Il Linguaggio Java

Concetti e costrutti di base

Hello world

```
/**
 * La classe Ciao implementa un'applicazione che
 * semplicemente scrive "Ciao!" sullo standard
 * output
 */
public class Ciao {
  public static void main(String[] args) {
    System.out.println("Ciao!"); //Display the string.
   }
}
```

Commenti

Java ha tre tipi di commenti, ognuno specificato da un diverso limitatore:

- **/** documentation */** (doc comment)
- **// text**
- **/* text */**

La classe

```
public class Ciao {
  public static void main(String[] args) {
    System.out.println("Ciao!");
  }
}
```

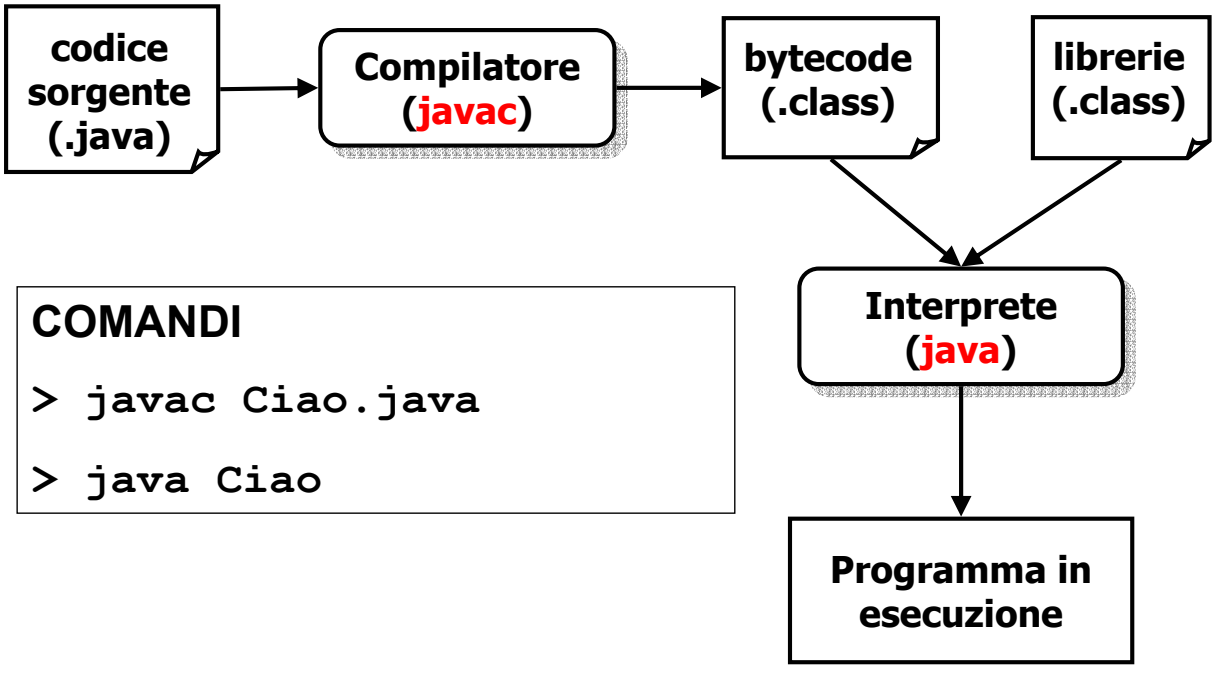
- In Java non esistono funzioni e variabili globali: ogni metodo o variabile è definito all' interno di una classe
- Lo scheletro di un programma Java consiste nella definizione di una classe

Il metodo main

```
public class Ciao {
  public static void main(String[] args) {
    System.out.println("Ciao!");
  }
}
```

- Il metodo **main** è l'entry point di un'applicazione Java
- Il metodo **main** è caratterizzato da tre modificatori
  - **public**: il metodo main può essere invocato (*è visibile*) dall'esterno
  - **static**: il metodo main è un metodo di classe
  - **void**: il metodo main non ha valore di ritorno

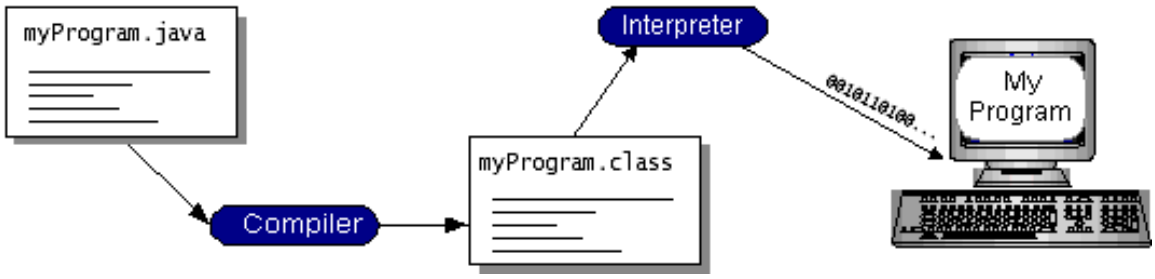
Compilazione ed esecuzione



COMANDI

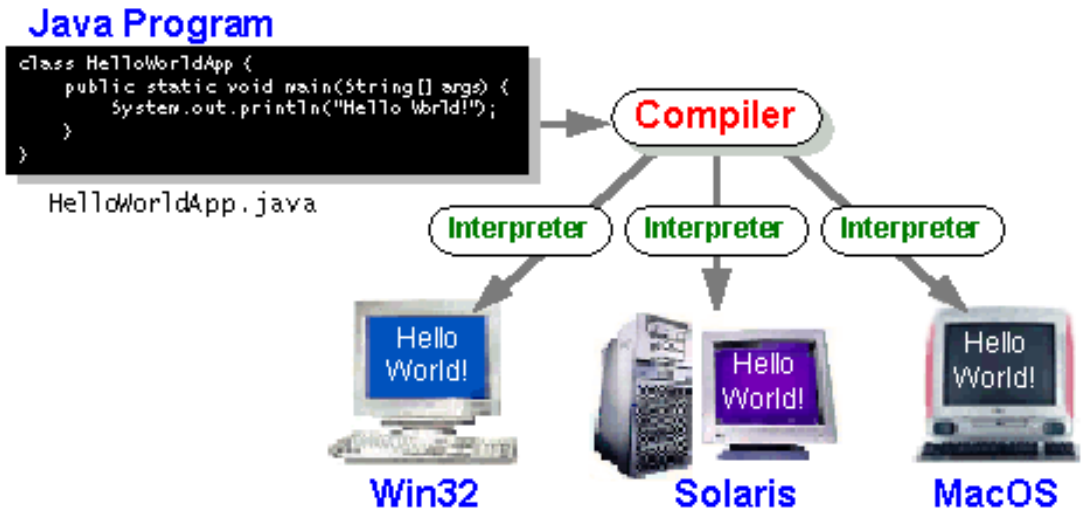
```
> javac Ciao.java

> java Ciao
```

Java Virtual Machine



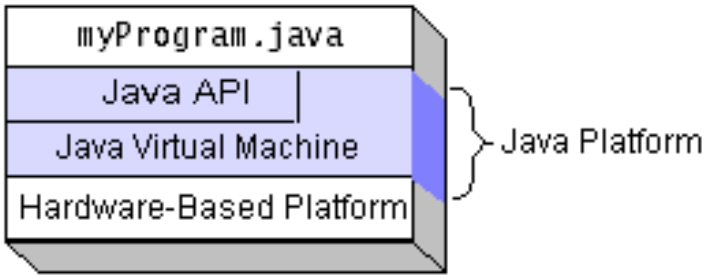
- Il **bytecode** è un linguaggio intermedio indipendente dalla piattaforma che viene interpretato dall'**interprete** Java
- Il bytecode è il linguaggio macchina della **Java Virtual Machine** (**JVM**)
- L'interprete implementa la JVM

Write once, run anywhere



write once, run anywhere

La piattaforma Java



- La piattaforma Java è solo SW ed
- è costituita da
  - JVM
  - Java API (packages)

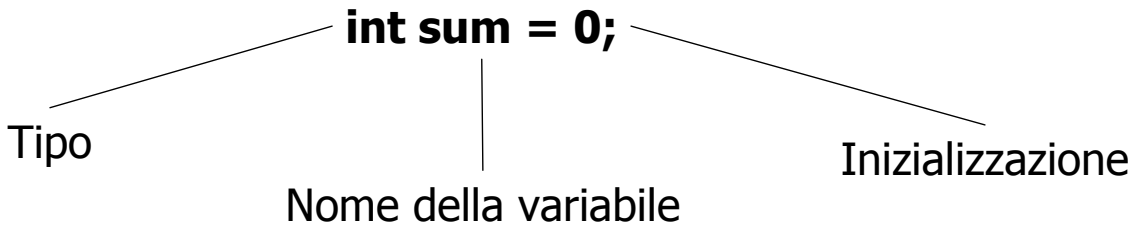
La classe BasicDemo

```
public class BasicsDemo {
    public static void main(String[] args) {
        int sum = 0;
        for (int current = 1; current <= 10; current++) {
            sum += current;
        }
        System.out.println("Sum = " + sum);
    }
}
```

- variabili
- operatori
- statement

Variabili

DICHIARAZIONE DI UNA VARIABILE



- Il nome di una variabile deve essere un **identificatore** Java, cioè una sequenza, possibilmente illimitata, di caratteri Unicode che inizia con una lettera
- Ogni variabile deve essere di un tipo dichiarato (tipizzazione forte)

Identificatori

Identifier:
IdentifierChars but not a *Keyword* or *BooleanLiteral* or *NullLiteral*

IdentifierChars:
JavaLetter
IdentifierChars JavaLetterOrDigit

JavaLetter:
any Unicode character that is a Java letter

JavaLetterOrDigit:
any Unicode character that is a Java letter-or-digit

ESEMPI: pippo, **i3**, $\alpha\beta\gamma$, **MAX_VALUE**

Identificatori

- **Java letter include**
  - Lettere maiuscole della codifica ASCII Latin: A–Z (\u0041–\u005a)
  - Lettere minuscole della codifica ASCII Latin: a–z (\u0061–\u007a)
  - l'ASCII underscore: _ (\u005f)
  - Il segno di dollaro: **$** (\u0024)
- **Java digit**
  - digits ASCII: 0-9 (\u0030–\u0039).

Nomi delle variabili

```
public class BasicsDemo {
    public static void main(String[] args) {
        int sum = 0;
        for (int current = 1; current <= 10; current++) {
            sum += current;
        }
        System.out.println("Sum = " + sum);
    }
}
```

nome qualificato (*qualified name*)

nome semplice (*simple name*)

Tipi di dato

Type:
PrimitiveType
ReferenceType

PrimitiveType:
NumericType
boolean

NumericType:
IntegralType
FloatingPointType

IntegralType: one of
byte short int long char

FloatingPointType: one of
float double

Tipi Primitivi

KEYWORD	Descrizione	Size/Format
	integer	
byte	byte-length integer	8 bit, compl. a due
short	short integer	16 bit, compl. a due
int	integer	32 bit, compl. a due
long	long integer	64 bit, compl. a due
	real	
float	Single precision floating point	32 bit, IEEE754
double	Double precision floating point	64 bit, IEEE754
	altri tipi	
boolean	valore booleano (**true**, **false**)	true, false
char	carattere	16 bit, Unicode

Tipi Primitivi

- Java definisce la dimensione dei tipi primitivi
- Java non ha l'operatore **sizeof**
- I tipi numerici sono tutti con segno
- Il tipo **boolean** ha due valori: **true** e **false**

Interi: letterali

Letterale	Tipo
178	int
0123	int (octal)
0x1FA	int (hex)
8864L	long
0123L	long (octal)
0x1FAL	long (hex)

Letterale	Tipo
37.266	double
37.266D	double
87.363F	float
26.77e3	double
' c '	char
true	boolean
false	boolean

Floating Point

- L'intervallo di rappresentabilità dei **float** è ±3.40282347E+38F, con 6–7 cifre decimali significative
- L'intervallo di rappresentabilità dei **double** è ±1.79769313486231570E+308, con 15 cifre decimali significative
- In accordo alla specifica IEEE 754 ci sono tre costanti
  - **Double.POSITIVE_INFINITY**
  - **Double.NEGATIVE_INFINITY**
  - **Double.NaN**

Floating point: portabilità

- **Il problema:** Java usa 64 bit per memorizzare un double ma alcuni processori utilizzano registri floating-point a 80 bit
- DUE APPROCCI ALTERNATIVI:
- **Sfruttare le risorse hw per le operazioni intermedie**
  - Vantaggi: maggiore precisione ed assenza di overflow
  - Svantaggi: i risultati sono diversi se si lavora sempre su 64 bit (problema di portabilità)
- **Troncare tutte le operazioni, anche quelle intermedie, a 64 bit**
  - Vantaggi: riproducibilità dei risultati
  - Svantaggi: minore precisione, overflow, prestazioni ridotte (il troncamento richiede tempo!)

Floating-point: portabilità

Approccio di Java

- Per default, le computazioni possono sfruttare la precisione estesa per le computazioni intermedie
- Se si vogliono risultati riproducibili allora si deve esplicitamente dichiarare di voler lavorare in ***strict floating-point*** (64 bit) anche nelle operazioni intermedie
- Esempio:
  public static strictfp void main(String[] args)

Operatori aritmetici

Gli operandi aritmetici possono essere usati solo con operandi numerici(*)

Operatore	Uso	Descrizione
+	**op1 + op2**	Somma **op1** e **op2**
-	**op1 - op2**	Sottrae **op2** da **op1**
*****	**op1 * op2**	Moltiplica **op1** per **op2**
/	**op1 / op2**	Divide **op1** per **op2**
%	**op1 % op2**	Calcola il resto della divisione di op1 per op2

(*) l'operatore + permette anche di concatenare le stringhe

Operatori aritmetici

Operatore	Uso	Descrizione
++	**op++**	Valuta **op** e lo incrementa di 1
++	**++op**	Incrementa **op** di 1 e poi lo valuta
--	**op--**	Valuta **op** e lo decrementa di 1
--	**--op**	Decrementa **op** di 1 e poi lo valuta
+	**+op**	Se **op** è **byte**, **short**, o **char** viene promosso a **int**
-	**-op**	Opposto di **op**

Operatori di relazione

Operatore	Uso	Ritorna true se
>	**op1 > op2**	**op1** è maggiore di **op2**
>=	**op1 >= op2**	**op1** è maggiore o uguale di **op2**
<	**op1 < op2**	**op1** è minore di **op2**
<=	**op1 <= op2**	**op1** è minore o uguale di **op2**
==	**op1 == op2**	**op1** e **op2** sono uguali
!=	**op1 != op2**	**op1** e **op2** non sono uguali

Operatori logici

Operatore	Uso	Ritorna true se
&&	**op1 && op2**	**op1** e **op2** sono entrambi **true**, in tal caso valuta **op2**
||	**op1 || op2**	o **op1** o **op2** è **true**, valuta **op2** se **op1** è **false**
!	**! op**	**op** è **false**
&	**op1 & op2**	**op1** e **op2** sono entrambi **true**, valuta sempre **op1** ed **op2**
|	**op1 | op2**	o **op1** o **op2** è **true**, valuta sempre **op1** ed **op2**
^	**op1 ^ op2**	se **op1** e **op2** sono diversi

Operatori shift e bitwise

Operator	Use	Operation
	shift	
>>	**op1 >> op2**	trasla i bit di **op1** a destra di **op2**
<<	**op1 << op2**	trasla i bit di **op1** a sinistra di **op2**
>>>	**op1 >>> op2**	trasla i bit di **op1** a destra di **op2** (unsigned)
	bitwise	
&	**op1 & op2**	and
|	**op1 | op2**	or
^	**op1 ^ op2**	xor
~	**~op2**	complement

Operatori di assegnamento (scorciatoie)

Operatore	Uso	Equivalente a
+=	op1 += op2	op1 = op1 + op2
-=	op1 -= op2	op1 = op1 - op2
*=	op1 *= op2	op1 = op1 * op2
/=	op1 /= op2	op1 = op1 / op2
%=	op1 %= op2	op1 = op1 % op2
&=	op1 &= op2	op1 = op1 & op2
|=	op1 |= op2	op1 = op1 | op2
^=	op1 ^= op2	op1 = op1 ^ op2
<<=	op1 <<= op2	op1 = op1 << op2
>>=	op1 >>= op2	op1 = op1 >> op2
>>>=	op1 >>>= op2	op1 = op1 >>> op2

Altri operatori

Operator	Use	Description
? :	op1 ? op2 : op3	Se op1 è **true**, ritorna **op2**. Altrimenti, ritorna **op3**.
[]	type []	Dichiara un array di tipo **type**
[]	type[op1]	Crea un array con **op1** elementi
[]	op1[op2]	Accede l'elemento di indice **op2** dell'array **op1**
.	op1.op2	È un riferimento al membro **op2** di **op1**
()	op1(params)	Dichiara o chiama il metodo **op1** con la lista di parametri **params**
(type)	(type) op1	Converts **op1** al tipo **type**
new	new op1	Crea un nuovo oggetto o un array
instanceof	op1 instanceof op2	Ritorna **true** se **op1** è un'istanza di **op2**

Nomi delle variabili

Il nome semplice di una variabile deve rispettare le seguenti regole:

- Deve essere un identificatore
- Deve essere diverso da una keyword, dai letterali booleani **true**, **false** e dalla parola riservata **null**
- Deve essere unico all'interno del suo ***scope***

I nomi delle variabili rispettano le seguenti convenzioni:

- Iniziano per lettera minuscola
- Se il nome è costituto da più parole, le parole sono concatenate ed iniziano, eccetto la prima, per lettera maiuscola (Es. isReady)

Variabili locali finali

```
final int aFinalVarr = 0;
```

- Il valore di una variabile **final** non può essere cambiato dopo che è stato inizializzato
- Una variabile **final** è l'equivalente delle costanti in altri linguaggi di programmazione
- L'inizializzazione di una variabile locale final può essere differito

```
final int aFinalVar;
…
aFinalVar = 0;
```

Conversioni di tipo

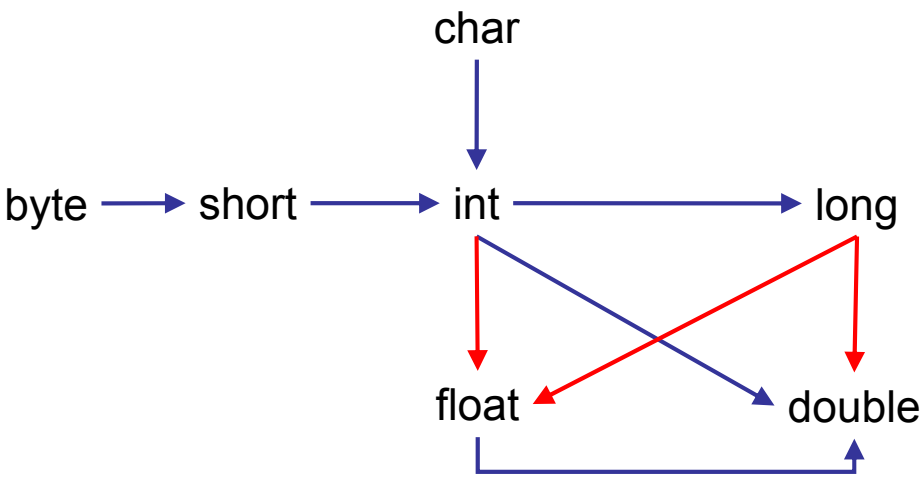
Contesti di conversione

- Assegnamento
- Invocazione di metodi
- Cast
- Conversione a stringa
- Promozione numerica

Categorie di conversione:

- Conversione identità
- Estensione e Riduzione di un tipo primitivo
- Estensione e Riduzione di un tipo riferimento
- Conversione a stringa
- Conversione di tipo Value set

Estensione/riduzione di un tipo primitivo



senza perdita di informazione

con possibile perdita di precisione

Assegnamento (I)

variabile = espressione

- Il tipo dell'espressione viene convertito al tipo della variabile
- Categorie di conversione permesse: identità ed estensione
- Se il tipo dell'espressione
  - non può essere convertito al tipo della variabile allora si ha un errore di compilazione
  - può essere convertito, allora l'espressione è *assegnabile* alla variabile

Assegnamento (II)

- Un assegnamento può dare luogo anche ad una conversione per riduzione se
  - l'espressione è un'*espressione costante* di tipo **byte**, **short**, **char** o **int**
  - il tipo della variabile è **byte**, **short** o **char**.
  - il valore dell'espressione(*) è rappresentabile nel tipo della variabile.

(*) Che è noto a compile-time perché l'espressione è costante

Conversione a stringa

```
int r = 1;
println("Risultato" + r);
```

- La conversione a stringa viene applicata agli operandi dell'operatore **+** quando uno di questi due è una stringa
  - l'altro operando viene convertito a stringa e
  - viene costruita una nuova stringa che é la concatenazione delle due stringhe;
  - i caratteri dell'operando di sinistra precedono quelli dell'operando di destra
- Ogni tipo può essere convertito a stringa

Operatore cast

(*TipoPrimitivo*)*Espressione*

- Il tipo dell'espressione viene convertito al tipo esplicitamente indicato
- Il cast permette di convertire un tipo primitivo in un altro tipo primitivo
- Categorie di conversione permesse: **conversioni identità**, **conversioni per estensione** e **conversioni per riduzione**
- Non tutti i cast sono permessi (alcuni cast possono produrre un errore a compile-time).
  - i cast **boolean** ← *NumericType* e *NumericType* ← **boolean** non sono permessi

Promozioni numeriche (I)

- Sono applicate agli operandi degli operatori aritmetici.
- Sono usate per convertire gli operandi ad un tipo comune in modo da poter eseguire l'operazione
- Permettono le categorie di conversione **identità** ed **estensione**
- Sono di due tipi
  - Promozioni numeriche unarie
  - Promozioni numeriche binarie

Promozioni numeriche unarie

- La promozione numerica unaria viene applicata a
  - l'operando della selezione per indice (array) **[]**
  - l'operando dell'operatore unario più **+**, **–** e **~** (complemento bit a bit)
  - ciascun operando, *separatamente*, degli operatori di traslazione **>>**, **<<** e **>>>**
    (*il tipo dell'espressione è dato dal tipo promosso dell'operando di sinistra*)

  secondo le seguenti regole:
  - se l'operando è **byte**, **short** o **char**, esso viene promosso a **int** per mezzo di una conversione per estensione
  - altrimenti l'operando non viene modificato

Promozioni numeriche binarie

- La promozione numerica binaria viene applicata agli operandi degli (altri) operatori aritmetici, compreso l'operatore condizionale **? :** , secondo le seguenti regole
  - se un operando è **double**, l'altro operando è convertito a **double**; altrimenti
  - se un operando è **float** l'altro viene convertito a **float**; altrimenti
  - se un operando è **long**, l'altro viene convertito a **long**; altrimenti
  - entrambi gli operandi sono convertiti a **int**

Statement

Block
EmptyStatement
ExpressionStatement
SwitchStatement
DoStatement
BreakStatement
ContinueStatement
ReturnStatement
LabeledStatement
IfThenStatement
IfThenElseStatement
WhileStatement
ForStatement
SynchronizedStatement
ThrowStatement
TryStatement

Blocco

Block:

{ *BlockStatements*$_{opt}$ **}**

BlockStatements:

BlockStatement

BlockStatements BlockStatement

BlockStatement:

LocalVariableDeclarationStatement

ClassDeclaration

Statement

Controllo del flusso

TIPO DI ISTRUZIONE	**KEYWORD**
ripetitive	**while**, **do-while**, **for**
condizionali	**if-else**, **switch-case**
gestione eccezioni	**try-catch-finally**, **throw**
salto	**break**, **continue**, **label:**, **return**

Variabili locali

LocalVariableDeclarationStatement:
    LocalVariableDeclaration **;**
LocalVariableDeclaration:
    final$_{opt}$ *Type VariableDeclarators*
VariableDeclarators:
    VariableDeclarator
    VariableDeclarators **,** *VariableDeclarator*
VariableDeclarator:
    VariableDeclaratorId
    VariableDeclaratorId **=** *VariableInitializer*
VariableDeclaratorId:
    Identifier
    VariableDeclaratorId **[ ]**
VariableInitializer:
    Expression
    ArrayInitializer

Scope di una variabile

Lo **scope** di una variabile

- definisce la regione di programma nell'ambito della quale la variabile può essere riferita per mezzo del suo **nome semplice**
- determina il momento in cui il sistema alloca e dealloca, rispettivamente, memoria alla variabile

Scope di una variabile locale