

Java SE: basi

- Programmazione procedurale in Java
 - Struttura del codice
 - Tipi di dato: primitivi e reference, array
 - Operatori, istruzioni condizionali e di loop
 - Metodi
- Le classi String, StringBuilder, Math
- Test del codice con JUnit
- Logging con JUL (java.util.Logger)
- Progetto di riferimento
 - <https://github.com/egalli64/jse> (*modulo 1*)

Struttura del codice /1

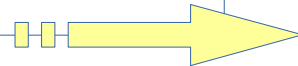
- Dichiarazioni

- **package**
 - Gruppo (omogeneo) a cui appartiene la classe
- **import**
 - Indica l'uso di classi definite in altri package
 - Eccezione, java.lang non richiede di essere importato
- **class**
 - Una sola "public" per file sorgente

- Commenti

- Multi-line
- Single-line
- Javadoc-style

```
/*  
 * A simple Java source file  
 */  
package m1.s02;  
  
import java.lang.Math; // not required  
  
/**  
 * @author manny  
 */  
public class Simple {  
    public static void main(String[] args) {  
        System.out.println(Math.PI);  
    }  
}  
  
class PackageClass {  
    // TBD  
}
```



Struttura del codice /2

- Parentesi

- Graffe

- Blocchi, body di classi e metodi

- Tonde

- Qui identificano metodi
 - Per la definizione – `main()` – lista dei parametri
 - Per l'invocazione – `println()` – lista degli argomenti

- Quadre

- Identificano array

- Punto e virgola

- Obbligatorio per indicare il termine di uno statement

```
/*
 * A simple Java source file
 */
package m1.s02;

import java.lang.Math; // not required



/**
 * @author manny
 */
public class Simple {
    public static void main(String[] args) {
        System.out.println(Math.PI);
    }
}

class PackageClass {
    // TBD
}
```

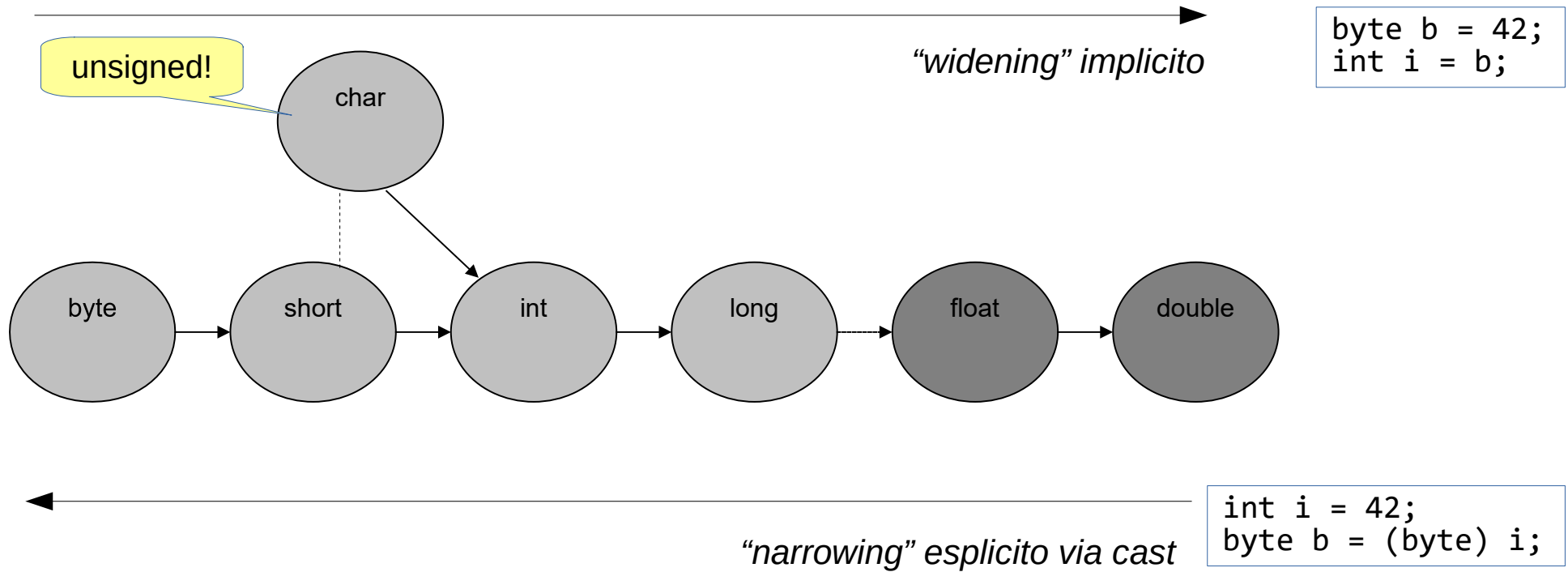
Variabili e tipi di dato

- Variabile
 - Locazione di memoria
 - Nome usato per accedere il dato nella memoria
 - Case sensitive
 - Non tutti i caratteri sono utilizzabili per un identificatore
- Tipo di dato
 - Determina il valore della variabile e le operazioni disponibili su di essa
 - In Java ci sono due famiglie di tipi di dato
 - Primitivi
 - Reference (class / interface)
 - Da **Java 10**, lo si può lasciare dedurre dal compilatore → **var**

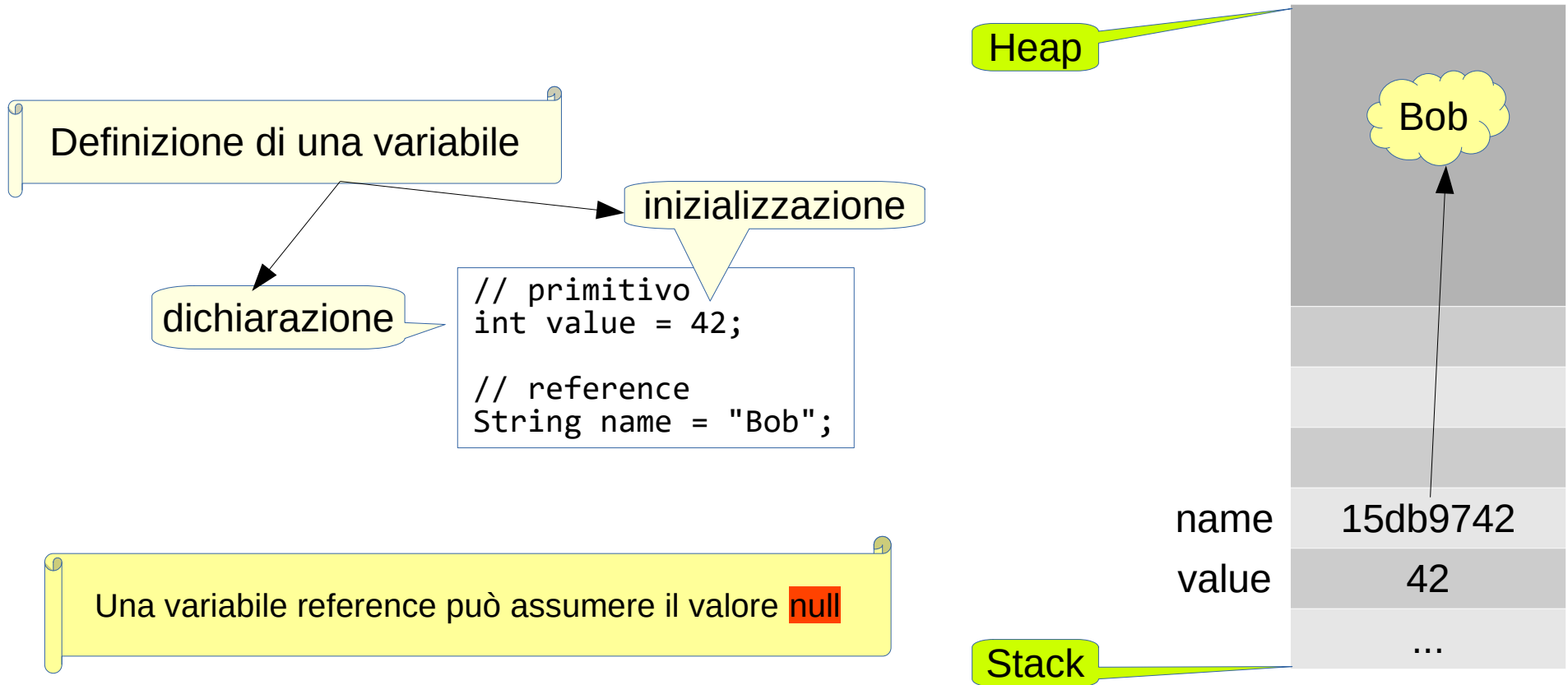
Tipi primitivi

bit			signed integer		floating point IEEE 754	
1(?)	boolean	false				
		true				
8			byte	-128		
				127		
16	char	'\u0000'	short	-32,768		
		'\uFFFF'		32,767		
32			int	-2^31	float	
64			long 	-2^63	double	

Cast tra primitivi



Primitivi vs Reference



Array

- Sequenza indicizzata – base 0 – di valori tutti dello stesso tipo (primitivo o reference), memorizzati nello **heap**.
- La sua dimensione è definita al momento della creazione, e non può più essere cambiata
- È un reference, ma non implementa metodi suoi, ha solo la proprietà (readonly) **length**
- Tentativo di accedere a un elemento esterno → **ArrayIndexOutOfBoundsException**
- Metodi di utilità nella classe **Arrays**: copyOf(), sort(), fill(), equals(), toString(), deepToString(), ...

```
int[] array = new int[12];  
array[0] = 7;  
  
int value = array[5];  
// value = array[12]; // exception
```

```
int[] array = { 1, 4, 3 };  
  
// array[array.length] = 21; // exception  
  
System.out.println(array.length); // 3
```

```
int[][] array2d = new int[4][5];  
  
int value = array2d[2][3];
```

[0][0]	[0][1]	[0][2]	[0][3]	[0][4]
[1][0]	[1][1]	[1][2]	[1][3]	[1][4]
[2][0]	[2][1]	[2][2]	[2][3]	[2][4]
[3][0]	[3][1]	[3][2]	[3][3]	[3][4]

String

- Un singolo carattere è rappresentato dal primitivo **char**
- **String** è una classe, un reference, le sue istanze sono oggetti nello heap
 - rappresenta una sequenza immutabile di caratteri
- **StringBuilder**, mutabile, è usata per creare stringhe complesse
 - Come indica il nome, implementa il pattern Builder

```
char c = 'x';  
String s = new String("hello");  
String t = "hello";
```

Forma standard (ma crea due oggetti!)

Forma semplificata (preferita!)

Operatori unari

++ incremento

-- decremento

prefisso: “naturale”

postfisso: ritorna il valore
prima dell'operazione

+ mantiene il segno corrente

- cambia il segno corrente

```
int value = 1;

System.out.println(value);      // 1

System.out.println(++value);   // 2
System.out.println(--value);    // 1

System.out.println(value++);   // 1
System.out.println(value);     // 2
System.out.println(value--);   // 2
System.out.println(value);     // 1

System.out.println(+value);    // 1
System.out.println(-value);    // -1
```

Operatori aritmetici e assegnamento

+	Addizione
-	Sottrazione
*	Moltiplicazione
/	Divisione (intera?)
%	Modulo

=	Assegnamento
+=	Aggiungi e assegna
-=	Sottrai e assegna
*=	...
/=	
%=	
...	

```
int a = 10;
int b = 3;
double c = 3.0;

a / b    // 3
a % b    // 1
a / 0    // ArithmeticException

a % c    // 1.0
c - 2.1  // 0.8999999999999999
c / 0    // Infinity
```

```
int alpha = 2;

alpha += 8;    // 10
alpha -= 3;    // 7
alpha *= 2;    // 14
alpha /= 2;    // 7
alpha %= 5;    // 2
```

Concatenazione di stringhe

- Operatori utilizzabili su stringhe (overload)
 - +** Se almeno un operando è una stringa, genera una nuova stringa che è la concatenazione dei due operandi
 - +=** Se la variabile a sinistra è una stringa, l'espressione a destra viene valutata come stringa. Il risultato della concatenazione viene assegnato alla variabile a sinistra.
- Da **Java 11**, **repeat()** è una specie di moltiplicazione per stringhe

```
System.out.println("Resistance" + " is " + "useless");  
System.out.println("Solution: " + 42);  
System.out.println(true + " or " + false);  
  
System.out.println("Vogons".repeat(3));
```

Operatori relazionali

<	Minore
<=	Minore o uguale
>	Maggiore
>=	Maggiore o uguale
==	Uguale
!=	Diverso

Reference vs primitivi!

```
int alpha = 12;
int beta = 21;
int gamma = 12;

System.out.println("alpha < beta? " + (alpha < beta));      // true
System.out.println("alpha < gamma? " + (alpha < gamma));    // false
System.out.println("alpha <= gamma? " + (alpha <= gamma));  // true

System.out.println("alpha > beta? " + (alpha > beta));      // false
System.out.println("alpha > gamma? " + (alpha > gamma));    // false
System.out.println("alpha >= gamma? " + (alpha >= gamma));  // true

System.out.println("alpha == beta? " + (alpha == beta));    // false
System.out.println("alpha == gamma? " + (alpha == gamma));  // true

System.out.println("alpha != beta? " + (alpha != beta));    // true
System.out.println("alpha != gamma? " + (alpha != gamma));  // false
```

Operatori logici (e bitwise)

“shortcut”
preferiti

&&	AND
	OR
!	NOT
&	AND
	OR
^	XOR
~	NOT

```
boolean alpha = true;  
boolean beta = false;
```

```
System.out.println(alpha && beta); // false  
System.out.println(alpha || beta); // true  
System.out.println(!alpha);       // false  
System.out.println(alpha & beta);  // false  
System.out.println(alpha | beta);  // true
```

```
int gamma = 0b101; // 5  
int delta = 0b110; // 6
```

0b → rappresentazione binaria (Java 7)

```
System.out.println(gamma & delta); // 4 == 0100  
System.out.println(gamma | delta); // 7 == 0111  
System.out.println(gamma ^ delta); // 3 == 0011  
System.out.println(~gamma);        // -6
```

Condizioni

- Se la condizione è vera, si esegue il blocco associato.
- Altrimenti, se presente si esegue il blocco “else”.

```
if (condition) {  
    // doSomething  
}  
  
// nextStep
```

```
if (condition) {  
    // doSomething  
} else {  
    // doSomethingElse  
}  
  
// nextStep
```

```
if (condition) {  
    // doSomething  
} else if (otherCondition) {  
    // doSomethingElse  
} else {  
    // doSomethingDifferent  
}  
  
// nextStep
```

switch

Scelta multipla su byte, short, char, int, String, **enum** ➡

```
int value = 42;
// ...

switch (value) {
case 1:
    // ...
    break;
case 2:
    // ...
    break;
default:
    // ...
    break;
}
```

```
String value = "1";
// ...

switch (value) {
case "1":
    // ...
    break;
case "2":
    // ...
    break;
default:
    // ...
    break;
}
```

```
public enum WeekendDay {
    SATURDAY, SUNDAY
}

WeekendDay day = WeekendDay.SATURDAY;
// ...

switch (day) {
case SATURDAY:
    // ...
    break;
case SUNDAY:
    // ...
    break;
}
```


Costanti simboliche con enum

- **enum** è un tipo speciale di classe i cui elementi sono un insieme di costanti
 - Nell'uso base, si fa riferimento solo al nomi delle costanti
 - Che comunque hanno un valore associato (default: 0, 1, ...)
 - Comodo per limitare la scelta in un range noto e limitato
 - Funziona bene in abbinamento allo switch

```
public enum WeekDay { MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY, SATURDAY, SUNDAY }
```

loop

```
while (condition) {  
    // ...  
  
    if (something) {  
        condition = false;  
    }  
}
```

```
do {  
    // ...  
  
    if (something) {  
        condition = false;  
    }  
} while (condition);
```

```
for (int i = 0; i < 5; i++) {  
    // ...  
  
    if (i == 2) {  
        continue;  
    }  
  
    // ...  
}
```

```
String[] array = new String[5];  
// ...  
for (String item : array) {  
    System.out.println(item);  
}
```

for each

```
for (;;) {  
    // ...  
  
    if (something) {  
        break;  
    }  
  
    // ...  
}
```

forever

Metodo

- In Java non esistono funzioni libere
- Un metodo è una funzione definita all'interno di una classe:
 - return type
 - primitivo, reference, o void
 - nome
 - lista dei parametri
 - [lista eccezioni che può tirare]
- Un metodo è associato a
 - una istanza della classe (default)
 - o all'intera classe (static)
- È una piccola macchina di Turing
 - Input: parametri
 - Output: valore ritornato al chiamante



```
public class Simple {  
    static String h() {  
        return "Hi";  
    }  
  
    int f(int a, int b) {  
        return a * b;  
    }  
  
    void g(boolean flag) {  
        if (flag) {  
            System.out.println("Hello");  
            return;  
        }  
        System.out.println("Goodbye");  
    }  
}
```

Parametri

- In Java i valori sono passati ai metodi “by value”
- Primitivi:
 - Il parametro è una copia del valore passato. La sua eventuale modifica non è osservabile dal chiamante
- Reference
 - Il parametro è una copia del reference passato. L' oggetto referenziato è lo stesso e dunque la sua eventuale modifica è osservabile dal chiamante
 - Nota che:
 - immutabili, come String, per definizione non possono essere modificati
 - ogni reference può essere null, va controllata prima dell'uso: Objects.requireNonNull()
 - Il metodo main ha per parametro un array di stringhe “args”, gli argomenti passati al programma

Alcuni metodi di String

- char **charAt**(int)
 - Carattere nella posizione indicata
 - int **compareTo**(String)
 - String **concat**(String)
 - boolean **contains**(CharSequence)
 - boolean **equals**(Object)
 - Confronto tra stringhe, “==” è sul reference!
 - int **indexOf**(int) // carattere!
 - int **indexOf**(String)
 - boolean **isEmpty**()
 - int **lastIndexOf**(int)
 - int **length**()
 - String **replace**(char, char) // replace all
 - String[] **split**(String)
 - String **substring**(int), String **substring**(int, int)
 - String **toLowerCase**()
 - String **toUpperCase**()
 - String **trim**()
- Tra i metodi statici:***
- String **format**(String, Object...)
 - String **join**(CharSequence, CharSequence...)
 - String **valueOf**(Object)

Alcuni metodi di StringBuilder

- `StringBuilder(int)`
- `StringBuilder(String)`
- `StringBuilder append(Object)`
- `char charAt(int)`
- `StringBuilder delete(int, int)`
- `void ensureCapacity(int)`
- `int indexOf(String)`
- `StringBuilder insert(int, Object)`
- `int length()`
- `StringBuilder replace(int, int, String)`
- `StringBuilder reverse()`
- `void setCharAt(int, char)`
- `void setLength(int)`
- `String toString()`

La classe Math

Costanti

- E – base del logaritmo naturale
- PI – pi greco

Alcuni metodi statici

- double abs(double) // int, ...
- int addExact(int, int) // multiply ...
- double ceil(double)
- double cos(double) // sin(), tan()
- double exp(double)
- double floor(double)
- double log(double)

... altri metodi statici

- double max(double, double) // int, ...
- double min(double, double) // int, ...
- double pow(double, double)
- double random()
- long round(double)
- double sqrt(double)
- double toDegrees(double) // approx
- double toRadians(double) // approx

Unit Test

- Verifica (nel folder test) la correttezza di una “unità” di codice, permettendone il rilascio da parte del team di sviluppo con maggior confidenza
- Un unit test, tra l'altro:
 - dimostra che una nuova feature ha il comportamento atteso
 - documenta un cambiamento di funzionalità e verifica che non causi malfunzionamenti in altre parti del codice
 - mostra come funziona il codice corrente
 - tiene sotto controllo il comportamento delle dipendenze

JUnit in Eclipse

- Right click sulla classe (Simple) da testare
 - New, JUnit Test Case
 - JUnit 4 o 5 (Jupiter)
 - Source folder dovrebbe essere specifica per i test
 - *Se richiesto*, add JUnit library to the build path
- Il wizard crea una nuova classe (SimpleTest)
 - I metodi che JUnit esegue sono quelli annotati @Test
 - Il metodo statico Assertions.fail() indica il fallimento di un test
- Per eseguire un test case: Run as, JUnit Test

Struttura di un test JUnit

- Ogni metodo di test dovrebbe
 - avere un nome significativo
 - essere strutturato in tre fasi
 - Preparazione
 - Esecuzione
 - Assert

```
public int negate(int value) {  
    return -value;  
}
```

Simple.java

SimpleTest.java

```
@Test  
public void negatePositive() {  
    Simple simple = new Simple();  
    int value = 42;  
    int expected = -42;  
  
    int result = simple.negate(value);  
  
    assertThat(result, equalTo(expected));  
}
```

@BeforeEach

- I metodi annotati @BeforeEach (Jupiter) o @Before (4) sono usati per la parte comune di inizializzazione dei test
- Ogni @Test è eseguito su una nuova istanza della classe, per assicurare l'indipendenza di ogni test
- Di conseguenza, ogni @Test causa l'esecuzione dei metodi @BeforeEach (o @Before)

```
private Simple simple;

@BeforeEach
public void init() {
    simple = new Simple();
}

@Test
public void negatePositive() {
    int value = 42;

    int result = simple.negate(value);

    assertThat(result, equalTo(-42));
}
```

JUnit assert

- Sono metodi statici definiti in `org.junit.jupiter.api.Assertions` (Jupiter) o `org.junit.Assert` (4)

- `assertTrue(condition)`
- `assertNull(reference)`
- `assertEquals(expected, actual)`
- `assertEquals(expected, actual, delta)`

```
assertEquals(.87, .29 * 3, .0001);
```

- `assert` Hamcrest-style, usano

- `org.hamcrest.MatcherAssert.assertThat()` e `matcher` (`org.hamcrest.CoreMatchers`, `org.hamcrest.Matchers`)

`assertThat(T, Matcher<? super T>)` n.b: convenzione opposta ai metodi classici: `actual` – `expected`

- `assertThat(condition, is(true))`
- `assertThat(actual, is(expected))`
- `assertThat(reference, nullValue())`
- `assertThat(actual, closeTo(expected, error))`
- `assertThat(actual, startsWith("Tom"))`
- `assertThat(name, not(startsWith("Bob")))`

Logging

- Necessità di tener traccia delle attività di una applicazione
- Comunicazione tra chi sviluppa il codice
 - Debugging, analisi dei flussi di esecuzione, ...
- L'uso di `System.out.println()` non è una soluzione accettabile
- Alcune tra le principali librerie utilizzate in Java
 - Java Logging / JUL (`java.util.logging`)
 - SLF4J (+ Logback o altri)
 - Apache Log4J

JUL

- Configurazione di default nella JRE/JDK in
 - Folder conf (o lib): `logging.properties`
- Ogni programma può avere la sua configurazione
 - `src/main/resource`
- Livelli di log tra OFF e ALL
 - `FINEST`, `FINER`, `FINE`, `CONFIG`, `INFO`, `WARNING`, `SEVERE`
- Formatter: prepara la stringa che verrà stampata
- Handler: gestisce la richiesta via console / file
- Per programmi molto semplici: global logger