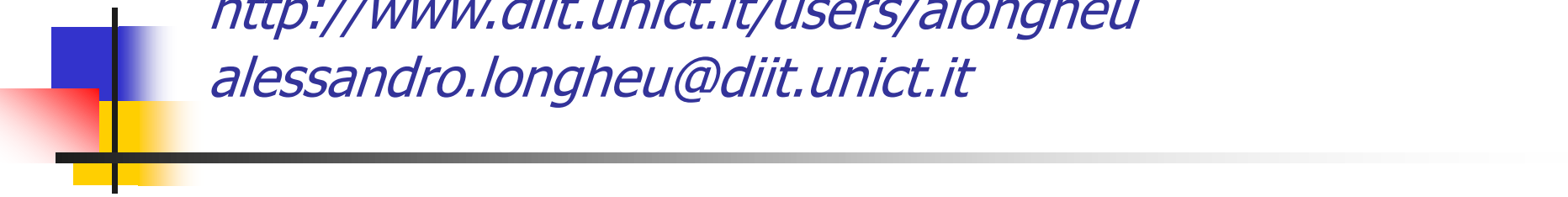
Linguaggi

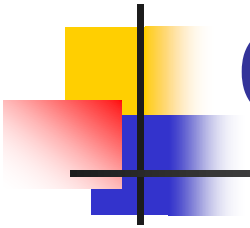
Corso di Laurea in Ingegneria delle Telecomunicazioni
A.A. 2010-2011

Alessandro Longheu
http://www.diit.unict.it/users/alongheu
alessandro.longheu@diit.unict.it



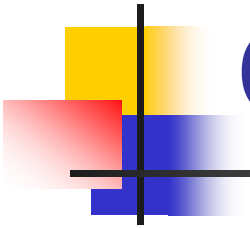
- lezione 05 -

Tipi primitivi ed Istruzioni in Java



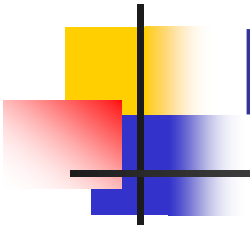
Caratteri ammessi

- Java usa un codice a 16 bit chiamato **Unicode**
- il primo **codice ASCII** è a 7 bit (ISO 646); con il bit 8, si hanno altri 128 caratteri, scelti in base alla lingua a da supportare, dando luogo agli standard **ISO 8859**; utilizzando 16 bit si possono rappresentare insiemi di caratteri fonetici e ideogrammi (indispensabile per cinese e giapponese); si ottiene il codice **UNICODE** a 16 bit; i primi 128 caratteri sono identici all'ISO 646 e i primi 256 sono gli stessi dell'ISO 8859-1; utilizzando infine 32 bit, si ottiene lo standard **ISO 10646**, che raccoglie i simboli utilizzati da tutte le lingue
- Pochi editor in realtà supportano i caratteri Unicode; è possibile utilizzare ovunque nel codice opportune **sequenze di escape** del tipo \uxxxx dove xxxx è una cifra esadecimale
- spesso occorre segnalare anche al compilatore che il sorgente è in Unicode, solitamente specificando un parametro nella linea di comando



Commenti

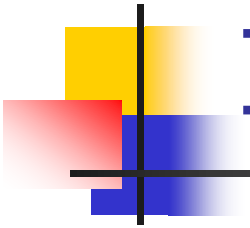
- 2 modi
  - // commento fino alla fine della linea
  - /* C-style, commento su più linee */
- Non è possibile il nesting dei commenti, ad esempio /* xxx /* zzz */ sss */ dà errore in compilazione perch il primo */ chiude il commento, lasciando al parser “sss” scoperto.



Keywords

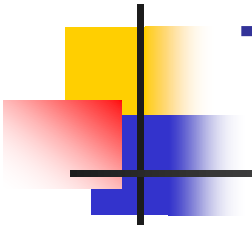
- abstract
- default
- if
- private
- this
- boolean
- do
- implements
- protected
- throw
- extends
- null*
- break
- double
- import
- public
- throws
- byte
- else
- instanceof
- return
- transient
- case
- int
- false*
- short
- try
- catch
- final
- interface
- static
- void
- char
- finally
- long
- strictfp
- volatile
- true*
- class
- float
- native
- super
- while
- const
- for
- new
- switch
- continue
- goto
- package
- synchronized

false, true e null sono valori costanti (literals), non proprio parole chiave



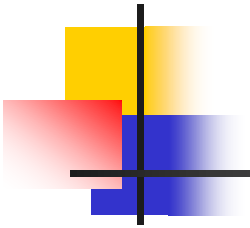
Identificatori

- Gli identificatori sono nomi di variabili e/o costanti
- Non possono essere gli stessi delle keyword
- Sono case-sensitive
- Possono avere qualsiasi lunghezza
- Si possono comporre utilizzando i seguenti caratteri:
  - Lettere e cifre in Unicode, per cui אקש, БЛЪЙЧ e ٣شج sono tutti identificatori validi; utilizzando l'unicode, alcuni caratteri sono graficamente simili se non uguali, anche se hanno codici unicode differenti; è quindi consigliabile utilizzare un alfabeto soltanto nel sorgente
  - Il carattere underscore (_) e dollaro ($)
  - Devono iniziare con una lettera



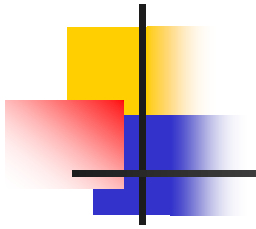
Tipi di dato

- Ogni entità che ha un valore (variabile, parametro, valore restituito da un metodo) deve avere un tipo.
- In Java ci sono due generi di tipi :
  - Tipi primitivi
  - Tipi riferimento



Tipi primitivi

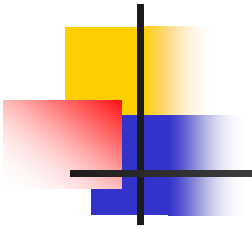
- char
- byte
- short
- int
- long
- boolean
- float
- double
- Le variabili di tipo primitivo non sono oggetti di una classe; per rientrare nel paradigma OO, sono state introdotte le **classi involucro (wrapper)**, ognuna modellante i tipi di dato primitivi; ogni oggetto di una classe wrapper ha un attributo che contiene il valore literal, e sono anche però corredati da metodi di utilità



Tipi primitivi

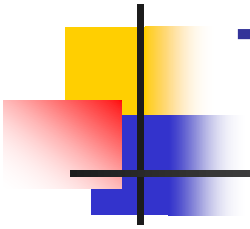
Data Type	Values	Storage (in bytes)
char	0 to 65535	2 (16 bits)
byte	–128 to 127	1 (8 bits)
short	–32768 to 32767	2 (16 bits)
int	–2147483648 to 2147483647	4 (32 bits)
long	–9223372036854775808 to 9223372036854775807	8 (64 bits)

- I tipi utilizzano tutti il segno ed operano con il complemento a due



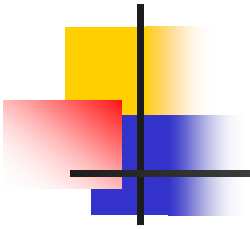
Tipi primitivi

- Boolean
  - boolean (1 bit) false e true
  - tipo autonomo totalmente disaccoppiato dagli interi: non si convertono boolean in interi e viceversa, neanche con un cast
  - le espressioni relazionali e logiche danno come risultato un boolean, non un int come in C
- float e double rappresentano numeri in virgola mobile secondo lo standard IEEE754



Tipi riferimento

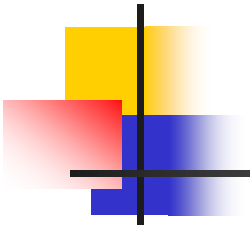
- Ogni cosa che non è un tipo primitivo è un "*reference type*".
- Ci sono tre tipi di reference types:
  - Classes
  - Arrays
  - Interfaces
- Literal
  - null



Costanti letterali

- Le **costanti letterali (literals)** sono valori costanti utilizzati in un programma.
- Esempi:

```
x = y + 3;
System.out.println("Ciao");
finished = false;
```

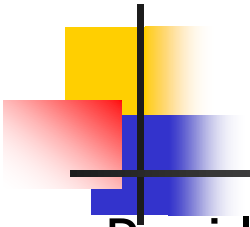


Costanti letterali

- I **caratteri** sono codificati utilizzando il codice unicode:
  - i caratteri singoli sono rappresentati come in C/C++ racchiusi da apici singoli, le stringhe invece richiedono le virgolette

    'a' 'b' 'C' '\n' '\t' "ciao" "prova"
  - Qualsiasi cosa che inizia con \u è trattata come il valore corrispondente al numero unicode

    '\u03df' '\u003f'



Costanti letterali

- Per i literals **numerici interi** (byte, short, int, long) di default si usano i numeri in base 10

  23 100 0 1234567

- Java supporta anche i numeri in base 16…

  0x23 0x1 0xaf3c21 0xAF3C21

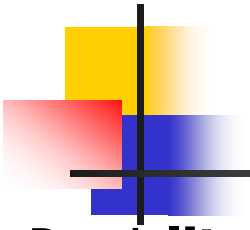
- …E in base base 8 (preceduto da 0, quindi un numero che inizia con 0 è automaticamente considerato ottale)

  023 01 0123724

- Se il literal ha il suffisso '**l**' o '**L**', è di tipo long (si usa di solito il maiuscolo per non confondere 'l' con '1')

  23L 100l 0xABCDEF01234L

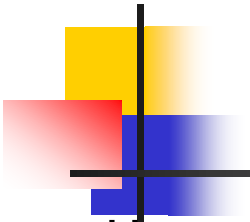
- senza la 'l' la costante è normalmente considerata int (anche se il numero fosse abbastanza piccolo da entrare in un byte o short)
- i **literal per il boolean** sono solo true e false



Costanti letterali

Per i **literals numerici in virgola mobile** (float e double) i valori sono quelli ammessi dallo standard IEEE-754:

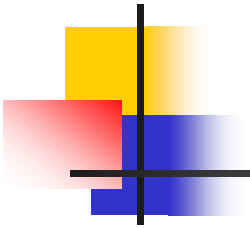
- float (4 byte) 10^{-45} ... + 10^{38}
- double (8 byte) 10^{-328} ... + 10^{308}
- Le costanti float terminano con la lettera F
  - 3.54 è una double (è possibile utilizzare l'estensione D o d)
  - 3.54F è una costante float
  - double x = 3.54; double x = 3.54F; float fx = 3.54F; sono OK
  - float fx = 3.54; è una frase illecita
- float o double possono essere espressi in base 16, in tal caso la mantissa è l'esponente di 2 (non 10), e la lettera è la 'p', ad esempio il numero 18 in base 10 può essere rappresentato come
  - 0x12p0, 0x1.2p4, 0x.12p8, 0x120p-8
- sono presenti +0.0 e -0.0, uguali per l'operatore == ma che possono fare ottenere risultati diversi quando nei calcoli
- Le classi wrapper Float e Double sono dotate delle costanti simboliche **NaN** (Not a number), usato per modellare 0/0, **POSITIVE_INFINITY** e **NEGATIVE_INFINITY** che possono rappresentare le operazioni che danno ±∞ come risultato, ad esempio 1/-0.0=-∞



Esempio 1

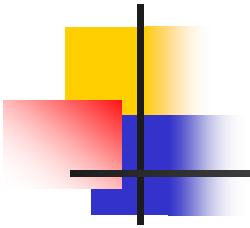
- Un programma su tre classi, tutte usate come componenti software (solo parte statica):
  - Una classe Esempio con il main
  - Le classi di sistema Math e System (Il nome di una classe (Math o System) definisce uno spazio di nomi)
- Math è la libreria matematica, e comprende solo costanti (ad esempio E o PI) e funzioni statiche (abs(), sin(), cos(), tan(), min(), max(), exp(), log(), pow()…)
- Per usare una funzione o una costante definita dentro Math o System occorre specificarne il nome completo, mediante la notazione puntata

```
public class EsempioMath {
      public static void main(String args[]){
         double x = Math.sin(Math.PI/3);
    System.out.println(x);
     }
}
```



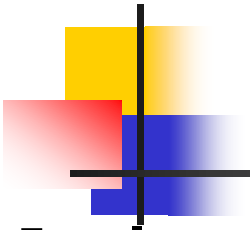
Esempio 2

- Costruire un componente software che permetta di ottenere la successione dei numeri primi (o, più in generale, il successivo numero di una sequenza)
- Progetto: poiché non esiste una formula per "produrre" i numeri primi, a ogni richiesta occorre
  - considerare il successivo numero dispari (a parte il 2)
  - controllare se è primo applicando la definizione (non ha altri divisori oltre se stesso e 1)
  - se è primo, esso costituisce il risultato; altrimenti, si deve considerare il numero dispari successivo.
- Adottiamo il **Crivello di Eratostene**
  - Specifica di I° livello: occorre provare a dividere N per tutti i numeri K: $K^2<N$: se nessuno risulta essere un divisore, allora N è primo
  - Specifica di II° livello: se N è 1, 2 o 3, allora è primo senz'altro. Altrimenti, se è un numero pari, non è primo. Se invece N è dispari e >3, occorre tentare tutti i possibili divisori da 3 in avanti, fino a $\sqrt{N}$.



Esempio 2

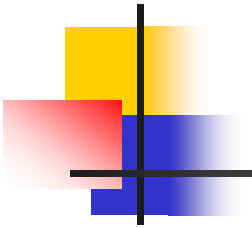
- Possibile implementazione:
  - una **variabile permanente privata lastPrime**, per mantenere lo stato (ultimo valore prodotto)
  - **una funzione privata isPrime(int p)** che, applicando l'algoritmo di Eratostene, restituisca true se il numero intero p è primo
  - **una funzione pubblica nextPrime()** che restituisca a ogni invocazione il successivo valore della sequenza.



Esempio 2

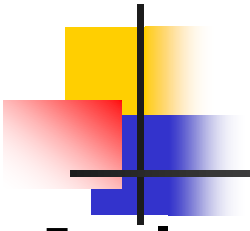
Implementazione in C:

```
#include <math.h>
static int isPrime(int n) {     /* invisibile fuori dal file */
        int max,i;
        if (n>=1 && n<=3) return true;  /* 1,2,3 → primi */
        if (n%2==0) return false;           /* numeri pari → no */
        max = sqrt(n);
        for(i=3; i<=max; i+=2) if (n%i==0) return false;
        return true;
    }
int nextPrime(void) {       /* unica funzione pubblica */
        static int lastprime = 0;          /* mantiene lo stato */
        if (lastprime>=0 && lastprime<=2) return ++lastprime;
        else {
                do { lastprime += 2; }
                while (!isPrime(lastprime));
                return lastprime;
                }
}
```



Esempio 2

- Nel caso di utilizzo di Java, non esistono visibilità associate ai moduli (la parola chiave static non ha più quel significato), piuttosto si specifica esplicitamente cosa sia privato e cosa pubblico tramite private e public
- Il modulo C viene sostituito dalla parte statica di una classe Java
  - le funzioni diventano funzioni statiche della classe
  - la variabile statica che manteneva lo stato, interna alla funzione lastprime, diventa una variabile statica della classe Java

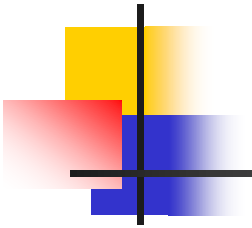


Esempio 2

Implementazione in Java:

```
public class NumeriPrimi {
    private static int lastPrime = 0;
    private static boolean isPrime(int p) {
    … l'algoritmo di verifica (Crivello di Eratostene) ...
    … identico all'implementazione in C… }
    public static int nextPrime() {
     generare un nuovo intero (dispari) e verificarlo, fino a che
     se ne trova uno per cui isPrime è true
     … identico all'implementazione in C…, a parte lastPrime,
     … già dichiarato sopra e quindi direttamente utilizzabile}
}
```

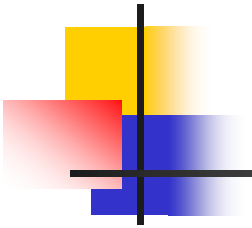
- È un puro componente software (ha solo la parte statica)
- **lastPrime** e **isPrime()** sono privati e come tali invisibili a chiunque fuori dalla classe
- La funzione **nextPrime()** è invece pubblica e come tale usabile da chiunque, dentro e fuori dalla classe



Variabili

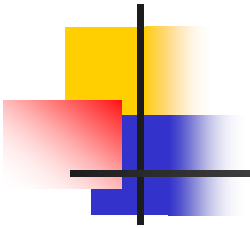
Diversi tipi di variabili:

- **Campi**, ovvero attributi di una classe o interfaccia
- **Variabili locali**, ad esempio *for(int i=0; i<10; i++) { … }*
  - possono essere dichiarate ovunque all'interno di un blocco
  - Una variabile locale può essere al più final (quindi immutabile dopo la sua inizializzazione, di solito effettuata nella dichiarazione ma non necessariamente (in tal caso sono note come blank final, e vanno comunque inizializzate prima di essere usate)
- **Parametri di un metodo**



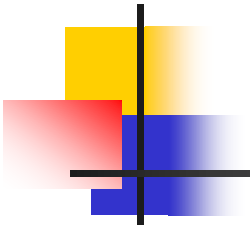
Gestione nomi

- La **gestione dei nomi delle variabili, metodi e classi** viene effettuata utilizzando gli **spazi dei nomi** (namespaces) per classificare la categoria, e lo **scoping** per controllare la visibilità del nome
- Esistono sei namespaces: package (librerie), tipi, campi, metodi, variabili locali, etichette
- La visibilità obbedisce invece alle seguenti **regole di ricerca**:
  - Variabili locali
  - Parametri del metodo (se il nome di cui si sta analizzando lo scope è dentro un metodo)
  - Campo della classe o interfaccia, compresi quelli ereditati o comunque accessibili
  - Se il nome è all'interno di un tipo innestato, si cerca nel tipo contenitore
  - Campo statico della classe o interfaccia, dichiarato in un'istruzione di importazione statica o statica su richiesta (import con *)



Operatori

- incremento/decremento
- relazionali
- aritmetici
- logici
- manipolazione di bit
- assegnazioni
- stringhe



Operatori

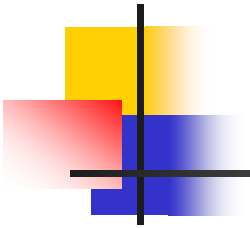
Operatori di incremento/decremento

- i++, i--, come in C
- possono essere prefissi (l'operazione è effettuata prima della valutazione) o postfissi (dopo):

```
int i=16;
System.out.println(++i + " " + i++ + " " + i);
```

  stampa 17 17 18

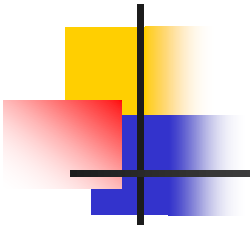
- *i [3] = i [3] +1* nessun problema
- *i [metodo()] = i [metodo()] + 1* il metodo viene invocato due volte, **se l'operazione è idempotente** va tutto bene, altrimenti potrebbero aversi risultati non attesi
- gli operatori possono essere applicati al tipo char e consentono di ottenere il precedente o successivo carattere Unicode



Operatori

Operatori relazionali

- standard >, <, >=, <=, ==, !=
- == e != sono gli unici validi per il tipo booleano
- il valore costante NaN dell'aritmetica mobile è tale per cui confrontato con qualsiasi altra cosa (compreso un altro NaN) con qualunque operatore restituisce false, tranne != che torna true, quindi
  - Double.NaN == Double.NaN è falsa
  - Double.NaN != Double.NaN è vera, e se x è un Nan, x!=x è vera
- Gli operatori == e != possono anche essere applicati ai tipi riferimento, e indicano se l'oggetto riferito è lo stesso; se gli oggetti referenziati sono distinti (in memoria, i due riferimenti puntano a zone diverse), l'operatore torna false, anche se i due oggetti sono equivalenti (stessi valori per attributi), quindi gli operatori **controllano l'identità e non l'equivalenza**



Operatori

Operatori aritmetici

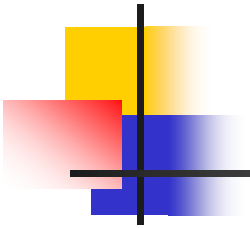
- standard +,-,*,/,%

Operatori logici

- standard &, |, !, ^ (XOR), &&, ||
- && e || sono **AND e OR condizionali**, che effettuano le stesse operazioni di & e | ma non valutano l'operando destro se la valutazione dell'operando sinistro consente di dare già il risultato dell'intera espressione (operatori cortocircuitati), così un'espressione come if (x>=0 && v[x]!=0) diventa accettabile

Operatori di manipolazione dei bit

- standard &, |, ^ (XOR), << (shift sinistro con riempimento di zeri), >> (shift destro riempendo con il MSB i bit di sinistra), >>> (come >> ma riempie di 0
- si applicano solo ai tipi interi
- sono distinti dagli omonimi logici, che si applicano ai booleani (il tipo dell'operando utilizzato determina l'effettivo operatore adottato)



Operatori

Operatore di assegnazione

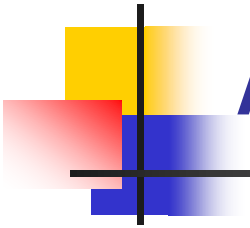
- l'operatore è il classico =, che richiede cast esplicito in caso di tipo diverso fra gli operandi

Operatore per stringhe

- l'operatore è il +, usato per il concatenamento

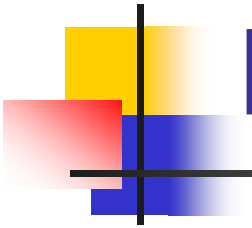
Altri operatori

- istanceof, che consente di sapere a quale classe appartiene un riferimento: *if (x istanceof ClassProva) …*
- new, che consente di creare un oggetto *Class x=new Class()*



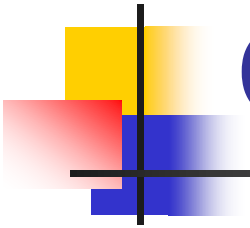
Aritmetica

- **Aritmetica intera**
  - modulare in complemento a due
  - Non genera ne overflow ne underflow ma genera riavvolgimenti (opera con modulo pari al limite ammesso per il tipo)
  - La divisione per zero genera **ArithmeticException**
  - La divisione effettua il troncamento (non l'approssimazione) dei decimali
- **Aritmetica in virgola mobile**
  - Genera overflow e underflow
  - Prevede l'uso di ±∞
- **Aritmetica in virgola mobile stretta e non stretta**
  - Uso del modificatore **strictfp** per un classe



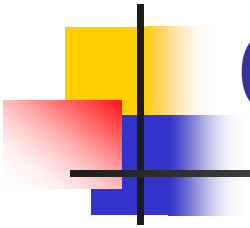
Espressioni

- Java valuta gli operatori sempre nello stesso ordine, da destra verso sinistra, e comunque prima di valutare l'espressione
- in caso di sollevamento di eccezione, la valutazione viene interrotta
- Se gli operandi sono di diverso tipo (ma compatibili fra loro, ad esempio short, int, long) viene eseguita una conversione implicita verso il tipo più esteso
- nella valutazione influisce la precedenza degli operatori
- nella valutazione influisce anche la associatività degli operatori, generalmente sinistra (a+b+c=(a+b)+c), ma talvolta anche destra (a=b=c equivale a a+(b=c))



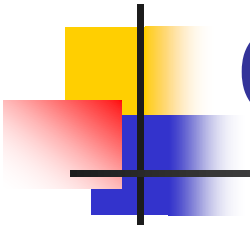
Conversioni di tipo

- Le conversioni di tipo possono essere **implicite (automatiche) o esplicite**
- Le conversioni possono inoltre essere verso un tipo che supporta un insieme di valori più esteso (**widening**) o verso un tipo con meno valori (**narrowing**)
- Esempi di conversioni automatiche:
  - short s=27 // da int a short (narrowing)
  - byte b=27 // da int a byte (narrowing)
  - short s=100000 // Errore (int troppo grande)
  - float f=27 // da int a float (widening)



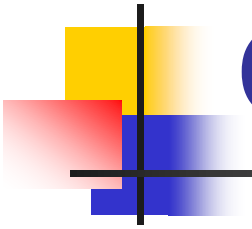
Conversioni di tipo

- Nelle conversioni automatiche è possibile una **perdita di precisione**, nonostante la verifica positiva del controllo sulla grandezza, ad esempio:
  long orig=9151314438521880576;
  float f=orig;
  long lose=(long)f;
  System.out.println(f);
  System.out.println(lose);
  - il valore lose risulta 9151314442816847872, diverso da orig perchè durante il passaggio a float si ha che f memorizza l'intero in modalità differente: f=9.1513144E18
- Le conversioni automatiche sono di **7 tipi**: conversioni widening o narrowing di tipi primitivi o riferimento (4), conversioni boxing o unboxing (2), conversioni di stringhe (1)
- Le conversioni automatiche possono avvenire in **5 contesti**: assegnazione, invocazione di metodi, promozioni numeriche (conversioni di tutti gli operandi di un'espressione), casting espliciti, concatenazione di stringhe



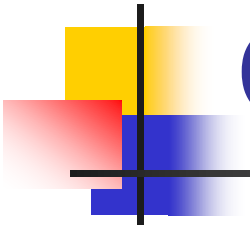
Conversioni di tipo

- Le **conversioni esplicite** sono in genere effettuate per operare un narrowing, pur potendo essere valide anche per il widening
- **Quando si converte da virgola mobile a intero** si perde la parte frazionaria, un valore NaN diventa l'intero 0, mentre valori troppo grandi o piccoli per diventare interi diventano MAX_VALUE o MIN_VALUE (costanti predefinite)
- **Quando invece si converte da double a float**, si può perdere precisione, ottenere uno zero o infinito, in tutti quei casi in cui il valore del double è fuori dal range del float
- **Quando invece si converte fra interi**, si eliminano i bit più significativi, per cui si possono avere ripercussioni sul segno, per esempio:
  - short s=-134
  - byte b=(byte)s
  - si ottiene b=122, perchè il 134 era fuori dal range del tipo byte



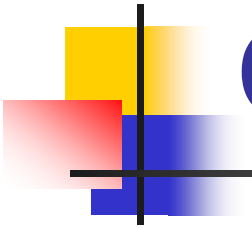
Controllo di flusso

- Le istruzioni per il controllo del flusso sono if-else, switch, while e do-while, for, break, continue, return
- il loro comportamento e sintassi sono sostanzialmente identici a quelli del C/C++



Controllo di flusso

- Nello **switch**, l'espressione di controllo può essere un tipo intero o enumerativo
- Nel **while**, il ciclo potrebbe non essere effettuato; se occorre che venga effettuato almeno una volta, si usa il do-while
- Il **for** prevede anche una **forma avanzata**:
  - for (<tipo> <variabile-del-ciclo> : <espressione>) corpo
  - L'espressione deve essere un oggetto che definisce l'insieme dei valori, ad esempio un enumerativo, un array o un qualsiasi oggetto che implementi l'interfaccia Iterable
  - il funzionamento è automatico, e prevede che la variabile assuma tutti i valori ammessi dall'espressione; tuttavia, è ammesso solo l'avanzamento e non è possibile intervenire sugli elementi dell'insieme dell'espressione



Controllo di flusso

- Il **break** può essere posto dentro un qualsiasi blocco di codice, oltre che dentro uno switch, e consente di terminare l'istruzione più interna di un for, switch, while o do, o di qualsiasi altra istruzione purchè etichettata
- il **continue** invece può solo stare dentro un ciclo (while, do, for) e consente di interrompe l'esecuzione del corpo del ciclo, trasferendo il controllo al ciclo stesso; se si utilizzano le etichette, si può trasferire il controllo all'istruzione etichettata presente dentro il ciclo
- break e continue sono **un'alternativa al goto** che limita i salti; sono ammesse per consentire forme più eleganti e meno contorte di quelle che si dovrebbero implementare in loro assenza

Controllo di flusso

```
for (int i=0;i<10;i++) {
   System.out.println("i is " + i);
   if (i==3) break;
}

outer: for (int j=0;j<5;j++) {
   for (int k=0;k<5;k++) {
     if (k==3) break outer;
     System.out.println("j,k: " + j + "," + k);
   }
}
```