tipo BankAccount
        momsSavings.transfer(1000, harrysChecking);
        harrysChecking.withdraw(400);

        // simulazione della fine del mese
        ((SavingsAccount)momsSavings).addInterest(); //e` necessario
        ((CheckingAccount)harrysChecking).deductFees(); //fare i cast
        System.out.println("Mom's savings balance = $"
                           + momsSavings.getBalance());
        System.out.println("Harry's checking balance = $"
                           + harrysChecking.getBalance());
    }
}
```

# Controllo del tipo e instanceof

- Se si vuole controllare il tipo dell'oggetto a cui una variabile oggetto si riferisce, si può usare un nuovo operatore: **instanceof**
  - È un operatore **relazionale** (restituisce valori booleani)
  - Restituisce **true** se la variabile oggetto (primo argomento) **sta puntando** a un oggetto del tipo specificato dal secondo argomento
  - Per esempio:

```
SavingsAccount collegeFund = new SavingsAccount(10);  
BankAccount anAccount = collegeFund;  
if (anAccount instanceof SavingsAccount)  
    SavingsAccount a = (SavingsAccount) anAccount;
```

Restituisce true

Possiamo effettuare il cast in tranquillità

# Controllo del tipo e instanceof

- Sintassi: `varOggetto instanceof NomeClasse`
- Restituisce
  - **true** se **varOggetto** contiene un riferimento a un oggetto della classe **NomeClasse** (o una sua sottoclasse)
  - **false** altrimenti
- In caso di valore restituito **true**, un eventuale cast di **varOggetto** a una variabile di tipo **NomeClasse** **NON lancia** l'eccezione **ClassCastException**
  - **Nota**: il risultato non dipende dal tipo di **varOggetto**, ma dal tipo dell'oggetto a cui essa si riferisce al momento dell'esecuzione