Lezione XXIII
Lu 7-Nov-2005

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
{
   private double balance;

   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;
   }
}
```

SavingsAccount

```
public class SavingsAccount
{
   private double interestRate;
   private double balance;

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

Riutilizzo del codice

- Come 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…
- L'ereditarietà risolve questi problemi

SavingsAccount

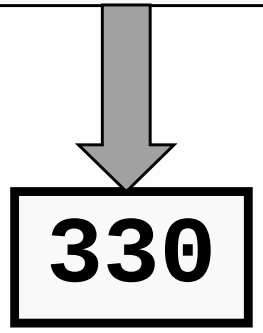
```
public class SavingsAccount extends BankAccount
{
   private double interestRate;
   public SavingsAccount(double rate)
   { interestRate = rate;
   }
   public void addInterest()
   { deposit(getBalance() * interestRate / 100);
   }
}
```

- Dichiariamo che **SavingsAccount** è una classe ***derivata*** da **BankAccount** (**extends**)
  - ne eredita tutte le caratteristiche
  - specifichiamo soltanto le peculiarità

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());
```

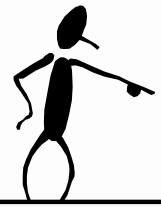


- La classe derivata si chiama
  - ***sottoclasse***
- La classe da cui si deriva si chiama
  - ***superclasse***

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

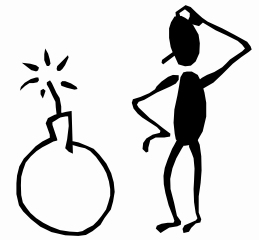


- Sintassi:

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

- 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**

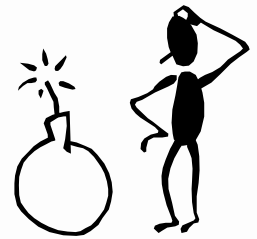
Confondere superclassi e sottoclassi



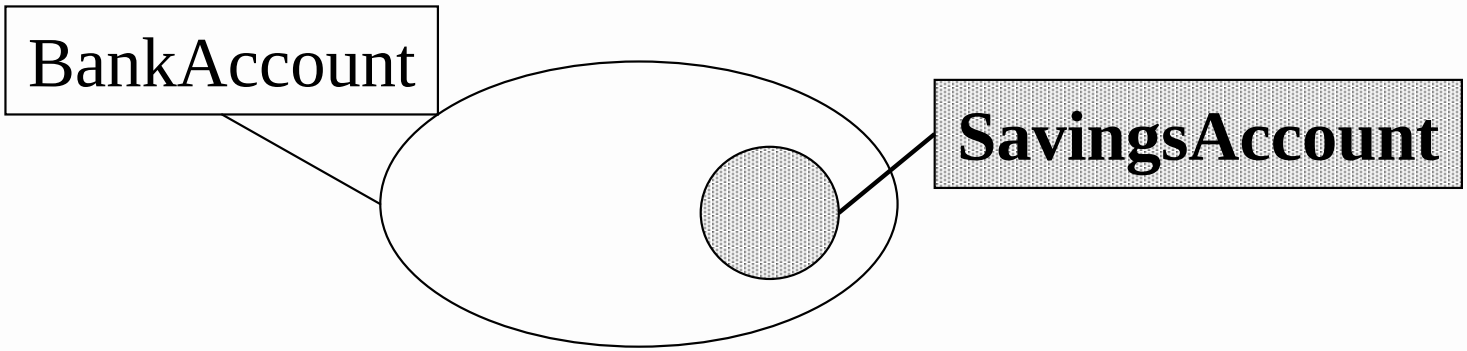
- Dato che oggetti di tipo **SavingsAccount** sono
  - un'***estensione*** di oggetti di tipo **BankAccount**
  - più "grandi" di oggetti di tipo **BankAccount**, perché hanno una variabile di esemplare in più
  - più "abili" di oggetti di tipo **BankAccount**, perché hanno un metodo in più

  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

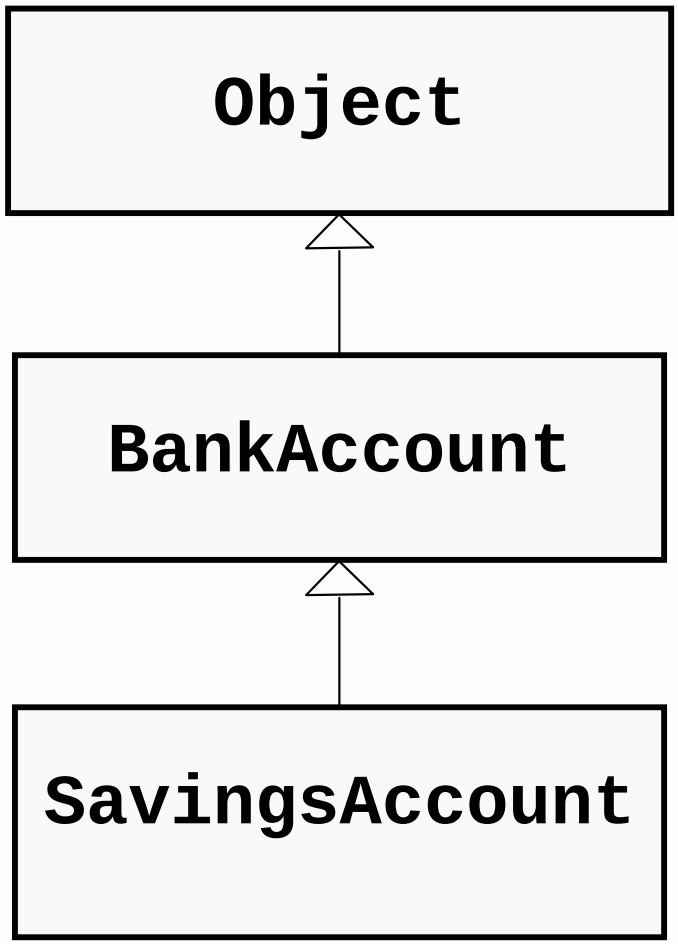


- I termini superclasse e sottoclasse derivano dalla ***teoria degli insiemi***
- I conti bancari di risparmio (gli oggetti di tipo **SavingsAccount**) costituiscono un ***sottoinsieme*** dell'insieme di tutti i conti bancari (gli oggetti di tipo **BankAccount**)



Rappresentazione UML

- 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 firma*** 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
   }
}
```

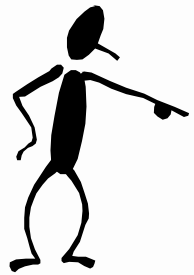
- Così però non funziona, perché il metodo diventa ricorsivo con ricorsione infinita!!! (manca il caso base…)

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***

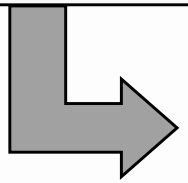
Sovrascrivere il metodo toString()

- 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 + "]”;
}
```

Sovrascrivere il metodo toString()

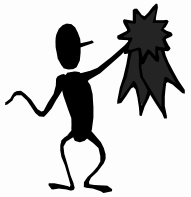
- Il metodo **toString()** di una classe viene invocato implicitamente anche quando si concatena un oggetto della classe con una stringa

```
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

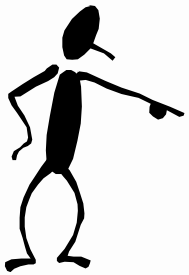
- Dato che la variabile **balance** è gestita dalla classe **BankAccount**, bisogna ***delegare*** a tale classe il compito di inizializzarla
- La classe **BankAccount** ha già un costruttore creato appositamente per gestire l'apertura di un conto con saldo diverso da zero
  - invochiamo tale costruttore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

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

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

Ereditarietà

- In Java una classe può estendere esplicitamente una sola superclasse:

```
public class SavingsAccount extends BankAccount //OK
{  ...
}
```

- L'ereditarietà multipla (ovvero la possibilità di derivare una classe da più superclassi) non e' supportata un Java

Il questionario a risposte multiple del 4-nov-2005

dom1: Allocazione della memoria in Java

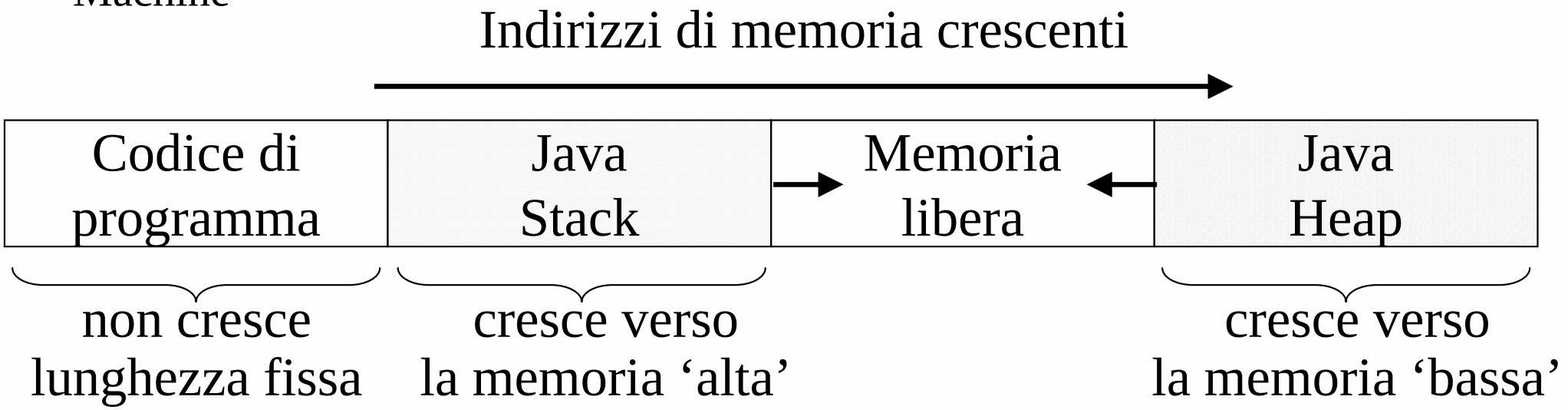
Domanda: Quale delle seguenti affermazioni relative allo spazio di memoria di un programma in Java e' corretta?

Risposte:

1. le variabili locali sono memorizzate nello spazo detto Java Heap
2. le variabili locali e i parametri dei metodi sono memorizzati nello spazio detto Java Stack
3. solo i parametri dei metodi sono memorizzati nello spazio detto Java Heap
4. gli oggetti istanziati sono memorizzati nello spazio detto java Stack
5. nessuna delle precedenti affermazioni e' corretta

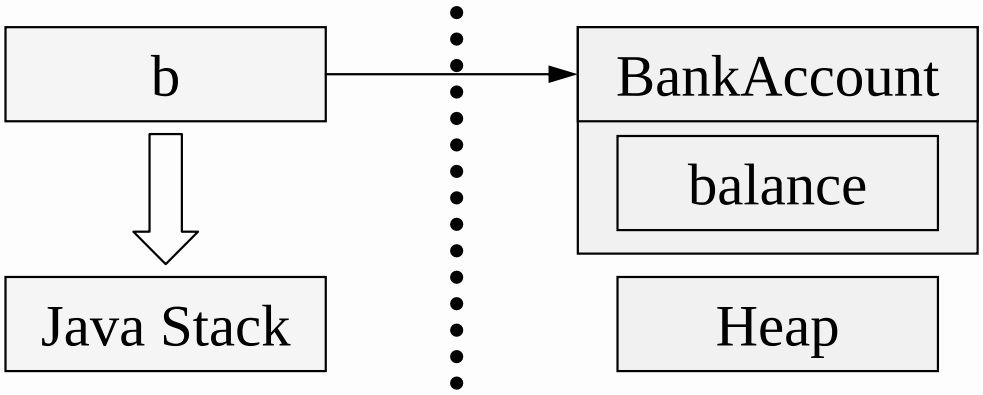
risp1: Allocazione della memoria in Java

- Schema della disposizione degli indirizzi di memoria nella Java Virtual Machine



- Le variabili parametro e locali sono memorizzate nel **Java Stack**
- Gli oggetti sono costruiti dall'operatore new nel **Java Heap**

```
BankAccount b = new BankAccount(1000);
```



dom4: Leggi di De Morgan

Domanda: Quale delle espressioni logiche sotto riportate e' equivalente alla seguente?

```
!(a > 0 && a < 10)
```

Risposte:

1. a <=0 && a >= 10
2. a <0 || a > 10
3. a <=0 || a >= 10
4. a <0 && a > 10
5. Nessuna delle risposte precedenti e' corretta

risp4: Leggi di De Morgan

In linguaggio Java

`!(a && b)` equivale a **`!a || !b`**

- Dimostrazione: tabelle di verità

a	*b*	*a && b*	*!(a && b)*
false	false	false	true
false	true	false	true
true	false	false	true
true	true	true	false

a	*b*	*!a*	*!b*	*!a || !b*
false	false	true	true	true
false	true	true	false	true
true	false	false	true	true
true	true	false	false	false

dom 8: Confronto lessicografico

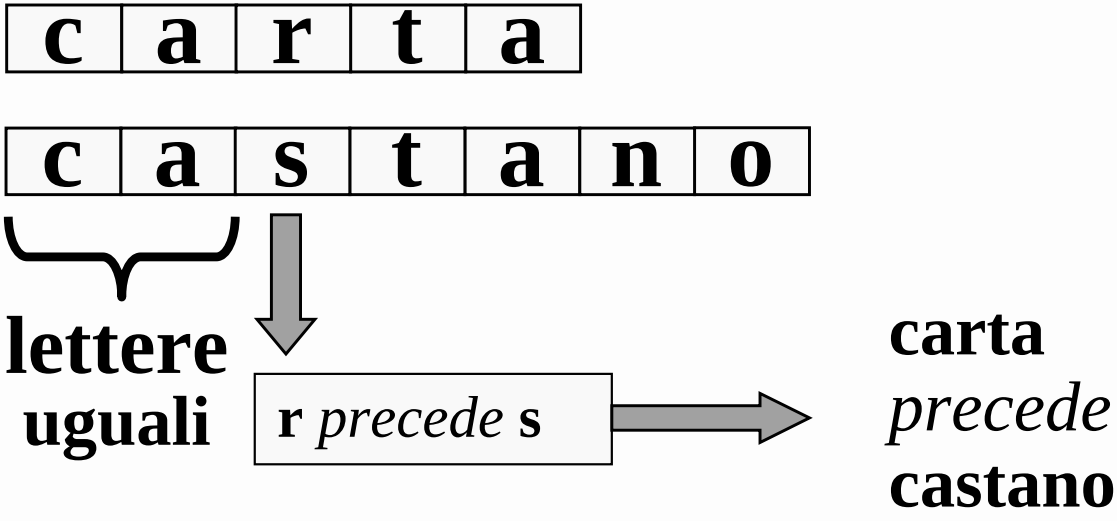
Domanda: Il confronto lessicografico fra due stringhe richiede un numero di confronti tra singoli caratteri

Risposte:

1. almeno uguale alla lunghezza della stringa piu' lunga
2. almeno uguale alla lunghezza della stringa piu' corta
3. almeno uguale alla somma delle lunghezze delle due stringhe
4. nessuna delle precedenti affermazioni e' corretta

Risp. 8: Confronto lessicografico

- Partendo dall'inizio delle stringhe si confrontano a due a due i caratteri in posizioni corrispondenti finché una delle stringhe termina oppure due caratteri sono diversi
  - se una stringa termina essa precede l'altra
    - se terminano entrambe sono uguali
  - altrimenti l'ordinamento tra le due stringhe è uguale all'ordinamento alfabetico tra i due caratteri diversi



dom11: Eliminare un elemento

Domanda: Il numero di operazioni (accessi a singoli elementi) necessarie per eliminare un elemento da un array riempito solo in parte in cui l'ordine sequenziale degli elementi non è importante

Risposte:

1. dipende solo dalla posizione dell'elemento da eliminare
2. non dipende dalla posizione dell'elemento da eliminare
3. nessuna delle precedenti affermazioni è corretta

Risp.11: Eliminare un elemento

- L’eliminazione di un elemento da un array richiede due algoritmi diversi
  - ***se l’ordine tra gli elementi dell’array non è importante*** (cioè se l’array realizza il concetto astratto di ***insieme***), allora ***è sufficiente copiare l’ultimo elemento dell’array nella posizione dell’elemento da eliminare*** e ridimensionare l’array (oppure usare la tecnica degli array riempiti soltanto in parte)

```
double[] values = {1, 2.3, 4.5, 5.6};
int indexToRemove = 0;
values[indexToRemove] =
   values[values.length - 1];
values = resize(values, values.length - 1);
```

dom14: Parametri formali ed effettivi

Domanda: Che risultato produce l'esecuzione della seguente classe P?

```
class P
{
  private static void f(int i, int j)
  {
    i--;
    if (j == 0) return;
    elese j--;
  }


  public static void main(String[] arge)
  {
    int a = 6;
    int b = 4;


    while (a > 0) f (a, b);
      System.out.println(a + " + " + b);
  }
}
```

Risposte:

1. stampa a standard output la stringa "6 + 4"
2. stampa a standard output la stringa "6 + 0"
3. il programma non si arresta
4. stampa a standard output la stringa "0 + 0"

dom14: Parametri formali ed effettivi

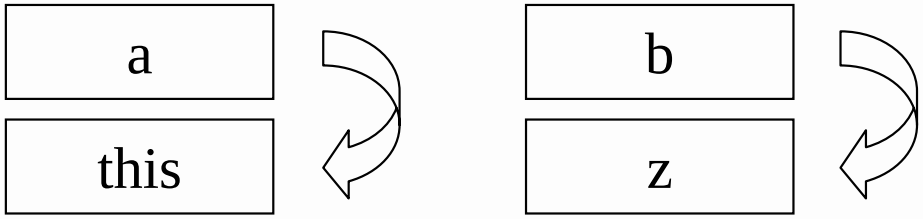
- I parametri espliciti che compaiono nella firma dei metodi e il parametro implicito this (usati nella realizzazione dei metodi) si dicono ***Parametri Formali*** del metodo

```
public Complex add(Complex z)
{
   return new Complex(this.re+z.re, this.im+z.im);
}
```

- I parametri forniti nell'invocazione ai metodi si dicono ***Parametri Effettivi*** del metodo

```
Complex a = new Complex(1,2);
Complex b = new Complex(2,3);
Complex c = a.add(b);
```

  - Al momento dell'esecuzione dell'invocazione del metodo, i **parametri effettivi** sono **copiati** nei **parametri formali**



dom23: metodo ricorsivo

Domanda: Che cosa calcola il seguente metodo ricorsivo invocato con il parametro effettivo n > 0?

```
static int somma(int n, int k)
{
 if (n == 1) return k;
 else return k + somma(n - 1, k);
}
```

Risposte:

1. la somma dei primi k numeri
2. il prodotto n * k
3. la somma dei primi n * k interi positivi
4. la somma dei primi n numeri primi positivi

Risp.23: metodo ricorsivo

```
static int somma(int n, int k)
{
 if (n == 1) return k;
 else return k + somma(n - 1, k);
}
```

```
somma(1, k) = k
somma(2, k) = k + somma(1, k) = k + k  = 2k
somma(3, k) = k + somma(2, k) = k + 2k = 3k
…
somma(n, k) = k + somma(n-1, k) = k + (n-1)k = n*k
```

dom31: calcolo tempi di esecuzione

Domanda: Sia dato un elenco di 1048 nomi non ordinati alfabeticamente. Se per ordinare l'elenco si impiega al meglio 1 ms, quanti millisecondi sono necessari per ordinare un elenco non ordinato di 1.000.000 elementi ?

Risposte:

1. meno di 10 ms
2. piu' di 10 ms ma meno di 100ms
3. piu' di 100 ms ma meno di 1 secondo
4. piu' di 1 secondo ma meno di 10 secondi
5. piu' di 10 secondi

dom31: calcolo tempi di esecuzione

$n_1 = 1048 \sim= 10^3$, $n_2 = 1000000 = 10^6$

$\tau_1 = 10^{-3}$ s, $\tau_2 = ?$

Al meglio => $t(n) = O(g(n))$, $g(n) = n \log n$

$t(n) \sim= k * n \log n$

$\tau_1 = t(n_1) \sim= k\, g(n_1)$

$\tau_2 = t(n_2) \sim= k\, g(n_2)$

$$\frac{\tau_2}{\tau_1} \sim= \frac{g(n_2)}{g(n_1)} \sim= \frac{n_2 \log n_2}{n_1 \log n_1}$$

$$\tau_2 \sim= \frac{n_2 \log n_2}{n_1 \log n_1} * \tau_1 \sim= \frac{10^6 * \log 10^6 * 10^{-3}}{10^3 * \log 10^3}$$

$$\tau_2 \sim= \frac{10^3 * 6}{3} * 10^{-3} \sim= 2 \text{ s}$$

Lezione XXIV
Ma 8-Nov-2005

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);
   }
}
```

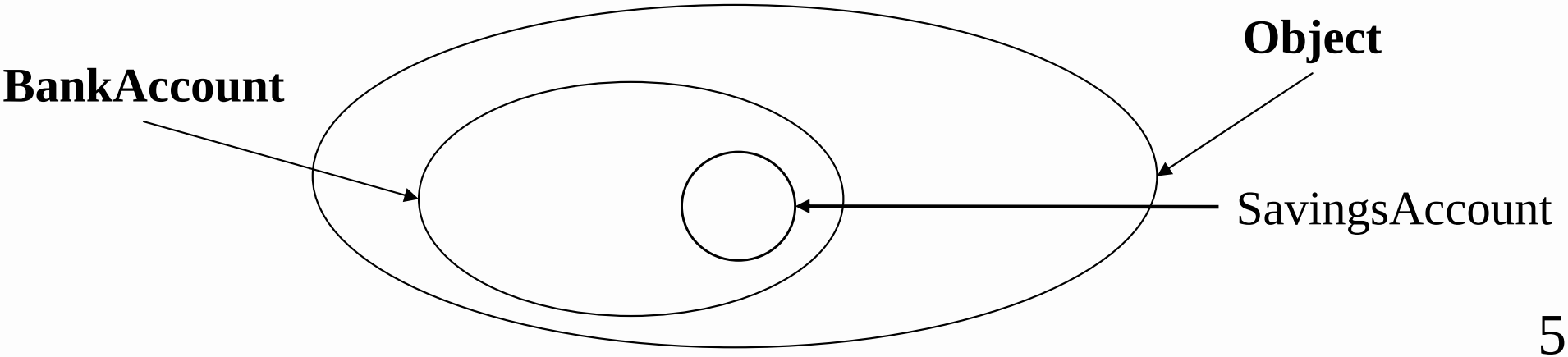
```
BankAccount acct1 = new BankAccount(500);
BankAccount acct2 = new BankAccount();
acct1.transferTo(acct2, 200);
System.out.println(acct1.getBalance());
System.out.println(acct2.getBalance());
```

```
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***

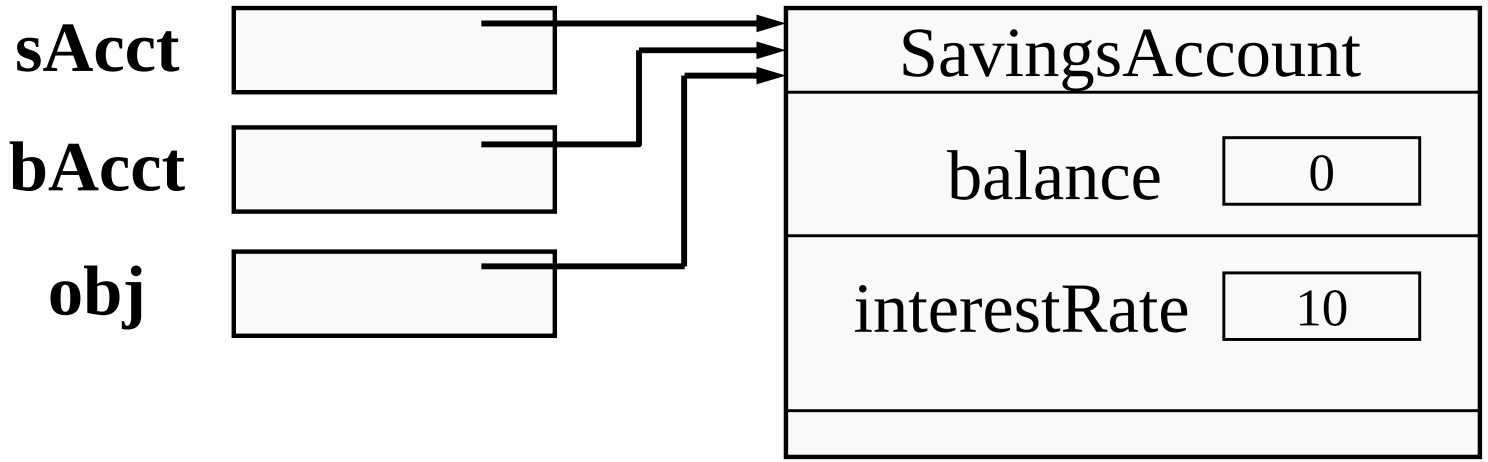
```
SavingsAccount sAcct = new SavingsAccount(10);
BankAccount bAcct = sAcct;
Object obj = sAcct;
```



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***

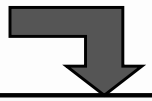


Conversione fra riferimenti

```
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 un oggetto di classe **BankAccount** o di una classe derivata da **BankAccount**
  - effettua la conversione automaticamente

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
}
```

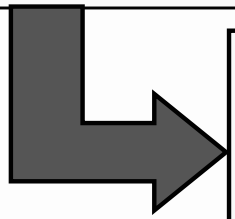
```
public void println(Object obj)
{  println(obj.toString());
}
public void println(String s)
{  ... // questo viene invocato per le
stringhe
}
```

- 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;
```



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

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

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 l'eccezione
// java.lang.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**

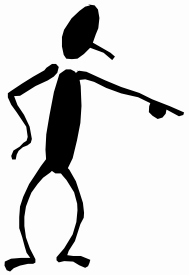
  l'invocazione di un metodo in runtime è sempre determinata dal tipo dell'oggetto effettivamente usato come parametro implicito, e NON dal tipo della variabile oggetto

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

Polimorfismo

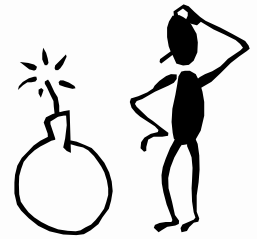
- Abbiamo già visto una forma di il polimorfismo a proposito dei ***metodi sovraccarichi***
  - l'invocazione del metodo **println()** si traduce in realtà nell'invocazione di un metodo scelto fra alcuni metodi diversi, in relazione al tipo del parametro esplicito
  - ***il compilatore decide quale metodo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

Mettere in ombra le variabili ereditate



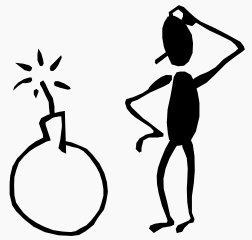
```
public class SavingsAccount extends
   BankAccount
{  ...
   public void addInterest()
   { deposit(getBalance() * interestRate/100);
   }
}
```

- 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

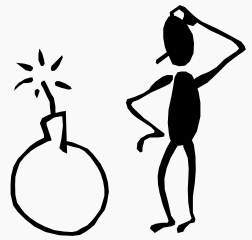
  balance has private access in BankAccount

- 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***

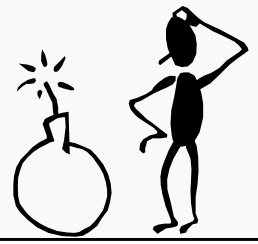
```
public class SavingsAccount extends
   BankAccount
{  ...
   public void addInterest()
   {  deposit(balance * interestRate / 100);
   }
   private double balance;
}
```

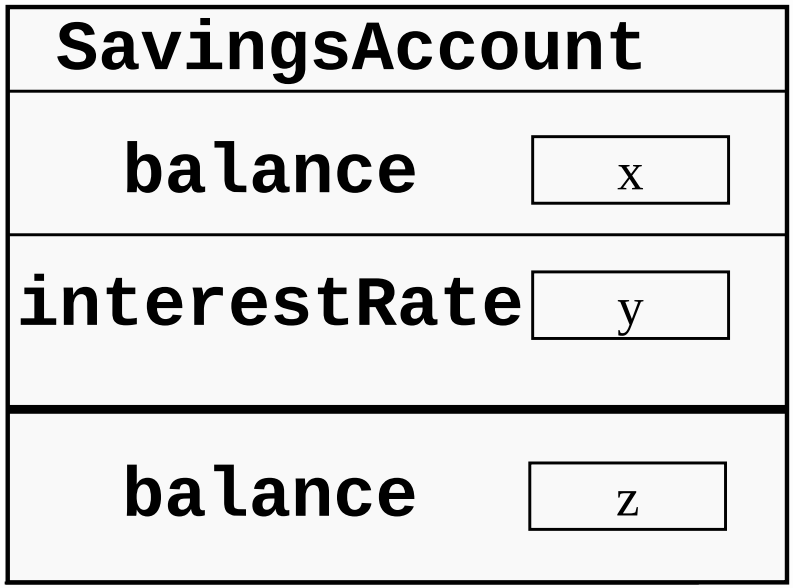


- In questo modo il compilatore non segnala alcun errore, ma ora **SavingsAccount** ha due variabili che si chiamano **balance**, ***una propria e una ereditata***, ***tra le quali non c'è alcuna relazione!***

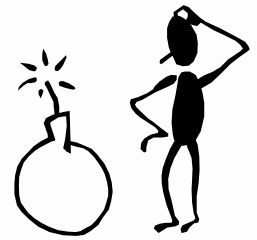
Mettere in ombra le variabili ereditate

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



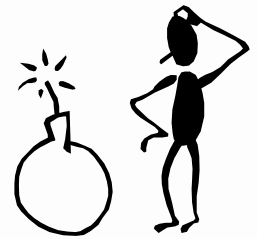


Mettere in ombra le variabili ereditate



- 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!***

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 una eccezione di tipo **SaldoInsufficienteException** nel metodo withdraw() della classe BankAccount quando si tenti di prelevare una somma maggiore del saldo del conto

```
…
if (amount > balance)
   throw new SaldoInsufficienteException(
                  “prelievo di “ + amount
                + “ eccede la”
                + “ disponibilita’ pari a “ + balance);
….
```

Progettare Eccezione

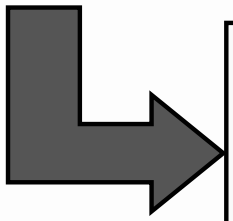
- Per costruire la nostra eccezione SaldoInsufficienteException si usa l'ereditarietà, estendendo:
- La classe Exception, se si vuole creare un'eccezione controllata
- La classe RuntimeException o una delle sue sottoclassi, se si vuole creare un'eccezione a gestione non obbligatoria
- Di una classe eccezione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

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

Progettare Eccezione

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

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



```
Exception in thread “main”
SaldoInsufficienteException: prelievo di
550 eccede la disponibilita’ pari a 500
at …
```

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!***

Lezione XXV
Me 9-Nov-2005

Ancora Ordinamento di Oggetti

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

```
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 esemplare
  - ***contiene soltanto le firme 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 e e y sono "uguali"
      - **`if (x.compareTo(y) == 0) …`**
  - ritornare un numero 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**, per ordinare un array di oggetti **BankAccount** che realizzano l'interfaccia Comparable si scrive

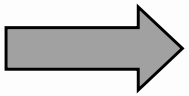
```
BankAccount[] v = new BankAccount[10];
// creazione dei singoli elementi
dell'array

v[0] = new BankAccount(10);
...
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)
```

Interfaccia Comparable Parametrica

```
public class BankAccount implements Comparable
{ public int compareTo(Object obj)
  {
    BankAccount b = (BankAccount) obj;
    return balance – b.balance;
  }
}
```

```
…
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);
}
```

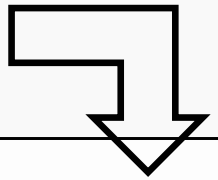
```
public class BankAccount
   implements Comparable<BankAccount>
{
  public int compareTo(BankAccount o)
  { if (balance < o.balance) return -1;
    if (balance > o.balance) return 1;
    return 0;
  }
}
```

- 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)
  { return balance – o.balance;
  }
}
```

```
…
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>
```

- Viene indicato l’errore dal compilatore

Ordinamento e ricerca “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 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

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 class BankAccount implements Comparable
{  public boolean equals(Object obj)
   {…}
   public int CompareTo(Object obj)
   {…}
}
```

- In generale negli oggetti Comparable che sovrascivono il metodo equals(), **l'ordinamento naturale** definito dal metodo **compareTo** e il **confronto di uguaglianza** fra oggetti definito dal metodo **equals** dovrebbero essere consistenti:
- **`x.equals(y) ritorna true <=> x.compareTo(y) ritorna 0`**
- Se questo non accade, nella documentazione della classe e' bene scrivere esplicitamente
  - "Notare: questa classe ha un ordinamento naturale che non è consistente con equals"

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 togli()`**
  - 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 Studente
  - Il parametro del metodo **`aggiungi()`** e' un **`Comparable`**
  - Il metodo **`togli()`** 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;
   }
}
```

Esempio: 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;
  }
}
```

Esempio: 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]);
…
```

Lezione XXVI
Gi 10-Nov-2005

Tipi di dati astratti e strutture dati

Tipi di dati astratti

- In Java si definisce un **tipo di dati astratto** con un'***interfaccia***
- Come sappiamo, un'interfaccia descrive un ***comportamento*** che sarà assunto da una classe che realizza l'interfaccia
  - è proprio quello che serve per definire un ADT
- Un tipo di dati astratto definisce ***cosa*** si può fare con una struttura dati che realizza l'interfaccia
  - la classe che rappresenta concretamente la struttura dati definisce invece ***come*** vengono eseguite le operazioni

Tipi di dati astratti

- Una ***struttura dati*** (***data structure***) è un modo sistematico di organizzare i dati in un contenitore e di controllarne le modalità d'accesso
  - in Java si definisce una struttura dati con una ***classe***
- Un ***tipo di dati astratto*** (**ADT**, ***Abstract Data Type***) è una rappresentazione ***astratta*** di una struttura dati, un modello che specifica:
  - il tipo di dati memorizzati
  - le operazioni che si possono eseguire sui dati
  - il tipo delle informazioni necessarie per eseguire le operazioni

Tipi di dati astratti

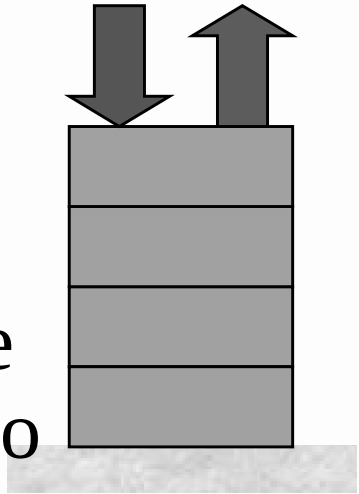
- Un tipo di dati astratto mette in generale a disposizione metodi per svolgere le seguenti azioni
  - ***inserimento*** di un elemento
  - ***rimozione*** di un elemento
  - ***ispezione*** degli elementi contenuti nella struttura
    - ***ricerca*** di un elemento all'interno della struttura
- I diversi ADT che vedremo si differenziano per le modalità di funzionamento di queste tre azioni

Il pacchetto `java.util`

- Il pacchetto **java.util** della libreria standard contiene molte definizioni di **ADT** (come **interfacce**) e molte **strutture dati** (come loro realizzazioni) in forma di **classi**
- La nomenclatura e le convenzioni usate in questo pacchetto sono, però, piuttosto diverse da quelle tradizionalmente utilizzate nella teoria dell'informazione (*purtroppo e stranamente*...)
- Quindi, proporremo un'esposizione teorica di ADT usando la terminologia tradizionale, senza usare il pacchetto **java.util** della libreria standard

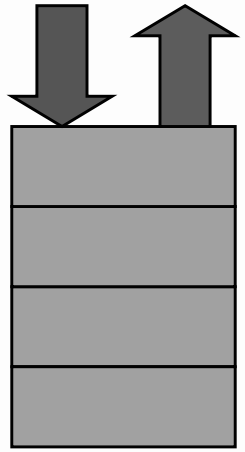
ADT Pila
(*Stack*)

Pila (*stack*)



- In una ***pila*** (***stack***) gli oggetti possono essere inseriti ed estratti secondo un comportamento definito **LIFO** (***Last In, First Out***)
  - l'ultimo oggetto inserito è il primo a essere estratto
  - il nome è stato scelto in analogia con una ***pila*** di piatti
- L'unico oggetto che può essere ispezionato è quello che si trova in cima alla pila
- Esistono molti possibili utilizzi di una struttura dati con questo comportamento
  - la JVM usa una pila per memorizzare l'elenco dei metodi in attesa durante l'esecuzione in un dato istante

Pila (*stack*)



- I browser per internet memorizzano gli indirizzi dei siti visitati in una struttura di tipo ***pila***. Quando l'utente visita un sito, l'indirizzo è inserito (***push***) nella pila. Il browser permette all'utente di saltare indietro (***pop***) al sito precedente tramite il pulsante "indietro"

- Gli editor forniscono generalmente un meccanismo di "***undo***" che cancella operazioni di modifica recente e ripristina precedenti stati del testo. Questa funzione di "undo" è realizzata memorizzando le modifiche in una struttura di tipo pila.

Pila (*stack*)

- I metodi che caratterizzano una pila sono
  - **push()** per inserire un oggetto in cima alla pila
  - **pop()** per eliminare l'oggetto che si trova in cima alla pila
  - **top()** per ispezionare l'oggetto che si trova in cima alla pila, senza estrarlo
- Infine, ogni ADT di tipo "contenitore" ha i metodi
  - **isEmpty()** per sapere se il contenitore è vuoto
  - **makeEmpty()** per vuotare il contenitore

Contenitore generico

```
public interface Container
{
  /**
    @return true se il contenitore e' vuoto, false altrimenti
  */
  boolean isEmpty();

  /**
    rende vuoto il contenitore
  */
  void makeEmpty();
}
```

```
public interface Stack extends Container
{...}
```

- Anche le interfacce, come le classi, possono essere “estese”
- Un’interfaccia eredita tutti i metodi della sua super-interfaccia
- Per realizzare un’interfaccia estesa occorre definire anche i metodi della sua super-interfaccia

Pila (*stack*)

```
public interface Stack extends Container
{
   void push(Object obj);
   Object pop() throws StackEmptyException;
   Object top() throws StackEmptyException;
}
```

- Definiremo tutti gli ADT in modo che possano genericamente contenere oggetti di tipo **Object**
  - in questo modo potremo inserire nel contenitore oggetti di qualsiasi tipo, perché qualsiasi oggetto può essere assegnato a un riferimento di tipo **Object**

Pila (*stack*)

```
public interface Stack extends Container
{  /**
     inserisce un elemento in cima alla pila
     @param obj l'elemento da inserire
   */
   void push(Object obj);

   /**
     rimuove l'elemento dalla cima della pila
     @return l'elemento rimosso
     @throws EmptyStackException se la pila e' vuota
   */
   Object pop() throws EmptyStackException;

   /**
     restituisce l'elemento in cima alla pila
     @return l'elemento in cima alla pila
     @throws EmptyStackException se la pila e' vuota
   */
   Object top() throws EmptyStackException;
}
```

Utilizzo della pila

- Per evidenziare la potenza della definizione di tipi di dati astratti come interfacce, supponiamo che qualcun altro abbia progettato la seguente classe

```
public class StackX implements Stack
{  ...
}
```

- Senza sapere come sia realizzata **StackX**, possiamo usarne un esemplare mediante il suo comportamento astratto definito in **Stack**
- Allo stesso modo, possiamo usare un esemplare di un'altra classe **StackY** che realizza **Stack**

```
public class StackY implements Stack
{  ...
}
```

```
public class ReverseStack // UN ESEMPIO
{ public static void main(String[] args)
  {  Stack s = new StackX();
     s.push("Pippo");
     s.push("Pluto");
     s.push("Paperino");
     printAndClear(s);
     System.out.println();
     s.push("Pippo");
     s.push("Pluto");
     s.push("Paperino");
     printAndClear(reverseAndClear(s));
  }
  private static Stack reverseAndClear(Stack st)
  {  Stack p = new StackY();
     while (!st.isEmpty())
        p.push(st.pop());
     return p;
  }
  private static void printAndClear(Stack st)
  {  while (!st.isEmpty())
       System.out.println(st.pop());
  }
}
```

```
Paperino
Pluto
Pippo

Pippo
Pluto
Paperino
```

Realizzazione della pila

- Per ***realizzare una pila*** è facile ed efficiente usare una struttura di tipo ***array*** "riempito solo in parte"
- Il solo problema che si pone è ***cosa fare quando l'array è pieno*** e viene invocato il metodo **push()**
  - la prima soluzione proposta prevede il ***lancio di un'eccezione***
  - la seconda soluzione proposta usa il ***ridimensionamento dell'array***
- Un altro errore da segnalare con un'eccezione è l'invocazione di **pop()** o **top()** quando la pila è vuota

Eccezioni nella pila

- Definiamo due eccezioni (che sono classi), **EmptyStackException** e **FullStackException**, come estensione di **RuntimeException**, in modo che chi usa la pila ***non sia obbligato a gestirle***

```
public class EmptyStackException
        extends RuntimeException
{
}
public class FullStackException
        extends RuntimeException
{
}
```

Definizione di un'eccezione

```
public class EmptyStackException
        extends RuntimeException
{  }
```

- La classe è **vuota**, a cosa serve?
  - serve a definire un nuovo tipo di dato (un'eccezione) che ha le stesse identiche caratteristiche della superclasse da cui viene derivato, ma di cui interessa porre in evidenza il ***nome***, che contiene l'informazione di identificazione dell'errore
- Con le eccezioni si fa spesso così
  - in realtà la classe non è vuota, perché contiene tutto ciò che eredita dalla sua superclasse

```
public class FixedArrayStack implements Stack
{
  private final int CAPACITY = 100;

  // v è un array riempito solo in parte
  protected Object[] v;  // considerare
  protected int vSize;   // protected
                         // uguale a private?
  public FixedArrayStack()
  {
    v = new Object[CAPACITY]; // 100 è il
    // numero scelto per l'applicazione

    /* per rendere vuota la struttura, invoco
       il metodo makeEmpty, è sempre meglio
       evitare di scrivere codice ripetuto */
    makeEmpty();
  }
  ...
```

```
    // dato che Stack estende Container,
    // occorre realizzare anche i suoi metodi

    /** rende vuota una pila */
    public void makeEmpty()
    {  vSize = 0;
    }

    /**
     restituisce true se la pila e' vuota
     @return true se vuota, false altrimenti
    */
    public boolean isEmpty()
    {  return (vSize == 0);
    }

    ...
}
```

```
public class FixedArrayStack implements Stack
{  ...
   public void push(Object obj)
   {  if (vSize == v.length)
         throw new FullStackException();
      v[vSize++] = obj;
   }

   public Object top() throws EmptyStackException
   {  if (isEmpty())
         throw new EmptyStackException();
      return v[vSize - 1];
   }

   public object pop() throws EmptyStackException
   {  Object obj = top();
      vSize--;
      v[vSize] = null; //garbage collection
      return obj;
   }
}
```

Pile di oggetti e pile di numeri

Pile di oggetti e pile di numeri

- Abbiamo visto che la pila così definita è in grado di gestire dati di qualsiasi tipo, cioè riferimenti ad oggetti di qualsiasi tipo (stringhe, conti bancari…)
- Non è però in grado di gestire dati dei tipi fondamentali definiti nel linguaggio Java (**int**, **double**, **char**…)
  - tali tipi di dati NON sono oggetti e NON possono essere assegnati a variabili riferimento di tipo **Object**
- Come possiamo gestire una pila di numeri?

Pile di numeri

- Possiamo ridefinire tutto

```
public interface IntStack
      extends Container
{  void push(int obj);
   int top();
   int pop();
}
```

```
public class FixedArrayIntStack implements IntStack
{  protected int[] v;  protected int vSize;
   public FixedArrayIntStack()
   {  v = new int[100]; makeEmpty(); }
   public void makeEmpty() {  vSize = 0; }
   public boolean isEmpty(){  return (vSize == 0); }
   public void push(int obj) throws FullStackException
   {  if(vSize == v.length)
         throw new FullStackException();
      v[vSize++] = obj; }
   public int top() throws EmptyStackException
   {  if (isEmpty()) throw new EmptyStackException();
      return v[vSize - 1]; }
   public int pop()
   {  int obj = top(); vSize--; return obj; }
}
```

Pile di numeri

- La ridefinizione della pila per ogni tipo di dato fondamentale ha alcuni **svantaggi**
  - occorre replicare il codice nove volte (i tipi di dati fondamentali sono otto), con poche modifiche
  - non esiste più il tipo di dati astratto **Stack**, ma esisterà **IntStack**, **DoubleStack**, **CharStack**, **ObjectStack**
- Cerchiamo un'alternativa, ponendoci un problema
  - è possibile "trasformare" un numero intero (o un altro tipo di dato fondamentale di Java) in un oggetto?
- La risposta è affermativa
  - si usano le ***classi involucro*** (***wrapper***)

Classi involucro

- Per dati **int** esiste la classe involucro **Integer**
  - il costruttore richiede un parametro di tipo **int** e crea un oggetto di tipo **Integer** che contiene il valore intero, "avvolgendolo" con la struttura di un oggetto

```
Integer intObj = new Integer(2);
Object obj = intObj; // lecito
```

  - gli oggetti di tipo **Integer** sono ***immutabili***
  - per conoscere il valore memorizzato all'interno di un oggetto di tipo **Integer** si usa il metodo non statico **intValue**, che restituisce un valore di tipo **int**

```
int x = intObj.intValue(); // x vale 2
```

Classi involucro

- Esistono classi involucro per tutti i tipi di dati fondamentali di Java, con i nomi uguali al nome del tipo fondamentale ma iniziale maiuscola
  - eccezioni di nomi: **Integer** e **Character**
- Ogni classe fornisce un metodo per ottenere il valore contenuto al suo interno, con il nome corrispondente al tipo
  - es: **booleanValue( )**, **charValue( )**, **doubleValue( )**
- Tutte le classi involucro si trovano nel pacchetto **java.lang** e realizzano l'interfaccia **Comparable**

Classi Involucro (Wrappers)

- Boolean
- Byte
- Character
- Short
- Integer
- Long
- Float
- Double

```
// tipo fondamentale del linguaggio
double x = 3.5;

// Oggetto che incapsula un tipo
// fondamentale
Double d = new Double(3.5);
```

```
Double d = new Double(3.5);
// d e' un oggetto di classe Double
```

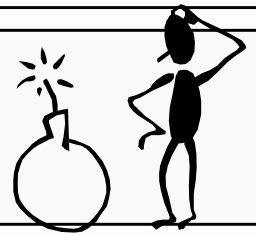
```
// auto-impacchettamento
// auto-boxing
Double d = 3.5 //java 5.0
```

```
double x = d.doubleValue();

// auto-unboxing
double x = d  //java 5.0
Double e = 1 + d;
```

Classi Involucro

```
// Oggetto che incapsula un tipo fondamentale
Double d = new Double(3.5);
String s = d.toString();   // OK
```

```
// tipo fondamentale
double x = 3.5;
String s = x.toString();
```



```
public class WrapperTester
{  public static void main(String[] args)
   {
      Double d = 3.5;
      String s = d.toString();
      System.out.println(s);

      double x = 3.5;
      s = x.toString();
      System.out.println(s);
    }
}
```

```
WrapperTester.java:9:double cannot be dereferenced
    s = x.toString();
        ^
```

Pila con ridimensionamento

- Definiamo una pila che non generi mai l'eccezione **FullStackException**

```
public class GrowingArrayStack implements Stack
{
  public void push(Object obj)
  {  if (vSize == v.length)
        v = resize(v, 2 * vSize);
     v[vSize++] = obj;
  }
  private Object[] resize(…) {…}

  ... // tutto il resto è identico!
}
```

- Possiamo evitare di riscrivere tutto il codice di **FixedArrayStack** in **GrowingArrayStack**?

Pila con ridimensionamento

```
public class GrowingArrayStack extends FixedArrayStack
{  public void push(Object obj)
   {  if (vSize == v.length)
         v = resize(v, 2 * vSize);
      super.push(obj);
   }
   private Object[] resize(…) {…}
}
```

- Il metodo **push()** sovrascritto deve poter accedere alle **variabili di esemplare** della superclasse
- Questo è consentito dalla definizione **protected**
  - le variabili **protected** sono come **private**, ma ***vi si può accedere anche dalle classi derivate (e dalle classi dello stesso pacchetto!!!)***

Accesso protected

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

Non usare variabili d'istanza con specificatore di accesso `protected`, se non in casi particolari ! ! !

Prestazioni della pila

- Il tempo di esecuzione di ogni operazione su una ***pila realizzata con array di dimensioni fisse*** è ***costante***, cioè non dipende dalla dimensione **n** della struttura dati stessa (non ci sono cicli…)
  - si noti che ***le prestazioni dipendono dalla definizione della struttura dati*** e non dalla sua interfaccia…
  - per valutare le prestazioni è necessario conoscere il codice che realizza le operazioni!

Prestazioni della pila

- Un’operazione eseguita in un tempo costante, cioè in un tempo che non dipende dalle dimensioni del problema, ha un andamento asintotico **`O(1)`**, perché
  - eventuali costanti moltiplicative vengono trascurate
- Ogni operazione eseguita su **FixedArrayStack** è quindi **`O(1)`**

Prestazioni della pila

- Nella realizzazione ***con array ridimensionabile***, l'unica cosa che cambia è l'operazione **push()**
  - "**alcune volte**" richiede un tempo `O(n)`
    - tale tempo è necessario per copiare tutti gli elementi nel nuovo array, all'interno del metodo **resize()**
    - il ridimensionamento viene fatto ogni `n` operazioni
  - cerchiamo di valutare il ***costo medio*** di ciascuna operazione
    - tale metodo di stima si chiama ***analisi ammortizzata*** delle prestazioni asintotiche

Analisi ammortizzata

- Dobbiamo calcolare il valore medio di **n** operazioni, delle quali
  - **`n-1`** richiedono un tempo **`O(1)`**
  - **una** richiede un tempo **`O(n)`**

$$\langle T(n) \rangle = [(n-1)*O(1) + O(n)]/n$$
$$= O(n)/n = O(1)$$

- Distribuendo il tempo speso per il ridimensiomento in parti uguali a tutte le operazioni **push()**, si ottiene quindi ancora **`O(1)`**

Analisi ammortizzata

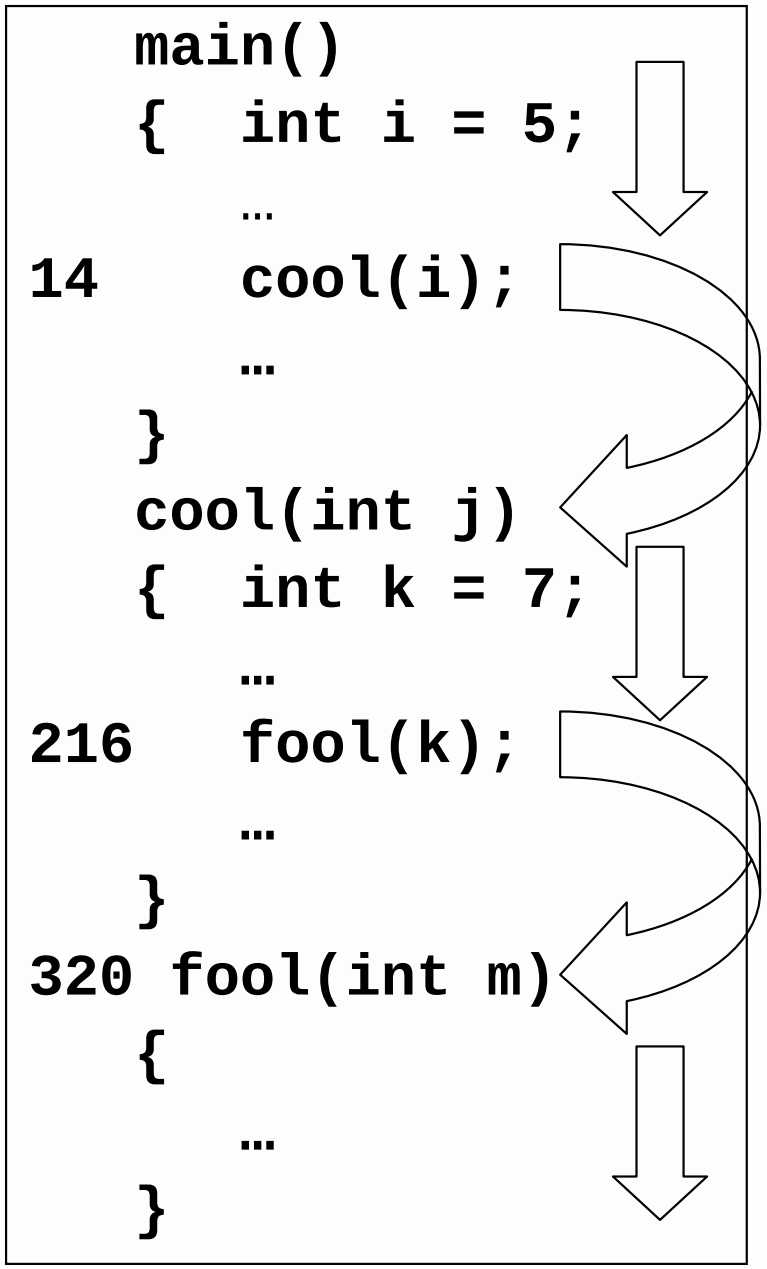
- Le prestazioni di **push()** con ridimensionamento rimangono **O(1)** per qualsiasi costante moltiplicativa usata per calcolare la nuova dimensione, anche diversa da 2
- Se, invece, si usa una costante ***additiva***, cioè la dimensione passa da **n** a **n + k**, si osserva che su **n** operazioni di inserimento quelle "lente" sono **n/k**

$$
\begin{aligned}
\langle T(n) \rangle &= [(n - n/k) * O(1) + (n/k) * O(n)]/n \\
&= [O(n) + n * O(n)]/n \\
&= O(n)/n + O(n) \\
&= O(1) + O(n) = O(n)
\end{aligned}
$$

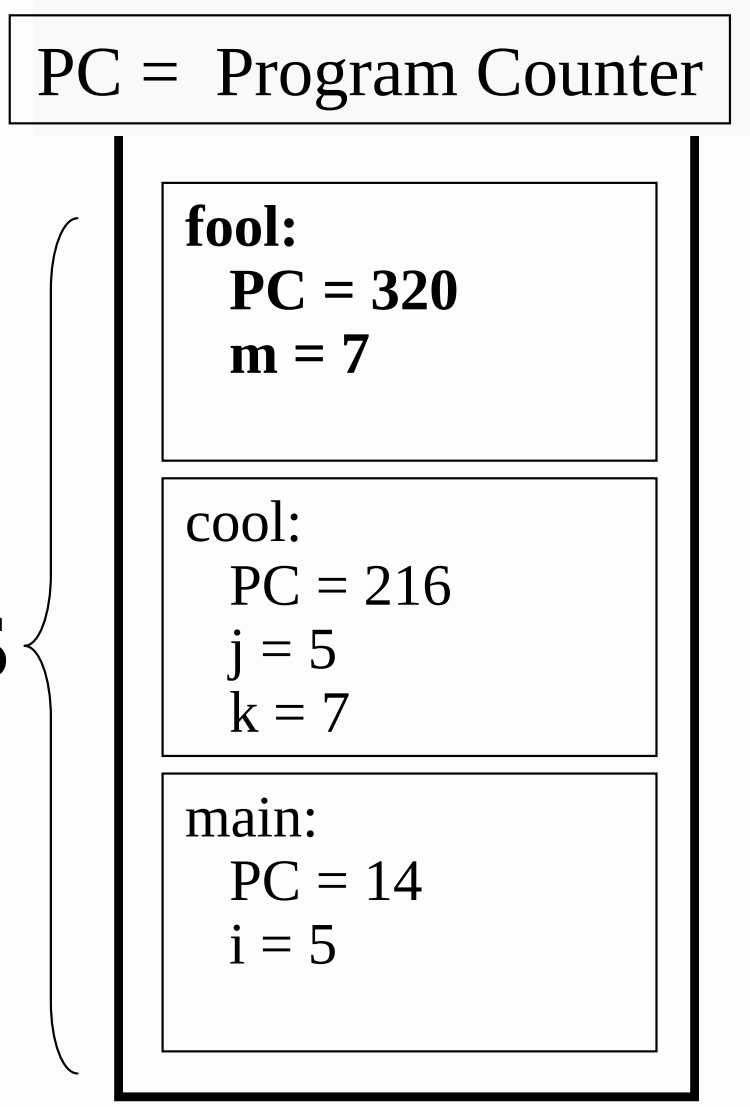
Pile nella Java Virtual Machine

- Ciascun programma java in esecuzione ha una propria pila chiamata **Java Stack** che viene usata per mantenere traccia delle **variabili locali** e di altre importanti informazioni relative ai metodi, man mano che questi sono invocati
- Più precisamente, durante l'esecuzione di un programma, JVM mantiene uno **stack** i cui elementi sono **descrittori** dello stato corrente dell'invocazione dei metodi (che non sono terminati)
- I descrittori sono denominati ***Frame.*** A un certo istante durante l'esecuzione, ciascun metodo sospeso ha un frame nel Java Stack

Java Stack (Runtime Stack)



Programma Java in memoria



Java Stack