



If a cascata, switch, operatori logici

Alla fine esercizi

# If a cascata

- Consideriamo una semplice classe che deve descrivere con una stringa gli effetti di un terremoto partendo da una valutazione dello stesso secondo la scala Richter
- A fasce di valori numerici sono associate descrizioni sintetiche degli effetti
- Come facciamo a individuare agevolmente la fascia di un certo valore dato?
- Ci viene in aiuto una tecnica che consiste nel mettere in cascata tanti comandi if-else quante sono le fasce

# Il codice

- Supponiamo di avere una variabile **richter** di tipo **double** e di restituire in **r** la descrizione:

```
if (richter >= 8.0)
    r = "Quasi la totalità delle strutture sono cadute";
else if (richter >= 7.0) // tra 7.0 e 8.0 (8.0 non compreso)
    r = "Molti edifici distrutti";
else if (richter >= 6.0) // 6.0 <= richter < 7.0
    r = "Molti edifici seriamente danneggiati e alcuni
    collassati";
else if (richter >= 4.5) // continua come sopra
    r = "Danni a edifici poco resistenti";
else if (richter >= 3.5)
    r = "Sentito da molte persone, non distruttivo";
else if (richter >= 0)
    r = "Generalmente non avvertito dalle persone";
else // richter < 0
    r = "Magnitudo negativa non valida";
```

# If a cascata

- Si noti che gli if non sono in sequenza, bensì sono annidati uno dentro l'altro

```
if (condizione)
```

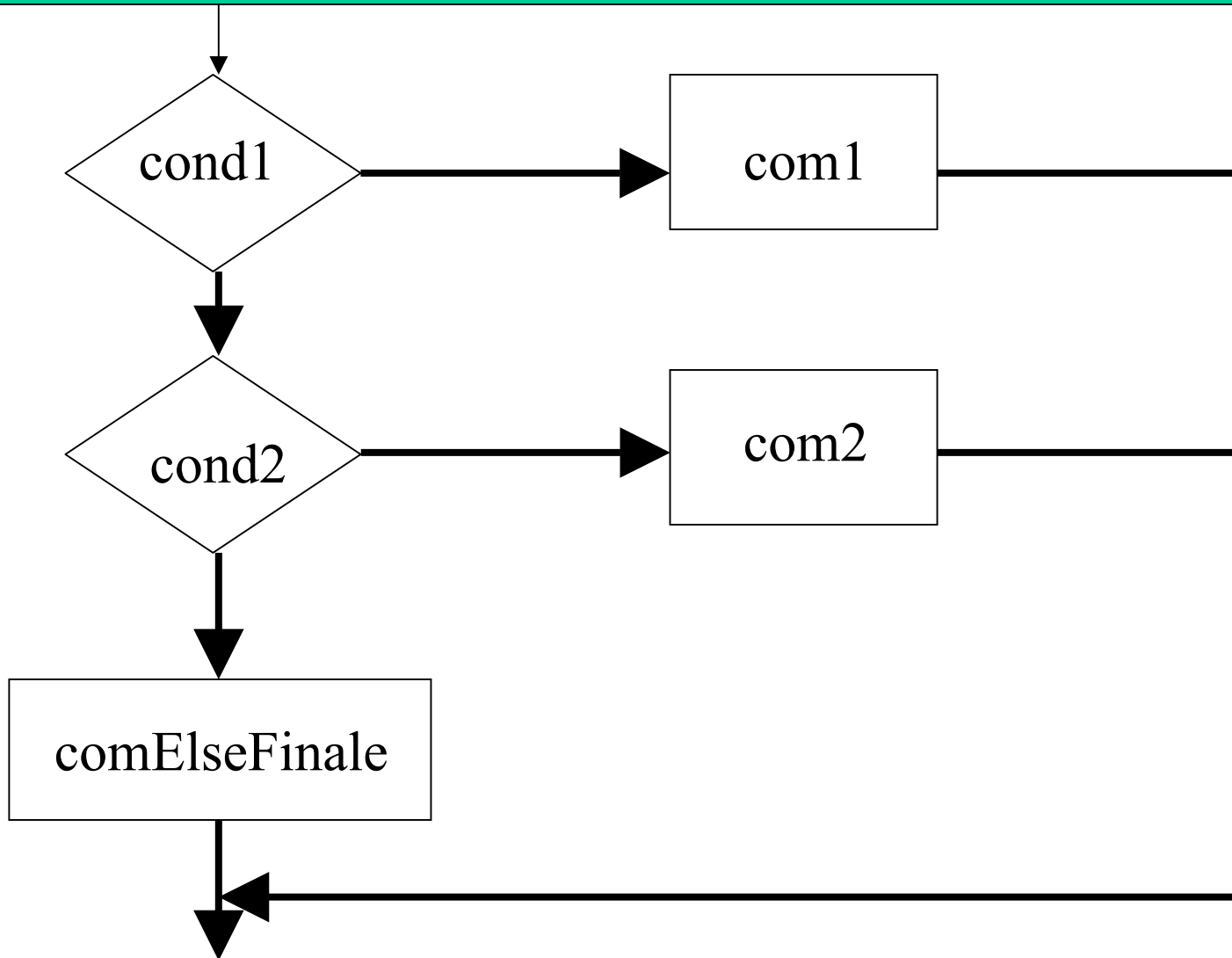
```
    comando
```

```
else if (condizione)
```

```
    comando
```

```
else ...
```

# If a cascata: diagramma



# If a cascata

- Se arrivo a valutare cond2 significa che cond1 è risultata falsa
- Se arrivo ad eseguire il comando dell'else finale significa che tutte le condizione sono risultate false
- Le condizioni sono valutate in sequenza
- I casi vanno individuati tenendo presente questi fatti

# If a cascata

- Ad esempio se avessimo cominciato dal grado più basso, invece che dal più alto, il programma non sarebbe stato giusto:

```
if (richter >= 0)
    r = "Generalmente non avvertito dalle persone";
else if (richter >= 3.5)
    r = "Sentito da molte persone, non distruttivo";
else if ....
```

- Se prendiamo `richter = 7.5` vediamo che la prima condizione è vera e quindi il programma stamperebbe “Generalmente non avvertito dalle persone” per un terremoto di grado 7.5 (e per un qualunque altro grado non negativo!)

# switch

- Per un caso particolare di if a cascata esiste un costrutto apposito
- Si usa quando le condizioni da verificare sono tutte dei confronti di una variabile intera con delle costanti
- Il costrutto si chiama **switch**
- Vediamo un esempio



# switch: il codice

```
switch ( digit ) { // Variabile intera digit
    case 1: englishDigit = "One"; break;
    case 2: englishDigit = "Two"; break;
    case 3: englishDigit = "Three"; break;
    case 4: englishDigit = "Four"; break;
    case 5: englishDigit = "Five"; break;
    case 6: englishDigit = "Six"; break;
    case 7: englishDigit = "Seven"; break;
    case 8: englishDigit = "Eight"; break;
    case 9: englishDigit = "Nine"; break;
    default: englishDigit = "Errore"; break;
}
```

# switch

- All'interno delle parentesi tonde si può mettere una variabile a valori interi
- Dentro al blocco si possono inserire quante righe “case” si vogliono. Il numero che segue **case** viene confrontato con la variabile e se sono uguali allora viene eseguito il codice che c'è dopo i due punti fino al prossimo “case” o al “default”
- Il caso default è vero se tutti i casi **precedenti** sono risultati falsi (la variabile ha valore diverso da tutti i casi)

# switch

- Attenzione!
- Se un certo “case” risulta vero allora viene eseguito il codice ad esso associato, **ma anche il codice di tutti i casi restanti!**
- Per evitare questo comportamento, che la maggior parte delle volte non è quello che si vuole, è possibile ricorrere al comando **break**

# break

- Il comando **break** serve per uscire dal blocco di codice corrente
- Il suo effetto è quello di buttare via il frame relativo al blocco corrente e a far proseguire l'esecuzione dal comando che segue il blocco
- **break** può essere usato solo all'interno del blocco di uno **switch** oppure all'interno del blocco di un ciclo (**while**, **do**, **for**)

# switch

- Nel nostro esempio abbiamo inserito **break** alla fine del codice di ogni caso
- Ad esempio se avessimo dimenticato di metterlo nel primo caso:

```
switch ( digit ) {  
    case 1: englishDigit = "One"; // manca  
    case 2: englishDigit = "Two"; break;  
    ...  
}
```

- Se `digit = 1` il vengono eseguiti gli assegnamenti del caso 1 e del caso 2 e quindi alla fine `englishDigit` vale "Two"!

# Else sospeso

- Abbiamo già visto questo argomento a Programmazione
- Anche in Java, come in Pascal e C, la risoluzione dell'ambiguità del comando condizionale viene risolta con questa convenzione:
- Ogni else è associato all'if non associato con un else" più vicino (andando a ritroso)

# Il tipo boolean

- **boolean** è un tipo base di Java che indica due possibili valori soltanto: vero o falso
- Le costanti **true** e **false** sono gli unici due valori booleani
- Una espressione booleana è una qualunque espressione che ha come valore un boolean
- Es: `amount <= balance`

# Metodi predicativi

- Essendo un tipo base, **boolean** può essere il valore restituito da un metodo
- I metodi di questo tipo si chiamano predicativi
- Un esempio su **BankAccount**:

...

```
public boolean isOverdrawn() {  
    return balance < 0;  
}
```

...

- Dice se il conto è in rosso (**true**) oppure no



# Uso di valori booleani

- I valori booleani sono richiesti nelle condizioni degli if e dei cicli
- In queste occasioni si possono inserire espressioni booleane, chiamate di metodi predicativi o variabili booleane
- `if (x > 0) ... // Espressione booleana`
- `if (myAccount.isOverdrawn()) ... // Metodo`
- `boolean inRosso = myAccount.isOverdrawn();  
if (inRosso) ... // Variabile booleana`

# Operatori logici

- Finora abbiamo visto espressioni booleane semplici, cioè che usano:
  - operatori di confronto fra valori
  - metodi predicativi
  - variabili booleane
- Tutti questi tipi di espressioni semplici possono venire combinati con gli operatori logici

# Operatori Logici

| Operatore Java    | Notazione logica                           | Descrizione      |
|-------------------|--------------------------------------------|------------------|
| <b>&amp;&amp;</b> | $\wedge$                                   | And logico pigro |
| <b>&amp;</b>      | $\wedge$                                   | And logico       |
| <b>  </b>         | $\vee$                                     | Or logico pigro  |
| <b> </b>          | $\vee$                                     | Or logico        |
| <b>!</b>          | $\neg$ o $\overline{\phantom{x}}$ o $\sim$ | Negazione        |

# Grammatica

$\langle \text{BoolExp} \rangle \rightarrow \langle \text{BoolExp} \rangle \mid \mid T$

$\mid \langle \text{BoolExp} \rangle \mid T$

$\mid T$

$T \rightarrow T \ \&\& \ F \mid T \ \& \ F \mid F$

$F \rightarrow !\langle \text{Operand} \rangle \mid \langle \text{Operand} \rangle$

$\langle \text{Operand} \rangle \rightarrow (\langle \text{BoolExp} \rangle) \mid$

$\mid \langle \text{Espressioni Booleane Semplici} \rangle$

# Grammatica

- Ne risulta che l'ordine di precedenza, da quello che lega di più a quello che lega di meno, è
  1. Negazione
  2. And
  3. Or
- Associatività per And e Or a sinistra

# Valutazione pigra

- Cosa significa esattamente “and pigro” e “or pigro”?
- La valutazione delle espressioni booleane può essere ottimizzata in questo modo:
- Si comincia a guardare il valore del primo operando sulla sinistra
- Se il valore di tutta l'espressione può essere inferito da subito allora il secondo operando non viene valutato per niente (versione pigra)

# Valutazione pigra: quando si applica

- L'espressione  
<primo operando> && <secondo operando>
- È sicuramente falsa se il primo operando ha valore **false**
- Oppure:  
<primo operando> || <secondo operando>
- È sicuramente vera se il primo operando ha valore **true**
- In questi due casi gli operatori pigri (&& e ||) non valutano per niente il secondo operando

# Valutazione pigra

- Naturalmente <primo operando> e <secondo operando> possono essere espressioni booleane qualsiasi (è l'albero di derivazione che ci dà la struttura e l'operatore “più in alto”)
- I corrispondenti operatori non pigri (& e !) valutano **sempre** il secondo operando
- La valutazione pigra è quella maggiormente usata anche perché, oltre all'ottimizzazione, permette di scrivere codice più compatto



# Valutazione pigra: esempio

- Un caso in cui è utile usare la valutazione pigra può essere quello di acquisizione di input e suo successivo parsing
- Sappiamo che leggendo da una finestra grafica una stringa `input`, se l'utente preme "Annulla", `input` avrà valore `null` e verrà sollevata un'eccezione dal metodo di parsing se proviamo a fare il parsing passando un riferimento `null` come stringa

# Valutazione pigra: esempio

- Il seguente `if` usa la valutazione pigra:

```
if (input != null && Integer.parseInt(input) > 0)
    System.out.println("Hai scritto un numero positivo");
else System.out.println("O hai cancellato o hai" +
    "scritto un numero non positivo");
```

- Se `input == null` allora il parsing non viene effettuato perché l'`&&` pigro assegna subito il valore `false` a tutta l'espressione
- In questo modo si evita di far sollevare un'eccezione a `Integer.parseInt` nel caso in cui l'utente ha annullato

# Esercizio

- Modificare il risolutore di equazioni di secondo grado in modo che
  - se il discriminante ha valore negativo stampi un messaggio che indichi che non ci sono soluzioni reali dell'equazione
  - se il discriminante è zero stampi un messaggio che indica che ci sono due soluzioni reali coincidenti + le soluzioni
  - se il discriminante è positivo stampi entrambe le soluzioni
- Aggiungere alla classe `RisolutoreEquazione2Grado` un metodo predicativo `hasSolution()` che indichi se l'equazione rappresentata dall'oggetto ha soluzioni reali

# Esercizio

- Scrivere un programma che prenda in ingresso un dato che descrive una carta da gioco con le seguenti abbreviazioni
  - A = Asso
  - 2...10 = punteggi delle carte
  - J = Jack
  - D = Donna
  - R = Re
  - Q = Quadri
  - C = Cuori
  - P = Picche
  - F = Fiori
- continua →

# Esercizio cont'd

- Il programma deve stampare la descrizione della carta a partire dal codice

- Esempio:

**Inserisci il codice:**

**DP**

**Donna di picche**

- Definire la classe Card il cui costruttore prende il codice inserito e il cui metodo getDescription restituisce la stringa di descrizione

# Esercizio

- Scrivere un programma che legga in input quattro stringhe e stampi quella più in alto e quella più in basso nell'ordine alfabetico

# Esercizio

- Un anno è bisestile se è divisibile per quattro (es. 1980) tranne nei casi in cui è divisibile per 100 (es. 1900)
- Tuttavia, è comunque un anno bisestile se è divisibile per 400 (es. 2000)
- Queste eccezioni, però, non erano in vigore prima del 15 ottobre 1582
- Scrivere un programma che chieda in ingresso un anno e che calcoli se l'anno è bisestile (implementare una classe Year con il metodo predicativo `isLeapYear()` – leap = bisestile in inglese)

# Esercizio

- Migliorare la classe BankAccount controllando:
  - Che non ci siano mai importi negativi come parametri di deposit e withdraw
  - Che non ci siano mai prelievi che portino ad un saldo negativo



# Esercizio

- Scrivere una classe i cui oggetti hanno nello stato i dati relativi a un dipendente, alle ore (e frazioni di esse) che ha lavorato in una certa settimana e alla retribuzione per ora
- Gli straordinari (oltre le 40 ore settimanali) vanno pagati il 150% delle ore normali
- Scrivere un metodo `getImportoPercepito` che ritorni il dovuto al dipendente

# Esercizio

- Realizzare una classe che rappresenta una serratura a combinazione che ha una ruota a 26 posizioni etichettate da A a Z (alfabeto inglese)
- La ruota deve essere impostata tre volte e la serratura si apre se la combinazione è corretta
- Quando la serratura, aperta, viene richiusa si può impostare una nuova combinazione
- Se la ruota viene impostata più di tre volte, le ultime tre impostazioni sono quelle che determinano se aprire o meno la serratura → continua

# Esercizio cont'd: interfaccia pubblica

```
/** Un oggetto serratura con combinazione può essere
    impostato tre volte a una certa posizione.
    La serratura si apre solo se la combinazione di tre
    lettere data è corretta
    */
public class CombinationLock {
    /** Costruisce una serratura con una data combinazione
        e la chiude
        @param aCombination la combinazione che deve
            essere una stringa di 3 lettere maiuscole
            dell'alfabeto inglese
        */
    public CombinationLock(String aCombination) {
    }
```

# Esercizio cont'd: interfaccia pubblica

```
/** Imposta la ruota su una posizione.  
    @param aPosition una stringa di una  
    sola lettera maiuscola  
    */  
public void setPosition(String  
                           aPosition) {  
    }  
/** Tenta di aprire la serratura  
    */  
public void unlock() {  
    }
```

# Esercizio cont'd: interfaccia pubblica

```
/** Controlla se la serratura è aperta.  
    @return true se la serratura è  
            attualmente aperta  
 */  
public boolean isOpen() {  
}  
/** Chiude la serratura lasciando la  
    combinazione attuale  
 */  
public void lock() {  
}
```

# Esercizio cont'd: interfaccia pubblica

```
/** Chiude la serratura e reimposta la
    combinazione
    Funziona solo se la serratura è
    davvero attualmente aperta
    @param aCombination la stringa
        contenente la nuova combinazione
    */
public void
    lockAndSetCombination(String
                            aCombination) {
}
}
```