# Rappresentazione di numeri in virgola mobile

# Numeri in virgola mobile

- I numeri razionali e reali devono essere rappresentati necessariamente in modo approssimato
  - Con **n** bit possiamo rappresentare $\mathbf{2^n}$ 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
  - Uso un numero fissato di byte per rappresentare la parte intera e per rappresentare la parte decimale
    - Differenza tra 1.23456 e 1.89929
    - Diffrenza tra189946584. 23456 e 189946584. 89929

# Numeri in virgola mobile

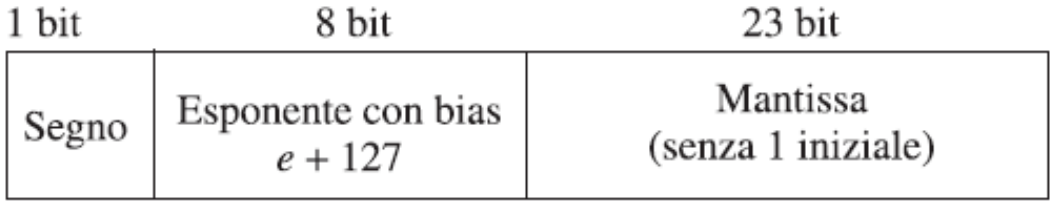
- Si utilizza una notazione a **mantissa** ed **esponente:**
  - **1024.3** viene rappresentato come **$1.0243*10^3$**
    - **1.0243** è **la mantissa**, **3** è **l'esponente**
  - Si divide e moltiplica per 10 spostando a sinistra o a destra la posizione della virgola (virgola mobile)

# Numeri in virgola mobile

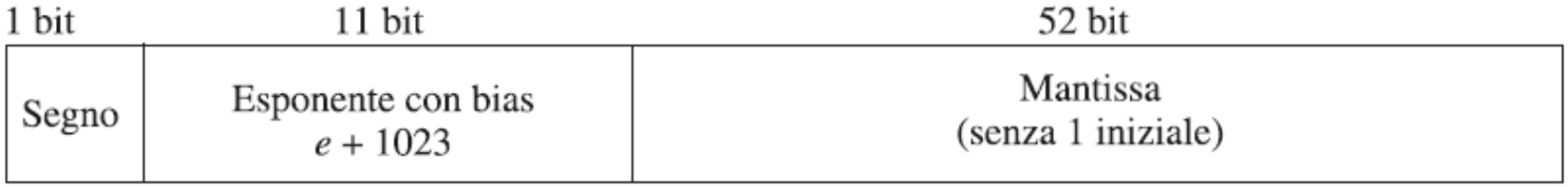
- Nella rappresentazione binaria si usa la **base 2** anziché **10**, ma il concetto non cambia
  - ovviamente, cambiando la base, cambia la mantissa
- Per convenzione la prima cifra significativa si trova immediatamente a sinistra del punto decimale
  - Quindi si moltiplica o si divide per 2 spostando a sinistra o a destra la posizione della virgola
    - $10011.101 = 1.0011101 \ \ 2^4$
    - $0.1101 = 1.101 \ 2^{-1}$

# Numeri in virgola mobile

- Lo standard internazionale **IEEE 754** definisce il formato e la disposizione dei bit di mantissa ed esponente (con i relativi segni)
  - Definito da IEEE Computer Society (Institute of Electrical and Electronics Engineers)
  - due formati, 32 bit e 64 bit




Singola precisione



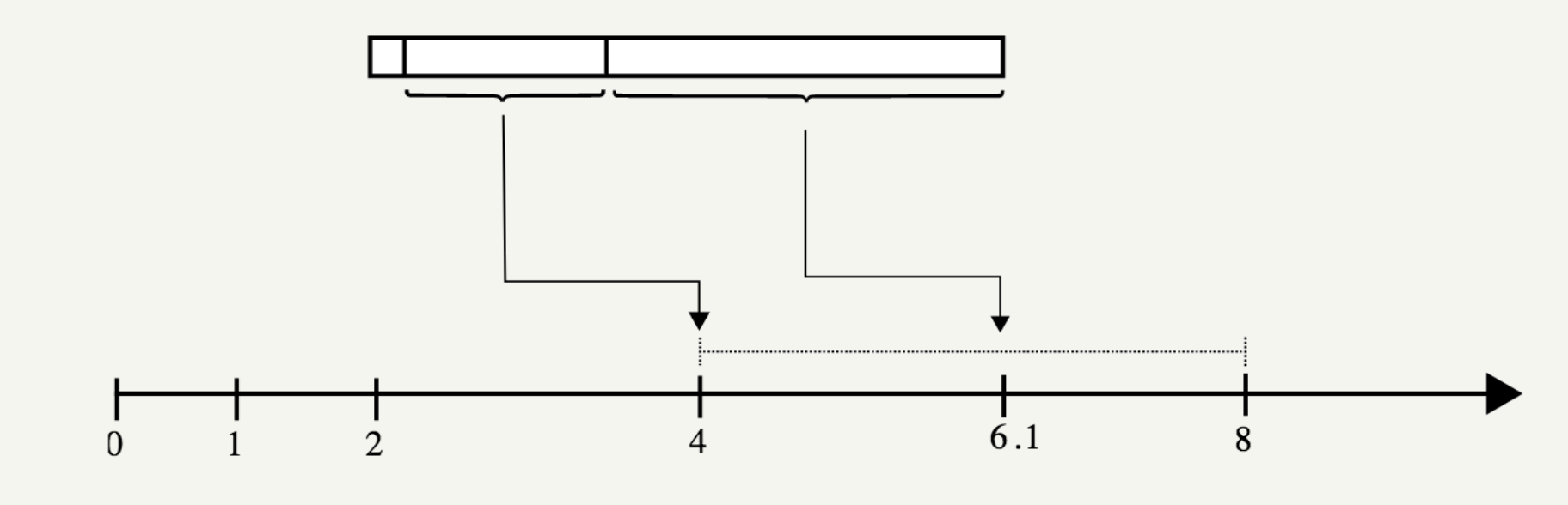

Doppia precisione

# Numeri speciali in virgola mobile

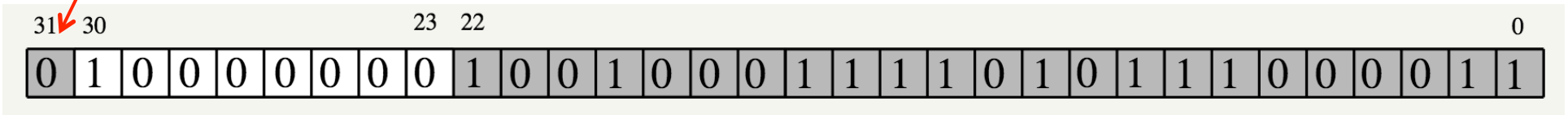
- La rappresentazione in virgola mobile dello standard **IEEE 754** include alcuni valori speciali
- Zero
  - Esponente = -127 (esponente con bias = 00000000)
  - Mantissa = 0
- Infinito
  - Esponente = +128 (esponente con bias = 11111111)
  - Mantissa = 0
- NaN (Not a Number, "non è un numero", es.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

# Visualizzazione di esponente e mantissa

- L'esponente indica l'intervallo di potenze di 2 nel quale il numero e' definito
- Tale intervallo e' suddiviso in $2^{23}$ parti
  - La mantissa indica la porzione di quell'intervallo che deve essere considerata per ottenere il numero



Positivo




$(10000000)_2 = 128$
Tolgo il bias: 128-127 = 1
E=1

$2^{-1}+2^{-4}+2^{-8}+2^{-9}+2^{-10}+2^{-11}+2^{-13}+2^{-15}+2^{-16}+2^{-17}+2^{-22}+2^{-23} =$

Oppure

$(2^{22}+2^{19}+2^{15}+2^{14}+2^{13}+2^{12}+2^{10}+2^{8}+2^{7}+2^{6}+2^{1}+2^{0}) \times 2^{-23} =$

M = 0.57000005460003

Valore: segno $1.M \times 2^E = +\ 1.57000005460003 * 2 = 3.1400001092$

# Virgola mobile a 32 bit (IEEE754)

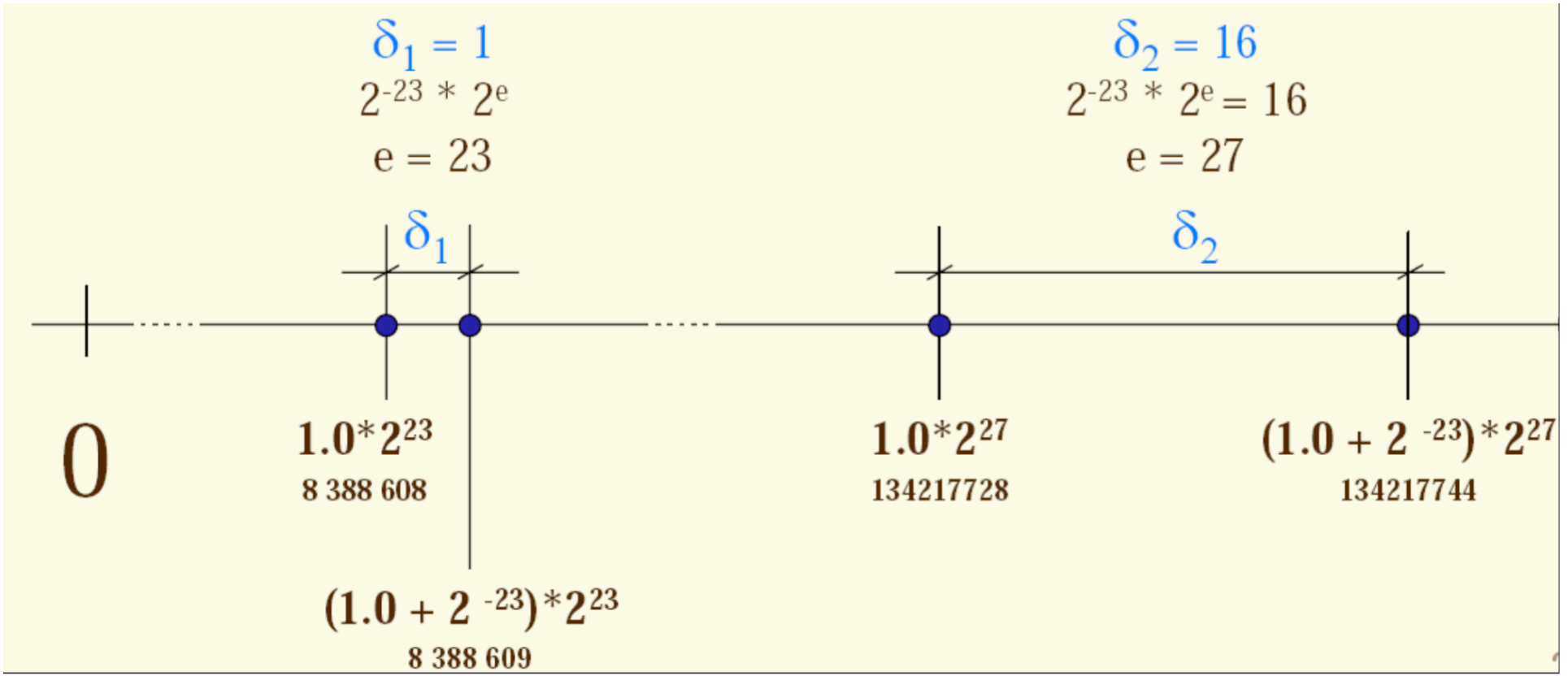
- Il numero più piccolo (positivo) rappresentabile è
  $\mathbf{1.00000000000000000000000_2 \times 2^{-126} \sim 1.8 \times 10^{-38}}$
- Il numero più grande rappresentabile è
  $\mathbf{1.11111111111111111111111_2 \times 2^{+127} \sim 3.4 \times 10^{+38}}$

- 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 $\delta$ è

$$\delta = 2^{-23} \times 2^{E}$$ (E è il valore dell'esponente)

# Densità dei numeri in virgola mobile

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



# Arrotondamento

- 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 x 100, il risultato è un numero appena inferiore a 435
- L'errore di arrotondamento viene amplificato dalla moltiplicazione
  - Provare per credere: scrivere un programma che stampi il risultato di 4.35 * 100. Dovreste visualizzare 434.99999999999994

# codice

# Errori nelle somme in virgola mobile

Si consideri la somma $10.5_{10} + 0.125_{10}$

- **Converto in binario e "traslo" il risultato per avere la cifra più significativa a sinistra del punto**
  - $10.5_{10} = 1010.1_2 = 1.0101_2 \times 2^3$
  - $0.125_{10} = 0.001_2 = 1.0_2 \times 2^{-3}$

- Per eseguire la somma bisogna **riportare entrambi i termini allo stesso esponente:**
  - Quindi $1.0_2 \times 2^{-3}$ **diventa** $0.000001 \times 2^3$
  - **10.5+0.125 =**
    - $1.0101_2 \times 2^3 + 0.000001_2 \times 2^3 = 1.010101_2 \times 2^3$

# Errori nelle somme in virgola mobile

- Se il numero di bit destinati alla mantissa fosse stato inferiore a 6, ad esempio 4 l'operazione avrebbe dato come risultato
  - **10.5+0.125 = 10.5 !!!!!!**
    - $1.0101_2 \times 2^3 + 0.0000_2 \times 2^3 = 1.010_2 \times 2^3$

a causa della necessaria approssimazione introdotta dalla rappresentazione

# Base 16 e base 8

# 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
  - **$01111111_2$ => $0111$ | $1111_2$ => $7F_{16}$ = 0x7F**
- Per convertire un numero naturale da esadecimale a binario si convertono le sue singole cifre

# 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
  - $(100010)_2 = (42)_8 = 042$
- 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

# 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

# UNICODE

- Codice **UNICODE** (http://www.unicode.org)
  - Usa 2 byte (16 bit) per ciascun carattere
  - Si possono rappresentare $\mathbf{2^{16} = 65536}$ 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

# Rappresentazione di caratteri

- Codice **ASCII** (American Standard Code for Information Interchange)
  - 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: **compatibilita'**

# ASCII Table

# 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 | [ENQUIRY] | 37 | 25 | % | 69 | 45 | E | 101 | 65 | e |
| 6 | 6 | [ACKNOWLEDGE] | 38 | 26 | & | 70 | 46 | F | 102 | 66 | f |
| 7 | 7 | [BELL] | 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 | [NEGATIVE 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 | [ENG 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] |

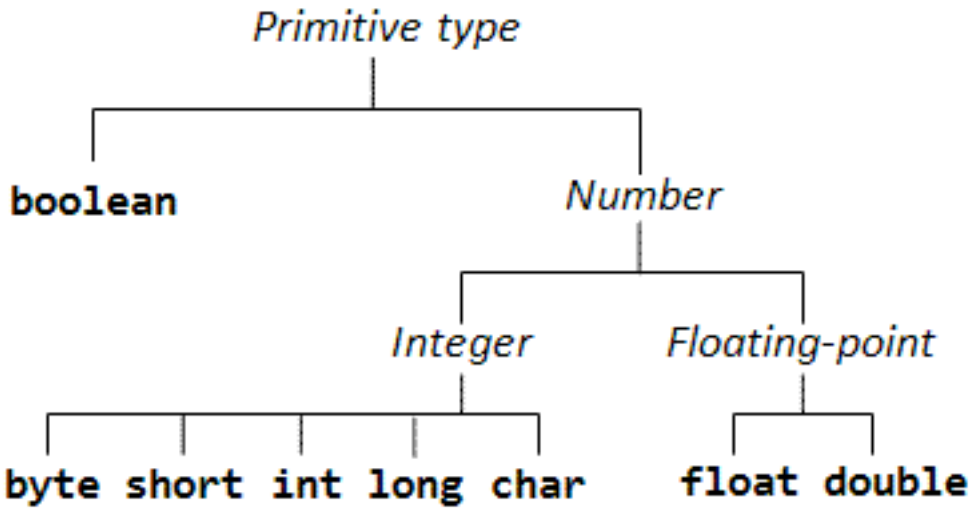
# Tipi di dato in Java

# Tipi di dati, valori e variabili

- Java è un linguaggio fortemente tipizzato e quindi ogni dato è di un ben preciso tipo noto al momento della compilazione del programma

- I tipi di dati in Java possono essere:
  - tipi primitivi
  - riferimenti ad un oggetto

# Gerarchia di tipi di dato primitivi in Java



# Tipi di dati primitivi - valori

- Dati numerici
  - Interi (complemento a 2)

Limiti dettati da quanto visto nella Rappresentazione delle informazioni!

$[-2^{n-1}, 2^{n-1}-1]$

| | | | |
|---|---|---|---|
| `byte` | 1 byte | -128 a 127 | con segno |
| `short` | 2 byte | -32768 a 32767 | |
| `int` | 4 byte | $-2*10^9$ a $2*10^9$ circa | |
| `long` | 8 byte | $-10^{20}$ a $10^{20}$ circa | |
| `char` | 2 byte | 0 a 65535 unicode | |

  - virgola mobile (numeri frazionari)

| | | |
|---|---|---|
| `float` | 4 byte | $\pm 10^{38}$, 7 cifre significative |
| `double` | 8 byte | $\pm 10^{308}$, 15 cifre significative |

- Booleani
  - `boolean true` o `false`

# Domanda

- Perché il massimo numero rappresentabile in Java con una variabile di tipo **int** è 2147483647?
- Una variabile **int** è rappresentata con **32 bit in complemento a 2**
- Quindi il massimo numero esprimibile è $\mathbf{2^{31} - 1 = 2147483647}$

# Tipi primitivi - descrizione

- **byte:** Variabile con segno (con rappresentazione in complemento a due) che rappresenta valori in un range [-128 e 127] (estremi inclusi)
- **short:** Numeri interi (con segno) in un range [-32768, 32767]
- **int:** Numeri interi (per default con segno, signed) in un range [$-2^{31}$, $2^{31}-1$]
- **long:** Numeri interi (per default con segno, signed) in un range [$-2^{63}$, $2^{63}-1$].

- Da Java 8 int e long possono essere *unsigned (senza segno)*
  - *unsigned int:* [0, $2^{32}-1$]
  - *unsigned long :* [0, $2^{64}-1$]

# Tipi primitivi - descrizione

- **float**: Numeri in virgola mobile in singola precisione secondo la specifica IEEE 754, utilizzando la rappresentazione segno, mantissa esponente.
  $(-1)^{segno} * mantissa * 2^{esponente}$

- **double**: Numeri in virgola mobile in doppia precisione secondo la specifica IEEE 754. La precisione con cui vengono rappresentati i numeri aumenta in virtù dell'aumento del numero di bit utilizzati.

# Tipi primitivi - descrizione

- boolean: serve a rappresentare solamente 2 valori: vero o falso (true o false).

- char: È utilizzato per la memorizzazione di caratteri del charset Unicode) nel range ['0x0000', '0xffff'] (in esadecimale) o equivalentemente [0,65535].

# Valori massimi e minimi

- Se servono i valori massimi/minimi dei numeri rappresentati con i vari tipi di dati non occorre ricordarli
- il pacchetto **java.lang** della libreria standard contiene una classe per ciascun tipo di dati fondamentali, in cui sono definiti questi valori come costanti

| | | |
|---|---|---|
| byte | **Byte.MIN_VALUE** | **Byte.MAX_VALUE** |
| short | **Short.MIN_VALUE** | **Short.MAX_VALUE** |
| int | **Integer.MIN_VALUE** | **Integer.MAX_VALUE** |
| long | **Long.MIN_VALUE** | **Long.MAX_VALUE** |
| float | **Float.MIN_VALUE** | **Float.MAX_VALUE** |
| double | **Double.MIN_VALUE** | **Double.MAX_VALUE** |

# Attenzione!

- In Java tutti i tipi di dati fondamentali per ***numeri interi*** usano internamente la rappresentazione in ***complemento a due***

- La JVM ***non segnala le condizioni di overflow*** nelle operazioni aritmetiche **si ottiene semplicemente un risultato errato**

# Attenzione

- L'unica operazione **aritmetica tra numeri interi** che genera una **eccezione** è la divisione con divisore zero: **ArithmeticException**

- Invece la dividendo per zero un numero float/ double si ottiene +/- Infinity

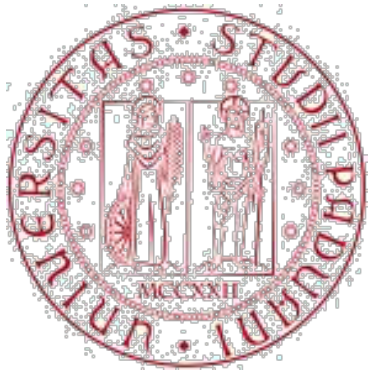
# Attenzione

- Quando l'intervallo di variabilità del tipo **int** non soddisfa le esigenze numeriche del problema, si usa il tipo **long**
  - massimo valore assoluto con una variabile long: circa **9** miliardi di miliardi
  - per assegnare un valore a una variabile **long** bisogna aggiungere un carattere **L** alla fine (Java assume che le costanti numeriche siano interi).
    - long l = 134584548667480648 20; **ERRORE**
    - long l = 134584548667480648 20L; **OK**

DIPARTIMENTO
DI INGEGNERIA
DELL'INFORMAZIONE

# Breve riassunto

## Variabili e costanti



# Le variabili

- In Java ogni **variabile** ha
  - **nome :** identificativo che viene utilizzato per "usare" la variabile
  - **tipo :**
    - determina l'insieme dei valori ammissibili (assegnabili)
    - determina l'occupazione di memoria
  - **contenuto :** valore associato
  - alcuni **attributi**
    - determinano la visibilità (public, protected, private) o specificano tipi particolari di variabili (static, final)

# Definizione (o dichiarazione) di variabili

- Sintassi:

```
nomeTipo nomeVariabile;
```

```
nomeTipo nomeVariabile = espressione;
```

- Scopo: definire la nuova variabile ***nomeVariabile***, di tipo ***nomeTipo***, ed eventualmente assegnarle il valore iniziale ***espressione***

# 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 ==);
  altri linguaggi usano simboli diversi per l'assegnazione (ad esempio, in linguaggio Pascal, :=)

# Definizione di costante

- Sintassi: `final nomeTipo NOME_COSTANTE = espressione;`

- Scopo: definire la costante NOME_COSTANTE di tipo nomeTipo, assegnandole il valore espressione, che non potrà più essere modificato
  - Nota: il compilatore segnala come errore semantico il tentativo di assegnare un nuovo valore ad una costante, dopo la sua inizializzazione
- Di solito in Java si usa la seguente convenzione
  - i nomi di costanti sono formati da lettere maiuscole
  - i nomi composti si ottengono attaccando le parole successive alla prima con un carattere di sottolineatura

# Literals, promozioni e casting

# Literals /1

- I valori costanti (=literals) per i numeri sono interpretate da Java come tipi int e double, a seconda che abbiano o meno la parte frazionaria
- Interi
  - Se il valore ricade nell'intervallo di un tipo di dato meno capace (byte o short) allora l'assegnamento avviene senza problemi, altrimenti errore in compilazione

```
byte value1 = 10;    //OK
byte value2 = 128;   //errore in compilazione
                     //max consentito 127

short value3 = 200;    //OK
short value4 = 33000; // errore in compilazione
                      // max consentito 32767
```

# Literals /2

- Interi
  - Se il valore è più grande del range del tipo int, ma rientra nel range del tipo long, devo esplicitare che si tratta di un valore long, aggiungendo **L** alla fine del numero

```
long l = 13458454867480648 20;     ERRORE
long l = 13458454867480648 20L;    OK
```

# Literals /3

- Numeri in virgola mobile
  - Anche se il valore ricade nell'intervallo del tipo float ho errore in compilazione
  - Devo esplicitamente dire che il valore va interpretato come float per poterlo assegnare: aggiungo f alla fine del numero
  - Posso anche inizializzare un numero in virgola mobile con un valore intero: in questo caso semplicemente la parte frazionaria corrispondera' a 0.

```
float f1 = 2.35;    // ERRORE
float f2 = 2.35f;   // OK
float f3 = 2;       // OK
System.out.println(f3);  // stampa: 2.0
```

# Literals /4

- Nota: non ci sono solo valori costanti numerici. Anche le stringhe o i caratteri hanno "literals";

```
char c = 'A';

String name = "Cinzia";
```

# Promozioni e casting

- Può capitare di dover spostare dei valori numerici fra variabili di tipo diverso
  - se un valore contenuto in una variabile di un certo tipo deve essere assegnato ad una variabile di un tipo "più capace", ad esempio:

```
byte b = 100;
int i = b;
```

OK!

  - il compilatore esegue una conversione automatica (***promozione***)

# Promozioni e casting

- Può capitare di dover spostare dei valori numerici fra variabili di tipo diverso
  - se un valore contenuto in una variabile di un certo tipo deve essere assegnato ad una variabile di un tipo "meno capace", ad esempio:

```
int i = 128;
byte b = i;
```

NO!

  - il compilatore non esegue una conversione ***perché potrebbe portare alla perdita di cifre significative***
  - provoca un errore in compilazione

    error: incompatible types: possible lossy conversion from int to byte byte b = i;

# Promozioni e casting

- in alcuni casi, il controllo del compilatore può risultare inopportuno

```
int i = 10;
byte b = i;
```

NO!

- il valore 10 può essere assegnato ad una variabile di tipo byte senza errori
- per ovviare a questo inconveniente, è possibile effettuare un **cast** (***conversione forzata)***

```
int i = 100;
byte b = (byte)i;
```

OK!

- in questo modo il compilatore viene "svincolato" dal ruolo di controllo della compatibilità fra tipo delle variabili e tipo dei valori assegnati

# Promozioni e casting

- Il casting esplicito si può fare anche tra tipi di dato diversi:

```
int    xInt;
double xDouble = 223.8644;

xInt = (int) xDouble;
```

- Il valore di xInt è 223 (la parte decimale viene troncata)
- E se faccio un cast esplicito a byte?

# wooclap