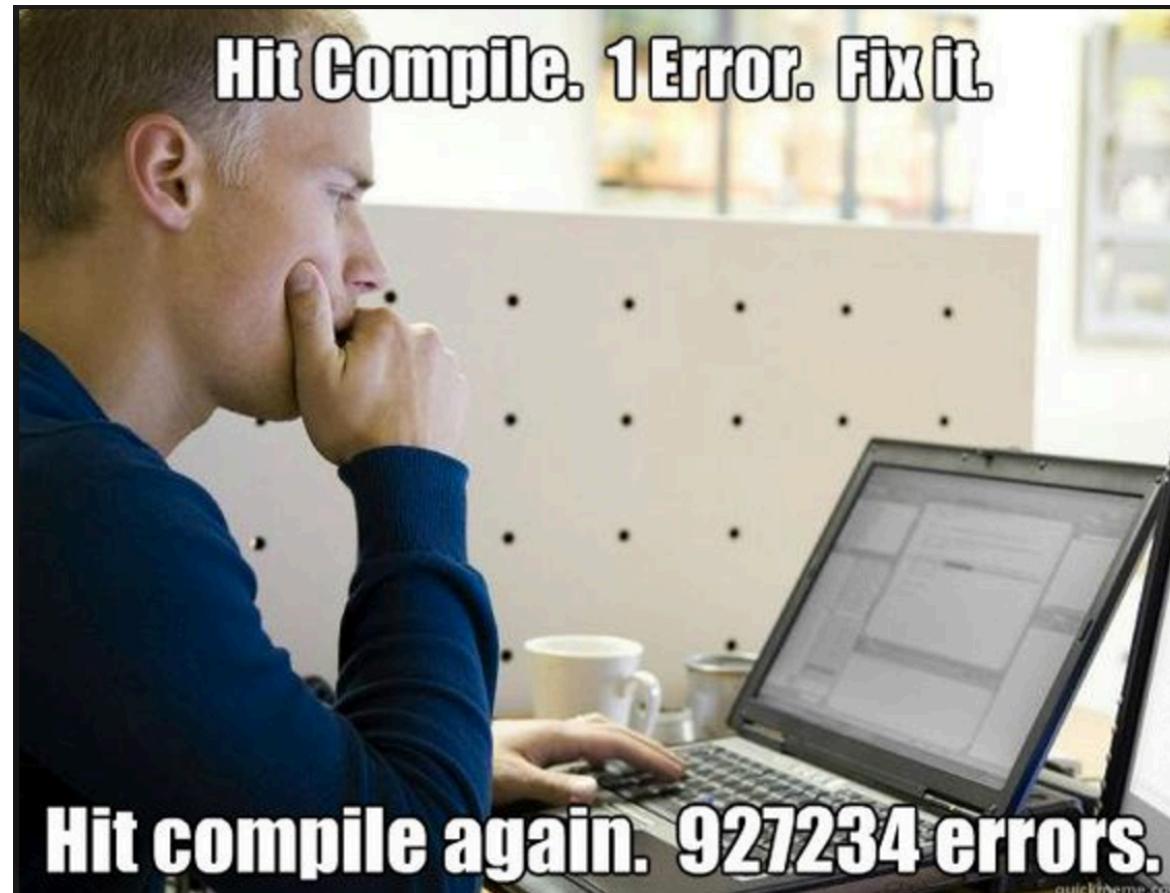




Errori di programmazione



Errori di programmazione

```
System.out.println("Hello, World!");
```

- L'attività di programmazione, come ogni altra **attività di progettazione**, è soggetta a errori di vario tipo

- **errori di sintassi o di compilazione** (lessicali/di sintassi)

```
System.aut.println("Hello, World!");
```

```
System.out.println("Hello, World!");
```

- **errori logici (di semantica) o di esecuzione (di runtime)**

```
System.out.println("Hell, World!");
```

Errori di sintassi

```
System.aut.println("Hello, World!");
```

- In questo caso il compilatore riesce agevolmente a individuare e segnalare l'errore di sintassi, perché identifica il nome di un oggetto (**simbolo**) che non è stato definito (**aut**) e sul quale non è in grado di “**decidere**”

posizione
(numero di
riga)

posizione
(nella riga)

```
C:\>javac Hello.java
HelloTester.java:6: cannot find symbol
  symbol : variable aut
           location: class java.lang.System
    System.aut.println("Hello, World!");
                                ^
1 error
```

diagnosi

Errori di sintassi

virgolette
mancanti

```
System.out.println("Hello, World!");
```

- Questo è invece un caso più complesso: viene giustamente segnalato il **primo errore**, una stringa non terminata, e viene evidenziato il punto dove *inizia* la stringa

```
HelloTester.java:5: unclosed string literal
    System.out.println("Hello, World!");
                                         ^
HelloTester.java:5: ';' expected
    System.out.println("Hello, World!");
                                         ^
HelloTester.java:7: reached end of file while
parsing
}
^
3 errors
```

Errori di sintassi

- Viene però segnalato anche un **secondo errore**
 - il compilatore *si aspetta di trovare un ';' in corrispondenza della fine dell'istruzione*
 - *Il ';' in realtà c'è*, ma il compilatore l'ha inserito all'interno della stringa, cioè *ha prolungato la stringa fino al termine della riga*

```
HelloTester.java:5: unclosed string literal
    System.out.println("Hello, World!");
                           ^
HelloTester.java:5: ';' expected
    System.out.println("Hello, World!");
                           ^
HelloTester.java:7: reached end of file while
parsing
}
^
3 errors
```

Errori logici

manca un carattere

```
System.out.println("Hello, World!");
```

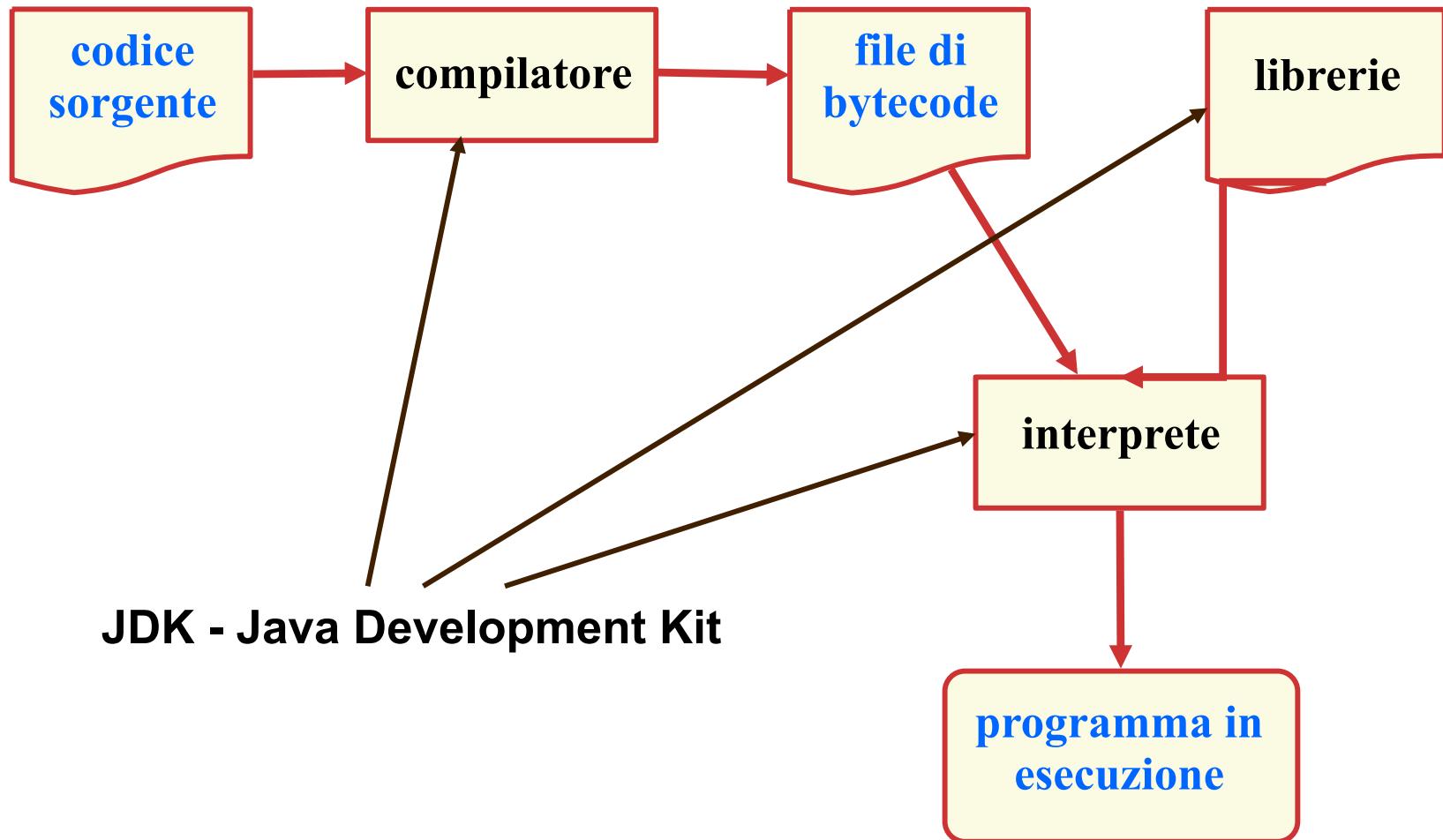
- Questo errore, invece, **non viene segnalato dal compilatore**, che non può sapere che cosa il programmatore abbia intenzione di far scrivere al programma sull'output standard
 - la compilazione va a buon fine**
 - si ha un errore durante l'**esecuzione** del programma, perché viene prodotto un output **diverso** dal previsto

```
C:\>java HelloTester  
Hello, World!
```

Altri... *errori*?

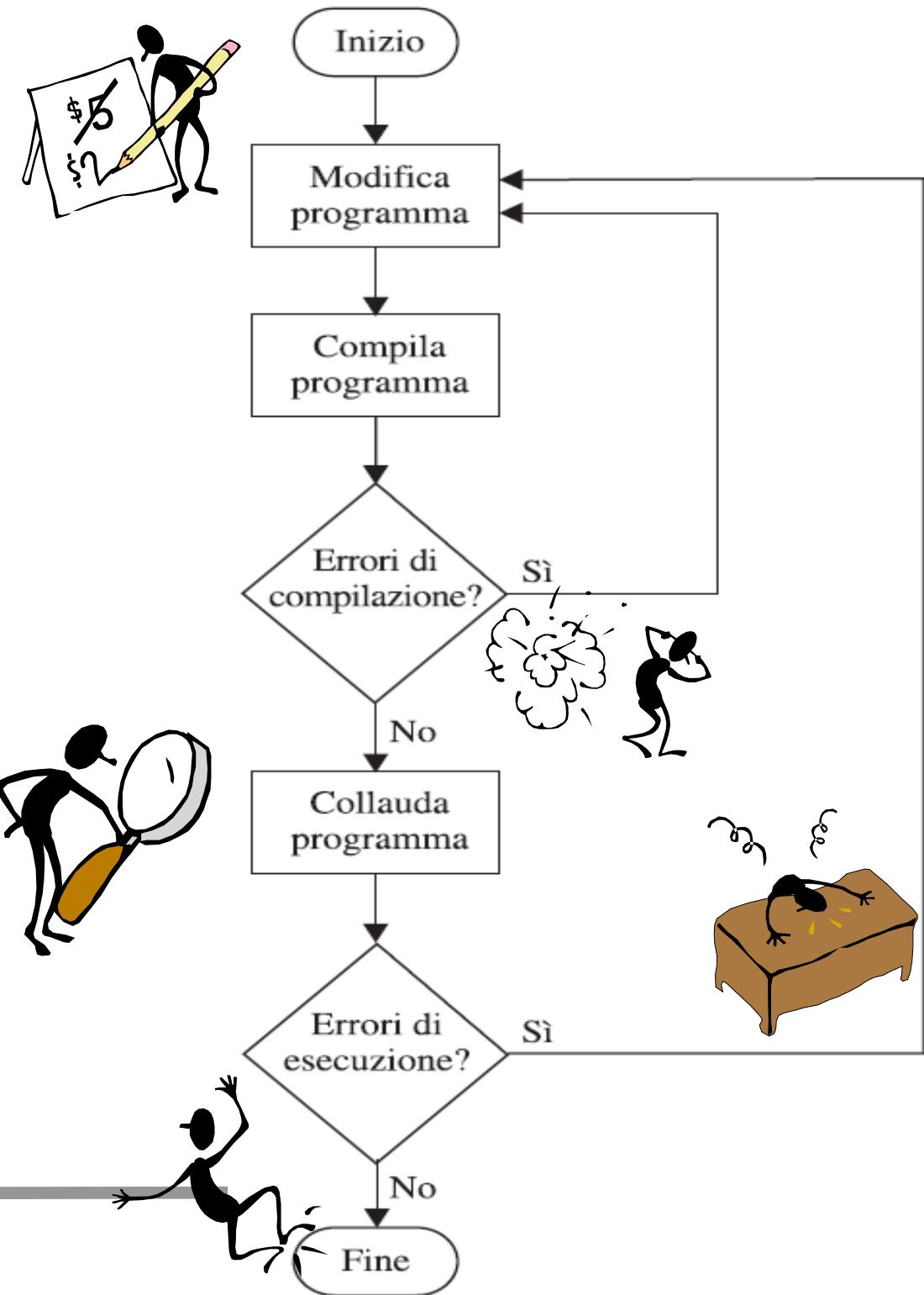
- e se provo a leggere un file che non esiste...?
- è un errore prevedibile?
- a volte si (errore del programmatore nello scrivere il nome del file)
- a volte no (file cancellato, il file è memorizzato in remoto e si perde la connessione)
- e se chiedo all'utente di inserire un numero e invece inserisce una stringa?
- ...

Il processo di programmazione in Java



E se qualcosa non funziona?

Il ciclo Modifica- Compila- Collauda



È tutto chiaro? ...

1. Se omettiamo i caratteri // dal file HelloTester.java senza eliminare la parte restante della riga otteniamo un errore di compilazione o di esecuzione?
2. Come si possono identificare gli errori logici di un programma?
3. Perché non posso collaudare un programma alla ricerca di errori di esecuzione se ci sono ancora errori di compilazione?

Notazione posizionale

- I numeri naturali sono definiti dal seguente insieme

$$N = \{0,1,2,\dots\}$$

- I numeri naturali che usiamo solitamente sono espressi nella **notazione posizionale** in **base decimale**
 - base decimale**: usiamo dieci cifre diverse (da 0 a 9)
 - notazione posizionale**: cifre uguali in posizioni diverse hanno significato diverso
 - hanno **peso diverso** pesano diversamente nella determinazione del valore del numero espresso)
 - il peso di una cifra è **uguale alla base** (10, in questo caso), elevata alla potenza pari alla posizione della cifra, posizione che si incrementa da destra a sinistra a partire da 0

$$434 = 4 \cdot 10^2 + 3 \cdot 10^1 + 4 \cdot 10^0$$



Notazione posizionale

- In generale con n cifre

$$a_{n-1}a_{n-2}\dots a_0 = a_{n-1}10^{n-1} + a_{n-2}10^{n-2} + \dots + a_010^0, a_k \in \{0,1,\dots,9\}$$

$$k = 0,1,\dots,n-1$$

- Oppure, se b è la base: $a_{n-1}a_{n-2}\dots a_0 = \sum_{k=0}^{n-1} a_k b^k$

- Anche i numeri razionali (frazioni di interi) si possono esprimere con notazione posizionale in base decimale
 - la **parte frazionaria** a destra del simbolo separatore (**punto, non virgola!**), si valuta con potenze negative

$$4.34 = 4 \cdot 10^0 + 3 \cdot 10^{-1} + 4 \cdot 10^{-2}$$

Notazione binaria

- I computer usano invece numeri rappresentati con notazione posizionale ***in base binaria***
 - la base binaria usa solo ***due cifre diverse***, 0 e 1
 - la conversione da base binaria a decimale è semplice

$$(1101)_2 = (1 \cdot 2^3 + 1 \cdot 2^2 + 0 \cdot 2^1 + 1 \cdot 2^0)_{10} = (13)_{10}$$

$$(1.101)_2 = (1 \cdot 2^0 + 1 \cdot 2^{-1} + 0 \cdot 2^{-2} + 1 \cdot 2^{-3})_{10} = (1.625)_{10}$$

- La rappresentazione binaria è più *naturale*, facile da manipolare per i computer
 - perché è meno complicato costruire circuiti logici che distinguono tra “acceso” e “spento”, piuttosto che fra ***dieci livelli diversi*** di tensione



Notazione binaria

- La conversione di un numero da base **decimale** a base **binaria** è, invece, più complessa
- Innanzitutto, la parte intera del numero va elaborata indipendentemente dalla eventuale parte frazionaria
 - la parte intera del numero decimale viene convertita nella parte intera del numero binario
 - la parte frazionaria del numero decimale viene convertita nella parte frazionaria del numero binario
 - la posizione del punto separatore rimane invariata

Numeri naturali: notazione binaria

- Per convertire ***la sola parte intera***, si divide il numero per 2, eliminando l'eventuale resto e continuando a dividere per 2 il quoziente ottenuto fino a quando non si ottiene quoziente uguale a 0
- Il numero binario si ottiene scrivendo ***la serie dei resti*** delle divisioni, ***iniziando dall'ultimo*** resto ottenuto
- **Attenzione:** non fermarsi quando si ottiene **quoziente 1**, ma proseguire fino a **0**
- **ESERCIZIO:** Come si dimostra?

100	/	2	=	50	resto	0
50	/	2	=	25	resto	0
25	/	2	=	12	resto	1
12	/	2	=	6	resto	0
6	/	2	=	3	resto	0
3	/	2	=	1	resto	1
1	/	2	=	0	resto	1

$$(100)_{10} = (1100100)_2$$

Numeri razionali: virgola fissa

- Per convertire la **sola parte frazionaria**, si moltiplica il numero per 2, sottraendo 1 dal risultato se è maggiore di 1 e continuando fino a quando non si ottiene **0** o un risultato già ottenuto in precedenza
 - Il numero binario si ottiene dalla serie delle
 - parti intere dei prodotti partendo dal primo
 - Se si ottiene un risultato già ottenuto in
 - precedenza, il numero sarà **periodico**
 - anche se non lo era in base decimale
- La rappresentazione completa si ottiene componendo la parte intera e la parte frazionaria:
 - Rappresentazione **in virgola fissa**: il separatore si trova sempre nello stesso punto rispetto alla sequenza di bit

$$\begin{array}{rcl} 0.35 \cdot 2 & = & 0.7 \\ 0.7 \cdot 2 & = & 1.4 \\ 0.4 \cdot 2 & = & 0.8 \\ 0.8 \cdot 2 & = & 1.6 \\ 0.6 \cdot 2 & = & 1.2 \\ 0.2 \cdot 2 & = & 0.4 \end{array}$$

$$(0.35)_{10} = (0.0\overline{10110})_2$$

$$(100.35)_{10} = (1100100.010\overline{110})_2$$



Rappresentazione di numeri naturali

- Con 8 cifre binarie (con 8 bit) si possono rappresentare 2^8 disposizioni, pari a $2^8=256$ numeri diversi:
 - $0_{10} = 0000\ 0000_2$
 - $1_{10} = 0000\ 0001_2$
 - $2_{10} = 0000\ 0010_2$
 - $3_{10} = 0000\ 0011_2$
 - $4_{10} = 0000\ 0100_2$
 - ...
 - $254_{10} = 1111\ 1110_2$
 - $255_{10} = 1111\ 1111_2 = 2^7 + 2^6 + 2^5 + 2^4 + 2^3 + 2^2 + 2^1 + 2^0$

In generale:

con una **rappresentazione a n bit**
si possono **rappresentare i 2^n**
numeri interi che sono compresi
nell'intervallo

$$[0, 2^n - 1] \cap \mathbb{Z}$$

Numeri interi relativi

- I numeri interi relativi sono l'insieme $Z = \{..., -2, -1, 0, +1, +2, ...\}$
 - la rappresentazione più naturale è quella detta **rappresentazione con modulo e segno**
 - il primo bit a sinistra (il **bit più significativo**) rappresenta il **segno**: 0 per il segno +, 1 per il segno -
 - **Definizione:** In un numero si dicono cifre **più significative** quelle che occupano le posizioni più a sinistra, **meno significative** quelle che occupano le posizioni più a destra
 - i rimanenti bit rappresentano il **modulo** del numero.
- $(101100)_2 = (-12)_{10}$ $(001100)_2 = (+12)_{10}$
- Se si usa una **rappresentazione a n bit**, si possono **rappresentare i $2^n - 1$ interi** compresi nell'intervallo
 - $[-(2^{n-1} - 1), 2^{n-1} - 1] \cap \mathbb{Z}$

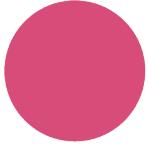
Rappresentazione in modulo/segno

- $+ 127_{10} = 0111\ 1111_2$
- $+ 126_{10} = 0111\ 1110_2$
- ... = ...
- $+ 001_{10} = 0000\ 0001_2$
- $+ 000_{10} = 0000\ 0000_2$ DUE RAPPRESENTAZIONI
- $- 000_{10} = 1000\ 0000_2$ DELLO ZERO
- $- 001_{10} = 1000\ 0001_2$
- ... = ...
- $- 126_{10} = 1111\ 1110_2$
- $- 127_{10} = 1111\ 1111_2$

Bit di segno +
(bit più significativo)

Bit di segno -
(bit più significativo)

In pratica questa rappresentazione non viene usata



Rappresentazione in modulo/segno

- **Problemi**

- C'è una doppia rappresentazione per lo zero (+0 e -0), per cui si "spreca" una configurazione
- L'algoritmo di addizione è complesso per numeri relativi rappresentati in modulo e segno

- **Addizione $S = A + B$**

- Se $\text{segno}(A) = \text{segno}(B)$
 - $\text{segno}(S) = \text{segno}(A)$, $|S| = (|A| + |B|)$
- altrimenti
 - se $|A| \geq |B|$
 - $\text{segno}(S) = \text{segno}(A)$, $|S| = (|A| - |B|)$
 - altrimenti
 - $\text{segno}(S) = \text{segno}(B)$, $|S| = (|B| - |A|)$

Complemento a due

- Una rappresentazione più efficiente è quella denominata **complemento a due**, così definita
 - dato un numero intero relativo
 - $a \in [-2^{n-1}, 2^{n-1} - 1] \cap \mathbb{Z}$
 - la sua rappresentazione in complemento a due
 - **se $a \geq 0$** : rappresentazione senza segno a **n bit** di a
 - **se $a < 0$** : rappresentazione senza segno a **n bit** di $(a+2^n)$
 - Perché il massimo numero rappresentabile in Java con una variabile di tipo **int** è, come vedremo, 2147483647?
 - Una variabile **int** è rappresentata in complemento a due con **32 bit**
 - Quindi il massimo numero è $2^{31} - 1 = 2147483647$

Complemento a due

- La sequenza di n bit $a_{n-1}a_{n-2} \dots a_1 a_0$

significa $-a_{n-1}2^{n-1} + a_{n-2}2^{n-2} + \dots + a_12^1 + a_02^0$

- $+ 127_{10} = 0111\ 1111_2$
- $+ 126_{10} = 0111\ 1110_2$
- $\dots = \dots$
- $+ 001_{10} = 0000\ 0001_2$
- $+ 000_{10} = 0000\ 0000_2$ RAPPRESENTAZIONE UNICA

-
- $- 001_{10} = 1111\ 1111_2$
 - $\dots = \dots$
 - $- 127_{10} = 1000\ 0001_2$
 - $- 128_{10} = 1000\ 0000_2$

Il bit più significativo
indica ancora il segno

DELLO ZERO

Con n=8 bit si rappresentano numeri da $-2^{n-1} = -128$ a $2^{n-1}-1 = 127$



Complemento a due

- Esempio: il numero **-13** in complemento a due **a 8 bit**
 - Ha la rappresentazione senza segno a 8 bit di
 - $-13 + 2^8 = -13 + 256 = 243$
 - Allora $(243)_{10} = (11110011)_2$ (verificare per esercizio)
 - Fortunatamente esiste un **procedimento più semplice**
 - “Inversione” in complemento a due
 - Scrivere la rappresentazione di **+13**: **00001101**
 - Scambiare gli 0 con gli 1 (“complemento a uno”):
 - **11110010**
 - Aggiungere 1 al risultato: **11110011**

Complemento a due

- Data una sequenza di bit che rappresentano un numero in binario, **dobbiamo conoscere il formato** in cui il numero è espresso prima di convertirlo in decimale
- Complemento a 2 a 8 bit
 - $1111\ 1111_2 = -2^7 + 2^6 + 2^5 + 2^4 + 2^3 + 2^2 + 2^1 + 2^0 = -1_{10}$
- Complemento a 2 a 16 bit
 - $0000\ 0000\ 1111\ 1111_2 = +2^7 + 2^6 + 2^5 + 2^4 + 2^3 + 2^2 + 2^1 + 2^0 = +255_{10}$
 - **Attenzione:** le cifre 0 a sinistra possono essere omesse
 - $0000\ 0000\ 1111\ 1111_2 = 1111\ 1111_2$
- Modulo e segno a 8 bit
 - $1111\ 1111_2 = -(+2^6 + 2^5 + 2^4 + 2^3 + 2^2 + 2^1 + 2^0) = -127_{10}$

Complemento a due

- Proprietà (facilmente dimostrabili)
 - il segno di un numero rappresentato in complemento a due è il bit più significativo (0 se positivo, 1 se negativo)
 - la parte restante della rappresentazione NON è il valore assoluto del numero
 - lo è soltanto per i numeri positivi
 - non ci sono più configurazioni “sprecate”
 - con n bit si rappresentano 2^n numeri diversi
 - per eseguire l'addizione di numeri rappresentati in complemento a due **si esegue semplicemente l'addizione binaria** delle rappresentazioni
 - al termine dell'addizione, NON bisogna considerare un eventuale riporto nella posizione n , cioè un'eventuale $(n+1)$ -esima cifra del risultato non ne fa parte

Somma in complemento a due

- Somma con le regole della somma in colonna
 - $0+0 = 0$ $0+1 = 1+0 = 1$ $1+1 = 0$ con **riporto** di 1

$$\begin{array}{r} \bullet \quad 0000\ 0101 + \quad \quad \quad +5 + \\ \bullet \quad 0000\ 0010 \quad \quad \quad \quad \quad +2 \\ \hline \bullet \quad 0000\ 0111 \quad \quad \quad \quad \quad +7 \end{array}$$

$$\begin{array}{r} 0000\ 0101 + \quad \quad \quad +5 + \\ 1111\ 1110 \quad \quad \quad \quad \quad -2 \\ \hline 1|0000\ 0011 \quad \quad \quad \quad \quad +3 \end{array}$$

- Un riporto a sinistra della cifra più significativa può essere ignorato **solo** a patto che si ottenga come risultato un numero rappresentabile con il n. di bit a disposizione

$$\begin{array}{r} \bullet \quad 0111\ 1111 +127+ \\ \bullet \quad 0000\ 0001 \quad \quad \quad 1 \\ \hline \bullet \quad 1000\ 0000 \quad -128 !! \end{array}$$

$$\begin{array}{r} 1000\ 0000 \quad -128 \\ 1111\ 1111 \quad \quad \quad -1 \\ \hline 1|0111\ 1111 \quad +127 !! \end{array}$$

Errori di trabocco

Overflow in complemento a due

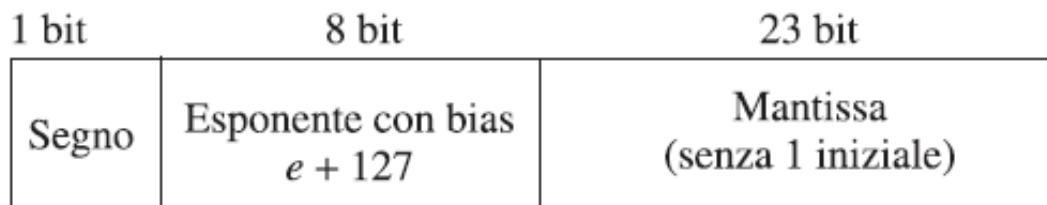
- **Errore di overflow (trabocco)**: quando il risultato supera i limiti di rappresentazione con i bit disponibili
 - Nell'addizione $a+b$ si ha overflow se
 - $a+b \notin [-2^{n-1}, 2^{n-1} - 1] \cap z$
 - Analizzando le due cifre più significative del risultato si può determinare se si è verificato overflow
 - Nell'addizione di due numeri in complemento a due si verifica overflow quando si ha un riporto in una sola delle due cifre più significative
 - **Esercizio**: eseguire la somma $6+2$ a **4 bit** (qual è il massimo intero positivo rappresentabile con 4 bit?)
 - **Esercizio**: eseguire la somma $(-3) + (-7)$ a **4 bit** (qual è il minimo intero negativo rappresentabile con 4 bit?)
 - In tutte le altre condizioni non si verifica overflow

Numeri reali in virgola mobile

- I numeri razionali e reali devono essere rappresentati necessariamente in modo approssimato
 - Con n bit possiamo rappresentare 2^n numeri, mentre in ogni intervallo ci sono infiniti numeri razionali o reali
 - La rappresentazione in **virgola fissa** vista prima non viene quasi mai usata in un computer
 - Si utilizza una notazione a **mantissa** ed **esponente**, come nel calcolo scientifico:
 - 1024.3 viene rappresentato come $1.0243 \cdot 10^3$
 - 1.0243 è la mantissa, 3 è l'esponente
 - È facile dividere e moltiplicare per 10, semplicemente spostando a sinistra o a destra la posizione della virgola (virgola mobile)

I numeri in virgola mobile

- Nella rappresentazione binaria si usa la **base 2** anziché **10**, ma il concetto non cambia
 - ovviamente, cambiando la base, cambia la mantissa
- Esiste uno standard internazionale (**IEEE 754**) che definisce esattamente il formato e la disposizione dei bit di mantissa ed esponente (con i relativi segni)
 - due formati, 32 bit e 64 bit
 - Numero normalizzato: il bit più significativo del numero si trova immediatamente a sinistra della virgola



Singola precisione



Doppia precisione

Numeri “speciali” in virgola mobile

- La rappresentazione in virgola mobile dello standard **IEEE 754** include alcuni valori speciali
 - **Zero**
 - Esponente = -127 (esponente con bias = 00000000)
 - Mantissa = 0
 - **Infinito**
 - Esponente = +128 (esponente con bias = 11111111)
 - Mantissa = 0
 - **NaN** (**Not a Number**, “non è un numero”)
 - Esponente = +128 (esponente con bias = 11111111)
 - Mantissa diversa da 0

Virgola mobile a 32 bit (IEEE754)

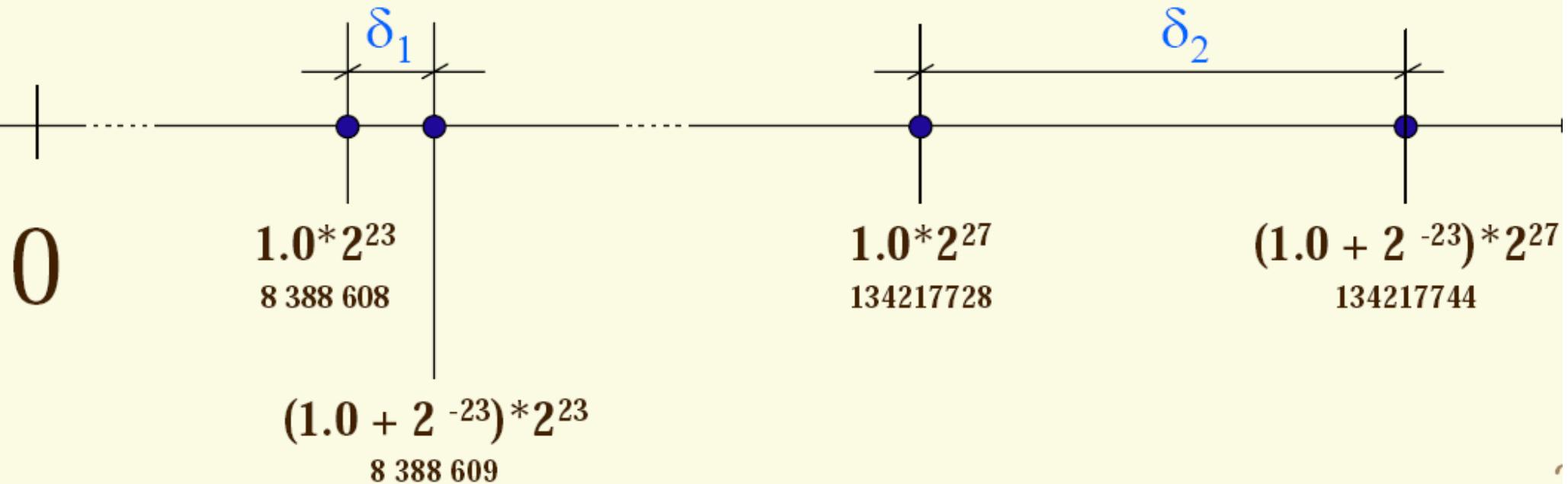
- Il numero **più piccolo** (positivo) rappresentabile è
- $1.00000000000000000000000_2 \times 2^{-126} \sim 1.8 \times 10^{-38}$
- Il numero **più grande** rappresentabile è
- $1.11111111111111111111111_2 \times 2^{+127} \sim 3.4 \times 10^{+38}$
- La **distanza** fra due numeri reali successivi rappresentabili **dipende dal valore dell'esponente**
 - i numeri più vicini differiscono per il valore del bit meno significativo della mantissa e perciò la loro distanza **δ** è
 - $\delta = 2^{-23} \times 2^E$ (E e' il valore dell'esponente)

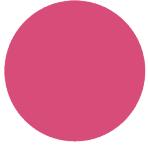
Densità numeri in virgola mobile

- Esempio in formato IEEE754: singola precisione (mantissa a 23 bit)

$$\delta_1 = 1$$
$$2^{-23} * 2^e$$
$$e = 23$$

$$\delta_2 = 16$$
$$2^{-23} * 2^e = 16$$
$$e = 27$$





Arrotondamento in virgola mobile

- Il numero 4.35 non ha una **rappresentazione esatta** nel sistema binario, proprio come 1/3 non ha una rappresentazione esatta nel sistema decimale
 - 4.35 viene rappresentato con un numero appena inferiore a 4.35
 - Se effettuiamo la moltiplicazione 4.35×100 , il risultato è un numero appena inferiore a 435
 - L'errore di arrotondamento viene amplificato dalla moltiplicazione

$$4.35 \times 100 = 434.99999999999999994 \neq 435$$



Errori nelle somme in virgola mobile

- Si consideri la somma **$10.5 + 0.125$**
 - $10.5_{10} = 1010.1_2 = 1.0101_2 \times 2^3$
 - $0.125_{10} = 0.001_2 = 1.0_2 \times 2^{-3}$
 - Per eseguire la somma bisogna **riportare entrambi i termini allo stesso esponente**:
 - $10.5 + 0.125 = 1.0101_2 \times 2^3 + 0.000001_2 \times 2^3 = 1.010101_2 \times 2^3$
 - Se il numero di bit destinati alla mantissa fosse stato inferiore a 6, l'operazione avrebbe dato per risultato **$10.5 + 0.125 = 10.5 !!!!$**
 - A causa della necessaria approssimazione introdotta dalla rappresentazione

Rappresentazione esadecimale

- È la rappresentazione in **base 16**
 - Si usano 16 cifre (simboli): **0, 1, ..., 9, A, B, C, D, E, F**
 - $A = 10_{10}$, $B = 11_{10}$, $C = 12_{10}$, $D = 13_{10}$, $E = 14_{10}$, $F = 15_{10}$
 - Viene usata per rappresentare numeri naturali (senza segno) binari o sequenze di bit in modo più compatto
 - Dato che $16 = 2^4$, per convertire un numero binario in esadecimale si raggruppano i bit a gruppi di quattro partendo da destra verso sinistra
 - $01111111_2 \Rightarrow 0111 | 1111_2 \Rightarrow 7F_{16} = 0x7F$
- Per convertire un numero naturale da esadecimale a binario si convertono le sue singole cifre

notazione
alternativa

Rappresentazione ottale

- A volte si usano numeri **in base otto** (sistema di numerazione ottale)
- Dato che $8 = 2^3$, si può facilmente passare dalla base binaria alla base ottale, raggruppando i bit tre a tre
 - $(100010)_2 = (42)_8$
- Attenzione: si raggruppano i bit tre a tre a partire da destra!
 - $(11100010)_2 = (342)_8$
- Per la conversione inversa
 - si sostituisce ciascuna cifra ottale con le corrispondenti tre cifre binarie, eliminando eventuali zeri a sinistra

Rappresentazione di caratteri

- I caratteri sono rappresentati come numeri naturali (senza segno):
 - A ciascun carattere viene associato un **numero naturale** (codice). L'uso di *Codici Standard* permette a computer di tipo diverso di scambiare testi
- Codice **UNICODE** (<http://www.unicode.org>)
 - Usa 2 byte (16 bit) per ciascun carattere
 - Si possono rappresentare $2^{16} = 65536$ caratteri
 - Praticamente tutti i caratteri degli alfabeti umani esistenti
- I primi 128 codici Unicode coincidono con l'insieme di caratteri **Basic Latin** noto anche come **ASCII** (American Standard Code for Information Interchange)
 - Noi utilizzeremo solo i caratteri ASCII

ASCII TABLE

Decimal	Hex	Char	Decimal	Hex	Char	Decimal	Hex	Char	Decimal	Hex	Char
0	0	[NULL]	32	20	[SPACE]	64	40	@	96	60	'
1	1	[START OF HEADING]	33	21	!	65	41	A	97	61	a
2	2	[START OF TEXT]	34	22	"	66	42	B	98	62	b
3	3	[END OF TEXT]	35	23	#	67	43	C	99	63	c
4	4	[END OF TRANSMISSION]	36	24	\$	68	44	D	100	64	d
5	5	[INQUIRY]	37	25	%	69	45	E	101	65	e
6	6	[ACKNOWLEDGE]	38	26	&	70	46	F	102	66	f
7	7	[REVERSE]	39	27	:	71	47	G	103	67	g
8	8	[BACKSPACE]	40	28	{	72	48	H	104	68	h
9	9	[HORIZONTAL TAB]	41	29	}	73	49	I	105	69	i
10	A	[LINE FEED]	42	2A	^	74	4A	J	106	6A	j
11	B	[VERTICAL TAB]	43	2B	+	75	4B	K	107	6B	k
12	C	[FORM FEED]	44	2C	,	76	4C	L	108	6C	l
13	D	[CARRIAGE RETURN]	45	2D	-	77	4D	M	109	6D	m
14	E	[SHIFT OUT]	46	2E	.	78	4E	N	110	6E	n
15	F	[SHIFT IN]	47	2F	/	79	4F	O	111	6F	o
16	10	[DATA LINK ESCAPE]	48	30	0	80	50	P	112	70	p
17	11	[DEVICE CONTROL 1]	49	31	1	81	51	Q	113	71	q
18	12	[DEVICE CONTROL 2]	50	32	2	82	52	R	114	72	r
19	13	[DEVICE CONTROL 3]	51	33	3	83	53	S	115	73	s
20	14	[DEVICE CONTROL 4]	52	34	4	84	54	T	116	74	t
21	15	[INERTIAL ACKNOWLEDGE]	53	35	5	85	55	U	117	75	u
22	16	[SYNCHRONOUS IDLE]	54	36	6	86	56	V	118	76	v
23	17	[END OF TRANS. BLOCK]	55	37	7	87	57	W	119	77	w
24	18	[CANCEL]	56	38	8	88	58	X	120	78	x
25	19	[END OF MEDIUM]	57	39	9	89	59	Y	121	79	y
26	1A	[SUBSTITUTE]	58	3A	:	90	5A	Z	122	7A	z
27	1B	[ESCAPE]	59	3B	;	91	5B	[123	7B	{
28	1C	[FILE SEPARATOR]	60	3C	<	92	5C	\	124	7C	
29	1D	[GROUP SEPARATOR]	61	3D	=	93	5D]	125	7D	}
30	1E	[RECORD SEPARATOR]	62	3E	>	94	5E	^	126	7E	-
31	1F	[UNIT SEPARATOR]	63	3F	?	95	5F	_	127	7F	[DEL]

Codice UNICODE: esempio

0400

Cyrillie

04FF

	040	041	042	043	044	045	046	047	048	049	04A	04B	04C	04D	04E	04F
0	È	À	Р	а	р	è	Ѡ	Ѱ	Ҫ	Г	Ќ	Ӯ	І	Ӑ	Ӡ	Ӵ
1	Ё	Б	С	б	с	ё	ѡ	ѱ	Ҫ	Г	Ќ	Ӯ	҂	Ӄ	Ӡ	Ӵ
2	҃	В	Т	в	т	ђ	҆	҈	ӷ	Ӯ	Ӯ	ӷ	Ӯ	ӷ	ӷ	ӷ
3	҄	Г	Ү	г	ү	ѓ	Ҋ	Ҽ	Ӧ	Ӯ	Ӯ	Ӯ	Ӯ	ӷ	ӷ	ӷ
4	҅	Д	Փ	դ	փ	҅	Ҽ	Վ	Ӧ	Ӯ	Ӯ	Ӯ	Ӯ	ӷ	ӷ	ӷ
5	҆	Е	Х	ե	խ	҆	Ҽ	Վ	Ӧ	Ӯ	Ӯ	Ӯ	Ӯ	ӷ	ӷ	ӷ
6	҇	Ж	Ц	ж	ց	҇	Ա	Վ	Ӧ	Ӯ	Ӯ	Ӯ	Ӯ	ӷ	ӷ	ӷ
7	҈	З	Ч	з	ч	҈	Ա	Վ	Ӧ	Ӯ	Ӯ	Ӯ	Ӯ	ӷ	ӷ	ӷ
8	҉	И	Ш	и	ш	҉	Ի	Վ	Ӧ	Ӯ	Ӯ	Ӯ	Ӯ	ӷ	ӷ	ӷ
9	Ҋ	Й	Щ	й	щ	Ҋ	Ի	Վ	Ӧ	Ӯ	Ӯ	Ӯ	Ӯ	ӷ	ӷ	ӷ
A	ҋ	Կ	Յ	կ	յ	ҋ	Ժ	Ծ	Ը	Կ	Ծ	Ը	Կ	Ծ	Ը	Ը
B	Ҍ	Լ	Ե	լ	յ	Ҍ	Ժ	Ծ	Ը	Կ	Ծ	Ը	Կ	Ծ	Ը	Ը
C	ҍ	Մ	Յ	մ	յ	ҍ	Խ	Ծ	Ը	Կ	Ծ	Ը	Կ	Ծ	Ը	Ը
D	Ҏ	Н	Э	հ	յ	Ҏ	Խ	Ծ	Ը	Կ	Ծ	Ը	Կ	Ծ	Ը	Ը
E	ҏ	Օ	Յ	օ	յ	ҏ	Հ	Ծ	Ը	Կ	Ծ	Ը	Կ	Ծ	Ը	Ը
F	Ґ	Պ	Я	պ	յ	Ґ	Յ	Ծ	Ը	Ր	Ծ	Ը	Ր	Ծ	Ը	Ը

0400

Cyrillie

	040	041	042	043	044	045	046	047	048	049
0	È	À	Р	а	р	è	Ѡ	Ѱ	Ҫ	Г
1	Ё	Б	С	б	с	ё	ѡ	ѱ	Ҫ	Г
2	҃	В	Т	в	т	ђ	҆	҈	ӷ	ӷ

Rappresentazione di caratteri

- Codice **ASCII** (American Standard Code for Information Interchange)
 - Sottoinsieme di **UNICODE**, ancora molto usato
 - usa 7 bit, si possono rappresentare $2^7 = 128$ caratteri
 - I primi 32 sono **caratteri di controllo**, in particolare:
 - 09: tabulatore '\t'
 - 10: nuova riga. '\n'
 - 13: invio '\r'
 - Si usa quasi sempre il codice **ASCII esteso**
 - usa 8 bit (**1 byte**) per codificare tutti i caratteri di alfabeti occidentali, si possono rappresentare $2^8 = 256$ caratteri
 - anche vocali accentate, lettere speciali (ß, ç ,...), ecc.
 - le sequenze con la prima cifra uguale a zero coincidono con il codice ASCII: **compatibilità**

Codice ASCII

- I caratteri da 32 a 127 sono caratteri stampabili
 - 32: spazio ''
 - 48-57: caratteri numerici, le cifre decimali '0', '1'...
 - 65-90, 97-122: caratteri alfabetici (maiuscoli e minuscoli)
 - 33-47, 58-64, 91-96, 123-127: caratteri di interpunzione

Carattere	Codice	Decimale	Carattere	Codice	Decimale
!	'\u0021'	33	@	'\u0040'	64
"	'\u0022'	34	A	'\u0041'	65
#	'\u0023'	35	B	'\u0042'	66
\$	'\u0024'	36	C	'\u0043'	67
%	'\u0025'	37	D	'\u0044'	68
&	'\u0026'	38	E	'\u0045'	69
'	'\u0027'	39	F	'\u0046'	70
('\u0028'	40	G	'\u0047'	71
)	'\u0029'	41	H	'\u0048'	72
*	'\u002A'	42	I	'\u0049'	73
+	'\u002B'	43	J	'\u004A'	74

Codice ASCII

Carattere	Codice	Decimale	Carattere	Codice	Decimale
,	'\u002C'	44	K	'\u004B'	75
-	'\u002D'	45	L	'\u004C'	76
.	'\u002E'	46	M	'\u004D'	77
/	'\u002F'	47	N	'\u004E'	78
0	'\u0030'	48	O	'\u004F'	79
1	'\u0031'	49	P	'\u0050'	80
2	'\u0032'	50	Q	'\u0051'	81
3	'\u0033'	51	R	'\u0052'	82
4	'\u0034'	52	S	'\u0053'	83
5	'\u0035'	53	T	'\u0054'	84
6	'\u0036'	54	U	'\u0055'	85
7	'\u0037'	55	V	'\u0056'	86
8	'\u0038'	56	W	'\u0057'	87
9	'\u0039'	57	X	'\u0058'	88
:	'\u003A'	58	Y	'\u0059'	89
:	'\u003B'	59	Z	'\u005A'	90
<	'\u003C'	60	['\u005B'	91
=	'\u003D'	61	\	'\u005C'	92
>	'\u003E'	62]	'\u005D'	93
?	'\u003F'	63	^	'\u005E'	94

Codice ASCII

Carattere	Codice	Decimale	Carattere	Codice	Decimale
-	'\u005F'	95	o	'\u006F'	111
'	'\u0060'	96	p	'\u0070'	112
a	'\u0061'	97	q	'\u0071'	113
b	'\u0062'	98	r	'\u0072'	114
c	'\u0063'	99	s	'\u0073'	115
d	'\u0064'	100	t	'\u0074'	116
e	'\u0065'	101	u	'\u0075'	117
f	'\u0066'	102	v	'\u0076'	118
g	'\u0067'	103	w	'\u0077'	119
h	'\u0068'	104	x	'\u0078'	120
i	'\u0069'	105	y	'\u0079'	121
j	'\u006A'	106	z	'\u007A'	122
k	'\u006B'	107	{	'\u007B'	123
l	'\u006C'	108		'\u007C'	124
m	'\u006D'	109	}	'\u007D'	125
n	'\u006E'	110	~	'\u007E'	126

Un programma che elabora numeri

```
public class Coins1
{ public static void main(String[] args)
    { int lit = 15000;      // lire italiane
        double euro = 2.35; // euro

        // calcola il valore totale
        double totalEuro = euro + lit / 1936.27;

        // stampa il valore totale
        String outMessage = "Valore totale in euro ";
        System.out.print(outMessage);
        System.out.println(totalEuro);

    }
}
```



L'uso delle variabili

- Ogni programma fa uso di **variabili**
- Le **variabili** sono spazi di memoria, identificati da un **nome**, che possono conservare **valori** di un **determinato tipo**
- Ciascuna variabile deve essere **definita**, indicandone il **tipo** ed il **nome**

```
int lit;
```

```
String outMessage;
```

- Una variabile può contenere soltanto valori del suo **stesso tipo**
- Nella **definizione di una variabile**, è possibile **assegnarle** un **valore iniziale**

```
int lit = 15000;
```

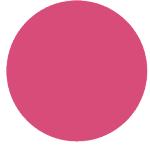
L'uso delle variabili

- Avremmo potuto scrivere un programma per risolvere lo stesso problema anche senza fare uso di variabili

```
public class Coins2
{ public static void main(String[] args)
    { System.out.print("Valore totale in euro ");
      System.out.println(2.35 + 15000 / 1936.27);
    }
}
```

- ma sarebbe stato **molto meno comprensibile** e **modificabile con difficoltà**

Attenzione:
questo visualizza il risultato dell'espressione



I nomi delle variabili (identificatori)

- La scelta dei nomi per le variabili è molto importante
 - È bene scegliere nomi che **descrivano adeguatamente** la funzione della variabile
- In Java, un nome (di variabile, di metodo, di classe...) può essere composto da **lettere**, da **numeri** e dal **carattere di sottolineatura** _ ma
 - deve iniziare con una lettera
 - non può essere una **parola riservata** o un **simbolo riservato**
 - Incluse le costanti **true**, **false**, **null**
 - non può contenere **separatori** o simboli di **operatori** di Java
 - Le lettere **maiuscole** sono **diverse** dalle **minuscole!**
 - Java è un linguaggio “**case-sensitive**”
 - È buona norma non usare in un programma nomi di variabili che differiscano soltanto per una maiuscola



I nomi delle variabili

- Parole/simboli **non** utilizzabili in nomi di variabili
 - **Parole riservate** di Java: le abbiamo già viste (App. C)
 - **Separatori**: i seguenti 9 caratteri sono utilizzati come separatori (caratteri di interpunkzione)

() { } [] ; , .

- **Operatori**: i seguenti 37 caratteri o token sono utilizzati nelle espressioni come operatori

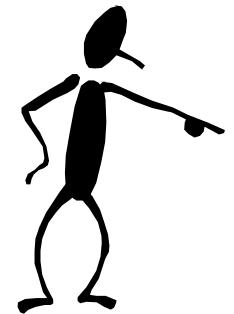
= > < ! ~ ? :

== <= >= != && || ++ --

+ - * / & | ^ % << >> >>>

+= -= *= /= &= |= ^= %= <<= >>= >>>=

Definizione di variabili



- Sintassi:

```
nomeTipo nomeVariabile;
```

```
nomeTipo nomeVariabile = espressione;
```

- Scopo: definire la nuova variabile **nomeVariabile**, di tipo **nomeTipo**, ed eventualmente assegnarle il valore iniziale **espressione**
- Di solito in Java si usano le seguenti **convenzioni**
 - *i nomi di variabili e metodi iniziano con una lettera minuscola*
lit **main**
 - *i nomi di classi iniziano con una lettera maiuscola*
Coins1
 - i nomi composti, in entrambi i casi, si ottengono attaccando le parole successive alla prima con la maiuscola (nomi a “forma di cammello”)

outMessage**totalEuro****MoveRectangle**

È tutto chiaro? ...

1. Di che tipo sono i valori `0` e `“0”`?
2. Quali dei seguenti identificatori sono validi?

`Greeting1`

`g`

`void`

`101dalmatians`

`Hello, World`

`<greeting>`

3. Definire una variabile adatta a memorizzare il vostro nome, usando un nome *a forma di cammello*

L'assegnazione

- Abbiamo visto come i programmi usino le variabili per memorizzare i valori da elaborare e i risultati dell'elaborazione
- Le **variabili** sono posizioni in memoria che possono conservare **valori** di un **determinato tipo**
- Il valore memorizzato in una variabile può essere **modificato**, non soltanto **inizializzato**...
- Il cambiamento del valore di una variabile si ottiene con un **enunciato di assegnazione**

L'assegnazione

```
public class Coins3
{ public static void main(String[] args)
    { int lit = 15000;          // lire italiane
        double euro = 2.35;     // euro
        double dollars = 3.05;   // dollari
        // calcola il valore totale
        // sommando successivamente i contributi
        double totalEuro = lit / 1936.27;
        totalEuro = totalEuro + euro;
        totalEuro = totalEuro + dollars * 0.72;
        System.out.print("Valore totale in euro ");
        System.out.println(totalEuro);
    }
}
```

L'assegnazione

- In questo caso il valore della **variabile totalEuro cambia** durante l'esecuzione del programma
 - per prima cosa la variabile viene **inizializzata** contestualmente alla sua **definizione**

```
double totalEuro = lit / 1936.27;
```

- poi la variabile viene **incrementata**, due volte
- mediante **enunciati di assegnazione**

```
totalEuro = totalEuro + euro;  
totalEuro = totalEuro + dollars * 0.72;
```

Molto
importante

L'assegnazione

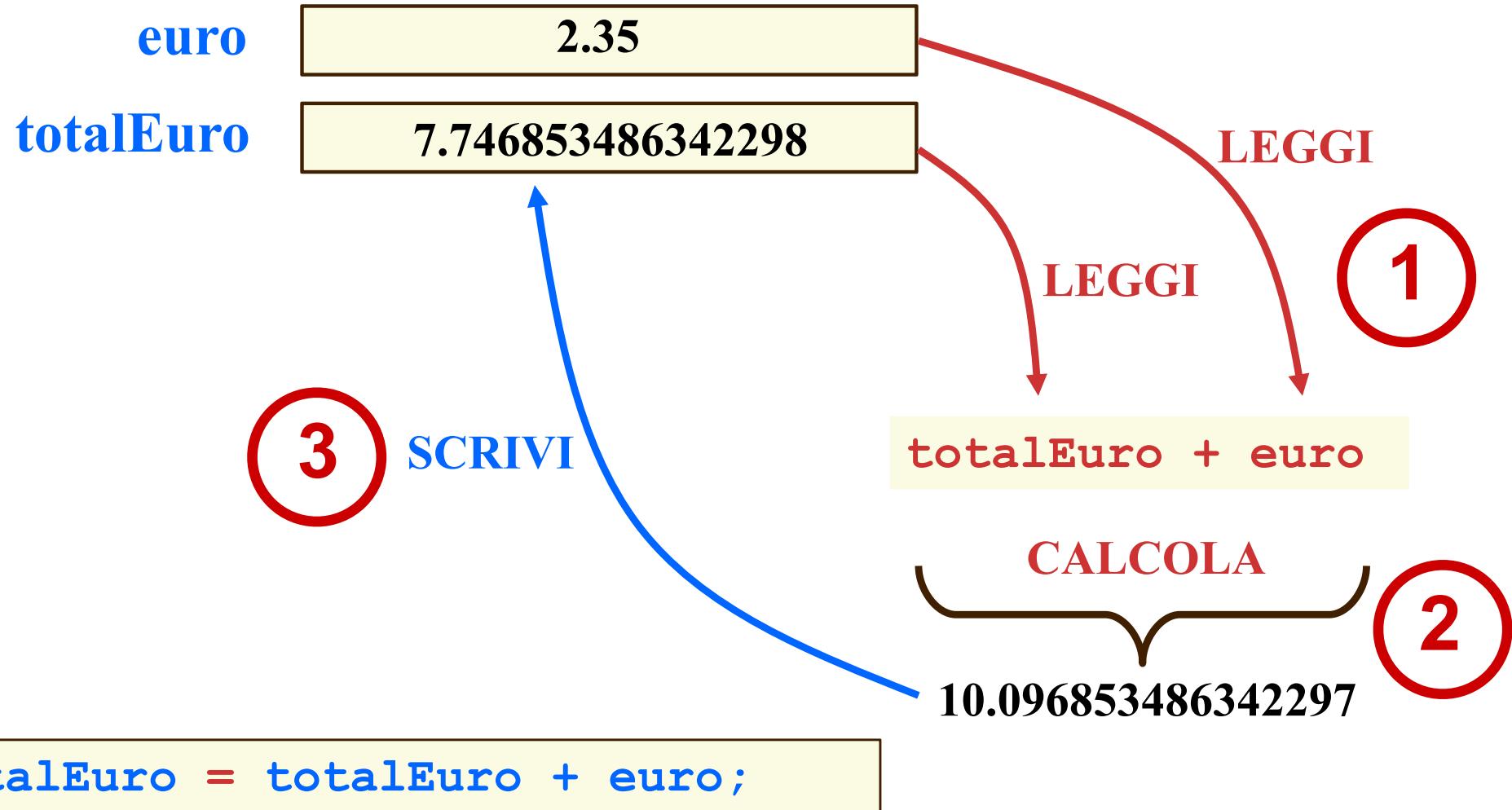
- Analizziamo l'enunciato di assegnazione

```
totalEuro = totalEuro + euro;
```

- Cosa significa? **Non** certo che la variabile **totalEuro** è *uguale* a se stessa più qualcos'altro...
- L'enunciato di assegnazione significa
 - Calcola il valore dell'espressione a destra del segno = e scrivi il risultato nella posizione di memoria assegnata alla variabile indicata a sinistra del segno =**

L'assegnazione

Molto
importante



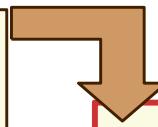
Assegnazione o definizione?

- Attenzione a non confondere la **definizione** di una variabile con un enunciato di **assegnazione**!

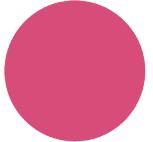
```
double totalEuro = lit / 1936.27;  
totalEuro = totalEuro + euro;
```

- La definizione di una variabile inizia specificando il **tipo** della variabile, l'assegnazione no
- **Una variabile può essere definita una volta sola**, mentre le si può assegnare un valore molte volte
- Il compilatore segnala come errore il tentativo di definire una variabile una seconda volta

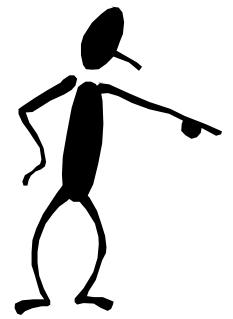
```
double euro = 2;  
double euro = euro + 3;
```



euro is already defined



Assegnazione



- Sintassi:

```
nomeVariabile = espressione;
```

- Scopo: assegnare il nuovo valore **espressione** alla variabile **nomeVariabile**
 - **Nota:** purtroppo Java (come C e C++) utilizza il segno **=** per indicare l'assegnazione, creando confusione con l'operatore di uguaglianza (che vedremo essere un doppio segno **=**, cioè **==**);
 - **Errore frequente:** confusione tra **=** e **==**
 - Altri linguaggi usano simboli diversi per l'assegnazione (ad esempio, il linguaggio Pascal usa **:=**)

È tutto chiaro? ...

1. L'espressione `12 = 12` è valida in Java?
2. Come si assegna il valore “Hello, Nina” alla variabile `greeting`, definita precedentemente nel codice?

Tipi numerici

- Questo programma elabora ***due tipi di numeri***
 - ***numeri interi*** per le lire italiane, che non prevedono l'uso di decimi e centesimi e quindi non hanno bisogno di una parte frazionaria
 - ***numeri frazionari*** (“in virgola mobile”) per gli euro, che prevedono l'uso di decimi e centesimi e assumono valori con il separatore decimale
- I ***numeri interi*** (positivi e negativi) si rappresentano in Java con il tipo di dati ***int***
- I ***numeri in virgola mobile*** (positivi e negativi, ***a precisione doppia***) si rappresentano in Java con il tipo di dati ***double***

Perché usare due tipi di numeri?

- In realtà sarebbe possibile usare numeri in virgola mobile anche per rappresentare i numeri interi, ma ecco due buoni motivi per non farlo
 - “**leggibilità**”: indicando esplicitamente che per le lire italiane usiamo un numero intero, rendiamo **evidente** il fatto che non esistono i decimali per le lire italiane
 - è importante rendere comprensibili i programmi!
 - “**tecnico**”: i numeri interi rappresentati come tipo di dati **int** sono più **efficienti**, perché **occupano meno spazio in memoria** e sono **elaborati più velocemente**

Alcune note sintattiche

- L'operatore che indica la divisione è `/`, quello che indica la moltiplicazione è `*`

```
lit / 1936.27
```

- Per scrivere numeri in virgola mobile, bisogna usare il **punto** come separatore decimale (uso anglosassone)

```
1936.27
```

- Quando si scrivono numeri, non bisogna indicare il punto separatore delle migliaia

```
15000
```

- I numeri in virgola mobile si possono anche esprimere in **notazione esponenziale**

```
1.93E3 // vale 1.93 × 103
```

Oggetti: Motivazioni

- Elaborando numeri (int, double) e stringhe si possono scrivere programmi interessanti, ma **programmi più utili** hanno bisogno di manipolare **dati più complessi**
 - conti bancari, dati anagrafici, forme grafiche...
- Il linguaggio Java gestisce questi dati complessi sotto forma di **oggetti**
- Gli oggetti e il loro **comportamento** vengono descritti mediante le **classi** e i loro **metodi**

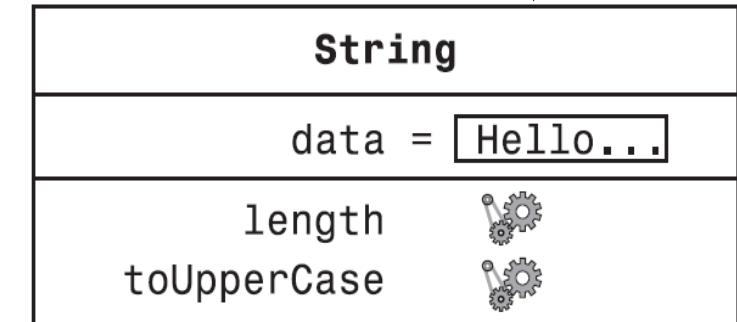
Oggetti

- Un **oggetto** è un'entità che può essere manipolata in un programma mediante l'invocazione di **metodi**

```
String greeting = "Hello, World!";
```

greeting

- **greeting** è un oggetto
- appartiene alla **classe String**
- si può *usare* mediante i suoi **metodi**
 - Ad esempio: **length**, **toUpperCase**, ...
- Per il momento, consideriamo che un oggetto sia una “scatola nera” (**black box**) dotata di
 - un'**interfaccia pubblica** (i metodi che si possono usare), che definisce il comportamento dell'oggetto
 - una **realizzazione (implementazione) nascosta** (il codice dei metodi ed i loro dati)



Classi

- Una classe
 - è una **fabbrica di oggetti**
 - gli oggetti che si creano sono **esemplari** (“istanze”, instance) di una classe, che ne è il prototipo
 - specifica i **metodi** che si possono invocare per gli oggetti che sono esemplari di tale classe (**l'interfaccia pubblica**)
 - definisce i **particolari** della realizzazione dei metodi (codice e dati)
 - Per ora non esaminiamo questo aspetto
 - Ma una classe è anche un “contenitore” di
 - metodi statici (**HelloTester** contiene il metodo **main**)
 - oggetti statici (**System** contiene l'oggetto **out**)

Finora abbiamo visto solo questo aspetto, che è forse quello meno importante

Utilizzare una classe

- Iniziamo lo studio delle classi analizzando come si usano oggetti di una classe che si suppone **già definita** da altri
 - Per esempio la classe **String**, già definita dagli sviluppatori di Java
 - È possibile usare oggetti di cui **non si conoscono** i dettagli realizzativi, conoscendo unicamente l'interfaccia pubblica della classe
 - Questo è un concetto molto importante della programmazione orientata agli oggetti
 - **Usare** oggetti di una classe e **realizzare** una classe sono due attività ben distinte!

Metodi

- Costituiscono l'**interfaccia pubblica** di una classe
 - Istruzioni valide:

```
String greeting = "Hello, World!";
int n = greeting.length();
String river = "Mississippi";
String bigRiver = river.toUpperCase();
```

- Istruzione non valida (il metodo non appartiene alla classe)

```
System.out.length();
```



Metodi, parametri esplicativi/impliciti

- Alcuni metodi necessitano di **valori in ingresso**, o **parametri esplicativi**, che specificino l'operazione da svolgere

```
System.out.println(greeting);
```

- greeting è un *parametro esplicito*
- Altri metodi no: tutte le informazioni necessarie sono memorizzate nell'oggetto corrispondente, il **parametro implicito**

```
int n = greeting.length();
```

- greeting è il *parametro implicito*

Metodi, valori restituiti

- Alcuni metodi restituiscono un valore

```
System.out.println(greeting.length());
```

- Il valore restituito da `length` viene usato come parametro esplicito di `println`
- Esempio più complesso

```
river.replace("issipp", "our");
```

- Due parametri esplicativi (le stringhe “issipp”, “our”)
- Un parametro implicito (l’oggetto `river`)
- Un valore restituito (la stringa “Missouri”)
- **ATTENZIONE:** `river` contiene ancora “Mississippi”

Definizioni di metodi: firma

```
public void println(String output)
public String replace(String target, String replac)
```

- La **definizione di un metodo** inizia con la sua **intestazione (firma, signature)**, composta da
 - uno **specificatore di accesso**
 - Indica quali altri metodi possono invocare il metodo
 - Un metodo **public** può essere invocato da qualsiasi altro metodo di qualsiasi altra classe
 - Un metodo può essere anche **private**
 - il tipo di dati **restituito** dal metodo (**String, void...**)
 - il **nome** (identificatore) del metodo (**println, replace, ...**)
 - un **elenco di parametri esplicativi**, eventualmente vuoto, tra **parentesi tonde**
 - di ogni parametro si indica il tipo ed il nome
 - più parametri sono separati da una virgola



Valore restituito di tipo void

- La dichiarazione di un metodo specifica quale sia il tipo di dati restituito dal metodo al termine della sua invocazione
 - Per esempio, il metodo **length()** restituisce un valore di tipo **int**

```
int n = river.length();
```

- Se un metodo **non restituisce alcun valore**, si dichiara che restituisce il tipo speciale **void** (assente, non valido...)

```
double b = System.out.println(river); // ERRORE  
System.out.println(river);           // OK
```



È tutto chiaro? ...

- Identificate i parametri impliciti ed esplicativi, e il valore restituito dall'invocazione del metodo

`river.length()`

- Identificate il risultato dell'invocazione

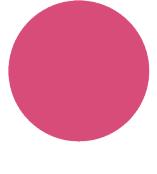
`river.replace("p", "s")`

(immaginando che `river` contenga la stringa “**Mississippi**”)

- Identificate il risultato dell'invocazione

`greeting.replace("World", "Dave").length()`

(immaginando che `greeting` contenga la stringa “**Hello, World!**”)



U

Guida in linea



- Le classi della libreria Java sono migliaia
 - Non serve usare la memoria, usiamo la documentazione!
- L'ambiente JDK fornisce la **documentazione API (Application Programming Interface)** per *l'utilizzo delle classi della libreria standard*
 - Si può **scaricare** seguendo le indicazioni sul sito del corso
 - È disponibile **online**: <https://docs.oracle.com/en/java/javase/13/docs/api/index.html>
 - È disponibile **online** anche sul sito dell'aula Taliercio
- Vengono inoltre forniti alcuni documenti in formato “**tutorial**” per la descrizione delle funzionalità di interi pacchetti ed esempi di programmi (“**demo**”)
- È anche disponibile **il codice sorgente di tutte le classi della libreria standard**, la cui lettura è interessante e utile, anche se spesso complessa

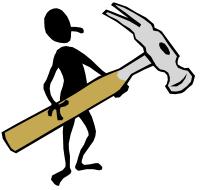
Guida in linea

<https://docs.oracle.com/en/java/javase/21/docs/api/index.html>

The screenshot shows a web browser window displaying the Java SE 19 & JDK 19 API Specification. The URL in the address bar is <https://docs.oracle.com/en/java/javase/21/docs/api/index.html>. The page title is "Java® Platform, Standard Edition & Java Development Kit Version 19 API Specification". A search bar at the top right contains the placeholder "Search". Below the title, it says "This document is divided into two sections: Java SE and JDK". The Java SE section describes the core Java platform APIs, while the JDK section describes APIs specific to the Java Development Kit. A navigation bar at the top includes links for OVERVIEW, MODULE, PACKAGE, CLASS, USE, TREE, PREVIEW, NEW, DEPRECATED, INDEX, HELP, and SEARCH. A table below lists all Java SE modules, each with a brief description. The "All Modules" tab is selected.

Module	Description
<code>java.base</code>	Defines the foundational APIs of the Java SE Platform.
<code>java.compiler</code>	Defines the Language Model, Annotation Processing, and Java Compiler APIs.
<code>java.datatransfer</code>	Defines the API for transferring data between and within applications.
<code>java.desktop</code>	Defines the AWT and Swing user interface toolkits, plus APIs for accessibility, audio, imaging, printing, and JavaBeans.
<code>java.instrument</code>	Defines services that allow agents to instrument programs running on the JVM.
<code>java.logging</code>	Defines the Java Logging API.
<code>java.management</code>	Defines the Java Management Extensions (JMX) API.
<code>java.management.rmi</code>	Defines the RMI connector for the Java Management Extensions (JMX) Remote API.
<code>java.naming</code>	Defines the Java Naming and Directory Interface (JNDI) API.
<code>java.net.http</code>	Defines the HTTP Client and WebSocket APIs.
<code>java.prefs</code>	Defines the Preferences API.
<code>java.rmi</code>	Defines the Remote Method Invocation (RMI) API.
<code>java.scripting</code>	Defines the Scripting API.
<code>java.se</code>	Defines the API of the Java SE Platform.
<code>java.security.jgss</code>	Defines the Java binding of the IETF Generic Security Services API (GSS-API).
<code>java.security.sasl</code>	Defines Java support for the IETF Simple Authentication and Security Layer (SASL).
<code>java.smartcardio</code>	Defines the Java Smart Card I/O API.
<code>java.sql</code>	Defines the JDBC API.
<code>java.sql.rowset</code>	Defines the JDBC RowSet API.
<code>java.transaction.xa</code>	Defines an API for supporting distributed transactions in JDBC.
<code>java.xml</code>	Defines the Java API for XML Processing (JAXP), the Streaming API for XML (StAX), the Simple API for XML (SAX), and the W3C Document Object Model (DOM) API.
<code>java.xml.crypto</code>	Defines the API for XML cryptography.
<code>jdk.accessibility</code>	Defines JDK utility classes used by implementors of Assistive Technologies.
<code>jdk.attach</code>	Defines the attach API.
<code>jdk.charsets</code>	Provides charsets that are not in <code>java.base</code> (mostly double byte and IBM charsets).
<code>jdk.compiler</code>	Defines the implementation of the <code>java.lang.Compiler</code> and its command line equivalent, <code>javac</code> .

Guida in linea: descrizione di classi



- La ***descrizione di una classe*** comprende
 - l'elenco di tutti i suoi metodi pubblici (la sua ***interfaccia***)
 - una sintetica descrizione testuale delle motivazioni alla base del progetto della classe e delle modalità del suo utilizzo
- Per ogni metodo, vengono indicati
 - il nome e la funzionalità svolta
 - il tipo ed il significato dei parametri richiesti, se ci sono
 - il tipo ed il significato del valore restituito, se c'è
 - le eventuali ***eccezioni*** lanciate in caso di errore

Descrizione di classi



The screenshot shows a browser window displaying the Java SE 19 & JDK 19 documentation for the `String` class. The URL is docs.oracle.com/en/java/javase/19/docs/api/java.base/java/lang/String.html. The page title is "Class String". The navigation bar includes links for OVERVIEW, MODULE, PACKAGE, CLASS (which is highlighted), USE, TREE, PREVIEW, NEW, DEPRECATED, INDEX, and HELP. The search bar contains the text "Search". The page content starts with the module (`java.base`) and package (`java.lang`). It then describes the `String` class, noting it extends `Object` and implements `Serializable`, `Comparable<String>`, `CharSequence`, `Constable`, and `ConstantDesc`. A code snippet shows the declaration of the `String` class. Below this, a note states that the `String` class represents character strings and that all string literals in Java programs are instances of this class. It also notes that strings are immutable. A code example demonstrates creating a string from an array of characters. Further examples show various string manipulation methods like `println`, concatenation, and substring extraction. A note about Unicode representation is provided, followed by a section on string comparison and a note about implementation details.

Module `java.base`
Package `java.lang`

Class `String`

`java.lang.Object`
`java.lang.String`

All Implemented Interfaces:

`Serializable`, `CharSequence`, `Comparable<String>`, `Constable`, `ConstantDesc`

```
public final class String
extends Object
implements Serializable, Comparable<String>, CharSequence, Constable, ConstantDesc
```

The `String` class represents character strings. All string literals in Java programs, such as "abc", are implemented as instances of this class.

Strings are constant; their values cannot be changed after they are created. String buffers support mutable strings. Because `String` objects are immutable they can be shared. For example:

```
String str = "abc";
```

is equivalent to:

```
char data[] = {'a', 'b', 'c'};  
String str = new String(data);
```

Here are some more examples of how strings can be used:

```
System.out.println("abc");  
String cde = "cde";  
System.out.println("abc" + cde);  
String c = "abc".substring(2, 3);  
String d = cde.substring(1, 2);
```

The class `String` includes methods for examining individual characters of the sequence, for comparing strings, for searching strings, for extracting substrings, and for creating a copy of a string with all characters translated to uppercase or to lowercase. Case mapping is based on the Unicode Standard version specified by the `Character` class.

The Java language provides special support for the string concatenation operator (+), and for conversion of other objects to strings. For additional information on string concatenation and conversion, see *The Java Language Specification*.

Unless otherwise noted, passing a null argument to a constructor or method in this class will cause a `NullPointerException` to be thrown.

A `String` represents a string in the UTF-16 format in which *supplementary characters* are represented by *surrogate pairs* (see the section *Unicode Character Representations* in the `Character` class for more information). Index values refer to char code units, so a supplementary character uses two positions in a `String`.

The `String` class provides methods for dealing with Unicode code points (i.e., characters), in addition to those for dealing with Unicode code units (i.e., char values).

Unless otherwise noted, methods for comparing Strings do not take locale into account. The `Collator` class provides methods for finer-grain, locale-sensitive String comparison.

Implementation Note:

The implementation of the string concatenation operator is left to the discretion of a Java compiler, as long as the compiler ultimately conforms to *The Java Language Specification*. For example, the `javac` compiler may implement the operator with `StringBuffer`, `StringBuilder`, or `java.lang.invoke.StringConcatFactory` depending on the JDK version. The implementation of string conversion is typically through the method `toString`, defined by `Object` and inherited by all classes in Java.

See *Java Language Specification*:

15.18.1 String Concatenation Operator + ↗

Descrizione di classi

The screenshot shows a web browser displaying the Java SE 19 & JDK 19 documentation for the `String` class. The URL is `docs.oracle.com`. The navigation bar includes links for OVERVIEW, MODULE, PACKAGE, CLASS (which is highlighted), USE, TREE, PREVIEW, NEW, DEPRECATED, INDEX, and HELP. The search bar at the top right contains the text "Search". Below the navigation bar, there are links for SUMMARY, NESTED, FIELD, CONSTR, and METHOD. The main content area is titled "Method Summary" and features a table with the following columns: Modifier and Type, Method, and Description. The table lists numerous methods of the `String` class, such as `charAt(int index)`, `chars()`, `codePointAt(int index)`, `codePointBefore(int index)`, `codePointCount(int beginIndex, int endIndex)`, `codePoints()`, `compareTo(String anotherString)`, `compareToIgnoreCase(String str)`, `concat(String str)`, `contains(CharSequence s)`, `contentEquals(CharSequence cs)`, `contentEquals(StringBuffer sb)`, `copyValueOf(char[] data)`, `copyValueOf(char[] data, int offset, int count)`, `describeConstable()`, `endsWith(String suffix)`, `equals(Object anObject)`, `equalsIgnoreCase(String anotherString)`, `format(String format, Object... args)`, `format(Locale l, String format, Object... args)`, `formatted(Object... args)`, and `getBytes()`.

Modifier and Type	Method	Description
char	<code>charAt(int index)</code>	Returns the char value at the specified index.
<code>IntStream</code>	<code>chars()</code>	Returns a stream of int zero-extending the char values from this sequence.
int	<code>codePointAt(int index)</code>	Returns the character (Unicode code point) at the specified index.
int	<code>codePointBefore(int index)</code>	Returns the character (Unicode code point) before the specified index.
int	<code>codePointCount(int beginIndex, int endIndex)</code>	Returns the number of Unicode code points in the specified text range of this <code>String</code> .
<code>IntStream</code>	<code>codePoints()</code>	Returns a stream of code point values from this sequence.
int	<code>compareTo(String anotherString)</code>	Compares two strings lexicographically.
int	<code>compareToIgnoreCase(String str)</code>	Compares two strings lexicographically, ignoring case differences.
<code>String</code>	<code>concat(String str)</code>	Concatenates the specified string to the end of this string.
boolean	<code>contains(CharSequence s)</code>	Returns true if and only if this string contains the specified sequence of char values.
boolean	<code>contentEquals(CharSequence cs)</code>	Compares this string to the specified <code>CharSequence</code> .
boolean	<code>contentEquals(StringBuffer sb)</code>	Compares this string to the specified <code>StringBuffer</code> .
static <code>String</code>	<code>copyValueOf(char[] data)</code>	Equivalent to <code>valueOf(char[])</code> .
static <code>String</code>	<code>copyValueOf(char[] data, int offset, int count)</code>	Equivalent to <code>valueOf(char[], int, int)</code> .
<code>Optional<String></code>	<code>describeConstable()</code>	Returns an <code>Optional</code> containing the nominal descriptor for this instance, which is the instance itself.
boolean	<code>endsWith(String suffix)</code>	Tests if this string ends with the specified suffix.
boolean	<code>equals(Object anObject)</code>	Compares this string to the specified object.
boolean	<code>equalsIgnoreCase(String anotherString)</code>	Compares this <code>String</code> to another <code>String</code> , ignoring case considerations.
static <code>String</code>	<code>format(String format, Object... args)</code>	Returns a formatted string using the specified format string and arguments.
static <code>String</code>	<code>format(Locale l, String format, Object... args)</code>	Returns a formatted string using the specified locale, format string, and arguments.
<code>String</code>	<code>formatted(Object... args)</code>	Formats using this string as the format string, and the supplied arguments.
<code>byte[]</code>	<code>getBytes()</code>	Encodes this <code>String</code> into a sequence of bytes using the default charset, storing the result into a new <code>byte array</code> .

Variabili oggetto

- *Una variabile oggetto* conserva non l'oggetto stesso, ma informazioni sulla sua posizione nella memoria del computer
 - è un *riferimento* o *puntatore*
 - Per definire una variabile oggetto si indica il nome della **classe** ai cui oggetti farà riferimento la variabile, seguito dal nome della **variabile stessa**

```
NomeClasse nomeOggetto;
```
 - La definizione di una variabile oggetto crea un riferimento **non inizializzato**, cioè la variabile non fa riferimento ad alcun oggetto

Costruire oggetti: l'operatore new

- Per ***creare un nuovo oggetto*** di una classe si usa **l'operatore new** seguito dal ***nome della classe*** e da una coppia di parentesi tonde

```
new NomeClasse (parametri) ;
```

- L'operatore **new crea un nuovo oggetto e ne restituisce un riferimento**, che può essere assegnato a una variabile oggetto del tipo appropriato

```
NomeClasse nomevar = new NomeClasse (parametri) ;
```

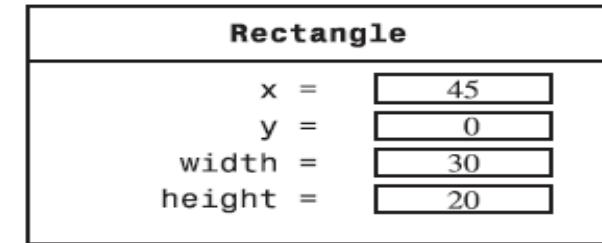
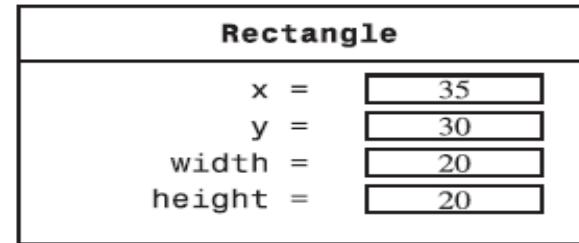
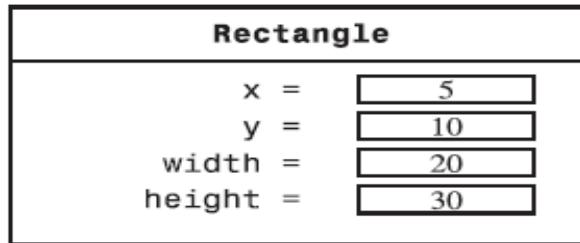
nomevar



NomeClasse



Esempio: la classe Rectangle



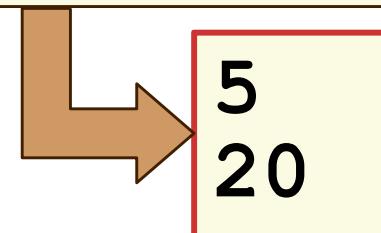
- Un rettangolo è descritto dalle coordinate (x,y) del suo vertice in alto a sinistra, e da larghezza e altezza.
- Per creare un rettangolo bisogna
 - Specificare x, y, width, height
 - Invocare l'operatore **new**
 - **Assegnare il rettangolo appena creato ad una variabile oggetto**

```
Rectangle box = new Rectangle(5, 10, 20, 20);
```

Copiare variabili

- Se si vuole ottenere anche con variabili oggetto lo stesso effetto dell'assegnazione di variabili di tipi numerici fondamentali, è necessario ***creare esplicitamente una copia dell'oggetto con lo stesso stato***, inizializzarlo adeguatamente e assegnarlo alla seconda variabile oggetto

```
Rectangle box = new Rectangle(5, 10, 20, 20);
Rectangle box2 = new Rectangle(box.getX(),
    box.getY(), box.getWidth(), box.getHeight());
box2.translate(15, 25);
System.out.println(box.getX());
System.out.println(box2.getX());
```



È tutto chiaro? ...

1. Come si costruisce un quadrato centrato nel punto di coordinate 100, 100 con lato di lunghezza 20?

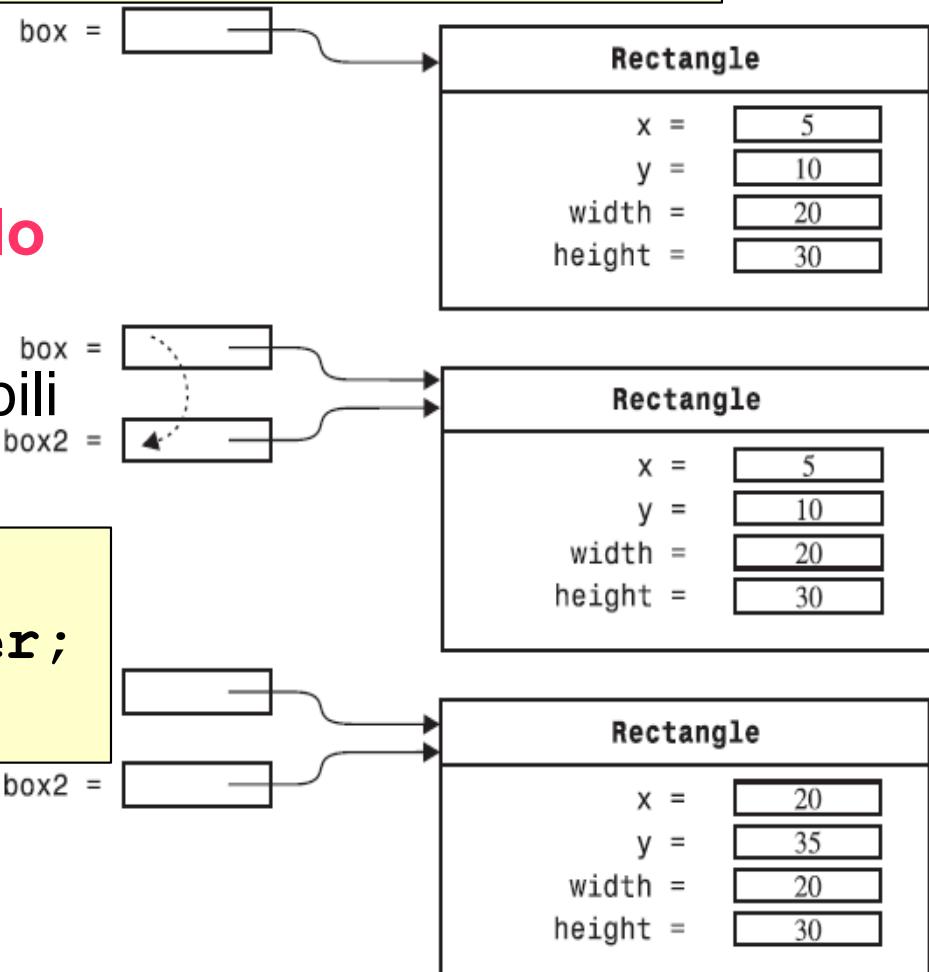
Riferimenti a oggetti

```
Rectangle box = new Rectangle(5, 10, 20, 30);  
Rectangle box2 = box;  
box2.translate(15,25);
```

- **box** e **box2** contengono un riferimento allo stesso oggetto
 - Usando **box** o **box2** **modifico lo stesso oggetto!**
- Comportamento diverso dalle variabili numeriche:

```
int luckyNumber = 13;  
int luckyNumber2 = luckyNumber;  
luckyNumber2 = 12;
```

- In questo caso il valore di **luckyNumber** **non cambia!**



Copiare variabili

- Le variabili di tipi numerici fondamentali indirizzano una posizione in memoria che contiene un **valore**, che viene copiato con l'assegnazione
 - cambiando il valore di una variabile, non viene cambiato il valore dell'altra
- Le variabili oggetto indirizzano una posizione in memoria che, invece, contiene un **riferimento ad un oggetto**, e solo tale riferimento viene copiato con l'assegnazione
 - modificando lo stato dell'oggetto, tale modifica è visibile da entrambi i riferimenti
 - **non viene creata una copia dell'oggetto!**

