

Java

*controllo del flusso di programma*

G. Prencipe  
prencipe@di.unipi.it

## Controllo del flusso

- In Java, i dati vengono manipolati utilizzando operatori
- Le scelte durante l'esecuzione vengono effettuate tramite comandi che controllano il flusso dell'esecuzione
- Molti dei comandi e degli operatori sono in Java sono comuni al C e al C++

## Operatori

- Un operatore prende uno o più argomenti e produce un nuovo valore
- Quasi tutti gli operatori funzionano solo con i tipi primitivi
- Le eccezioni sono
  - = Assegnamento
  - == Uguaglianza
  - != Diversoche funzionano con tutti gli oggetti
- **String** supporta anche + e += (concatenazione)

## Precedenza

- La precedenza degli operatori stabilisce come le espressioni sono valutate
  - Moltiplicazione e divisione hanno precedenza su addizione e sottrazione
  - Per rendere esplicite le precedenze, si possono usare le parentesi

$a = x + y - 2/2 + z$

Diverso da

$a = x + (y - 2)/(2 + z)$

## Esercizio

---

- Provare a usare l'operatore che concatena le stringhe
  - Stampate a console diverse parole e interi concatenati

## Assegnamento

---

- L'assegnamento è ottenuto con l'operatore =
  - `a=4;`
  - Non si possono assegnare valori alle costanti
- L'assegnamento che coinvolge valori primitivi è semplice
  - `a=b`, con a e b primitivi: il valore di b è copiato in a
  - Chiaramente, modifiche ad a non influenzano b

## Assegnamento fra oggetti

---

- Diversa è la situazione con l'assegnamento che coinvolge oggetti
- Ricordiamo che manipolare un oggetto significa manipolare il suo riferimento (maniglia)
- Quindi, assegnare "un oggetto a un altro" vuol dire copiare un riferimento da un posto a un altro

## Assegnamento fra oggetti

---

- `c=d`, con c e d oggetti: c e d puntano entrambi all'oggetto che era inizialmente riferito solo da d
- In altre parole, **non viene duplicato** l'oggetto riferito da d!!

## Esempio

Quali saranno i  
valori attesi delle tre  
`println`?

```
class Number {int i;}

public class Assignment {
    public static void main(String[] args) {
        Number n1 = new Number();
        Number n2 = new Number();
        n1.i = 9;
        n2.i = 47;
        System.out.println("1: n1.i: " + n1.i + " ", n2.i: " + n2.i);
        n1 = n2;
        System.out.println("2: n1.i: " + n1.i + " ", n2.i: " + n2.i);
        n1.i = 27;
        System.out.println("3: n1.i: " + n1.i + " ", n2.i: " + n2.i);
    }
}
```

## Esempio

- Quindi, alla fine `n1` e `n2` contengono lo *stesso* riferimento, che punta allo *stesso* oggetto
  - Il riferimento originale che conteneva `n1` è stato in effetti sovrascritto, e il relativo oggetto perduto (verrà rimosso dal garbage collector)
- Questo fenomeno è noto con il nome di *aliasing*, ed è il modo con cui Java tratta gli oggetti

## Esempio

- Domanda: come potremmo fare nel nostro esempio per evitare l'*aliasing*?
  - Cioè tenere i due oggetti separati, ma assegnare loro gli stessi valori?
  - ????

## Esempio

- Domanda: come potremmo fare nel nostro esempio per evitare l'*aliasing*?
  - Cioè tenere i due oggetti separati, ma assegnare loro gli stessi valori?
  - `n1.i=n2.i;`
  - Manipolare i campi negli oggetti non sempre è pulito e va comunque *contro* i buoni principi della programmazione OO
    - Ricordate? Interfacce, mandare messaggi agli oggetti, e simili....

## Aliasing

- L'aliasing occorre anche quando si passano oggetti ai metodi come argomenti
- Vediamo un esempio

## Esempio

In molti linguaggi, f() farebbe una copia di y al suo interno (non modificando l'oggetto)  
In Java, è passato un riferimento, e quindi viene modificato l'oggetto esterno a f()

```
class Letter {char c;}  
public class PassObject {  
    static void f(Letter y) {y.c = 'z';}  
    public static void main(String[] args) {  
        Letter x = new Letter();  
        x.c = 'a';  
        System.out.println("1: x.c: " + x.c);  
        f(x);  
        System.out.println("2: x.c: " + x.c);  
    }  
} ///:~
```

Quindi, quali sono i risultati della **println()**?

## Esempio

```
class Letter {char c;}  
public class PassObject {  
    static void f(Letter y) {y.c = 'z';}  
    public static void main(String[] args) {  
        Letter x = new Letter();  
        x.c = 'a';  
        System.out.println("1: x.c: " + x.c);  
        f(x);  
        System.out.println("2: x.c: " + x.c);  
    }  
} ///:~
```

## Operatori matematici

- Sono gli stessi disponibili nei comuni linguaggi di programmazione
  - +, -, \*, /, %
  - Nota: la divisione intera tronca, invece di arrotondare il risultato
- Si può eseguire una operazione e un assegnamento con la stessa istruzione
  - x+=4
    - Aggiunge 4 a x e assegna il risultato a x

## Esempio

- Osserviamo l'uso degli operatori matematici con l'esempio MathOps....
- Nell'esempio viene utilizzata la classe **Random**
  - Controllare nella documentazione cosa fa e dare un'occhiata ai suoi metodi

## MathOps.java

- Prima definiamo due metodi statici per stampare interi e float

```
public class MathOps {  
    static void printInt(String s, int){  
        System.out.println(s + " = " + i);  
    }  
    static void printFloat(String s, float f){  
        System.out.println(s + " = " + f);  
    }  
    /* continua */  
}
```

## MathOps.java

```
public static void main(String[] args) {  
    Random rand = new Random(); //Crea un generatore di numeri casuali  
    int i, j, k; //Test con gli interi  
    j = rand.nextInt(100) + 1; //Sceglie un numero tra 1 e 100  
    k = rand.nextInt(100) + 1;  
    printInt("j", j); printInt("k", k);  
    i = j + k; printInt("j + k", i);  
    i = j - k; printInt("j - k", i);  
    i = k / j; printInt("k / j", i);  
    i = k * j; printInt("k * j", i);  
    i = k % j; printInt("k % j", i);  
    j %= k; printInt("j %= k", j);  
    /* continua */  
}
```

## MathOps.java

```
float u,v,w; //Test con i float  
v = rand.nextFloat(); //Esistono anche nextLong() e nextDouble()  
w = rand.nextFloat();  
printFloat("v", v); printFloat("w", w);  
u = v + w; printFloat("v + w", u);  
u = v - w; printFloat("v - w", u);  
u = v * w; printFloat("v * w", u);  
u = v / w; printFloat("v / w", u);  
u += v; printFloat("u += v", u);  
u -= v; printFloat("u -= v", u);  
u *= v; printFloat("u *= v", u);  
u /= v; printFloat("u /= v", u);  
}
```

## Operatori unari

- Come operatori unari troviamo - e + con i significati che ci attendiamo
  - - inverte il segno
  - + in pratica non ha alcun effetto

```
x = -a
x = a * -b
x = a * (-b)
```

## Auto-incremento e -decremento

- Gli operatori di auto-incremento e auto-decremento sono simili a quelli presenti in C
- Non solo modificano la variabile (o espressione) a cui sono applicati, ma producono come risultato il valore della variabile stessa
  - ++a; è equivalente a a=a+1;

## Auto-incremento e -decremento

- Esistono due versioni di questi operatori
  - **Prefisso**: l'operatore compare prima della variabile (o espressione)
    - Viene prima effettuata l'operazione, e poi prodotto il valore
  - **Postfisso**: l'operatore compare dopo la variabile (o espressione)
    - Viene prima prodotto il valore, e poi effettuata l'operazione

## Esempio

Quali sono i risultati delle **println()**?

```
public class AutoInc {
    public static void main(String[] args) {
        int i = 1;
        System.out.println("i : " + i);
        System.out.println("++i : " + ++i);    // Pre-incremento
        System.out.println("i++ : " + i++);    // Post-incremento
        System.out.println("i : " + i);
        System.out.println("--i : " + --i);    // Pre-decremento
        System.out.println("i-- : " + i--);    // Post-decremento
        System.out.println("i : " + i);
    }
} //~
```

## Operatori relazionali

- Questi operatori generano risultati **booleani**
  - Valutano la relazione esistente tra gli operandi
  - Producono **true** se la relazione è vera, **false** altrimenti
- Sono
  - >, <, ≥, ≤
  - == (equivalenza)
  - != (non equivalenza)
- Bisogna stare attenti quando si usano == e != con gli oggetti....

## Esercizio

- Provare a scrivere una classe con unico metodo il **main**
- Creare nel **main** due **Integer**
- Stampare a console i risultati di == e != applicati ai due oggetti
- Notate niente di strano??

## Codice dell'esercizio

Il risultato è  
**false e true**  
Come mai??

```
public class Equivalence {  
    public static void main(String[] args) {  
        Integer n1 = new Integer(47);  
        Integer n2 = new Integer(47);  
        System.out.println(n1 == n2);  
        System.out.println(n1 != n2);  
    }  
} ///:~
```

## Equivalenza fra oggetti

- L'esercizio produce come risultato **false e true**
- Il motivo è che gli operatori di uguaglianza sono applicati ai *referimenti* degli oggetti
- È vero che il *contenuto* dei due oggetti è lo stesso, ma i due *referimenti* sono ovviamente diversi (essendo due oggetti *diversi*)

## Equivalenza fra oggetti

- Per confrontare il contenuto di due oggetti bisogna usare il metodo **equals()** che esiste per *tutti* gli oggetti (non primitivi, per cui vanno benissimo == e !=)
- Provate a utilizzarlo nell'esercizio appena svolto

## Equivalenza fra oggetti

```
public class EqualsMethod {  
    public static void main(String[] args) {  
        Integer n1 = new Integer(47);  
        Integer n2 = new Integer(47);  
        System.out.println(n1.equals(n2));  
    }  
} ///:~
```

Ora il risultato è **true**!!

## Ma la storia non è finita....

- Proviamo a creare una nuova classe **Value** che contiene un'unica variabile **int**
- Nel **main** creiamo due oggetti **Value**, e confrontiamoli con **equals()**
- Cosa succede??

## EqualsMethod2

Il risultato è di nuovo **false**

Come mai??

```
class Value {int i;}  
public class EqualsMethod2 {  
    public static void main(String[] args) {  
        Value v1 = new Value();  
        Value v2 = new Value();  
        v1.i = v2.i = 100;  
        System.out.println(v1.equals(v2));  
    }  
} ///:~
```



## equals()

- Il risultato è di nuovo **false** perché per default **equals()** confronta i *referimenti*
- Per ottenere il confronto fra i *contenuti* degli oggetti, bisogna fare *overriding*
  - *Overriding*: cambiare il comportamento di una funzionalità in una sottoclasse
- Questo perché il confronto dipende da come sono implementati gli oggetti e quindi cambia a seconda dell'oggetto
- La maggior parte delle classi della libreria Java implementano **equals()** in modo da confrontare il *contenuto* degli oggetti

## Esercizio

- Scrivere un metodo che prende due **Stringhe** come argomento, e utilizza tutti i gli operatori relazionali di confronto per confrontare le due **Stringhe**, e stampa i risultati
  - Per == e != eseguire anche il test con **equals()**
- Nel **main**, invocare il metodo con differenti **Stringhe**

## Operatori logici

- Gli operatori logici producono un valore **booleano**
  - Sono: AND (&&), OR (||), NOT (!)
  - Producono **true** o **false** a seconda della relazione logica esistente fra gli argomenti

## Esempio

```
public class Bool {
    public static void main(String[] args) {
        Random rand = new Random();
        int i = rand.nextInt(100);
        int j = rand.nextInt(100);
        System.out.println("i = " + i);
        System.out.println("j = " + j);
        System.out.println("i > j is " + (i > j));
        System.out.println("i < j is " + (i < j));
        System.out.println("i >= j is " + (i >= j));
        System.out.println("i <= j is " + (i <= j));
        System.out.println("i == j is " + (i == j));
        System.out.println("i != j is " + (i != j));
        /* continua */
    }
}
```

## Esempio

```
// Un intero non può essere trattato come un booleano
//! System.out.println("i && j is " + (i && j));
//! System.out.println("i || j is " + (i || j));
//! System.out.println("!i is " + !i);
System.out.println("(i < 10) && (j < 10) is " + ((i < 10) &&
(j < 10)) );
System.out.println("(i < 10) || (j < 10) is " + ((i < 10) ||
(j < 10)) );
}
} ///:~
```

## Operatori logici

- Come visto nell'esempio, gli operatori logici in Java possono essere usati *solo* con valori **boolean**
- *Non* possono essere usati con valori *non-boolean* come in C e in C++
- Nota: un valore **boolean** è convertito automaticamente in un testo appropriato se è usato dove è attesa una **Stringa**
  - Esercizio: provare a stampare a console il risultato di una operazione logica

## Operatori logici

- Nell'esempio visto, gli **int** possono essere sostituiti con gli altri tipi primitivi (eccetto **boolean**)
- Nota: il confronto fra numeri in floating-point è stretto
  - In altre parole, due numeri che differiscono per una piccolissima frazione, non saranno considerati *uguali* nei risultati dei confronti

## Operatori logici

- Le espressioni logiche sono valutate solo quando tutta l'intera espressione può essere valutata in modo non ambiguo
- Quindi, le parti di una espressione logica potrebbero non essere valutate
- Vediamo un esempio

## Esempio

Scrivete l'esempio e fatelo girare....

```
public class ShortCircuit {
    static boolean test1(int val) {
        System.out.println("test1(" + val + ")");
        System.out.println("result: " + (val < 1));
        return val < 1;
    }
    static boolean test2(int val) {
        System.out.println("test2(" + val + ")");
        System.out.println("result: " + (val < 2));
        return val < 2;
    }
}

/* continua */
```

## Esempio

Scrivete l'esempio e fatelo girare....

```
static boolean test3(int val) {
    System.out.println("test3(" + val + ")");
    System.out.println("result: " + (val < 3));
    return val < 3;
}

public static void main(String[] args) {
    if(test1(0) && test2(2) && test3(2))
        System.out.println("expression is true");
    else
        System.out.println("expression is false");
}

} //::~~
```

## Esempio

- Cosa notate nei risultati?
- Il secondo test è **false**, e quindi il terzo non viene fatto!!
  - *Short-circuiting*

## Operatori bitwise

- Questi operatori permettono di effettuare confronti fra coppie di bit (non sono molto usati)
  - AND (&) restituisce 1 se entrambi i bit sono uno, altrimenti 0
  - OR (|) restituisce 1 se almeno un bit è uno, e 0 se entrambi sono 0
  - XOR (^) restituisce 1 se uno solo dei due bit è 1, ma non entrambi
  - NOT (~) è un operatore *unario*: restituisce il complemento del bit in input
- Possono essere combinati con = per unirli con l'assegnamento
  - &=, |=, ^=

## Operatori di shift

- Anch'essi servono a manipolare bit
- $a \ll b$  ( $a \gg b$ ): effettua uno shift a sinistra (destra) di  $a$  del numero di bit specificato da  $b$
- Viene usata l'estensione con segno
  - Se il valore è positivo, vengono inseriti 0 nei bit alti, altrimenti 1
- Java ha introdotto anche lo shift senza segno
  - Indipendentemente dal segno, vengono inseriti zeri ( $\gg$ ,  $\ll$ )
  - Non esiste in C o C++

## Esempio

```
public class URShift {  
    public static void main(String[] args) {  
        int i = -1;  
        System.out.println(i >>= 10);  
        long l = -1;  
        System.out.println(l >>= 10);  
        short s = -1;  
        System.out.println(s >>= 10);  
        byte b = -1;  
        System.out.println(b >>= 10);  
        b = -1;  
        System.out.println(b >>= 10);  
    }  
} //~
```

## Esempio generale

- Vediamo un esempio che mostra gli operatori bitwise e di shift
- BitManipulation.java

## Operatore if-else

- Non stiamo parlando del comando di if-then-else
- Questo è un operatore ternario che produce un valore  
 $\text{exp-boolean} ? \text{valore0} : \text{valore1}$
- Se l'espressione **booleana** è **true**, viene restituito **valore0**, altrimenti **valore1**

## Operatore if-else

- In alternativa si può usare il comando if-then-else, ma l'operatore produce codice più compatto

```
i<10 ? i*100 : i*10
```

Contro

```
if (i<10)
    return i*100
else
    return i*10
```

## Operatore per stringhe +

- Serve per concatenare **String**he
- Se una espressione inizia con **String**, allora tutti i suoi operandi che seguono devono essere **String**he

```
int x=0, y=1, z=2;
String sString = "x, y, z ";
System.out.println(sString + x + y + z);
```
- Il compilatore Java converte **x**, **y**, e **z** nella loro rappresentazione **String**, invece di addizzionarli

## Operatore di cast

- Serve per cambiare un tipo in un altro (quando necessario)
  - Ad esempio, quando estraiamo elementi da una collezione

```
Void casts() {
    int i = 200;
    long i = (long)i;
    long i2 = (long)200;
}
```
- Come si vede, è possibile fare il cast sia su una variabile che su un valore numerico

## Letterali

- Per specificare al compilatore che vogliamo rappresentare i dati in una forma particolare, bisogna usare delle lettere specifiche
- Letterali vengono usati anche per rappresentare esponenti

## Letterali

- Esadecimale: inizia con 0x, seguito da 0-9 e a-f
- Ottale: inizia con 0 e poi cifre 0-7
- Float: termina con f o F
- Long: termina con l o L
- Double: termina con d o D
- e: significa "10 alla"
  - 1e-47 significa  $1 \times 10^{-47}$
  - Esponenti sono trattati tipicamente come **double**; quindi se li vogliamo **float** dobbiamo fare un *cast*

## Esempio

- Come esempio, leggiamo il codice di **Literals.java**
- Nel testo (Thinking in Java) viene riportato un interessante esempio (**AllOps.java**) che utilizza tutti gli operatori visti

## Controllo dell'esecuzione

- Viene effettuato con i tipici comandi del C
  - **if-else**
  - **while**
  - **do-while**
  - **for**
  - **switch**
- Ricordiamo che Java non permette l'utilizzo dei numeri come **booleani** (come in C)
  - Si possono usare solo **booleani** nelle guardie dei comandi per il controllo dell'esecuzione

## if-else

- Assume le due seguenti forme

```
if(exp-boolean)
    comando

if(exp-boolean)
    comando
else
    comando
```

## Esercizio

---

- Scrivere una classe **IfElse** che abbia
  - Un metodo **test()** che prende due argomenti interi, e
    - Restituisce 1 se il primo è maggiore del secondo
    - Restituisce -1 se il primo è minore del secondo
    - Restituisce 0 se sono uguali
  - Scrivere il **main** che stampa a console il risultato di **test()** invocato su 3 coppie di interi che riproducano i 3 possibili risultati

## return

---

- Specifica che quale valore restituisce un metodo
  - Restituisce quel valore immediatamente
    - Quindi è possibile avere in return **espressioni** da restituire
- ```
return a*b+c
```
- Vedi esempio IfElse2 (da confrontare con IfElse)

## Iterazione -- while

---

```
while(exp-boolean)  
    comando
```

- Esercizio: scrivere una classe **WhileTest** che contenga il **main** in cui vengono generati numeri casuali fino a che una particolare condizione non è verificata
  - Generare i numeri casuali invocando il metodo statico **random()** in **Math**

## Iterazione -- do-while

---

```
do  
    comando  
while(exp-boolean)
```

- L'unica differenza con il while, è che il comando viene eseguito sempre almeno una volta

## Iterazione -- for

`for(init; exp-boolean; passo)`  
comando

- **init**, **exp-boolean** e **passo** possono essere vuote
- Esercizio: scrivere una classe **ListCharacters** con un **main** che stampa i valori dei primi 128 caratteri solo se sono minuscoli
  - Il **for** itera sui primi 128 **int**
  - Utilizzare il metodo statico **isLowerCase()** in **Character**

## Esercizio -- ListCharacter

- Notare che nella soluzione la variabile **i** è dichiarata nel **for**
  - Lo scope di **i** è delimitato dal **for**
- È possibile definire più variabili nel **for**, ma devono essere dello stesso tipo

`for (int i=0, j=1; i<10 && j != 11; i++, j++)`

- Notare l'utilizzo dell'operatore **,**

## Iterazione -- break e continue

- Nel corpo di ogni comando di iterazione è possibile controllare il flusso dell'iterazione con **break** e **continue**
  - **break** esce dal ciclo senza eseguire il resto del comando nel ciclo
  - **continue** interrompe il ciclo corrente e inizia il successivo

## Cicli infiniti

- Si possono ottenere in due modi

`while(true)`

e

`for(;;)`



## Java e il goto

- In Java il **goto** non esiste
- Ma esiste qualcosa che gli assomiglia
  - Le etichette: identificatori seguiti da due punti
    - Ad esempio **etichetta**:
  - L'unico posto sensato per avere etichette è esattamente *prima* di una iterazione
    - Non è bene inserire comandi fra l'etichetta e l'inizio dell'iterazione

## Etichette

```
label1:  
iterazione-esterna {  
    iterazione-interna {  
        ....  
        break;  
        ....  
        continue;  
        ....  
        continue label1;  
        ....  
        break label1;  
    }  
}
```

## Etichette

```
label1:  
iterazione-esterna {  
    iterazione-interna {  
        ....  
        break;  
        ....  
        continue;  
        ....  
        continue label1;  
        ....  
        break label1;  
    }  
}
```

Interrompe l'iterazione-interna e passa all'iterazione-esterna

## Etichette

```
label1:  
iterazione-esterna {  
    iterazione-interna {  
        ....  
        break;  
        ....  
        continue;  
        ....  
        continue label1;  
        ....  
        break label1;  
    }  
}
```

Interrompe il ciclo corrente e prosegue al prossimo ciclo dell'iterazione-interna

## Etichette

```
label1:
iterazione-esterna {
    iterazione-interna {
        ....
        break;
        ....
        continue;
        ....
        continue label1;
        ....
        break label1;
    }
}
```

Interrompe l'iterazione-interna e quella esterna, e va a **label1**

Si continua l'iterazione, ma da quella esterna

## Etichette

```
label1:
iterazione-esterna {
    iterazione-interna {
        ....
        break;
        ....
        continue;
        ....
        continue label1;
        ....
        break label1;
    }
}
```

Interrompe l'iterazione-interna e quella esterna, e va a **label1**

Non viene continuata alcuna l'iterazione

In pratica si salta alla fine dell'iterazione etichettata

## Esempio etichette

- LabeledFor.java: esempio di uso etichette con il **for**
- Analogo è l'utilizzo delle etichette nel caso di **while**
  - Vedi esempio **LabeledWhile.java**

## Etichette

- È importante ricordare che che *l'unica* ragione per utilizzare etichette in Java è quando si hanno cicli annidati e si vuole “navigare” fra i vari cicli

## Switch

- È un comando di *selezione*
  - Seleziona fra vari pezzi di codice in base alla valutazione di un selettore intero

```
switch(selettore-intero)
case valore-intero1: comando; break
case valore-intero2: comando; break
case valore-intero3: comando; break
default: comando;
```

## Switch

È una espressione  
che produce un  
valore intero

- È un comando di *selezione*
  - Seleziona fra vari pezzi di codice in base alla valutazione di un selettore intero

```
switch(selettore-intero)
case valore-intero1: comando; break
case valore-intero2: comando; break
case valore-intero3: comando; break
default: comando;
```

## Switch

Lo **switch** confronta  
i valore-intero con il  
selettore-intero

- È un comando di *selezione*
  - Seleziona fra vari pezzi di codice in base alla valutazione di un selettore intero

```
switch(selettore-intero)
case valore-intero1: comando; break
case valore-intero2: comando; break
case valore-intero3: comando; break
default: comando;
```

## Switch

Lo **switch** confronta  
i valore-intero con il  
selettore-intero

- È un comando di *selezione*
  - Seleziona fra vari pezzi di codice in base alla valutazione di un selettore intero

```
switch(selettore-intero)
case valore-intero1: comando; break
case valore-intero2: comando; break
case valore-intero3: comando; break
default: comando;
```

Se si trova una corrispondenza,  
si esegue il relativo comando

## Switch

Lo **switch** confronta  
il valore-intero con il  
selettore-intero

- È un comando di *selezione*
  - Seleziona fra vari pezzi di codice in base alla valutazione di un selettore intero

```
switch(selettore-intero)
case valore-intero1: comando; break
case valore-intero2: comando; break
case valore-intero3: comando; break
default: comando;
```

Se non si trova alcuna corrispondenza,  
si esegue il comando di **default**

## Switch

- Nella struttura mostrata, ogni **case** termina con un **break**
  - In questo modo si salta alla fine dello **switch**
- Questo è il modo convenzionale per costruire uno **switch**
  - Il **break** non è obbligatorio
  - Se manca, viene eseguito anche il codice del successivo **case**, fino a incontrare un **break** o la fine dello **switch**

## Switch

- Come già detto, il selettore intero deve essere una espressione che valuta a **int** o **char**
- Per utilizzare selettori di tipo diverso, bisogna usare **if** annidati

## Esercizio

- Scrivere una classe **VocaliEConsonanti** con un **main** che genera 100 lettere casuali e determina se esse sono vocali o consonanti
  - Utilizzare lo **switch**
- Generare le lettere con  
`Math.random() * 26 + 'a';`
  - `Math.random()` genera un numero casuale tra 0 e 1
  - Lo si moltiplica per 26 per generare un numero casuale tra 0 e 26 (il numero delle lettere dell'alfabeto inglese)
  - Si somma un offset: il valore della 'a'

## Note all'esercizio

- **Math.random()** genera un **double**, quindi è necessario un *cast* a **char**
  - La 'a' nel calcolo di **c** è convertita automaticamente in **double** per permettere la moltiplicazione, così come il valore **26**
- Cosa fa un *cast* a **char**?
  - Cioè, se ho 29.7 e facciamo un *cast* a **char**, cosa otteniamo, 29 o 30?
  - Proviamo con un esempio: CastingNumbers.java

## CastingNumbers.java

- Dai risultati si vede che fare un *cast* da un **float** o un **double** a un valore intero *tronca* sempre il numero
- Un'altra domanda riguarda **Math.random()**: sappiamo che produce un valore tra 0 e 1
  - Ma 0 e 1 sono inclusi o no?
  - Proviamo per esercizio

## Esercizio

- Scrivere una classe **RandomBounds** con un **main** che accetta un argomento da linea di comando
  - *Lower*: genera numeri casuali con **Math.random()** fino a che non è generato 0.0
  - *Upper*: genera numeri casuali con **Math.random()** fino a che non è generato 1.0
- Provate a lanciare il programma....

## Esercizio

- Con entrambi i possibili input bisogna interrompere manualmente il programma
- Infatti le probabilità che vengano generati proprio 0.0 e 1.1 è bassissima, considerato che ci sono  $2^{62}$  differenti double tra 0 e 1
- In realtà, 0.0 è incluso tra i possibili output di **Math.random()**, mentre 1.0 no

