



Alma Mater Studiorum-Università di Bologna  
Scuola di Ingegneria

---

Tipi base & dintorni  
**ESERCITAZIONE AUTONOMA**

*Corso di Laurea in Ingegneria Informatica*  
Anno accademico 2021/2022

**Prof. ENRICO DENTI**

*Dipartimento di Informatica – Scienza e Ingegneria (DISI)*



# RIASSUNTO (1/3)

---

- In C, il main è scritto *a livello di file*
  - nessun livello di protezione specifico
- In Java e C#, il main è scritto *dentro una classe*
  - è *statico*, perché deve esistere dall'inizio alla fine del programma
  - è *pubblico*, perché deve essere visibile dall'esterno
- In Scala, il main è scritto *dentro un object*
  - è *implicitamente statico*, perché l'intero object lo è
  - è *implicitamente pubblico*, perché quello è il default in Scala
- In Kotlin, il main è scritto *a livello di file* (ma è un package..)
  - è *implicitamente statico*, perché il package di cui fa parte lo è
  - è *implicitamente pubblico*, perché quello è il default in Kotlin



# RIASSUNTO (2/3)

---

- Per questo motivo:
  - in C, gli argomenti dalla linea di comando *includono il nome del file*
  - in Java, C#, Scala, Kotlin invece, *non lo includono*
- Il main riceve *sostanzialmente* un array di stringhe
  - in C, ciò significa passargli un intero (argc) e dei puntatori a carattere (argv)
  - in Java, C#, Scala, Kotlin il main riceve un unico argomento: un *array di String* (la sintassi varia un po' da un linguaggio all'altro)



# RIASSUNTO (3/3)

---

- Per stampare:
  - in C, si include `stdio.h` e si usa `printf`
  - **in Java**, non si include nulla (link dinamico): si invoca il servizio `println` del componente `System.out`
  - **in C#**, non si include nulla (link dinamico): si invoca il servizio `WriteLine` del componente `System.Console`
  - **in Scala**, non si include nulla (link dinamico): si invoca il servizio `println` (il componente `Console` è implicito)
  - **in Kotlin**, non si include nulla (link dinamico): si invoca il servizio `println` (il componente `kotlin.io` è implicito)



# STAMPARE GLI ARGOMENTI

- Serve un ciclo
  - in C, da 1 ad `argc-1` (escludiamo il nome stesso del programma)
  - in Java e C#, da **0** a **`args.length`**-1 (in C#, **`Length`**)
  - in Scala e Kotlin, da **0** a **`args.size`**-1

```
public class Esempio1Var {  
    public static void main(String[] args){  
        if (args.length == 0)  
            System.out.println("Nessun argomento");  
        else  
            for (int i=0; i<args.length; i++){  
                System.out.println(  
                    "argomento " + i + ": " + args[i]);  
            }  
    }  
}
```

Java

~C#

# STAMPARE GLI ARGOMENTI

- Serve un ciclo
  - in C, da 1 ad `argc-1` (escludiamo il nome stesso del programma)
  - in Java e C#, da 0 a `args.length` (in C#, `Length`)
  - in Scala e Kotlin, da 0 a `args.size-1`

```
public class Esempio1Var {  
    public static void main(String[] args) {  
        if (args.length == 0)  
            System.out.println("No arguments");  
        else  
            for (int i=0; i<args.length; i++) {  
                System.out.println(  
                    "arg " + i + ": " + args[i]);  
            }  
    }  
}
```

Java

La proprietà pubblica **length** (read-only) contiene la dimensione fisica dell'array

**NOVITÀ:** si possono definire variabili *in ogni punto del programma*, non più solo all'inizio come in C; in particolare **si può definire l'indice dentro al ciclo `for`**, con scope limitato al ciclo stesso.



# STAMPARE TUTTI GLI ARGOMENTI

- Serve un ciclo
  - in C, da 1 ad `argc-1` (escludiamo il nome stesso del programma)
  - in Java e C#, da 0 a `args.length` (in C#, `Length`)
  - in Scala e Kotlin, da 0 a `args.size-1`

```
public class Esempio1Var {  
    public static void main(String[] args){  
        if (args.length == 0)  
            System.out.println("Nessun argomento");  
        else  
            for (int i=0; i<args.length; i++){  
                System.out.println(  
                    "argomento " + i + ": " + args[i]);  
            }  
    }  
}
```

Java

**NOVITÀ:** l'operatore **+** concatena stringhe  
e nel farlo converte anche in stringa ciò che stringa non è! ☺

# STAMPARE TUTTI GLI ARGOMENTI

```
fun main(args: Array<String>) {  
    if (args.size == 0)  
        println("Nessun argomento")  
    else  
        for (i: Int in 0..args.size-1) {  
            println("argomento " + i + ": " + args[i]);  
        }  
}
```

Kotlin

In Kotlin, il ciclo **for** ha una sintassi innovativa che evita di esprimere direttamente l'incremento

La proprietà pubblica **size** (read-only) contiene la dimensione fisica dell'array

```
def main(args: Array[String]): Unit = {  
    if (args.size == 0)  
        println("Nessun argomento")  
    else  
        for (i: Int <- 0 to args.size-1) {  
            println("argomento " + i + ": " + args(i));  
        }  
}
```

Scala

Anche in Scala, il ciclo **for** ha una nuova sintassi

La proprietà pubblica **size** (read-only) contiene la dimensione fisica dell'array

In Scala, si accede agli array con le parentesi tonde **()** anziché quadre



# ESECUZIONE

```
C:> java Esempio1Var alfa beta gamma
```

Java

```
Argomento 0: alfa
```

```
Argomento 1: beta
```

```
Argomento 2: gamma
```

```
C:> java Esempio1Var alfa "beta gamma"
```

```
Argomento 0: alfa
```

```
Argomento 1: beta gamma
```

Java

```
1 public class Esempio1Var {  
2  
3     public static void Main(string[] args){  
4         if (args.Length == 0)  
5             System.Console.WriteLine("Nessun argomento");  
6         else  
7             for (int i=0; i<args.Length; i++)  
8                 System.Console.WriteLine("Argomento " + i + ": " + args[i]);  
9     }  
10 }  
11  
12 }
```

```
----- C# Run -----  
argomento 0: alfa  
argomento 1: beta  
argomento 2: gamma  
argomento 3: delta
```

C#

```
1 object CicloArgomenti {  
2     def main(args: Array[String]) : Unit = {  
3         if (args.size == 0)  
4             println("Nessun argomento");  
5         else  
6             for (i <- 0 to args.size-1){  
7                 println("argomento " + i + ": " + args[i]);  
8             }  
9     }  
10 }  
11  
12 }
```

```
----- Scala run -----  
argomento 0: alfa  
argomento 1: beta  
argomento 2: gamma  
argomento 3: delta
```

Scala

```
fun main(args: Array<String>) {  
    if (args.size == 0)  
        println("Nessun argomento");  
    else  
        for (i in 0..args.size-1){  
            println("argomento " + i + ": " + args[i]);  
        }  
}
```

```
argomento 0: alfa  
argomento 1: beta  
argomento 2: gamma  
argomento 3: delta
```

Kotlin



# CALCOLI MATEMATICI

- Si usa la libreria matematica
  - in C, da includere nell'header: `#include <math.h>`
  - in Java e C#, **Math** (in C#, **System.Math**): non c'è `#include`

```
public class Esempio2 {  
    public static void main(String[] args){  
        System.out.println( Math.sin(Math.PI/3) );  
    }  
}
```

- Per prove rapide in Java si può anche usare **jshell**
  - ambiente interattivo, accetta singole istruzioni

```
C:\Users\Enrico Dentice> jshell  
Welcome to JShell -- version 11.0.1  
For an introduction type: /help intro  
  
jshell> System.out.println( Math.sin(Math.PI/3) );  
0.8660254037844386
```



# NUMERI INTERI

- Interi con segno

- Java: **byte** (1 byte) -128 ... +127 Scala/Kotlin: **Byte**  
C#: **sbyte**
- Java/C#: **short** (2 byte) -32768 ... +32767 Scala/Kotlin: **Short**
- Java/C#: **int** (4 byte)  $-2 \cdot 10^9$  .....  $+2 \cdot 10^9$  Scala/Kotlin: **Int**
- Java/C#: **long** (8 byte)  $-9 \cdot 10^{18}$  .....  $+9 \cdot 10^{18}$  Scala/Kotlin: **Long**

NB: le costanti `long` terminano con la lettera `L`

- Interi senza segno (solo C#)

- **byte** (1 byte) 0... 255
- **ushort** (2 byte) 0... 65535
- **uint** (4 byte) 0...  $4 \cdot 10^9$
- **ulong** (8 byte) 0...  $1.8 \cdot 10^{19}$

NB: le costanti `ulong` terminano con la lettera `L`

# ESPERIMENTI CON GLI INT

- Esperimenti in C

```
int main() {  
    int a = 32767;  
    int b = a+1;  
    printf("%d, %d\n", a, b);  
    short int c = 32767;  
    short int d = c+1;  
    printf("%d, %d\n", c, d);  
}
```

Il compilatore C è di bocca buona, accetta tutto  
Però poi, a run time...

Compiled in 302.644 ms  
Executing...  
32767, 32768  
32767, -32768

- Esperimenti in Java

```
jshell> int a= 32767; int b = a+1;  
a ==> 32767  
b ==> 32768  
  
jshell> short a = 32767; short b = a+1;  
a ==> 32767  
Error:  
incompatible types: possible lossy conversion from int to short  
short b = a+1;  
          ^_^^
```

Il compilatore Java intercetta  
l'errore, *ma solo* perché  
aveva convertito a+1 in int!

# ESPERIMENTI CON GLI INT

- Esperimenti in C

```
int main() {  
    char a = 125;  
    printf("%d\n", a);  
    a++; ++a;  
    printf("%d\n", a);  
    printf("%d\n", ++a);  
}
```

Anche qui, overflow

```
Compiled in 257.473 ms  
Executing...  
125  
127  
-128
```

- Esperimenti in Java

```
jshell> byte a = 125;  
a ==> 125  
  
jshell> a++;  
$6 ==> 125  
  
jshell> ++a;  
$7 ==> 127  
  
jshell> ++a;  
$8 ==> -128
```

E infatti stavolta, che non ci sono conversioni di mezzo, non si accorge del problema → overflow

# NUMERI REALI

- Standard IEEE-754

- Java/C#: **float** (4 byte)  $-10^{45} \dots +10^{38}$  Scala/Kotlin: **Float**
- Java/C#: **double** (8 byte)  $-10^{324} \dots +10^{308}$  Scala/Kotlin: **Double**
- **float**: circa 6-7 cifre decimali significative (precisione:  $6 \cdot 10^{-8}$ )  
NB: le costanti **float** terminano con la lettera F
- **double**: circa 14-15 cifre decimali significative (precisione:  $1 \cdot 10^{-16}$ )

- Fuori standard

- solo C#: **decimal** (16 byte)  $-10^{28} \dots +10^{28}$
- **decimal**: circa 28-29 cifre decimali significative (precisione:  $1 \cdot 10^{-28}$ )  
NB: le costanti **decimal** terminano con la lettera M  
PRO: molto preciso, perché internamente usa la base 10;  
ciò è utile nei calcoli finanziari  
CONTRO: molto più lento (20 volte) e range più ridotto



# NUMERI REALI: COMPATIBILITÀ

- In Java, C#, Scala sono ammessi solo *gli assegnamenti che non causano perdita di informazione*.

Quindi, ad esempio:

La frase `double x = 3.54F;` è *lecita* (da `float` a `double` non si perde precisione)

La frase `float f = 3.54;` è *illecita* (da `double` a `float` si perderebbe precisione)

In C#, anche un valore `decimal` non può essere assegnato a una variabile `float` o `double`, poiché si perderebbe in precisione.

La frase `double d = 3.54M;` è quindi *illecita* (come sopra).



# NUMERI REALI: ESEMPIO IN Java

- Esperimento al volo con `jshe11`

```
jshe11> double x = Math.sin(Math.PI/3);  
x ==> 0.8660254037844386  
  
jshe11> System.out.println(x);  
0.8660254037844386  
  
jshe11> float f = Math.sin(Math.PI/3);  
Error:  
incompatible types: possible lossy conversion from double to float  
float f = Math.sin(Math.PI/3);  
           ^-----^  
  
jshe11> float f = (float) Math.sin(Math.PI/3);  
f ==> 0.8660254
```





# Java: LIBRERIA MATEMATICA

- La libreria matematica Java è nella classe **Math**
  - funzioni statiche per qualunque calcolo
- Documentazione Javadoc:

OVERVIEW MODULE PACKAGE **CLASS** USE TREE DEPRECATED INDEX HELP Java SE 13 & JDK 13

SUMMARY: NESTED | FIELD | CONSTR | METHOD    DETAIL: FIELD | CONSTR | METHOD    SEARCH:

**Module** java.base  
**Package** java.lang  
**Class Math**

java.lang.Object  
java.lang.Math

```
public final class Math
extends Object
```

The class Math contains methods for performing basic numeric operations such as the elementary exponential, logarithm, square root, and trigonometric functions.

Unlike some of the numeric methods of class StrictMath, all implementations of the equivalent functions of class Math are not defined to return the bit-for-bit same results. This relaxation permits better-performing implementations where strict reproducibility is not required.

By default many of the Math methods simply call the equivalent method in StrictMath for their implementation. Code generators are encouraged to use platform-specific native libraries or microprocessor instructions, where available, to provide higher-performance implementations of Math methods. Such higher-performance implementations still must conform to the specification for Math.

The quality of implementation specifications concern two properties, accuracy of the returned result and monotonicity of the method. Accuracy of the floating-point Math methods is measured in terms of *ulps*, units in the last place. For a given floating-point format, an ulp of a specific real number value is the distance between the two floating-point values bracketing that numerical value. When discussing the accuracy of a method as a whole rather than at a specific argument, the number of ulps cited is for the worst-case error at any argument. If a method always has an error less than 0.5 ulps, the method always returns the floating-point number nearest the exact result; such a method is *correctly rounded*. A correctly rounded method is generally the best a floating-point approximation can be; however, it is impractical for many floating-point methods to be correctly rounded. Instead, for the Math class, a larger error bound of 1 or 2 ulps is allowed for certain methods. Informally, with a 1 ulp error bound, when the exact result is a representable number, the exact result should be returned as the computed result; otherwise, either of the two floating-point values which bracket the exact result may be returned. For exact results large in magnitude, one of the endpoints of the bracket may be infinite. Besides accuracy at individual arguments, maintaining proper relations between the method at different arguments is also important. Therefore, most methods with more than 0.5 ulp errors are required to be *semi-monotonic*: whenever the mathematical function is non-decreasing, so is the floating-point approximation, likewise, whenever the mathematical function is non-increasing, so is the floating-point approximation. Not all approximations that have 1 ulp accuracy will automatically meet the monotonicity requirements.

The platform uses signed two's complement integer arithmetic with int and long primitive types. The developer should choose the primitive type to ensure that arithmetic operations consistently produce correct results, which in some cases means the operations will not overflow the range of values of the computation. The best practice is to choose the primitive type and algorithm to avoid overflow. In cases where the size is int or long and overflow errors need to be detected, the methods addExact, subtractExact, multiplyExact, and toIntExact throw an ArithmeticException when the results overflow. For other arithmetic operations such as divide, absolute value, increment by one, decrement by one, and negation, overflow occurs only with a specific minimum or maximum value and should be checked against the minimum or maximum as appropriate.

ALMA MATER STUDIORUM - UNIVERSITÀ DI BOLOGNA



# Java: LIBRERIA MATEMATICA

- La libreria matematica Java è nella classe **Math**
  - funzioni statiche per qualunque calcolo
- Documentazione Javadoc:

Field Summary		
Fields		
Modifier and Type	Field	Description
static double	<b>E</b>	The double value that is closer than any other to <i>e</i> , the base of the natural logarithms.
static double	<b>PI</b>	The double value that is closer than any other to <i>pi</i> , the ratio of the circumference of a circle to its diameter.

Method Summary		
All Methods	Static Methods	Concrete Methods
Modifier and Type	Method	Description
static double	<b>abs</b> (double a)	Returns the absolute value of a double value.
static float	<b>abs</b> (float a)	Returns the absolute value of a float value.
static int	<b>abs</b> (int a)	Returns the absolute value of an int value.
static long	<b>abs</b> (long a)	Returns the absolute value of a long value.
static double	<b>acos</b> (double a)	Returns the arc cosine of a value; the returned angle is in the range 0.0 through <i>pi</i> .
static int	<b>addExact</b> (int x, int y)	Returns the sum of its arguments, throwing an exception if the result overflows an int.
static long	<b>addExact</b> (long x, long y)	Returns the sum of its arguments, throwing an exception if the result overflows a long.
static double	<b>asin</b> (double a)	Returns the arc sine of a value; the returned angle is in the range <i>-pi/2</i> through <i>pi/2</i> .
static double	<b>atan</b> (double a)	Returns the arc tangent of a value; the returned angle is in the range <i>-pi/2</i> through <i>pi/2</i> .
static double	<b>atan2</b> (double y, double x)	Returns the angle <i>theta</i> from the conversion of rectangular coordinates (x, y) to polar coordinates (r, <i>theta</i> ).
static double	<b>cbrt</b> (double a)	Returns the cube root of a double value.
static double	<b>ceil</b> (double a)	Returns the smallest (closest to negative infinity) double value that is greater than or equal to the argument and is equal to a mathematical integer.



# ESEMPI (anche CURIOSI..)

- Funzioni "classiche"
  - `rint` arrotonda un valore reale all'intero più vicino (risultato reale)
  - `round` arrotonda un valore reale all'intero più vicino (risultato intero)
  - `pow` effettua l'elevamento a potenza ( $x^y$ )
- Ci sono anche funzioni per operazioni "curiose"
  - `hypot` calcola  $x^2+y^2$  senza errori di overflow/underflow intermedi
  - `log1p` calcola  $\log_e(1+p)$

```
jshell> Math.hypot(3,4)
$1 ==> 5.0

jshell> Math.hypot(8,15)
$2 ==> 17.0

jshell> Math.hypot(1,1)
$3 ==> 1.4142135623730951
```

```
jshell> Math.pow(3,5)
$8 ==> 243.0
```

```
jshell> Math.log1p(0.3)
$4 ==> 0.26236426446749106

jshell> Math.log1p(0.1)
$5 ==> 0.09531017980432487
```

```
jshell> Math.rint(12.3456)
$6 ==> 12.0

jshell> Math.round(12.3456)
$7 ==> 12
```

# CALCOLARE SUI REALI

- In analisi matematica i calcoli sui reali sono spesso espressi *solo il modo simbolico*
- Un calcolatore, invece, *fa i calcoli* e questo può portare a risultati.. inattesi!
  - in matematica,  $0/a$  fa sempre 0, *senza segno*
  - e in Java..? C#..?

```
jshell> -0.0/1.0
$1 ==> -0.0

jshell> 0.0/1.0
$2 ==> 0.0

jshell> 0.0/1.0 == -0.0/1.0
$3 ==> true
```

Java mantiene il segno nel risultato, MA ciò non ne altera la semantica

```
1 using System;
2
3 public class Program
4 {
5     public static void Main()
6     {
7         Console.WriteLine(-0.0/1.0);
8         Console.WriteLine(0.0/1.0);
9         Console.WriteLine(-0.0/1.0==0.0/1.0);
10    }
11 }
```

```
0
0
True
```

C#, più correttamente, risponde semplicemente 0

# CALCOLARE SUI REALI

- In analisi matematica esistono  $\pm\infty$  e le **forme indeterminate**
  - in matematica,  $a/0$  fa  $\pm\infty$  (il segno è quello di  $a$ )
  - in matematica,  $0/0$  è una forma indeterminata
  - e in Java e C#..?

```
jshell> 3.0/0.0
$2 ==> Infinity

jshell> -3.0/0.0
$3 ==> -Infinity

jshell> (3.0/0.0)+(-3.0/0.0)
$4 ==> NaN

jshell> 0.0/0.0
$5 ==> NaN

jshell> (3.0/0.0)-(-3.0/0.0)
$6 ==> Infinity
```

Lo standard IEEE-754 incorpora le nozioni di **infinito** e **Not-A-Number (NaN)**

L'aritmetica dei **reali** perciò gestisce sia gli infiniti, sia le forme indeterminate! ☺☺

NB: ciò non si applica ai Decimal di C#

```
4 {
5     public static void Main()
6     {
7         Console.WriteLine(3.0/0.0);
8         Console.WriteLine(-3.0/0.0);
9         Console.WriteLine((3.0/0.0)-(-3.0/0.0));
10        Console.WriteLine((3.0/0.0)-(3.0/0.0));
11    }
12 }
```

```
∞
-∞
∞
NaN
```



# CALCOLARE SUGLI INTERI

- Ciò non vale per i tipi interi
  - in matematica, magari sì...
  - ... ma in Java e C#, no
  - MOTIVO: i tipi interi non seguono lo standard IEEE-754 !

L'aritmetica degli **interi** dà **errore** in caso di divisioni per zero.

```
jshell> 0 / 0
Exception java.lang.ArithmeticException: / by zero
at (#7:1)
```

```
C# Compilation error (line 7, col 21): Division by constant zero
```

# CARATTERI (e stringhe): ESEMPIO

- In Java, i caratteri sono Unicode (16 bit, UTF-16)
  - una costante carattere è racchiusa fra apici singoli
- Un carattere può essere convertito:
  - in `int` (o `short`, `long`) con un cast
  - in `String` con l'apposita funzione statica `String.valueOf`
    - per C#, funzione `Char.ToString`
- Esperimenti al volo con `jshell`

```
jshell> char ch = 'A'
ch ==> 'A'

jshell> (int) ch
$27 ==> 65

jshell> String s = String.valueOf(ch)
s ==> "A"
```

```
1 using System;
2
3 public class Program
4 {
5     public static void Main()
6     {
7         char ch = 'A';
8         Console.WriteLine(ch);
9         Console.WriteLine((int)ch);
10        string s = Char.ToString(ch);
11        Console.WriteLine(s);
12    }
13 }
```



C#



# CARATTERI (e stringhe): ESEMPIO

- Ogni carattere è rappresentato internamente in Unicode
  - ma non si vede direttamente..
- Si può ottenere facilmente la sequenza di byte UTF
  - bisogna però partire da una stringa, NON da un carattere
  - **in Java** si usa l'apposito metodo **getBytes (...)**

```
jshell> s.getBytes("UTF8")
$29 ==> byte[1] { 65 }

jshell> s.getBytes("UTF16")
$30 ==> byte[4] { -2, -1, 0, 65 }

jshell> s.getBytes("UTF32")
$31 ==> byte[4] { 0, 0, 0, 65 }
```

UTF8 è chiaro  
UTF32 anche  
**Ma... UTF16?**

In hex: FE, FF  
è il marcatore "little endian"





# CARATTERI (e stringhe): ESEMPIO

- Ogni carattere è rappresentato internamente in Unicode
  - ma non si vede direttamente..
- Si può ottenere facilmente la sequenza di byte UTF
  - bisogna però partire da una stringa, NON da un carattere
  - **in Java** si usa l'apposito metodo `getBytes (...)`

I due byte alti indicano se la sequenza che segue è *big endian* (MSB first) o *little endian* (LSB first)

```
jshell> s.getBytes("UTF16")
$30 ==> byte[4] { -2, -1, 0, 65 }
jshell> s.getBytes("UTF32")
$31 ==> byte[4] { 0, 0, 0, 65 }
```

UTF-16 allows a Byte Order Mark, a code point with the value **U+FEFF**, to **precede** the first actual coded value.

Everything in **Java** is stored in **big-endian** order. **Internally**, some platforms use **big-endian** or **little-endian (Intel)**.



# CARATTERI (e stringhe): ESEMPIO

- Ogni carattere è rappresentato internamente in Unicode
  - ma non si vede direttamente..
- Si può ottenere facilmente la sequenza di byte UTF
  - bisogna però partire da una stringa, NON da un carattere
  - in **C#** si usa l'apposito metodo **GetBytes (...)**

```
char ch = 'A';  
string s = Char.ToString(ch);  
Console.WriteLine(BitConverter.IsLittleEndian);  
Console.WriteLine(BitConverter.ToString(System.Text.Encoding.UTF8.GetBytes(s)));  
Console.WriteLine(BitConverter.ToString(System.Text.Encoding.Unicode.GetBytes(s)));  
Console.WriteLine(BitConverter.ToString(System.Text.Encoding.UTF32.GetBytes(s)));
```

In **C#** si percepisce la struttura interna: infatti, chiedendolo si scopre che è **little-endian** (Intel)

```
True  
41  
41-00  
41-00-00-00
```

41H = 65

Anche la stampa lascia percepire la struttura interna (Intel = **little-endian**)



# TIPI PRIMITIVI vs TIPI-OGGETTO in Java

- In Java i tipi base (`int`, `short`, `long`, `char`, `float`, `double`, `boolean`) sono *tipi primitivi*, non tipi di oggetti
  - non si può chiedere a un valore primitivo di «fare qualcosa»
    - non si può chiedere al carattere 'A' quale sia la sua controparte minuscola, o se sia una vocale, o altro
    - analogamente, non si può chiedere all'intero 12 se sia pari, o quale sia il suo segno, etc.
- Invece che "fare domande" a un valore primitivo bisogna *chiamare una qualche funzione di una qualche libreria*
- In Java, esistono classi "libreria" di nome *simile* a tutti i tipi primitivi: **Integer**, **Long**, **Character**, **Float**, **Double**, etc.
  - NON VANNO CONFUSE COI TIPI-OGGETTO DI Scala e Kotlin!



# TIPI PRIMITIVI vs TIPI-OGGETTO in Java

---

- Ad esempio, in Java la classe-libreria **Character** offre
  - i metodi **toUpperCase** / **toLowerCase** che producono la controparte maiuscola / minuscola del carattere fornito
  - il metodo **isWhiteSpace** che verifica se il carattere fornito è un "bianco" (spazio, tab, newline, ritorno a capo, form feed, etc.)

```
jshell1> Character.toUpperCase('a')  
$9 ==> 'A'  
  
jshell1> Character.toUpperCase('A')  
$10 ==> 'A'  
  
jshell1> Character.toUpperCase('è')  
$11 ==> 'È'
```

```
jshell1> Character.isWhitespace('a')  
$15 ==> false  
  
jshell1> Character.isWhitespace(' ')  
$16 ==> true  
  
jshell1> Character.isWhitespace('\t')  
$17 ==> true  
  
jshell1> Character.isWhitespace('\n')  
$18 ==> true  
  
jshell1> Character.isWhitespace('\r')  
$19 ==> true
```



# TIPI PRIMITIVI vs TIPI-OGGETTO in Java

- Ad esempio, in Java la classe-libreria **Character** offre
  - il metodo **digit** restituisce il valore numerico di quel certo carattere *nella base specificata*
    - ovviamente, il carattere dev'essere nel range fra 0 e base-1
    - altrimenti, viene restituito convenzionalmente -1 come segnale di errore

```
jshell> Character.digit('8',16)
$28 ==> 8
jshell> Character.digit('8',10)
$29 ==> 8
jshell> Character.digit('8',7)
$30 ==> -1
jshell> Character.digit('B',10)
$31 ==> -1
jshell> Character.digit('B',16)
$32 ==> 11
```

- Il carattere '8' in base 10 o 16 indica il valore intero *otto*, mentre in base 7 non esiste
- Analogamente, il carattere 'B' denota il valore *undici* in base 16, ma non esiste in base 10



# TIPI PRIMITIVI vs TIPI-OGGETTO in Java

- Analogamente, in Java la classe-libreria **Integer** offre
  - il metodo **signum** che restituisce il segno del numero fornito
  - i metodi **rotateLeft** / **rotateRight** che restituiscono il numero ottenuto facendo uno *shift left* / *right* dei bit del numero (ossia moltiplicandolo / dividendolo per 2, salvo overflow)

```
jshell> Integer.signum(12)
$33 ==> 1

jshell> Integer.signum(-12)
$34 ==> -1

jshell> Integer.signum(0)
$35 ==> 0
```

```
jshell> Integer.rotateLeft(6,1)
$42 ==> 12

jshell> Integer.rotateLeft(6,2)
$43 ==> 24

jshell> Integer.rotateRight(6,2)
$44 ==> -2147483647

jshell> Integer.rotateRight(6,1)
$45 ==> 3
```

6 → 00000000...00000110

6 shift left 1 → 00000000...00001100

6 shift left 2 → 00000000...00011000

6 shift right 1 → 00000000...00000011

6 shift right 2 → 10000000...00000001



# TIPI PRIMITIVI vs TIPI-OGGETTO in C#

- In C#, la differenza è più sfumata
  - i tipi primitivi in realtà sono classi
    - ad esempio, `int` è in realtà un sinonimo per la classe `Int32`
  - MA possono essere trattati "quasi sempre" come se fossero tipi primitivi standard, per comodità
  - tuttavia, la loro natura di "veri oggetti" fa comodo in più occasioni
- Ergo, in C# *si può* chiedere a un valore di «fare qualcosa»
  - anziché `string s = Char.ToString(ch)`
    - si può scrivere `string s = ch.ToString()`
      - per le costanti: `string s = 'B'.ToString()`
- Approccio più compiutamente «object-oriented»

Funzione  
statica

Metodo



# TIPI BASE in Scala e Kotlin

---

- In Scala e Kotlin, la dicotomia si supera: i tipi primitivi *non esistono più*, al loro posto veri ci sono veri *tipi-oggetto*!
- Conseguenza:
  - **Int, Float, Double...** sono *veri tipi di oggetti*
  - le costanti numeriche sono *veri oggetti*, istanze di tali tipi
- Quindi, in Scala e Kotlin *si può* (anzi, si deve...!) chiedere a un valore di «fare qualcosa»
  - approccio totalmente «object-oriented»



# TIPI BASE in Scala e Kotlin



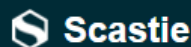
Kotlin Playground is an online sandbox to explore Kotlin programming language. Browse code samples.

```
/**
 * You can edit, run, and share this code
 * play.kotlinlang.org
 */

fun main() {
    var f : Float = 3.54.toFloat();
    var s : String = 'B'.toString();
    println(f);
    println(s);
}
```

3.54  
B

Osserva: in Scala, il metodo `toFloat` non vuole le parentesi di chiamata (principio di accesso uniforme)  
In Kotlin, invece, le parentesi di chiamata ci vogliono sempre (come in Java e C#)



Editor

Save New Format Run Message

Build Settings

```
1 object Example {
2
3   def main(args: Array[String]): Unit = {
4
5     var f : Float = 3.54.toFloat();
6     var s : String = 'B'.toString();
7     Console.println(f);
8     Console.println(s);
9   }
10
11 }
```

3.54  
B



# CONVERSIONI STRINGA → NUMERO

- Per convertire stringhe in numeri, in Java e C# si usano perciò apposite funzioni di libreria:
  - in Java, `Integer.parseInt` e `Double.parseDouble`
  - in C#, `Convert.ToInt32` e `Convert.ToDouble`
- In Scala e Kotlin, si invocano invece appositi metodi:
  - in Scala, `toInt` e `toDouble` (senza parentesi)
  - in Kotlin, `toInt` e `toDouble` (con parentesi)
- MA questi metodi pretendono stringhe «giuste»!
  - altrimenti... BOOM!

```
jshell1> Integer.parseInt("33")
$1 ==> 33

jshell1> Integer.parseInt("-3")
$2 ==> -3

jshell1> Integer.parseInt("-3a")
Exception java.lang.NumberFormatException:
```



# CONVERSIONI STRINGA → NUMERO

- In Java e C#, per evitare tale errore si può solo controllare la stringa prima di convertirla
- In Scala e Kotlin, ci sono metodi appositi che considerano anche il caso «sfortunato» - ad esempio, per il Double:
  - in Scala, `toDoubleOption` (senza parentesi)
  - in Kotlin, `toDoubleOrNull` (con parentesi)

```
object Example {  
  def main(args: Array[String]): Unit = {  
    var d : Double = "-3.4".toDouble;  
    Console.println(d);  
  
    d = "0.34".toDouble;  
    Console.println(d);  
  
    //d = "3a".toDouble; // explode  
    var e = "3a".toDoubleOption; // non explode  
    Console.println(e);  
  }  
}
```

```
-3.4  
0.34  
None
```

```
fun main() {  
  var d : Double = "-3.4".toDouble();  
  println(d);  
  
  d = "0.34".toDouble();  
  println(d);  
  
  //d = "3a".toDouble(); // explode  
  var e = "3a".toDoubleOrNull(); // non explode  
  println(e);  
}
```

```
-3.4  
0.34  
null
```

# Un piccolo esercizio completo

## Equazioni di 2° grado



# ESERCIZIO: EQUAZIONI DI 2° GRADO

- Specifica: fornire i coefficienti  $a$ ,  $b$ ,  $c$  dalla riga di comando
  - diventano i tre argomenti (stringa) `args[0]`, `args[1]`, `args[2]`
  - ma sono stringhe, non numeri!
- L'algoritmo è ben noto:
  - per estrarre la radice: `Math.sqrt`
  - ovviamente, vanno distinti i vari casi: radici reali, immaginarie, etc.
- **Problemino: come stampare le soluzioni immaginarie?**
  - in matematica di solito si scrive «  $3 \pm 2i$  »
  - servirebbe il carattere '±' : esiste?

**Sì.. 😊 MA la sua codifica dipende dal set di caratteri in uso 😞**

Nel prompt comandi:      codice 00B1 (in decimale, 177)    '`\u00B1`'

Nelle finestre Windows:   codice 00F1 (in decimale, 241)   '`\u00F1`'



# ESERCIZIO: EQUAZIONI DI 2° GRADO

- Cosa vogliamo? Partiamo dall'output atteso:

```
E>java Equation 1 5 6
Soluzioni reali: -2.5±0.5

E>java Equation 1 -5 6
Soluzioni reali: 2.5±0.5

E>java Equation 1 0 1
Soluzioni immaginarie: -0.0±1.0i

E>java Equation 1 0 -1
Soluzioni reali: -0.0±1.0

E>java Equation 1 -2 1
Soluzioni coincidenti: 1.0
```

**PROVATE VOI A SCRIVERE E  
COLLAUDARE LA SOLUZIONE !**

Solo dopo, girate pagina...



# ESERCIZIO: EQUAZIONI DI 2° GRADO

- Il collaudo
  - una serie di affermazioni "mi aspetto che...."
  - studiate per coprire tutti i casi (soluzioni reali distinte, soluzioni immaginarie, soluzioni coincidenti – occhio agli errori numerici...)
- Cosa ci aspettiamo?
  - che con argomenti numerici 1, 5, 6 le soluzioni siano...
  - che con argomenti numerici 1, -5, 6 le soluzioni siano...
  - che con argomenti numerici 1, 0, 1 le soluzioni siano...
  - che con argomenti numerici 1, 0, -1 le soluzioni siano...
  - che con argomenti numerici 1, 0, 0 le soluzioni siano...
  - che con argomenti numerici 0, 0, 1 le soluzioni siano... Impossibile
  - che con argomenti numerici 0, 1, -2 le soluzioni siano...  $x = 2$  (1° grado)
  - .. e se gli argomenti NON sono numerici (es. 1, 0, aa) ?



# ESERCIZIO: EQUAZIONI DI 2° GRADO

---

**Sicuro/a di voler vedere la soluzione...?**

Ti farebbe MOLTO meglio svilupparla e  
collaudarla da solo/a...





# ESERCIZIO: EQUAZIONI DI 2° GRADO

- Soluzione imperfetta: se  $a=0$ , esplode!

```
public class Equation {
    public static void main(String[] args){
        if (args.length!=3) {
            System.out.println("Errore: fornire i tre coefficienti a, b, c separati da spazi");
        }
        else {
            double a = Double.parseDouble(args[0]);
            double b = Double.parseDouble(args[1]);
            double c = Double.parseDouble(args[2]);
            double delta = b*b-4*a*c;
            if (delta<0) {
                double q = Math.sqrt(-delta)/(2*a);
                double p = -b/(2*a);
                System.out.println("Soluzioni immaginarie: " + p + '\u00B1' + q + "i");
            }
            else if (delta>0) {
                double q = Math.sqrt(delta)/(2*a);
                double p = -b/(2*a);
                System.out.println("Soluzioni reali: " + p + '\u00B1' + q);
            }
            else /* delta==0 */ {
                double p = -b/(2*a);
                System.out.println("Soluzioni coincidenti: " + p);
            }
        }
    }
}
```

Versione per prompt comandi  
(per le finestre grafiche Windows  
servirebbe il carattere 00F1)

**LA SOLUZIONE CORRETTA?**  
*Buon lavoro... 😊😊*



# ESERCIZIO: EQUAZIONI DI 2° GRADO

- Soluzione corretta (con finezze di stampa)

```
public class Equation {
    public static void main(String[] args){
        if (args.length!=3) {
            System.out.println("Errore: fornire i tre coefficienti a, b, c separati da spazi");
        }
        else {
            double a = Double.parseDouble(args[0]);
            double b = Double.parseDouble(args[1]);
            double c = Double.parseDouble(args[2]);
            if (a==0) {
                System.out.println("Equazione di 1 grado: " + b + "x" + (c>=0?"+"+"") + c + " = 0");
                System.out.println("Soluzione: " + (-c/b));
            }
            else {
                System.out.println("Equazione di 2 grado: " + a + "x^2" + (b>=0?"+"+"") + b + "x" + (c>=0?"+"+"") + c + " = 0");
                double delta = b*b-4*a*c;
                if (delta<0) {
                    double q = Math.sqrt(-delta)/(2*a);
                    double p = -b/(2*a);
                    // System.out.println("Soluzioni immaginarie: " + p + '\u00B1' + q + "i"); // per prompt comandi (codice 177)
                    System.out.println("Soluzioni immaginarie: " + p + '\u00F1' + q + "i"); // per finestre grafiche windows (codice 241)
                }
                else if (delta>0) {
                    double q = Math.sqrt(delta)/(2*a);
                    double p = -b/(2*a);
                    // System.out.println("Soluzioni reali: " + p + '\u00B1' + q); // per prompt comandi (codice 177)
                    System.out.println("Soluzioni reali: " + p + '\u00F1' + q); // per finestre grafiche windows (codice 241)
                }
                else /* delta==0 */ {
                    double p = -b/(2*a);
                    System.out.println("Soluzioni coincidenti: " + p);
                }
            }
        }
    }
}
```