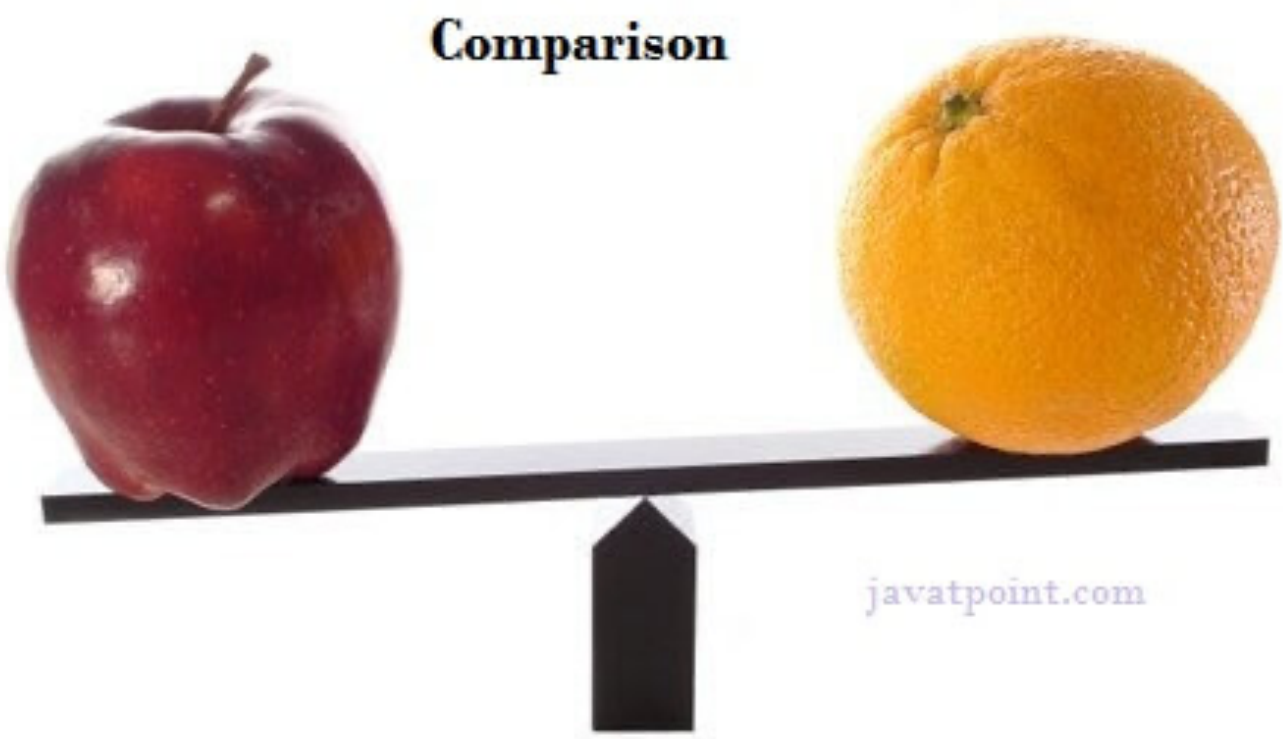
Decisioni: Confronto di stringhe



Confronto di stringhe

- Per confrontare stringhe si usa il metodo **equals**

```
if (s1.equals(s2))
```

- Per confrontare stringhe ignorando la differenza tra maiuscole e minuscole si usa

```
if (s1.equalsIgnoreCase(s2))
```

- Non usare **mai** l'operatore di uguaglianza per confrontare stringhe! **Usare sempre equals**
  - Se si usa l'operatore uguaglianza, il successo del confronto sembra essere deciso in maniera "casuale"
  - In realtà dipende da come è stata progettata la Java Virtual Machine e da come sono state costruite le due stringhe

Attenzione perché NON è un errore di sintassi

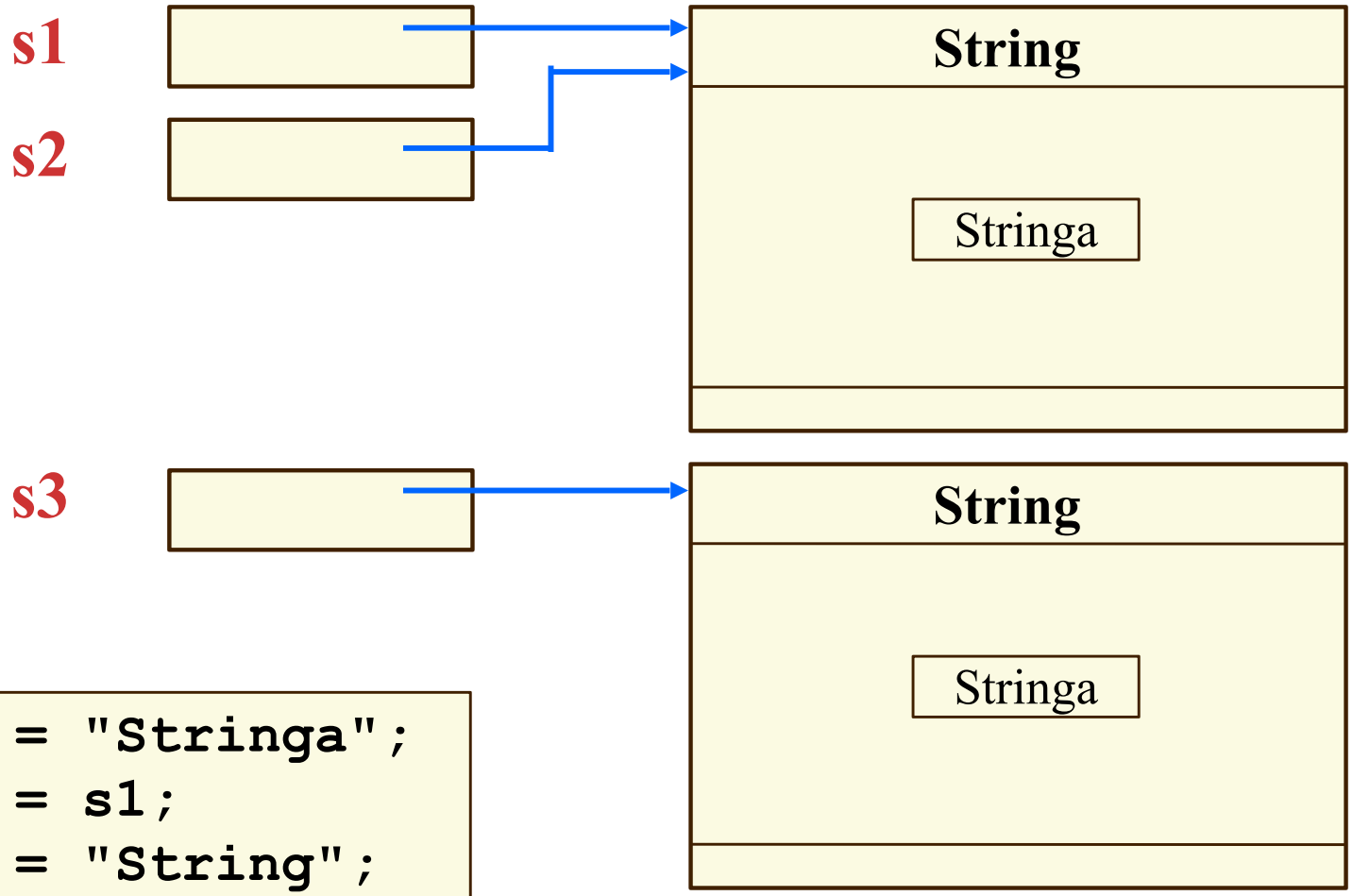
Confronto di stringhe

- Confrontando con l'operatore di uguaglianza due riferimenti a stringhe si verifica se i riferimenti puntano allo stesso oggetto stringa

```
String s1 = "Stringa";
String s2 = s1;
String s3 = "String";
s3 = s3 + "a"; // s3 contiene "Stringa"
```

- Il confronto **s1 == s2** è **vero**, perché puntano allo stesso oggetto stringa
- Il confronto **s1 == s3** è **falso**, perché puntano ad oggetti diversi, anche se tali oggetti hanno lo stesso contenuto (sono "identici")

Confronto di stringhe




```
String s1 = "Stringa";
String s2 = s1;
String s3 = "String";
s3 = s3 + "a";
```

Confronto di stringhe

- Confrontando invece con il metodo **equals** due riferimenti, si verifica se i riferimenti **puntano** a stringhe con lo stesso contenuto

```
String s1 = "Stringa";
String s2 = s1;
String s3 = "String";
s3 = s3 + "a"; // s3 contiene "Stringa"
```

- Il confronto **s1.equals(s3)** è **vero**, perché puntano a due oggetti String identici

- Nota: per verificare se un riferimento si riferisce a **null**, si può usare invece l'operatore di uguaglianza e non il metodo **equals**

```
if (s == null)
```

Ordinamento lessicografico

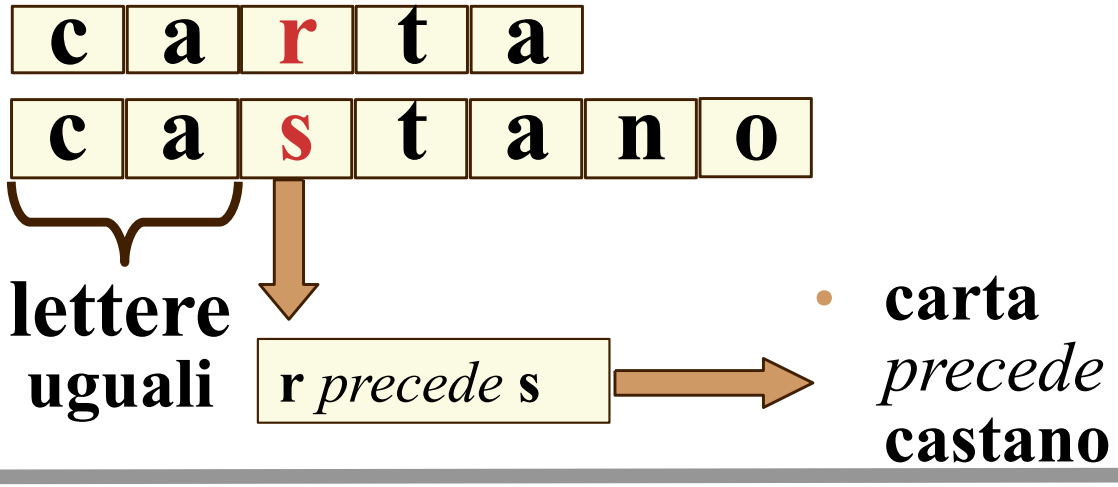
- Se due stringhe sono diverse, è possibile conoscere la relazione che intercorre tra loro secondo **l'ordinamento lessicografico**, simile al comune ordinamento alfabetico
- Il confronto lessicografico tra stringhe si esegue con il metodo **compareTo**

```
if (s1.compareTo(s2) < 0)
```

- Il metodo compareTo restituisce un valore int
  - negativo se s1 precede s2 nell'ordinamento
  - positivo se s1 segue s2 nell'ordinamento
  - zero se s1 e s2 sono identiche

Confronto lessicografico

- Partendo dall'inizio delle stringhe, si confrontano 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



Confronto lessicografico

- Il confronto lessicografico genera un ordinamento simile a quello di un comune dizionario
- …con qualche **differenza**…
  - Tra i caratteri non ci sono solo le lettere
    - i numeri precedono le lettere
    - tutte le lettere maiuscole precedono tutte le lettere minuscole
    - il carattere di "**spazio bianco**" precede tutti gli altri caratteri
- L'ordinamento lessicografico è definito dallo standard **Unicode**, http://www.unicode.org

Confrontare oggetti

Confronto di oggetti

- Come per le stringhe, l'operatore **==** tra due variabili oggetto verifica se i due riferimenti puntano allo stesso oggetto, e **non** l'uguaglianza tra oggetti
- Il metodo **equals** si può applicare a **qualsiasi oggetto**
  - È definito nella classe **Object**, da cui derivano tutte le classi
    - il metodo **equals** di **Object** usa l'operatore di uguaglianza
  - Ma è compito di ciascuna classe **ridefinire** il metodo **equals**, come fa la classe **String**
- Il metodo **equals** di ciascuna classe deve effettuare il **confronto delle caratteristiche** (variabili di esemplare) degli oggetti di tale classe
  - Per ora usiamo **equals** solo per classi di libreria standard
  - non facciamo confronti tra oggetti di classi definite da noi

Ordinamento di oggetti

- Il metodo **compareTo** visto per le stringhe può essere applicato a **molti altri oggetti**, (ma **non** tutti perché **non** è definito nella classe **Object**)
- È compito di ciascuna classe **definire** in maniera opportuna il metodo **compareTo**, come fa la classe **String**, secondo una opportuna nozione di ordinamento
- Il metodo **compareTo** di una classe restituirà sempre un valore int
  - **negativo** se obj1 precede obj2 nell'ordinamento
  - **positivo** se obj1 segue obj2 nell'ordinamento
  - **zero** se obj1 e obj2 sono identici

```
if (obj1.compareTo(obj2) < 0)
```

È tutto chiaro? …

1. Qual è il valore di **`s.length()`** se **`s`** contiene **(a)** un riferimento alla stringa vuota `""` **(b)** un riferimento alla stringa `" "` contenente solo uno spazio **(c)** il valore **`null`** ?
2. Quali di questi confronti hanno errori di sintassi? Quali hanno poca utilità dal punto di vista logico?

```
String a = "1";
String b = "one";
double x = 1;
double y = 3 * (1.0 / 3);
```

(a) `a == "1";` (b) `a == null;` (c) `a.equals("");`
(d) `a == b;` (e) `a == x;` (f) `x == y;`
(g) `x - y == null;` (h) `x.equals(y);`

Alternative multiple (capitolo 5)

Sequenza di confronti

- Se si hanno più di due alternative, si usa una **sequenza di confronti**

```
if (richter >= 8)
   System.out.println("Terremoto molto forte");
else if (richter >= 6)
   System.out.println("Terremoto forte");
else if (richter >= 4)
   System.out.println("Terremoto medio");
else if (richter >= 2)
   System.out.println("Terremoto debole");
else if (richter >= 0)
   System.out.println("Terremoto molto debole");
else
   System.out.println("Numeri negativi non validi");
```

Significa che
6<=richter<8

Sequenza di confronti

- Il codice seguente non funziona, perché stampa "Terremoto molto debole" per qualsiasi valore di richter
- Se si fanno confronti di tipo "**maggiore di**" si devono scrivere ***prima i valori più alti***, e viceversa

```
if (richter >= 0)     // NON FUNZIONA!
   System.out.println("Terremoto molto debole");
else if (richter >= 2)
   System.out.println("Terremoto debole");
else if (richter >= 4)
   System.out.println("Terremoto medio");
else if (richter >= 6)
   System.out.println("Terremoto forte");
else
   System.out.println("Terremoto molto forte");
```

Sequenza di confronti

- Se non si rendono **mutuamente esclusive** le alternative, usando le clausole **else**, non funziona
  - se richter vale 3, stampa **sia** “Terremoto debole” **sia** “Terremoto molto debole”

```
if (richter >= 8)     // NON FUNZIONA!
   System.out.println("Terremoto molto forte");
if (richter >= 6)
   System.out.println("Terremoto forte");
if (richter >= 4)
   System.out.println("Terremoto medio");
if (richter >= 2)
   System.out.println("Terremoto debole");
if (richter >= 0)
   System.out.println("Terremoto molto debole");
```

Diramazioni annidate

Diramazioni annidate

- Aliquote per categorie d'imposta federali (1992)
  - Per semplicità usiamo il sistema fiscale americano e non quello italiano

Se il vostro stato civile è "non coniugato"		Se il vostro stato civile è "coniugato"	
Scaglione fiscale	**Aliquota**	**Scaglione fiscale**	**Aliquota**
\$ 0 ... \$ 21 450	15%	\$ 0 ... \$ 35 800	15%
Reddito superiore a \$ 21 450 fino a \$ 51 900	28%	Reddito superiore a \$ 35 800 fino a \$ 86 500	28%
Reddito superiore a \$ 51 900	31%	Reddito superiore a \$ 86 500	31%

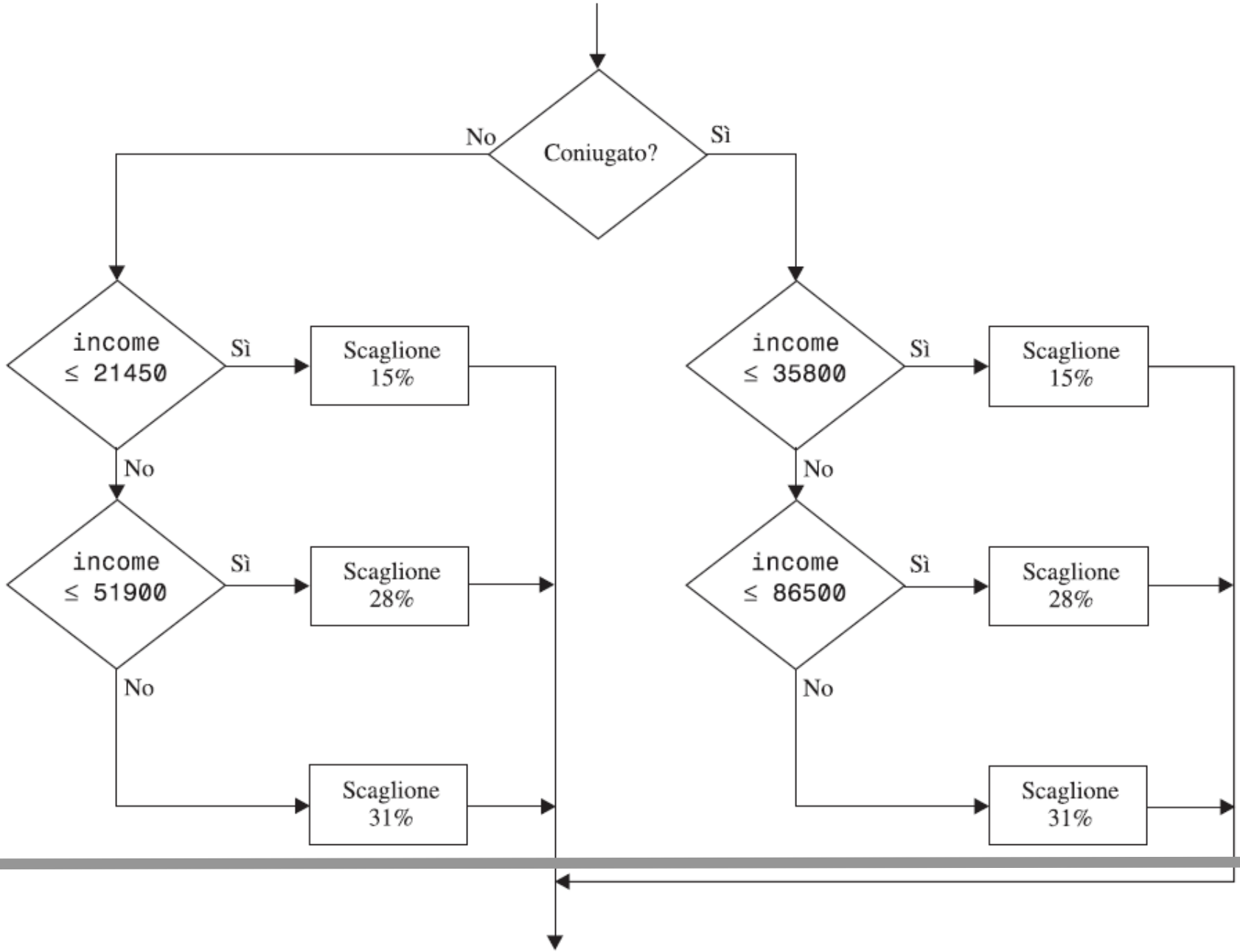
- Ci sono **due livelli** nel processo decisionale
  - **Prima** dobbiamo scegliere lo stato civile
  - **Poi**, per ciascuno stato civile, dobbiamo scegliere lo scaglione di reddito

Diramazioni annidate

- Usiamo **due** diramazioni annidate
  - un enunciato **if**
  - **all'interno del corpo** di un altro enunciato **if**

```
if (status == SINGLE)
{   if (income <= SINGLE_BRACKET1)
       ...
    else if (income <= SINGLE_BRACKET2)
           ...
    else
           ...
}
else
{   if (income <= MARRIED_BRACKET1)
...
    else if (income <= MARRIED_BRACKET2)
           ...
    else
           ...
}
```

Diramazioni annidate

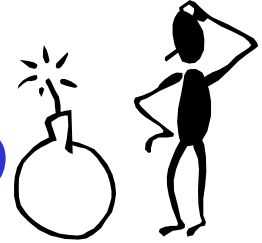


È tutto chiaro? …

1. L'enunciato if/else/else per la scala Richter verifica prima i valori più elevati. Si può invertire l'ordine delle verifiche?
2. Alcuni contestano l'applicazione di aliquote più elevate ai redditi più elevati perché dopo aver pagato le tasse si potrebbe rimanere con meno soldi pur avendo guadagnato di più. Dove è l'errore in questo ragionamento?

Il problema dell'else sospeso

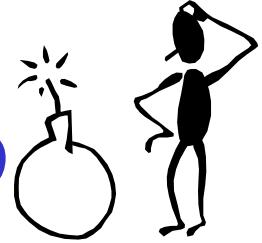
Il problema dell'else sospeso



- Nell'esempio seguente i livelli di rientro suggeriscono che la clausola **else** si riferisca al primo enunciato **if**
  - ma il compilatore ignora i rientri!
  - Il risultato ottenuto è il **contrario** di ciò che si voleva
- La regola sintattica è che **una clausola else appartiene sempre all'enunciato if più vicino**

```
if (richter >=0)
    if (richter <= 4)
        System.out.println("Terremoto leggero");
else // non funziona!!!
    System.out.println("Numeri negativi non validi");
```

Il problema dell'else sospeso



- Per ottenere il risultato voluto, bisogna "nascondere" il secondo enunciato if all'interno di un blocco di enunciati, inserendo una coppia di parentesi graffe
  - per evitare problemi con l'else sospeso, è meglio **racchiudere sempre** il corpo di un enunciato if tra parentesi graffe, anche quando sono inutili

```
if (richter >=0)
{     if (richter <= 4)
          System.out.println("Terremoto leggero");
}
else
     System.out.println("Numeri negativi non validi");
```

Visibilità delle variabili

- Se il valore finale di una variabile usata nel corpo di un enunciato if/else deve essere visibile al di fuori del corpo, bisogna definirla **prima** dell'enunciato if/else
- Poiché una variabile definita nel corpo di un enunciato if/else non è più definita dopo di esso, è possibile (e comodo) usare di nuovo **lo stesso nome** successivamente nel codice

```
double b = ...;
if (b < 10)
{   double c = ...;
    ...modifica b e c ...
}
// qui b è visibile, mentre c non lo è
```

Espressioni booleane

Il tipo di dati booleano

- Ogni espressione in Java ha un **valore**
  - **x + 10** espressione aritmetica, valore **numerico**
  - **x < 10** espressione relazionale, **valore booleano**
- Un'espressione relazionale può avere solo due valori:
  - **vero** o **falso** (true o false)

- I valori **true** e **false** non sono numeri, né oggetti, né classi: appartengono a un tipo di dati diverso, detto **booleano**
  - è un **tipo fondamentale** in Java, come quelli numerici
  - Il nome deriva da quello del matematico George Boole (1815-1864), pioniere della logica

Le variabili booleane

- Il tipo di dati **boolean**, come tutti gli altri, consente la definizione di variabili e l'assegnazione di valori

```
boolean a = true;
```

- A volte è comodo utilizzare **variabili booleane** per memorizzare valori di passaggi intermedi in cui è opportuno scomporre verifiche troppo complesse
- Altre volte l'uso di una variabile booleana rende più leggibile il codice

```
int x;
...
a = x>0;
```

- Spesso le variabili booleane sono chiamate **flag** (bandiere), perché possono assumere soltanto due valori: su e giù, come una bandiera

Metodi predicativi

- Così vengono chiamati metodi che restituiscono valori di tipo **booleano**
  - Solitamente **verificano** una condizione sullo stato di un oggetto
  - Solitamente iniziano con "**is**" oppure "**has**"
- La classe **Character** contiene metodi predicativi statici
  - isDigit, isLetter, isUpperCase, isLowerCase
- La classe **Scanner** contiene metodi predicativi per verificare il contenuto dell'input:
  - hasNextInt, hasNextDouble, …
- Metodi predicativi possono essere usati come condizioni di enunciati **if**

```
if (Character.isUpperCase(ch))
// esegue se il carattere ch è maiuscolo
```

Operatori booleani

Gli operatori booleani o logici

- Gli operatori booleani o logici servono a svolgere **operazioni su valori booleani**

```
if (x > 10 && x < 20)
// esegue se x è maggiore di 10 e minore di 20
```

- L'operatore **&&** (*and, e*) combina due o più condizioni in una sola, che risulta vera **se e solo se sono tutte vere**
- L'operatore **||** (*or, oppure*) combina due o più condizioni in una sola, che risulta vera **se e solo se almeno una è vera**
- L'operatore **!** (*not, non*) **inverte** il valore di un'espressione booleana

Gli operatori booleani o logici

A	*B*	*A && B*
true	true	true
true	false	false
false	*qualsiasi*	false

Tabelle di verità

A	*B*	*A || B*
true	*qualsiasi*	true
false	true	true
false	false	false

A	*!A*
true	false
false	true

Valutazione pigra

Valutazione di operatori logici

- Più operatori booleani possono essere usati in un'unica espressione

```
if ((x > 10 && x < 20) || x > 30)
```

- In Java la valutazione di un'espressione con operatori booleani viene effettuata con una strategia detta **cortocircuito** (o **valutazione pigra**)
  - La valutazione dell'espressione termina appena è possibile decidere il risultato

- Nel caso sopra, se **x** = **15**, l'ultima condizione non viene valutata, perché sicuramente l'espressione è vera

Precedenza degli operatori logici

- In un'espressione booleana con più operatori, la valutazione viene fatta **da sinistra a destra**, dando la precedenza all'operatore not, poi all'operatore and, infine all'operatore or
- L'ordine di valutazione può comunque essere alterato dalle **parentesi tonde**
  - Meglio usare qualche parentesi non necessaria piuttosto che sbagliare...

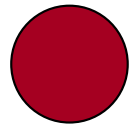
```
if (!(x < 0 || x > 10))
// esegue se x è compreso tra 0 e 10,
// estremi inclusi
```

```
if (!x < 0 || x > 10)
// esegue se x è maggiore o uguale a 0
```

Precedenza degli operatori

Priority	Operators	Operation	Associativity
1	[ ]	array index	left
	()	method call	
	.	member access	
2	++	pre- or postfix increment	right
	--	pre- or postfix decrement	
	+ -	unary plus, minus	
	~	bitwise NOT	
	!	boolean (logical) NOT	
	(type)	type cast	
	new	object creation	
3	* / %	multiplication, division, remainder	left
4	+ -	addition, substraction	left
	+	string concatenation	
5	<<	signed bit shift left	left
	>>	signed bit shift right	
	>>>	unsigned bit shift right	
6	< <=	less than, less than or equal to	left
	> >=	greater than, greater than or equal to	
	instanceof	reference test	
6	< <=	less than, less than or equal to	left
	> >=	greater than, greater than or equal to	
	instanceof	reference test	
7	==	equal to	left
	!=	not equal to	
8	&	bitwise AND	left
	&	boolean (logical) AND	
9	^	bitwise XOR	left
	^	boolean (logical) XOR	
10	|	bitwise OR	left
	|	boolean (logical) OR	
11	&&	boolean (logical) AND	left
12	||	boolean (logical) OR	left
13	? :	conditional	right
14	=	assignment	right
	*= /= += -= %= <<= >>= >>>= &= ^= |=	combinated assignment (operation and assignment)	

Associatività degli operatori



- Cosa succede quando gli operatori in un'espressione hanno la stessa precedenza?
  - Esempio: gli operatori * e **%** hanno la stessa precedenza

```
int n = 2 * x % 2;
```

- Per ogni gruppo di operatori con la stessa precedenza sono definite regole di associatività, che può essere **da sinistra a destra** o **da destra a sinistra**.
  - Gli operatori * e **%** hanno associatività da sinistra a destra, ovvero la prima operazione che verrà effettuata sarà quella il cui operatore si trova più a sinistra.
  - Nell'esempio, la prima operazione ad essere effettuata sarà quindi l'operazione di moltiplicazione

È tutto chiaro? …

- In quali condizioni il seguente enunciato visualizza “false”?

```
System.out.println(x > 0 || x < 0);
```

- Riscrivere l’espressione seguente evitando di effettuare il confronto con il valore false

```
if (Character.isDigit(ch) == false)
```

Leggi di De Morgan

- Due leggi utili per **semplificare** espressioni logiche
- Stabiliscono un criterio per convertire un'espressione "negata" in una espressione "affermata"

1a legge `!(A && B)` è uguale a `!A || !B`

2a legge `!(A || B)` è uguale a `!A && !B`

- Gli operatori **not** vengono **spostati** su ciascuna delle espressioni coinvolte
- Gli operatori **and** e gli operatori **or** vengono **scambiati**

- **Esempio**: queste due espressioni sono equivalenti (abbiamo usato la seconda legge di De Morgan)

```
if (!(x < 0 || x > 10))
```

```
if ( x >=0 && x <= 10 )
```

Leggi di De Morgan: dimostrazione

- Si dimostrano direttamente scrivendo le tabelle di verità
  - Prima legge

false	false	false	true	false	false	true	true	true
false	true	false	true	false	true	true	false	true
true	false	false	true	true	false	false	true	true
true	true	true	false	true	true	false	false	false

  - Seconda legge

false	false	false	true	false	false	true	true	true
false	true	true	false	false	true	true	false	false
true	false	true	false	true	false	false	true	false
true	true	true	false	true	true	false	false	false

L’enunciato switch

L'enunciato switch

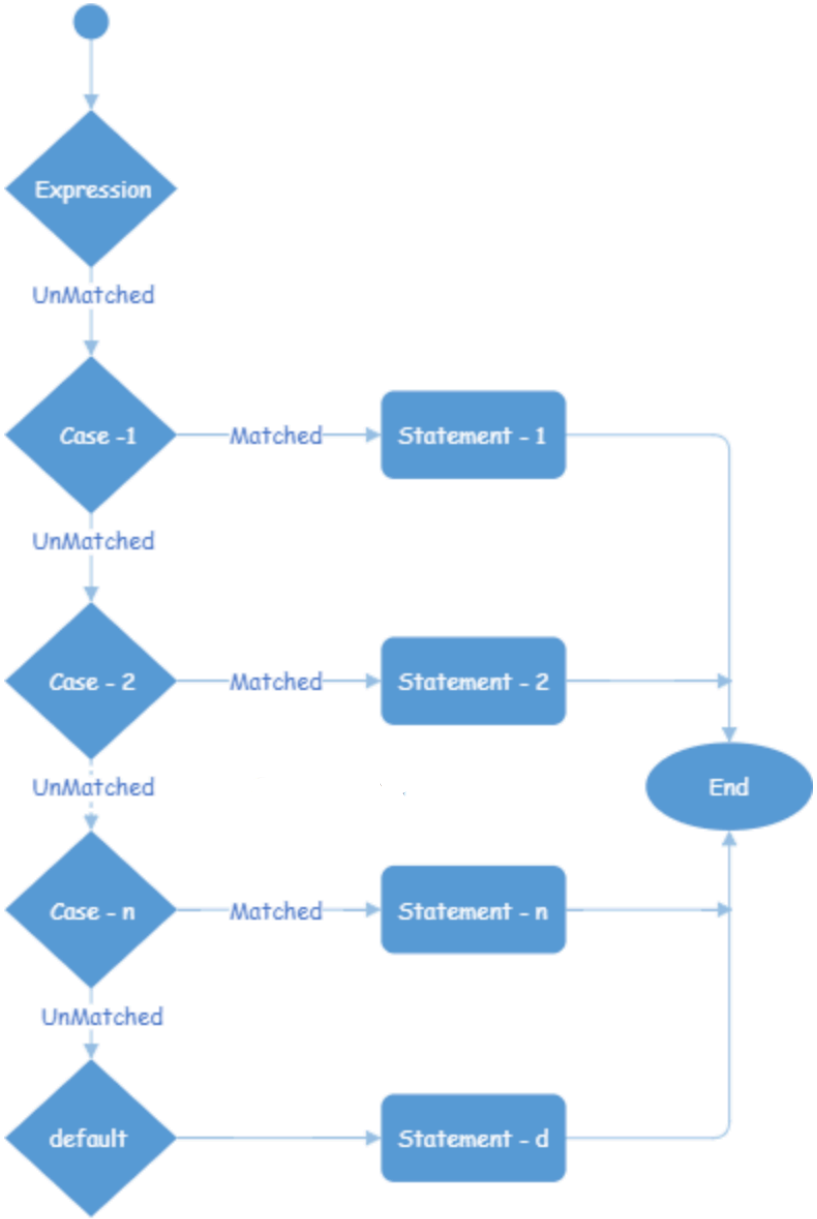
- Una sequenza che confronti ***un'unica variabile intera*** con diverse ***alternative costanti*** può essere realizzata con un enunciato **switch**

```
int x;
int y;
...
if (x == 1)
   y = 1;
else if (x == 2)
   y = 4;
else if (x == 4)
   y = 16;
else
   y = 0;
```

```
int x;
int y;
...
switch (x)
{  case 1: y = 1; break;
   case 2: y = 4; break;
   case 4: y = 16; break;
   default: y = 0; break;
}
```

La clausola **default è opzionale** e serve a determinare una porzione di codice che sarà comunque eseguita quando non viene verificata nessuna clausola

L'enunciato switch



L'enunciato switch

- **Vantaggio**: non bisogna ripetere il nome della variabile da confrontare
- **Svantaggio**: non si può usare se la variabile da confrontare **non è** int, byte, short, char (o le rispettive classi involucro), od oggetti String
- **Svantaggio**: non si può usare se uno dei valori da confrontare non è costante
- **Svantaggio**: ogni **case** deve terminare con un enunciato **break**, altrimenti viene eseguito anche il corpo del **case** successivo! Questo è fonte di molti errori...

L'enunciato switch

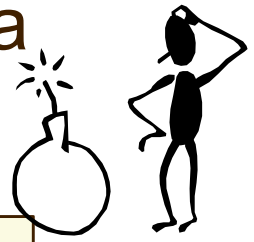
```
int day = 4;
switch (day) {
  case 1:
    System.out.println("Monday");
    break;
  case 2:
    System.out.println("Tuesday");
    break;
  case 3:
    System.out.println("Wednesday");
    break;
  case 4:
    System.out.println("Thursday");
    break;
  case 5:
    System.out.println("Friday");
    break;
  case 6:
    System.out.println("Saturday");
    break;
  case 7:
    System.out.println("Sunday");
    break;
}
// Outputs "Thursday" (day 4)
```

```
public class SwitchDemo {
    public static void main(String[] args) {

        int month = 8;
        String monthString;
        switch (month) {
            case 1:  monthString = "January";
                     break;
            case 2:  monthString = "February";
                     break;
            case 3:  monthString = "March";
                     break;
            case 4:  monthString = "April";
                     break;
            case 5:  monthString = "May";
                     break;
            case 6:  monthString = "June";
                     break;
            case 7:  monthString = "July";
                     break;
            case 8:  monthString = "August";
                     break;
            case 9:  monthString = "September";
                     break;
            case 10: monthString = "October";
                     break;
            case 11: monthString = "November";
                     break;
            case 12: monthString = "December";
                     break;
            default: monthString = "Invalid month";
                     break;
        }
        System.out.println(monthString);
    }
}
```

Consigli utili

Errori con operatori relazionali

- Alcune espressioni “naturali” con operatori relazionali sono errate, ma per fortuna il compilatore le rifiuta



```
if (0 <= x <= 1)          // NON FUNZIONA!
```

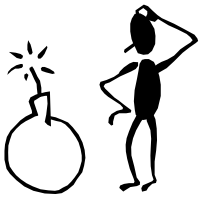
```
if (0 <= x && x <= 1) // OK
```

```
if (x && y > 0)           // NON FUNZIONA!
```

```
if (x > 0 && y > 0)       // OK
```

- Perché il compilatore le rifiuta?

Errori con operatori relazionali



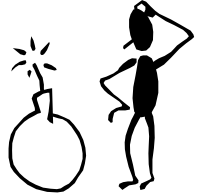
- Il compilatore analizza l'espressione logica e trova due operatori di confronto, quindi esegue il primo da sinistra e decide che il risultato sarà un valore booleano

```
if (0 <= x <= 1) x++; // NON FUNZIONA!
```

- Successivamente, si trova a dover applicare il secondo operatore relazionale a due operandi, il primo dei quali è di tipo boolean, mentre il secondo è di tipo int

```
operator <= cannot be applied to boolean,int
if (0 <= x <= 10) x++;
           ^
1 error
```

Errori con operatori relazionali



- Il compilatore analizza l'espressione logica e trova un operatore relazionale **>** (che ha la precedenza sull'operatore booleano **&&**), il cui risultato sarà un valore di tipo boolean

```
if (x && y > 0) x++; // NON FUNZIONA!
```

- Successivamente, si trova ad applicare l'operatore booleano **&&** a due operandi, il primo dei quali è di tipo **int**, mentre il secondo è di tipo **boolean**

```
operator && cannot be applied to int,boolean
if (x && y > 0) x++;
           ^
1 error
```

Rientri e Tabulazioni

- Decidere il numero ideale di caratteri bianchi da usare per ogni livello di rientro è molto arduo
- In questo corso consigliamo di usare tre/quattro caratteri

```
if (amount <= balance)
{  balance = balance - amount;
   if (amount > 20000000)
   {  System.out.println("Esagerato!");
   }
}
```

- Consigliamo anche di non usare i “caratteri di tabulazione”, che di solito generano un rientro di otto caratteri, eccessivo

Disposizione delle graffe

- Incolonnare le parentesi graffe

```
if (...)
{   ...;
    ...;
}
```

- Eventualmente lasciare su una riga da sola anche la graffa aperta

```
if (...)
{
    ...;
    ...;
}
```

- Evitare questa disposizione
  - è più difficile trovare la coppia!

```
if (...) {
    ...;
    ...;
}
```

Iterazioni (Capitolo 6)

Problema

- Riprendiamo un problema visto nella prima lezione, per il quale abbiamo **individuato un algoritmo** senza realizzarlo
  - Problema: Avendo depositato ventimila euro in un conto bancario che produce il 5% di interessi all'anno, capitalizzati annualmente, quanti anni occorrono affinché il saldo del conto arrivi al doppio della cifra iniziale?
  - Abbiamo bisogno di un programma che **ripeta** degli enunciati (capitalizzazione degli interessi annuali, incremento del conto degli anni), **finché** non si realizza la condizione desiderata

Algoritmo che risolve il problema

1. All'anno 0 il saldo è 20000
2. **Ripetere** i passi 3 e 4 **finché** il saldo è minore del doppio di 20000, poi passare al punto 5
3. Aggiungere 1 al valore dell'anno corrente
4. Il nuovo saldo è il valore del saldo precedente moltiplicato per 1.05 (cioè aggiungiamo il 5%)
5. Il risultato è il valore dell'anno corrente

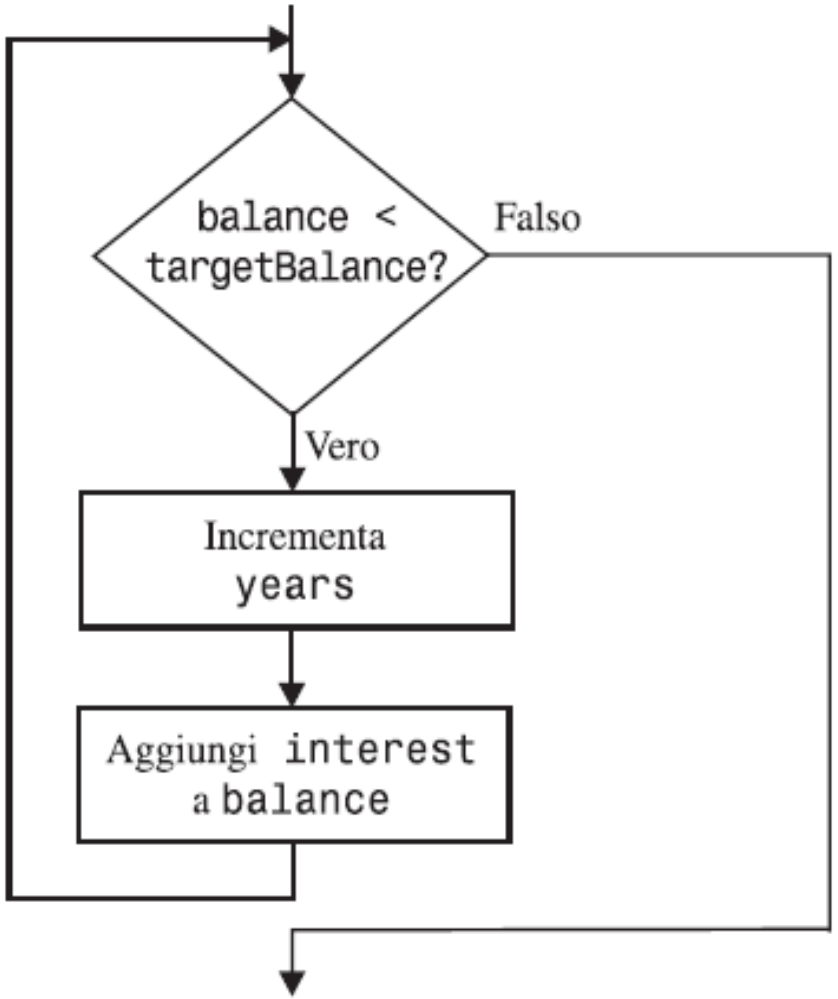
- L'enunciato **while** consente la realizzazione di programmi che devono eseguire ripetutamente una serie di azioni finché è verificata una condizione

Il ciclo while

- Sintassi:
- Scopo:
  - eseguire un enunciato finché la condizione è vera

```
while (condizione)
    enunciato
```

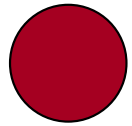
- Il **corpo** del ciclo **while** può essere un enunciato qualsiasi, quindi anche un blocco di enunciati
- L'enunciato **while** realizza un **ciclo**
  - per capire cosa significa questa espressione è utile osservare la rappresentazione del codice mediante **diagrammi di flusso**



Soluzione 1 (usando BankAccount)

```
public class SimpleInvestmentTester
{
    public static void main(String[] args)
    {
        final double INITIAL_BALANCE = 20000;
        final double RATE = 5;
        // definizione e inizializzazione
        BankAccount acct = new BankAccount(INITIAL_BALANCE);
        int year = 0;  // anno

        while (acct.getBalance() < 2 * INITIAL_BALANCE)
        {   year++;  // incremento dell'anno
            double interest = acct.getBalance() * RATE/100;
            acct.deposit(interest); // modifica saldo
        }

        System.out.println("L'investimento " +
                           "raddoppia in " +year+ " anni");
    }
}
```

Soluzione 2: classe Investment

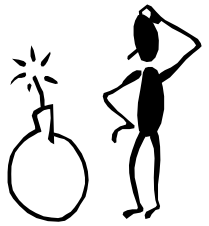


```
public class Investment
{
    public Investment(double aBalance, double aRate)
    {  balance = aBalance;
       rate = aRate;
       years = 0;
    }
    public void waitForBalance(double targetBalance)
    {  while (balance < targetBalance)
       {  years++;
          double interest = balance * rate / 100;
          balance = balance + interest;
       }
    }
    public double getBalance()
    {  return balance; }

    public int getYears()
    {  return years; }

    private double balance;
    private double rate;
    private int years;
}
```

Soluzione 2: classe InvestmentTester

```
public class InvestmentTester
{
   public static void main(String[] args)
   {
      final double INITIAL_BALANCE = 10000;
      final double RATE = 5;
      Investment invest
            = new Investment(INITIAL_BALANCE, RATE);
      invest.waitForBalance(2 * INITIAL_BALANCE);
      int years = invest.getYears();
      System.out.println("The investment doubled after "
            + years + " years");
   }
}
```

Cicli infiniti



- Esistono errori logici che impediscono la terminazione di un ciclo, generando un **ciclo infinito**
  - l'esecuzione del programma continua **ininterrottamente**
- Bisogna **arrestare** il programma con un comando del sistema operativo, o addirittura **riavviare** il computer

```
int year = 0;
while (year < 20)
{  double interest = balance * rate / 100;
   balance = balance + interest;
   // qui manca year++
}
```

Ctrl + C

```
int year = 20;
while (year > 0)
{  year++;  // doveva essere year--
   double interest = balance * rate / 100;
   balance = balance + interest;
}
```

È tutto chiaro? …

- Quante volte viene eseguito il seguente ciclo?

```
while (false)
  // enunciato
```

- Cosa succede nel programma **SimpleInvestmentTester** se **RATE** nel metodo main vale 0?

Cicli for

Ciclo for

- Molti cicli hanno questa forma

```
i = inizio;
while (i < fine)
{   enunciati
    i++;
}
```

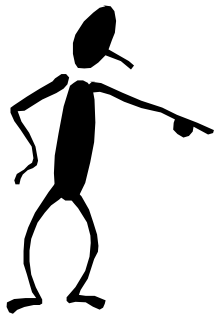
- Per comodità esiste il **ciclo `for` equivalente**

```
for (i = inizio; i < fine; i++)
{   enunciati
}
```

- Non è necessario che l'incremento sia di una sola unità, né che sia positivo, né che sia intero

```
for (double x = 2; x > 0; x = x - 0.3)
{   enunciati
}
```

L'enunciato `for`



- Sintassi:

```
for (inizializzazione; condizione; aggiornamento)
    enunciato
```

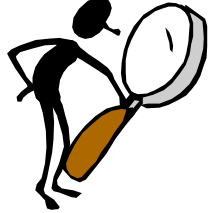
- Scopo: eseguire un'**inizializzazione**, poi **ripetere** l'esecuzione di un enunciato ed effettuare un **aggiornamento** finché la **condizione** è vera
- Nota: l'inizializzazione può contenere la definizione di una variabile, che **sarà visibile soltanto all'interno del corpo del ciclo**

```
for (int y = 1; y <= 10; y++)
{   …
}
// qui y non è più definita
```

Esempio: invertire una stringa

```
import java.util.Scanner;

public class ReverseTester
{  public static void main(String[] args)
   {  Scanner in = new Scanner(System.in);
      System.out.println("Inserire una stringa:");
      String s = in.nextLine();
      String r = "";
      for (int i = 0; i < s.length(); i++)
      {  char ch = s.charAt(i);
         // aggiungi all'inizio
         r = ch + r;
      }
      System.out.println(s + " invertita è " + r);
   }
}
```

Visibilità delle variabili



- Se il valore finale di una variabile di controllo del ciclo deve essere visibile al di fuori del corpo del ciclo, bisogna definirla **prima** del ciclo
- Poiché una variabile definita nell'inizializzazione di un ciclo non è più definita dopo il corpo del ciclo, è possibile (e comodo) usare di nuovo **lo stesso nome** in altri cicli

```
double b = ...;
for (int i = 1; i < 10 && b < c; i++)
{   ...
    modifica b
}
// qui b è visibile, mentre i non lo è
for (int i = 3; i > 0; i--)
    System.out.println(i);
```

È tutto chiaro? …

1. Quante volte viene eseguito il seguente ciclo for?

```
for (i = 0; i <= 10; i++)
  System.out.println(i * i);
```

Cicli annidati

Problema

- Vogliamo stampare una tabella con i valori delle potenze x^y, per ogni valore di **x** tra 1 e 4 e per ogni valore di **y** tra 1 e 5

```
1     1     1     1     1
2     4     8    16    32
3     9    27    81   243
4    16    64   256  1024
```

- Innanzitutto, bisogna stampare **4 righe**

```
for (int x = 1; x <= 4; x++)
{   // stampa la riga x-esima della tabella
    ...
}
```

Problema

- Per stampare la riga x-esima, bisogna calcolare e stampare i valori x^1, x^2, x^3, x^4 e x^5, cosa che si può fare facilmente con un ciclo **for**

```
// stampa la riga x-esima della tabella
for (int y = 1; y <= 5; y++)
{   int p = (int)Math.round(Math.pow(x, y));
    System.out.print(p + " ");
}
System.out.println(); // va a capo
```

- Ogni iterazione del ciclo stampa un valore, seguito da uno spazio bianco per separare valori successivi; al termine si va a capo

Soluzione

- Mettendo insieme le due “soluzioni parziali” si risolve il problema, mediante due cicli “**annidati**” (nested) , cioè “uno dentro l’altro”

```
for (int x = 1; x <= 4; x++)
{  // stampa la riga x-esima della tabella
   for (int y = 1; y <= 5; y++)
   {  // stampa il valore y-esimo
      // della riga x-esima
      int p = (int)Math.round(Math.pow(x, y));
      System.out.print(p + " ");
   }
   System.out.println(); // va a capo
}
```

```
1 1 1 1 1
2 4 8 16 32
3 9 27 81 243
4 16 64 256 1024
```

Le colonne non sono allineate!

Soluzione

- Incolonnare dati di lunghezza variabile è un problema frequente

Ora ci sono tre cicli annidati!

```
final int COLUMN_WIDTH = 5;
for (int x = 1; x <= 4; x++)
{   for (int y = 1; y <= 5; y++)
    {   // converte in stringa il valore
        String p = "" + (int)Math.round(Math.pow(x, y));
        // aggiunge gli spazi necessari
        while (p.length() < COLUMN_WIDTH)
            p = " " + p;
        System.out.print(p);
    }
    System.out.println();
}
```

```
    1    1    1    1    1
    2    4    8   16   32
    3    9   27   81  243
    4   16   64  256 1024
```

Il problema del “ciclo e mezzo”

Il problema del "ciclo e mezzo"

- Un ciclo del tipo
  - "**fai** qualcosa, **verifica** una condizione, **fai** qualcos'altro e **ripeti** il ciclo se la condizione era vera"
- non ha una struttura di controllo predefinita in Java e deve essere realizzata con un "trucco", come quello di usare una variabile booleana, detta **variabile di controllo** del ciclo
- Una struttura di questo tipo si chiama anche "**ciclo e mezzo**" o ciclo ridondante (perché c'è qualcosa di "aggiunto", di innaturale…)

Il problema del "ciclo e mezzo"

- Situazione tipica: l'utente deve inserire un **insieme di valori**, la cui **dimensione** non è predefinita
- Si realizza un ciclo **while**, dal quale si esce soltanto quando si verifica la condizione all'interno del ciclo

La lettera "**Q**" (Quit) è un **valore sentinella** che segnala che l'immissione dei dati è terminata

```
boolean done = false;
while (!done)
{ System.out.println("Valore?");
  String input = in.next();
  if (input.equalsIgnoreCase("Q"))
      done = true;
   else
      ... // elabora line
}
```

Ciclo e mezzo: il programma QEater

```
import java.util.Scanner;

public class QEater
{
    public static void main(String[] args)
    {
        Scanner in = new Scanner(System.in);
        boolean done = false;
        while(!done)
        {
            System.out.println("Voglio una Q");
            String input = in.nextLine();
            if (input.equals("Q"))
            {
                System.out.println("Grazie!");
                done = true;
            }
            else
            {    System.out.println("Allora non hai capito...");
            }
        }
    }
}
```

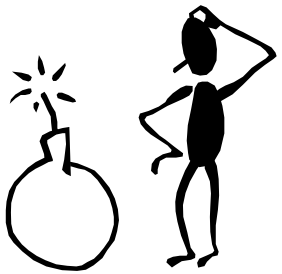
Break, continue, e codice spaghetti

```
while (true)
{ String input = in.next();
  if (input.equalsIgnoreCase("Q"))
    break;
  else
    ... // elabora line
}
```

- Soluzione alternativa alla struttura "ciclo e mezzo":
  - usare un ciclo infinito **while(true)** e l'enunciato **break**
  - L'enunciato **break** provoca la terminazione del ciclo
  - Esiste anche l'enunciato **continue**, che fa proseguire l'esecuzione dalla fine dell'iterazione attuale del ciclo
- L'uso di **break** e **continue** è non necessario e sconsigliabile
  - perchè contribuisce a creare **codice spaghetti**
  - ovvero rappresentato da diagrammi di flusso pieni di linee, difficili da leggere e comprendere

Variabili non inizializzate

Variabili non inizializzate

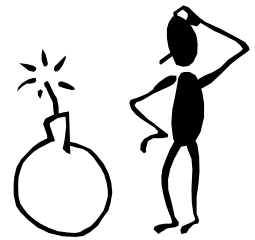


- È buona regola fornire sempre un **valore di inizializzazione** nella definizione di variabili

```
int lit;
```

- Cosa succede altrimenti?
  - la definizione di una variabile “crea” la variabile, cioè le riserva uno spazio nella memoria primaria (la quantità di spazio dipende dal tipo della variabile)
  - tale spazio di memoria non è “vuoto”, una condizione che non si può verificare in un circuito elettronico, ma contiene un valore “casuale” (in realtà contiene l’ultimo valore attribuito a quello spazio da un precedente programma… valore che a noi non è noto)

Variabili non inizializzate

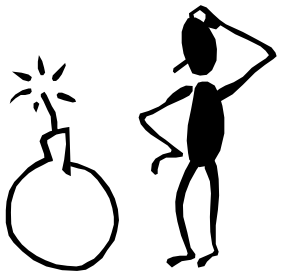


- Se si usasse il valore di una variabile prima di averle assegnato un qualsiasi valore, il programma si troverebbe a elaborare quel valore che "casualmente" si trova nello spazio di memoria riservato alla variabile

ERRORE

```
public class Coins6 // NON FUNZIONA!
{  public static void main(String[] args)
   {  int lit;
      double euro = 2.35;
      double totalEuro = euro + lit / 1936.27;
      System.out.print("Valore totale in euro ");
      System.out.println(totalEuro);
   }
}
```

Variabili non inizializzate

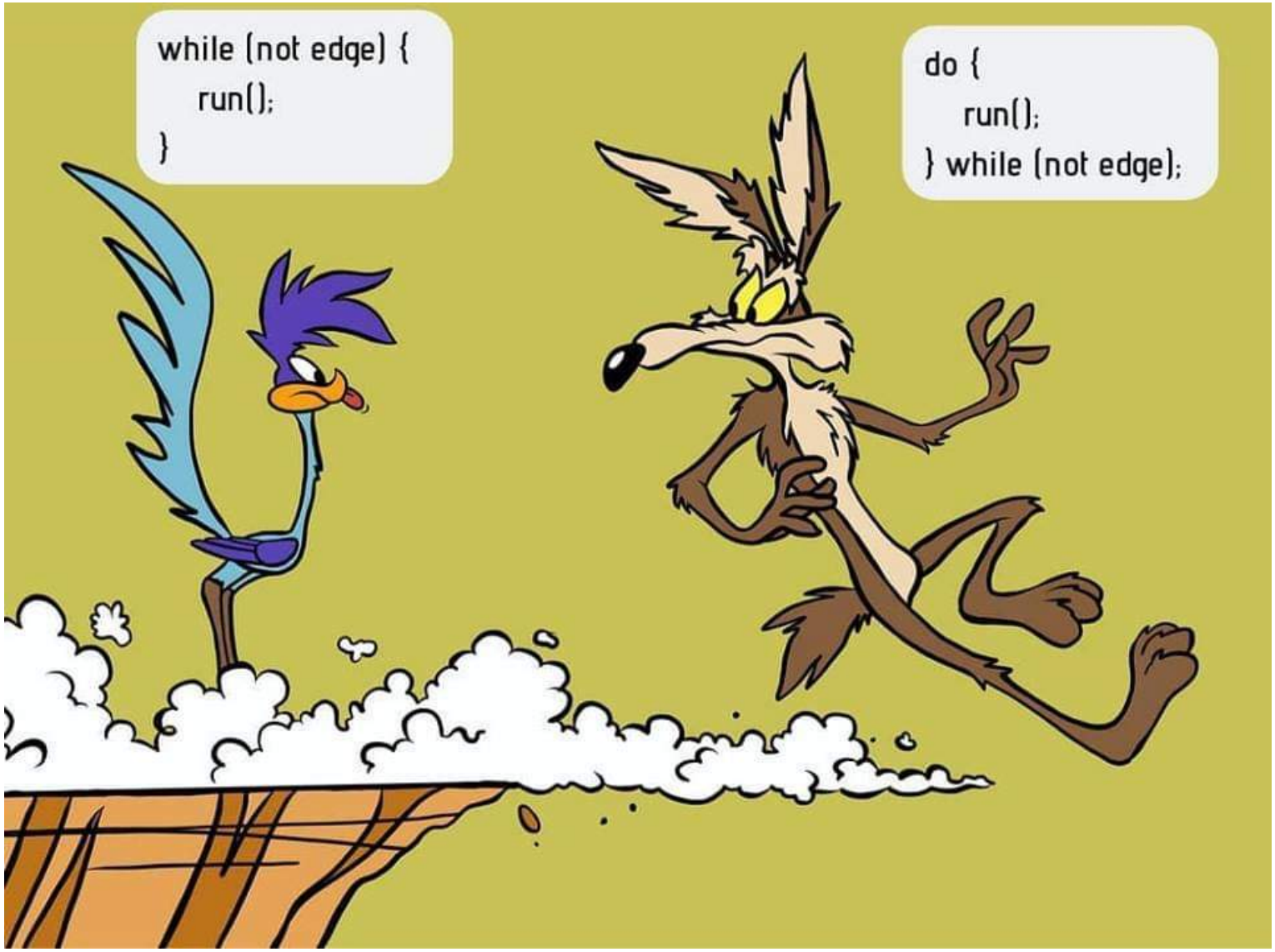


- Questo problema provoca insidiosi errori di esecuzione in molti linguaggi di programmazione
  - il compilatore Java, invece, segnala come errore l'utilizzo di variabili a cui non sia mai stato assegnato un valore (mentre non è un errore la sola definizione...)

Coins6.java:5: variable lit might not have been initialized

  - questi errori non sono sintattici, bensì logici, ma vengono comunque individuati dal compilatore, perché si tratta di errori semantici (cioè di comportamento del programma) individuabili in modo automatico

Cicli do



Ciclo do

- Capita di dover eseguire il corpo di un ciclo almeno una volta, per poi ripeterne l'esecuzione se è verificata una particolare condizione
  - Esempio: leggere un valore in ingresso, eventualmente rileggerlo finché non viene introdotto un valore "valido"
- Si può usare un ciclo while "innaturale"

```
// si usa un'inizializzazione "ingiustificata"
double rate = 0;
while (rate <= 0)
{   System.out.println("Inserire il tasso:");
    rate = console.readDouble();
}
```

- ma per comodità esiste il **ciclo do**

```
double rate;
do
{   System.out.println("Inserire il tasso:");
    rate = console.readDouble();
} while (rate <= 0);
```

Cicli: errori e consigli

Errori per scarto di uno

```
int years = 0;   // o forse int years =1; ???
while (balance < 2 * initialBalance)
{       // o forse balance <= 2 * initialBalance ???
   years++;
   double interest = balance * rate / 100;
   balance = balance + interest;
}
System.out.println("Investimento raddoppiato dopo"
                        + years + " anni.");
```

- Come evitare questi errori per scarto di uno?
  - Provare con alcuni semplici casi di prova
  - Investimento iniziale 100 euro, tasso 50%
    - Allora years deve essere inizializzato a 0
  - Tasso di interesse 100%
    - Allora la condizione deve essere < e non <=

Definire bene i limiti

- E se n fosse **negativo**?
  - La condizione **i!=n** sarebbe **sempre vera**
  - Molto meglio **i<=n**

```
for (int i = 1; i != n; i++)
```

```
for (i = a; i < b; i+=c)
// oppure
for (i = a; i <= b; i+=c)
```

- Imparare a **contare** le iterazioni
  - Il primo ciclo (**asimmetrico**) viene eseguito **(b-a)/c** volte
  - Il secondo (**simmetrico**) viene eseguito **(b-a+1)/c** volte

- Due intestazioni
  - Sono **equivalenti**
  - Nella prima i limiti sono simmetrici
  - Nella seconda sono asimmetrici ma il codice è più leggibile

```
for (i = 0; i <= s.length() - 1; i++)
// è corretta, ma è meglio questa:
for (i = 0; i < s.length(); i++)
```

Punti e virgola mancanti o di troppo

```
sum = 0;
int i;
for (i = 1; i <= 10; i++);
   sum = sum + i;
System.out.println(sum);
```

- C'è un "**;**" di troppo!
  - il corpo del ciclo è **vuoto**
  - L'istruzione di aggiornamento di **sum** viene eseguita una sola volta, **dopo l'uscita** dal ciclo

```
for (years = 1; (balance = balance + balance * rate / 100)
                < targetBalance; years++)  // ;
System.out.println(years);
```

- In questo caso tutte le operazioni necessarie sono **nell'intestazione del ciclo**
  - Il corpo del ciclo ***deve*** essere vuoto
  - Ma abbiamo dimenticato il ";" dopo l'intestazione
  - Quindi l'istruzione di stampa viene considerata parte del corpo

Cicli for di "cattivo gusto"

```
for (rate = 5; years-- > 0;
System.out.println(balance))
{   enunciati
}
```

- È un ciclo for sintatticamente corretto
  - ma contiene **condizioni estranee**, che producono risultati difficilmente comprensibili

```
for (int i = 1; i <= years; i++)
{   if (balance >= targetBalance)
       i = i + 1;
    else
       ...
}
```

- L'intestazione è corretta
  - ma il contatore viene modificato nel corpo del ciclo
- Sono cicli for **di cattivo gusto!**

Department of Information Engineering

Progettazione di classi (capitolo 8) (capitolo 3 – sez. 3.6 e 3.7)

Parametri espliciti/impliciti
Il parametro this

I parametri dei metodi

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

- Cosa succede quando invochiamo il metodo?

```
account.deposit(500);
```

- L'esecuzione del metodo dipende da due valori
  - il **riferimento** all'oggetto account
  - il **valore** 500
- Quando viene eseguito il metodo, il suo **parametro esplicito** amount assume il valore 500
  - esplicito perché compare nella firma del metodo

I parametri dei metodi

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

- Abbiamo già detto che un metodo può avere parametri espliciti e/o impliciti
  - **amount** è il **parametro esplicito** del metodo
  - **balance** si riferisce alla variabile di esemplare balance della classe **BankAccount**, ma sappiamo che di tale variabile esiste una copia per ogni oggetto

- Alla variabile **balance** di quale oggetto si riferisce il metodo?
  - si riferisce alla variabile che appartiene all'oggetto con cui viene invocato il metodo, ma come fa?

Il parametro implicito dei metodi

```
account.deposit(500);
```

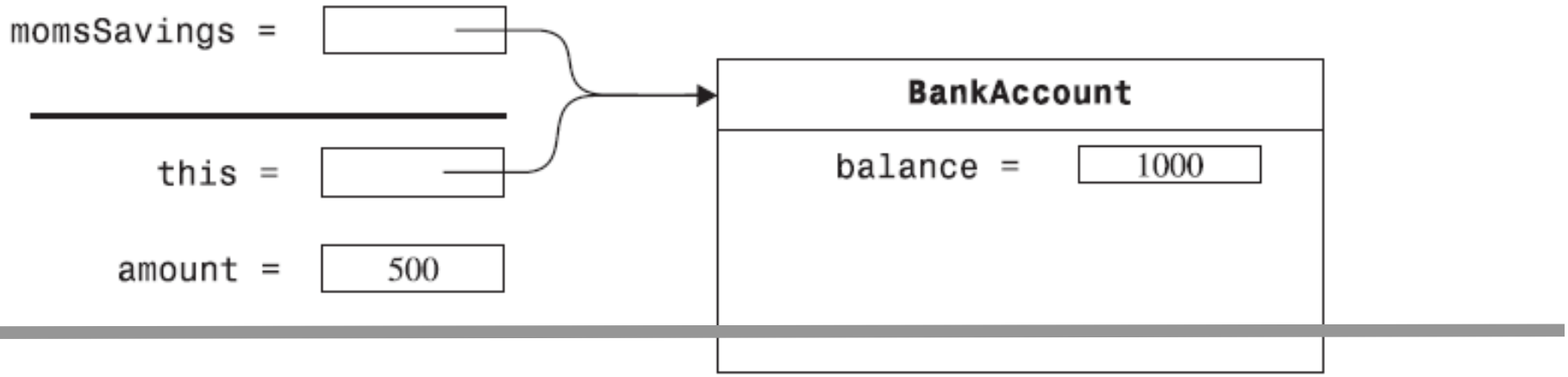
- All'interno di ogni metodo il riferimento all'oggetto con cui viene eseguito il metodo si chiama **parametro implicito** e si indica con la parola chiave **this**
  - in questo caso, **this** assume il valore di **account** all'interno del metodo **deposit**
- Ogni metodo ha sempre uno e un solo parametro implicito, dello stesso tipo della classe a cui appartiene il metodo
  - eccezione: i metodi statici **non** hanno parametro implicito
- Il parametro implicito **non deve essere dichiarato** e si chiama sempre **this**

Uso del parametro implicito

- La vera sintassi del metodo dovrebbe essere

```
public void deposit(double amount)
{   this.balance = this.balance + amount;
}
// this è di tipo BankAccount
```

- Ma Java consente una comoda scorciatoia:
  - se un metodo si riferisce a un campo di esemplare, il compilatore costruisce automaticamente un riferimento al campo di esemplare dell'oggetto rappresentato dal parametro implicito **this**



È tutto chiaro? …

1. Quanti e quali sono i parametri impliciti ed espliciti del metodo **withdraw** di **BankAccount**? Quali sono i loro nomi e tipi?
2. Qual è il significato di **this.amount** nel metodo **deposit**? Oppure, se l'espressione è priva di significato, per quale motivo lo è?

Membri di classe “statici”

Classi "di utilità" e metodi statici

- Esistono classi che non servono a creare oggetti ma contengono **metodi statici** e **costanti**.
  - Queste si chiamano solitamente **classi di utilità**
  - La classe **Math** è un esempio di questo tipo di classi
  - La classe **Numeric** scritta da noi è un altro esempio
- **Esempio**: una classe **Financial**

```
public class Financial
{
   public static double percentOf(double p, double a)
   { return (p / 100) * a; }
   // qui si possono aggiungere altri metodi finanziari
}
```

- Non si devono **creare oggetti** di tipo **Financial** per usare i metodi della classe:

```
double tax = Financial.percentOf(taxRate, total);
```

Variabili statiche

- Vogliamo modificare **BankAccount** in modo che
  - il suo stato contenga anche un ***numero di conto***

```
public class BankAccount
{   ...
    private int accountNumber;
}
```

  - il numero di conto sia assegnato dal costruttore
    - ogni conto deve avere un numero diverso
    - i numeri assegnati devono essere progressivi, iniziando da 1

Soluzione

- **Prima idea** (che non funziona…)
  - variabile per memorizzare l'ultimo n. di conto assegnato

```
public class BankAccount
{   ...
    private int accountNumber;
    private int lastAssignedNumber = 0;
    ...
    public BankAccount()
    {   lastAssignedNumber++;
        accountNumber = lastAssignedNumber;
    }
}
```

- Questo costruttore non funziona perché la variabile **lastAssignedNumber** è una ***variabile di esemplare***
  - ne esiste una copia per ogni oggetto
  - risultato: tutti i conti hanno il numero di conto **1**

Variabili statiche

- Ci serve una ***variabile condivisa da tutti gli oggetti della classe***
  - una variabile con questa semantica si ottiene con la dichiarazione **static**

```
public class BankAccount
{   ...
    private static int lastAssignedNumber;
}
```

- Una variabile **static** (***variabile di classe***) è condivisa da tutti gli oggetti della classe
- Ne esiste **un'unica copia** indipendentemente da quanti oggetti siano stati creati (**anche zero**)

Variabili statiche

- Ora il costruttore funziona

```
public class BankAccount
{   ...
    private int accountNumber;
    private static int lastAssignedNumber = 0;
    ...
    public BankAccount()
    {   lastAssignedNumber++;
        accountNumber = lastAssignedNumber;
    }
}
```

- Ogni metodo (o costruttore) di una classe può **accedere** alle variabili statiche della classe e **modificarle**

Variabili statiche

- Le variabili statiche **non** possono (da un punto di vista logico) essere inizializzate nei costruttori
  - Il loro valore verrebbe inizializzato di nuovo **ogni volta che si costruisce un oggetto**, perdendo il vantaggio di avere una variabile condivisa!
- Bisogna inizializzarle quando si dichiarano

```
private static int lastAssignedNumber = 0;
```

- Questa sintassi si può usare anche per le variabili di esemplare, anziché usare un costruttore
  - Ma **non** è una buona pratica di programmazione

Variabili statiche

- Nella programmazione a oggetti, l'utilizzo di variabili statiche deve **essere limitato**
  - Il comportamento di metodi che usano variabili statiche **non** dipende solo dai loro parametri (implicito ed espliciti)
  - In ogni caso, le variabili statiche devono essere **private**, per evitare accessi indesiderati
- È invece pratica comune (senza controindicazioni) usare **costanti statiche**, come nella classe Math

```
public class Math
{   ...
    public static final double PI =
                3.14159265358979323846;
}
```

  - Queste sono di norma **public** e accessibili **qualificandone** il nome. Ad esempio: **Math.PI**

È tutto chiaro? …

1. Citare due variabili statiche della classe **System**

Categorie di variabili (sez. 3.7)
Ciclo di vita di una variabile

Ciclo di vita di una variabile

- Sappiamo che in Java esistono quattro diversi tipi di variabili
  - variabili ***locali*** (all'interno di un metodo)
  - variabili ***parametro*** (dette ***parametri formali***)
  - variabili ***di esemplare*** o di istanza
  - variabili ***statiche*** o di classe
- Hanno in comune il fatto di contenere valori appartenenti a un tipo ben preciso.
- Differiscono per quanto riguarda il loro ***ciclo di vita***
  - cioè l'intervallo di **tempo** in cui, dopo essere state create, continuano ad **occupare lo spazio in memoria** a loro riservato

Ciclo di vita di una variabile

- Una ***variabile locale***
  - ***viene creata*** quando viene eseguito l'enunciato in cui è definita
  - ***viene eliminata*** quando l'esecuzione del programma esce dal **blocco di enunciati** in cui la variabile è definita
    - se non è definita all'interno di un blocco di enunciati, viene eliminata quando l'esecuzione del programma esce dal metodo in cui la variabile viene definita
- Una ***variabile parametro (formale)***
  - ***viene creata*** quando viene invocato il metodo
  - ***viene eliminata*** quando l'esecuzione del metodo termina

Ciclo di vita di una variabile

- Una ***variabile statica***
  - ***viene creata*** quando la macchina virtuale Java carica la classe per la prima volta
  - ***viene eliminata*** quando la classe viene scaricata dalla macchina virtuale Java
  - ai fini pratici, possiamo dire che ***esiste sempre***...
- Una ***variabile di esemplare***
  - ***viene creata*** quando viene creato l'oggetto a cui appartiene
  - ***viene eliminata*** quando l'oggetto viene eliminato

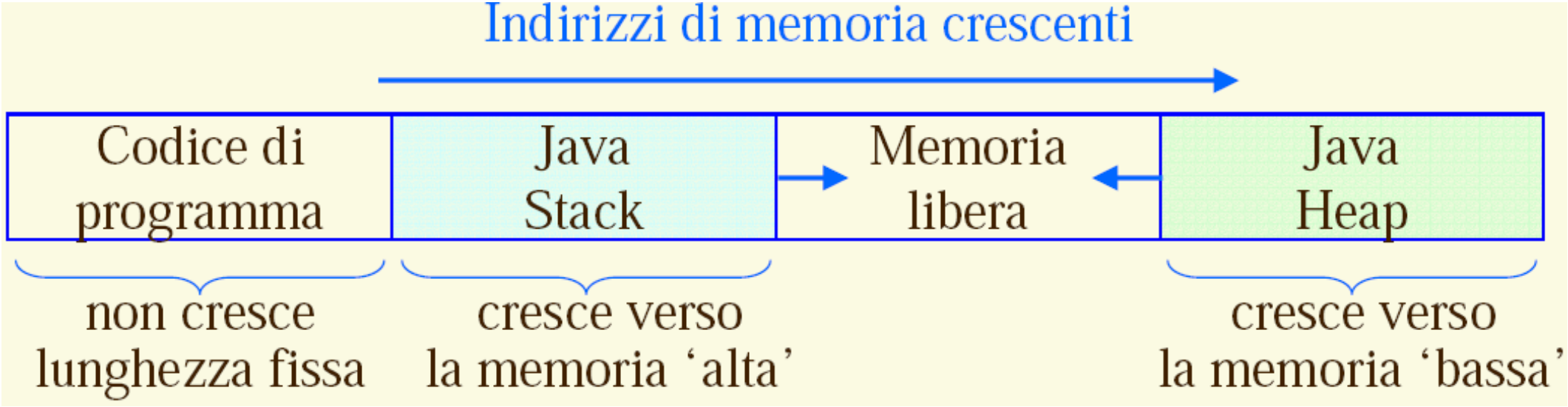
Eliminazione di oggetti

- Un oggetto è a tutti gli effetti "**inutilizzabile**" quando non esiste più **nessun riferimento** ad esso
- Se un programma abbandona molti oggetti in memoria, può **esaurire la memoria** a disposizione
  - La **JVM** effettua automaticamente la gestione della memoria durante l'esecuzione di un programma
  - Usa un meccanismo di **garbage collection**
    - Viene periodicamente riciclata, cioè resa di nuovo libera, la memoria eventualmente occupata da oggetti che non abbiano più un riferimento nel programma

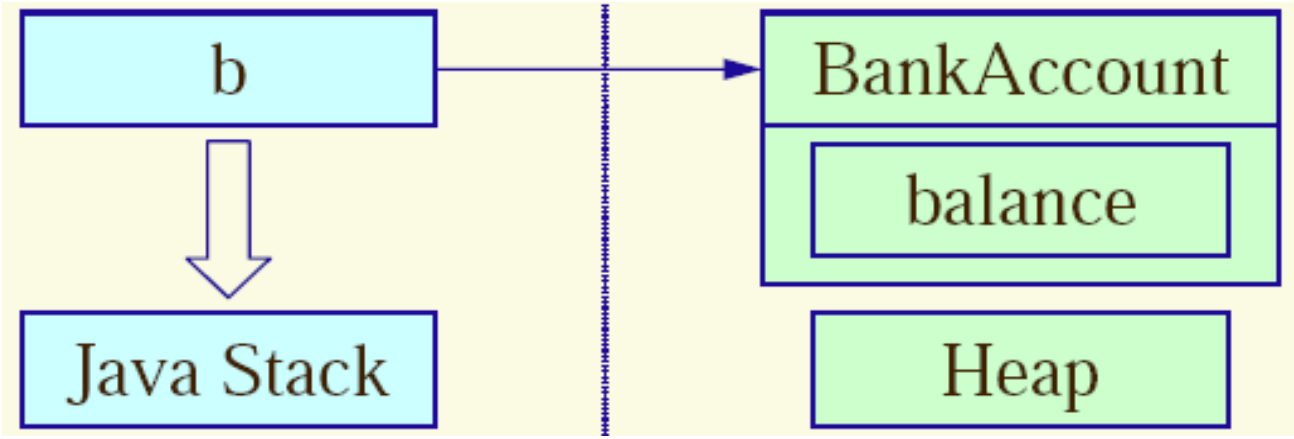
Allocazione della memoria in java

- Al momento dell'**esecuzione**, a ciascun programma java viene assegnata un'**area di memoria**
  - Una parte della memoria serve per **memorizzare il codice**; quest'area è **statica**, ovvero non modifica le sue dimensioni durante l'esecuzione del programma
  - La **Java Stack** è un'area **dinamica** (ovvero che cambia dimensione durante l'esecuzione) in cui vengono memorizzate i **parametri e le variabili locali** dei metodi
    - **Stack** significa **pila**
  - La **Java Heap** è un'altra area **dinamica** in cui vengono **creati oggetti** durante l'esecuzione dei metodi di un programma, usando lo speciale operatore new
    - **Heap** significa **cumulo**, **mucchio**

Modello della memoria in java



- L'istruzione `BankAccount b = new BankAccount();`
  - Produce questo effetto in memoria:



Ambiti di visibilità di variabili

Visibilità di variabili locali

- Per evitare conflitti, dobbiamo conoscere **l'ambito di visibilità** di ogni tipo di variabile
  - Ovvero la **porzione** del programma all'interno della quale si può accedere ad essa
- **Esempio**: due variabili locali con lo stesso nome
  - Funziona perché gli ambiti di visibilità sono **disgiunti**

```
public class RectangleTester
{
   public static double area(Rectangle rect)
   {  double r = rect.getWidth() * rect.getHeight();
      return r; }
   public static void main(String[] args)
   {  Rectangle r = new Rectangle(5, 10, 20, 30);
      double a = area(r);
      System.out.println(r); }
}
```

Visibilità di variabili locali

- Anche qui gli ambiti di visibilità sono **disgiunti**:

```
if (x >= 0)
{  double r = Math.sqrt(x);
   . . . } // la visibilità di r termina qui
else
{  Rectangle r = new Rectangle(5, 10, 20, 30);
   // OK, questa è un'altra variabile r
   . . . }
```

- Invece l'ambito di visibilità di una variabile **non** può contenere la definizione di un'altra variabile locale con lo stesso nome:

```
Rectangle r = new Rectangle(5, 10, 20, 30);
if (x >= 0)
{  double r = Math.sqrt(x);
   // Errore: non si può dichiarare un'altra var. r qui
   . . . }
```

Visibilità di membri di classe

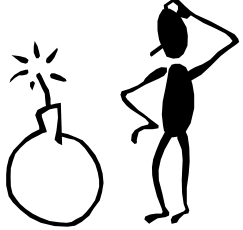
- Membri **private** hanno visibilità di classe
  - Qualsiasi metodo di una classe può accedere a variabili e metodi della stessa classe
- Membri **public** hanno visibilità al di fuori della classe
  - A patto di renderne **qualificato** il nome, ovvero:
  - Specificare il nome della classe per membri **static**
    - **Math.PI, Math.sqrt(x)**
  - Specificare l'oggetto per membri **non static**
    - **account.getBalance()**
- Non è necessario qualificare i membri appartenenti a una stessa classe
  - Perché ci si riferisce **automaticamente** al parametro implicito **this**

Visibilità di membri di classe

- Esempio: qualifica sottintesa dal parametro implicito

```
public class BankAccount
{
   public void transfer(double amount, BankAccount other)
   {  withdraw(amount); // cioè this.withdraw(amount);
      other.deposit(amount);
   }
public void withdraw(double amount)
   {  if (balance > amount) // cioè this.balance
     balance = balance - OVERDRAFT_FEE;
     //cioè BankAccount.OVERDRAFT_FEE
       else ...
   }
   ...
private static double OVERDRAFT_FEE = 5;
   ...
}
```

Visibilità sovrapposte

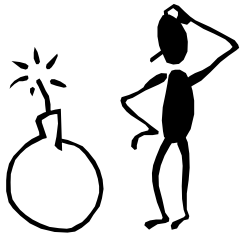


- Purtroppo gli ambiti di visibilità di una variabile locale e di una variabile di esemplare possono **sovrapporsi**

```
public class Coin
{
   ...
   public double getExchangeValue(double exchangeRate)
   {
      double value; // Variabile locale
      ...
      return value;
   }
   private String name;
   private double value; // Campo di esemplare omonimo
}
```

- Viene segnalato un errore in compilazione?

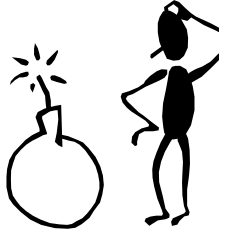
Visibilità sovrapposte



- **Non** viene segnalato alcun errore in compilazione
  - Java specifica che in casi come questo **prevale** il nome della variabile locale
  - La variabile di esemplare viene **"messa in ombra"** (**shadowed**)
- Questa scelta è giustificata dal fatto che la variabile di esemplare può sempre essere qualificata usando il parametro **this**

- Un analogo effetto di **shadowing** è prodotto da una variabile locale su una variabile statica omonima
  - La variabile statica deve essere qualificata con il nome della classe

Visibilità sovrapposte



- **Errore molto comune**:
  - utilizzare accidentalmente lo **stesso nome** per una variabile locale e un campo di esemplare

```
public class Coin
{
   ...
   public Coin(double aValue, String aName)
   {
      value = aValue;
      String name = aName; //ahi: abbiamo dichiarato una
                             nuova variabile name
   }
   ...
   private String name;
   private double value;
}
```

È tutto chiaro? …

1. Qual è l'ambito di visibilità delle variabili **amount** e **newBalance** del metodo **deposit** di **BankAccount**?
2. Qual è l'ambito di visibilità del campo di esemplare **balance** di **BankAccount**?