



Capitolo 4

Tipi primitivi e tipi enumerativi

1 Espressioni

- Operatori di assegnamento e incremento ed effetti collaterali
- Precedenza e associatività degli operatori
- Lazy evaluation
- Tipi numerici interi
- Conversioni di tipo: promozioni e cast
- Tipi numerici in virgola mobile
- Il tipo `char`

2 Tipi enumerativi

3 L'istruzione `switch`

4 Metodi e campi statici

5 Classi involucro

Espressione

È una porzione di codice Java, che ha:

- un **tipo** determinato al momento della compilazione
- un **valore** determinato al momento dell'esecuzione.

Espressione

È una porzione di codice Java, che ha:

- un **tipo** determinato al momento della compilazione
- un **valore** determinato al momento dell'esecuzione.

Esempi

- `i + j`

Se `i` e `j` sono di tipo `int` è un'espressione di tipo `int`

Espressione

È una porzione di codice Java, che ha:

- un **tipo** determinato al momento della compilazione
- un **valore** determinato al momento dell'esecuzione.

Esempi

- `i + j`
Se `i` e `j` sono di tipo `int` è un'espressione di tipo `int`
- `"ciao".toUpperCase()`
ha tipo `String` (il suo risultato è un riferimento a un oggetto `String`)

- Letterali

25 letterale di tipo `int`

"pippo" letterale di tipo `String`

...

...

- Letterali

25 letterale di tipo `int`

"pippo" letterale di tipo `String`

... ...

- Variabili

`tipo` il tipo associato alla variabile in fase di dichiarazione
`valore` il valore contenuto nella variabile

- Letterali

25 letterale di tipo `int`

"pippo" letterale di tipo `String`

... ...

- Variabili

`tipo` il tipo associato alla variabile in fase di dichiarazione
`valore` il valore contenuto nella variabile

- Invocazione di metodo

`tipo` il tipo restituito dal metodo
`valore` risultato dell'esecuzione del metodo

- Letterali

25 letterale di tipo `int`

"pippo" letterale di tipo `String`

... ...

- Variabili

`tipo` il tipo associato alla variabile in fase di dichiarazione
`valore` il valore contenuto nella variabile

- Invocazione di metodo

`tipo` il tipo restituito dal metodo
`valore` risultato dell'esecuzione del metodo

- Le espressioni composte sono costituite combinando espressioni mediante operatori

Operatori binari (`int × int → int`)

<code>+</code>	somma
<code>-</code>	sottrazione
<code>*</code>	prodotto
<code>/</code>	divisione
<code>%</code>	resto della divisione
<code>...</code>	

Operatori binari (`int` × `int` → `int`)

+	somma
-	sottrazione
*	prodotto
/	divisione
%	resto della divisione
...	

- Le regole di **precedenza** sono le stesse stabilite dall'algebra. Possono essere modificate usando le parentesi tonde

$$4 * (i + j) - (3 + k * (i - j))$$

Operatori binari (`int × int → int`)

+	somma
-	sottrazione
*	prodotto
/	divisione
%	resto della divisione
...	

- Le regole di **precedenza** sono le stesse stabilite dall'algebra. Possono essere modificate usando le parentesi tonde

$$4 * (i + j) - (3 + k * (i - j))$$

- e + possono essere utilizzati anche come **operatori unari in notazione prefissa**

L'operatore di assegnamento (=)

Espressioni di assegnamento

L'operatore = dà luogo ad espressioni del tipo

variabile = espressione

con

tipo il tipo della variabile alla sinistra dell'operatore

valore il valore dell'espressione alla destra dell'operatore

L'operatore di assegnamento (=)

Espressioni di assegnamento

L'operatore = dà luogo ad espressioni del tipo

variabile = *espressione*

con

tipo il tipo della *variabile* alla sinistra dell'operatore

valore il valore dell'*espressione* alla destra dell'operatore

- Affinché un'espressione di assegnamento sia corretta il tipo di *espressione* deve essere compatibile con quello di *variabile*,...
... cioè il valore di *espressione* deve essere assegnabile a *variabile*

L'operatore di assegnamento (=)

Espressioni di assegnamento

L'operatore = dà luogo ad espressioni del tipo

variabile = *espressione*

con

tipo il tipo della variabile alla sinistra dell'operatore

valore il valore dell'espressione alla destra dell'operatore

- Affinché un'espressione di assegnamento sia corretta il tipo di *espressione* deve essere compatibile con quello di *variabile*,...
... cioè il valore di *espressione* deve essere assegnabile a *variabile*
- Un'*espressione di assegnamento* seguita da un "punto e virgola" (;) dà luogo ad un'*istruzione*

Esempi

- Sia `x` di tipo `int`

```
x = 10;
```

è un'espressione corretta con

```
tipo int  
valore 10
```

Esempi

- Sia `x` di tipo `int`

```
x = 10;
```

è un'espressione corretta con

tipo `int`
valore `10`

- Sia `s` di tipo `String`

```
s = "pippo".toUpperCase()
```

è un'espressione corretta con

tipo `String`
valore il riferimento alla stringa `"PIPP0"`

```
int i, j;  
i = 3;  
j = i + (i = 5);
```

Dopo l'esecuzione

- i contiene il valore 5
- j contiene il valore 8

```
int i, j;  
i = 3;  
j = (i = 5) + i;
```

Dopo l'esecuzione

- i contiene il valore 5
- j contiene il valore 10

Effetti collaterali

```
int i, j;  
i = 3;  
j = i + (i = 5);
```

Dopo l'esecuzione

- i contiene il valore 5
- j contiene il valore 8

```
int i, j;  
i = 3;  
j = (i = 5) + i;
```

Dopo l'esecuzione

- i contiene il valore 5
- j contiene il valore 10

- La valutazione dell'espressione ha un **effetto collaterale**, modifica il valore della variabile x

Effetti collaterali

```
int i, j;  
i = 3;  
j = i + (i = 5);
```

Dopo l'esecuzione

- i contiene il valore 5
- j contiene il valore 8

```
int i, j;  
i = 3;  
j = (i = 5) + i;
```

Dopo l'esecuzione

- i contiene il valore 5
- j contiene il valore 10

- La valutazione dell'espressione ha un **effetto collaterale**, modifica il valore della variabile x
- La presenza di effetti collaterali rende i programmi più difficili da leggere

Abbreviazioni per l'operatore di assegnamento

Se x e y sono variabili di tipo numerico (interi o in virgola mobile):

$$x += y \quad \Rightarrow \quad x = x + y$$

$$x -= y \quad \Rightarrow \quad x = x - y$$

$$x *= y \quad \Rightarrow \quad x = x * y$$

$$x /= y \quad \Rightarrow \quad x = x / y$$

$$x \% = y \quad \Rightarrow \quad x = x \% y$$

Operatori di incremento (++) e decremento (--)

- Sono operatori unari, si possono utilizzare *prefissi* o *postfissi* (cambia la semantica).

Operatori di incremento (++) e decremento (--)

- Sono operatori unari, si possono utilizzare *prefissi* o *postfissi* (cambia la semantica).
- Si applicano a *variabili* di tipo numerico

Operatori di incremento (++) e decremento (--)

- Sono operatori unari, si possono utilizzare *prefissi* o *postfissi* (cambia la semantica).
- Si applicano a *variabili* di tipo numerico

Notazione prefissa

```
i = 1;  
j = ++i; // i = j = 2
```

prima viene incrementata la variabile, poi viene valutata l'espressione

Operatori di incremento (++) e decremento (--)

- Sono operatori unari, si possono utilizzare *prefissi* o *postfissi* (cambia la semantica).
- Si applicano a *variabili* di tipo numerico

Notazione prefissa

```
i = 1;  
j = ++i; // i = j = 2
```

prima viene incrementata la variabile, poi viene valutata l'espressione

Notazione postfissa

```
i = 1;  
j = i++; // i = 2, j = 1
```

prima viene valutata l'espressione, poi viene incrementata la variabile

Effetti collaterali

La valutazione di espressioni contenenti ++ e -- da luogo a **effetti collaterali**

```
int x = 3;
```

```
int y = 4;
```

(1) `x++ + y + x`

Effetti collaterali

La valutazione di espressioni contenenti ++ e -- da luogo a **effetti collaterali**

```
int x = 3;
```

```
int y = 4;
```

(1) `x++ + y + x`

(2) `++x + y + x;`

Effetti collaterali

La valutazione di espressioni contenenti ++ e -- da luogo a **effetti collaterali**

```
int x = 3;
```

```
int y = 4;
```

(1) `x++ + y + x`

(2) `++x + y + x;`

(3) `if (x++ == --y)`

```
    z = x + y;
```

```
else
```

```
    z = x - y;
```

Precedenza

Specifica il grado di priorità di un operatore rispetto ad un altro e quindi **l'ordine secondo il quale vengono applicati**.

Precedenza degli operatori

Precedenza

Specifica il grado di priorità di un operatore rispetto ad un altro e quindi **l'ordine secondo il quale vengono applicati**.

Esempio

In che ordine vengono applicati gli operatori nella seguente espressione?

$a + b * c$

Precedenza degli operatori

Precedenza

Specifica il grado di priorità di un operatore rispetto ad un altro e quindi **l'ordine secondo il quale vengono applicati**.

Esempio

In che ordine vengono applicati gli operatori nella seguente espressione?

$$a + b * c$$

La moltiplicazione ha un più alto grado di precedenza rispetto alla somma, quindi ad **a** viene sommato il valore di **b * c**

Precedenza degli operatori

Precedenza

Specifica il grado di priorità di un operatore rispetto ad un altro e quindi **l'ordine secondo il quale vengono applicati**.

Esempio

In che ordine vengono applicati gli operatori nella seguente espressione?

$$a + b * c$$

La moltiplicazione ha un più alto grado di precedenza rispetto alla somma, quindi ad a viene sommato il valore di $b * c$

L'ordine di applicazione degli operatori **può essere modificato** mediante l'uso delle parentesi tonde

$$(a + b) * c$$

Associatività degli operatori

Quando due operatori hanno **la stessa precedenza**, come viene stabilito l'ordine in cui le operazioni devono essere eseguite?

Associatività degli operatori

Quando due operatori hanno **la stessa precedenza**, come viene stabilito l'ordine in cui le operazioni devono essere eseguite?

Regole di associatività

Stabiliscono l'ordine nel quale le operazioni vengono eseguite:

- eseguite da sinistra a destra (**left-to-right**)
- eseguite da destra a sinistra (**right-to-left**)

Associatività degli operatori

Quando due operatori hanno **la stessa precedenza**, come viene stabilito l'ordine in cui le operazioni devono essere eseguite?

Regole di associatività

Stabiliscono l'ordine nel quale le operazioni vengono eseguite:

- eseguite da sinistra a destra (**left-to-right**)
- eseguite da destra a sinistra (**right-to-left**)

Tranne gli operatori di assegnamento, tutti gli **operatori binari** sono associativi a sinistra (**left-to-right**)

Associatività degli operatori

Quando due operatori hanno **la stessa precedenza**, come viene stabilito l'ordine in cui le operazioni devono essere eseguite?

Regole di associatività

Stabiliscono l'ordine nel quale le operazioni vengono eseguite:

- eseguite da sinistra a destra (**left-to-right**)
- eseguite da destra a sinistra (**right-to-left**)

Tranne gli operatori di assegnamento, tutti gli **operatori binari** sono associativi a sinistra (**left-to-right**)

Gli assegnamenti e gli operatori unari sono associativi a destra (**right-to-left**)

Associatività degli operatori

Quando due operatori hanno **la stessa precedenza**, come viene stabilito l'ordine in cui le operazioni devono essere eseguite?

Regole di associatività

Stabiliscono l'ordine nel quale le operazioni vengono eseguite:

- eseguite da sinistra a destra (**left-to-right**)
- eseguite da destra a sinistra (**right-to-left**)

Tranne gli operatori di assegnamento, tutti gli **operatori binari** sono associativi a sinistra (**left-to-right**)

Gli assegnamenti e gli operatori unari sono associativi a destra (**right-to-left**)

`a = b += c = -d` viene valutata come `a = (b += (c = (-d)))`

Lazy evaluation

Normalmente il risultato di un operatore è determinato dopo quello dei relativi operandi.

Normalmente il risultato di un operatore è determinato dopo quello dei relativi operandi.

Eccezioni

- Operatori booleani `&&` e `||`

Normalmente il risultato di un operatore è determinato dopo quello dei relativi operandi.

Eccezioni

- Operatori booleani `&&` e `||`

lazy evaluation (valutazione “pigra” o *cortocircuitata*)

Se, avendo valutato solo una parte di un'espressione booleana, è già possibile determinare il risultato dell'espressione, la parte che resta non viene valutata.

Normalmente il risultato di un operatore è determinato dopo quello dei relativi operandi.

Eccezioni

- Operatori booleani `&&` e `||`

lazy evaluation (valutazione “pigra” o *cortocircuitata*)

Se, avendo valutato solo una parte di un'espressione booleana, è già possibile determinare il risultato dell'espressione, la parte che resta non viene valutata.

- Operatore condizionale

L'operatore condizionale ? :

È un operatore ternario

condizione ? *espressione1* : *espressione2*

- *condizione* è un'espressione booleana
- *espressione1* e *espressione2* sono espressioni dello stesso tipo (o di tipi compatibili)

L'operatore condizionale ? :

È un operatore ternario

condizione ? *espressione1* : *espressione2*

- *condizione* è un'espressione booleana
- *espressione1* e *espressione2* sono espressioni dello stesso tipo (o di tipi compatibili)

Il **tipo** dell'espressione è lo stesso di *espressione1* e *espressione2*

L'operatore condizionale ? :

È un operatore ternario

condizione ? *espressione1* : *espressione2*

- *condizione* è un'espressione booleana
- *espressione1* e *espressione2* sono espressioni dello stesso tipo (o di tipi compatibili)

Il **tipo** dell'espressione è lo stesso di *espressione1* e *espressione2*

Il **valore** dell'espressione è:

- il valore di *espressione1* se *condizione* è valutata **true**
- il valore di *espressione2* se *condizione* è valutata **false**

Tipi numerici interi

- Rappresentano numeri interi con segno

Tipi numerici interi

- Rappresentano numeri interi con segno
- Si distinguono per il **range** di valori che possono essere rappresentati

Tipi numerici interi

- Rappresentano numeri interi con segno
- Si distinguono per il **range** di valori che possono essere rappresentati
- La **quantità di memoria** occupata da una variabile, e quindi il **range** dei valori rappresentabili, sono stabiliti dal linguaggio

Tipi numerici interi

- Rappresentano numeri interi con segno
- Si distinguono per il **range** di valori che possono essere rappresentati
- La **quantità di memoria** occupata da una variabile, e quindi il **range** dei valori rappresentabili, sono stabiliti dal linguaggio

Tipo	Bit	Range
byte	8 bit	da -2^7 a $2^7 - 1$
short	16 bit	da -2^{15} a $2^{15} - 1$
int	32 bit	da -2^{31} a $2^{31} - 1$
long	64 bit	da -2^{63} a $2^{63} - 1$

Letterale

Sequenze di cifre decimali

... eventualmente precedute dai segni + o -

... eventualmente seguite da `l` o `L` (per i long)

Letterale

Sequenze di cifre decimali

... eventualmente precedute dai segni + o -

... eventualmente seguite da `l` o `L` (per i `long`)

```
long x = 100L;
```

```
int y = 4000;
```

Letterale

Sequenze di cifre decimali

... eventualmente precedute dai segni + o -

... eventualmente seguite da `l` o `L` (per i long)

```
long x = 100L;
```

```
int y = 4000;
```

Per `byte` e `short`:

- non esistono letterali specifici, si utilizzano i letterali di tipo `int`
- il compilatore controlla che il valore del letterale sia nel range ammesso

```
byte b1 = 125;
```

```
byte b2 = 4000; //non viene compilato
```

Operatori binari

+	somma	-	sottrazione
*	prodotto	/	divisione
%	resto della divisione		

Operatori binari

+	somma	-	sottrazione
*	prodotto	/	divisione
%	resto della divisione		

$x + y$

Qual'è il suo tipo?

Operatori binari

+	somma	-	sottrazione
*	prodotto	/	divisione
%	resto della divisione		

$x + y$

Qual'è il suo tipo?

Se x e y sono **entrambe** di tipo:

- **int** allora $x + y$ è di tipo **int**

Operatori binari

+	somma	-	sottrazione
*	prodotto	/	divisione
%	resto della divisione		

$x + y$

Qual'è il suo tipo?

Se x e y sono **entrambe** di tipo:

- **int** allora $x + y$ è di tipo **int**
- **long** allora $x + y$ è di tipo **long**

+	somma	-	sottrazione
*	prodotto	/	divisione
%	resto della divisione		

$x + y$

Qual'è il suo tipo?

Se x e y sono **entrambe** di tipo:

- **int** allora $x + y$ è di tipo **int**
- **long** allora $x + y$ è di tipo **long**
- **short** allora $x + y$ è di tipo **int**

+	somma	-	sottrazione
*	prodotto	/	divisione
%	resto della divisione		

$x + y$

Qual'è il suo tipo?

Se x e y sono **entrambe** di tipo:

- **int** allora $x + y$ è di tipo **int**
- **long** allora $x + y$ è di tipo **long**
- **short** allora $x + y$ è di tipo **int**
- **byte** allora $x + y$ è di tipo **int**

```
int y;  
long x, z;  
  
x = y + z
```

- il valore di **y** viene convertito nel corrispondente valore di tipo **long** prima di effettuare l'operazione

Conversioni implicite

```
int y;  
long x, z;  
  
x = y + z
```

- il valore di **y** viene convertito nel corrispondente valore di tipo **long** prima di effettuare l'operazione
- il risultato dell'espressione **y + z** è di tipo **long**

Conversioni implicite

```
int y;  
long x, z;  
  
x = y + z
```

- il valore di **y** viene convertito nel corrispondente valore di tipo **long** prima di effettuare l'operazione
- il risultato dell'espressione **y + z** è di tipo **long**

In generale i valori di un tipo più ristretto vengono “promossi” a un tipo più ampio

Conversioni implicite

```
int y, z;  
long x;  
  
x = y + z
```

Conversioni implicite

```
int y, z;  
long x;  
  
x = y + z
```

- il risultato dell'espressione $y + z$ è di tipo `int`

Conversioni implicite

```
int y, z;  
long x;  
  
x = y + z
```

- il risultato dell'espressione $y + z$ è di tipo `int`
- il valore di $y + z$ viene promosso al tipo `long` prima di effettuare l'assegnamento

Conversioni implicite fra int e long

```
x = y + z;
```

Possibili casi

x	y	z	y + z	x = y + z	promozioni
---	---	---	-------	-----------	------------

Conversioni implicite fra int e long

```
x = y + z;
```

Possibili casi

	x	y	z	y + z	x = y + z	promozioni
1	int	int	int	int	int	nessuna

Conversioni implicite fra int e long

```
x = y + z;
```

Possibili casi

	x	y	z	y + z	x = y + z	promozioni
1	int	int	int	int	int	nessuna
2	long	int	int	int	long	ris. di y + z

Conversioni implicite fra int e long

```
x = y + z;
```

Possibili casi

	x	y	z	y + z	x = y + z	promozioni
1	int	int	int	int	int	nessuna
2	long	int	int	int	long	ris. di y + z
3	long	long	int	long	long	val. di z

Conversioni implicite fra int e long

```
x = y + z;
```

Possibili casi

	x	y	z	y + z	x = y + z	promozioni
1	int	int	int	int	int	nessuna
2	long	int	int	int	long	ris. di y + z
3	long	long	int	long	long	val. di z
4	long	int	long	long	long	val. di y

Conversioni implicite fra int e long

```
x = y + z;
```

Possibili casi

	x	y	z	y + z	x = y + z	promozioni
1	int	int	int	int	int	nessuna
2	long	int	int	int	long	ris. di y + z
3	long	long	int	long	long	val. di z
4	long	int	long	long	long	val. di y
5	long	long	long	long	long	nessuna

Conversioni implicite fra int e long

```
x = y + z;
```

Possibili casi

	x	y	z	y + z	x = y + z	promozioni
1	int	int	int	int	int	nessuna
2	long	int	int	int	long	ris. di y + z
3	long	long	int	long	long	val. di z
4	long	int	long	long	long	val. di y
5	long	long	long	long	long	nessuna
6	int	long	int	long	impossibile	

Conversioni implicite fra int e long

```
x = y + z;
```

Possibili casi

	x	y	z	y + z	x = y + z	promozioni
1	int	int	int	int	int	nessuna
2	long	int	int	int	long	ris. di y + z
3	long	long	int	long	long	val. di z
4	long	int	long	long	long	val. di y
5	long	long	long	long	long	nessuna
6	int	long	int	long	impossibile	
7	int	int	long	long	impossibile	

Conversioni implicite fra int e long

```
x = y + z;
```

Possibili casi

	x	y	z	y + z	x = y + z	promozioni
1	int	int	int	int	int	nessuna
2	long	int	int	int	long	ris. di y + z
3	long	long	int	long	long	val. di z
4	long	int	long	long	long	val. di y
5	long	long	long	long	long	nessuna
6	int	long	int	long	impossibile	
7	int	int	long	long	impossibile	
8	int	long	long	long	impossibile	

Operatore di cast (forzatura)

(nome_del_tipo) espressione

Conversioni esplicite

Operatore di cast (forzatura)

(nome_del_tipo) espressione

Esempio

```
int x;  
long y;  
  
x = (int)y;
```

Operatore di cast (forzatura)

(nome_del_tipo) espressione

Esempio

```
int x;  
long y;  
  
x = (int)y;
```

- La conversione fra un tipo ampio e uno più ristretto **può comportare perdita di informazione**

Conversioni esplicite

Operatore di cast (forzatura)

(nome_del_tipo) espressione

Esempio

```
int x;  
long y;  
  
x = (int)y;
```

- La conversione fra un tipo ampio e uno più ristretto **può comportare perdita di informazione**
- Utilizzando l'operatore di cast il programmatore **dichiara di essere cosciente della possibile perdita di informazione**

Tipi numerici in virgola mobile

- Tipi primitivi, rappresentano numeri **floating point** (standard IEEE 754)

Tipi numerici in virgola mobile

- Tipi primitivi, rappresentano numeri **floating point** (standard IEEE 754)
- Si distinguono per il **range** e la **precisione** dei valori che consentono di rappresentare

Tipo	Bit	Range
float	32 bit	da $\pm 1.4E-45$ a $\pm 3.4E+38$
double	64 bit	da $\pm 4.9E-324$ a $\pm 1.8E+308$

Tipi numerici in virgola mobile

- Tipi primitivi, rappresentano numeri **floating point** (standard IEEE 754)
- Si distinguono per il **range** e la **precisione** dei valori che consentono di rappresentare

Tipo	Bit	Range
float	32 bit	da $\pm 1.4E-45$ a $\pm 3.4E+38$
double	64 bit	da $\pm 4.9E-324$ a $\pm 1.8E+308$

- Gli operatori sono gli stessi definiti sui tipi numerici interi

Notazione scientifica

2E10

+1.2e-5

-444.333E12

Notazione decimale

123.456

-77.0

+144.333

Notazione scientifica

```
2E10  
+1.2e-5  
-444.333E12
```

Notazione decimale

```
123.456  
-77.0  
+144.333
```

- Letterali `float`: vengono indicati postponendo `f` o `F`

```
float x = 3.25e10;  
float y = 13.25F;  
  
float f = 1E299; //non viene accettato
```

Il compilatore controlla che il valore di un letterale `float` sia nel range ammesso

Lo standard IEEE 754 prevede anche l'uso di 5 valori speciali:

- **infinito** positivo e negativo
- **zero** positivo e negativo
- **NaN** (Not a Number)

Lo standard IEEE 754 prevede anche l'uso di 5 valori speciali:

- **infinito** positivo e negativo
- **zero** positivo e negativo
- **NaN** (Not a Number)

```
double inf = 1.0/0.0;      // Infinity
double negInf = -1.0/0.0; // -Infinity
double zero = 1.0/Inf     // 0.0
double negZero = -1.0/inf; // -0.0
double NotANum = 0.0/0.0; // NaN
```

Conversioni implicite (promozioni)

`int` \Rightarrow `long` \Rightarrow `float` \Rightarrow `double`

Nelle conversioni implicite *non si ha perdita di informazione*

Conversioni implicite (promozioni)

`int` \Rightarrow `long` \Rightarrow `float` \Rightarrow `double`

Nelle conversioni implicite *non si ha perdita di informazione*

Le conversioni implicite vengono effettuate nelle espressioni aritmetiche che coinvolgono `int`, `long`, `float` e `double`

Esempio

```
int i;  
long L;  
float f;  
double d;
```

(1) `d = L * (f + d)`

(2) `f = i + d`

- Non tutti gli `int` e i `long` sono rappresentati da `float` e `double`

- Non tutti gli `int` e i `long` sono rappresentati da `float` e `double`
- Può esserci `perdita di precisione` dovuta alle approssimazioni

- Non tutti gli `int` e i `long` sono rappresentati da `float` e `double`
- Può esserci `perdita di precisione` dovuta alle approssimazioni

Esempio

```
int x = 2109876543; //x = 2109876543
```

- Non tutti gli `int` e i `long` sono rappresentati da `float` e `double`
- Può esserci `perdita di precisione` dovuta alle approssimazioni

Esempio

```
int x = 2109876543; //x = 2109876543
```

```
float y = x; //y = 2.10987648E9
```

- Non tutti gli `int` e i `long` sono rappresentati da `float` e `double`
- Può esserci `perdita di precisione` dovuta alle approssimazioni

Esempio

```
int x = 2109876543; //x = 2109876543
float y = x; //y = 2.10987648E9
int z = (int) y; //z = 2109876480
```

Conversioni esplicite (cast)

`double` \Rightarrow `float` \Rightarrow `long` \Rightarrow `int`

Conversioni esplicite (cast)

`double` ⇒ `float` ⇒ `long` ⇒ `int`

Può verificarsi perdita di informazione

Conversioni esplicite (cast)

`double` \Rightarrow `float` \Rightarrow `long` \Rightarrow `int`

Può verificarsi perdita di informazione

Esempi

```
int i;  
byte b;  
double x = 127.3;  
double y = 2.7;
```

```
(1) i = (int)x    // i = 127;
```

Conversioni esplicite (cast)

`double` \Rightarrow `float` \Rightarrow `long` \Rightarrow `int`

Può verificarsi perdita di informazione

Esempi

```
int i;  
byte b;  
double x = 127.3;  
double y = 2.7;
```

(1) `i = (int)x` `// i = 127;`

(2) `b = (byte)(x + y);` `// b = -126`

Conversioni esplicite (cast)

`double` \Rightarrow `float` \Rightarrow `long` \Rightarrow `int`

Può verificarsi perdita di informazione

Esempi

```
int i;  
byte b;  
double x = 127.3;  
double y = 2.7;
```

- (1) `i = (int)x` // `i = 127`;
- (2) `b = (byte)(x + y)`; // `b = -126`
- (3) `i = (int)(x + j)`; // `i = 130`

Conversioni esplicite (cast)

`double` \Rightarrow `float` \Rightarrow `long` \Rightarrow `int`

Può verificarsi perdita di informazione

Esempi

```
int i;  
byte b;  
double x = 127.3;  
double y = 2.7;
```

- (1) `i = (int)x` `// i = 127;`
- (2) `b = (byte)(x + y);` `// b = -126`
- (3) `i = (int)(x + j);` `// i = 130`
- (4) `i = (int)x + (int)j;` `// i = 129`

Esempio: media

```
int x, y, z;  
double media;  
...
```

Esempio: media

```
int x, y, z;  
double media;  
...
```

- `media = (x + y + z) / 3;`

non è il valore che ci interessa

Esempio: media

```
int x, y, z;  
double media;  
...
```

- `media = (x + y + z) / 3;`

non è il valore che ci interessa

- **Soluzione 1**

```
media = (x + y + z) / 3.0;
```

Esempio: media

```
int x, y, z;  
double media;  
...
```

- `media = (x + y + z) / 3;`

non è il valore che ci interessa

- **Soluzione 1**

```
media = (x + y + z) / 3.0;
```

- **Soluzione 2**

```
media = (double)(x + y + z) / 3;
```

Conversioni implicite al tipo String

Se almeno un argomento di `+` è un riferimento a `String`

Conversioni implicite al tipo String

Se almeno un argomento di + è un riferimento a `String`



+ rappresenta la **concatenazione** di stringhe

Conversioni implicite al tipo String

Se almeno un argomento di `+` è un riferimento a `String`



`+` rappresenta la **concatenazione** di stringhe

Se uno degli operandi dell'operatore di **concatenazione** è un valore numerico

Conversioni implicite al tipo String

Se almeno un argomento di `+` è un riferimento a `String`



`+` rappresenta la **concatenazione** di stringhe

Se uno degli operandi dell'operatore di **concatenazione** è un valore numerico



viene convertito in un oggetto di tipo `String` prima di effettuare la concatenazione

Conversioni implicite al tipo String

Se almeno un argomento di `+` è un riferimento a `String`



`+` rappresenta la **concatenazione di stringhe**

Se uno degli operandi dell'operatore di **concatenazione** è un valore numerico



viene convertito in un oggetto di tipo `String` prima di effettuare la concatenazione

```
int i = 1;  
double pi = 3.14;
```

```
(1) out.print("La somma vale ");  
    out.println(i + pi);
```

Conversioni implicite al tipo String

Se almeno un argomento di `+` è un riferimento a `String`



`+` rappresenta la **concatenazione di stringhe**

Se uno degli operandi dell'operatore di **concatenazione** è un valore numerico



viene convertito in un oggetto di tipo `String` prima di effettuare la concatenazione

```
int i = 1;  
double pi = 3.14;
```

- (1)

```
out.print("La somma vale ");  
out.println(i + pi);
```
- (2)

```
out.println("La somma vale " + i + pi);
```

Conversioni implicite al tipo String

Se almeno un argomento di `+` è un riferimento a `String`



`+` rappresenta la **concatenazione di stringhe**

Se uno degli operandi dell'operatore di **concatenazione** è un valore numerico



viene convertito in un oggetto di tipo `String` prima di effettuare la concatenazione

```
int i = 1;  
double pi = 3.14;
```

- (1)

```
out.print("La somma vale ");  
out.println(i + pi);
```
- (2)

```
out.println("La somma vale " + i + pi);
```
- (3)

```
out.println("La somma vale " + (i + pi));
```

Conversioni implicite al tipo String

Quando uno degli operandi dell'operatore di concatenazione è un **referimento**

Conversioni implicite al tipo String

Quando uno degli operandi dell'operatore di concatenazione è un **referimento**



viene prodotta una stringa richiamando il metodo **toString** dell'oggetto riferito

Conversioni implicite al tipo String

Quando uno degli operandi dell'operatore di concatenazione è un **riferimento**



viene prodotta una stringa richiamando il metodo **toString** dell'oggetto riferito

```
String s1, s2;  
Frazione f;  
...  
s1 = s2 + f; //equivalente a s1 = s2 + f.toString();
```

Conversioni implicite al tipo String

Quando uno degli operandi dell'operatore di concatenazione è un **riferimento**



viene prodotta una stringa richiamando il metodo **toString** dell'oggetto riferito

```
String s1, s2;  
Frazione f;  
...  
s1 = s2 + f; //equivalente a s1 = s2 + f.toString();
```

Ogni classe è dotata di un metodo **toString**, anche se non esplicitamente definito dal programmatore.

Il tipo char

Rappresenta caratteri [Unicode](#)

Il tipo char

Rappresenta caratteri [Unicode](#)

Letterali

- Singoli caratteri racchiusi fra apici singoli (')
- Sequenze di escape

Il tipo char

Rappresenta caratteri [Unicode](#)

Letterali

- Singoli caratteri racchiusi fra apici singoli (')
- Sequenze di escape

Esempio

```
char c;  
c = '\u0041';    //rappresentazione esadecimale  
                 //del carattere A  
  
c = '\t';        //carattere di tabulazione  
  
c = 'A';
```

Operatori di confronto

È possibile confrontare due caratteri utilizzando gli operatori di confronto. L'ordine è stabilito dalla posizione del carattere nella tabella Unicode.

Operatori di confronto

È possibile confrontare due caratteri utilizzando gli operatori di confronto. L'ordine è stabilito dalla posizione del carattere nella tabella Unicode.

- Nella tabella Unicode i caratteri alfabetici compaiono nell'ordine alfabetico

```
'a' < 'b' //true  
'b' < 'c' //true  
...  
c >= 'a' && c <='z'
```

- Le maiuscole precedono le minuscole

```
'Z' < 'a' //true  
'A' < 'B' //true  
...  
c >= 'A' && c <='Z'
```

Sequenze di escape

Rappresentano caratteri speciali

Sequenza di escape	Unicode	Significato
<code>\b</code>	<code>\u0008</code>	backspace BS
<code>\t</code>	<code>\u0009</code>	horizontal tab HT
<code>\n</code>	<code>\u000a</code>	linefeed LF
<code>\f</code>	<code>\u000c</code>	form feed FF
<code>\r</code>	<code>\u000d</code>	carriage return CR
<code>\"</code>	<code>\u0022</code>	double quote
<code>\'</code>	<code>\u0027</code>	single quote '
<code>\\</code>	<code>\u005c</code>	backslash

char come tipo intero

- In Java anche il tipo `char` è un tipo intero (interi senza segno su 16 bit).

char come tipo intero

- In Java anche il tipo `char` è un tipo intero (interi senza segno su 16 bit).
- È possibile utilizzare gli operatori disponibili per i tipi interi.

char come tipo intero

- In Java anche il tipo `char` è un tipo intero (interi senza segno su 16 bit).
- È possibile utilizzare gli operatori disponibili per i tipi interi.
- **Conversione implicita** di `char` a `int`

char come tipo intero

- In Java anche il tipo **char** è un tipo intero (interi senza segno su 16 bit).
- È possibile utilizzare gli operatori disponibili per i tipi interi.
- **Conversione implicita** di char a int

```
for(char c = 'a'; c <= 'z'; c++)  
    out.println(c);
```

char come tipo intero

- In Java anche il tipo **char** è un tipo intero (interi senza segno su 16 bit).
- È possibile utilizzare gli operatori disponibili per i tipi interi.
- **Conversione implicita** di char a int

```
for(char c = 'a'; c <= 'z'; c++)  
    out.println(c);
```

Conversioni

```
char c = 'a';  
int i;  
  
i = c;  
c = i; //NO
```

char come tipo intero

- In Java anche il tipo **char** è un tipo intero (interi senza segno su 16 bit).
- È possibile utilizzare gli operatori disponibili per i tipi interi.
- **Conversione implicita** di char a int

```
for(char c = 'a'; c <= 'z'; c++)  
    out.println(c);
```

Conversioni

```
char c = 'a';    char c = 'a';  
int i;          byte b;  
  
i = c;          b = c; //NO  
c = i; //NO     c = b; //NO
```

char come tipo intero

- In Java anche il tipo **char** è un tipo intero (interi senza segno su 16 bit).
- È possibile utilizzare gli operatori disponibili per i tipi interi.
- **Conversione implicita** di char a int

```
for(char c = 'a'; c <= 'z'; c++)  
    out.println(c);
```

Conversioni

char c = 'a';	char c = 'a';	char c = 'a';
int i;	byte b;	short s;
i = c;	b = c; //NO	s = c; //NO
c = i; //NO	c = b; //NO	c = s; //NO

Sommario: Tipi primitivi e tipi enumerativi

1 Espressioni

- Operatori di assegnamento e incremento ed effetti collaterali
- Precedenza e associatività degli operatori
- Lazy evaluation
- Tipi numerici interi
- Conversioni di tipo: promozioni e cast
- Tipi numerici in virgola mobile
- Il tipo char

2 Tipi enumerativi

3 L'istruzione switch

4 Metodi e campi statici

5 Classi involucro

Tipi enumerativi

- Sono definiti e documentati in modo simile alle classi

Tipi enumerativi

- Sono definiti e documentati in modo simile alle classi
- I possibili valori del tipo **sono stabiliti all'atto della definizione del tipo** (all'atto dell'implementazione)

Tipi enumerativi

- Sono definiti e documentati in modo simile alle classi
- I possibili valori del tipo **sono stabiliti all'atto della definizione del tipo** (all'atto dell'implementazione)
- Una variabile di tipo enumerativo viene **definita** allo stesso modo delle usuali variabili riferimento

Tipi enumerativi

- Sono definiti e documentati in modo simile alle classi
- I possibili valori del tipo **sono stabiliti all'atto della definizione del tipo** (all'atto dell'implementazione)
- Una variabile di tipo enumerativo viene **definita** allo stesso modo delle usuali variabili riferimento

Il tipo enumerativo MeseDellAnno

I suoi possibili valori sono, nell'ordine:

```
GENNAIO FEBBRARIO MARZO APRILE MAGGIO GIUGNO LUGLIO AGOSTO  
SETTEMBRE OTTOBRE NOVEMBRE DICEMBRE
```

Tipi enumerativi

- Sono definiti e documentati in modo simile alle classi
- I possibili valori del tipo **sono stabiliti all'atto della definizione del tipo** (all'atto dell'implementazione)
- Una variabile di tipo enumerativo viene **definita** allo stesso modo delle usuali variabili riferimento

Il tipo enumerativo MeseDellAnno

I suoi possibili valori sono, nell'ordine:

```
GENNAIO FEBBRARIO MARZO APRILE MAGGIO GIUGNO LUGLIO AGOSTO  
SETTEMBRE OTTOBRE NOVEMBRE DICEMBRE
```

```
MeseDellAnno mese;
```

Utilizzare i valori di un tipo enumerativo

tipo_enumerativo.valore

Utilizzare i valori di un tipo enumerativo

tipo_enumerativo.valore

```
mese = MeseDellAnno.APRILE
```

Utilizzare i valori di un tipo enumerativo

```
tipo_enumerativo.valore
```

```
mese = MeseDellAnno.APRILE
```

- Gli enumerativi sono particolari **classi** (parola riservata **enum** invece di **class**)

Utilizzare i valori di un tipo enumerativo

tipo_enumerativo.valore

```
mese = MeseDellAnno.APRILE
```

- Gli enumerativi sono particolari **classi** (parola riservata **enum** invece di **class**)
- I valori di tipo enumerativo sono **riferimenti a oggetti**

Utilizzare i valori di un tipo enumerativo

```
tipo_enumerativo.valore
```

```
mese = MeseDellAnno.APRILE
```

- Gli enumerativi sono particolari **classi** (parola riservata **enum** invece di **class**)
- I valori di tipo enumerativo sono **riferimenti a oggetti**
- Le variabili di tipo enumerativo sono **variabili riferimento**

Utilizzare i valori di un tipo enumerativo

```
tipo_enumerativo.valore
```

```
mese = MeseDellAnno.APRILE
```

- Gli enumerativi sono particolari **classi** (parola riservata **enum** invece di **class**)
- I valori di tipo enumerativo sono **riferimenti a oggetti**
- Le variabili di tipo enumerativo sono **variabili riferimento**
- È possibile assegnare **null** a una variabile di tipo enumerativo

Utilizzare i valori di un tipo enumerativo

```
tipo_enumerativo.valore
```

```
mese = MeseDellAnno.APRILE
```

- Gli enumerativi sono particolari **classi** (parola riservata **enum** invece di **class**)
- I valori di tipo enumerativo sono **riferimenti a oggetti**
- Le variabili di tipo enumerativo sono **variabili riferimento**
- È possibile assegnare **null** a una variabile di tipo enumerativo
- **Non si possono costruire istanze del tipo enumerativo**: si possono solo utilizzare quelle disponibili tramite le costanti

Come tutti gli oggetti, mettono a disposizione `toString()`

Come tutti gli oggetti, mettono a disposizione `toString()`

Comuni a tutti gli enumerativi

- `public String name()`
Restituisce il nome della costante enumerativa.

Come tutti gli oggetti, mettono a disposizione `toString()`

Comuni a tutti gli enumerativi

- `public String name()`
Restituisce il nome della costante enumerativa.
- `public int ordinal()`
Restituisce il numero ordinale del valore enumerativo che esegue il metodo, cioè restituisce il numero che identifica la posizione del valore enumerativo all'interno della sequenza dei valori previsti per il tipo enumerativo. Il primo valore del tipo enumerativo ha convenzionalmente ordinale 0.

```
MeseDellAnno mese = MeseDellAnno.APRILE;  
  
out.println(mese.toString());    // "Aprile"  
out.println(mese.ordinal());     // 3  
out.println(mese.name());       // "APRILE"
```

Uno specifico tipo enumerativo può **fornire metodi propri**

Uno specifico tipo enumerativo può **fornire metodi propri**

Metodi propri di MeseDellAnno

- `public MeseDellAnno successivo()`
Restituisce il valore dell'enumerativo che rappresenta il mese successivo a quello che esegue il metodo.

Uno specifico tipo enumerativo può **fornire metodi propri**

Metodi propri di MeseDellAnno

- `public MeseDellAnno successivo()`
Restituisce il valore dell'enumerativo che rappresenta il mese successivo a quello che esegue il metodo.
- `public MeseDellAnno precedente()`
Restituisce il valore dell'enumerativo che rappresenta il mese precedente a quello che esegue il metodo.

Metodi propri di MeseDellAnno

- `public int numeroGiorni()`

Restituisce il numero dei giorni del mese (28 se l'oggetto che esegue il metodo è FEBBRAIO).

Metodi propri di MeseDellAnno

- `public int numeroGiorni()`

Restituisce il numero dei giorni del mese (28 se l'oggetto che esegue il metodo è FEBBRAIO).

- `public int numeroGiorni(int anno)`

Restituisce il numero dei giorni del mese, relativamente all'anno specificato come argomento. Pertanto, nel caso degli anni bisestili, per il mese di febbraio restituisce 29.

Metodi propri di MeseDellAnno

- `public int numeroGiorni()`
Restituisce il numero dei giorni del mese (28 se l'oggetto che esegue il metodo è FEBBRAIO).
- `public int numeroGiorni(int anno)`
Restituisce il numero dei giorni del mese, relativamente all'anno specificato come argomento. Pertanto, nel caso degli anni bisestili, per il mese di febbraio restituisce 29.
- `public int numeroGiorni(boolean bisestile)`
Restituisce il numero dei giorni del mese; nel caso di febbraio se l'argomento è `true` e restituisce 29, se è `false` restituisce 28.

Sommario: Tipi primitivi e tipi enumerativi

1 Espressioni

- Operatori di assegnamento e incremento ed effetti collaterali
- Precedenza e associatività degli operatori
- Lazy evaluation
- Tipi numerici interi
- Conversioni di tipo: promozioni e cast
- Tipi numerici in virgola mobile
- Il tipo char

2 Tipi enumerativi

3 L'istruzione `switch`

4 Metodi e campi statici

5 Classi involucro

L'istruzione switch: sintassi

```
switch (espressione) {  
  case val1:  
    ist1;  
  case val2:  
    ist2;  
    ...  
  case valN:  
    istN;  
  default:  
    ist;  
}
```

L'istruzione switch: sintassi

- *espressione* (selettore)
di tipo char, byte, short o int o
enumerativo

```
switch (espressione) {  
    case val1:  
        ist1;  
    case val2:  
        ist2;  
    ...  
    case valN:  
        istN;  
    default:  
        ist;  
}
```

L'istruzione switch: sintassi

```
switch (espressione) {  
  case val1:  
    ist1;  
  case val2:  
    ist2;  
    ...  
  case valN:  
    istN;  
  default:  
    ist;  
}
```

- *espressione* (selettore)
di tipo char, byte, short o int o
enumerativo
- *val1, ..., valN* (etichette)
espressioni costanti assegnabili al tipo del
selettore

L'istruzione switch: sintassi

```
switch (espressione) {  
  case val1:  
    ist1;  
  case val2:  
    ist2;  
    ...  
  case valN:  
    istN;  
  default:  
    ist;  
}
```

- *espressione* (**selettore**)
di tipo char, byte, short o int o
enumerativo
- *val1, ..., valN* (**etichette**)
espressioni costanti assegnabili al tipo del
selettore
- *ist1, ..., istN, ist*
istruzioni singole oppure sequenze di
istruzioni

L'istruzione switch: sintassi

```
switch (espressione) {  
  case val1:  
    ist1;  
  case val2:  
    ist2;  
    ...  
  case valN:  
    istN;  
  default:  
    ist;  
}
```

- *espressione* (**selettore**)
di tipo char, byte, short o int o enumerativo
- *val1, ..., valN* (**etichette**)
espressioni costanti assegnabili al tipo del selettore
- *ist1, ..., istN, ist*
istruzioni singole oppure sequenze di istruzioni
- il caso **default** è opzionale (e può trovarsi in qualsiasi posizione)

L'istruzione switch: esecuzione

```
switch (espressione) {  
  case val1:  
    ist1;  
  case val2:  
    ist2;  
    ...  
  case valN:  
    istN;  
  default:  
    ist;  
}
```

L'istruzione switch: esecuzione

(1) Viene valutato il **selettore**

```
switch (espressione) {  
  case val1:  
    ist1;  
  case val2:  
    ist2;  
    ...  
  case valN:  
    istN;  
  default:  
    ist;  
}
```

L'istruzione switch: esecuzione

- (1) Viene valutato il **selettore**
- (2) Se è uguale a *valK* si inizia a eseguire a partire da *istK*

```
switch (espressione) {  
  case val1:  
    ist1;  
  case val2:  
    ist2;  
    ...  
  case valN:  
    istN;  
  default:  
    ist;  
}
```

L'istruzione switch: esecuzione

- (1) Viene valutato il **selettore**
- (2) Se è uguale a *valK* si inizia a eseguire a partire da *istK*
- (3) Se non c'è un'etichetta uguale al valore del selettore, ma è presente l'etichetta **default**, si inizia a eseguire dall'istruzione *ist*

```
switch (espressione) {  
  case val1:  
    ist1;  
  case val2:  
    ist2;  
    ...  
  case valN:  
    istN;  
  default:  
    ist;  
}
```

L'istruzione switch: esecuzione

```
switch (espressione) {  
  case val1:  
    ist1;  
  case val2:  
    ist2;  
    ...  
  case valN:  
    istN;  
  default:  
    ist;  
}
```

- (1) Viene valutato il **selettore**
- (2) Se è uguale a *valK* si inizia a eseguire a partire da *istK*
- (3) Se non c'è un'etichetta uguale al valore del selettore, ma è presente l'etichetta **default**, si inizia a eseguire dall'istruzione *ist*
- (4) Se non c'è un'etichetta uguale al valore del selettore e non c'è l'etichetta **default**, si passa a eseguire il codice che si trova dopo il blocco switch

L'istruzione switch: esecuzione

```
switch (espressione) {  
  case val1:  
    ist1;  
  case val2:  
    ist2;  
    ...  
  case valN:  
    istN;  
  default:  
    ist;  
}
```

- (1) Viene valutato il **selettore**
- (2) Se è uguale a *valK* si inizia a eseguire a partire da *istK*
- (3) Se non c'è un'etichetta uguale al valore del selettore, ma è presente l'etichetta **default**, si inizia a eseguire dall'istruzione *ist*
- (4) Se non c'è un'etichetta uguale al valore del selettore e non c'è l'etichetta **default**, si passa a eseguire il codice che si trova dopo il blocco `switch`
- (5) Se si incontra un'istruzione **break**, si passa a eseguire il codice che si trova dopo il blocco `switch`

Sommario: Tipi primitivi e tipi enumerativi

1 Espressioni

- Operatori di assegnamento e incremento ed effetti collaterali
- Precedenza e associatività degli operatori
- Lazy evaluation
- Tipi numerici interi
- Conversioni di tipo: promozioni e cast
- Tipi numerici in virgola mobile
- Il tipo char

2 Tipi enumerativi

3 L'istruzione switch

4 Metodi e campi statici

5 Classi involucro

Sommario: Tipi primitivi e tipi enumerativi

1 Espressioni

- Operatori di assegnamento e incremento ed effetti collaterali
- Precedenza e associatività degli operatori
- Lazy evaluation
- Tipi numerici interi
- Conversioni di tipo: promozioni e cast
- Tipi numerici in virgola mobile
- Il tipo char

2 Tipi enumerativi

3 L'istruzione switch

4 Metodi e campi statici

5 Classi involucro

Servizi forniti dalle classi anziché dai singoli oggetti

Servizi forniti dalle classi anziché dai singoli oggetti

Usati per:

- Costruire oggetti della classe stessa a partire da oggetti o valori di altro tipo

Servizi forniti dalle classi anziché dai singoli oggetti

Usati per:

- Costruire oggetti della classe stessa a partire da oggetti o valori di altro tipo

Conversione di un `int` nella stringa che lo rappresenta

Servizi forniti dalle classi anziché dai singoli oggetti

Usati per:

- Costruire oggetti della classe stessa a partire da oggetti o valori di altro tipo

Conversione di un `int` nella stringa che lo rappresenta

- Fornire operazioni utili su oggetti o su tipi primitivi

Metodi statici

Servizi forniti dalle classi anziché dai singoli oggetti

Usati per:

- Costruire oggetti della classe stessa a partire da oggetti o valori di altro tipo

Conversione di un `int` nella stringa che lo rappresenta

- Fornire operazioni utili su oggetti o su tipi primitivi

La classe `Math` del package `java.lang` mette a disposizione metodi **statici** per realizzare le operazioni numeriche di base

Servizi forniti dalle classi anziché dai singoli oggetti

Usati per:

- Costruire oggetti della classe stessa a partire da oggetti o valori di altro tipo

Conversione di un `int` nella stringa che lo rappresenta

- Fornire operazioni utili su oggetti o su tipi primitivi

La classe `Math` del package `java.lang` mette a disposizione metodi **statici** per realizzare le operazioni numeriche di base

- Definire proprietà che influenzano il comportamento di tutti gli oggetti di una classe

Metodi statici

Servizi forniti dalle classi anziché dai singoli oggetti

Usati per:

- Costruire oggetti della classe stessa a partire da oggetti o valori di altro tipo

Conversione di un `int` nella stringa che lo rappresenta

- Fornire operazioni utili su oggetti o su tipi primitivi

La classe `Math` del package `java.lang` mette a disposizione metodi **statici** per realizzare le operazioni numeriche di base

- Definire proprietà che influenzano il comportamento di tutti gli oggetti di una classe

Stabilire il formato di rappresentazione di una data

Riconoscibili dalla presenza della parola riservata `static` nell'intestazione.

Riconoscibili dalla presenza della parola riservata `static` nell'intestazione.

Classe String

```
public static String valueOf(int i)
```

Restituisce il riferimento alla stringa che rappresenta il valore dell'int fornito come argomento.

Riconoscibili dalla presenza della parola riservata `static` nell'intestazione.

Classe String

```
public static String valueOf(int i)
```

Restituisce il riferimento alla stringa che rappresenta il valore dell'int fornito come argomento.

Sono invocati utilizzando come destinatario del messaggio **il nome della classe** anziché il riferimento a un oggetto.

Invocazione di metodo statico

```
nome_classe.nome_metodo(lista_argumenti)
```

```
java.lang.Math
```

- `public static double cos(double a)`
- `public static double log(double a)`
- `public static double log10(double a)`

java.lang.Math

- `public static double cos(double a)`
- `public static double log(double a)`
- `public static double log10(double a)`

java.lang.Integer

- `public static int parseInt(String s)`

Restituisce il valore di tipo `int` corrispondente alla sequenza di cifre decimali contenuta nella stringa fornita come argomento.

Se la stringa fornita come argomento non rappresenta un numero intero si verifica un errore in fase di esecuzione.

java.lang.Math

- `public static double cos(double a)`
- `public static double log(double a)`
- `public static double log10(double a)`

java.lang.Integer

- `public static int parseInt(String s)`
Restituisce il valore di tipo `int` corrispondente alla sequenza di cifre decimali contenuta nella stringa fornita come argomento.
Se la stringa fornita come argomento non rappresenta un numero intero si verifica un errore in fase di esecuzione.
- `public static String toBinaryString(int i)`

Sono **variabili** immutabili definite a livello di una classe, utilizzate in genere per costanti utili.

Sono **variabili** imm modificabili definite a livello di una classe, utilizzate in genere per costanti utili.

```
java.lang.Math
```

- `static final double E`

La base del logaritmo naturale.

Campi statici

Sono **variabili** immutabili definite a livello di una classe, utilizzate in genere per costanti utili.

```
java.lang.Math
```

- `static final double E`
La base del logaritmo naturale.
- `static final double PI`
La costante π .

Campi statici

Sono **variabili** imm modificabili definite a livello di una classe, utilizzate in genere per costanti utili.

`java.lang.Math`

- `static final double E`

La base del logaritmo naturale.

- `static final double PI`

La costante π .

Accesso a un campo statico

`nome_classe.nome_campo`

Sommario: Tipi primitivi e tipi enumerativi

1 Espressioni

- Operatori di assegnamento e incremento ed effetti collaterali
- Precedenza e associatività degli operatori
- Lazy evaluation
- Tipi numerici interi
- Conversioni di tipo: promozioni e cast
- Tipi numerici in virgola mobile
- Il tipo char

2 Tipi enumerativi

3 L'istruzione switch

4 Metodi e campi statici

5 Classi involucro

- L'introduzione dei tipi primitivi nel linguaggio Java è dovuta principalmente a questioni di efficienza.

- L'introduzione dei tipi primitivi nel linguaggio Java è dovuta principalmente a questioni di efficienza.
- Può risultare utile rappresentare dati di tipi primitivi sotto forma di oggetti.

- L'introduzione dei tipi primitivi nel linguaggio Java è dovuta principalmente a questioni di efficienza.
- Può risultare utile rappresentare dati di tipi primitivi sotto forma di oggetti.
- In `java.lang` è prevista una classe `involucro` per ogni tipo primitivo:

<code>byte</code>	<code>Byte</code>	<code>float</code>	<code>Float</code>
<code>short</code>	<code>Short</code>	<code>double</code>	<code>Double</code>
<code>int</code>	<code>Integer</code>	<code>boolean</code>	<code>Boolean</code>
<code>long</code>	<code>Long</code>	<code>char</code>	<code>Character</code>

Campi statici

- `static final int MIN_VALUE`
Minimo valore intero rappresentabile tramite un `int`.

Campi statici

- `static final int MIN_VALUE`
Minimo valore intero rappresentabile tramite un `int`.
- `static final int MAX_VALUE`
Massimo valore intero rappresentabile tramite un `int`.

La classe involucro Integer

Campi statici

- `static final int MIN_VALUE`
Minimo valore intero rappresentabile tramite un `int`.
- `static final int MAX_VALUE`
Massimo valore intero rappresentabile tramite un `int`.

Costruttori

- `public Integer(int x)`
Costruisce un nuovo oggetto che rappresenta il numero intero fornito come argomento.

La classe involucro Integer

Campi statici

- `static final int MIN_VALUE`
Minimo valore intero rappresentabile tramite un `int`.
- `static final int MAX_VALUE`
Massimo valore intero rappresentabile tramite un `int`.

Costruttori

- `public Integer(int x)`
Costruisce un nuovo oggetto che rappresenta il numero intero fornito come argomento.
- `public Integer(String str)`
Costruisce un nuovo oggetto che rappresenta il numero intero fornito sotto forma di stringa tramite l'argomento. Se la stringa non rappresenta un numero intero si verifica un errore.

- `public int compareTo(Integer altro)`

Confronta l'intero rappresentato dall'oggetto che esegue il metodo con quello fornito tramite il parametro. Restituisce zero se i due valori coincidono, un valore minore di zero se il valore rappresentato dall'oggetto è minore di quello fornito tramite il parametro, un valore maggiore di zero altrimenti.

- `public int compareTo(Integer altro)`
Confronta l'intero rappresentato dall'oggetto che esegue il metodo con quello fornito tramite il parametro. Restituisce zero se i due valori coincidono, un valore minore di zero se il valore rappresentato dall'oggetto è minore di quello fornito tramite il parametro, un valore maggiore di zero altrimenti.
- `public int intValue()`
Restituisce un `int` uguale al valore rappresentato dall'oggetto.

- `public int compareTo(Integer altro)`
Confronta l'intero rappresentato dall'oggetto che esegue il metodo con quello fornito tramite il parametro. Restituisce zero se i due valori coincidono, un valore minore di zero se il valore rappresentato dall'oggetto è minore di quello fornito tramite il parametro, un valore maggiore di zero altrimenti.
- `public int intValue()`
Restituisce un `int` uguale al valore rappresentato dall'oggetto.
- `public long longValue()`
Restituisce un `long` uguale al valore rappresentato dall'oggetto.

- `public static int parseInt(String s)`

Restituisce un `int` uguale al valore rappresentato dalla stringa fornita come argomento. Ad esempio, se la stringa è "1234", restituisce il valore 1234.

Nel caso la stringa non rappresenti un intero, si verifica un errore in fase di esecuzione.

Integer: metodi statici

- `public static int parseInt(String s)`

Restituisce un `int` uguale al valore rappresentato dalla stringa fornita come argomento. Ad esempio, se la stringa è "1234", restituisce il valore 1234.

Nel caso la stringa non rappresenti un intero, si verifica un errore in fase di esecuzione.

- `public static Integer valueOf(String s)`

Restituisce un oggetto di tipo `Integer` che rappresenta il valore intero rappresentato dalla stringa fornita come argomento.

Nel caso la stringa non rappresenti un intero, si verifica un errore in fase di esecuzione.

Costruttore

- `public Character(char value)`
Costruisce l'oggetto che rappresenta il carattere specificato come argomento.

La classe involucro Character

Costruttore

- `public Character(char value)`

Costruisce l'oggetto che rappresenta il carattere specificato come argomento.

Metodi statici

- `public static boolean isDigit(char ch)`

Determina se il carattere specificato come argomento è una cifra. Il metodo restituisce `true` se il carattere è una cifra, e `false` in caso contrario.

Costruttore

- `public Character(char value)`

Costruisce l'oggetto che rappresenta il carattere specificato come argomento.

Metodi statici

- `public static boolean isDigit(char ch)`

Determina se il carattere specificato come argomento è una cifra. Il metodo restituisce `true` se il carattere è una cifra, e `false` in caso contrario.

- `public static boolean isLetter(char ch)`

Determina se il carattere specificato come argomento è una lettera. Il metodo restituisce `true` se il carattere è una lettera, e `false` in caso contrario.

- `public static boolean isLowerCase(char ch)`

Determina se il carattere specificato come argomento è una lettera minuscola. Il metodo restituisce `true` se il carattere è una lettera minuscola, e `false` in caso contrario.

La classe involucro Character

- `public static boolean isLowerCase(char ch)`

Determina se il carattere specificato come argomento è una lettera minuscola. Il metodo restituisce `true` se il carattere è una lettera minuscola, e `false` in caso contrario.

- `public static boolean isUpperCase(char ch)`

Determina se il carattere specificato come argomento è una lettera maiuscola. Il metodo restituisce `true` se il carattere è una lettera maiuscola, e `false` in caso contrario.

- `public static boolean isLowerCase(char ch)`

Determina se il carattere specificato come argomento è una lettera minuscola. Il metodo restituisce `true` se il carattere è una lettera minuscola, e `false` in caso contrario.

- `public static boolean isUpperCase(char ch)`

Determina se il carattere specificato come argomento è una lettera maiuscola. Il metodo restituisce `true` se il carattere è una lettera maiuscola, e `false` in caso contrario.

- `public static boolean isLetterOrDigit(char ch)`

Determina se il carattere specificato come argomento è una lettera o una cifra. Il metodo restituisce `true` se il carattere è una lettera o una cifra, e `false` in caso contrario.

- `public static char toLowerCase(char ch)`
Converte il carattere specificato come argomento nel corrispondente carattere minuscolo.

- `public static char toLowerCase(char ch)`
Converte il carattere specificato come argomento nel corrispondente carattere minuscolo.
- `public static char toUpperCase(char ch)`
Converte il carattere specificato come argomento nel corrispondente carattere maiuscolo.