
Tipi di dato, variabili ed espressioni

Emilio Di Giacomo e Walter Didimo

Richiami generali

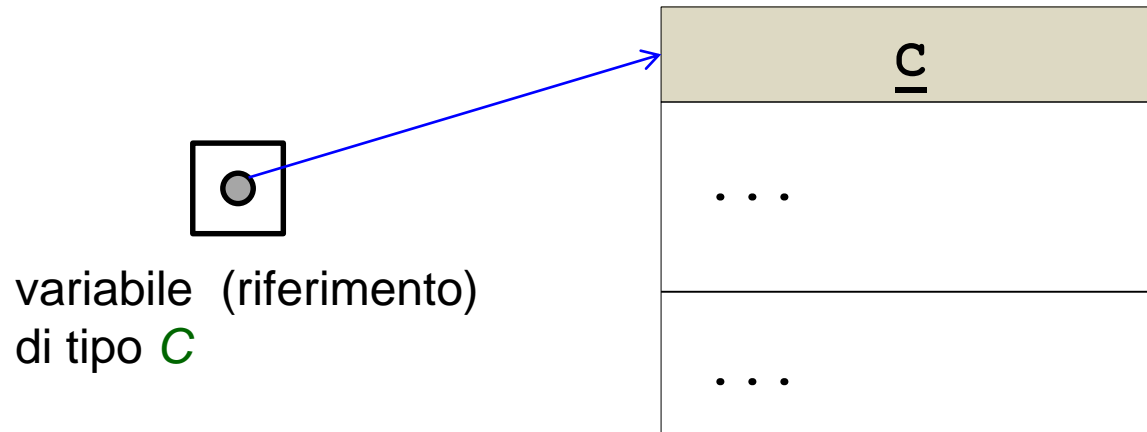
- tipo di dato: specifico dominio di valori; le due principali categorie in Java sono:
 - tipi primitivi: dati di tipo numerico, caratteri, boolean
 - tipi riferimento: riferimenti ad oggetti
- variabile: contenitore che può memorizzare valori di uno specifico tipo di dato, detto tipo della variabile
- espressione: costrutto del linguaggio al quale rimane associato un valore di uno specifico tipo di dato, detto tipo dell'espressione

Tipi riferimento

- I tipi riferimento sono associati alle classi:
 - ovunque venga definita una classe C , resta automaticamente definito il tipo riferimento C
 - l'insieme dei valori di tipo C sono i possibili riferimenti ad oggetti di tipo C
- Ad esempio, definendo una classe di nome *Rettangolo* , definisco implicitamente il tipo riferimento *Rettangolo*
 - i valori di tipo *Rettangolo* sono i riferimenti ad oggetti della classe *Rettangolo*

Variabili riferimento

- Una variabile (riferimento) di tipo **C** può memorizzare soltanto riferimenti ad oggetti della classe **C**



- Sintassi per definire una variabile riferimento di tipo **C**

C <nome variabile>

Il letterale null

- In realtà una variabile riferimento può anche memorizzare il valore *null*
 - *null* è una parola chiave del linguaggio Java, che indica un valore riferimento costante (letterale), il valore nullo
 - quando una variabile riferimento non riferenzia alcun oggetto, può essere utile assegnarle il valore *null*

Espressioni riferimento

- Ad una espressione riferimento di tipo C rimane associato un riferimento di tipo C
 - le variabili di tipo C e il letterale *null* sono espressioni elementari di tipo C
 - Altre espressioni riferimento di tipo C sono:
 - istruzioni di creazione di oggetti di tipo C
 - invocazioni di metodi che restituiscono un riferimento di tipo C

Assegnazioni a variabili

- Ad una variabile *var* di tipo *C* si può assegnare il valore di una espressione *<esp>* di tipo *C*:

var = <esp>

- La stessa istruzione *var = <esp>* è una espressione di tipo *C*, il cui valore associato è lo stesso memorizzato in *var*

Invocazione di metodi

- Se *<esp>* è una espressione di tipo *C*, si può invocare un metodo di istanza di *C* con la sintassi:

<esp>.<nome metodo> (<parametri>)

- Se *<esp>* ha valore *null* viene generato un errore di esecuzione chiamato: *NullPointerException*

Tipi primitivi

- I tipi primitivi in Java sono un insieme predefinito e non estendibile, e modellano tipi di dati particolarmente frequenti in ogni programma

TIPO PRIMITIVO	DOMINIO VALORI	NUMERO BIT
byte	numeri interi nell'intervallo $[-2^7, 2^7 - 1]$	8
short	numeri interi nell'intervallo $[-2^{15}, 2^{15} - 1]$	16
int	numeri interi nell'intervallo $[-2^{31}, 2^{31} - 1]$	32
long	numeri interi nell'intervallo $[-2^{63}, 2^{63} - 1]$	64
float	numeri reali tra $\approx 1,4 \cdot 10^{-45}$ e $\approx 3,40 \cdot 10^{38}$	32
double	numeri reali tra $\approx 4,9 \cdot 10^{-324}$ e $\approx 1,80 \cdot 10^{308}$	64
char	caratteri dell'alfabeto Unicode	16
boolean	{true, false}	8

Definizione di variabili primitive

- Le variabili primitive si definiscono allo stesso modo di quelle riferimento; ad esempio, per definire una variabile di tipo *long* e di nome *interoGrande*, basta scrivere:

long interoGrande;

- Lo spazio di memoria riservato ad una variabile primitiva è utilizzato direttamente per memorizzarvi i valori

Espressioni intere e letterali

- Le espressioni intere hanno valori di tipo *byte*, *short*, *int* o *long*.
- Le più semplici espressioni intere sono i letterali interi (costanti intere)
 - ogni numero in base 10 viene automaticamente considerato come una espressione di tipo *int*
 - se però un letterale è assegnato ad una variabile *byte* o *short* che può effettivamente contenerlo, allora è considerato espressione di tipo *byte* o *short*
 - una costante terminata da una *l* o *L* viene considerata espressione di tipo *long*
 - I prefissi 0 o 0x indicano costanti in base 8 o 16

Esempi di letterali interi

- *15* espressione *int* che vale 15
- *15L* espressione *long* che vale 15
- *017* espressione *int* (in ottale) che vale 15
- *0xF* espressione *int* (in esadecimale) che vale 15

- *short a = 15* espressione *short*

Operatori aritmetici di base

- I letterali e le variabili intere sono espressioni elementari; Java offre degli [operatori aritmetici](#) che consentono di comporre espressioni complesse
- Gli operatori aritmetici di base sono gli stessi del C:
 - + somma
 - sottrazione
 - * moltiplicazione
 - / divisione intera (tronca la parte decimale)
 - % resto della divisione intera

Parentesi

- Come in C, si possono usare parentesi per forzare le regole di precedenza degli operatori
 - si possono usare solo parentesi tonde, con un livello qualunque di annidamento

- Ad esempio, la seguente espressione vale **10**

$$((5-2)*7)\%3+10/(4-3)$$

Ancora sul tipo delle espressioni intere

- Ogni espressione intera che fa uso di operatori viene considerata come espressione:
 - di tipo *int*, se non contiene operandi *long*
 - di tipo *long*, se contiene operandi *long* (ad esempio $5 + 5L$ è una espressione di tipo *long*)
- *Osservazione*: Se *a* e *b* sono variabili *short*, l'espressione $a+b$ è di tipo *int* !!

Assegnazioni di espressioni intere

- Per assegnare il valore di una espressione intera *<esp>* ad una variabile intera *var* dello stesso tipo, basta scrivere

var = <esp>

- In alcuni casi è anche possibile assegnare espressioni di un certo tipo intero a variabili intere di tipo diverso – analizzeremo più avanti questi casi

Operatori di assegnazione composta

- Anche in Java come in C esistono gli operatori di assegnazione composta

ESPRESSIONE	ESPRESSIONE EQUIVALENTE
<code>a = a+b</code>	<code>a += b</code>
<code>a = a-b</code>	<code>a -= b</code>
<code>a = a*b</code>	<code>a *= b</code>
<code>a = a/b</code>	<code>a /= b</code>
<code>a = a%b</code>	<code>a %= b</code>

- e gli operatori di incremento e decremento

ESPRESSIONE	ESP. EQUIVALENTE	ESP. EQUIVALENTE	ESP. EQUIVALENTE
<code>a = a+1</code>	<code>a += 1</code>	<code>a++</code>	<code>++a</code>
<code>a = a-1</code>	<code>a -= 1</code>	<code>a--</code>	<code>--a</code>

Overflow

- Anche in Java, come in C, si può avere una situazione di overflow a seguito dell'esecuzione di un calcolo
- Ad esempio, se assegno ad una variabile *int* l'espressione $2147483647 + 1$ cosa accade? (ricordiamo che 2147483647 è il massimo valore *int*)
 - non vengono segnalati errori, ma viene assegnato alla variabile un valore errato (nel caso specifico il valore -2^{31} cioè -2147483648)

Espressioni in virgola mobile

- Le espressioni in virgola mobile sono di tipo *double* o *float*
- I letterali *double* sono costanti numeriche con virgola, espresse usando il “.” come separatore tra cifre intere e decimali; si può anche usare la notazione scientifica xEy , equivalente a $x10^y$
- Esempio di letterali *double* equivalenti:
 1.23 $0.00123E3$ $12.3E-1$
- Si può anche esplicitare un letterale *double*, mettendo il suffisso *d* o *D* al termine del numero
- Una costante numerica con virgola sarà considerata letterale *float* solo se specificato con il suffisso *f* o *F* (esempio $1.23F$)

Operatori per espressioni con virgola

- Abbiamo tutti gli operatori che esistono anche in C, cioè tutti gli operatori usati per le espressioni intere
- A differenza del C in Java esiste anche l'operatore di resto per i tipi in virgola mobile
- Se x ed y sono numeri con virgola, l'espressione $x \% y$ vale $x - n*y$, dove n è il numero di volte per il quale y sta interamente in x
 - l'espressione $5.0\%2.5$ vale 0
 - l'espressione $5.0\%2.4$ vale 0.2
- Ricordiamo che l'operatore di divisione si comporta diversamente a seconda del tipo degli operandi:
 - $15/2$ è una espressione intera che vale 7
 - $15.0/2.0$ è una espressione con virgola che vale 7.5

Espressioni numeriche miste

- Come in C, è possibile definire espressioni con operandi numerici di tipo misto, cioè alcuni interi e altri con virgola; il tipo di una espressione mista è quello con dominio più ampio tra quelli coinvolti:
 - $7/2.0$ è una espressione *double* di valore 3.5
 - $7/2 * 3.0$ è una espressione *double* di valore 9.0 ; infatti, viene prima valutata la sottoespressione $7/2$, che è di tipo *int* e vale 3 ; poi viene valutata $3 * 3.0$, che è di tipo *double* e vale 9.0 .

Overflow e Underflow

- Quando il valore assoluto di una espressione in virgola mobile è troppo alto si parla di overflow (verso $+\infty$ o $-\infty$); se è troppo basso si parla di underflow (verso $+0.0$ o -0.0)
- Java introduce i valori *Infinity* e *-Infinity* per indicare $+\infty$ e $-\infty$ (vedremo poi come richiamarli)
 - ad esempio, la seguente istruzione genera un overflow verso $-\infty$ e stampa a video *-Infinity*:
*System.out.print (-1.0E+300*1.0E+300);*
 - Invece questa istruzione stampa -0.0 per via di un underflow: *System.out.print (-1.0E-300*1.0E-300);*

Forme indeterminate

- In Java anche una espressione del tipo $1.0/0$ varrebbe *Infinity* (in altri linguaggi potrebbe causare un errore del tipo “Division By Zero”)
- In presenza di una espressione in virgola mobile che contiene una forma indeterminata (es. $0.0/0$) il valore associato all’espressione è *NaN* (Not a Number)

Espressioni di tipo char

- Ad una espressione di tipo *char* rimane associato un carattere dell'alfabeto Unicode
- I letterali *char* sono i singoli caratteri Unicode, e si possono esprimere in uno dei seguenti modi:
 - mettendo il carattere tra *apici singoli* (es. *'a'*)
 - tramite una sequenza di escape della forma *'\uxxxx'*, dove *xxxx* indica la posizione del carattere nella tabella Unicode, in esadecimale (es. *'\u0061'* indica il carattere di posizione 97, cioè il carattere *'a'*)
 - sequenze di escape (es. *'\n'*, *'\t'*, ..)

Operatori tra char

- E' possibile formare espressioni *char* più complesse, usando gli operatori aritmetici
 - l'operatore agisce tra i numeri di posizione dei caratteri coinvolti
 - ad esempio il frammento seguente stampa a video il carattere che occupa la posizione *97*; infatti, *'0'* and *'1'* occupano le posizioni *48* e *49*, rispettivamente

```
char c = '0' + '1';
```

```
System.out.print (c);
```

- Le espressioni di tipo *char* sono in realtà espressioni di tipo *int*; questo può portare a problemi di assegnamento di una espressione *char* ad una variabile di tipo *char* (lo vedremo dopo)

Conversioni tra tipi di dati

- Abbiamo visto che in C possiamo sempre assegnare ad una variabile di tipo t_1 un valore di tipo t_2
- Avviene una conversione implicita (o cast implicito)
 - dal valore dell'espressione di tipo t_2 viene ricavato un valore rappresentato secondo la codifica del tipo di t_1 , che viene poi assegnato alla variabile
- Ciò può portare ad una perdita di precisione se il dominio di valori di t_2 NON è contenuto in quello di t_1

Conversioni tra tipi di dati

- In Java è possibile effettuare una conversione implicita solo nel caso in cui non ci sia una perdita di precisione
- Ad esempio il seguente codice è legittimo:

```
short a = 255;
```

```
int b = a;
```

- Invece il seguente codice non verrebbe compilato

```
int a = 255;
```

```
short b = a;
```

viene generato l'errore “.. *possible loss of precision*”

Conversione esplicita

- Si può tuttavia forzare un'operazione di assegnazione che può causare perdita di precisione, tramite una conversione esplicita
- Nel precedente esempio si può scrivere:

```
int a = 255;
```

```
short b = (short)a;
```

- Se il valore convertito è al di fuori del dominio di valori del tipo verso cui si converte, si avrà una perdita di precisione

Conversione esplicita e char

- Il tipo *char* è a tutti gli effetti assimilabile ad un tipo numerico, che usa 16 bit per codificare numeri *non negativi* nell'intervallo $[0, 2^{16} - 1]$
- Ne segue che:
 - è necessario usare l'operatore di cast per assegnare valori interi con segno (*byte*, *short*, *int* o *long*) ad una variabile *char*
 - è necessario usare l'operatore di cast per assegnare un *char* ad una variabile *short* o *byte*, perché il dominio positivo dei *char* è più ampio di quello di *short* o di *byte*
 - si può assegnare un *char*, senza conversione esplicita, ad una variabile *int*, *long*, *float* o *double*

Ulteriori osservazioni sui char

- Il seguente codice non è valido:

```
char c1 = 'a';
```

```
char c2 = 'b';
```

```
char c3 = c1+c2;
```

perché l'espressione *c1+c2* è di tipo *int*; occorre dunque scrivere:

```
char c1 = 'a';
```

```
char c2 = 'b';
```

```
char c3 = (char)(c1+c2);
```

- Invece si può assegnare ad una variabile *char* il valore di una espressione composta soltanto da letterali *char*

Ulteriori osservazioni sui char

- Se c è una variabile *char* è consentito scrivere $c++$
- Se c è una variabile *char* non è consentito scrivere $c = c+1$, perché $c+1$ è una espressione *int*
- Si può invece scrivere $c = (\text{char})(c+1)$

Il tipo boolean

- A differenza del C in Java esiste esplicitamente un tipo per rappresentare i valori di verità
- Tale tipo è il tipo *boolean*
- Il nome *boolean* è in onore del matematico britannico George Boole (1815 - 1864), ideatore dell'algebra che porta il suo nome ([algebra booleana](#))
- Poiché un valore *boolean* può essere solo *true* o *false*, in linea teorica è sufficiente un solo bit per la rappresentazione di questo dato; per questioni di ordine pratico però, Java usa 8 bit (1 byte)

Espressioni di tipo boolean

- Una espressione *boolean* viene anche chiamata predicato: i letterali *true* o *false* sono i predicati più elementari
- E' possibile costruire predicati più complessi attraverso l'uso di due tipi di operatori:
 - operatori logici: operano tra valori *boolean* e restituiscono un valore *boolean*
 - operatori relazionali: operano tra valori numerici (inclusi i *char*) e restituiscono un valore *boolean*

Operatori logici

- Gli operatori logici in Java sono i seguenti:
 - `&&` AND logico
 - `//` OR logico (inclusivo)
 - `^` OR logico esclusivo (XOR)
 - `!` NOT (negazione)
- I primi tre sono operatori binari, mentre il NOT è un operatore unario
- L'operatore NOT ha precedenza rispetto agli altri e l'operatore AND ha precedenza rispetto agli OR

Operatori logici: semantica

- Ecco la tavola di verità degli operatori logici binari

a	b	a && b	a b	a ^ b
false	false	false	false	false
false	true	false	true	true
true	false	false	true	true
true	true	true	true	false

- Ecco la tavola di verità del NOT

a	!a
false	true
true	false

Operatori logici: precedenze

- L'operatore NOT ha precedenza rispetto agli altri e l'operatore AND ha precedenza rispetto agli OR
- E' possibile usare parentesi per forzare le regole di precedenze (come nelle espressioni numeriche)
- Nelle espressioni $a \ \&\& \ b$ oppure $a \ \|\ b$, la JVM valuta il secondo operando solo se necessario
 - cioè a meno che il valore del primo non sia già sufficiente a dedurre il risultato dell'espressione

Operatori relazionali

- Gli operatori relazionali Java sono i seguenti:
 - == uguale a
 - != diverso da
 - > maggiore di
 - < minore di
 - >= maggiore o uguale a
 - <= minore o uguale a
- Sono tutti operatori binari che hanno lo stesso valore che gli si attribuisce nella matematica e in C

Operatori relazionali: esempi

- Ecco alcuni esempi di semplici espressioni *boolean* che usano operatori relazionali

ESPRESSIONE	VALORE
10>5	true
10.0<5.0	false
10.0==10	true
5<=6	true
7.4!=7.41	true
'a'>'z'	false
'a'>'Z'	true
'a'>'5'	true

Operatori relazionali e riferimenti

- Gli operatori relazionali $==$ e $!=$ possono anche essere usati per confrontare riferimenti tra oggetti
 - Se a e b sono variabili riferimento di tipo C , l'espressione $a == b$ vale *true* se e solo se a e b referenziano lo stesso oggetto
 - Viceversa, $a != b$ vale *true* se e solo se a e b referenziano oggetti diversi

Le costanti in Java

- Oltre alle variabili, in Java si possono definire dei contenitori chiamati costanti
- Una costante è come una variabile alla quale però può essere dato *un solo valore durante tutto il suo ciclo di vita*
 - il valore di una costante è tipicamente assegnato contestualmente alla sua dichiarazione
- Una costante viene dichiarata come una variabile (tipo + nome), ma la dichiarazione è preceduta dalla parola chiave final

Costanti: un esempio

```
import fond.io.*;

public class CalcoloMisureCerchio{
    /* Calcola l'area e il perimetro di un cerchio di raggio inserito dall'utente */

    public static void main(String[] args){
        InputWindow in = new InputWindow();
        double r = in.readDouble("Inserire il raggio");
        final double PIGRECO = 3.14159; // costante
        double area = r*r*PIGRECO;
        double perimetro = 2*r*PIGRECO;
        OutputWindow out = new OutputWindow();
        out.write("Area del cerchio = ");
        out.writeln(area);
        out.write("Perimetro del cerchio = ");
        out.writeln(perimetro);
    }
}
```

L'importanza di usare costanti

- Ovviamente è possibile fare a meno delle costanti, usando direttamente i letterali nel codice
 - nell'esempio precedente potevamo scrivere direttamente `3.14159` ovunque usavamo `PIGRECO`
- Tuttavia, usare le costanti consente di modificare in modo minimale il codice in caso di modifica dei valori costanti usati nel codice
 - usare direttamente i letterali può comportare modifiche multiple nel codice

Perché i tipi primitivi?

- Java nasce con l'intento di trattare ogni dato come se fosse un oggetto; tuttavia i tipi primitivi fanno eccezione
- In linea teorica nulla vieterebbe di trattare come classi i tipi di dati primitivi
- I tipi primitivi sono stati introdotti perché è più efficiente (sia in termini di tempo che di memoria) manipolarli rispetto agli oggetti

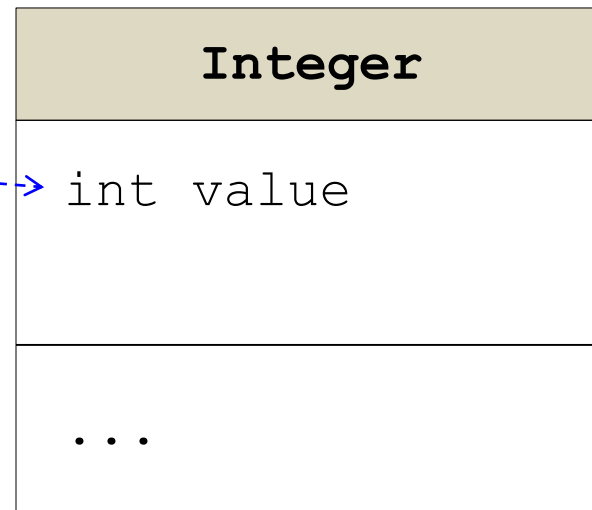
Le classi wrapper

- Nel package *java.lang* vengono fornite delle classi che rappresentano le tipologie di dati riconducibili a dati primitivi; tali classi si chiamano classi wrapper (classi involucro)
- Per ogni tipo primitivo *t* esiste una classe wrapper che rappresenta *t*: le classi *Byte*, *Short*, *Integer*, *Long*, *Float*, *Double*, *Character*, *Boolean* modellano oggetti che rappresentano rispettivamente valori primitivi di tipo *byte*, *short*, *int*, *long*, *float*, *double*, *char*, *boolean*

Struttura di un oggetto wrapper

- Un oggetto di una classe wrapper ha una struttura molto semplice; esso “incarta” (cioè mantiene in memoria) il valore primitivo da rappresentare, che costituisce dunque il suo stato

valore primitivo
rappresentato
dall'oggetto



Immutabilità degli oggetti wrapper

- Ogni oggetto wrapper in Java è immutabile, cioè il suo stato non può essere cambiato (non esistono metodi pubblici per farlo); questo significa che una volta creato l'oggetto, il valore primitivo che esso rappresenta non può essere cambiato

Creazione di oggetti wrapper

- Nel seguito indichiamo genericamente con *W* il nome di una classe wrapper e con *w* il tipo primitivo corrispondente
- Un oggetto di tipo *W* può essere creato con un costruttore al quale si specifica il valore di tipo *w* che esso rappresenterà

new Integer (100);

- Ad eccezione di *Character*, si può anche usare un costruttore al quale passare il valore primitivo descritto come stringa

new Integer ("100");

Metodi delle classi wrapper

- Il metodo di istanza *wValue()* restituisce il valore primitivo incartato da un oggetto di tipo *W*

```
Integer ogg = new Integer (100);
```

```
int a = ogg.intValue();
```

← ad *a* viene assegnato
il valore 100

Metodi delle classi wrapper

- Ogni classe wrapper W , tranne *Character*, ha un metodo di classe *valueOf* (*String s*) che restituisce un oggetto di tipo W che incarta il valore primitivo w descritto da s

Integer.valueOf ("100"); ←----- restituisce un oggetto *Integer* che incarta il valore 100

- Ogni classe wrapper W , tranne *Character*, ha un metodo di classe *parseX* (*String s*) che restituisce il valore primitivo w descritto da s ($X \in \{Int, Double, Boolean, \dots\}$)

Integer.parseInt ("100"); ←----- restituisce il valore 100

Costanti nelle classi wrapper

- All'interno di ogni classe wrapper *W* sono definite alcune utili costanti statiche:
 - *MIN_VALUE* e *MAX_VALUE* indicano il minimo e il massimo valore di tipo *w* (ad esempio *Integer.MIN_VALUE* vale -2^{31} e *Integer.MAX_VALUE* vale $2^{31}-1$;
 - *Double.NEGATIVE_INFINITY* e *Double.POSITIVE_INFINITY* indicano i “valori” $-\infty$ e $+\infty$ mentre *Double.NaN* rappresenta una forma indeterminata;
 - esistono analoghe costanti nella classe *Float*

Autoboxing

- l'importanza delle classi wrapper è anche legata all'esigenza di poter creare collezioni di dati primitivi che possano essere manipolati come fossero degli oggetti (lo vedremo più avanti nel corso)
- Per agevolare il passaggio da un dato di tipo *W* ad un dato di tipo primitivo *w* e viceversa, le recenti versioni di Java adottano dei meccanismi automatici di conversione implicita, chiamati autoboxing
- Si può ad esempio scrivere:
 - *int i = ogg* dove *ogg* è un *Integer* (unboxing);
 - *Integer ogg = 100* (inboxing);

La classe Math

- Gli operatori matematici di base per i tipi primitivi non sono sempre sufficienti per effettuare calcoli matematici complessi
- Nel package *java.lang* è definita la classe *Math*, che non dispone di costruttori pubblici e che offre soltanto metodi pubblici statici
 - ogni metodo statico è finalizzato al calcolo di una qualche specifica funzione matematica

La classe Math: alcuni metodi

- La tabella seguente elenca alcuni dei metodi statici della classe *Math*

PROTOTIPO DEL METODO	VALORE RESTITUITO
<code>double abs(double a)</code>	valore assoluto di a
<code>double sqrt(double a)</code>	radice quadrata di a
<code>double pow(double a, double b)</code>	valore della potenza a^b
<code>double sin(double a)</code>	seno dell'angolo a (in radianti)
<code>double cos(double a)</code>	coseno dell'angolo a (in radianti)
<code>double tan(double a)</code>	tangente dell'angolo a (in radianti)
<code>double log(double a)</code>	logaritmo in base e di a
<code>double log10(double a)</code>	logaritmo in base 10 di a

La classe Math: costanti

- Nella classe *Math* sono inoltre definite alcuni costanti notevoli:
 - *Math.PI* il valore di π , con la migliore approssimazione possibile sotto forma di *double*
 - *Math.E* la costante e , cioè il numero di Nepero

La classe Math: esempio 1

```
import fond.io.*;

public class CalcoloMisureCerchio1{
    /* Calcola l'area e il perimetro di un cerchio di raggio inserito
       dall'utente, facendo uso della classe Math */

    public static void main(String[] args){
        InputWindow in = new InputWindow();
        double r = in.readDouble("Inserire il raggio");
        double area = Math.pow(r, 2)*Math.PI;
        double perimetro = 2*r*Math.PI;
        OutputWindow out = new OutputWindow();
        out.write("Area del cerchio = ");
        out.writeln(area);
        out.write("Perimetro del cerchio = ");
        out.writeln(perimetro);
    }
}
```

La classe Math: esempio 2

```
import fond.io.*;

public class CalcoloDistanzaPunti{
    /* Calcola la distanza tra due punti del piano
       p1=(x1,x2) e p2=(x2,y2) */

    public static void main(String[] args){
        InputWindow in = new InputWindow();
        double x1 = in.readDouble("Punto 1: coordinata x?");
        double y1 = in.readDouble("Punto 1: coordinata y?");
        double x2 = in.readDouble("Punto 2: coordinata x?");
        double y2 = in.readDouble("Punto 2: coordinata y?");
        double d; // distanza tra p1 e p2
        d = Math.sqrt(Math.pow(x1-x2, 2)+Math.pow(y1-y2, 2));
        OutputWindow out = new OutputWindow();
        out.write("Distanza = ");
        out.writeln(d);
    }
}
```


La classe String

- Abbiamo detto che gli oggetti di tipo *String* rappresentano sequenze di caratteri (cioè sequenze di valori *char*)
 - gli oggetti *String* sono usati molto di frequente, così come i tipi primitivi
 - la classe *String* gode di alcune peculiarità che la accomunano ai tipi primitivi
- La stringa rappresentata da un oggetto *String* costituisce lo stato dell'oggetto, e verrà anche chiamata valore dell'oggetto *String*
- Così come gli oggetti wrapper, anche gli oggetti *String* sono immutabili

Creazione di oggetti String: letterali

- Sappiamo che un letterale stringa costituisce di fatto un oggetto della classe *String*; si può ad esempio scrivere la seguente istruzione

```
String str = "ciao mondo!";
```

- Nella variabile *str* viene memorizzato il riferimento all'oggetto definito usando il letterale *"ciao mondo!"*
- Ad un letterale stringa rimane associato un solo oggetto!
- La stringa senza caratteri "" si chiama anche [stringa vuota](#)

Creazione di oggetti `String` con `new`

- Come per ogni altra classe, è possibile istanziare oggetti di tipo `String` anche usando l'operatore `new`
- In particolare, il costruttore pubblico `String (String s)` permette di creare un oggetto `String` come copia di quello passato come parametro

```
String str = new String("ciao mondo");
```
- A differenza dei letterali, l'uso dell'operatore `new` dà sempre luogo alla creazione di un oggetto `String` differente

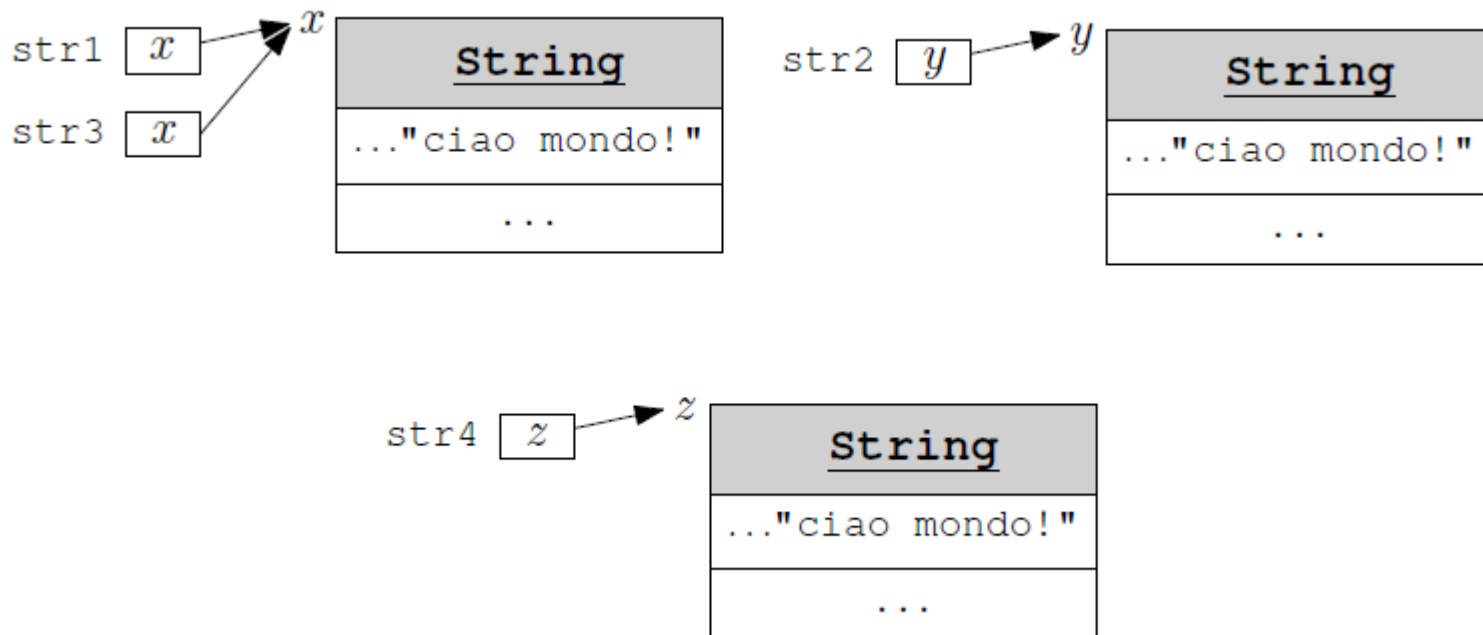
Creazione di oggetti String: esempio

String str1 = "ciao mondo!";

String str2 = new String("ciao mondo!");

String str3 = "ciao mondo!";

String str4 = new String(str1);



L'operatore di concatenazione

- L'operatore `+` applicato a due oggetti *String*, *str1* e *str2*, fornisce un nuovo oggetto *String* il cui valore è la concatenazione (giustapposizione) dei valori di *str1* e di *str2*
- Ad esempio, la seguente istruzione memorizza in *str* il riferimento ad un oggetto *String* di valore *“ciao mondo!”*

```
String str = “ciao”+” mondo!”;
```

- L'operatore `+` applicato alle stringhe si chiama dunque [operatore di concatenazione](#)

L'operatore di concatenazione

- L'operatore `+` può anche essere applicato tra uno oggetto `s` di tipo `String` e un dato primitivo; in questo caso, il risultato sarà un oggetto `String` il cui valore è ottenuto concatenando il valore di `s` con la stringa che descrive il dato primitivo
- *Esempio 1:* `String str = 10 + "rose";`
 - l'oggetto referenziato da `str` ha il valore `"10 rose"` (il valore intero `10` è dapprima convertito nella stringa `"10"`, e quindi concatenato con `"rose"`).
- *Esempio 2:* `String str = 10 + 15 + "rose";`
 - l'oggetto referenziato da `str` ha il valore `"25 rose"` (vengono prima sommati `10` e `15`; il risultato viene concatenato a `"rose"`)

Metodi della classe String

- La classe *String* dispone di numerosi metodi di istanza e di classe, per la manipolazione di stringhe; nel seguito vedremo alcuni di questi metodi

Il metodo length

- Metodo di istanza che restituisce il numero di caratteri della stringa
- Prototipo: *int length()*
- Ad esempio, l'istruzione *"mondo".length()*, fornisce come risultato 5

Il metodo charAt

- Metodo di istanza che restituisce il carattere di dell'oggetto ricevente, che occupa la posizione specificata (il primo carattere ha sempre posizione 0!!)
- Prototipo: *char charAt(int index);*
- Ad esempio, l'istruzione *System.out.print("mondo".charAt(2))* stampa a video il carattere 'n'

Il metodo equals

- Metodo di istanza che confronta il valore di due oggetti *String* e che restituisce *true* se tali valori sono uguali, e *false* altrimenti
- Prototipo: *boolean equals (String s)*; (in realtà il parametro formale è un oggetto della classe *Object* (descritta in capitoli successivi))
- Ad esempio, il seguente frammento di codice stampa *true*

```
String str1 = new String("ciao mondo!");  
String str2 = new String("ciao mondo!");  
System.out.print(str1.equals(str2));
```

Il metodo equals: osservazione

```
String str1 = new String("ciao mondo!");  
String str2 = new String("ciao mondo!");  
System.out.print(str1.equals(str2));
```

- L'espressione *str1.equals(str2)* vale *true* perché gli oggetti referenziati da *str1* e da *str2*, sebbene siano diversi, hanno lo stesso valore

Il metodo compareTo

- Metodo di istanza che confronta “lessicograficamente” due stringhe
- L’ordine lessicografico è simile all’ordinamento alfabetico, ma distingue tra minuscola e maiuscola:
 - “alfa” precede “alfabeto”
 - “castello” segue “canile”
 - “Beta” precede “alfa”
- Prototipo: *int compareTo(String s)*; restituisce:
 - un valore negativo se il ricevente precede *s* lessicograficamente
 - un valore positivo se il ricevente segue *s* lessicograficamente
 - *0* se le stringhe hanno lo stesso valore

Il metodo indexOf

- Metodo di istanza che verifica se una stringa è contenuta in un'altra
- Prototipo: *int indexOf(String s)*
 - restituisce la posizione iniziale di *s* all'interno della stringa ricevente, se *s* è contenuta nella stringa ricevente, altrimenti restituisce *-1*
- *Esempi:*
 - *“ciao”.indexOf(“ia”) → vale 1*
 - *“ciao”.indexOf(“cio”) → vale -1*

Il metodo substring

- Estrae una sottostringa da una stringa
- Prototipi:
 - *String substring (int a, int b)*: restituisce la sottostringa dell'oggetto ricevente dalla posizione *a* (inclusa) alla posizione *b* (escluso)
 - *String substring (int a)*: restituisce la sottostringa dell'oggetto ricevente dalla posizione *a* alla fine
- *Esempi*:
 - *“ciao mondo!”.substring(2, 6) → vale “ao m”*
 - *“ciao mondo!”.substring(5) → vale “mondo!”*

I metodi toUpperCase e toLowerCase

- Metodi di istanza per creare stringhe con sole lettere maiuscole o sole lettere minuscole
- Prototipo:
 - *String toUpperCase()*
 - *String toLowerCase ()*
- *Esempi:*
 - *“cIAo”.toLowerCase() → vale “ciao”*
 - *“cIAo”.toUpperCase() → vale “CIAO”*

Il metodo `valueOf`

- Metodo di classe che restituisce una stringa che descrive un valore primitivo
- Prototipo:
 - *String valueOf (t valore), dove t è un dato primitivo*
- *Esempi:*
 - *String.valueOf(1500) → vale “1500”*
 - *String.valueOf(30.4) → vale “30.4”*