

Alma Mater Studiorum-Università di Bologna

Scuola di Ingegneria

OOP: linguaggi e piattaforme

Corso di Laurea in Ingegneria Informatica

Anno accademico 2025/2026

Prof. ENRICO DENTI

Dipartimento di Informatica – Scienza e Ingegneria (DISI)



IL PUNTO DI VISTA PROCEDURALE

I linguaggi "classici", come il C, adottano il **punto di vista procedurale**

- l'enfasi è sull'**operazione da svolgere** (primo argomento)
- «chi» la svolge è in secondo piano (se c'è...)

Infatti, per svolgere l'operazione *operation* sul componente *comp* con certi *argomenti*, si scrive tipicamente:

operation(*comp*, *argomenti*)

COSA fare

CHI la fa

dati necessari

Esempio: **fprintf(fout, "Hello!");**



IL PUNTO DI VISTA PROCEDURALE

L'approccio *procedurale*:

- è naturale in un mondo semplice, dove c'è un solo ("ovvio") destinatario delle operazioni
 - architettura monolitica: "chi" svolge le operazioni è scontato
 - focus sull'algoritmo, ossia la sequenza *di azioni* da svolgere
- mostra tutti i suoi limiti in presenza di *sistemi software*
 - architettura *multi-componente*: molte entità interagiscono fra loro
 - interazione più che / accanto a algoritmica: **il focus non è più solo sulle operazioni da svolgere, ma su CHI faccia COSA**
 - focus su **come distribuire le responsabilità** fra i componenti
 - conseguente necessità di **evidenziare A CHI CI SI RIVOLGE** per richiedere una certa operazione / un certo SERVIZIO



L'INVERSIONE DEL PUNTO DI VISTA

Nell'approccio a oggetti, *l'enfasi è sull'oggetto a cui ci si rivolge*

- una entità dotata di una propria identità
- con le sue proprietà
- in grado di svolgere certi servizi (operazioni)

Questo porta a *invertire il punto di vista*:

- enfasi non più sull'operazione svolta (da qualcuno)
- ma sull'entità che la svolge → *l'oggetto*

Vi stupisce? NON DOVREBBE, visto che *lo fate continuamente!*

- ogni volta che fate "doppio clic" o "tap" su un'icona *vi concentrate sul "chi" deve fare qualcosa, non sull'operazione: "mandate un messaggio" all'entità e le chiedete di fare qualcosa (APRIRSI / ESEGUIRSI)*
- non aprite il programma/app, per poi scegliere l'operazione da menù!



IL PUNTO DI VISTA «A OGGETTI»

Nell'approccio a oggetti:

- si mette in primo piano **chi** svolge l'operazione
- l'operazione da svolgere passa *in secondo piano*

Per esprimere questo cambiamento di prospettiva:

- si adotta una *notazione sintattica che enfatizzi il cambiamento*
- si riutilizza a tal fine la *notazione puntata* già in uso per le **struct**, attribuendole però *nuovo significato*

Per chiedere al componente **comp** di svolgere **operation**, si scrive:

comp. **operation** (*argomenti*)

a CHI mi
rivolgo

COSA gli
chiedo di fare

dati necessari
per farla



NOTAZIONE PUNTATA

- In questi linguaggi, la notazione puntata indica anche la *selezione di un'operazione* fra quelle offerte da un'entità
- Si usa parlare di *metodo* per richiedere un *servizio*

Ad esempio, in Java la frase:

Java

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

richiede al componente `System.out` di svolgere il servizio `println`
(`out` è a sua volta un componente dell'entità `System`)

Analogamente, in C# la frase:

C#

`System.Console.WriteLine("Hello!");`

richiede al componente `System.Console` di svolgere il servizio
`WriteLine` (`Console` è a sua volta un componente nell'entità `System`)



JAVA 25: NOVITÀ

- In Java 25 è stato aggiunto il **nuovo componente IO**, che consente di scrivere la stessa cosa più compattamente

In Java 25, la frase precedente si può riscrivere come:

```
IO.println("Hello!");
```

Java 25

richiede al componente **IO** di svolgere il servizio **println**

- MA: Java 25 è uscito da pochissimo (ottobre 2025) e al mondo ci sono **milionи di applicazioni Java** scritte nel modo standard precedente → **System.out** non va dimenticato: è e sarà ancora con noi per *moltissimi anni!*
 - in questo corso faremo uso di entrambi

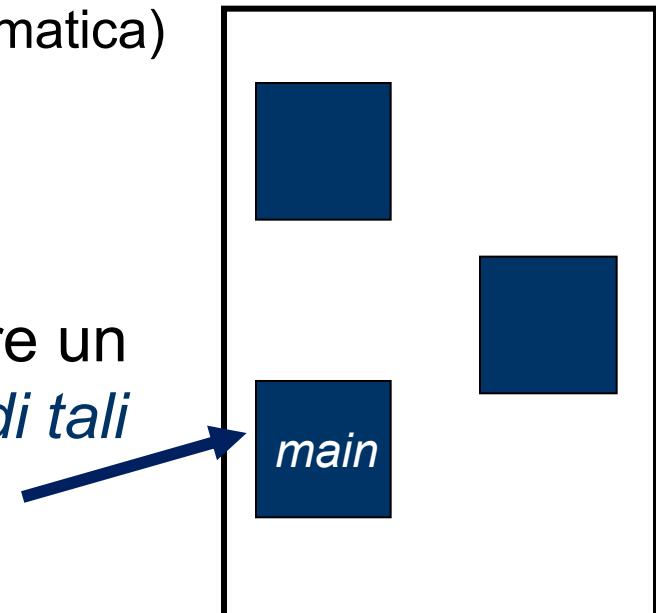


APPLICAZIONI A OGGETTI: ARCHITETTURA

Una applicazione è strutturata come *un insieme di entità*:

- alcune sono statiche, ossia esistono *prima* dell'inizio del programma e permangono *per tutta la sua durata*
 - librerie (prive di stato, es. libreria matematica)
 - moduli software statici (oggetti singoli)
 - definizioni di tipi

Poiché ogni applicazione deve avere un punto di partenza prestabilito, *una di tali entità statiche contiene il main*

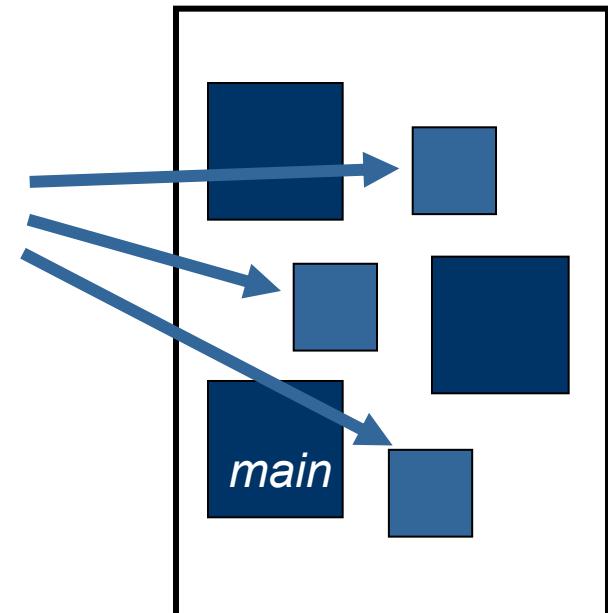




APPLICAZIONI A OGGETTI: ARCHITETTURA

Una applicazione è strutturata come *un insieme di entità*:

- alcune sono *statiche*, ossia esistono *prima* dell'inizio del programma e permangono *per tutta la sua durata*
- altre invece sono *dinamiche*, ossia vengono *create durante l'esecuzione* solo al momento del bisogno





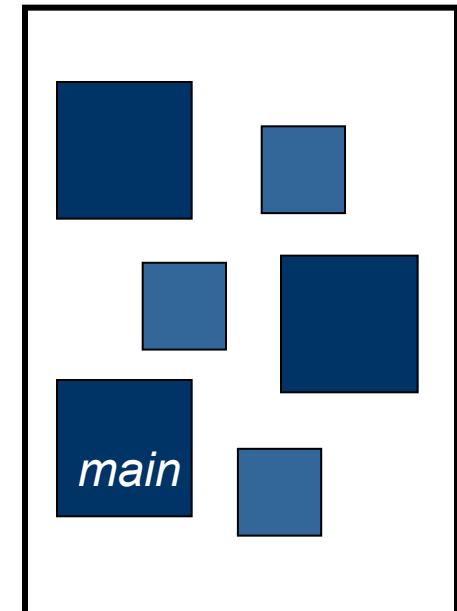
APPLICAZIONI A OGGETTI: ARCHITETTURA

Una applicazione è strutturata come *un insieme di entità*:

- alcune sono *statiche*, ossia definite prima dell'inizio del programma e visibili per tutta la sua durata
- altre sono *dinamiche*, ossia create durante l'esecuzione al momento del bisogno

CLASSI
o oggetti singleton

OGGETTI
dinamici





REQUISITI PER NUOVI LINGUAGGI

- Java, C#, Scala, Kotlin sono linguaggi *progettati ex novo*, facendo tesoro delle esperienze (e degli errori) precedenti
- Ispirati «di base» a C e C++, ma *senza il requisito della piena compatibilità all'indietro*
- Obiettivo 1: sostituire meccanismi e costrutti linguistici poco chiari, sintatticamente obsoleti ed error-prone con *nuovi meccanismi e costrutti evoluti*
 - intrinsecamente più sicuri, chiari, di più alto livello
 - pensati per *gestire la complessità* ed evitare la gestione diretta (error-prone) di tanti/troppi dettagli
- Obiettivo 2: intercettare a compile-time quanti più errori possibile: *un errore non scoperto prima esploderà dopo...*



REQUISITI PER NUOVI LINGUAGGI

- Alcune idee
 - sostituzione dei puntatori con *riferimenti*
 - dereferenziamento automatico
 - allocazione e deallocazione automatica della memoria
 - **type safety** =
type system stringente + *type inference* + controlli a run-time
- E inoltre, più recentemente (Scala & Kotlin soprattutto):
 - **null safety** = disincentivo e controllo nell'uso di *puntatori null*
 - **type inference** ancora più evoluta
 - **distinzione valori / variabili**
 - preferenza a **strutture immutabili**, che garantiscono safety
 - funzioni come **first-class entities** & **lambda expressions**
 - stile più funzionale

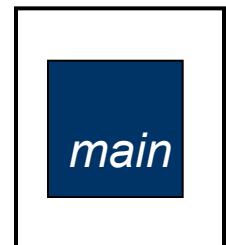


L'APPLICAZIONE PIÙ SEMPLICE: Java & C#

La più semplice applicazione possibile è costituita da **un singolo componente (*singleton*)**, che definisce **soltanto il main**

PRIMA DIFFERENZA RISPETTO AL C:

- in C, il main è *semplicemente scritto in un file, non è racchiuso in alcun costrutto linguistico*
- qui invece *il main dev'essere posto in una classe pubblica ed essere esso stesso pubblico (criteri di protezione)*



```
public class MyProg {  
    ...  
    // il main (anch'esso pubblico)  
    ...  
}
```

Java

C#

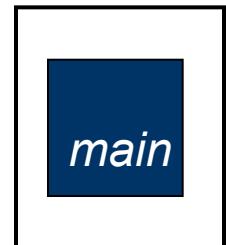


L'APPLICAZIONE PIÙ SEMPLICE: Scala & Kotlin

Scala e Kotlin sono analoghi, con leggere differenze:

(segue)

- in Scala la qualifica *public* è predefinita → non va specificata; inoltre, i componenti singleton si chiamano *object*
- in Kotlin, il componente *object* che contiene il main è *implicito* → non occorre specificarlo



```
object MyProg {  
    ...  
    // il main Scala  
    ...  
}
```

Scala

```
// object implicit  
    ...  
    // il main Kotlin  
    ...
```

Kotlin

NB: Scala 3 ha introdotto una forma più compatta, che vedremo dopo.



IL MAIN

Nei nuovi linguaggi, il main evolve rispetto al C:

SECONDA DIFFERENZA RISPETTO AL C:

- in C, il *main* può avere o non avere argomenti, `argc` / `argv`
- nei nuovi linguaggi il *main* ha sempre come *unico argomento* un singolo oggetto: un *array di stringhe*

TERZA DIFFERENZA RISPETTO AL C:

- in C, il *main* può avere tipo di ritorno `void` o `int`
- in Java, il *main* ha sempre tipo di ritorno `void` (NON `int`)
- in C#, il *main* può avere tipo di ritorno `void` o `int`
- in Scala e Kotlin, il *main* ha sempre tipo di ritorno `Unit` (~ `void`)

QUARTA DIFFERENZA RISPETTO AL C:

- In Scala e Kotlin, le funzioni si definiscono in modo diverso rispetto al C



IL MAIN: Java & C#

```
public class MyProg {  
    public static void main(String[] args) {  
        ...  
    }  
}
```

Java: solo **void**

Java

```
public class MyProg {  
    public static void Main(string[] args) {  
        ...  
    }  
}
```

C#: **void o int**

Differenze maiuscole/minuscole

C#

Nuova posizione per le parentesi quadre []

C#: argomento **args** opzionale



IL MAIN: Java, Scala, Kotlin

```
public class MyProg {  
    public static void main(String[] args) {  
        ...  
    }  
}
```

Java

```
object MyProg {  
    def main(args: Array[String]):Unit = {  
    }  
}
```

Nuova keyword **Array** anziché []

Nuova keyword

Sintassi di tipo postfissa

:Unit se omesso,
può essere *inferito*

= obbligatorio

Scala

```
// object implicit  
fun main(args: Array<String>):Unit {  
}
```

Nuova keyword **Array** anziché []

Nuova keyword

Sintassi di tipo postfissa

:Unit se omesso,
può essere *inferito*

Kotlin



IL CASO PIÙ SEMPLICE: C, Java, C#

```
// file MyProg.c
int main(int argc, char* argv[]) {
    int x=3, y=4; int z = x+y;
}
```

C

```
public class MyProg {
    public static void main(String[] args) {
        int x=3, y=4; int z = x+y;
    }
}
```

Java

```
public class MyProg {
    public static void Main(string[] args) {
        int x=3, y=4; int z = x+y;
    }
}
```

C#

Java e C# ammettono l'assegnamento
di più variabili in un'unica istruzione

Java e C# ammettono l'assegnamento
di più variabili in un'unica istruzione



IL CASO PIÙ SEMPLICE: Java, Scala, Kotlin

```
public class MyProg {  
    public static void main(String[] args) {  
        int x=3, y=4; int z = x+y;  
    }  
}
```

Java

```
object MyProg {  
    def main(args: Array[String]):Unit = {  
        var x:Int = 3; var y:Int = 4; var z:Int = x+y;  
    }  
}
```

Scala

In Scala e Kotlin, le variabili sono introdotte dalla parola chiave **var** (o **val** se immodificabili) e la specifica di tipo è postfissa

Inoltre, non è ammesso dichiarare più variabili nella stessa istruzione: ogni variabile va definita separatamente, con il suo **var** (o **val**)

```
fun main(args: Array<String>):Unit {  
    var x:Int = 3; var y:Int = 4; var z:Int = x+y;  
}
```

Kotlin



CONVENZIONI DI NAMING

Si applicano le seguenti convenzioni per classi e file:

- una classe deve avere un nome *chiaro ed espressivo*, che *inizi per maiuscola* e segua la *convenzione CamelCase*
- ogni file dovrebbe *contenere una sola classe* e il *nome del file* dovrebbe *coincidere con nome della classe*

Per quanto riguarda il codice:

- funzioni e variabili devono avere anch'esse un nome *chiaro ed espressivo*, che *inizi per minuscola* (tranne C#) e segua poi la *convenzione CamelCase*
- le costanti devono avere un nome *tutto maiuscolo*

Non è gradito l'uso dell'underscore (_).



CONVENZIONI DI NAMING

Più precisamente, per quanto riguarda le classi:

- in generale, un file dovrebbe *contenere una sola classe* e il **nome del file** dovrebbe *coincidere con nome della classe*
- in **Java**, questa regola è rigida: ogni file può contenere una sola *classe pubblica* (può poi contenerne altre *non pubbliche*)
- in **C#** **Scala** **Kotlin** la regola è meno rigida: è raccomandato che *il file si chiami come la classe*, ma è comunque possibile raggruppare più classi (anche pubbliche) nello stesso file
 - in tal caso, il nome del file tipicamente coincide con quello della classe più importante, o viene attribuito sulla base della funzione logica complessivamente svolta dall'insieme
 - in Scala, per differenziare la situazione multi-class, si raccomanda per il file l'uso di un nome che inizi per minuscola (*camelCase*)



CONVENZIONI DI NAMING

Più precisamente, per quanto riguarda le classi:

- in **Java**, il file deve chiamarsi *esattamente come la classe* e avere estensione **.java**
- in **C#**, il file dovrebbe chiamarsi *come la classe* e deve avere estensione **.cs**
- in **Scala**, il file dovrebbe chiamarsi *esattamente come la classe (o object)* e deve avere estensione **.scala**
- in **Kotlin**, il file dovrebbe chiamarsi *esattamente come la classe (o object)* e deve avere estensione **.kt**

NB: «esattamente come la classe» significa *maiuscole/minuscole comprese*, senza eccezioni.



COMPILAZIONE: C, Java, C#

COMPILAZIONE C

C:> cc MyProg.c

produce MyProg.exe

C

L'EXE ottenuto è eseguibile direttamente sullo specifico sistema operativo

COMPILAZIONE Java

C:> javac MyProg.java

produce MyProg.class

Java

Il file ottenuto è eseguibile sull'infrastruttura Java

NB: dev'essere installato il JDK e dev'essere nel PATH

Non sono la stessa cosa!

COMPILAZIONE C#

C:> csc MyProg.cs

produce MyProg.exe

C#

Il file ottenuto è un EXE eseguibile sull'infrastruttura .NET

NB: dev'essere installato il .NET Framework e dev'essere nel PATH



COMPILAZIONE: Scala, Kotlin

Scala e Kotlin sono costruiti per funzionare *sulla stessa infrastruttura di Java*: la Java Virtual Machine (JVM)

Pertanto:

- la compilazione si lancia coi rispettivi compilatori
 - `javac` per Java
 - `scalac` e `kotlinc` rispettivamente per Scala e Kotlin
- ma **il risultato è comunque costituito da file `.class`**
 - perché Kotlin e Scala su JVM sono basati sulla piattaforma Java, con cui tra l'altro sono *interoperabili*

NB: entrambi questi linguaggi possono compilare anche per altre piattaforme, la JVM non è l'unica opzione



COMPILAZIONE: Scala, Kotlin

COMPILAZIONE Scala

C:> `scalac MyProg.scala` *produce MyProg.class
+ alcuni file accessori*

Scala

Il file ottenuto è eseguibile **sull'infrastruttura Java** (+ librerie Scala)

COMPILAZIONE Kotlin

C:> `kotlinc MyProg.kt` *produce MyProgKt.class*

Kotlin

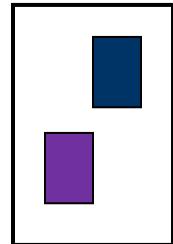
Il file ottenuto è eseguibile **sull'infrastruttura Java** (+ librerie Kotlin)



UN ESEMPIO CON DUE ENTITÀ

Un programma costituito da *due entità*:

- la nostra **Esempio1**, che definisce il main
- una **classe fornita dall'infrastruttura**



Obiettivo:

- stampare a video la classica frase di benvenuto sfruttando il servizio di stampa fornito dal «sistema»

In **Java** e **C#**, la classe **System** rappresenta *il sistema sottostante* con tutti i suoi sotto-componenti e i relativi servizi

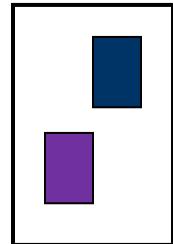
- uno di tali componenti è il *dispositivo standard di uscita* (*chiamato **out** in Java e **Console** in C#*)
- tale componente offre, fra gli altri, il *servizio di stampa* (*chiamato **println** in Java e **WriteLine** in C#*)



UN ESEMPIO CON DUE ENTITÀ

Un programma costituito da *due entità*:

- la nostra **Esempio1**, che definisce il main
- una **classe fornita dall'infrastruttura**



Obiettivo:

- stampare a video la classica frase di benvenuto sfruttando il servizio di stampa fornito dal **«sistema»**

In **Scala** e **Kotlin** la situazione è analoga, ma la suddivisione interna dei servizi è leggermente diversa:

- in Scala il *dispositivo di uscita* si chiama **Console** come in C#, ma il *servizio di stampa* si chiama **println**, come in Java
- in Kotlin il *dispositivo di uscita* è nel componente **kotlin.io** e il *servizio di stampa* si chiama anche lì **println**, come in Java



UN ESEMPIO CON DUE ENTITÀ: C, Java, C#

```
<include stdio.h>
int main(int argc, char* argv[]) {
    printf("%s", argv[1]);
}
```

C

```
public class Esempio1 {
    public static void main(String[] args) {
        System.out.println(args[0]);
    }
}
```

Java

In Java e C#, `args[0]` non è il nome
del programma: è già il primo argomento

```
public class Esempio1 {
    public static void Main(string[] args) {
        System.Console.WriteLine(args[0]);
    }
}
```

C#



UN ESEMPIO CON DUE ENTITÀ: C, Java, C#

```
<include stdio.h>
int main(int argc, char* argv[]) {
    printf("%s", argv[1]);
}
```

C

```
public class Esempio1 {
    public static void main(String[] args) {
        IO.println(args[0]);
    }
}
```

Java 25

In Java 25, `println` è anche un metodo dell'oggetto `IO`.
Si può quindi scrivere più brevemente `IO.println`.

```
public class Esempio1 {
    public static void Main(string[] args) {
        System.Console.WriteLine(args[0]);
    }
}
```

C#



UN ESEMPIO CON DUE ENTITÀ: Java, Scala, Kotlin

```
public class Esempio1 {  
    public static void main(String[] args) {  
        System.out.println(args[0]);  
    }  
}
```

Java

```
object Esempio1 {  
    def main(args: Array[String]):Unit = {  
        Console.println(args(0));  
    }  
}
```

Scala

Scala: parentesi tonde
per accedere agli array

```
public fun main(args: Array<String>):Unit {  
    kotlin.io.println(args[0]);  
}
```

Anche in Scala e
Kotlin, `args[0]`
è il 1° argomento

In Kotlin, `println` è un metodo dell'oggetto `kotlin.io`.
Si può scrivere più brevemente anche solo `println`.

Kotlin



UN ESEMPIO CON DUE ENTITÀ: Java, Scala, Kotlin

Stesso codice in versione più compatta:

```
public class Esempio1 {  
    public static void main(String[] args) {  
        IO.println(args[0]);  
    }  
}
```

Java 25

Si scrive **IO.println**

```
object Esempio1 {  
    def main(args: Array[String]):Unit = {  
        println(args(0));  
    }  
}
```

Scala

Si scrive solo **println**

```
public fun main(args: Array<String>):Unit {  
    println(args[0]);  
}
```

Kotlin

Si scrive solo **println**



COMPILAZIONE: linguaggi a confronto

COMPILAZIONE C

C:> cc Esempio1.c

C

produce Esempio1.exe

COMPILAZIONE Java

C:> javac Esempio1.java

Java

produce Esempio1.class

COMPILAZIONE C#

C:> csc Esempio1.cs

C#

produce Esempio1.exe (.NET)

COMPILAZIONE Scala

C:> scalac Esempio1.scala

Scala

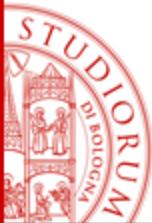
*produce Esempio1.class
ed Esempio1\$.class*

COMPILAZIONE Kotlin

C:> kotlinc Esempio1.kt

Kotlin

produce Esempio1Kt.class



ESECUZIONE SULL'INFRASTRUTTURA

Per eseguire il programma occorre:

- in C, eseguire l'eseguibile autocontenuto prodotto dal compilatore
- negli altri linguaggi, *invocare l'infrastruttura* specificando *la classe che contiene il main*

Per invocare l'infrastruttura:

- in **Java**, si esegue l'interprete (strato-ponte) **java**
- in **.NET**, su Windows si esegue direttamente l'eseguibile; su altre piattaforme, si invoca l'interprete **dotnet**
- in **Scala**, si esegue l'interprete (strato-ponte) **scala**
- in **Kotlin**, si esegue l'interprete (strato-ponte) **kotlin**



ESECUZIONE: linguaggi a confronto

C:> **Esempio1** alfa beta

C

alfa *L'exe è eseguito direttamente sul sistema operativo*

C:> **java Esempio1** alfa beta

Java

alfa *Si invoca l'interprete Java e gli si passa la classe col main*

C:> **Esempio1** alfa beta

C#

alfa *L'exe .NET è eseguito sull'infrastruttura .NET*

NOTA: *sembra* uguale al primo, ma non funziona se sulla macchina non è installato il .NET Framework; con .NET, invocare esplicitamente l'interprete **dotnet**

C:> **scala run -cp . -M Esempio1 --** alfa beta

Scala

alfa *Si invoca l'interprete Scala (Java + librerie) con argomenti*

NOTA: da Scala 3, lo script **scala** richiede alcuni argomenti extra (v. sopra)

C:> **kotlin Esempio1Kt** alfa beta

Kotlin

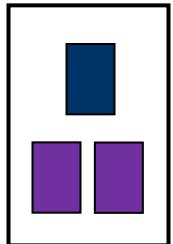
alfa *Si invoca l'interprete Kotlin (Java + librerie) con argomenti*



UN TERZO ESEMPIO (1/4)

Un programma costituito da *tre classi*:

- la nostra **Esempio2**, che definisce il **main**
- due entità di infrastruttura
 - una per stampare, come prima
 - una per fare calcoli: il componente «libreria matematica»
- Chi è e come si chiama la libreria matematica?
 - in **Java** è l'entità (classe): **Math**
 - in **C#** è l'entità (classe): **System.Math**
 - in **Scala** è l'entità:
scala.math
 - in **Kotlin** è l'entità:
kotlin.math
- Cosa offre?
 - Costanti (e , π)
 - Decine di funzioni utili



NB: in C#, Scala e Kotlin, Math (o math) è un sotto-componente di qualcos'altro (System o altra entità «top level»); in Java no.



UN TERZO ESEMPIO (2/4)

```
<include stdio.h>
<include math.h>
int main() {
    printf("%f", sin(M_PI/3) );
}
```

Spazio di nomi unico =
rischio di *clash* + *necessità di differenziare tutti i nomi* ☹

C

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

Java

```
public class Esempio2 {
    public static void Main(string[] args) {
        System.Console.WriteLine(
            System.Math.Sin(System.Math.PI/3) );
    }
}
```

Spazio di nomi *intrinsecamente modulare* = no clash ☺ ☺

C#



UN TERZO ESEMPIO (3/4)

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

Java

```
object Esempio2 {  
    def main(args: Array[String]):Unit = {  
        println(scala.math.sin(scala.math.Pi/3))  
    }  
}
```

Scala

NB: poiché Scala e Kotlin sono interoperabili con Java, è anche possibile scrivere `Math.sin(Math.PI/3)`, usando il componente Java dell'infrastruttura sottostante.

```
// file Esempio2.kt  
fun main(args: Array<String>):Unit {  
    println(kotlin.math.sin(kotlin.math.PI/3))  
}
```

Kotlin



UN TERZO ESEMPIO (4/4)

C:> **Esempio2**

0,866025403784439

Eseguito direttamente sul sistema operativo

C

C:> **java Esempio2**

0,866025403784439

Eseguito sull'infrastruttura Java

Java

C:> **Esempio2**

0,866025403784439

Eseguito sull'infrastruttura .NET

C#

C:> **scala run -cp . -M Esempio2**

0,866025403784439

Eseguito sull'infrastruttura Java + librerie Scala

Scala

C:> **kotlin Esempio2Kt**

0,866025403784439

Eseguito sull'infrastruttura Java + librerie Kotlin

Kotlin



AMBIENTI ONLINE

Per sperimentare senza dover installare niente sul proprio computer (o usando un tablet o smartphone), si possono usare gli *ambienti online*

- disponibili per praticamente ogni linguaggio
- accessibili con un comune browser
- spesso chiamati «playground»
 - multilinguaggio: <https://code.labstack.com/java>
 - **Java** <https://www.studytonight.com/code/playground/java/>
 - **C#** <https://dotnetfiddle.net/srx9kM>
 - **Scala** <https://scastie.scala-lang.org/>
<https://scalafiddle.io/>
<https://www.katacoda.com/courses/scala/playground>
 - **Kotlin** <https://play.kotlinlang.org/>



AMBIENTI ONLINE

The image displays six screenshots of online environments for writing and running code:

- .NET Fiddle**: A C# playground interface with tabs for Options, Language (C#), Project Type (Console), Compiler (.NET 4.7.2), and a preview window showing a simple application that prints "Hello from .NET".
- Kotlin**: An online sandbox for Kotlin. It shows a code editor with a sample program that initializes a counter, increments it, and prints its value. The output window shows the result: 0.8660254.
- Scastie**: An online Scala editor. The code editor contains a sample program for a counter class. The output window shows the generated Java code for the counter.
- codingground**: An online Scala compiler. It shows a code editor with a sample program for a counter class. The output window shows the command \$scalac *.scala.
- sl**: A platform for learning programming. It features a Tutorials section with categories like NUMPY, TKINTER, KOTLIN, and JAVASCRIPT. Below it is a Java Compiler section with tabs for Practice Coding and Beacon.
- labstack**: An online Java editor. It shows a code editor with a sample Java program that prints "Hello from Java!". The output window shows the command java Main.



Il problema della documentazione



UN ALTRO ASPETTO: LA DOCUMENTAZIONE

- È noto che un buon programma dovrebbe essere "ben documentato"...
- *...ma l'esperienza insegna che quasi mai ciò viene fatto!*
 - “non c’è tempo”, “ci si penserà poi”...
 - ..e alla fine la documentazione non c’è! ☹
- **Java** prende atto che *la gente non scrive documentazione* e quindi fornisce uno *strumento per produrla automaticamente a partire da particolari commenti* nel programma: ***javadoc***
- Un commento JavaDoc classico inizia con ***/***** (anziché ***/****)
 - poi, termina normalmente con ****/***
 - può essere **in testa a una classe** o a ***singole funzioni***



COMMENTI IN JAVADOC CLASSICO

- Cosa si può scrivere in un commento JavaDoc classico?
 - testo normale
 - testo HTML (con link se necessario)
 - tag
- *Flessibile, ma... potenzialmente assai verboso*

```
/** File Esempio.java
 * Applicazione Java da linea di comando
 * Stampa la classica frase di benvenuto
@author Enrico Denti
@version 1.0, 02/02/2025
*/
public class Esempio0 {
    public static void main(String args[]) {
        System.out.println("Hello!");
    }
}
```



COMMENTI IN JAVADOC CLASSICO

- Un commento assai verboso:

- codici HTML come ``,
``, etc.
- tag come `{@link}`,
`{@code}`, etc.

• Conseguenza: leggibilità
per un umano *non del
tutto soddisfacente*

• Per questo, Java 23 ha
introdotto una *sintassi
alternativa basata sul
markdown*.

```
/**  
 * Returns a hash code value for the object. This method is  
 * supported for the benefit of hash tables such as those provided by  
 * {@link java.util.HashMap}.  
 * <p>  
 * The general contract of {@code hashCode} is:  
 * <ul>  
 * <li>Whenever it is invoked on the same object more than once during  
 * an execution of a Java application, the {@code hashCode} method  
 * must consistently return the same integer, provided no information  
 * used in {@code equals} comparisons on the object is modified.  
 * This integer need not remain consistent from one execution of an  
 * application to another execution of the same application.  
 * <li>If two objects are equal according to the {@link  
 * #equals(Object) equals} method, then calling the {@code  
 * hashCode} method on each of the two objects must produce the  
 * same integer result.  
 * <li>It is not required that if two objects are unequal  
 * according to the {@link #equals(Object) equals} method, then  
 * calling the {@code hashCode} method on each of the two objects  
 * must produce distinct integer results. However, the programmer  
 * should be aware that producing distinct integer results for  
 * unequal objects may improve the performance of hash tables.  
 * </ul>  
 *  
 * {@implSpec}  
 * As far as is reasonably practical, the {@code hashCode} method defined  
 * by class {@code Object} returns distinct integers for distinct objects.  
 *  
 * @return a hash code value for this object.  
 * @see java.lang.Object#equals(java.lang.Object)  
 * @see java.lang.System#identityHashCode  
 */
```



JAVADOC IN JAVA 23+: MARKDOWN

- **Markdown** è un modo assai diffuso di scrivere documenti che siano al contempo *leggibili* ma anche facilmente trasformabili in HTML
- A differenza di un commento Javadoc classico, che inizia con `/**` e procede fino al `*/`, **un commento Javadoc-Markdown prevede all'inizio di ogni riga il tag `///`**
- Per gli usi più comuni, scrivere in Markdown è più rapido e leggibile dell'equivalente scrittura HTML.
 - elenchi puntati introdotti da trattini ‘-’
 - enfatizzazioni evidenziate da underscore ‘_’
 - codice espresso con backtick `code`
 - tante forme di link [link]



JAVADOC IN JAVA 23: MARKDOWN

- Lo stesso esempio di prima riscritto in Markdown:

- leggibilità per un umano
migliorata
- possibilità di usare anche asterischi e sequenze come `/* ... */` nel testo
- supporto semplificato per tavelle
- e altre feature carine

<https://openjdk.org/jeps/467>

```
/// Returns a hash code value for the object. This method is
/// supported for the benefit of hash tables such as those provided by
/// [java.util.HashMap].
///
/// The general contract of `hashCode` is:
///
/// - Whenever it is invoked on the same object more than once during
///   an execution of a Java application, the `hashCode` method
///   must consistently return the same integer, provided no information
///   used in `equals` comparisons on the object is modified.
///   This integer need not remain consistent from one execution of a
///   application to another execution of the same application.
/// - If two objects are equal according to the
///   [equals][#equals(Object)] method, then calling the
///   `hashCode` method on each of the two objects must produce the
///   same integer result.
/// - It is not required that if two objects are unequal
///   according to the [equals][#equals(Object)] method, then
///   calling the `hashCode` method on each of the two objects
///   must produce distinct integer results. However, the programmer
///   should be aware that producing distinct integer results for
///   unequal objects may improve the performance of hash tables.
///
/// @implSpec
/// As far as is reasonably practical, the `hashCode` method defined
/// by class `Object` returns distinct integers for distinct objects.
///
/// @return a hash code value for this object.
/// @see java.lang.Object#equals(java.lang.Object)
/// @see java.lang.System#identityHashCode
```



UN ESEMPIO "COMMENTATO" IN JAVADOC CLASSICO

```
/** File Esempio.java
 * Applicazione Java da linea di comando
 * Stampa la classica frase di benvenuto
@author Enrico Denti
@version 1.0, 02/02/2025
*/
public class Esempio0 {
    /**
     * Il main.
     * Stampa la classica frase di benvenuto
     */
    public static void main(String args[]) {
        System.out.println("Hello!");
    }
}
```

Informazioni di documentazione sulla classe intera

Informazioni di documentazione sul singolo metodo



Generazione automatica di documentazione

In Java, per produrre la relativa documentazione:

javadoc -d docs Esempio0.java

Produce nella cartella **docs**
un manuale HTML

Si consulti la
documentazione
di Javadoc per i dettagli.

In C#, analoga funzione
è svolta dal compilatore
csc con opzione /doc

The screenshot shows a Javadoc-generated HTML page for the class `Esempio0`. At the top, it says "Class Esempio0" and shows the inheritance chain: `java.lang.Object` → `Esempio0`. Below this is the class definition: `public class Esempio0 extends Object`. A red box highlights the Javadoc comment: `Applicazione Java da linea di comando. Stampa la classica frase di benvenuto`. To the right, a yellow callout bubble points to this text with the text: "La frase riportata è quella estratta dai commenti". Further down, there's a "Constructor Summary" section with a "Constructors" tab selected, showing the constructor `Esempio0()`. In the "Method Summary" section, the "All Methods" tab is selected, showing the method `main(String[] args)` with the description "Il main.". Below this, a "Methods inherited from class java.lang.Object" section lists methods like `clone`, `equals`, `finalize`, etc. At the bottom, there's a "Constructor Details" section for the constructor `Esempio0()`. To the right, another yellow callout bubble points to this section with the text: "I tag come @author, @version vengono inseriti solo a richiesta".



UN ESEMPIO "COMMENTATO" IN JAVADOC MARKDOWN

```
/// Applicazione Java da linea di comando.
```

```
/// Stampa la classica frase di benvenuto
```

```
/// @author Enrico Denti
```

```
/// @version 1.0, 15/1/2025
```

```
public class Esempio0Var {
```

```
    /// Il main.
```

```
    /// Stampa la classica frase di benvenuto
```

```
    public static void main(String args[]) {
```

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

```
}
```

```
}
```

Informazioni di documentazione sulla classe intera

Informazioni di documentazione sul singolo metodo

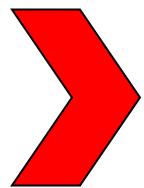


Generazione automatica di documentazione

In Java, per produrre la relativa documentazione:

javadoc -d docs Esempio0Var.java

Produce nella cartella
docs lo stesso
manuale HTML



The screenshot displays the generated Java documentation for the class `Esempio0Var`. The documentation is structured as follows:

- Class Esempio0Var**: Shows the inheritance chain from `java.lang.Object` to `Esempio0Var`.
- Constructor Summary**: Lists the constructor `Esempio0Var()`.
- Method Summary**: Lists the static method `main(String[] args)` with a brief description: "Il main."
- Methods inherited from class java.lang.Object**: Lists common methods like `clone`, `equals`, `finalize`, etc.
- Constructor Details**: Shows the definition of the `Esempio0Var()` constructor.



UN ALTRO ASPETTO: LA DOCUMENTAZIONE

- In **Scala** un analogo strumento è **Scaladoc**
 - segue al 99% la stessa sintassi di Javadoc
- In **Kotlin** un analogo strumento è **Kdoc**
 - la sintassi è un mix fra quella di Javadoc e il *markdown*
- In **C#** il compilatore può estrarre da commenti '**///**' un file **XML**, la cui elaborazione è lasciata però ad altri strumenti
 - strumenti come SandCastle, GhostDoc, NDoc3 generano vari formati (tipicamente, manuali in stile MSDN) a partire dall'XML

ESEMPIO IN C#

```
/// <summary>
/// testo di commento
/// </summary>
public class MyClass{ }
...
```



Java 25, Scala 3, Kotlin: curva di apprendimento e nuove forme compatte



IL PROBLEMA DELLA CURVA DI APPRENDIMENTO

- Java è spesso insegnato anche come primo linguaggio di programmazione, ma in tale contesto *tutta la struttura che «sta intorno» al main è un ostacolo all'apprendimento*
 - voi sapevate già cosa fosse un main dal C, ma chi vedesse i programmi di poco fa per la prima volta, cosa capirebbe...?
- Per questo, Java 25 ha introdotto i *file sorgenti compatti*
 - particolarmente adatti per neofiti
 - non richiedono di mettere il main in una classe → lo si può scrivere anche direttamente a livello di file (come in C, Kotlin, e.. Scala 3)
 - non richiedono di specificare che il main sia public, static
 - non richiedono neppure di specificare gli argomenti dalla riga di comando (l'array di stringhe)



IL PROBLEMA DELLA CURVA DI APPRENDIMENTO

- Programma scritto in Java standard

```
public class Esempio0 {  
    public static void main(String args[]) {  
        System.out.println("Hello!");  
    }  
}
```

Java

- Lo stesso programma scritto in forma compatta:

```
public class Esempio0 {  
    public static void main(String args[]) {  
        System.out.println("Hello!");  
    }  
}
```

Java 25

Molto più semplice per chi si avvicina
alla programmazione per la prima volta



IL PROBLEMA DELLA CURVA DI APPRENDIMENTO

- Lo stesso programma scritto in forma compatta:

```
// file ProvaJavaLight
void main() {
    System.out.println("Hello!");
}
```

Java 25

- Tuttavia, *dietro le quinte* viene generato un main standard, *incapsulato in una classe normale, inserita dal compilatore*
 - il nome della classe generata dal compilatore è *uguale al nome del file* in cui è scritto il main

ProvaJavaLight.class

03/01/2026 14:45

ProvaJavaLight.java

03/01/2026 14:46



IL PROBLEMA DELLA CURVA DI APPRENDIMENTO

- Lo stesso programma scritto in forma compatta:

```
// file ProvaJavaLight  
void main() {  
    System.out.println("Hello!");  
}
```

Java 25

- Tuttavia, *dietro le quinte* viene generato un main standard, *incapsulato in una classe normale, inserita dal compilatore*
 - il nome della classe generata dal compilatore è *uguale al nome del file* in cui è scritto il main
 - Poiché voi siete Ingegneri e dovete sapere cosa succede realmente, *in questo corso useremo preferibilmente la forma standard*, che garantisce la massima compatibilità.



IL PROBLEMA DELLA CURVA DI APPRENDIMENTO in Scala

- La stessa tematica è stata affrontata anche da Scala, nel contesto del redesign da Scala 2 a Scala 3
- In Scala 3, come in Kotlin, il main viene di norma definito *direttamente a livello di file, senza object attorno*
 - tale object è poi aggiunto «dietro le quinte» dal compilatore
 - come in Java, non richiede di specificare obbligatoriamente gli argomenti dalla riga di comando (l'array di stringhe)
- Di più: consente perfino di *chiamare il main con un nome diverso*, a propria scelta → max personalizzazione, ma...
...la trasparenza e la leggibilità...? 🤔
 - per riconoscere il main viene quindi introdotto il **tag @main**
 - il nome della classe generata dal compilatore è in questo caso assunto *uguale al nome della funzione-main*



IL PROBLEMA DELLA CURVA DI APPRENDIMENTO in Scala

- Programma scritto in Scala 2 (o Scala «prolisso»)

```
object Esempio1 {  
    def main(args: Array[String]):Unit = {  
        println("Hello");  
    }  
}
```

Scala

- Lo stesso programma scritto in Scala 3:

```
object Esempio1 {  
    @main Può chiamarsi come si vuole  
    def main(args: Array[String]):Unit = {  
        println("Hello");  
    }  
}
```

Scala 3

Più sintetico e anche più semplice
per chi è agli inizi



IL PROBLEMA DELLA CURVA DI APPRENDIMENTO in Scala

- Lo stesso programma scritto in Scala 3 (o quasi):

```
@main  
def main() = {  
    println("Hello");  
}
```

Scala 3

Qui, la classe generata si chiamerà «main» (sarà generata anche una serie di entità accessorie)

- In realtà, Scala consente anche di evitare le {} nel caso di funzioni il cui corpo sia costituito da un'unica istruzione

```
@main  
def main() =  
    println("Hello");
```

Scala 3

main\$package\$.class	03/01/2026 15:12
main\$package.class	03/01/2026 15:12
main\$package.tasty	03/01/2026 15:12
main.class	03/01/2026 15:12
main.scala	03/01/2026 15:12
main.tasty	03/01/2026 15:12



IL PROBLEMA DELLA CURVA DI APPRENDIMENTO in Scala

- Se lo avessimo chiamato con un altro nome:

```
@main  
def ProvaLight() = {  
    println("Hello");  
}
```

Scala 3

Qui, la classe generata si chiamerà «ProvaLight» (+entità accessorie)

- Versione super compatta senza {}:

```
@main  
def ProvaLight() =  
    println("Hello");
```

Scala 3

📄 ProvaLight\$package\$.class	03/01/2026 15:11
📄 ProvaLight\$package.class	03/01/2026 15:11
📄 ProvaLight\$package.tasty	03/01/2026 15:11
📄 ProvaLight.class	03/01/2026 15:08
📝 ProvaLight.scala	03/01/2026 15:11
📄 ProvaLight.tasty	03/01/2026 15:08



IL PROBLEMA DELLA CURVA DI APPRENDIMENTO in Kotlin

- Un discorso analogo vale anche per Kotlin
- Versione classica...

```
fun main():Unit {  
    println("Hello");  
}
```

Kotlin

- ... e versioni compatte:

```
fun main() {  
    println("Hello");  
}
```

Kotlin

```
fun main() = println("Hello");
```

Kotlin

Anche Kotlin consente di eliminare le {} nel caso di funzioni il cui corpo sia una singola istruzione: occorre allora un =



Le basi del linguaggio: tipi base



TIPI PRIMITIVI: sì o no?

- Java mantiene la nozione di *tipo primitivo* del C
 - pur estendendoli e ridefinendoli
 - approccio «conservativo» dovuto a ragioni storiche (chi proveniva dal C era abituato ad averli) e di prestazioni (all'epoca)
 - MA l'esperienza ha dimostrato che non è stata una grande idea!
- I linguaggi definiti successivamente (C#, Scala, Kotlin) li *sostituiscono* invece con opportuni *tipi di oggetti*
 - obiettivi: *uniformità & drastica semplificazione*
 - non vi sono più le ragioni storiche di vent'anni fa, né problemi di prestazioni («sotto banco» il tipo primitivo può anche esistere..)
 - talora «spacciati» per tipi primitivi mascherati (C#)
 - in Scala e Kotlin, si nota un leggero cambio di nome (`int` → `Int`)



BOOLEAN

- Un boolean non è più sinonimo di «intero 0/1»
- È un **tipo autonomo, *totalmente disaccoppiato* dagli interi**
 - le espressioni relazionali e logiche danno come risultato un boolean, non un int come in C
 - intenzionalmente non si convertono boolean in interi e viceversa, *neanche con cast* (bisogna scriversi funzioni apposite)
- SINTASSI
 - **Java** **boolean** (tipo primitivo)
 - **C#** **bool** (tipo «primitivo»)
 - **Scala** / **Kotlin** **Boolean** (tipo di oggetto)

Gli unici valori ammessi sono le costanti **false** e **true**, che *non sono 0 e 1*



BOOLEAN: IL CASO C#

- solo **C#** fornisce, nel componente **Convert**, una serie di funzioni di utilità che coprono anche questa casistica

```
public static void Main()
{
    bool a = true, b = false;
    int x = Convert.ToInt16(a); Console.WriteLine(x);
    int y = Convert.ToInt16(b); Console.WriteLine(y);
    int n = 51, z = 0;
    bool t = Convert.ToBoolean(n); Console.WriteLine(t);
    bool f = Convert.ToBoolean(z); Console.WriteLine(f);
}

1
0
True
False
```



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`

- Solo C#: interi senza segno

- **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`



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: $1 \cdot 10^{-16}$)
- Estensioni language-specific
 - solo C# **decimal** (16 byte) $-10^{28} \dots +10^{28}$
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 base 10;
ciò è utile nei calcoli finanziari

Contro: range più ridotto, occupa più spazio, velocità



NUMERI REALI: COMPATIBILITÀ

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

Ad esempio, in Java e C#:

- la frase **double x = 3.54F;** è *leccita* (da **float** a **double** non si perde precisione)
- la frase **float f = 3.54;** è *illeccita* (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 anch'essa *illeccita*

- In **Kotlin** invece le conversioni *implicite* non sono *mai* ammesse: è una scelta di progetto!



COMPATIBILITÀ FRA REALI

Conversioni implicite

Esempio corretto:

```
public class Esempio2 {  
    public static void main(String[] args) {  
        double x = 3.54F;                                // OK  
        double z = Math.sin(Math.PI/3);                  // OK  
    }  
}
```

Java

~C#

Esempio errato:

```
public class Esempio2 {  
    public static void main(String[] args) {  
        float f = 3.54;                                // NO!  
        float f = Math.sin(Math.PI/3);                  // NO!  
    }  
}
```

Java

~C#

ERRORE DI COMPILAZIONE
Possible loss of precision
Found double, required float



COMPATIBILITÀ FRA REALI

Conversioni implicite

Esempio corretto (sintassi classica «prolissa»):

```
object Esempio2 {  
    def main(args: Array[String]) = {  
        val z:Double = 3.54F; // OK  
        val x:Double = scala.math.sin(scala.math.Pi/3) // OK  
    }  
}
```

Tipo di ritorno : **Unit inferito**

Scala

Se non si specificasse il tipo di ritorno, la *type inference* dedurrebbe per la variabile lo stesso tipo dell'espressione alla destra dell' =

Esempio errato (sintassi classica «prolissa»):

```
object Esempio2 {  
    def main(args: Array[String]) = {  
        val z:Float = 3.54; // NO  
        val x:Float = scala.math.sin(scala.math.Pi/3) // NO  
    }  
}
```

Scala

ERRORE DI COMPILAZIONE
Found double, required float



COMPATIBILITÀ FRA REALI

Conversioni implicite

Esempio corretto (sintassi classica «prolissa»):

```
object Esempio2 {  
    def main(args: Array[String]) = {  
        val z:Double = 3.54F; // OK  
        val x:Double = scala.math.sin(scala.math.Pi/3) // OK  
    }  
}
```

Tipo di ritorno : **Unit inferito**

Scala

Se non si specificasse il tipo di ritorno, la *type inference* dedurrebbe per la variabile lo stesso tipo dell'espressione alla destra dell' =

Nuova sintassi Scala 3 senza { }:

*il codice va **indentato** a ogni livello di parentesi*

```
@main  
def Esempio2() =  
    val z:Double = 3.54F // OK  
    val x:Double = scala.math.sin(scala.math.Pi/3) // OK
```

Scala 3



COMPATIBILITÀ FRA REALI

Conversioni implicite

In Kotlin, l'esempio prima corretto diviene errato:

```
fun main(args: Array<String>) {  
    val z:Double = 3.54F; // NO  
    val x:Double = kotlin.math.sin(kotlin.math.PI/3) // OK  
}
```

Kotlin

ERRORE DI COMPILAZIONE

The floating-point literal does not conform to the expected type Float

L'esempio errato resta errato:

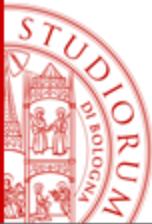
Tipo di ritorno :Unit inferito

```
fun main(args: Array<String>) {  
    val z:Float = 3.54; // NO  
    val x:Float = kotlin.math.sin(kotlin.math.PI/3) // NO  
}
```

Kotlin

ERRORE DI COMPILAZIONE

Type mismatch: inferred type is Double but Float was expected



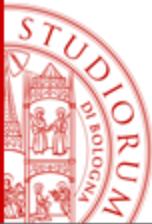
PERCHÉ KOTLIN FA DIVERSAMENTE?

- In **Kotlin** prevale l'idea che le conversioni debbano **essere esplicite anche quando non c'è perdita di informazione**, per far emergere sempre il Design Intent del progettista
- Un approccio moderno, ispirato dal principio di *type safety*

L'esempio corretto richiede *conversione esplicita*:

```
fun main(args: Array<String>) {  
    val z:Double = 3.54F.toDouble();      //OK  
    val x:Double = kotlin.math.sin(kotlin.math.PI/3) // OK  
}
```

Kotlin



COMPATIBILITÀ FRA REALI

Conversioni esplicite

- Se si vuole *consapevolmente* usare un `float` per memorizzare un valore `double`, accettando la *perdita di precisione che ne deriverà*, occorre *asserirlo esplicitamente*
 - in `Java` e `C#` con un *cast*
 - in `Scala` e `Kotlin` con opportune *funzioni di conversione* della forma `toXXX()`
- MOTIVO: per consentire un'operazione potenzialmente rischiosa, occorre che *il progettista renda esplicito il suo Design Intent*
 - ossia, dica chiaramente, scrivendo qualcosa, che ciò non è il frutto di una svista, ma è suo *preciso intendimento*



COMPATIBILITÀ FRA REALI

Conversioni esplicite

- SINTASSI

- in **Java** e **C#** il *cast* permette di specificare il tipo target scrivendolo, fra parentesi, davanti all'espressione da convertire

Ad esempio:

- la frase **float f = 3.54;** è *illecita*
 - la frase **float f = (float) 3.54;** è *lecita*

- in **Scala** e **Kotlin** *il cast non esiste*: si usano al suo posto opportune *funzioni di conversione* della forma **toXXX()**

Ad esempio, in Kotlin:

- la frase **val z:Double = 3.54F;** è *illecita*
 - la frase **val z:Double = 3.54F.toDouble();** è *lecita*



COMPATIBILITÀ FRA REALI

Conversioni esplicite in Scala e Kotlin

- In **Scala**, in ossequio al *principio di accesso uniforme*, le funzioni di conversione *non hanno* le parentesi finali

Esempi:

```
val z:Double = 3.54F.toDouble;  
val f:Float  = 3.54.toFloat;
```

- In **Kotlin** invece *le hanno* come d'abitudine:

Esempi:

```
val z:Double = 3.54F.toDouble();  
val f:Float  = 3.54.toFloat();
```



ESPERIMENTI ONLINE IN C#

.NET Fiddle New Fork Run Share Collaborate Tidy Up Getting Started Q

C# PlayGround by Alen Vadassery

```
⚠ 1 using System;
  2
  3 public class Esempio2 {
  4     public static void Main(string[] args){
  5         float f = System.Math.Sin(System.Math.PI/3);
  6         System.Console.WriteLine(f);
  7     }
  8 }
```

C#, senza conversione esplicita: ERRORE

Compilation error (line 5, col 12): Cannot implicitly convert type 'double' to 'float'. An explicit conversion exists (are you missing a cast?)

.NET Fiddle New Fork Run Share

C# PlayGround by Alen Vadassery

```
⚠ 1 using System;
  2
  3 public class Esempio2 {
  4     public static void Main(string[] args){
  5         float f = (float) System.Math.Sin(System.Math.PI/3);
  6         System.Console.WriteLine(f);
  7     }
  8 }
```

C#, con conversione esplicita: OK

0.8660254



ESPERIMENTI ONLINE IN Scala

The screenshot shows two side-by-side Scala repl sessions in Scastie.

Scala:

```
object Example {  
  def main(args:Array[String]):Unit = {  
    var x : Double = 3.54F;  
    println(x)  
  }  
}  
3.5399999618530273
```

Scala 3:

```
@main  
def Esempio2() =  
  val z:Double = 3.54F // OK  
  println(z)  
5  
3.5399999618530273
```

A blue callout bubble points to the first session with the text: "Scala, conversione隐式转换 without loss of information: OK". A green button labeled "Scala" is at the bottom of the left session.

The screenshot shows a Scala repl session in Scastie.

```
object Example {  
  def main(): Unit = {  
    var f : Float = 3.54;  
    type mismatch;  
    found : Double(3.54)  
    required: Float  
    println(f);  
  }  
}
```

A yellow callout bubble points to the error message with the text: "Scala, no explicit conversion: ERROR".

The screenshot shows a Scala repl session in Scastie.

```
object Example {  
  def main(args: Array[String]): Unit = {  
    var f : Float = 3.54.toFloat;  
    Console.println(f);  
  }  
}
```

A blue callout bubble points to the result "3.54" with the text: "Scala, conversion esplicita col metodo toFloat: OK".



ESPERIMENTI ONLINE in Kotlin

Kotlin Playground is an online sandbox to explore Kotlin programming language. Browse code samples directly in the browser

Playground Hand

```
/*
fun main() {
    var x : Double = 3.54F;
    println(x)
}
```

The floating-point literal does not conform to the expected type Double

Kotlin, senza conversione esplicita: ERRORE

Kotlin Playground is an online sandbox to explore Kotlin programming language. Browse code samples directly in the browser

Playground

```
/*
fun main() {
    var x : Double = 3.54F.toDouble();
    println(x)
}
```

3.5399999618530273

Kotlin, conversione esplicita tramite il metodo toDouble: OK



CARATTERI (1/2)

- A differenza del C, «carattere» non è più sinonimo di «byte»
 - 127 caratteri non bastano più da un molto tempo!
 - il mondo non ospita solo le culture occidentali...
- Nuovo approccio: un «carattere» di 2 byte (UTF-16)
 - primi 127 caratteri uguali ad ASCII, primi 255 ad ANSI / ASCII
- Standard UNICODE
 - *Basic Multilingual:* 2 byte = 65536 «code point»
 - *Supplementary characters:* altri 16×65536 «code point»
 - "Support for supplementary characters is a common business requirement in East Asian markets. Government applications require them to correctly represent rare Chinese characters. Publishing applications need them to represent the full set of historical and variant characters. The Hong Kong government defined characters needed for Cantonese"



CARATTERI (2/2)

- Standard UNICODE
 - full range da 000000_H a $10FFFF_H$ (1.114.112 caratteri)
 - suddivisi in 17 «piani» da 65.536 caratteri ciascuno ($5+16=21$ bit)
 - *Basic Multilingual* piano 0: range da 00000_H a $0FFFF_H$
 - *Supplementary Multilingual*: piano 1: range da 10000_H a $1FFFF_H$
 - Ulteriori caratteri: piani 2+: range da 20000_H a $10FFFF_H$

Un carattere («code point») si indica con la notazione $U+nnnn$

NB: l'intervallo da D800 a DFFF del piano 0 non è assegnato (serve per UTF-16)

Piano	Intervallo	Descrizione
0	000000-00FFFF	Basic Multilingual
1	010000-01FFFF	Supplementary Multilingual
2	020000-02FFFF	Supplementary Ideographic
3	030000-03FFFF	Tertiary Ideographic
...	040000-0DFFFF	<i>non usati</i>
14	0E0000-0EFFFF	Supplementary Special-purpose
15	0F0000-0FFFFF	Supplementary Private Use Area A
16	100000-10FFFF	Supplementary Private Use Area B



UNICODE Basic Multilingual

- U+0000 to U+007F: [Basic Latin](#)
 - U+0080 to U+00FF: [Latin-1 Supplement](#)
 - U+0100 to U+017F: [Latin Extended-A](#)
 - U+0180 to U+024F: [Latin Extended-B](#)
 - U+0250 to U+02AF: [IPA Extensions](#)
 - U+02B0 to U+02FF: [Spacing Modifier Letters](#)
 - U+0300 to U+036F: [Combining Diacritical Marks](#)
 - U+0370 to U+03FF: [Greek and Coptic](#)
 - U+0400 to U+04FF: [Cyrillic](#)
 - U+0500 to U+052F: [Cyrillic Supplement](#)
 - U+0530 to U+058F: [Armenian](#)
 - U+0590 to U+05FF: [Hebrew](#)
 - U+0600 to U+06FF: [Arabic](#)
 - U+0700 to U+074F: [Syriac](#)
 - U+0750 to U+077F: [Arabic Supplement](#)
 - U+0780 to U+07BF: [Thaana](#)
 - U+0900 to U+097F: [Devanagari](#)
 - U+0980 to U+09FF: [Bengali](#)
 - U+0A00 to U+0A7F: [Gurmukhi](#)
 - U+0A80 to U+0AFF: [Gujarati](#)
 - U+0B00 to U+0B7F: [Oriya](#)
 - U+0B80 to U+0BFF: [Tamil](#)
 - U+0C00 to U+0C7F: [Telugu](#)
 - U+0C80 to U+0CFF: [Kannada](#)
 - U+0D00 to U+0D7F: [Malayalam](#)
 - U+0D80 to U+0DFF: [Sinhala](#)
 - U+0E00 to U+0E7F: [Thai](#)
 - U+0E80 to U+0EFF: [Lao](#)
 - U+0F00 to U+0FFF: [Tibetan](#)
 - U+1000 to U+109F: [Myanmar](#)
 - U+10A0 to U+10FF: [Georgian](#)
 - U+1100 to U+11FF: [Hangul Jamo](#)
 - U+1200 to U+137F: [Ethiopic](#)
 - U+1380 to U+139F: [Ethiopic Supplement](#)
 - U+13A0 to U+13FF: [Cherokee](#)
 - U+1400 to U+167F: [Unified Canadian Aboriginal Syllabary](#)
 - U+1680 to U+169F: [Ogham](#)
 - U+16A0 to U+16FF: [Runic](#)
 - U+1700 to U+171F: [Tagalog](#)
 - U+1720 to U+173F: [Hanunoo](#)
 - U+1740 to U+175F: [Buhid](#)
 - U+1760 to U+177F: [Tagbanwa](#)
 - U+1780 to U+17FF: [Khmer](#)

- U+1780 to U+17FF: [Khmer](#)
 - U+1800 to U+18AF: [Mongolian](#)
 - U+1900 to U+194F: [Limbu](#)
 - U+0400 to U+04FF: [Cyrillic](#)
 - U+3000 to U+303F: [CJK Symbols and Punctuation](#)
 - U+3040 to U+309F: [Hiragana](#)
 - U+30A0 to U+30FF: [Katakana](#)

- U+0500 to U+052F: [Cyrillic Supplement](#)
 - U+0530 to U+058F: [Armenian](#)
 - U+0590 to U+05FF: [Hebrew](#)

- U+0600 to U+06FF: Arabic

۱	۲	۳	۴	۵	۶	۷	۸	۹	۰
ع	ظ	ط	ض	ص	ش	س	ز	ر	ذ
ب	ت	ث	ج	چ	چ	ت	ت	ب	ا
ا	و	ؤ	ئ	ى	ى	ي	ي	ا	ء
۱	۲	۳	۴	۵	۶	۷	۸	۹	٪
۰	۱	۲	۳	۴	۵	۶	۷	۸	*
۰	۱	۲	۳	۴	۵	۶	۷	۸	۹
۰	۱	۲	۳	۴	۵	۶	۷	۸	۹
۰	۱	۲	۳	۴	۵	۶	۷	۸	۹
۰	۱	۲	۳	۴	۵	۶	۷	۸	۹

- U+3000 to U+303F: [CJK Symbols and Punctuation](#)



UNICODE OVERVIEW

Scripts

European Scripts	African Scripts	South Asian Scripts	East Asian Scripts
Armenian	Bamum	Bengali	Bopomofo
Armenian Ligatures	Bamum Supplement	<i>Brahmi</i>	Bopomofo Extended
Coptic	Egyptian Hieroglyphs (1MB)	Devanagari	CJK Unified Ideographs (Han) (28MB)
Coptic in Greek block	Ethiopic	Devanagari Extended	CJK Extension-A (6.3MB)
Cypriot Syllabary	Ethiopic Supplement	<i>Gujarati</i>	CJK Extension B (3.0MB)
Cyrillic	Ethiopic Extended	<i>Gurmukhi</i>	CJK Extension C (2.8MB)
Cyrillic Supplement	Ethiopic Extended-A	<i>Kaithi</i>	CJK Extension D
Cyrillic Extended-A		<i>Kannada</i>	(see also Unihan Database)
Cyrillic Extended-B		<i>Kharoshthi</i>	CJK Compatibility Ideographs (.5MB)
Georgian	N'Ko	Lepcha	CJK Radicals / KangXi Radicals
Georgian Supplement	Osmanya	<i>Limbu</i>	CJK Radicals Supplement
Glagolitic	Tifinagh	Malayalam	CJK Strokes
Gothic	Vai	Meetei Mayek	Ideographic Description Characters
Greek	Middle Eastern Scripts	Ol Chiki	Hangul Jamo
Greek Extended	Arabic	Oriya	Hangul Jamo Extended-A
Latin	Arabic Supplement	Saurashtra	Hangul Jamo Extended-B
Latin-1 Supplement	Arabic Presentation Forms-A	Sinhala	Hangul Compatibility Jamo
Latin Extended-A	Arabic Presentation Forms-B	Syloti Nagri	Halfwidth Jamo
Latin Extended-B	Aramaic, Imperial	Tamil	Hangui Syllables (.7MB)
Latin Extended-C	Avestan	Telugu	Hiragana
Latin Extended-D	Carian	Thaana	Katakana
Latin Extended Additional	Cuneiform (1MB)	Vedic Extensions	Katakana Phonetic Extensions
Latin Ligatures	Cuneiform Numbers and Punctuation	Southeast Asian Scripts	Kana Supplement
Fullwidth Latin Letters	Old Persian	Batak	Halfwidth Katakana
Linear B	Ugaritic	Balinese	Kanbun
Linear B Syllaby	Hebrew	Buginese	Lisu
Linear B Ideograms	Hebrew Presentation Forms	Cham	Yi
Ogham	Lydian	Javanese	Yi Syllables (.5MB)
Old Italic	Mandaic	Kayah Li	Yi Radicals
Phaistos Disc	Old South Arabian	Khmer	American Scripts
Runic	Pahlevi, Inscriptional	Khmer Symbols	Cherokee
Shavian	Parthian, Inscriptional	Lao	Deseret
Phonetic Symbols	Phoenician	Myanmar	Unified Canadian Aboriginal Syllabics
IPA Extensions	Samaritan	Myanmar Extended-A	UCAS Extended
Phonetic Extensions	Syriac	New Tai Lue	Other
Phonetic Extensions Supplement	Central Asian Scripts	Rejang	Alphabetic Presentation Forms
Modifier Tone Letters	Mongolian	Sundanese	Halfwidth and Fullwidth Forms
Spacing Modifier Letters	Old Turkic	Tai Le	ASCII Characters
Superscripts and Subscripts	Phags-Pa	Tai Tham	
Combining Diacritics	Tibetan	Tai Viet	
Combining Diacritical Marks		Thai	
Combining Diacritical Marks Supplement		Philippine Scripts	
Combining Half Marks		Buhid	
		Hanunoo	

ESEMPIO di Unicode Supplementary Characters

Tengwar
Cirth
Old Persian



UNICODE OVERVIEW

Questi supplementary character li conoscete..?
Code point da $1F600_H$ in poi

Smileys & Emotion								
Nº	Code	Browser	Sample	GMail	SB	DCM	KDDI	CLDR Short Name
1	U+1F600	😊	😊	😊	—	—	—	grinning face
2	U+1F603	😊	😊	😊	😊	😊	😊	grinning face with big eyes
3	U+1F604	😊	😊	😊	😊	—	—	grinning face with smiling eyes
4	U+1F601	😁	😁	😁	😁	😁	😁	beaming face with smiling eyes
5	U+1F605	😆	😆	😆	—	😆	—	grinning squinting face
6	U+1F606	😅	😅	😅	😅	😅	—	grinning face with sweat
7	U+1F923	🤣	🤣	—	—	—	—	rolling on the floor laughing
8	U+1F602	😂	😂	😂	😂	—	😂	face with tears of joy
9	U+1F642	😊	😊	😊	—	—	—	slightly smiling face



CARATTERI: SINTASSI

- Java / C# **char** (tipo primitivo in Java)
- Scala / Kotlin **Char** (tipo di oggetto)
- Conversioni carattere ↔ intero:
 - in Java e Scala conversione *automatica*
 - in C# conversione implicita tramite cast (nel verso int → char)
 - in Kotlin conversione *esplicita* tramite appositi metodi toXXX()

```
public static void main(String[] args)
{
    char ch = 'A';
    int x = ch;
    ch = 72;
    System.out.println(ch);
}
```

----- Java Run
H

Java

C# PlayGround by Alen Vadassery

```
1 using System;
2
3 public class Program
4 {
5     public static void Main()
6     {
7         char ch = 'A';
8         int x = ch;
9         ch = (char) 72;
10        Console.WriteLine(ch);
11    }
12 }
```

H

C#

```
fun main(args: Array<String>) : Unit {
    var ch : Char = 'A';
    var x : Int = ch.toInt();
    ch = 72.toChar();
    println(ch);
}
```

Kotlin

```
def main(args: Array[String]) : Unit = {
    var ch : Char = 'A';
    var x : Int = ch;
    ch = 72;
    println(ch);
}
```

Scala



DA UNICODE A UTF

- Le migliaia di caratteri possibili in Unicode pongono il problema di come specificare caratteri *non presenti sulle tastiere*
- Si usa una *codifica semi-numerica*: '`\u2122`'
- Unicode però si limita ad assegnare codici ai caratteri:
non dice come debbano essere mappati su sequenze di byte
- **UTF = Unicode Transformation Format**
a mapping from every Unicode code point to a unique byte sequence
 - UTF-8 a lunghezza variabile, usa da 1 a 4 byte per carattere
 → molto usato per testo, email, ...
 - UTF-16 a lunghezza variabile, usa 2 o 4 byte; i caratteri extra sono rappresentati da *coppie di codici* usando il range riservato fra D800-DFFF → utile in programmazione
 - UTF-32 usa sempre 4 byte → semplice, ma vorace di memoria



PERCHÉ UTF?

- Unicode elenca migliaia di caratteri e li numera, ma ci sono tanti modi di «scrivere concretamente» quei numeri
- Storicamente, ogni piattaforma faceva un po' da sé
 - ASCII standard per tutti, ma solo per i 127 caratteri inglesi..
 - .. da lì in poi, tanti standard diversi incompatibili fra loro
 - e non parliamo del ritorno a capo: CR o CR+LF? (Mac/Unix vs. Win)
- **UTF** è una sorta di «*lingua franca*» per far interoperare macchine e piattaforme *anche molto diverse fra loro*
 - **UTF-8 in particolare è usatissimo per le email, o negli editor per assicurarsi che il formato sia leggibile anche su altri computer**
 - se un testo non è UTF e lo condividi con qualcun altro (che magari ha un Mac mentre tu hai Windows), molti caratteri risulteranno «sbagliati» o illeggibili – a partire dalle *lettere accentate!*



UTF-8

- UTF-8
 - usa 1 solo byte per i primi 128 caratteri → **compatibile ASCII**
 - usa 2 byte per i successivi 1920 caratteri (quasi tutti i più usati)
 - usa 3 byte per i rimanenti caratteri del Basic Multilingual
 - usa 4 byte solo per gli altri piani Unicode (tra cui molte Emojis...)

UTF-8 online calculator

www.browserling.com/tools/utf8-encoder
www.browserling.com/tools/utf8-decoder

char	Code point	Valore in binario	UTF-8
'\$'	U+0024	010 0100 (7 bit significativi)	00100100
'£'	U+00A3	000 1010 0011 (11 bit significativi)	11000010 1010 0011
'€'	U+20AC	0010 0000 1010 1100 (16 bit sign.)	11100010 10000010 10101100
😺	U+01F608	0 0001 1111 0110 0000 1000 (21 bit)	11110000 10011111 10011000 10001000



UTF-8

- UTF-8
 - usa 1 solo byte per i primi 128 caratteri → **compatibile ASCII**
 - usa 2 byte per i successivi 1920 caratteri (quasi tutti i più usati)
 - usa 3 byte per i rimanenti caratteri del Basic Multilingual
 - usa 4 byte solo per gli altri piani Unicode (tra cui molte Emojis...)

UTF-8 online calculator

www.browserling.com/tools/utf8-encoder
www.browserling.com/tools/utf8-decoder

char	Code point	Binary representation
'\$'	U+0024	00000000 00000000 00000000 00000000
'£'	U+00A3	00000000 00000000 00000000 01000011
'€'	U+20AC	00000000 00000000 00000000 10101010
𩫔	U+01F608	0 0001 1111 0110 0000 1000 (21 bit) 11110000 10011111 10011000 10001000

Occhio alle Emoji nei messaggi.. !

Sono supplementary character, occupano l'equivalente di 4 caratteri standard ciascuno!



UTF-16

- UTF-16
 - usa 2 byte per i primi 65536 caratteri (il Basic Multilingual)
 - usa 4 byte per gli altri piani Unicode
 - più complesso ma efficiente → usato in Java, .NET, macOS
 - per distinguere le sequenze di 2 byte da quelle di 4 byte, queste ultime sono rappresentate tramite una coppia di valori nel range D800-DFFF, che Unicode (Basic Multilingual) mantiene riservati

char	Code point	Valore in binario	UTF-16
'\$'	U+0024	010 0100 (7 bit significativi)	00000000 00100100
'£'	U+00A3	000 1010 0011 (11 bit significativi)	00000000 1010 0011
'€'	U+20AC	0010 0000 1010 1100 (16 bit sign.)	00100000 10101100
😺	U+01F608	0 0001 1111 0110 0000 1000 (21 bit)	11011000 00111101 11011110 00001000

\uD83D \uDE08



UTF-16

- **UTF-16**

- i caratteri fino a FFFF sono codificati senza modifiche
- quelli da 10000 in su, si esprimono come *coppia surrogata*

UTF-16 online calculator

<https://convertcodes.com/utf16-encode-decode-convert-string/>

- MA l'ordine con cui sono memorizzati in memoria può variare
→ 4 sotto-codifiche lecite: UTF-16 (2 versioni), UTF-16LE, UTF-16BE

char	Code point	UTF-16 binario	UTF-16 hex		
𧈧	U+01F608	11011000 00111101 11011110 00001000	D8 3D DE 08		
UTF-16 hex		UTF-16 (LE)	UTF-16 (BE)	UTF-16BE	UTF-16LE
D8 3D DE 08		FF,FE,3D,D8,08,DE	FE,FF,D8,3D,DE,08	D8,3D,DE,08	3D,D8,08,DE

\uFEFF è un marcatore usato per distinguere in modo automatico Little Endian da Big Endian



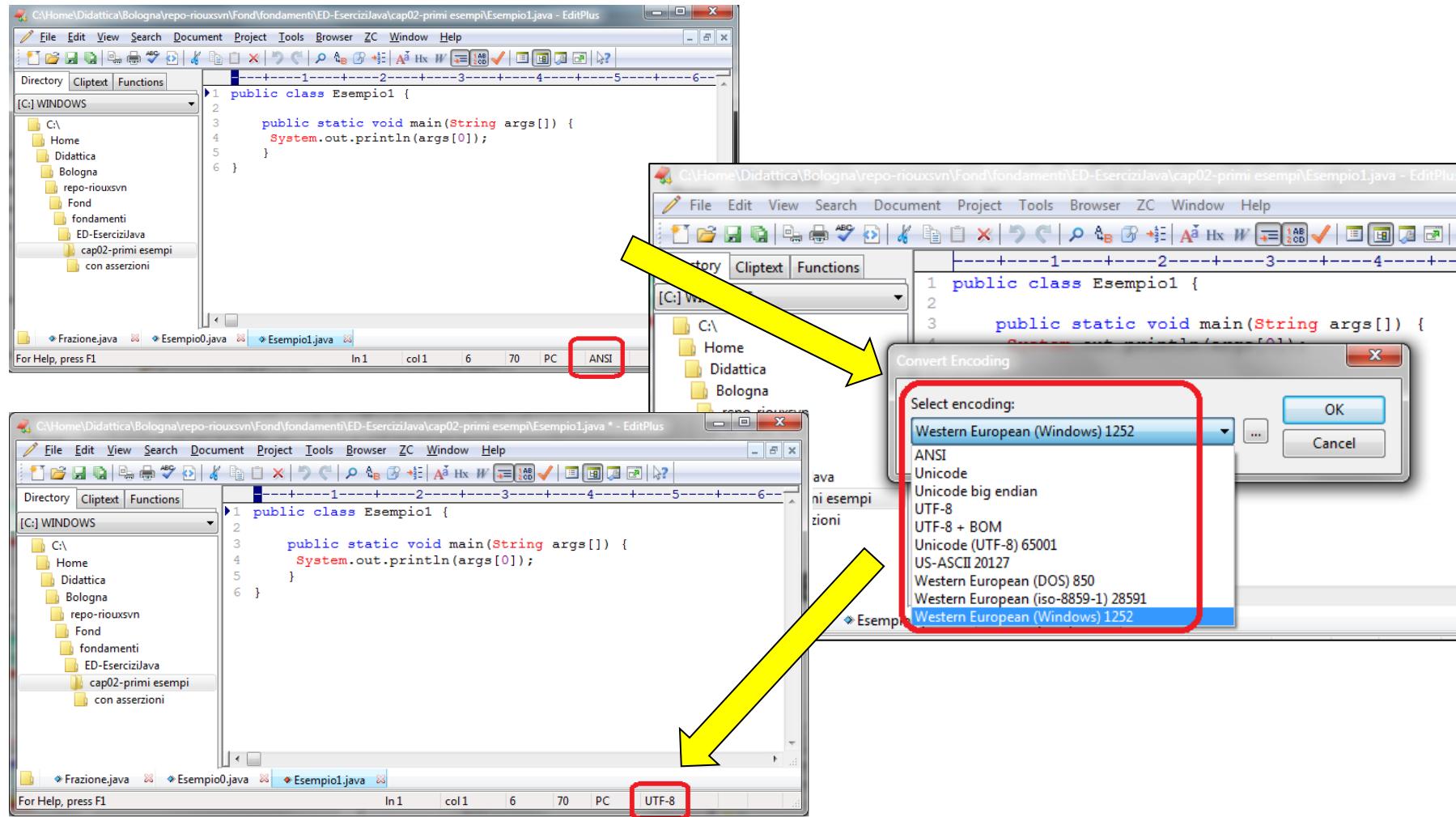
UTF-32

- UTF-32
 - usa sempre e comunque 4 byte per tutti i caratteri Unicode
 - molto semplice, MA usa una *quantità sproporzionata di memoria!*
 - nel 99,99% dei casi, i caratteri usati sono Basic Multilingual, che richiederebbero solo 2 byte; per non parlare dei primi 128 caratteri ASCII, che ne richiederebbero uno solo!
 - Morale: vantaggio più apparente che reale (gli editor di testo devono comunque gestire i caratteri combinati, gli ideogrammi..)

char	Code point	Valore in binario	UTF-32
'\$'	U+0024	010 0100 (7 bit significativi)	0000000000000000 000000000100100
'£'	U+00A3	000 1010 0011 (11 bit significativi)	0000000000000000 0000000010100011
'€'	U+20AC	0010 0000 1010 1100 (16 bit sign.)	0000000000000000 0010000010101100
🎭	U+01F608	0 0001 1111 0110 0000 1000 (21 bit)	0000000000000001 1111011000001000



UTF NEGLI EDITOR





UTF NEGLI EDITOR

