Alma Mater Studiorum-Università di Bologna
Scuola di Ingegneria

Classi e Oggetti

Corso di Laurea in Ingegneria Informatica

Anno accademico 2021/2022

Prof. ENRICO DENTI

Dipartimento di Informatica – Scienza e Ingegneria (DISI)

Componenti software singleton (statici)

COMPONENTI SOFTWARE

Che componenti software avete conosciuto finora?

- componenti statici *senza stato* (*librerie*)
  - non c'è stato: solo collezioni di funzioni *prive di effetti collaterali*
  - ne basta una sola copia in ogni progetto: tutti possono usarla
- componenti statici *con stato* (*moduli* come *singleton*)
  - c'è uno stato *unico e globale* (variabili globali static)
  - possono essere visti come *oggetti singleton:* ne esiste una e una sola copia nel progetto ed è giusto che sia così
- tipi di dato astratti (ADT)
  - c'è una *definizione di tipo unica e condivisa* (typedef) + funzioni che operano su quel tipo di dato (header + implementazione)
  - il cliente crea tante variabili (oggetti) di quel tipo quante ne occorrono

COMPONENTI SOFTWARE

Che componenti software avete conosciuto finora?

- componenti statici *senza stato* (*librerie*)
  - non c'è stato: solo collezio
  - ne basta una sola copia in

Esempi: libreria matematica, libreria stringhe, libreria matrici, etc.

- componenti statici *con stato* (*moduli* come *singleton*)
  - c'è uno stato *unico e glob*
  - possono essere visti come
    copia nel progetto ed è giusto che sia così

Esempi: risorse singleton (centro di stampa, contatore unico..)

- tipi di dato astratti (ADT)
  - c'è una *definizione di tipo* u
    operano su quel tipo di dato
  - il cliente crea tante variabili (oggetti) di quel tipo quante ne occorrono

Esempi: tutti i tipi da voi definiti! Frazioni, Liste, Contatori..

COMPONENTI SOFTWARE in C

Che componenti software avete conosciuto finora?

- componenti statici *senza stato* (*librerie*)

Header file con le dichiarazioni delle funzioni

File di implementazione con le definizioni delle funzioni

- componenti statici *con stato* (*moduli* come *singleton*)

Header file con le dichiarazioni delle funzioni

File di implementazione con le definizioni delle funzioni e variabili statiche per lo stato

- tipi di dato astratti (ADT)

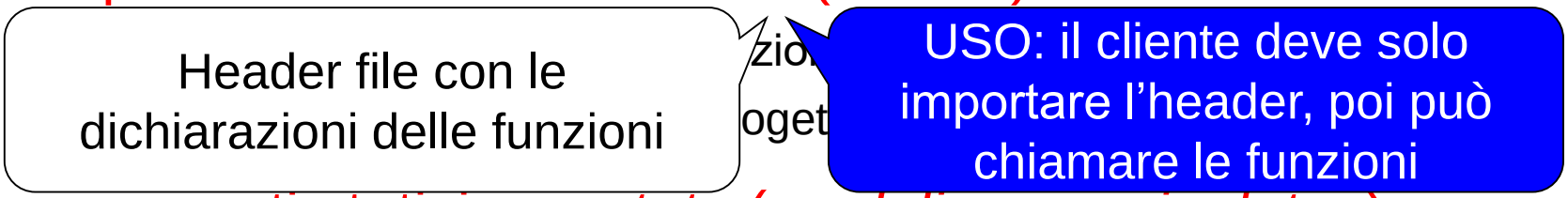
Header file con le dichiarazioni delle funzioni

File di implementazione con le definizioni delle funzioni

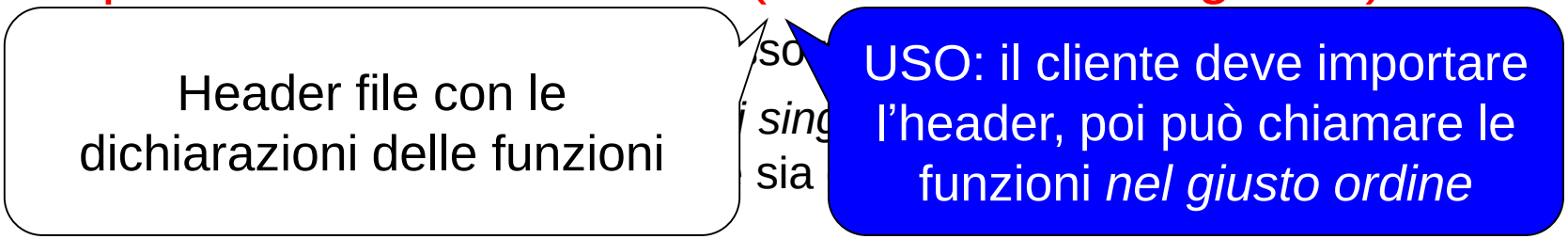
COMPONENTI SOFTWARE in C

Che componenti software avete conosciuto finora?

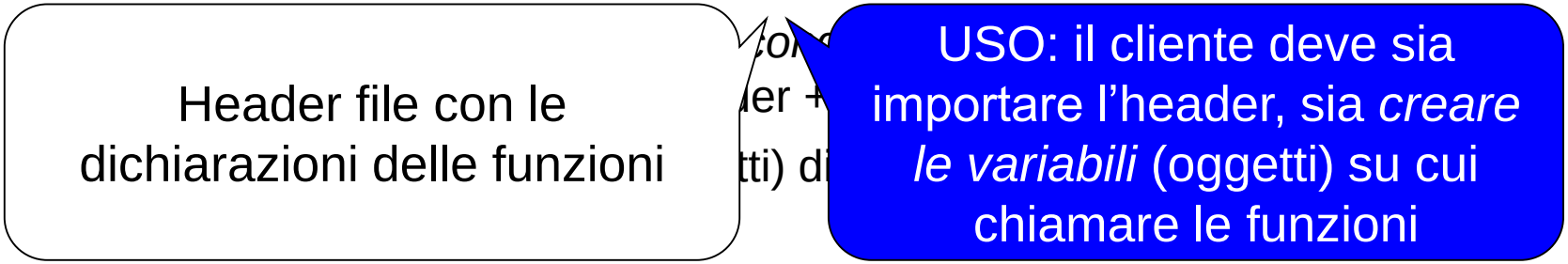
- componenti statici *senza stato* (*librerie*)



- componenti statici *con stato* (*moduli* come *singleton*)



- tipi di dato astratti (ADT)



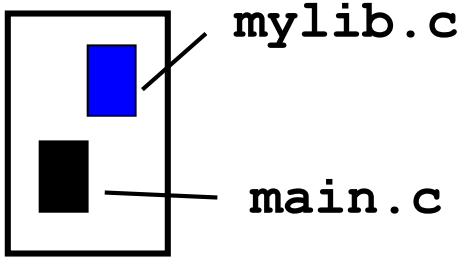
LIBRERIE: ESEMPIO in C

```
#include <stdio.h>
#include "mylib.h"
main() {
   int x;
   scanf("%d", &x);
   printf("Esito: %d", myfun(x));
}
```

Dichiara la funzione
`int myfun(int x);`

Non sono necessarie operazioni preliminari per usare le funzioni

Architettura progetto:



Implementa la funzione
`int myfun(int x);`

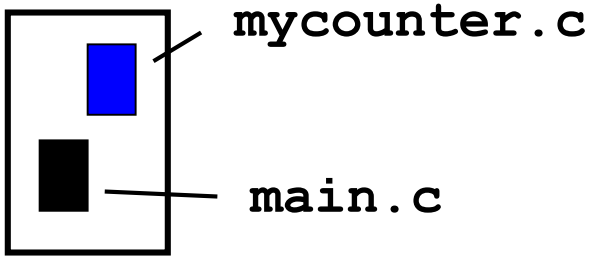
MODULI SINGLETON: ESEMPIO in C

```
#include <stdio.h>
#include "mycounter.h"
main() {
   reset();
   printf("Stato risorsa: %s", mystate() );
}
```

Dichiara le funzioni per usare l'entità «counter»

Non sono necessarie operazioni preliminari, ma *c'è un ordine da osservare* per inizializzare correttamente la risorsa

Architettura progetto:



Implementa le funzioni **reset**, **mystate**, etc.

ADT: ESEMPIO in C

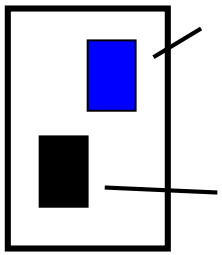
```
#include <stdio.h>
#include "fraction.h"
main() {
   Fraction f1;
   Fraction *pf2;
   pf2 = malloc(…);
   initfrac(&f1,3,4); initfrac(pf2,6,7);
}
```

Dichiara il tipo e le funzioni che caratterizzano l'entità «frazione»

Occorre preliminarmente *creare le variabili* (o puntatori) che rappresentano i singoli oggetti. Se puntatori, vanno anche allocati.

ATTENZIONE: la typedef svela la struttura interna del tipo

Architettura progetto:



fraction.c

main.c

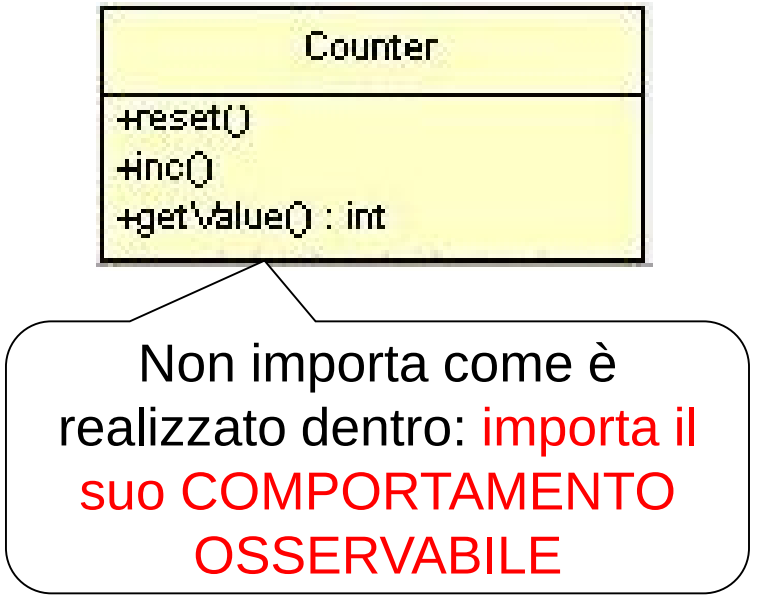
Implementa le funzioni **initfrac**, etc.

UN ESEMPIO CONCRETO: IL "CONTATORE"

Cos'è un *contatore* ? Dobbiamo definirlo noi.

Possibile definizione (*comportamento osservabile*):
è un componente caratterizzato da un valore (intero e variabile nel tempo) e tre *operazioni pubbliche:*

- **`reset`** per impostare il contatore a un valore iniziale noto (ad es. 0)
- **`inc`** per incrementare il valore attuale del contatore
- **`getValue`** per recuperare il valore attuale del contatore sotto forma di numero intero.

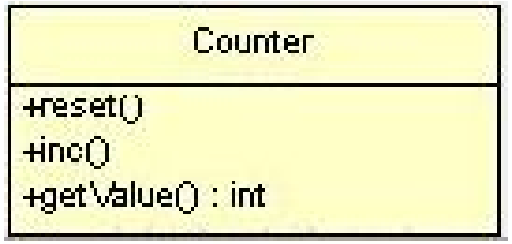


USARE (& COLLAUDARE) IL “CONTATORE”

- Per *usare* questo componente serve solo conoscere il suo *comportamento osservabile*: *non come è fatto dentro!*
  - anzi: conoscere i dettagli interni sarebbe *controproducente*, perché potrebbe portarci a inserire dipendenze inopportune.
- Fra tali usi c'è anche il collaudo: per progettarlo *non si deve sapere come è fatto dentro, anzi!*
  - il collaudo va progettato "a priori", perché il suo scopo è proprio *validare il componente*
- Dunque, come potremmo collaudare un contatore?
  - come minimo, test di ogni operazione in diverse situazioni, *con special cura per i casi limite e i casi particolari*.

CONTATORE: PIANO DI COLLAUDO

- Poiché il contatore espone tre operazioni, occorre sicuramente collaudarle tutte, singolarmente
- Come esprimere il "risultato atteso" del collaudo?
  - *non certo* indicando cosa stamperà a video!
  - occorre *dire cosa ci aspettiamo che avvenga* in risposta a una certa operazione (o sequenza di operazioni)
- Possibile piano di collaudo:
  - mi aspetto che eseguendo nell'ordine **`reset`** e **`inc`**, il valore del contatore sia 1
  - mi aspetto che eseguendo **`reset`** e 4 **`inc`**, il valore sia 4; e che tale rimanga anche successivamente
  - …eccetera…



PIANO DI COLLAUDO: PERCHÉ?

- A che serve questo?
  - ora possiamo assegnare la realizzazione ad altri
  - quando ci porteranno il componente realizzato, *avremo modo di controllare la bontà e la qualità della altrui realizzazione*
  - se non ci convince, *non lo paghiamo!*
- Il piano di collaudo diventa *parte integrante delle specifiche* del "capitolato contrattuale"
  - chi accetta la nostra commessa di realizzazione del contatore SA che il suo lavoro sarà CONTROLLATO E VALIDATO da noi SECONDO QUESTE SPECIFICHE
  - patti chiari, amicizia lunga

PIANO DI COLLAUDO: COME ?

MA.. come esprimiamo tutto questo in pratica?

- solo a parole..?
  - possibile, ma adatto solo a piccoli progetti
  - insostenibile per progetti complessi, con piani di collaudo ampi
  - occorre *uscire dall'artigianato* per approdare all'ingegneria, che richiede *specifiche formalizzate e strumenti di supporto*
- collaudo formalizzato e strumentato
  - tramite il concetto di *asserzione* disponibile in Java, C#, Scala, Kotlin
    ```
    c.reset(); c.inc(); assert c.getValue()==1;
    ```
  - idea sfruttata estensivamente in Java dallo strumento *JUnit* che esegue automaticamente decine di test riportandone i risultati
  - in tal caso i test sono scritti a parte, *non nel programma* (che così resta leggibile) e le asserzioni assumono una forma ad hoc
  - MORALE: il progettista progetta i test e lo strumento li esegue

CONTATORE: UNA REALIZZAZIONE IN C

- In C non abbiamo supporto per le attività di collaudo: dovremo *arrangiarci da soli* (nei limiti del possibile…)
- Una prima possibile scelta: contatore = modulo con stato
  - Stato del contatore *inaccessibile dall'esterno ossia fuori dal file di definizione* → keyword **`static`**
  - Accesso allo stato solo attraverso *funzioni (pubbliche)*:

```
void reset(void);
void inc(void);
int  getValue(void);
```

`mcounter.h`

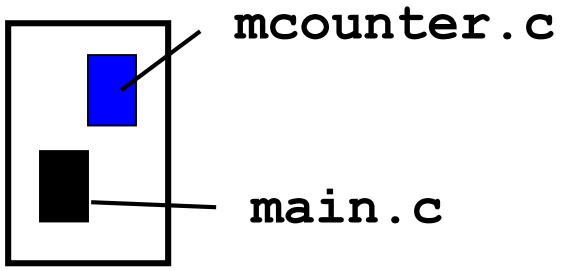
- Per USARE il contatore, si deve:
  - aggiungere al progetto il sorgente del contatore (es. **`mcounter.c`**)
  - includere nel file cliente (es. **`mymain.c`**) l'opportuno file *header* (es. **`mcounter.h`**) che dichiari le tre operazioni sopra citate

UN POSSIBILE CLIENTE

```
/* mymain.c */
#include <stdio.h>
#include "mcounter.h"
main() {
   reset(); inc(); int v = getValue();
   printf("Esito: %s", v==1 ? "ok" : "failed" );
}
```

```
void reset(void);
void inc(void);
int getValue(void);
```

OSSERVA: *questo main sta* applicando "artigianalmente" il Piano di Collaudo…!



NOTE:

- Non occorre creare esplicitamente il contatore, perché esso coincide col modulo **`mcounter.c`** che fa parte del progetto
- Le operazioni non hanno più l'argomento contatore perché è implicito su quale contatore agiscano: l'unico presente!

UNA POSSIBILE REALIZZAZIONE IN C

mcounter.h

```
void reset(void);
void inc(void);
int  getValue(void);
```

Questa realizzazione è accettata come corretta *solo se supera il collaudo* svolto come indicato nel Piano di Collaudo.

mcounter.c

```
static int stato;
void reset(){stato=0;}
void inc(){stato++;}
int getValue(){return stato;}
```

Stato *inaccessibile dall'esterno del file*

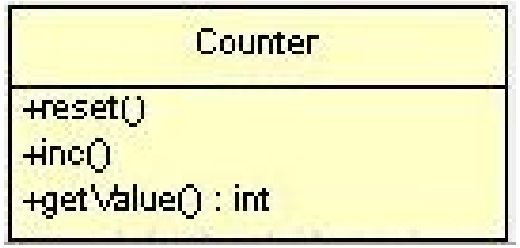
IL "CONTATORE" COME ENTITÀ STATICA SINGLETON

Lo stesso progetto di *contatore* può essere realizzato come *opportuno componente software* in Java, C#, Scala, Kotlin

- Lo stato (statico) sarà *protetto* dalla keyword **`private`**
- L'accesso allo stato sarà possibile solo attraverso le *funzioni (operazioni)* **`public`**, *anch'esse statiche*:

```
void reset();
void inc();
int getValue();
```

NB: niente **`void`** nella lista argomenti



- Per USARE il contatore, basterà:
  - compilare il componente e tenerlo nella stessa directory del cliente
  - senza necessità di includere alcuno header nel file client

IL "CONTATORE" COME ENTITÀ STATICA SINGLETON

- In Java e C#, il componente assume la forma di *classe* (parola chiave **`class`**) con soli membri statici
- In Scala e Kotlin, è previsto un apposito costrutto linguistico introdotto dalla parola chiave **`object`**
  - compilato «sotto banco» in una classe con membri statici ☺

`Contatore.java (o .cs)`

```
public class Contatore {

  ...

}
```

Java C#

`Contatore.scala (o .kt)`

```
public object Contatore {

  ...

}
```

NB: in Scala non c'è la keyword public (è la visibilità di default)

Scala Kotlin

UNA POSSIBILE REALIZZAZIONE in Java o C#

Contatore.java (o .cs)

```
public class Contatore {
   private static int stato;
   public static void reset(){stato=0;}
   public static void inc(){stato++;}
   public static int getValue(){return stato;}
}
```

Lo stato è *privato* e perciò inaccessibile dall'esterno della classe Contatore

Java

C#

Le signature delle funzioni sono *pubbliche* → visibili da fuori

MA il corpo delle funzioni resta comunque inaccessibile dall'esterno

- il livello di protezione (privato/ pubblico) è espresso esplicitamente
- ora è un *costrutto linguistico* (la classe) a racchiudere in un'unica entità lo stato del contatore e le operazioni che agiscono su esso, *non più solo un contenitore fisico (il file)* → SALTO DI QUALITÀ ESPRESSIVA
- è tutto statico, perché si tratta di un componente software che deve esistere per tutto il tempo di vita del programma.

IL CORRISPONDENTE CLIENTE

MyMain.java (o .cs)

```
public class MyMain {
  public static void main(String[] args) {
    Contatore.reset(); Contatore.inc();
    System.out.println(Contatore.getValue()==1);
  }
}
```

C#: **string** (minuscolo)

C#: **System.Console.WriteLine**

Java

C#

NOTE:

- Non occorre creare esplicitamente il contatore, perché esso coincide con la classe Contatore: è un'entità statica, *caricata al bisogno*
- Le operazioni non hanno più l'argomento perché è implicito su quale contatore agiscono – l'unico possibile, cioè Contatore medesima
- Si usa la "notazione puntata" per invocare funzioni pubbliche (METODI) del componente software statico Contatore

VARIANTE: COLLAUDO CON ASSERZIONE INLINE

MyMain.java (o .cs)

```
public class MyMain {
  public static void main(String[] args) {
    Contatore.reset(); Contatore.inc();
    assert Contatore.getValue()==1;
    System.out.println(true);
  }
}
```

Java

C#

C#: analogo ma con parentesi
`System.Diagnostics.Debug.Assert(...);`

NOTE:

- l'asserzione è eseguita solo se esplicitamente richiesto: è codice di debug che non servirà nella versione finale, non deve appesantire
- Java: compilare normalmente, ma eseguire con **`-ea`**
- C#: compilare con **`/D:DEBUG`** poi eseguire normalmente
- *Se l'asserzione è falsa, il programma abortisce.*

VARIANTE: COLLAUDO CON ASSERZIONE INLINE

MyMain.java (o .cs)

```
public class MyMain {
  public static void main(String[] args) {
    Contatore.reset(); Contatore.inc();
    assert Contatore.getValue()==2;
    System.out.println(true);
  }
}
```

Java

C#

Asserzione falsa

Giustamente, l'esito negativo del collaudo *non* va solo a finire in una stampa: CAUSA UN VERO ERRORE!

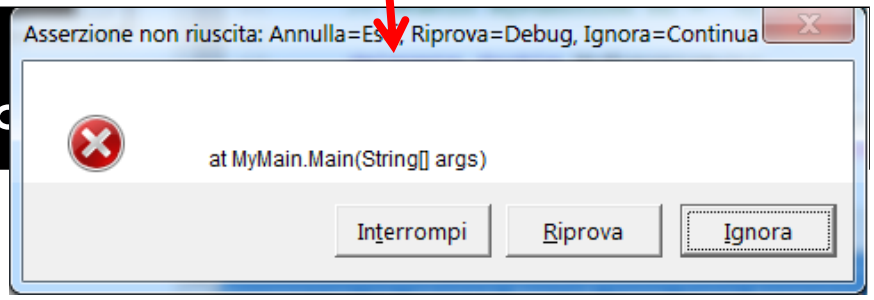
ESECUZIONE JAVA

```
C:> java -ea MyMain
```

Exception in thread "main" java.lang.AssertionError at MyMain.main(MyMain.java:4)

ESECUZIONE C#

```
C:> csc /D:DEBUG MyMain.cs Co
```



In base alla configurazione del vostro .NET Framework, può essere necessario aggiungere **/reference:System.dll**

ULTERIORE VARIANTE: COLLAUDO ESTERNO

MyMain.java (o .cs)

```
public class MyMain {
  public static void main(String[] args) {
    Contatore.reset(); Contatore.inc();
    System.out.println(Contatore.getValue())
  }
}
```

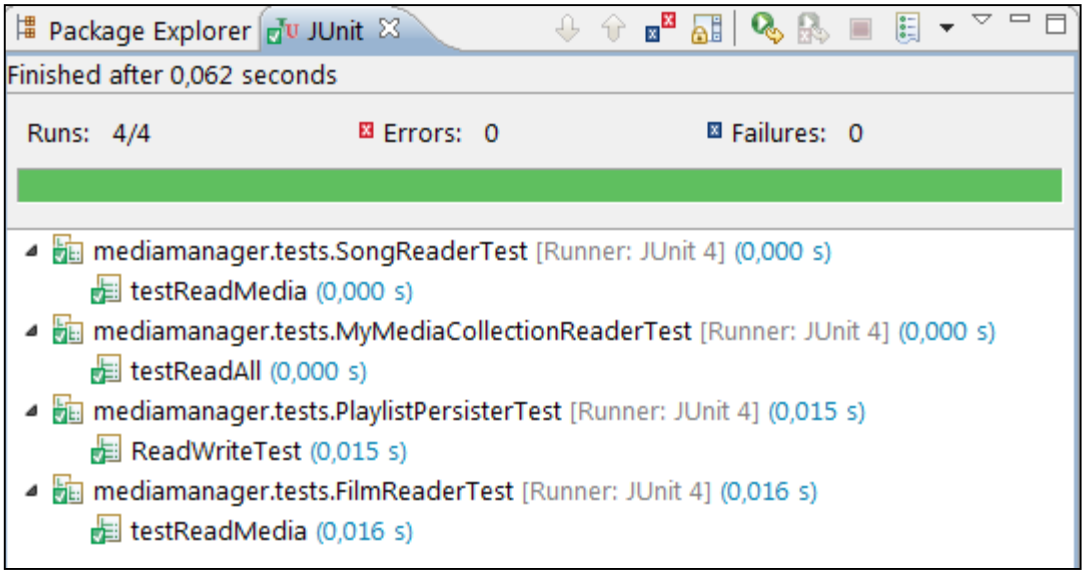
Niente più codice di collaudo in mezzo alla "business logic"

Java

C#

- Il codice di collaudo viene spostato in una *SUITE DI TEST* che diventa *parte integrante del progetto:* in effetti, lo convalida!
  - anche in sede d'esame il vostro progetto sarà validato così: *i test saranno parte dello "start kit" che vi daremo – max trasparenza!*
  - Voi stessi sarete *i primi collaudatori del vostro lavoro:* saprete se soddisfa le specifiche – anche *durante* il compito – e quindi saprete se consegnare e cosa state consegnando (niente alibi.. ☺)

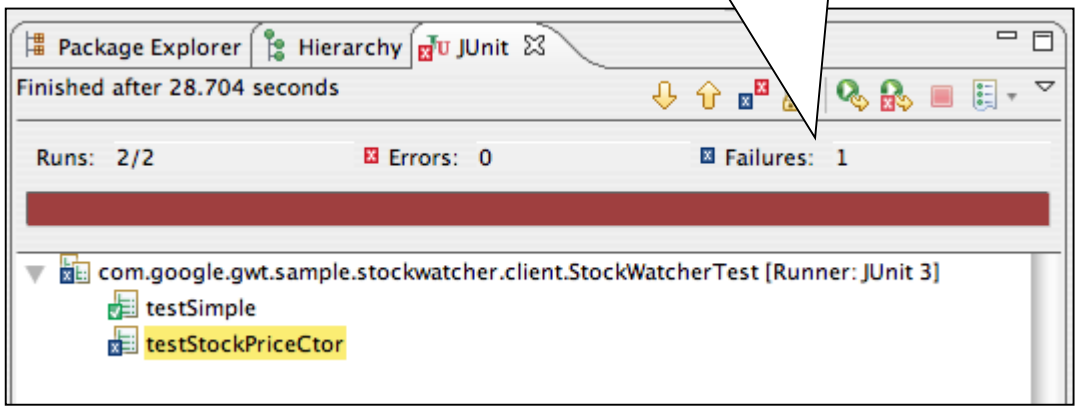
COLLAUDO ESTERNO: LO STRUMENTO JUnit



Esempio di esecuzione di un piano di collaudo in JUnit: test positivi

Java

Controesempio: 2 test, di cui 1 fallito



A failure is when one of your assertions fails.

An error is when some other exception occurs, one you haven't tested for and didn't expect.

COMPILAZIONE

COMPILAZIONE C

```
C:> cc mcounter.c mymain.c
```
produce mymain.exe

COMPILAZIONE JAVA

```
C:> javac Contatore.java
```
produce Contatore.class

```
C:> javac MyMain.java
```
produce MyMain.class

se si vuole un JAR eseguibile:

[predisporre il file INFO con la specifica di dove sia il main]

```
C:> jar cmf INFO MyMain.jar *.class
```
produce il JAR

COMPILAZIONE C#

```
C:> csc /target:library Contatore.cs
```
produce Contatore.dll

```
C:> csc /reference:Contatore.dll MyMain.cs
```
produce l'exe

in alternativa:

```
C:> csc Contatore.cs MyMain.cs
```
produce un EXE unico

Questo secondo EXE è un unico assembly contenente le due classi (può essere comodo specificare un nome diverso con l'opzione **`/out`**)

ESECUZIONE

```
C:> mymain
ok
```

Nel primo caso (due .class separati)

```
C:> java MyMain
true
```

Nel secondo caso (JAR eseguibile)

```
C:> java -jar MyMain.jar
true
```

Il bytecode **`Contatore.class`** dev'essere nella stessa cartella

Nel primo caso (exe+dll separati)

```
C:> MyMain
true
```

Nel secondo caso (exe unico)

```
C:> MyMain
true
```

La libreria **`Contatore.dll`** dev'essere nella stessa cartella

ULTERIORE EVOLUZIONE: SCALA e KOTLIN

- Scala e Kotlin fanno un passo in più: gli *oggetti statici sigleton* vengono etichettati **`object`** anziché **`class`**
  - la keyword diversa sottolinea che si tratta di entità singole
- Inoltre, sparisce la keyword **`static`**, ritenuta obsoleta
  - i singleton object sono *intrinsecamente* sempre statici
- Cambia anche la sintassi
  - per definire funzioni → keyword **`fun`** (kotlin) / **`def`** (scala)
  - per definire variabili e valori → keyword **`var`** & **`val`**
- La specifica di tipo diventa *postfissa* (type inference)

IL CONTATORE SINGLETON
da JAVA a SCALA e KOTLIN

```
public class Contatore {
   private static int stato;
   public static void reset(){stato=0;}
   public static void inc(){stato++;}
   public static int getValue(){return stato;}
}
```

Java

```
object Contatore {
    private var stato : Int = 0;
    def reset() {stato=0;}
    def inc() {stato += 1;}
    def getValue() : Int = {return stato;}
}
```

Specifica di tipo *postfissa*

Inizializzazione obbligatoria

L'operatore ++ non esiste

Scala

```
public object Contat
    private var stato : Int = 0;
    public fun reset() {stato=0;}
    public fun inc() {stato++;}
    public fun getValue() : Int {return stato;}
}
```

Specifica di tipo *postfissa*

Inizializzazione obbligatoria

Kotlin

COLLEGAMENTO STATICO vs. DINAMICO

In C:

- ogni sorgente include dichiarazioni
- si compila ogni file sorgente
- *si collegano (link) i file oggetto*
- si crea un *eseguibile* che non contiene più riferimenti esterni
- *max efficienza & max rigidità: ogni modifica implica il rebuild dell'intero eseguibile* e la sua redistribuzione

In tale schema:

- il compilatore accetta l'uso delle entità esterne "con riserva"
- il linker verifica la presenza delle definizioni risolvendo i *riferimenti incrociati* fra i file

In Java & co.:

- non esistono dichiarazioni
- si compilano le singole classi
- non si collegano staticamente le classi
- non esiste più l'eseguibile monolitico: l'applicazione è un pacchetto di classi → si esegue quella contenente il `main`
- max flessibilità: in caso di modifiche, basta ricompilare la classe modificata

In questo nuovo schema:

- il compilatore verifica subito il corretto uso delle altre classi (perché *sa dove trovarle nel file system*)
- le classi vengono *caricate e collegate* solo *al momento dell'uso*

UN PICCOLO TEST IN JAVA

Si supponga di disporre delle classi **`Point`** e **`Contatore`** (non importa cosa fanno) e si consideri il seguente main:

```
class TestClassLoading {
  public static void main(String[] a
   System.out.println( "inizio" );
   if (…) {
      System.out.println("caso if");
      Point.operation();
   }
   else {
      System.out.println("caso else");
      Contatore.reset();
   }
  }
}
```

Premesso che entrambe devono essere presenti a *compile-time*, se a *run time* un .class manca..

…e la condizione è vera, il JRE carica e usa la classe **`Point`**

L'eventuale mancanza di **`Contatore`** è irrilevante!

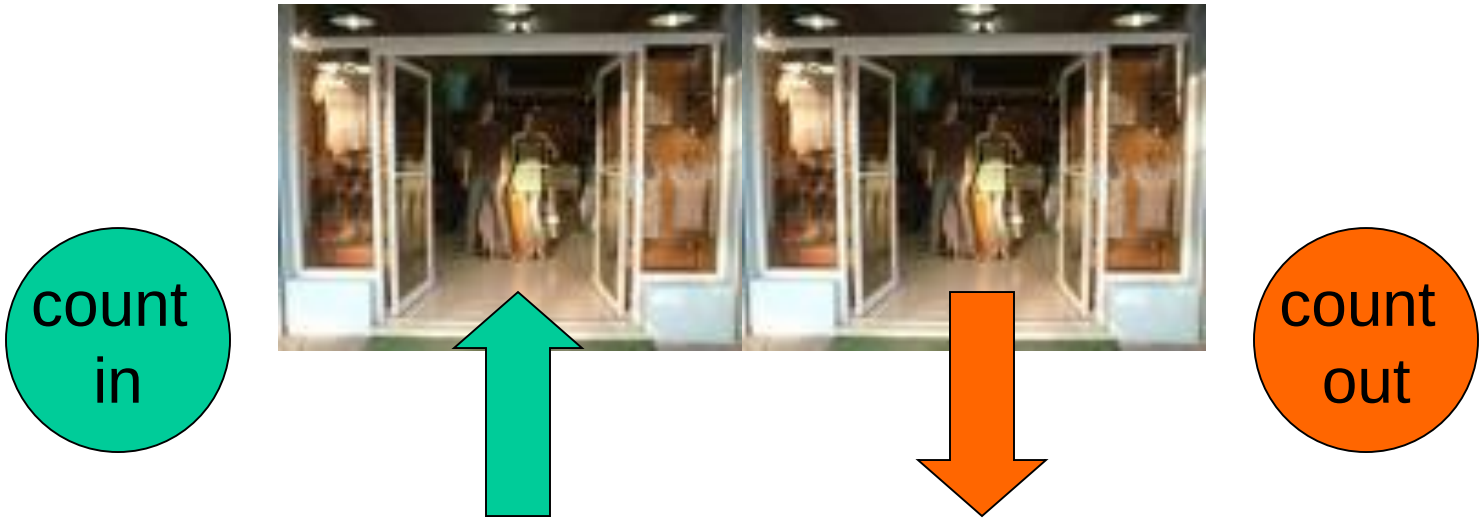
…e la condizione è falsa, il JRE carica e usa la classe **`Contatore`**: è l'eventuale mancanza di **`Point`**, stavolta, a essere irrilevante!

Limiti dei componenti software statici

UNO SCENARIO PROBLEMATICO (1/2)

Si supponga di voler *contare le persone in entrata* e le *persone in uscita* da un museo o da un grande magazzino

- A livello di principio, il problema è facile: *basta avere due contatori,* attivati ciascuno da un sensore sulla porta



UNO SCENARIO PROBLEMATICO (2/2)

Dunque, dovrebbe essere altrettanto semplice costruire il sistema software…

… MA il nostro contatore è un singleton: un pezzo unico!

- *NON possiamo averne due,* perché non possiamo includere due volte lo stesso componente nel progetto!
  - il C non consente che un modulo (file) sia incluso due volte
  - neanche in Java e C# ammettono che la stessa classe possa comparire due volte nell'applicazione

Occorre cambiare approccio.

UNO SCENARIO PROBLEMATICO (2/2)

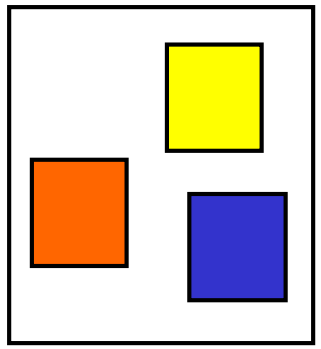
La soluzione *non* è il copia & incolla!

- copiare Contatore in Contatore2 duplicherebbe tutto,
- renderebbe *un incubo* la successiva manutenzione,
- e comunque *non è un approccio scalabile!*
  Se ne serve un altro che si fa, Contatore3.. ? E poi..?

Serve un approccio alternativo, intrinsecamente scalabile.

CLASSI COME COMPONENTI STATICI: LIMITI

- Come componenti software statici, le classi sono certamente un passo avanti rispetto ai "file" (moduli) del C
  - protezione (parte pubblica vs. parte privata)
  - costrutto linguistico che delimita il componente
  - collegamento e caricamento dinamici
- Tuttavia, sono componenti *statici singleton* e come tali *possono esistere* in un'applicazione solo *in copia unica*
  - una classe o fa parte di un'applicazione, o non ne fa parte
- Questo spesso non basta:
  - Va bene per LIBRERIE (singleton per definizione)
  - Va bene per entità «SINGLETON PER NATURA»
  - ma NON per componenti «GENERAL PURPOSE», che spesso nei sistemi servono in *copie multiple.*



VERSO COMPONENTI DINAMICI

Realisticamente, una applicazione non banale *non può essere fatta solo di componenti statici singleton*.

Servono *entità più dinamiche e flessibili*

- di cui si possano avere più copie…
- …che possano essere create durante l'esecuzione al momento del bisogno → *dinamicità*
- …sulla base di un «template» prestabilito → *tipo*

Approccio ADT revisionato e arricchito.

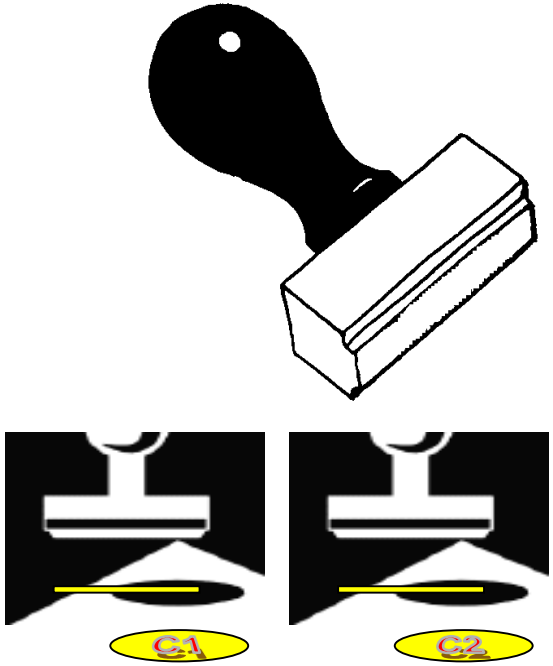
Classi come tipi
Oggetti come istanze di classi

IL CONCETTO DI OGGETTO

- Un OGGETTO è un componente software creato sulla base di un “modello” che ne definisce il *tipo*.
- Tale modello assomiglia a un *timbro*: *definisce le caratteristiche degli oggetti creati a sua immagine.*

Gli oggetti creati sulla base di un dato "timbro" (ossia, di un certo *tipo*)

- condividono:
  - la stessa *struttura interna*
  - le stesse *operazioni*
  - lo stesso *funzionamento*
- mantenendo però ciascuno la propria *identità.*



OGGETTI COME ISTANZE DI TIPI

- Un classico modo per procurarsi un "timbro" è *appunto quello di* definire un *tipo di dato*
- Così, facendo, gli oggetti diventano *istanze* di quel tipo
  - I linguaggi di programmazione che seguono questo approccio offrono costrutti per permettere di definire i propri tipi di dato…
  - … ma *L'ESPRESSIVITÀ PUÒ VARIARE MOLTO!*
    - alcuni linguaggi hanno costrutti per definire *solo campi-dati*, altri permettono di specificare *anche le operazioni* su tali dati
    - alcuni permettono di esprimere *forme di protezione* dei dati e/o delle operazioni, mentre altri non offrono tale possibilità
    - alcuni hanno un sistema di tipi "forte", altri più debole
    - alcuni permettono di esprimere anche *relazioni fra tipi*
    - alcuni permettono di definire *operatori* sui propri tipi
    - ecc ecc

OGGETTI COME ISTANZE DI TIPI

- Un classico modo per procurarsi un "timbro" è *appunto quello di* definire un *tipo di dato*
- Così, facendo, gli oggetti diventano *istanze* di quel tipo
  - I linguaggi di programmazione che seguono questo approccio offrono costrutti per permettere di definire i propri tipi di dato…
  - … ma *L'ESPRESSIVITÀ PUÒ VARIARE MOLTO*
    - alcuni linguaggi hanno costrutti per definire solo dati, altri permettono di definire anche le *operazioni* su tali dati
    - alcuni permettono di esprimere *forme di protezione* dei dati e/o delle operazioni, mentre altri non offrono tale possibilità
    - alcuni hanno un sistema di tipi "forte", altri più debole
    - alcuni permettono di esprimere anche *relazioni fra tipi*
      - alcuni permettono di definire *operatori* sui propri tipi
      - ecc ecc

C

Java & co

Java & co

C

Java & co

C

Java & co

C#, Scala,Kotlin

OGGETTI COME ISTANZE DI TIPI

- Si possono così *istanziare* tanti oggetti di quel *tipo* quanti ne occorrono
  - si risolve il problema dei componenti in copia unica
- Non basta: in un sistema di tipi "ben fatto", la *strut-tura interna degli oggetti* dovrebbe essere *protetta*
  - i clienti dovrebbero poter accedere agli oggetti *solo tramite le operazioni pubbliche fornite,* MAI tramite accesso diretto ai dati
  - sacri principi: ENCAPSULATION, INFORMATION HIDING

Che forma assume tutto questo
in C, Java, C#, Scala, Kotlin, … ?

Ripartiamo da ciò che conosciamo in C.

TIPI DI DATO ASTRATTO (ADT)

Un tipo di dato astratto (ADT) definisce una categoria concettuale con le sue proprietà:

- *definizione di tipo* su un dominio D
- *insieme di operazioni ammissibili* su tale dominio.

In C, gli ADT si definiscono tramite il costrutto **`typedef`**

- si possono creare tante entità di quel tipo quante ne servono
- tali entità sono espresse tramite *variabili* di quel tipo

Purtroppo, però, **`typedef`** non supporta l'incapsulamento

- la struttura interna dell'ADT, pur lasciata concettualmente sullo sfondo, è in realtà perfettamente visibile e nota a tutti perché *i file header vanno inclusi ovunque l'ADT venga usato*
- non vi è alcuna reale possibilità di *impedire usi errati* degli oggetti, perché l'accesso è *sostanzialmente libero!*

IL SOLITO ESEMPIO: CONTATORE COME ADT

- Per prima cosa si deve definire il tipo di dato astratto *"contatore"* tramite un'opportuna **`typedef`**:

```
typedef …… contatore;
```

- In questa fase *non importa come il contatore sia fatto,* poiché ciò è irrilevante per i clienti che lo useranno
  - però, poi, la **`typedef`** dovrà essere *inclusa da tutti i clienti* e quindi in realtà tutti sapranno come è fatto ☹
- Quindi si devono specificare le operazioni ammesse, con la relativa signature:

contatore.h

```
typedef ... contatore;
void reset(contatore*);
void inc(contatore*);
int  getValue(contatore);
```

IL CONTATORE COME ADT: USO

- Queste informazioni sono sufficienti per *usare* e *collaudare* il nuovo tipo **`contatore`**
  - per compilare il cliente basta includere l'header, che contiene la definizione del tipo (typedef) e le *dichiarazioni* di funzioni.
  - Il cliente può istanziare tanti contatori quanti gliene occorrono:

```
#include "contatore.h"
main() {
 int v1, v2;
 contatore c1, c2;
 reset(&c1); reset(&c2);
 inc(&c1); inc(&c1); inc(&c2);
 v1=getValue(c1);
 v2=getValue(c2);
}
```

```
typedef …… contatore;
void reset(contatore*);
void inc(contatore*);
int getValue(contatore);
```

PIANO DI COLLAUDO: dire a priori *cosa ci si aspetta come risultato* per **v1 e v2….**

UNA PRIMA IMPLEMENTAZIONE

- La struttura interna del contatore diventa *rilevante* quando giunge il momento di *realizzarlo.*
  - Il piano di collaudo, *stabilito a priori,* servirà per validarlo
- Se ora scegliamo di rappresentare lo stato con un intero, avremo:

```
typedef int contatore;
```

- Sotto questa ipotesi, la *definizione* delle operazioni precedentemente dichiarate assume la forma:

contatore1.c

```
#include "contatore.h"
void reset(contatore* pc) { *pc = 0;   }
void inc(contatore* pc)   { (*pc)++;   }
int  getValue(contatore c){ return c;  }
```

Supera il collaudo?

include typedef

UNA SECONDA IMPLEMENTAZIONE

- VARIANTE: *rappresentare lo stato con una struttura* che racchiuda un intero:

```
typedef struct {int value;} contatore;
```

- Perché? Perché molti tipi di uso corrente sono tipi strutturati e dunque seguono questo schema.
- Sotto questa ipotesi, la *definizione* delle operazioni precedentemente dichiarate assume la forma:

contatore2.c

```
#include "contatore.h"
void reset(contatore* pc){ pc -> value =0;
void inc(contatore* pc)  { (pc ->value)++;
int getValue(contatore c){ return c.value;}
```

Supera il collaudo?

include typedef

UNA TERZA IMPLEMENTAZIONE

- Come ulteriore alternativa, potremmo rappresentare lo stato con una stringa di '`I`' *(notazione del detenuto):*

```
typedef char contatore[21];
```

Qui "" indica 0, "`I`" indica 1, "`II`" indica 2, etc.

OK solo fino a 20

- Sotto questa ipotesi, la *definizione* delle operazioni precedentemente dichiarate assume la forma:

`contatore3.c`

Supera il collaudo?

```
#include "contatore.h"
void reset(contatore* pc){(*pc)[0] = '\0'; }
void inc(contatore* pc) {
  int x = strlen(*pc);(*pc)[x]='I'; (*pc)[x+1]='\0'; }
int getValue(contatore c){ return strlen(c); }
```

BILANCIO

Definire gli ADT in C tramite **`typedef`** è possibile, ma:

- non c'è unitarietà fra parte-dati (espressa da `typedef`) e parte-operazioni (scritte successivamente e altrove)
- non c'è protezione dall'uso improprio, perché tutti vedono `typedef` e dunque *tutti possono aggirarla*
- le signature delle operazioni fanno trasparire dettagli (puntatori…) che non si dovrebbero vedere a questo livello

Conclusione:

livello di espressività inadeguato

TIPI DI DATO NEI LINGUAGGI A OGGETTI

Nel linguaggi a oggetti come Java, C#, Scala, Kotlin:

- i tipi sono espressi tramite *classi* (senza **`static`**)
  - in Java e C#, tale costrutto era stato usato finora per definire componenti singleton, usando solo membri *statici*
  - ora, lo stesso costrutto è usato *in modo diverso (niente più membri statici)* per uno *scopo diverso*

Il costrutto **`class`**

- con keyword **`static`** (solo Java e C#) definisce un *componente software statico singleton*
- senza keyword **`static`** (in tutti i linguaggi: Java, C#, Scala, Kotlin) definisce un *tipo di dato*

CLASSI in JAVA e C#

Parte STATICA

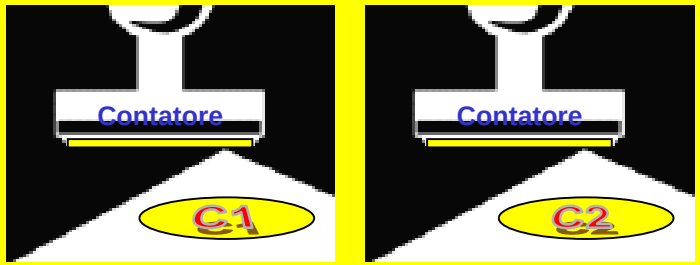
Definizione ADT

PARTE STATICA:

- Definisce un componente software
- *I dati e le operazioni della classe* intesa come componente software sono qualificati **`static`**

PARTE DI DEFINIZIONE DI TIPO:

- Definisce un tipo, un "timbro"
- usabile per *creare oggetti* fatti a immagine somiglianza del "timbro"



Spesso una classe Java ha *una sola* delle due parti, perché svolge *uno solo* di questi due ruoli.

Ma a volte fa comodo che li svolga entrambi…

CLASSI E OGGETTI SINGLETON in SCALA e KOTLIN

OGGETTI singleton

Keyword **`object`**

CLASSI come tipi

Keyword **`class`**

Scala e Kotlin adottano invece due costrutti distinti

- **`object`** (che non ha keyword **`static`**) definisce un *componente software statico singleton*
- **`class`** definisce sempre e solo un *tipo di dato*

L'ADT CONTATORE: dalla versione C…

Riconsideriamo una delle realizzazioni del contatore come ADT in C – in particolare, la seconda:

contatore.h

```
typedef struct {int value;} contatore;
void reset(contatore *pc);
void inc(contatore *pc);
int  getValue(contatore c);
```

contatore.c

```
#include "contatore.h"
void reset(contatore *pc){ pc -> value=0; }
void inc(contatore *pc) { (pc -> value)++;}
int  getValue(contatore c){return c.value;}
```

include typedef

L'ADT CONTATORE: …alla versione JAVA e C#

La **classe `Counter`** riunisce in sé ***tutta*** la definizione dell' ADT: definizione dei dati + implementazione dei metodi

contatore.h

```
typedef struct {int value;} contatore;
void reset(contatore *pc);
void inc(contatore *pc);
int  getValue(contatore c);
```

contatore.c

```
#include "contatore.h"
void reset(contatore *pc){ pc -> value=0; }
void inc(contatore *pc) { (pc -> value)++;}
int  getValue(contatore c){return c.value;}
```

Counter.java (o .cs)

```
public class Counter {
   private int value;
   public void reset()    { value = 0; }
   public void inc()      { value++;   }
   public int  getValue(){ return value; }
}
```

Quasi identica in Scala e Kotlin

L'ADT CONTATORE: …alla versione JAVA e C#

La **classe `Counter`** riunisce in sé ***tutta*** la definizione del-l' ADT: definizione dei dati + implementazione dei metodi

contatore.h

```
typedef struct {int value;} contatore;
void reset(contatore *pc);
void inc(contatore *pc);
int  getValue(contatore c);
```

Counter.java (o .cs)

```
public class Counter {
   private int value;
   public void reset()    { value = 0; }
   public void inc()      { value++;   }
   public int  getValue(){ return value;}
}
```

contatore.c

```
#include "contatore.h"
void reset(contatore *pc){ pc -> value=0; }
void inc(contatore *pc) { (pc -> value)++;}
int  getValue(contatore c){return c.value;}
```

L'ADT CONTATORE: …alla versione JAVA e C#

La **classe `Counter`** riunisce in sé ***tutta*** la definizione dell' ADT: definizione dei dati + implementazione dei metodi

contatore.h

```
typedef struct {int value;} contatore;
void reset(contatore *pc);
void inc(contatore *pc);
int  getValue(contatore c);
```

Counter.java (o .cs)

```
public class Counter {
   private int value;
   public void reset()    { value = 0; }
   public void inc()      { value++;   }
   public int  getValue() { return value; }
}
```

contatore.c

```
#include "contatore.h"
void reset(contatore *pc){ pc -> value=0; }
void inc(contatore *pc) { (pc -> value)++;}
int  getValue(contatore c){return c.value;}
```

Non occorre più passare l'argomento contatore esplicitamente ☺

L'ADT CONTATORE: …alla versione JAVA e C#

La **classe `Counter`** riunisce in sé ***tutta*** la definizione del-l' ADT: definizione dei dati + implementazione dei metodi

contatore.h

```
typedef struct {int value;} contatore;
void reset(contatore *pc);
void inc(contatore *pc);
int  getValue(contatore c);
```

Counter.java (o .cs)

```
public class Counter {
   private int value;
   public void reset()    { value = 0; }
   public void inc()      { value++;   }
   public int  getValue() { return value;}
}
```

contatore.c

```
#include "contatore.h"
void reset(contatore *pc){ pc -> value=0; }
void inc(contatore *pc) { (pc -> value)++;}
int  getValue(contatore c){return c.value;}
```

Si può usare direttamente il dato **`value`** definito sopra: *non occorre più passarlo come argomento!*

CLASSI COME ADT
NEI LINGUAGGI A OGGETTI

Si riuniscono quindi nell'unico costrutto *CLASSE*

- la definizione del tipo intesa come *dati* (ex **`typedef`** del C)
- le funzioni che su esso opereranno

specificando altresì il *livello di protezione* di ciascuna.

Counter.java (o .cs)

```
public class Counter {
  private int value;
  public void reset()    { value = 0; }
  public void inc()      { value++;   }
  public int  getValue(){ return value; }
}
```

Dati

Operazioni

*Utilizzo diretto del dato **`value`** nel corpo delle funzioni*
Elimina la necessità di passare l'argomento e relativi puntatori ☺

CLASSI COME ADT NEI LINGUAGGI A OGGETTI

NOTARE: non figura la keyword **`static`** !

Questa classe costituisce infatti una *definizione di tipo*, *non un componente software statico!*

Counter.java (o .cs)

```
public class Counter {
  private int value;
  public void reset()    { value = 0; }
  public void inc()      { value++;   }
  public int  getValue(){ return value; }
}
```

`value` è *privato* e come tale *accessibile solo alle operazioni definite nella classe*

Le signature delle funzioni sono *pubbliche* → accessibili da fuori

Il corpo delle funzioni è comunque inaccessibile da fuori

CLASSI COME ADT NEI LINGUAGGI A OGGETTI

```
public class Counter {
   private int value;
   public void reset()    { val = 0; }
   public void inc()      { val++;   }
   public int  getValue(){ return val; }
}
```

Java

```
class Counter {
   private var value : Int = 0;
   def reset() : Unit = { value = 0; }
   def inc()   : Unit = { value +=1; }
   def getValue(): Int = { return value; }
}
```

In Scala e Kotlin, il campo **value** va subito inizializzato

Scala

```
public class Counter {
   private var value : Int = 0;
   fun reset() : Unit { value = 0; }
   fun inc()   : Unit { value +=1; }
   fun getValue(): Int { return value; }
}
```

In Scala e Kotlin, il campo **value** va subito inizializzato

Kotlin

E ORA?

Per costruire un'applicazione, ci serve comunque un main

- esso deve esistere dall'inizio alla fine del programma
- in Java e C# ci serve una *classe con parte statica* in cui metterlo (in Scala e Kotlin sarà un *object*)

MyMain

Classe con solo parte statica (il main)

Cosa dovrà fare tale main ?

- presumibilmente, creare e usare contatori
- quindi, dev'esserci una *classe ADT* che definisca il tipo Counter

Counter

Classe con (sola) definizione di tipo

Già, ma.. *come si creano e si usano nuovi contatori?*

OGGETTI come ISTANZE DI CLASSI

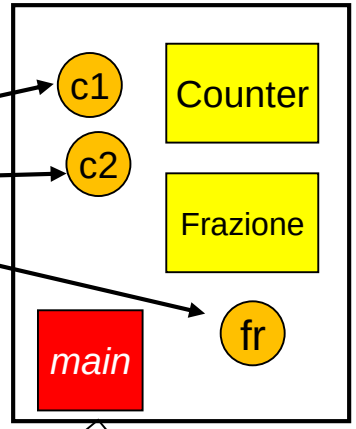
Gli *OGGETTI DINAMICI sono componenti software* creati a immagine e somiglianza di una *CLASSE ADT*

- si possono creare tutte le *istanze* che servono
- la creazione avviene al momento del bisogno, *durante l'esecuzione*

CLASSI
come ADT

CLASSI/object
Entità statiche

OGGETTI
(dinamici)



La allocazione dei nuovi oggetti avviene in memoria *heap*, come le strutture dati dinamiche del C

RIPRENDENDO UN ATTIMO IL CLIENTE DEL CONTATORE in C…

Questo main istanziava vari "contatori", ma lo faceva *senza far uso della memoria dinamica:*

```
#include "contatore.h"
main() {
  int v1, v2;
  contatore c1, c2;
  reset(&c1); reset(&c2);
  inc(&c1); inc(&c1); inc(&c2);
  v1=getValue(c1); v2=getValue(c2);
}
```

normali variabili C

VARIANTE CON MEMORIA DINAMICA

Questo invece istanzia "contatori" *facendo uso della memoria dinamica:*

```
#include "contatore.h"
main() {
  int v1, v2;
  contatore *c1, *c2;
  c1 = (…)malloc(sizeof(contatore)),
  c2 = (…)malloc(sizeof(contatore));
  reset(c1); reset(c2);
  inc(c1); inc(c1); inc(c2);
  v1=getValue(*c1); v2=getValue(*c2);
  free(c1); free(c2);
}
```

puntatori C

allocazione in memoria dinamica

deallocazione esplicita

OGGETTI DINAMICI COME ISTANZE DI CLASSI ADT

Gli *OGGETTI DINAMICI* sono creati *al bisogno* a immagine e somiglianza di una *CLASSE ADT* tramite l'operatore **`new`**

- agisce similmente alla **`malloc`** del C
- in Kotlin **`new`** è sottinteso, quindi la parola chiave è abrogata
- in Scala **`new`** è divenuta opzionale dalla versione 2.13

- Strutturalmente, ogni oggetto *è composto dai dati specificati dalla sua classe*
  - in modo simile alle variabili di tipo **`struct`** del C
- Funzionalmente, su ogni oggetto si possono *invocare le operazioni pubbliche* previste dalla sua classe

NOVITÀ: non occorre occuparsi della distruzione degli oggetti e della deallocazione della memoria, c'è il *garbage collector !*

UN MAIN CHE MANIPOLA CONTATORI in Java e C#

```
public class MyMain {
 public static void main(String[] args) {
     int v1, v2;
     Counter c1, c2;
     c1 = new Counter();
     c2 = new Counter();
     c1.reset(); c2.reset();
     c1.inc(); c1.inc(); c2.inc();
     v1 = c1.getValue(); v2 = c2.getValue();
     System.out.println(v1);
     System.out.println(v2);
 }
}
```

C#: `Main`

C#: `string`

Java

~C#

RIFERIMENTI anziché puntatori

Creazione oggetti in memoria dinamica

C#: adattare

Deallocazione di memoria automatica, a cura del *garbage collector*

LO STESSO MAIN in Scala e Kotlin

```
object MyMain {
  def main(args: Array[String]) {
    var c1 : Counter = new Counter();
    var c2 : Counter = new Counter();
    c1.reset(); c2.reset();
    c1.inc(); c1.inc(); c2.inc();
    var v1 : Int = c1.getValue();
    var v2 : Int = c2.getValue();
    println(v1); println(v2);
  }
}
```

Scala

~Kotlin

In Scala e Kotlin, le dichiarazioni multiple non sono permesse

Inoltre, è anche richiesto di *inizializzare subito* variabili e riferimenti

Kotlin: minimi adattamenti

- **`def` → `fun`**
- eliminare l'operatore **`new`**
- evitare **`object`** esterno

Type inference: volendo, si può evitare di specificare **`:Int`** nella definizione di **`v1`** e **`v2`** perché ciò può essere dedotto dal compilatore, dato il tipo di ritorno di **`getValue`**.

Da Scala 2.13, la keyword **`new`** è opzionale e può essere tolta, come in Kotlin

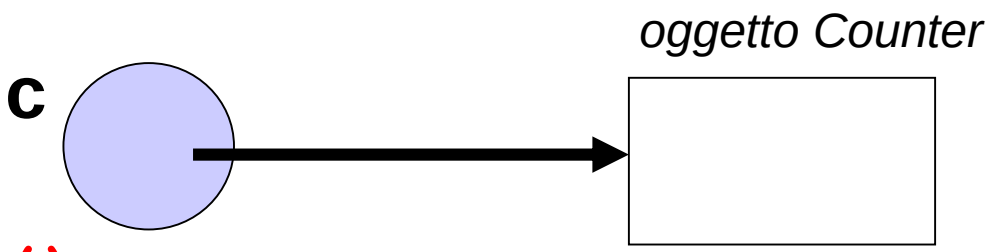
CREAZIONE DI OGGETTI DINAMICI

Per creare un oggetto dinamico:

- prima si definisce un *riferimento,* il cui tipo è *il nome della classe che fa da modello*
- *poi si crea dinamicamente l'oggetto* tramite ***new***

Esempio:

```
Counter c;
```




```
c = new Counter();
```

La distruzione degli oggetti invece è automatica

- il *garbage collector* elimina gli oggetti non più referenziati
- si eliminano i rischi relativi all'errata deallocazione della memoria

Un nuovo esempio di ADT: le frazioni

FRAZIONI COME ADT

In matematica, una frazione è caratterizzata da

- una coppia di interi (n,d), solitamente scritta n/d (d≠0)
- una serie di operazioni ammissibili
  - "costruzione" di una frazione da due interi
    DUBBIO: "costruzione" significa inizializzazione..?
  - accesso a numeratore e denominatore
  - riduzione ai minimi termini
    DUBBIO: altera la frazione data o ne produce una nuova..?
  - test di uguaglianza (condizione di equivalenza)

Rappresentarla significa quindi definirne:

- la struttura (in C: **`typedef`**; in Java & co., una classe)
- le operazioni (in C: header file; in Java& co., la classe)

LE GRANDI QUESTIONI

Inizializzare, Creare, Costruire

- *Inizializzare:*
  dare valore iniziale *a un oggetto che già esiste*
- *Creare:*
  allocare memoria per un *nuovo oggetto*
- *Costruire:*
  creare l'oggetto + inizializzarlo

Questione chiave:
attribuzione delle responsabilità
CHI FA / PUÒ FARE COSA ?

Modificare vs. Costruire

- *Modificare:*
  l'operazione *altera irrevocabilmente* l'oggetto ricevuto
- *Costruire:*
  l'operazione costruisce un *nuovo oggetto modificato*

NEL CASO DELLE FRAZIONI…

Possibili scelte:

- costruzione di una frazione da due interi
  - opzione 1: il cliente crea (come nel Counter) e poi inizializza
  - opzione 2: il cliente *delega la creazione*, inizializza soltanto
  - opzione 3: il cliente crea, ma *delega l'inizializzazione*
- riduzione ai minimi termini
  - è una variante del caso precedente: la frazione ridotta è, di fatto, una nuova frazione (num e den diversi)
  - opzione 1: il cliente passa la frazione per riferimento e la funzione la altera irrevocabilmente, "riducendola" ai minimi termini
  - opzione 2: il cliente passa la frazione per valore, e la funzione costruisce e restituisce una *nuova frazione-risultato,* senza alterare quella ricevuta

ADT FRAZIONE in C

frazioni.h

```
typedef struct {int num, den;} frazione;
void init(frazione *f, int n, int d);
int getNum(frazione f);
int getDen(frazione f);
frazione minTerm(frazione f);
int equals(frazione f1, frazione f2);
```

OSSERVA: `int` è un boolean sotto falso nome

frazione.c

```
#include "frazioni.h"
void init(frazione* pf, int n, int d) {
  pf -> num = n; pf -> den = d;
}
...
```

include **typedef**

UN POSSIBILE CLIENTE IN C

Questo main istanzia frazioni *dinamicamente:*

```
#include "frazioni.h"
main() {
  frazione *f1, *f2, f3;
  f1 = (…)malloc(…); init(f1,3,4);
  f2 = (…)malloc(…); init(f2,6,8);
  f3 = minTerm(*f2);
  printf("%d/%d\n", getNum(*f1), getDen(*f1));
  printf("%d/%d\n", getNum(*f2), getDen(*f2));
  printf("%s",
         equals(*f1,*f2) ? "uguali":"diverse");
  free(f1); free(f2);
}
```

Puntatori

Inizializzazione delegata

Alternanza di casi con * e senza *

Necessaria deallocazione esplicita

FRAZIONI COME ADT IN JAVA

Usando una classe ADT al posto di **`typedef`**+header

- si ricostituisce l'unitarietà concettuale dell'ADT
  - dati (typedef) e operazioni su quei dati stanno *insieme*
- si rende uniforme l'interfaccia di accesso
  - i riferimenti sono gestiti automaticamente: niente più **`*`**
- l'infrastruttura garantisce protezione
  - la struttura interna del tipo non è accessibile fuori dalla classe
- si sfrutta la gestione automatica della memoria dinamica
  - operatore **`new`** al posto dell'obsoleta **`malloc`**
  - deallocazione automatica tramite *garbage collector*

ADT FRAZIONE in JAVA

Frazione.java

```
public class Frazione {
  private int num, den;
  public void init(int n, int d){
       num = n; den = d;
  }
  public int getNum() { return num; }
  public int getDen() { return den; }
  public boolean equals(Frazione f2){
       return … // condizione di equivalenza
  }
  public Frazione minTerm(){
       return … // la nuova frazione ridotta
  }
}
```

Java

~C#

Livelli di protezione

Inizializzazione

`int` diventa `boolean`

OSSERVA: tutte le funzioni hanno un argomento in meno (il primo è implicito)

UN POSSIBILE CLIENTE IN JAVA

Questo main istanzia frazioni *dinamicamente:*

```
public class MyMain{
 public static void main(String[] args) {
  Frazione f1, f2, f3;
  f1 = new Frazione(); f1.init(3,4);
  f2 = new Frazione(); f2.init(6,8);
  f3 = f2.minTerm();
  System.out.println(
     f1.getNum() + "/" + f1.getDen() );
  System.out.println(
     f2.getNum() + "/" + f2.getDen());
  System.out.println(
     f1.equals(f2) ? "uguali" : "diverse");
 }
}
```

Java

~C#

Riferimenti

`init` inizializza

Notazione uniforme, senza più *

Concatenazione stringhe con +

Come sarà `equals` ?

Deallocazione automatica ed implicita

UN ESPERIMENTO INTERATTIVO

In Jshell:

```
jshell> Frazione f = new Frazione()
f ==> Frazione@6500df86

jshell> f.init(3,4)

jshell> f.getNum() + "/" + f.getDen()
$19 ==> "3/4"
```

Java

FRAZIONI COME ADT IN JAVA

Si aprono ampi spazi di progetto:

- separazione fra metodi che *non modificano* (*get accessor*) o che invece *modificano* (*mutator*, *set accessor*) lo stato
- il primo argomento (l'oggetto stesso) è sempre *implicito*
  - è quello su cui viene invocato il metodo
  - la lista argomenti contiene solo oggetti «diversi dal primo»
- creazione delegata
  - operatore **`new`** al posto dell'obsoleta **`malloc`**
  - deallocazione automatica tramite *garbage collector*
  - ma la costruzione manuale (tramite init) è scomoda ed error-prone

Occorre semplificare il pattern di costruzione

CONFRONTO C vs. OOP: ADT

frazioni.h

```
typedef struct {int num, den;} frazione;
void init(frazione *f, int n, int d);
int getNum(frazione f);
int getDen(frazione f);
frazione minTerm(frazione f);
int equals(frazione f1, frazione f2);
```

Frazione.java

Java

~C#

```
public class Frazione {
  private int num, den;
  public void init(int n, int d){…}
  public int getNum() { return num; }
  public int getDen() { return den; }
  public boolean equals(Frazione f2){ condizione di equivalenza }
  public Frazione minTerm(){ return la nuova frazione ridotta }
}
```

OSSERVA: tutti i metodi hanno un argomento in meno (il primo è implicito)

CONFRONTO C vs. OOP: MAIN

```c
// MyMain.c
void main() {
   frazione *f1, *f2, f3;
   f1 = (…)malloc(…);
   init(f1,3,4);
   f2 = (…)malloc(…);
   init(f2,6,8);
   f3 = minTerm(*f2);
   printf("%d/%d\n",
    getNum(*f1),getDen(*f1));
   printf("%d/%d\n",
    getNum(*f2),getDen(*f2));
   printf("%s",
    equals(*f1,*f2) ?
        "uguali" : "diverse");
   free(f1); free(f2);
}
```

```
public class MyMain {
 public static void main(
   Frazione f1, f2, f3;
   f1 = new Frazione();
   f1.init(3,4);
   f2 = new Frazione();
   f2.init(6,8);
   f3 = f2.minTerm();
   System.out.println(
    f1.getNum() + "/" + f1.getDen());
   System.out.println(
    f2.getNum() + "/" + f2.getDen());
   System.out.println(
   f1.equals(f2) ?
    "uguali" : "diverse");
 }
}
```

Java

~C#

Notazione uniforme, senza più *

Concatena stringhe

`int` diventa `boolean`

Deallocazione automatica della memoria heap (garbage collector)
Niente più **`free`** !

CONFRONTO C vs. OOP: MAIN

```c
// MyMain.c
void main() {
   frazione *f1, *f2, f3;
   f1 = (…)malloc(…);
   init(f1,3,4);
   f2 = (…)mall
   init
   f3 =
   prin
    get
   printf("%d/%d\n",
    get
   prin
    equ

   free
}
```

```
public class MyMain {
 public static void main(
   Frazione f1, f2, f3;
   f1 = new Frazione();
   f1.init(3,4);
   f2 = new Frazione();

                          erm();
                          intln(
                          "/" + f1.getDen());
                          tem.out.println(
                          "/" + f2.getDen());
                          intln(

                          verse");
}
```

Java

~C#

TIPICO SCHEMA DI COSTRUZIONE OGGETTI

- prima si alloca memoria (creazione)
- poi si riempiono i campi (inizializzazione)

MA È INOPPORTUNO FARLO IN DUE TEMPI

- Rischio di dimenticare l'inizializzazione
- Sarebbe meglio unificare le due fasi
- In C: anziché **`malloc+init`**, unica funzione **`make`**

UNA DIVERSA FRAZIONE IN C

frazioni.h

```
typedef struct {int num, den;} frazione;
frazione* make(int n, int d);
int getNum(fra
int g
frazi
int e
```

IDEA: una funzione (make) che incapsuli i due passi

- Prima crea (malloc), poi inizializza (init)
- Restituisce un (puntatore a) un oggetto **completamente configurato → *costruisce***

..e in Java?

- Anche in Java & co. le due fasi si possono unificare, grazie al ***costruttore***
- in effetti, la sintassi **`new Frazione(…)`** sembra già predisposta per accettare argomenti…!

CONFRONTO C vs. OOP: MAIN

```c
// MyMain.c
void main() {
   frazione *f1, *f2, f3;
   f1 = make(3,4);
   f2 = make(6,8);
   f3 = r   erm(*f2);
   free(f1); free(f2);
}
```

Tipico schema di costruzione in C

- Si appoggia a una funzione **`make`** predisposta da noi
- Approccio intelligente, ma *basato sul fai-da-te*

```
public class MyMain {
 public static void main(
   Frazione f1, f2, f3;
   f1 = new Frazione(3,4);
   f2 = new Frazione(6,8);
   f3 = f  minTerm();
 }
}
```

Java

~C#

In Java, C# o Scala, il costruttore svolge lo stesso compito in modo standard

- la sintassi **`new Frazione(…)`** accetta argomenti
- Sarà il costruttore *definito da noi* a stabilire cosa farne.

CURIOSITÀ: in Kotlin, la sintassi concisa abolisce la keyword **`new`**
Si scrive direttamente **`f1 = Frazione(…)`**

CREAZIONE DI OGGETTI vs. COSTRUZIONE DI OGGETTI

Java

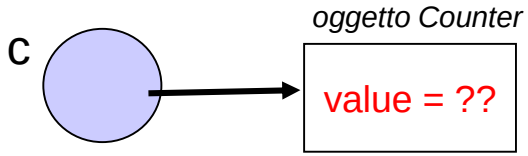
~C#

Finora, abbiamo CREATO oggetti:

- definendo prima un *riferimento* del tipo opportuno
- *creando poi l'oggetto* tramite l'operatore **`new`**

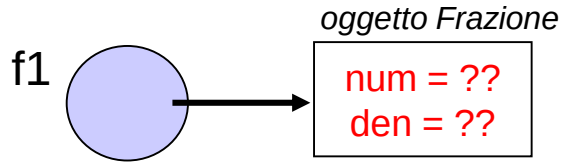
Esempio 1:

```
Counter c = new Counter();
```



Esempio 2:

```
Frazione f1 = new Frazione();
```



Tuttavia, gli oggetti così creati *non sono ancora inizializzati:* per dare loro un valore occorre invocare appositi metodi (**`reset`**, **`init`**)

COSTRUZIONE DI OGGETTI

- Più in generale, molti errori nel software sono causati tradizionalmente da *mancate inizializzazioni* di variabili.
- Per ovviare a questo problema, praticamente tutti i linguaggi a oggetti introducono il ***costruttore***:
  un metodo particolare che automatizza l'inizializzazione
  - non viene mai invocato *esplicitamente* dall'utente!
  - è invocato *automaticamente* ogni volta che si crea un nuovo oggetto di una data classe: non si può evitare, né dimenticare!
- Per questo, *COSTRUIRE è più che CREARE*
  - CREARE ricorda l'azione-base del "riservare memoria"
  - COSTRUIRE denota una attività più ampia, mirata a *«confezionare» un oggetto completo, pronto per l'uso.*

COSTRUTTORI in Java e C#

Java

~C#

In Java e C#, il costruttore:

- ha un *nome fisso*, *uguale al nome della classe*
- non ha tipo di ritorno, neppure **`void`**
  - il suo scopo non è “calcolare qualcosa”, ma *inizializzare* un oggetto
- può *non essere unico*
  - vi possono essere *più costruttori*, per inizializzare l’oggetto in *situazioni diverse*
  - tali costruttori si differenziano in base *alla lista dei parametri*
- esiste sempre: in mancanza di una definizione esplicita, il compilatore inserisce un *costruttore di default*
  - è quello che è scattato in automatico negli esempi precedenti
  - fa il minimo: inizializza le variabili numeriche a 0, i riferimenti a *null* e… invoca un "costruttore predefinito"

COSTRUZIONE DI DEFAULT

Java

~C#

Gli oggetti degli esempi precedenti sono stati:

- creati *esplicitamente* da noi tramite **`new`**
- costruiti *implicitamente* dal costruttore di defaut

```
Counter c = new Counter();
Frazione f1 = new Frazione();
```

- Il costruttore di default viene inserito dal compilatore *solo se la classe non prevede esplicitamente alcun costruttore*
- È comunque opportuno definirne sempre uno esplicitamente, non foss'altro per motivi di leggibilità e chiarezza

COSTRUTTORI PERSONALIZZATI

DIMENSIONE PROGETTUALE:
Quali e quanti costruttori prevedere? E perché?

- per il **`Counter`**, ha senso poter specificare all'atto della costruzione il *valore iniziale desiderato*
  → costruttore a un argomento
- per la **`Frazione`**, ha senso poter specificare all'atto della costruzione *numeratore e denominatore*
  → costruttore a due argomenti

COSTRUTTORE PRIMARIO: è il costruttore fondamentale che inizializza l'oggetto *nel caso più generale.*

+ COSTRUTTORI AUSILIARI per *casi particolari di uso frequente*

COSTRUTTORI PERSONALIZZATI

Costruttore primario (caso generale)

- per il **`Counter`**, costruttore a un argomento *dal valore iniziale desiderato*
- per la **`Frazione`**, costruttore a due argomenti a partire da *numeratore e denominatore*

Eventuali costruttori ausiliari (casi di uso frequente)

- per il **`Counter`**, costruttore di default *che inizi dal valore più usato* (es. 1 anziché 0)
- per la **`Frazione`**, costruttore a un argomento (il solo numeratore) per *esprimere i valori interi* (quindi, den=1)

Counter CON COSTRUTTORI

```
public class Counter {
   private int value;

   public Counter(){ value = 1; }

   public Counter(int v){ value = v; }

   public void reset() { value = 0; }
   public void inc()   { value++; }
   public int getValue() { return value;}
   public boolean equals(Counter that){..} ...
}
```

Costruttore ausiliario personalizzato

Costruttore primario a un argomento

Java

~C#

SCALA & KOTLIN: sintassi simile (ma non identica)

Si evidenzia il costruttore primario e se ne spostano gli argomenti a livello di dichiarazione della classe.

Frazione CON COSTRUTTORI

```
public class Frazione {
  private int num, den;
  public Frazione(int n, int d){
       num = n; den = d;
  }
  public Frazione(int n){
       num = n; den = 1;
  }
  ...
}
```

Java

~C#

Costruttore primario a due argomenti

Costruttore ausiliario a un solo argomento

Nessun costruttore di default a zero argomenti, perché non avrebbe senso! Non esiste una frazione "di default", più comune delle altre (o sì..?)

ESPERIMENTO INTERATTIVO

In Jshell, dopo aver modificato la classe (con /edit):

```
jshell> Frazione f  = new Frazione()
|  Error:
|  no suitable constructor found for Frazione(no arguments)
|      constructor Frazione.Frazione(int,int) is not applicable
|        (actual and formal argument lists differ in length)
|      constructor Frazione.Frazione(int) is not applicable
|        (actual and formal argument lists differ in length)
|  Frazione f  = new Frazione();
|                ^------------^

jshell> Frazione f  = new Frazione(5,6)
f ==> Frazione@71423665

jshell> f.getNum() + "/" + f.getDen()
$22 ==> "5/6"

jshell> Frazione f  = new Frazione(2)
f ==> Frazione@6fc6f14e

jshell> f.getNum() + "/" + f.getDen()
$24 ==> "2/1"
```

Java

OSSERVA: poiché non esiste alcun costruttore a zero argomenti (non avrebbe senso), il tentativo di creare una frazione «di default» viene *rigettato*

Counter: ESEMPIO DI MAIN

Questo main istanzia e produce contatori *dinamicamente:*

Java

~C#

```
public class Esempio4 {
 public static void main(String[] args){
   Counter c1 = new Counter();
   c1.inc();
   Counter c2 = new Counter(10);
   c2.inc();
   assert(c1.getValue()==2);  // collaudo
   assert(c2.getValue()==11); // collaudo
 }
}
```

Qui scatta il costruttore/0
→ **c1** inizializzato a 1

Qui scatta il costruttore/1
→ **c2** inizializzato a 10

Frazione: ESEMPIO DI MAIN

Questo main istanzia e produce frazioni *dinamicamente:*

Java

~C#

```
public class MyMain {
 public static void main(String[] args) {
  Frazione f1, f2, f3;
  f1 = new Frazione(3,4);
  f2 = new Frazione(6,8);
  f3 = f2.minTerm();
  System.out.println(…);
 }
}
```

Costruttori: il cliente crea, il costruttore inizializza

Frazione: ALL TOGETHER

Java

C#

```
public class Frazione {
  private int num, den;
  public Frazione(int n, int d){
       num = n; den = d;
  }
  public Frazione(int n){
       num = n; den = 1;
  }
  public int getNum() { return num; }
  public int getDen() { return den; }
  public boolean equals(Frazione f2){ condizione di equivalenza }
  public Frazione minTerm(){ return la nuova frazione ridotta }
}
```

Frazione: UN MAIN DI PROVA

Java

~C#

```
public static void main(String[] args) {
    Frazione f1, f2, f3;
    f1 = new Frazione(3,4);
    f2 = new Frazione(6,8);
    f3 = f2.minTerm();
    System.out.println(
       f1.getNum() + "/" + f1.getDen() );
    System.out.println(
       f2.getNum() + "/" + f2.getDen());
    System.out.println(
       f1.equals(f2) ? "uguali" : "diverse");
}
```

COSTRUTTORE PRIMARIO in Scala e Kotlin

- In Scala e Kotlin, il costruttore primario è conglobato *nell'intestazione stessa della classe*
  - sintassi che evidenzia l'essenziale ed evita duplicazioni
  - disponibili più varianti sintattiche per il trattamento degli argomenti

```
public class Frazione {
  private int num, den;
  public Frazione(int n, int d){ num = n; den = d; }
  public Frazione(int n){ num = n; den = 1; }
  …
}
```

Costruttore primario in Java

Java C#

```
public class Frazione(val num: Int, val den: Int) {
  ...
```

Costruttore primario Scala/Kotlin conglobato *nell'intestazione della classe*

Keyword assente in Scala

Gli argomenti possono essere *con o senza* val / var, con diversa semantica

Scala Kotlin

COSTRUTTORI AUSILIARI in Scala e Kotlin

- In Scala e Kotlin, i costruttori ausiliari sono spesso sostituiti da opportuni *valori di default* degli argomenti del primario
  - sintassi chiara e concisa che evita duplicazioni
  - (si possono comunque definire altri costruttori, vedremo come)

```
public class Frazione {
  private int num, den;
  public Frazione(int n, in                    }
  public Frazione(int n){ num = n; den = 1; }
  …
}
```

Costruttore ausiliario in Java

Java C#

```
public class Frazione(val num: Int, val den: Int = 1)
{
  ...
```

Keyword assente in Scala

Costruttore ausiliario sostituito da un opportuno *valore di default*

Scala Kotlin

COSTRUTTORI AUSILIARI in Scala e Kotlin

- In Scala e Kotlin, i parametri di classe
  - che diventano gli argomenti del costruttore primario
- possono essere etichettati in tre modi diversi:
  - nessuna etichetta → *nessun accessor generato* (variabile locale al costruttore)
  - etichetta **`val`** → *generato accessor in sola lettura* (campo pubblico ma read-only)
  - etichetta **`var`** → *generati accessor in lettura e scrittura* (campo pubblico read/write)

```
public class Frazione(val num: Int, val den: Int = 1){
    ...
}
```

Ad esempio, **qui `num` e `den` sono campi pubblici read-only**
Ergo, *non occorre più la coppia di accessor getNum/getDen*:
da fuori, si può accedere a **`num`** e **`den`** per nome

Frazione: ALL TOGETHER

Scala

```
class Frazione(val num:Int, val den:Int = 1) {
  // Niente più costruttori espliciti: sono generati
  // automaticamente dai parametri di classe
  // Niente più accessor: i parametri di classe val
  // sono pubblici in modalità read-only
  def equals(f:Frazione) : Boolean = {condizione di equivalenza }
  def minTerm() : Frazione = { return la nuova frazione ridotta }
}
```

Kotlin

```
public class Frazione(val num:Int, val den:Int = 1){
  // costruttori: idem come sopra
  // accessor: idem come sopra
   public fun equals(f:Frazione) : Boolean { cond. equiv. }
   public fun minTerm() : Frazione { return frazione ridotta }
}
```

Frazione: UN MAIN DI PROVA

Scala

```
def main(args: Array[String]) : Unit =  {
    val f1 = new Frazione(3,4);
    val f2 = new Frazione(6,8);
    val f3 = f2.minTerm();
    println( f1.num + "/" + f1.den );
    println( f2.num + "/" + f2.den );
    println( if f1.equals(f2) then "uguali"
                              else "diverse");
}
```

Accesso diretto (ma protetto!) ai campi pubblici read-only

Da Scala 2.13, l'operatore **new** è stato reso opzionale

Kotlin

```
fun main(args: Array<String>) : Unit {
    val f1 = Frazione(3,4);
    val f2 = Frazione(6,8);
    ...
}
```

Ricorda: in Kotlin è abolito l'operatore **new** (implicito)

COSTRUTTORI… PUBBLICI?

- Per poter istanziare oggetti di una certa classe, essa deve prevedere almeno un costruttore *pubblico*
  - in assenza di costruttori pubblici, è impossibile istanziare oggetti di quella classe
  - il costruttore di default ovviamente è pubblico
- *Costruttori non pubblici* hanno senso per *scopi particolari* in situazioni specifiche
  - di solito, *impedire la costruzione incontrollata* di oggetti
    - due "lunedì" ? tre mesi di "marzo"?
  - *l'ereditarietà* come vedremo fa spesso uso di costruttori "protetti"…

RECAP

Dichiarazioni di variabili e riferimenti nei diversi linguaggi

RECAP: DICHIARAZIONE VARIABILI

- In Java e C#, la sintassi è come in C:

  nometipo nomevariabile

  ```
  Frazione f1 = new Frazione(3,4);
  ```

- In Scala e Kotlin, la sintassi prevede la keyword **`var`** (o **`val`** se non modificabili) e la specifica di tipo postfissa:

  ```
  var f1 : Frazione = new Frazione(3,4); // Scala
  var f1 : Frazione = Frazione(3,4);     // Kotlin
  ```

  Inoltre, in Scala e Kotlin la specifica di tipo può essere omessa se è deducibile dal contesto:

  ```
  var f1 = new Frazione(3,4); // new opzionale in Scala 3
  var f1 = Frazione(3,4);     // new assente in Kotlin
  ```

RECAP: DICHIARAZIONE VARIABILI

- Per analogia, da Java 10 (2018) anche Java ha introdotto la keyword **`var`** per consentire di omettere l'indicazione di tipo quando esso può essere dedotto dal contesto:

```
var f1 = new Frazione(3,4);          // stile Java 10
```

- In C#, tale caratteristica è presente da C# 3.0 (2007)
- Pro & Contro:
  - Pro: sintassi più snella e meno verbosa
  - Contro: può rendere meno comprensibile il codice *se usata dove non è «evidente» il tipo degli oggetti* → non usare «a tappeto» solo «perché è nuova / di moda», ma sempre *con buon senso*
  - RICORDA: dopo tutto, Java nacque abbastanza «verboso» proprio per rimediare al C che era spesso fin troppo sintetico.. ☺

RECAP: DICHIARAZIONE VARIABILI

LO ZIO ENRICO CONSIGLIA:

- ok alla sintassi con **`var`** quando il tipo della variabile è comunque scritto *in esplicito* «poco lontano» :

```
Frazione f1 = new Frazione(3,4);  // stile classico
var f1 = new Frazione(3,4);       // OK !!
```

- pensarci due volte invece quando il tipo della variabile non è scontato o immediatamente deducibile:

```
var x = q;  // COSA DIAVOLO È x ? DI CHE TIPO ERA q?
```

In particolare…

RECAP: DICHIARAZIONE VARIABILI

LO ZIO ENRICO CONSIGLIA:

- …occhio alle costanti numeriche: possono causare facilmente fraintendimenti!

```
int x = 8;      // stile classico
var x = 8;      // OK, è un buon sostituto

long z = 77;    // stile classico
var  z = 77;    // NO! così z è un int !!
```

```
jshell> var x = 8
x ==> 8

jshell> long z = 77
z ==> 77

jshell>  x = z
|  Error:
|  incompatible types: possible lossy conversion from long to int
|    x = z
|        ^
```

```
jshell> var z = 77
z ==> 77

jshell> x = z
x ==> 77
```

RECAP: DICHIARAZIONE VARIABILI

LO ZIO ENRICO CONSIGLIA:

- …occhio alle costanti numeriche: possono causare facilmente fraintendimenti!
  - per i long si può ovviare con la specifica notazione nelle costanti, che prevede la L finale…

```
long z = 77L;   // stile classico
var  z = 77L;   // OK, così z è un long
```

```
jshell> var z = 77L
z ==> 77

jshell> x = z
|  Error:
|  incompatible types: possible lossy conversion from long to int
|  x = z
|      ^
```

RECAP: DICHIARAZIONE VARIABILI

LO ZIO ENRICO CONSIGLIA:

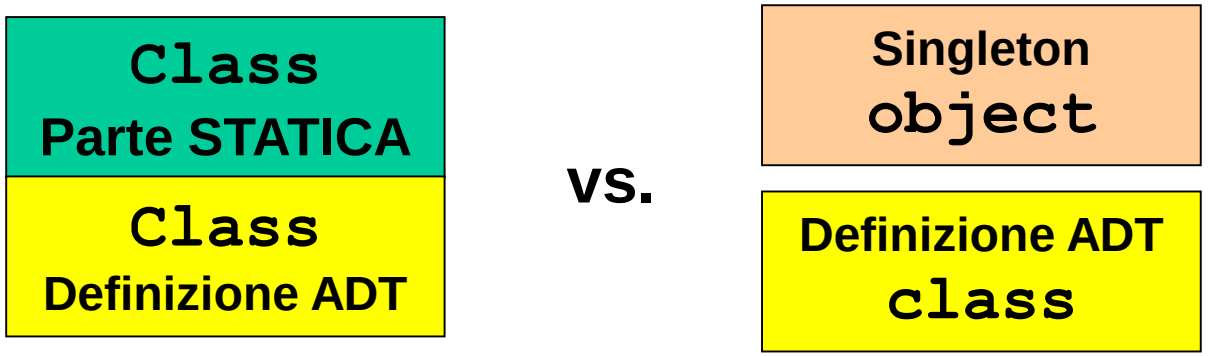
- …occhio alle costanti numeriche: possono causare facilmente fraintendimenti!
  - .. ma per short o byte non esiste una sintassi specifica
    e se usiamo il cast, abbiamo perso tutto il vantaggio!

```
jshell> var z = (short) 77
z ==> 77

jshell> short s = z;
s ==> 77

jshell> short s = x;
|  Error:
|  incompatible types: possible lossy conversion from int to short
|  short s = x;
|            ^
|
```

RECAP

Componenti statici vs. ADT

UN COSTRUTTO O MEGLIO DUE ?



Un costrutto unico (Java, C#) o due distinti (Scala, Kotlin)?

Il costrutto unico:

- svolge efficacemente *entrambi i ruoli,* come richiesto da alcuni design pattern di Ingegneria del software
- facilita il design di componenti «misti»
- offre una *transizione più facile* dal C

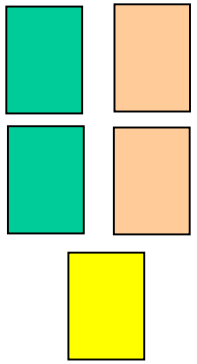
Due costrutti distinti:

- *distinguono meglio* i due ruoli, come opportuno nella gran parte dei casi
- eliminano la keyword «`static`» che richiama questioni antiche di basso livello (gestione memoria) anziché focalizzare sugli aspetti ingegneristici

COSA SCEGLIERE?

Riprendiamo la categorizzazione iniziale:

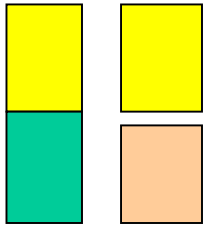
- librerie (componenti statici *senza stato*)
- moduli come singleton (con stato)
- tipi di dato (ADT)



A volte però la distinzione non è così netta:

- accade spesso di dover definire tipi di dato

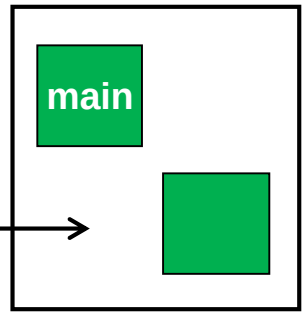
  \+ operazioni accessorie *su ALTRI tipi (correlati)*
  - frazioni + operazioni su *array di frazioni*
  - stringhe + operazioni di *conversione da numero a stringa*



ALCUNE ARCHITETTURE DI ESEMPIO

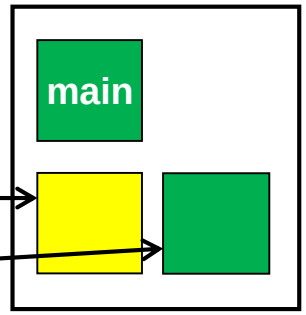
Scenario 1

- classe principale con il main
- *libreria per matrici*



Scenario 2

- classe principale con il main
- *tipo Frazione*
- *libreria per frazioni*



Scenario 3

- classe principale con il main
- *tipo Triango*lo [+ mini-main extra di test]