Decisioni

Obiettivi

- Imparare ad alterare il flusso di esecuzione delle istruzioni del programma in base alla verifica di determinate condizioni
- Costrutto sintattico if
  - Clausola else
- Confronto tra numeri
- Confronto tra oggetti
- Condizioni complesse

L'enunciato *if*

- Una delle caratteristiche fondamentali di un programma è la capacità di prendere decisioni

- Per prendere decisioni all'interno di un programma si usa ***l'enunciato if***

L'enunciato if

- Il metodo withdraw di BankAccount consente di prelevare tutto il denaro che si vuole
  - il saldo balance può diventare **negativo**

```
balance = balance - amount;
```

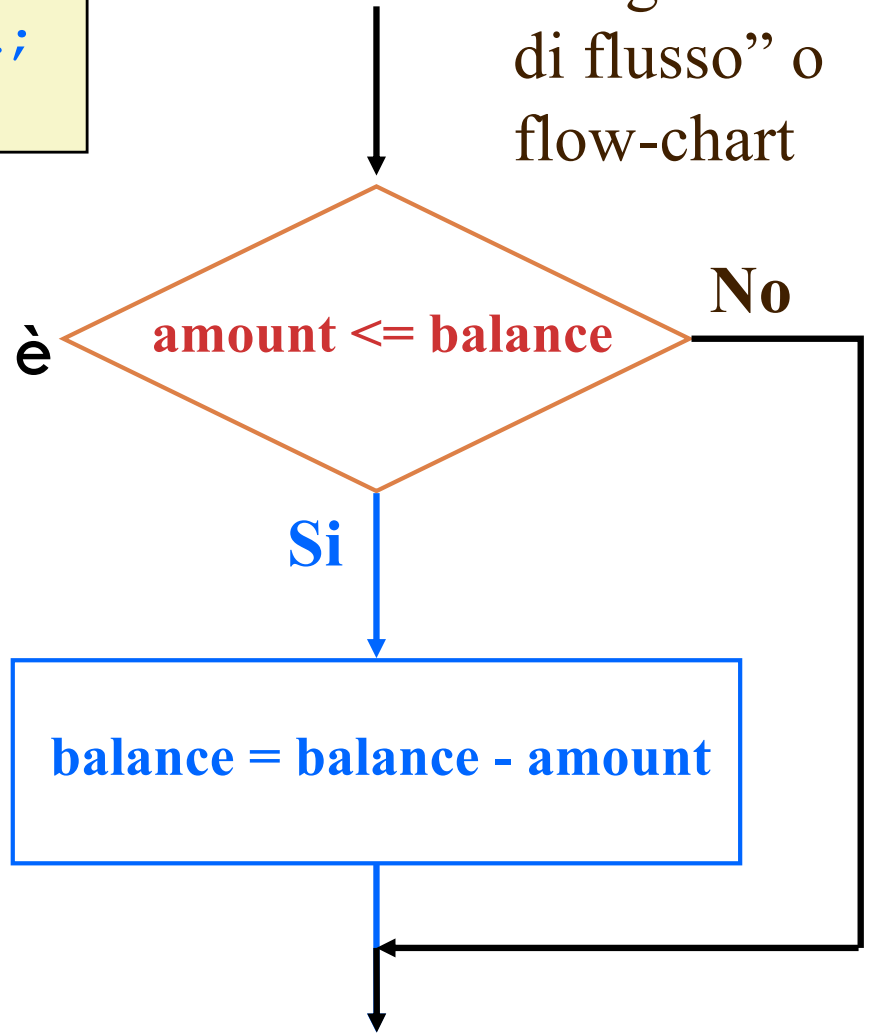
- È una situazione assai poco realistica!
- Il programma deve **controllare** il saldo e agire di conseguenza, consentendo il prelievo oppure no

L'enunciato if

```
if (amount <= balance){
   balance = balance - amount;
}
```

- L'enunciato **if** si usa per realizzare una decisione ed è diviso in due parti
  - una *verifica di condizione*
  - un *corpo*
- **Il corpo viene eseguito <u>se e solo se</u> la verifica ha successo**

Questo è un "diagramma di flusso" o flow-chart



Un nuovo problema

- Proviamo ora a visualizzare un messaggio d'errore se (e solo se!) il prelievo non è consentito

```
if (amount <= balance){
   balance = balance - amount;
}
if (amount > balance){
   System.out.println("Conto scoperto");
}
```

Sembra che funzioni!

Due problemi: uno e' piu' che altro un fastidio (e possibile fonte di errore), l'altro un guaio (e probabile fonte di errore)!

Un nuovo problema

- **Primo problema** di questo codice: se si modifica la prima condizione, bisogna ricordarsi di modificare anche la seconda
  - es. viene concesso un **FIDO** sul conto, che può così "andare in rosso"

```
if (amount <= balance + FIDO){
   balance = balance - amount;
}
if (amount > balance + FIDO){
   System.out.println("Conto scoperto");
}
```

- In generale, è DECISAMENTE preferibile **evitare**, per quanto possibile, **casi in cui una modifica richiede necessariamente un'altra modifica** conseguente in un diverso punto del codice

Un nuovo problema

```
if (amount <= balance){
   balance = balance - amount;
}
if (amount > balance){
   System.out.println("Conto scoperto");
}
```

Sembra che funzioni!

Invece no…

- **Secondo problema** di questo codice: se il corpo del primo **if** viene eseguito, la verifica del secondo **if** usa il ***nuovo*** valore di **balance**, dando luogo a un errore logico

Un nuovo problema

Sembra che funzioni! **Invece no…**

```
// assumiamo balance = 100
// assumiamo amount = 60

if (amount <= balance){          // 60<=100? VERO!
   balance = balance - amount;   // balance = 40
}
if (amount > balance){           // 60>40? VERO!
   System.out.println("Conto scoperto"); // STAMPO!
}
```

Un nuovo problema

- Questa situazione accade ***ogni volta che si preleva più della metà del saldo disponibile***: il prelievo viene correttamente eseguito (diminuendo il saldo), ma viene anche (erroneamente) visualizzato il messaggio di "Conto scoperto"
  - I due corpi, secondo l'algoritmo, dovrebbero essere sempre eseguiti **IN ALTERNATIVA**, ma così come è scritto non funziona
  - Come possiamo codificare correttamente questo algoritmo?

La clausola *else*

- Per realizzare un'***alternativa***, si utilizza **else**, la clausola facoltativa dell'enunciato **if**

```
if (amount <= balance){
   balance = balance - amount;
}
else{
   System.out.println("Conto scoperto");
}
```

- **Vantaggio**: ora c'è ***una sola verifica***
  - se la verifica ha successo, viene eseguito il ***primo*** corpo dell'enunciato **if/else**
  - *altrimenti*, viene eseguito il ***secondo*** corpo
    - NON VIENE RIPETUTA LA VERIFICA

E senza else?

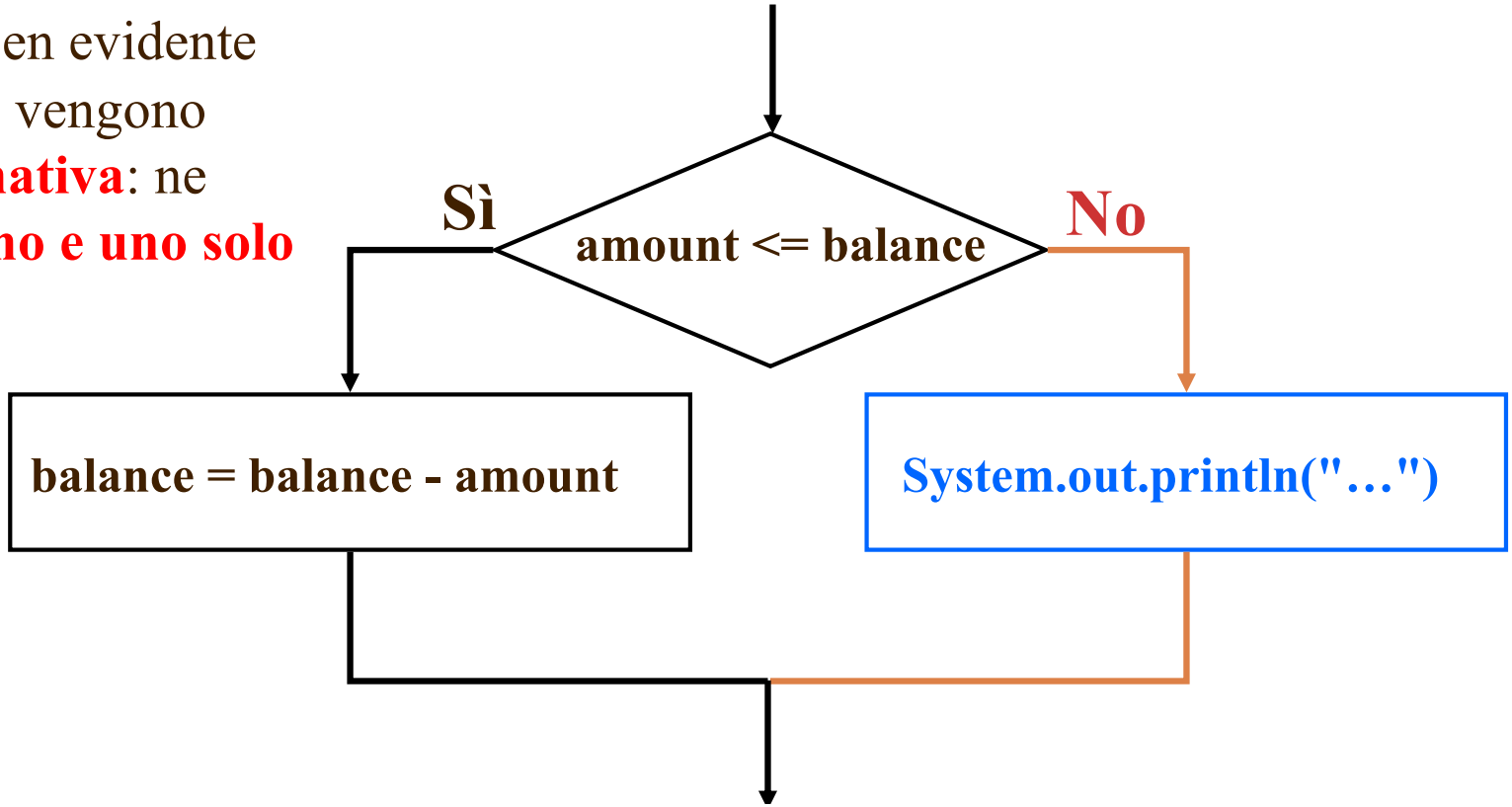
- Realizzare correttamente il medesimo algoritmo senza utilizzare la clausola **else**

```
double testBalance = balance; // eventualmente + FIDO...
if (amount <= testBalance){
   balance = balance - amount; // la modifica di balance
}                              // non ha ovviamente
                               // effetto su testBalance
if (amount > testBalance){
   System.out.println("Conto scoperto");
}
// nel seguito, ovviamente, bisogna usare balance,
// che contiene il valore correttamente aggiornato
// del saldo; testBalance non serve più
```

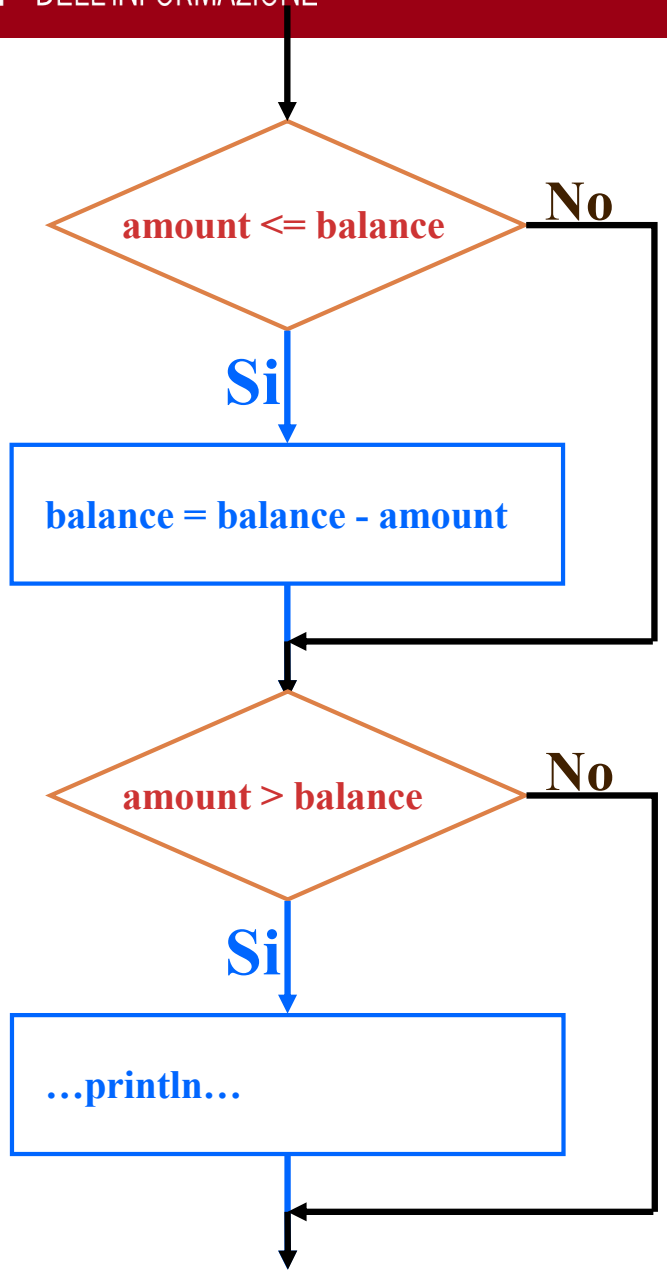
La clausola else

```
if (amount <= balance){
   balance = balance - amount;
}
else{
   System.out.println("Conto scoperto");
}
```

È graficamente ben evidente che i due blocchi vengono eseguiti in **alternativa**: ne viene eseguito **uno e uno solo**

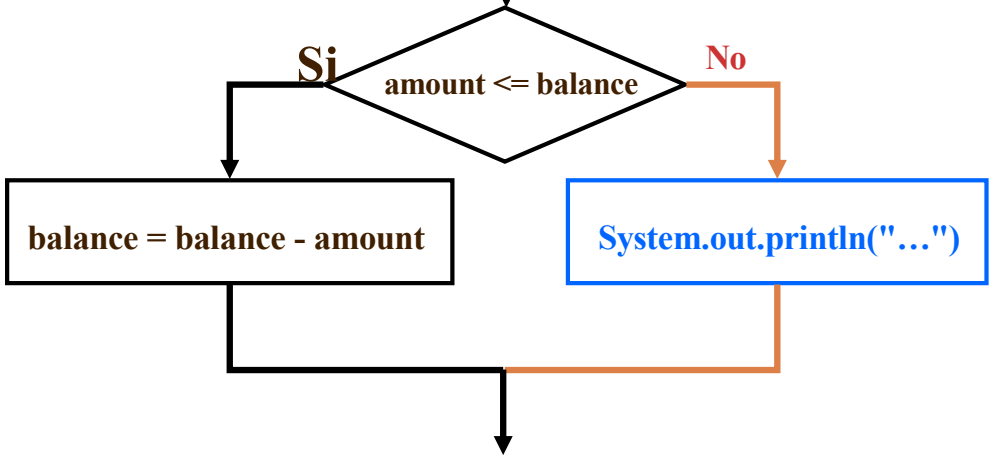


Confronto con e senza "else"




```
if (amount <= balance){
   balance = balance - amount;
}
if (amount > balance){
   System.out.println("Conto scoperto");
}
```

```
if (amount <= balance){
   balance = balance - amount;
}
else{
   System.out.println("Conto scoperto");
}
```



L'enunciato *if*

- Sintassi:

```
if (condizione)
   enunciato1
```

```
if (condizione)
   enunciato1
else
   enunciato2
```

- Scopo: eseguire ***enunciato1*** se e solo se la *condizione* è **vera**; se è presente la clausola **else**, eseguire ***enunciato2*** se e solo se la *condizione* è **falsa**
- Spesso il corpo di un enunciato **if** è costituito da ***più enunciati*** da eseguire ***in sequenza***; racchiudendo tali enunciati tra una coppia di parentesi graffe { } si crea un ***blocco di enunciati***, che può essere usato come corpo

```
if (amount <= balance)
{  balance = balance - amount;
   System.out.println("Prelievo accordato");
}
```

Errore frequente

- Se voglio scrivere

```
if (amount <= balance)
{   balance = balance - amount;
    System.out.println("Prelievo accordato");
}
```

- Ma mi dimentico le parentesi…

Solo questo enunciato e' legato all'if

```
if (amount <= balance)
    balance = balance - amount;
    System.out.println("Prelievo accordato");
```

- La stampa di "Prelievo accordato" avverrà anche se amount>balance e di fatto il prelievo non c'è stato!

Tipi di enunciato in Java

- Enunciato semplice, ad esempio

```
balance = balance - amount;
```

- Enunciato composto, ad esempio

```
if (x >= 0) x = x - 1;
```

- Blocco di enunciati

```
{ zero o più enunciati di qualsiasi tipo }
```

- **Ognuno di questi tre tipi di enunciati è, sintatticamente, un enunciato**
  - Quindi, può costituire il corpo di `if` o di `else`

Confrontare valori numerici

Confrontare valori

- Le ***condizioni*** dell'enunciato **if** sono molto spesso dei ***confronti tra due valori numerici***

```
if (x >= 0)
```

- Gli ***operatori di confronto*** si chiamano ***operatori relazionali***

Attenzione: negli operatori costituiti da due caratteri ***non*** vanno inseriti spazi intermedi

Java	Math Notation	Description
>	>	Greater than
>=	≥	Greater than or equal
<	<	Less than
<=	≤	Less than or equal
==	=	Equal
!=	≠	Not equal

Operatori relazionali

- Fare molta attenzione alla differenza tra operatore relazionale ==

  e

  operatore di assegnazione =

```
a = 5;          // assegna 5 ad a

if (a == 5)     // esegue enunciato
   enunciato    // se e solo se a è uguale a 5

if (a = 5)      // fortunatamente (e diversamente
   enunciato    // da altri linguaggi)
                // ERRORE IN COMPILAZIONE
```

Confrontare valori in virgola mobile

Confronto di numeri in virgola mobile

- I numeri in virgola mobile hanno una precisione limitata ed i calcoli possono introdurre errori di arrotondamento e troncamento
- Tali errori sono ***inevitabili*** e bisogna fare molta attenzione nella formulazione di verifiche che coinvolgono numeri in virgola mobile

Square.java

```
double r = Math.sqrt(2);
double x = r * r;
if (x == 2){
   System.out.println("OK");
}
else{
   System.out.println("Non ci credevi?");
}
```

Confronto di numeri in virgola mobile

- Per fare in modo che gli errori di arrotondamento non influenzino la logica del programma, i confronti tra numeri in virgola mobile devono prevedere una ***tolleranza***

$$|x - y| \leq \varepsilon \cdot \max(|x|, |y|) \qquad \text{con } \varepsilon = 10^{-14}$$

(questo valore di ε è ragionevole per **double**)

- Il codice per questa verifica di ***uguaglianza con tolleranza*** è (dopo aver definito **EPSILON**)

```
Math.abs(x - y) <=
    EPSILON * Math.max(Math.abs(x), Math.abs(y))
```

Confronto di numeri in virgola mobile

- Possiamo definire un metodo statico che verifichi l'uguaglianza con tolleranza

```
public class Numeric
{  public static boolean approxEqual(double x, double y)
   {
      final double EPSILON = 1E-14;
      double xyMax = Math.max(Math.abs(x),Math.abs(y));
      return Math.abs(x - y) <= EPSILON * xyMax;
   }
}
```

```
double r = Math.sqrt(2);
if (Numeric.approxEqual(r * r, 2)){
   System.out.println("Tutto bene...");
}
else{
   System.out.println("Questo non succede...");
}
```

Confrontare 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**

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

Confronto di stringhe

```
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

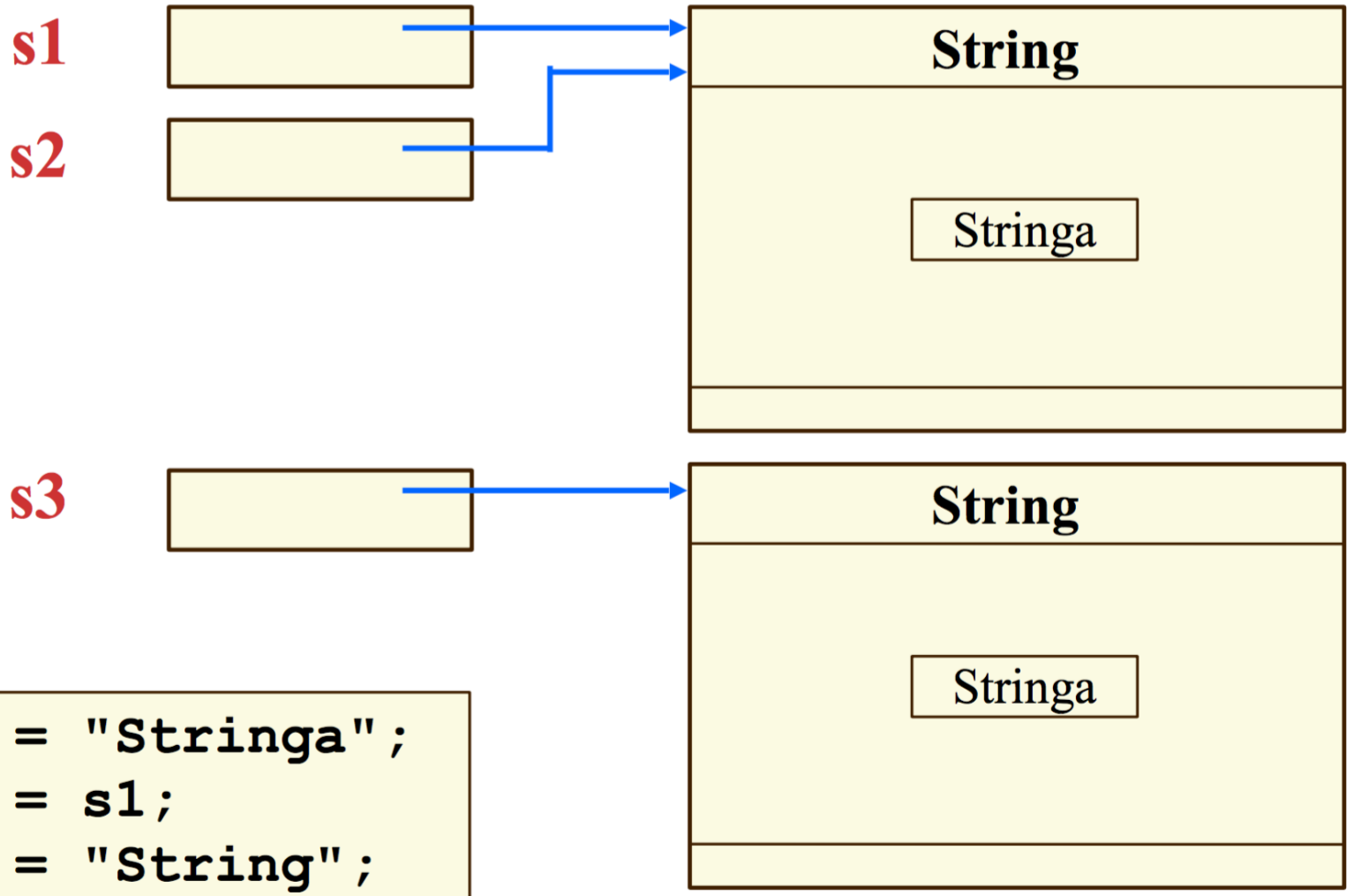
```
String s1 = "Stringa";
String s2 = s1;
String s3 = "String";
s3 = s3 + "a"; // s3 contiene "Stringa"
```

- Il confronto **s1.equals(s3)** è **vero**, perché tali oggetti hanno lo stesso contenuto (sono "identici")

- Nota: per verificare se un riferimento si riferisce a null si può utilizzare l'operatore di uguaglianza

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

Confronto di stringhe




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

Ordinamento lessicografico per stringhe

Ordinamento lessicografico

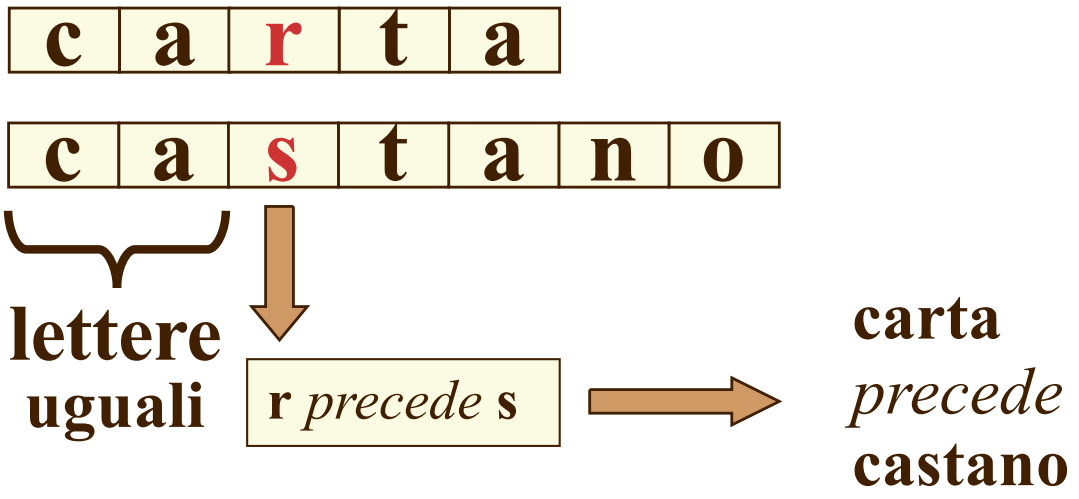
- Se due stringhe sono diverse, è possibile conoscere la relazione che intercorre tra loro secondo **l'ordinamento lessicografico**, simile al comune ordinamento alfabetico
  - Il confronto lessicografico tra stringhe si esegue con il metodo compareTo

```
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 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



Confronto lessicografico

- L'ordinamento tra caratteri in Java è **simile** all'ordinamento alfabetico comune, con qualche differenza… anche perché tra i caratteri non ci sono solo le lettere! Ad esempio
  - 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

Confronto di 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**

Confronto di oggetti

- 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

Ordinamento di oggetti

- 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)
```

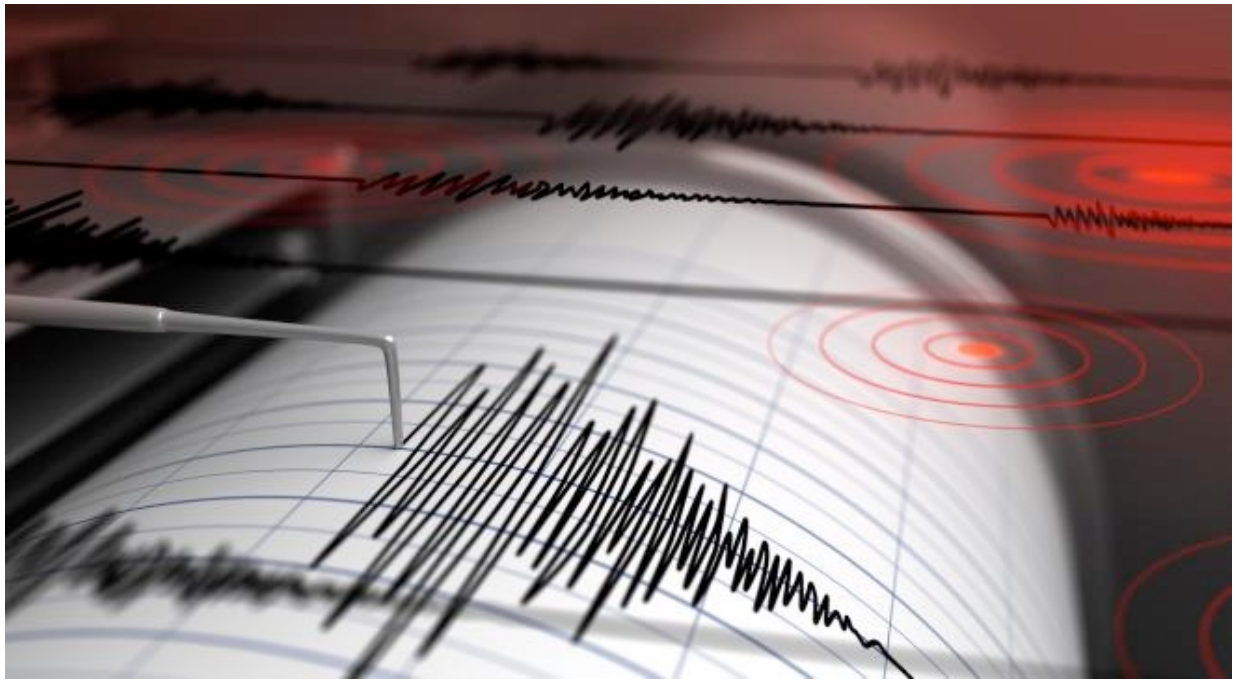
Table 2 Relational Operator Examples

Expression	Value	Comment
`3 <= 4`	`true`	3 is less than 4; <= tests for "less than or equal".
`3 =< 4`	**Error**	The "less than or equal" operator is <=, not =<. The "less than" symbol comes first.
`3 > 4`	`false`	> is the opposite of <=.
`4 < 4`	`false`	The left-hand side must be strictly smaller than the right-hand side.
`4 <= 4`	`true`	Both sides are equal; <= tests for "less than or equal".
`3 == 5 - 2`	`true`	== tests for equality.
`3 != 5 - 1`	`true`	!= tests for inequality. It is true that 3 is not 5 – 1.
`3 = 6 / 2`	**Error**	Use == to test for equality.
`1.0 / 3.0 == 0.333333333`	`false`	Although the values are very close to one another, they are not exactly equal. See Section 5.2.2.
`"10" > 5`	**Error**	You cannot compare a string to a number.
`"Tomato".substring(0, 3).equals("Tom")`	`true`	Always use the `equals` method to check whether two strings have the same contents.
`"Tomato".substring(0, 3) == ("Tom")`	`false`	Never use == to compare strings; it only checks whether the strings are stored in the same location. See Common Error 5.2 on page 192.

Alternative multiple

Problema: la scala Richter

- Scrivere una porzione di codice in cui, in base al valore registrato durante un terremoto utilizzando la scala Richter, si visualizzi un messaggio riguardante l'intensità del terremoto stesso.



Sequenze di confronti

- Se si hanno più di due alternative, si usa una **sequenza di confronti (voi mettete sempre le parentesi!)**

```
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
6<= richter <8

Domanda: è importante l'ordine?

```
if (richter >= 0)
     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");
```

Sequenze di confronti

- Il secondo codice 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");
```

Domanda: è necessario l'else?

```
if (richter >= 8)
   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");
```

Sequenze 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

Caso di studio

- Calcoliamo le tasse secondo le aliquote del sistema federale americano nel 2008

Table 4 Federal Tax Rate Schedule

If your status is Single and if the taxable income is	the tax is	of the amount over
at most $32,000	10%	$0
over $32,000	$3,200 + 25%	$32,000
If your status is Married and if the taxable income is	**the tax is**	**of the amount over**
at most $64,000	10%	$0
over $64,000	$6,400 + 25%	$64,000

- Ci sono due livelli nel processo decisionale
  - Sposato S/N? E in base alla risposta, diversi scaglioni di reddito con corrispondenti aliquote

Variabili d'istanza

- Avrò bisogno di due informazioni per descrivere un oggetto di una classe TaxReturn
  - Lo stato: int status: 1=single, 2=sposato
  - Il reddito: double income;
- E un costruttore

```
public class TaxReturn{

    // variabili d'istanza
    private int status;
    private double income;

    public TaxReturn(int aStatus, double anIncome){
        status = aStatus; income = anIncome;
    }
}
```

Costanti

- Definisco anche delle costanti per fissare le aliquote e i limiti degli scaglioni

```
public static final int SINGLE = 1;
public static final int MARRIED = 2;

public static final double RATE1 = 0.10;
public static final double RATE2 = 0.25;

public static final double SINGLE_LIMIT = 32000;
public static final double MARRIED_LIMIT = 64000;
```

Metodi

- Avrò bisogno di un metodo che calcoli le tasse in corrispondenza dei valori di esemplare di status e income
  - Non ha bisogno di parametri esterni
  - Restituisce un valore di tipo double

```
public class TaxReturn{

    // variabili d'istanza
    // costanti
    // costruttore

    // metodi
    public double getTax(){…}
}
```

Diramazioni annidate

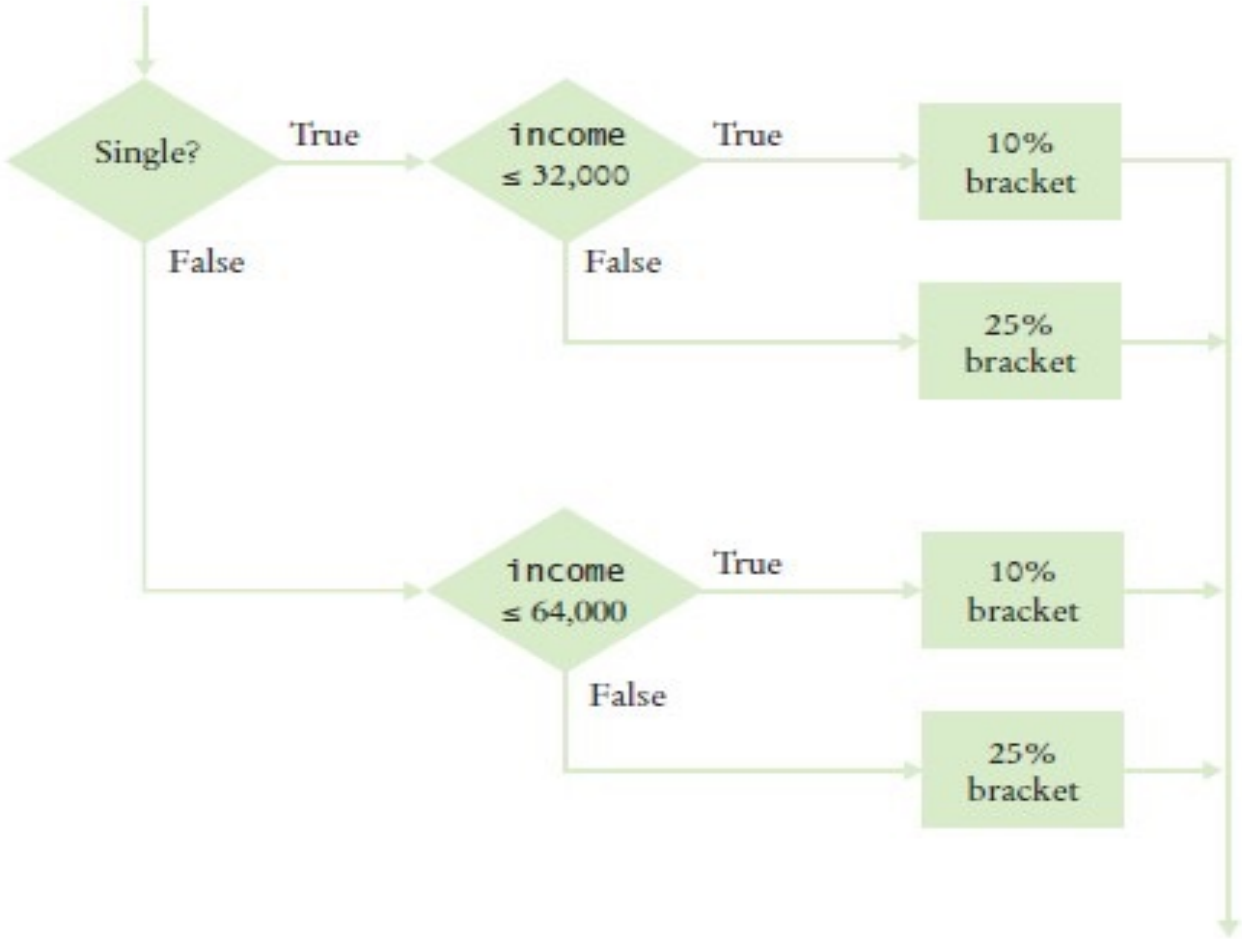


Figure 5 Income Tax Computation

Diramazioni annidate

```
public double getTax(){
   double tax1 = 0;
   double tax2 = 0;
   if (status == SINGLE){ // caso non sposato
      if (income <= SINGLE_LIMIT){
         tax1 = RATE1*income;
      }
      else {
         tax1 = RATE1*SINGLE_LIMIT;
         tax2 = RATE2*(income-SINGLE_LIMIT);
      }
   }
   else{ // caso sposato
      if (income <= MARRIED_LIMIT){
         tax1 = RATE1*income;
      }
      else {
         tax1 = RATE1*MARRIED_LIMIT;
         tax2 = RATE2*(income-MARRIED_LIMIT);
      }
   }
   return tax1+tax2;
}
```

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

Come usare la classe TaxReturn?

- Creo un'altra classe TaxCalculator, che metto in un file TaxCalculator.java e che rendo eseguibile

```
import java.util.Scanner;

public class TaxCalculator{
     public static void main(String[] args){
          // acquisisco i dati da input
          Scanner in = new Scanner(System.in);
          System.out.println("Inserisci il tuo reddito");
          double reddito = in.nextDouble();
          System.out.println("Sei sposato/a? (S/N)");
          String input = in.next();
          . . .
     }
}
```

Come usare la classe TaxReturn?

```
import java.util.Scanner;

public class TaxCalculator{
   public static void main(String[] args){
      . . .
      // interpreto la risposta sullo stato civile
      int stato;
      if(input.equals("S")){        //Attenzione!
         stato = TaxReturn.MARRIED;
      }
      else{
         stato = TaxReturn.SINGLE;
      }
      // creo un oggetto di tipo TaxReturn e agisco su di esso
      TaxReturn aTaxReturn = new TaxReturn(stato,reddito);
      System.out.println("Tasse: "+aTaxReturn.getTax());
    }
}
```

Come seleziono i test case?

- Coprire tutti i "percorsi" if/else e i casi limite

Test Case	Married	Expected Output	Comment
30,000	N	3,000	10% bracket
72,000	N	13,200	3,200 + 25% of 40,000
50,000	Y	5,000	10% bracket
104,000	Y	16,400	6,400 + 25% of 40,000
32,000	N	3,200	boundary case
0		0	boundary case

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**

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

- Ma… il compilatore ignora i rientri!
  - Il risultato ottenuto **non** è ciò che si voleva
  - La regola sintattica è che **una clausola else appartiene sempre all'enunciato if più vicino**

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 usare di nuovo **lo stesso nome** successivamente nel codice (ma fare molta attenzione!)

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

Riassunto enunciati decisionali

- Enunciato if

```
if (condizione){
    // istruzioni se condizione vera
}
```

  - Se la condizione e' vera esegui le istruzioni presenti nel blocco {…} che segue; se falsa non fare nulla

- Enunciato if con clausola else
  - Se la condizione e' vera esegui le istruzioni presenti nel blocco {…} che segue;
  - Altrimenti esegui il corpo dell'else

```
if (condizione){
  // istruzioni se condiz vera
}
else{
  // istruzioni se condiz falsa
}
```

Riassunto enunciati decisionali

Alternative multiple

```
if (condiz1){
  // istruzioni se condiz1 vera
}
else if (condiz2){
  // istruzioni se condiz1 falsa e
  // condiz2 vera
}
else if (condiz3){
  // istruzioni se condiz1 falsa,
  // condiz2 falsa e condiz3 vera
}
```

```
if (cond1){
     if (condiz2){
       // istruzioni se condiz1 e condiz2 vere
     }
     else{
       // istruzioni se condiz1 vera e condiz2 falsa
     }
}
else{
     if (condiz3){
       // istruzioni se condiz1 falsa e condiz3 vera
     }
     else{
       // istruzioni se condiz1 e condiz3 falsa
     }
}
```

Diramazioni annidate

Riassunto enunciati decisionali

- Confronti
  - Attenzione al confronto tra tipi double!
    - Fare confronto approssimato
  - Attenzione al confronto tra stringhe
    - Metodo non statico equals() : s1.equals(s2);
      - Restituisce true se le stringhe sono uguale carattere per carattere
    - Metodo non statico compareTo() : s1.compareTo(s2);
      - Restituisce un numero negativo se s1 precede s2 in ordine lessicografico
      - Restituisce un numero positivo se s1 segue s2 in ordine lessicografico
      - Restituisce 0 se s1 e s2 sono uguali
    - Confronto con ==
      - s1==s2restituisce true se i riferimenti alle stringhe s1 e s2 sono uguali (caso particolare quando assegno stringhe letterali uguali)

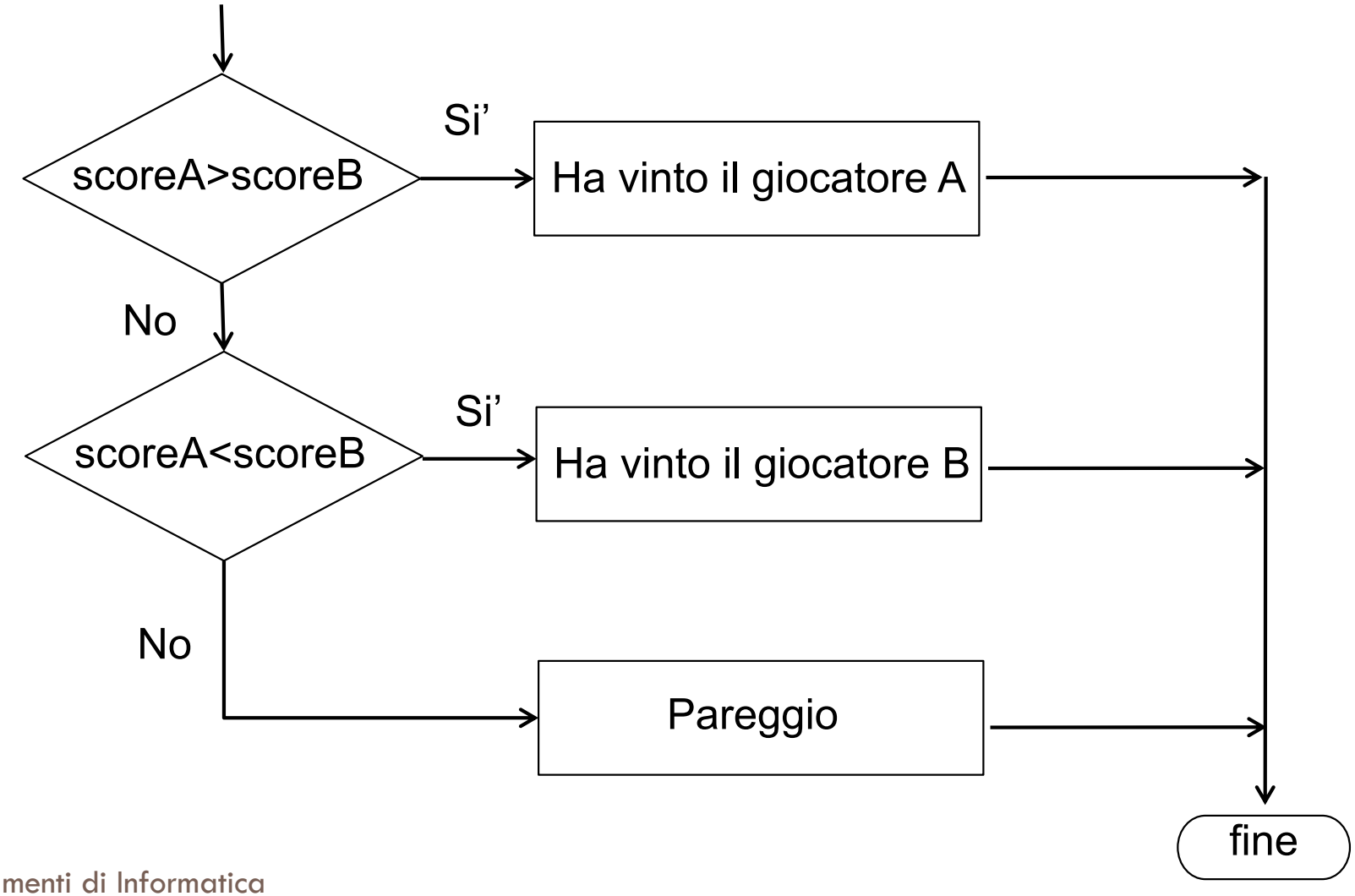
Riassunto enunciati decisionali

- Attenzione al confronto tra generici oggetti della classe C
  - Metodo non statico equals() : o1.equals(o2);
    - Se non sovrascrivo equals() nella classe C, restituisce true se i riferimenti agli oggetti o1 e o2 sono uguali
    - Se sovrascrivo equals() l'uguaglianza dipende da come la definisco io (tipicamente uguaglianza tra variabili d'istanza)
  - Metodo non statico compareTo() : o1.compareTo(o2);
    - Solo per oggetti "ordinabili"
    - Restituisce un numero negativo se o1 precede s2 nell'ordine che stabilisco
    - Restituisce un numero positivo se o1 segue o2 nell'ordine che stabilisco
    - Restituisce 0 se o1 e o2 sono uguali
  - Confronto con ==
    - o1==o2restituisce true se i riferimenti agli oggetti o1 e o2 sono uguali

Esercizio

- In un programma di un gioco il punteggio di due giocatori è mantenuto in due variabili intere scoreA e scoreB. Scrivere il flowchart e poi il frammento di codice corrispondente per mandare in uscita un messaggio che indichi se ha vinto il giocatore A, se ha vinto il giocatore B oppure se hanno pareggiato.

Flowchart


scoreA>scoreB
Si'
Ha vinto il giocatore A
No
scoreA<scoreB
Si'
Ha vinto il giocatore B
No
Pareggio
fine

Frammento di codice

```
if (scoreA > scoreB){
   System.out.println("Ha vinto il giocatore A");
}
else if (scoreA < ScoreB){
   System.out.println("Ha vinto il giocatore B");
}
else {
   System.out.println("Pareggio");
}
```

NB: perché funzioni devo metterlo nel main di una classe (es. Winner, salvata in un file Winner.java), dichiarare nel main due variabili intere scoreA e scoreB e inizializzarle (oppure leggere i loro valori da input)

Conversioni fra stringhe e numeri

Conversione di stringhe in numeri

- A volte si ha una stringa che contiene un valore numerico e si vuole assegnare tale valore a una variabile di tipo numerico, per poi elaborarlo

```
String password = "md50";
String ageString = password.substring(2);
// ageString contiene "50"
// NON FUNZIONA!
int age = ageString;
```

```
incompatible types
found    : java.lang.String
required: int
```

- Il compilatore segnala l'errore semantico perché non si può convertire automaticamente una stringa in un numero, dato che ***non vi è certezza che il suo contenuto rappresenti un valore numerico***
  - **Si ricordi che il comportamento del compilatore non dipende dai valori ma solo dai tipi…**

Conversione di stringhe in numeri

- La conversione corretta si ottiene invocando il metodo statico **parseInt** della classe **Integer**

```
int age = Integer.parseInt(ageString);
// age contiene il numero 50
```

Conversione di stringhe in numeri

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

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

  - **Non** viene usata la localizzazione, quindi la stringa deve contenere **sempre il punto** come separatore decimale
  - **Oppure usate la notazione scientifica**

Conversione di stringhe in numeri

- ***Cosa succede se la stringa passata come argomento non contiene un numero?***
  - i metodi **Integer.parseInt** e **Double.parseDouble** lanciano un'eccezione di tipo **NumberFormatException** e il programma termina segnalando l'errore

- Abbiamo già visto casi in cui il verificarsi di una eccezione arresta il programma
  - **StringIndexOutOfBoundsException** in **substring**

Conversione di stringhe in numeri

- Il meccanismo generale di segnalazione di errori in Java consiste nel "lanciare" (***throw***) un'***eccezione***
  - si parla anche di ***sollevare*** (*raise*) un'eccezione, ma "lanciare" è il termine più utilizzato

Ripasso: Conversione di numeri in stringhe

- Per ***convertire*** un numero in stringa si può ***concatenare il numero con la stringa vuota***

```
int ageNumber = 10;
String ageString = "" + ageNumber;
// ageString contiene "10"
```

Si sfrutta la semantica della concatenazione

- È però forse più "elegante" (e più comprensibile) utilizzare il metodo (statico) **toString** della classe **Integer**

```
int ageNumber = 10;
String ageString = Integer.toString(ageNumber);
```

Ripasso: Conversione di numeri in stringhe

- Per valori di tipo **double** si usa, analogamente, la stringa restituita da **Double.toString**
  - **Non** viene usata la localizzazione, quindi la stringa generata contiene **sempre il punto** come separatore decimale

Esempio

```
int n = 1534;
String s = Integer.toString(n); // s="1534"
```

```
String s = "1534";
int n = Integer.parseInt(s); // n=1534
```

Espressioni booleane

Espressioni relazionali

- Ogni espressione ha un valore
  - **x + 10** espressione aritmetica, valore numerico
  - **x < 10** espressione relazionale, valore booleano

- Un'espressione relazionale ha un valore vero o falso (**true** o **false**)

Il tipo di dati booleano

- I valori **true** e **false** non sono numeri, né oggetti, né classi: appartengono ad un tipo di dati diverso, detto **booleano**, dal nome del matematico George Boole (1815-1864), pioniere della logica

- è un tipo **fondamentale** in Java, come quelli numerici

Le variabili booleane

- Il tipo di dati boolean, come tutti gli altri tipi di dati, 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

Le variabili booleane

- Altre volte l'uso di una variabile booleana rende più leggibile il codice

- Spesso le variabili booleane vengono chiamate flags (bandiere), perché possono assumere soltanto due valori, cioè trovarsi in due soli stati possibili: su e giù, come una bandiera

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

Tabelle di verità

A	B	A&&B
true	true	true
true	false	false
false	true	false
false	false	false

AND: vero solo se A e B entrambe vere

A	B	A || B
true	true	true
true	false	true
false	true	true
false	false	false

OR: falso solo se A e B entrambe false

NOT: se A vero, allora !A falso e viceversa

A	!A
true	false
false	true

Gli operatori booleani o logici

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

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

- La valutazione di un'espressione con operatori booleani viene effettuata con una strategia detta "cortocircuito"
  - **la valutazione dell'espressione termina appena è possibile decidere il risultato**

Gli operatori booleani o logici

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

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

- Nell'esempio:
  - se x vale 15, l'ultima condizione non viene valutata, perché sicuramente l'intera espressione è vera

Gli operatori booleani o 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

```
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
```

Errori con operatori relazionali

- Alcune espressioni con operatori relazionali che ci appaiono "***naturali***" (per l'abitudine matematica) sono sintatticamente **errate** in Java, 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
```

Leggi di De Morgan

- Due leggi per semplificare espressioni logiche
  - Criterio per convertire un'espressione "negata" in un'espressione "affermata"

1. Legge `!(A && B)` è uguale a `!A || !B`
2. Legge `!(A || B)` è uguale a `!A && !B`

  - Gli operatori not vengono spostati sulle singole variabili
  - Gli operatori AND e OR vengono scambiati

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

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

Prima legge di De Morgan

A	B	A&&B	!(A&&B)
true	true	true	false
true	false	false	true
false	true	false	true
false	false	false	true

!A	!B	!A || !B
false	false	false
false	true	true
true	false	true
true	true	true

Inoltre… !(!A) = A

Table 5 Boolean Operator Examples

Expression	Value	Comment
`0 < 200 && 200 < 100`	`false`	Only the first condition is true.
`0 < 200 || 200 < 100`	`true`	The first condition is true.
`0 < 200 || 100 < 200`	`true`	The `||` is not a test for "either-or". If both conditions are true, the result is true.
`0 < x && x < 100 || x == -1`	`(0 < x && x < 100) || x == -1`	The `&&` operator has a higher precedence than the `||` operator (see Appendix B).
`0 < x < 100`	**Error**	**Error:** This expression does not test whether `x` is between 0 and 100. The expression `0 < x` is a Boolean value. You cannot compare a Boolean value with the integer 100.
`x && y > 0`	**Error**	**Error:** This expression does not test whether `x` and `y` are positive. The left-hand side of `&&` is an integer, `x`, and the right-hand side, `y > 0`, is a Boolean value. You cannot use `&&` with an integer argument.
`!(0 < 200)`	`false`	`0 < 200` is `true`, therefore its negation is `false`.
`frozen == true`	`frozen`	There is no need to compare a Boolean variable with `true`.
`frozen == false`	`!frozen`	It is clearer to use `!` than to compare with `false`.

F