Alma Mater Studiorum-Università di Bologna
Scuola di Ingegneria

Classi e Oggetti

Corso di Laurea in Ingegneria Informatica

Anno accademico 2025/2026

Prof. ENRICO DENTI

Dipartimento di Informatica – Scienza e Ingegneria (DISI)

RIPRENDIAMO L'ESEMPIO "CONTATORE"

In C, ne abbiamo studiato due tipologie:

- come componente singleton (modulo dotato di stato)
- come Abstract Data Type (typedef + funzioni)

mcounter.h

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

mcounter.c

```
static int stato;

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

contatore.h

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

contatore.c

```
#include "contatore.h"

// sono possibili diverse realizzazioni,
// in dipendenza della scelta che si
// effettua in typedef
```

RIPRENDIAMO L'ESEMPIO "CONTATORE"

In C, ne abbiamo studiato due versioni:

- come componente singleton (modulo dotato di stato)
- come Abstract Data Type (typedef + funzioni)

mcounter.h

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

mcounter.c

```
static int stato;

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

Il componente singleton:

- è facile da usare, perché non richiede di definire alcuna variabile
- è protetto, perché il suo stato interno è inaccessibile
- ma non è adatto in tutte quelle situazioni in cui servano *più copie* del componente o comunque maggiore dinamicità

RIPRENDIAMO L'ESEMPIO "CONTATORE"

In C, ne abbiamo studiato due versioni:

- come componente singleton (modulo dotato di stato)
- come Abstract Data Type (typedef + funzioni)

mcounter.h

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

mcounter.c

```
static int stato;

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

Uso del componente singleton:

mymain.c

```
#include <stdio.h>
#include "mcounter.h"
int main() {
 reset(); inc(); int v = getValue();
 printf("Esito: %s", v==1 ? "ok" :
        "failed" );
 // oppure: assert(v==1);
 return 0;
}
```

RIPRENDIAMO L'ESEMPIO "CONTATORE"

In C, ne abbiamo studiato due versioni:

- come componente singleton (modulo dotato di stato)
- come Abstract Data Type (typedef + funzioni)

Al contrario, l'ADT:

- lascia al cliente l'onere di definire le variabili (oggetti) che gli servono, nonché la responsabilità di inizializzarle
- *non è protetto*, perché in realtà tutti vedono la typedef e quindi tutti sanno come è fatto e possono accedervi
- ma consente all'utente di definire anche *più copie* del componente, *eventualmente anche in modo dinamico*

contatore.h

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

contatore.c

```
#include "contatore.h"

// sono possibili diverse realizzazioni,
// in dipendenza della scelta che si
// effettua in typedef
```

RIPRENDIAMO L'ESEMPIO "CONTATORE"

In C, ne abbiamo studiato due versioni:

- come componente singleton (modulo dotato di stato)
- come Abstract Data Type (typedef + funzioni)

Un possibile uso dell'ADT:

mymain.c

```
#include "contatore.h"
int main() {
  int v1, v2;
  contatore c1, c2;
  reset(&c1); reset(&c2);
  inc(&c1); inc(&c1); inc(&c2);
  v1 = getValue(c1);
  v2 = getValue(c2);
  return 0;
}
```

contatore.h

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

contatore.c

```
#include "contatore.h"

// sono possibili diverse realizzazioni,
// in dipendenza della scelta che si
// effettua in typedef
```

RIPRENDIAMO L'ESEMPIO "CONTATORE"

In particolare, sviluppammo tre implementazioni:

```
#include "contatore.h"
```

```
typedef int contatore;
```

```
void reset(contatore* pc) { *pc = 0;   }
void inc(contatore* pc)   { (*pc)++;   }
int  getValue(contatore c){ return c;  }
```

```
#include "contatore.h"
```

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

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

```
#include "contatore.h"
```

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

```
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); }
```

Il caso della risorsa singleton

CHE FORMA ASSUMONO QUESTE REALIZZAZIONI IN Java & C# ?

- Il componente singleton (modulo dotato di stato) assume in Java e C# la forma di *classe con soli membri statici*

mcounter.h

C

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

mcounter.c

C

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

Contatore.java (o .cs)

Java

C#

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

CHE FORMA ASSUMONO QUESTE REALIZZAZIONI IN Java & C# ?

- Da notare che:
  - è *tutto statico* perché si tratta di un componente software che deve esistere per tutto il tempo di vita del programma
  - il livello di protezione (privato/pubblico) è espresso *esplicitamente*
  - ora è un *costrutto linguistico* a *racchiudere in un'unica entità* lo *stato* del contatore e le *operazioni* che agiscono su esso (non più solo un contenitore fisico come il file) → *salto di qualità espressiva*

Contatore.java (o .cs)

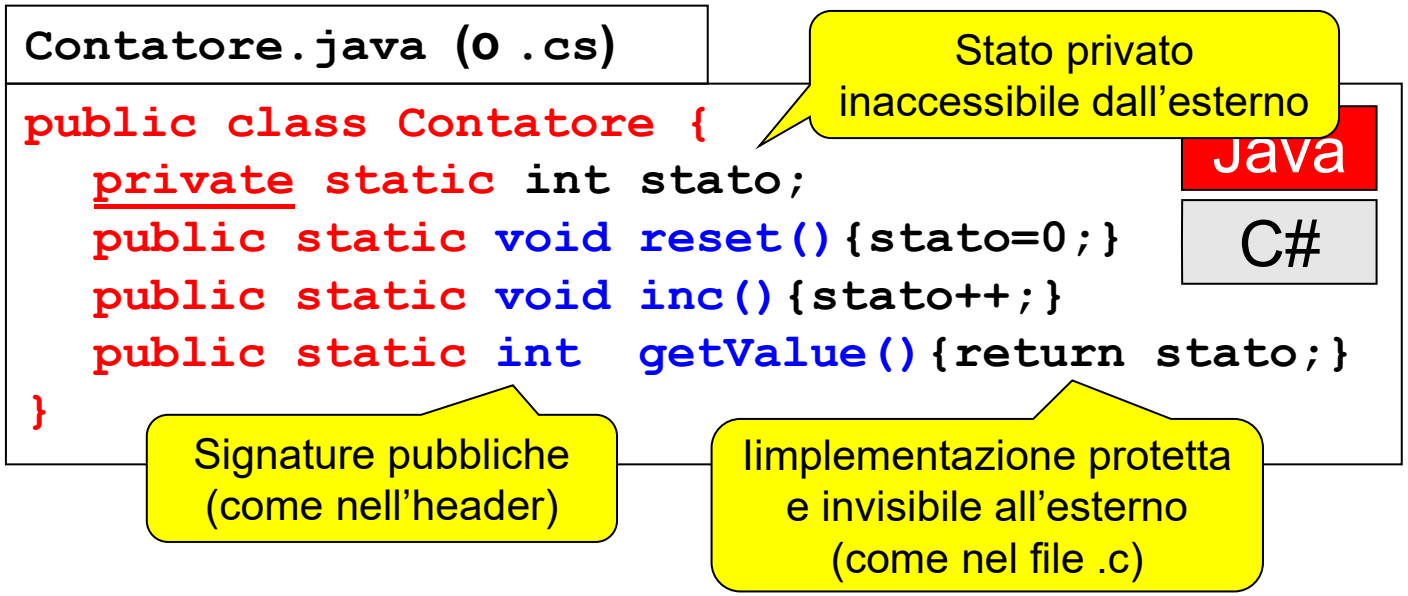
```
public class Contatore {
   private static int stato;
   public static void reset(){stato=0;}
   public static void inc(){stato++;}
   public static int  getValue(){return stato;}
}
```

Java

C#

CHE FORMA ASSUMONO QUESTE REALIZZAZIONI IN Java & C# ?

- In particolare:
  - lo stato è privato e perciò *inaccessibile dall'esterno* della classe
  - le *signature* delle funzioni sono pubbliche, quindi *visibili* da fuori
  - MA il *corpo* delle funzioni resta comunque *inaccessibile* da fuori



CHE FORMA ASSUMONO QUESTE REALIZZAZIONI IN Java & C# ?

- In realtà, per naming convention:
  - in Java i nomi delle funzioni devono iniziare per *minuscola*
  - in C# i nomi delle funzioni devono iniziare per *maiuscola*

Contatore.java

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

Iniziali minuscole

Java

Contatore.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;}
}
```

Iniziali maiuscole

C#

DAL PROGETTO AL CODICE.. AUTOMATICAMENTE

- Oggi, codice come questo può anche essere generato automaticamente da strumenti di AI, *a patto di formulare la giusta domanda*

Gemini

scrivi una classe java Contatore che rappresenti un contatore statico, dotato di stato, chiamato "stato". Deve avere metodi per incrementare il valore (inc), azzerarlo (reset) e recuperarlo (getValue)

Classe Contatore con stato statico denominato "stato"

Java

```java
public class Contatore {

    private static int stato; // Valore del contatore

    public static void inc() {
        stato++;
    }

    public static void reset() {
        stato = 0;
    }

    public static int getValue() {
        return stato;
    }
}
```

Classe Contatore in C#

C#

```csharp
public static class Contatore
{
    private static int stato; // Valore dello stato

    public static void Inc()
    {
        stato++;
    }

    public static void Reset()
    {
        stato = 0;
    }

    public static int GetValue()
    {
        return stato;
    }
}
```

In C# Si può qualificare static anche l'intera classe, oltre ai singoli membri

COME SI USANO?

- È semplicissimo: non occorre alcuna *#include* o direttiva simile, perché nei nuovi linguaggi il *link è dinamico*
- Le funzioni *statiche* del **`Contatore`** si invocano semplicemente dal main *con il loro nome completo,* nella forma

`NomeClasse.nomeFunzione(argomenti)`

MyMain.java (o .cs)

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

C#: `Main`

C#: `string`

C#: `System.Console.WriteLine`

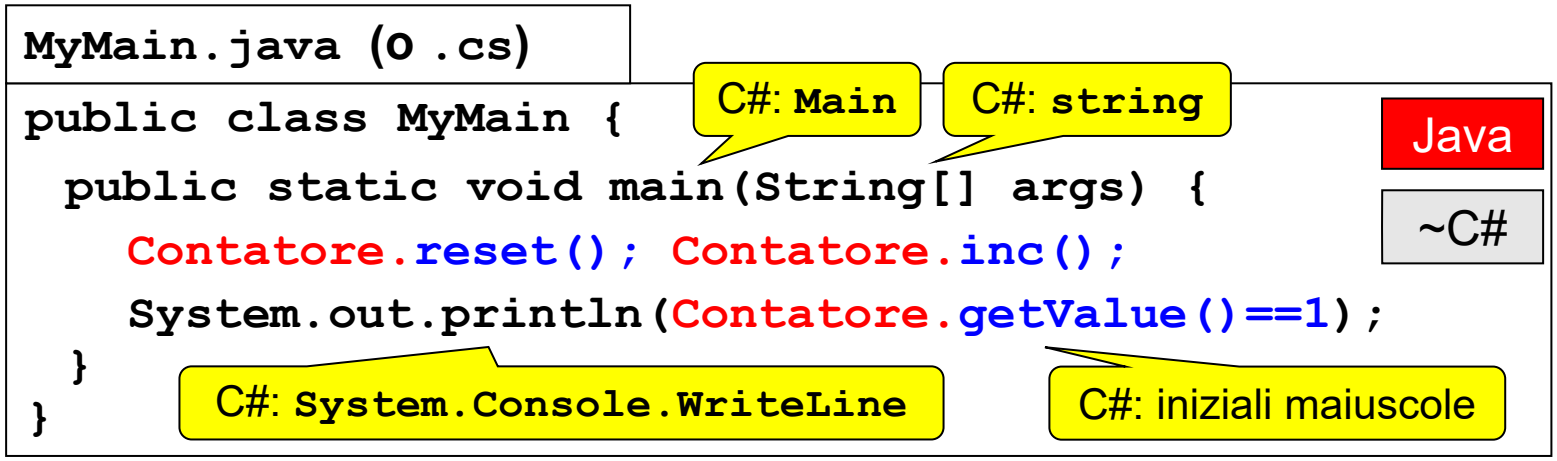
C#: iniziali maiuscole

Java

C#

COME SI USANO?

- OSSERVA:
  - *non occorre creare esplicitamente il contatore* perché esso coincide con la classe, che è un'entità *statica* e preesistente
  - non occorre alcuna *#include*: le classi vengono caricate *al momento del bisogno*, cercandole nel *classpath*



ANCHE QUI, L'AUTOMATIZZAZIONE…

mostrami un esempio d'uso in un main che usi questa classe: prima azzeri il contatore poi lo incrementi e infine verifichi che il valore sia 1, stampando il risultato di tale verifica

C#

```
C#

using System;

public class Program
{
    public static void Main(string[] args)
    {
        // Azzera il contatore
        Contatore.Reset();

        // Incrementa il contatore
        Contatore.Inc();

        // Verifica che il valore sia 1
        bool isValoreUno = Contatore.GetValue() == 1;

        // Stampa il risultato della verifica
        Console.WriteLine("Il valore è 1: " + isValoreUno);
    }
}
```

la stessa cosa in Java grazie

Java

```
public class Main {

    public static void main(String[] args) {
        // Azzera il contatore
        Contatore.reset();

        // Incrementa il contatore
        Contatore.inc();

        // Verifica che il valore sia 1
        boolean isValoreUno = Contatore.getValore() == 1;

        // Stampa il risultato della verifica
        System.out.println("Il valore è 1: " + isValoreUno);
    }
}
```

COMPILAZIONE

COMPILAZIONE C — C

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

COMPILAZIONE Java — 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#

```
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 MyMain.exe all-in-one
```

Quest'ultimo .exe è un unico assembly contenente le due classi (può essere comodo specificare un nome diverso con l'opzione **`/out`**)

ESECUZIONE

ESECUZIONE C

C

```
C:> mymain
ok
```

ESECUZIONE Java (i due `.class` sono nella stessa cartella)

Java

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

ESECUZIONE Java con JAR eseguibile:

```
C:> java –jar MyMain.jar
true
```

ESECUZIONE C# (caso exe+dll separati, ma posti nella stessa cartella)

```
C:> MyMain
true
```

ESECUZIONE C# (caso .exe unico)

```
C:> MyMain
true
```

C#

VARIANTE: COLLAUDO CON ASSERZIONE

- Possiamo utilmente sostituire la stampa con un'asserzione
  - a differenza della stampa, l'asserzione è codice di debug che non servirà nella versione finale e non deve appesantire
  - per questo, è eseguita *solo se esplicitamente richiesto:* se a runtime essa risulta falsa, *il programma abortisce con messaggi d'errore*
- Operativamente:
  - in Java, occorre compilare normalmente, ma eseguire poi con `-ea`
  - in C#, invece, occorre compilare con `/D:DEBUG` ed eseguire poi normalmente
- Sintassi:
  - in Java `assert condition;`
  - in C# `System.Diagnostics.Debug.Assert(condition);`

VARIANTE: COLLAUDO CON ASSERZIONE

- Main con asserzione

MyMain.java

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

Java

Asserzione vera

MyMain.cs

```
public class MyMain {
  public static void Main(string[] args) {
     Contatore.Reset(); Contatore.Inc();
     System.Diagnostics.Debug.Assert(
                  Contatore.GetValue()==1);
  }
}
```

C#

Asserzione vera

VARIANTE: COLLAUDO CON ASSERZIONE

- Main con asserzione: esecuzione
  - se l'asserzione è vera, il programma termina silenziosamente
  - se è *falsa*, il programma abortisce con messaggio d'errore

MyMain.java

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

Java

Asserzione falsa

```
C:> java -ea MyMain
Exception in thread "main" java.lang.AssertionError at MyMain.main(MyMain.java:4)
```

VARIANTE: COLLAUDO CON ASSERZIONE

- Main con asserzione: esecuzione
  - se l'asserzione è vera, il programma termina silenziosamente
  - se è *falsa*, il programma abortisce con messaggio d'errore

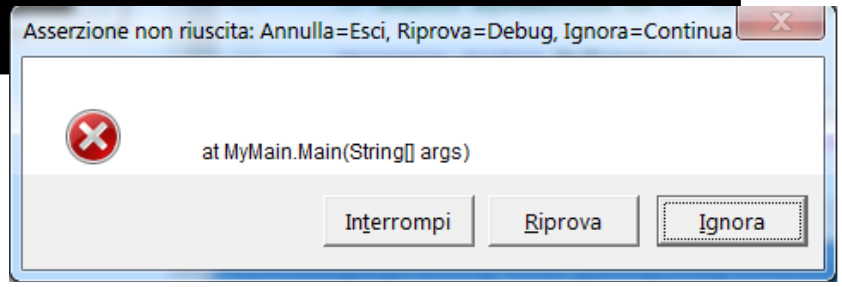
MyMain.cs

C#

```
public class MyMain {
  public static void Main(string[] args) {
     Contatore.Reset(); Contatore.Inc();
      System.Diagnostics.Debug.Assert(
                 Contatore.GetValue()==2);
  }
}
```

Asserzione falsa

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



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

GENERAZIONE AUTOMATICA

invece della stampa finale, inserisci un'asserzione

Java

```java
public class Main {

    public static void main(String[] args) {
        // Azzera il contatore
        Contatore.reset();

        // Incrementa il contatore
        Contatore.inc();

        // Verifica che il valore sia 1 usando un'asserzione
        assert Contatore.getValore() == 1 : "Il valore del contato

        // Stampa un messaggio di successo
        System.out.println("Il valore del contatore è 1");
    }
}
```

CHE FORMA ASSUMONO QUESTE REALIZZAZIONI IN Scala & 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 infatti *intrinsecamente* sempre statici
- Cambia anche la sintassi
  - per definire funzioni → keyword **`def`** (Scala) / **`fun`** (Kotlin)
  - per definire variabili e valori → keyword **`var`** & **`val`**
  - specifica di tipo *postfissa* (con *type inference* evoluta)

IL CONTATORE SINGLETON
da Java a Scala & Kotlin

Java / ~C#

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

Scala

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

Inizializzazione obbligatoria

L'operatore ++ non esiste

Kotlin

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

Inizializzazione obbligatoria

GENERAZIONE AUTOMATICA

 la classe precedente in Scala

Scala

```scala
object Contatore {

  private var stato: Int = 0 // Valore dello stato

  def inc(): Unit = {
    stato += 1
  }

  def reset(): Unit = {
    stato = 0
  }

  def getValore(): Int = {
    stato
  }
}
```

```scala
object Main {

  def main(args: Array[String]): Unit = {
    // Azzera il contatore
    Contatore.reset()

    // Incrementa il contatore
    Contatore.inc()

    // Verifica che il valore sia 1 usando un
    assert(Contatore.getValore() == 1, "Il va

    // Stampa un messaggio di successo
    println("Il valore del contatore è 1")
  }
}
```

 ora in Kotlin

Kotlin

```kotlin
object Contatore {

    private var stato: Int = 0 /

    fun inc() {
        stato++
    }

    fun reset() {
        stato = 0
    }

    fun getValore(): Int {
        return stato
    }
}
```

Il tipo di ritorno Unit può essere inferito…

```kotlin
fun main() {
    // Azzera il contatore
    Contatore.reset()

    // Incrementa il contatore
    Contatore.inc()

    // Verifica che il valore sia 1 usando
    require(Contatore.getValore() == 1) { "

    // Stampa un messaggio di successo
    println("Il valore del contatore è 1")
}
```

GENERAZIONE AUTOMATICA

la classe precedente in Scala

Scala

```scala
object Contatore {

  private var stato: Int = 0 // Valore dello stato

  def inc(): Unit = {
    stato += 1
  }

  def reset(): Unit = {
    stato = 0
  }

  def getValore(): Int = {
    stato
  }
}
```

```scala
object Main {

  def main(args: Array[String]): Unit = {
    // Azzera il contatore
    Contatore.reset()

    // Incrementa il contatore
    Contatore.inc()

    // Verifica che il valore sia 1 usando un
    assert(Contatore.getValore() == 1, "Il va

    // Stampa un messaggio di successo
    println("Il valore del contatore è 1")
  }
}
```

nella versione Kotlin, aggiungi il tipo di ritorno Unit nei due

Kotlin

Versione Kotlin con tipi di ritorno Unit

Kotlin

```kotlin
object Contatore {

    private var stato: Int = 0 // Valore dello stato

    fun inc(): Unit { // Aggiunto il tipo di ritorno Unit
        stato++
    }

    fun reset(): Unit {
        stato = 0
    }
```

..o può essere reso esplicito

```kotlin
fun main() {
    // Azzera il contatore
    Contatore.reset()

    // Incrementa il contatore
    Contatore.inc()

    // Verifica che il valore sia 1 usando
    require(Contatore.getValore() == 1) {

    // Stampa un messaggio di successo
    println("Il valore del contatore è 1")
}
```

USO DEL CONTATORE SINGLETON
da Java a Scala & Kotlin

Java

~C#

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

Scala

```
object MyMain { // sintassi classica, Scala 2
  def main(args: Array[String]) = {
     Contatore.reset(); Contatore.inc();
     println(Contatore.getValue()==1);
  }
}
```

`:Unit` inferito automaticamente

`;` a fine riga non indispensabile

Kotlin

```
// object implicito, file MyMain.kt
  public fun main(args: Array<String>) {
     Contatore.reset(); Contatore.inc();
     println(Contatore.getValue()==1);
  }
```

`:Unit` inferito automaticamente

`;` a fine riga non indispensabile

USO DEL CONTATORE SINGLETON in Java 25 e Scala 3

Java 25

```
// classe implicita, file MyMain.java
void main() {
    Contatore.reset(); Contatore.inc();
    IO.println(Contatore.getValue()==1);
}
```

Scala 3

```
@main // sintassi compatta Scala 3
  def MyMain() =
    Contatore.reset(); Contatore.inc();
    println(Contatore.getValue()==1);
```

; a fine riga non indispensabile

Collegamento statico vs. collegamento dinamico

DAL LINK STATICO DEL C AL LINK DINAMICO DI Java & co.

- In C, il linker collega *staticamente* tutte le librerie e produce a priori l'eseguibile auto-contenuto
- In Java & co., il link diventa dinamico:
  - non esiste più un eseguibile auto-contenuto
  - le classi vengono caricate *solo se servono*, dinamicamente, al momento del bisogno, cercandole nel *classpath*
- Conseguenze:
  - non occorrono più le direttive #include, né gli header del C
  - è possibile aggiornare una classe *senza dover rifare tutto il build:* semplicemente, la prossima volta verrà caricata la nuova classe ☺
  - se una classe serve sono in certi casi, *non sarà caricata negli altri*

LINK DINAMICO: ESPERIMENTO

- 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[] args){

   System.out.println( "inizio" );

   if (…) {
      System.out.println("caso if");
      Point.operation();
   }
   else {
      System.out.println("caso else");
      Contatore.reset();
   }
  }
}
```

OSSERVA:

- nel primo ramo dell' **`if`**, si usa solo la classe **`Point`**
- nel secondo ramo, si usa solo la classe **`Contatore`**

DUBBIO:
che impatto ha ciò su *compilazione* ed *esecuzione*?

LINK DINAMICO: ESPERIMENTO

- 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[] args){

   System.out.println( "inizio" );

   if (…) {
      System.out.println("caso if");
      Point.operation();
   }
   else {
      System.out.println("caso else");
      Contatore.reset();
   }
  }
}
```

In compilazione *tutte* le classi devono essere presenti, perché il compilatore deve verificare le chiamate.

In esecuzione, invece, basta che sia presente *la sola classe usata nel ramo eseguito:* l'altra non viene neppure caricata, perché non ce n'è bisogno!

RECAP: LINK 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*

Il caso dell'ADT

CHE FORMA ASSUMONO QUESTE REALIZZAZIONI IN Java & C# ?

- L'Abstract Data Type (in C, typedef + funzioni) assume la forma di *classe SENZA membri statici*
- Per mettere a fuoco il passaggio, riconsideriamo una delle realizzazioni del contatore come ADT in C

contatore.h

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

C

contatore.c

```
#include "contatore.h"

// possibili diverse realizzazioni
```

C

- In particolare, la seconda (con la **`struct`**):

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

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

C

L'ADT Contatore IN Java & 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;}
```

Quasi identica in Scala e Kotlin

L'ADT Contatore IN Java & 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;}
```

L'ADT Contatore IN Java & 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;}
```

OSSERVA: non occorre più passare l'argomento contatore esplicitamente ☺

L'ADT Contatore IN Java & 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;}
```

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 nel costrutto *classe:*
  - la definizione del tipo intesa come *nome* (ex nome della **`typedef`**), che è resa *pubblica*
  - la definizione del tipo intesa come *dati* (ex struttura della **`typedef`**), che è invece mantenuta *privata*
  - le dichiarazioni delle funzioni che su esso operano (ex header .h), che sono ovviamente *pubbliche*
  - le definizioni/implementazioni delle funzioni che su esso operano (ex file .c), che sono invece mantenute *private*
- Ogni elemento è dunque etichettato col *livello di protezione* appropriato al ruolo che svolge.

CLASSI COME ADT NEI LINGUAGGI A OGGETTI

- OSSERVA: non c'è nulla di statico!
  - è la prova che siamo di fronte alla definizione di un ADT, non di un singleton (pur utilizzando lo stesso costrutto linguistico *classe*)
- Inoltre, lo stato è protetto!
  - nessuno può accedere a ***value*** da fuori, violandone l'integrità! ☺

Counter.java (o **.cs**)

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

Java

~C#

CLASSI COME ADT NEI LINGUAGGI A OGGETTI

- Si semplifica di molto anche il passaggio degli argomenti:
  - l'adozione di un costrutto linguistico unitario rende infatti possibile *l'utilizzo diretto del dato* **`value`** *nel corpo delle funzioni*, eliminando la necessità di passare l'argomento e i relativi puntatori ☺
  - il dato **`value`** è visibile solo all'interno del blocco {…} della classe, quindi solo le funzioni della classe possono accedervi

`Counter.java` (o **`.cs`**)

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

Java

~C#

L'ADT Contatore da Java a Scala & Kotlin

- Anche Scala e Kotlin introducono il costrutto **`class`** per definire ADT
  - però, Java & C# lo usano anche per i singleton
  - Scala e Kotlin, invece, come sappiamo, per i singleton usano la keyword **`object`** (che assorbe anche **`static`**)
- Le differenze sintattiche sono minime
  - sia in Scala che in Kotlin, è obbligatorio inizializzare i dati già in fase di dichiarazione (vedremo i diversi modi per farlo)
  - in Scala non esiste il qualificatore **`public`**, perché è il default
  - in Scala 3, la sintassi compatta permette di eliminare molte { }

CLASSI COME ADT NEI LINGUAGGI A OGGETTI

Java / ~C#

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

Scala

```
class Counter { // sintassi classica
   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

Kotlin

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

CLASSI COME ADT NEI LINGUAGGI A OGGETTI

Java

~C#

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

Scala 3

```
class Counter:
   private var value : Int = 0
   def reset() = value = 0
   def inc()   = value +=1
   def getValue() = value
```

Sintassi senza { }
Le indentazioni sono significative

Senza { } si elimina anche la keyword **return**

Kotlin

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

GENERAZIONE AUTOMATICA

scrivi una classe Java Counter che rappresenti un contatore, dotato di stato, chiamato "value". Deve avere metodi per incrementare il valore (inc), azzerarlo (reset) e recuperarlo (getValue)

Java

```java
public class Counter {

    private int value; // Valore dello stato

    public
        th
    }

    public void inc() {
        value++;
    }

    public void reset() {
        value = 0;
    }

    public int getValue() {
        return value;
    }
}
```

Ci sono cose in più.. che conosceremo presto

la stessa cosa in Scala

Scala

```scala
class Counter {

  private var value: Int = 0 // Valore dello stato

  def inc(): Unit = {
    value += 1
  }

  def reset(
    value =
  }

  def getVal
    value
  }
}
```

la stessa classe Counter in Kotlin

Kotlin

```kotlin
class Counter {

    private var value: Int = 0 //

    fun inc() {
        value++
    }

    fun reset() {
        value = 0
    }

    fun getValue(): Int {
        return value
    }
}
```

la stessa cosa in C#

```csharp
public class Counter
{
    private int _value; // Valore dello stato

    public void Inc()
    {
        _value++;
    }

    public void Reset()
    {
        _value = 0;
    }

    public int GetValue()
    {
        return _value;
    }
```

Ci sono cose in più.. che conosceremo presto

GENERAZIONE AUTOMATICA

Certo! Ecco come puoi scrivere la classe `Counter` usando la quiet syntax di Scala 3:

Code

Alternativa, chiedendo di usare la sintassi compatta di Scala 3

```
class Counter:
  // Stato del contatore
  private var value: Int = 0

  // Metodo per incrementare il valore
  def inc(): Unit =
    value += 1

  // Metodo per azzerare il valore
  def reset(): Unit =
    value = 0

  // Metodo per recuperare il valore attuale
  def getValue: Int =
    value
```

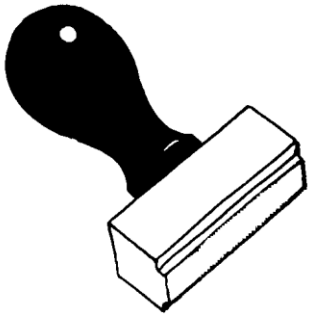
```
// Esempio di utilizzo
@main def runCounter(): Unit =
  val counter = Counter()

  counter.inc()
  counter.inc()
  println(counter.getValue) // Stampa 2

  counter.reset()
  println(counter.getValue) // Stampa 0
```

L'ADT COME MODELLO

- La definizione dell'ADT costituisce una sorta di «modello», che potrà poi essere usato per creare *oggetti* («variabili») di quel certo tipo.
- L'ADT assomiglia dunque a un *timbro: definisce le caratteristiche degli oggetti che verranno creati a sua immagine,* che condivideranno:
  - la stessa *struttura interna* (dati)
  - le stesse *operazioni* (dichiarazioni di funzioni)
  - lo stesso *funzionamento* (implementazione di funzioni)

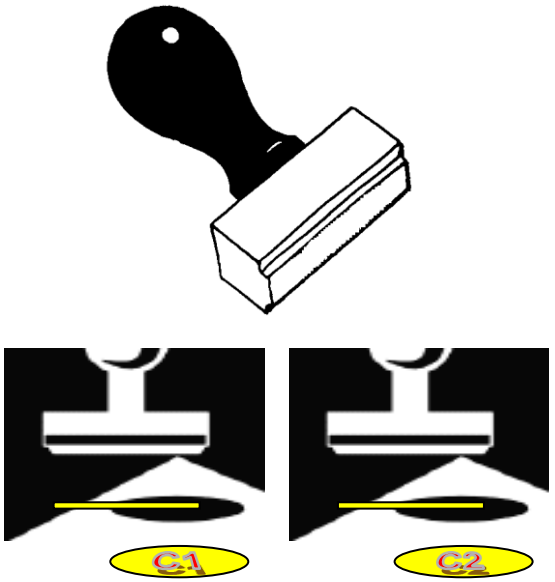


DAL TIPO ALL'OGGETTO

- Ogni atto creativo («timbrata») dà luogo a un *nuovo oggetto* di quel certo tipo, *distinto* dai precedenti
- Gli oggetti creati sulla base dello stesso modello/ADT condividono dunque le stesse caratteristiche, ma hanno ciascuno la propria *identità* e il proprio *stato*

Nell'esempio a lato:

- usando il «timbro» **`Counter`** due volte si creano *due distinti oggetti,* ***`c1`*** *e* ***`c2`***
- essi condividono la stessa *struttura interna,* le stesse *operazioni* e lo stesso *funzionamento*
- MA ciascuno ha la propria *identità* e il proprio *stato*



OGGETTI COME ISTANZE DI TIPI

- Si usa dire che gli oggetti sono *istanze* di un dato tipo
- Per creare oggetti si deve dunque *istanziare* il 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

- Si usa dire che gli oggetti sono *istanze* di un dato tipo
- Per creare oggetti si deve dunque *istanziare* il 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 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
    - 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

C#, Scala,Kotlin

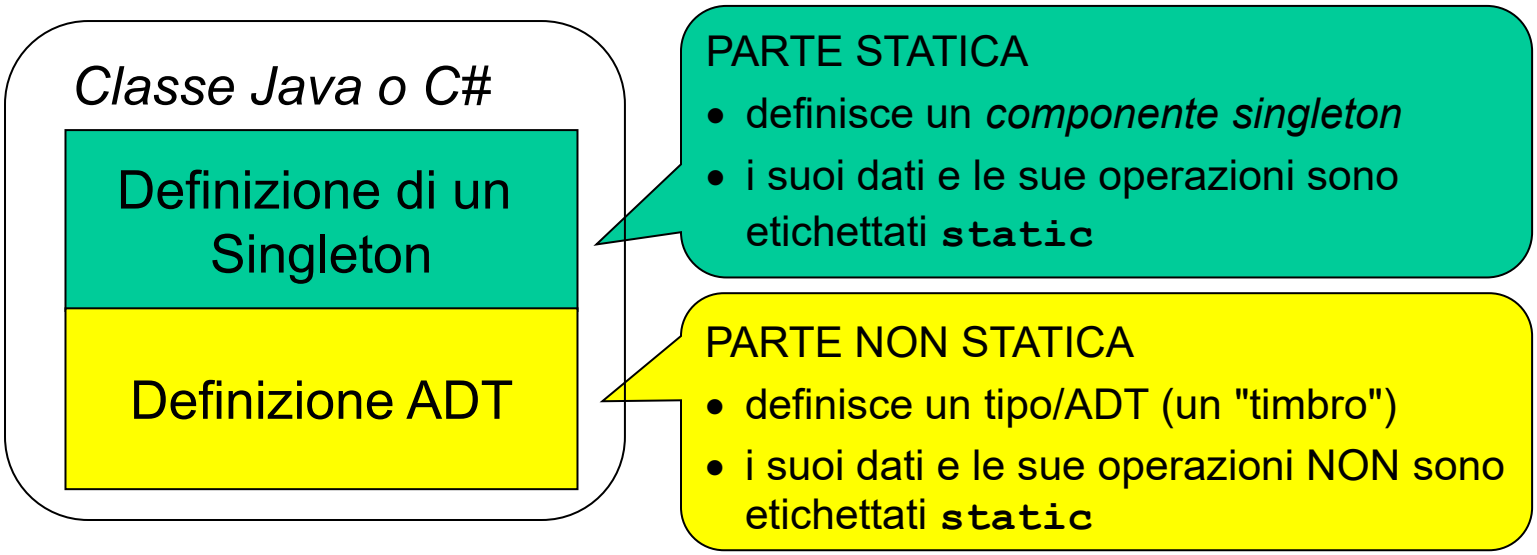
CLASSI in Java & C# vs. CLASSI in Scala & Kotlin

- In tutti i linguaggi a oggetti considerati, gli ADT sono espressi tramite *classi* (senza parti statiche)
- In Java e C#, il costrutto **`class`** serve anche a definire oggetti singleton, espressi da *membri statici*
- In Scala e Kotlin, invece, gli oggetti singleton sono definiti con il costrutto **`object`**, i cui membri sono *già intrinsecamente* statici.

- Dunque, in Java e C# il costrutto class gioca *due ruoli*, che in Scala e Kotlin sono invece assegnati a due costrutti diversi.

CLASSI in Java & C#
vs. CLASSI in Scala & Kotlin

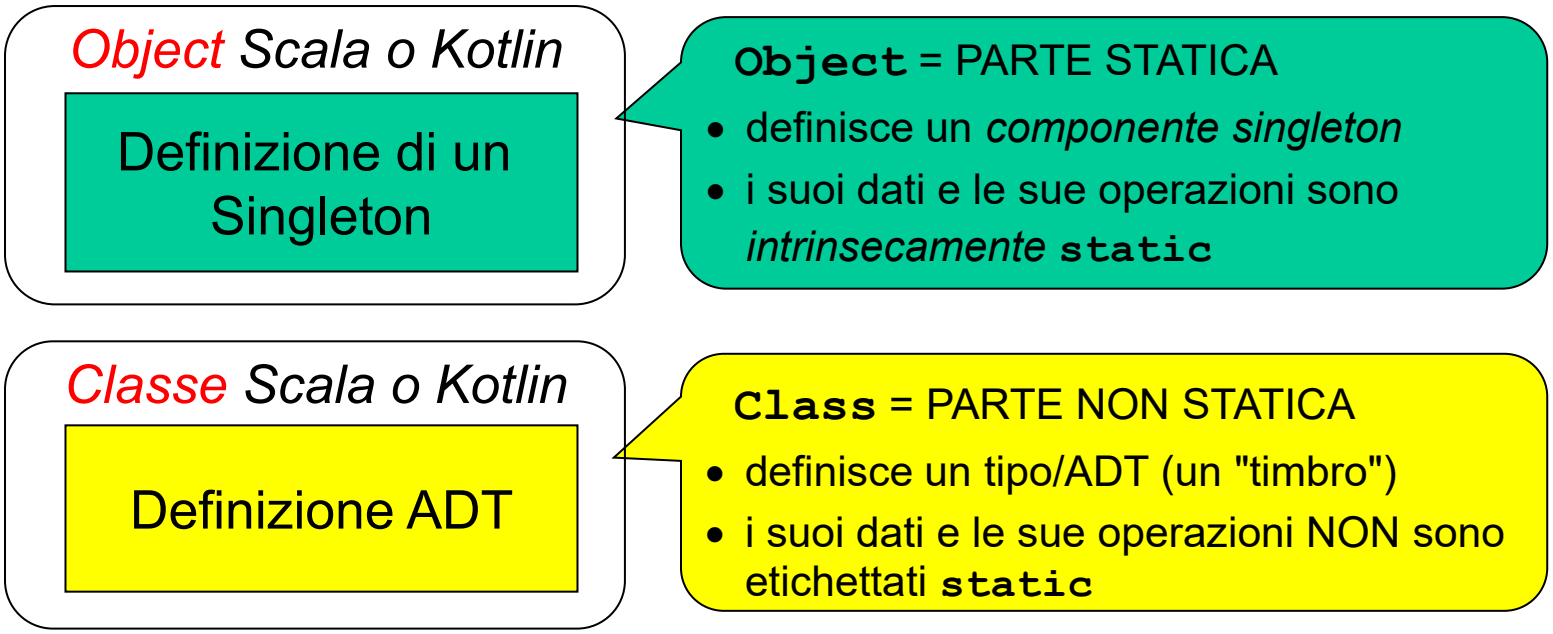
- In Java e C#, il costrutto **`class`** svolge *due ruoli*
  - i membri statici (se presenti) definiscono un singleton
  - i membri non statici (se presenti) definiscono un ADT



Spesso una classe Java o C# ha *una sola* delle due parti, perché svolge *uno solo* di questi due ruoli. Ma può fare comodo che li svolga entrambi…☺

CLASSI in Java & C# vs. CLASSI in Scala & Kotlin

- In Scala e Kotlin, il costrutto **`class`** svolge *un solo ruolo*
  - ha solo membri non statici e si usa solo per definire un ADT
  - l'altro ruolo è svolto dal costrutto separato **`object`**



USARE L'ADT: in C

- Usare un ADT significa *creare oggetti («variabili») di quel tipo* nella propria applicazione – cioè, in primis, nel main
- In C, ciò significa *definire una o più variabili (o puntatori)* del tipo desiderato e usarli per manipolare gli oggetti richiesti

ESEMPIO 1 (senza puntatori)

C

```
#include "contatore.h"

int main() {
   int v1, v2;
   contatore c1, c2;
   reset(&c1); reset(&c2);
   inc(&c1); inc(&c1); inc(&c2);
   v1 = getValue(c1); v2 = getValue(c2);
   return 0;
}
```

USARE L'ADT: in C

- L'esempio precedente usava variabili C «classiche»
- In alternativa si possono *definire puntatori:* i veri «oggetti» saranno allora allocati dinamicamente, tramite *malloc*

ESEMPIO 2 (con puntatori)

C

```
#include "contatore.h"

int main() {
   int v1, v2;
   contatore *c1, *c2;
   c1 = (contatore*)malloc(sizeof(contatore));
   c2 = (contatore*)malloc(sizeof(contatore));
   reset(c1); reset(c2);
   inc(c1); inc(c1); inc(c2);
   v1=getValue(*c1); v2=getValue(*c2);
   free(c1); free(c2);
   return 0;
}
```

puntatori C

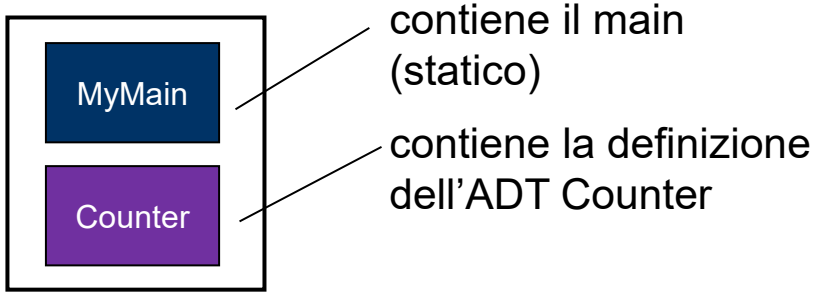
allocazione in memoria dinamica

deallocazione esplicita

USARE L'ADT NEI LINGUAGGI A OGGETTI

- Usare un ADT significa creare oggetti («variabili») di quel tipo nella propria applicazione – cioè, in primis, nel main
- Il main è certamente statico
  - deve esistere dall'inizio alla fine del programma
  - in Java e C# sarà un *membro statico* di una *classe*
  - in Scala e Kotlin sarà in un *object* (intrinsecamente statico)
- Se il main istanzia oggetti di un certo ADT (es. **`Counter`**), tale ADT sarà espresso da una classe (non statica)

Struttura dell' applicazione



ISTANZIARE OGGETTI

- Già, ma… *come si istanziano (creano) gli oggetti?*
- Gli oggetti «non singleton» sono sempre *entità dinamiche*, create *dinamicamente al momento del bisogno*
  - si fa uso della memoria dinamica (heap)
  - le variabili «classiche» restano (SE ci sono..) solo per i tipi base
- Per *istanziare oggetti* occorre dunque sempre far uso di puntatori, qui ridenominati *riferimenti*
  - operano a un *livello di astrazione più alto* dei puntatori C
- L'allocazione della memoria viene svolta dall'operatore **`new`**
  - concettualmente analogo alla **`malloc`** del C
  - ma *molto* più comodo e high-level ! ☺

OGGETTI DINAMICI COME ISTANZE DI CLASSI ADT

- Gli *oggetti dinamici* sono dunque creati *al bisogno* a immagine e somiglianza di una *classe ADT,* tramite l'operatore **`new`**
  - in Java e C#, esso va *sempre scritto esplicitamente*
  - in Kotlin l'operatore **`new`** è *sempre sottinteso* → keyword *abrogata*
  - in Scala la keyword **`new`** è divenuta *opzionale* dalla versione 2.13
- Novità cruciale: non occorre più occuparsi della dealloca-zione della memoria, c'è il *garbage collector !*
  - si evitano a priori tutti i bug (tanti…) dovuti all'errata gestione della memoria ☺☺

USARE L'ADT
dal caso C ai linguaggi OOP

Il main C con puntatori:

C

```
#include "contatore.h"
int main() {
   int v1, v2;
   contatore *c1, *c2;
   c1 = (contatore*)malloc(sizeof(contatore));
   c2 = (contatore*)malloc(sizeof(contatore));
   reset(c1); reset(c2);
   inc(c1); inc(c1); inc(c2);
   v1 = getValue(*c1); v2 = getValue(*c2);
   free(c1); free(c2);
   printf("%d\n",v1); printf("%d\n",v1);
   return 0;
}
```

puntatori

allocazione esplicita con molti (troppi) dettagli

deallocazione esplicita

USARE L'ADT
dal caso C ai linguaggi OOP

Il main C con puntatori:

C

```
#include "contatore.h"
int main() {
   int v1, v2;
   contatore *c1, *
   c1 = (contatore*
   c2 = (contatore*
   reset(c1); reset
   inc(c1); inc(c1)
   v1 = getValue(*c
   free(c1); free(c
   printf("%d\n",v1
   return 0;
}
```

Il main Java

Java

~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);
 }
}
```

Riferimenti

Allocazione esplicita con l'operatore **new** senza inutili dettagli!

Invocazione funzioni in stile «a oggetti» (notazione puntata)

deallocazione implicita (garbage collection)

USARE L'ADT: Java & C#

- OSSERVA: sono spariti tutti gli asterischi!
  - MOTIVO: il riferimento è molto più di un puntatore, gode del *dereferenziamento automatico!*

```
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);
 }
}
```

Java

~C#

Riferimenti

Dereferenziamento automatico!

USARE L'ADT: Scala & Kotlin

Scala

```
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);
  }
}
```

Keyword **new** opzionale

Specifiche di tipo **:Counter** e **:Int** eliminabili (type inference)

Kotlin

```
// object implicito, file MyMain.kt
  fun main(args: Array<String>) {
      var c1 : Counter = Counter();
      var c2 : Counter = 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);
  }
```

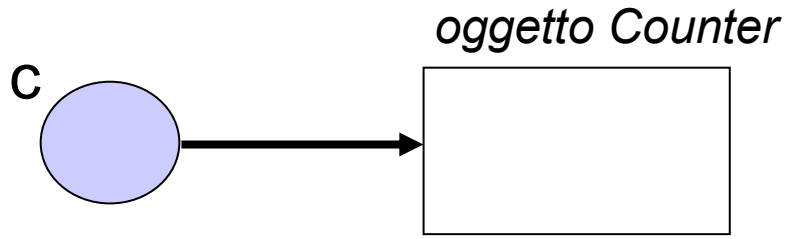
Keyword **new** abrogata

RECAP: CREAZIONE DI OGGETTI

- Per creare un oggetto dinamico, si agisce esplicitamente:
  - *prima* si definisce un riferimento del «giusto» tipo (il tipo è il la classe ADT che fa da modello)
  - *poi* si crea dinamicamente l'oggetto, tramite **`new`**

ESEMPIO

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



- MA a differenza dei puntatori, i *riferimenti* sono *disciplinati* e *protetti*: *l'indirizzo memorizzato è inaccessibile!* ☺
- La distruzione degli oggetti è inoltre *automatica*
  - il *garbage collector* elimina gli oggetti non più referenziati
  - si eliminano i rischi relativi all'errata deallocazione della memoria

RECAP: ACCESSO AGLI OGGETTI

- Dopo che l'oggetto è stato creato, si possono *invocare su di esso* le *funzionalità pubbliche* previste dalla classe
- Lo si fa con la classica notazione puntata, nella forma

  `riferimento.nomeFunzione(argomenti)`

- OSSERVA: c'è il dereferenziamento automatico!
  - sebbene il riferimento sia un puntatore, non occorre dereferenziarlo esplicitamente per accedere ai suoi membri (dati e funzioni), come accade invece in C con `*` o `->`
  - la notazione puntata «ingloba» il dereferenziamento
  - MOTIVO: non essendo possibile manipolare gli indirizzi (perché il contenuto del riferimento è inaccessibile), ogni azione non può che riferirsi all'oggetto puntato ☺

Un nuovo esempio di ADT: le frazioni

FRAZIONI: ANALISI (1/2)

- In matematica, una frazione è caratterizzata da
  - una coppia di interi (n,d), d≠0, solitamente scritta «n/d»
  - una serie di operazioni ammissibili
- Quali operazioni?
  - "*costruzione*" di una frazione a partire da una coppia di 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 con un'altra frazione*
    DUBBIO: «uguale» significa numeratore e denominatore identici, o rispetto della condizione di equivalenza ?

LE GRANDI QUESTIONI

- Inizializzare, Creare, o Costruire?
  - Inizializzare = *dare valore iniziale* a un oggetto che *già esiste*
  - Creare = *allocare memoria* per un *nuovo oggetto*
  - Costruire = creare l'oggetto + inizializzarlo
- Modificare oggetti esistenti o sintetizzare nuovi oggetti?
  - Modificare = alterare l'oggetto ricevuto
  - Sintetizzare = costruire un nuovo oggetto modificato
- Attribuzione delle responsabilità
  - Chi fa cosa?
  - Cosa è responsabilità del *chiamante* e cosa del *chiamato*?

FRAZIONI: ANALISI (2/2)

Alcune possibili scelte:

- costruzione di una frazione da due interi
  - opzione 1: il cliente crea e inizializza anche
  - 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

FRAZIONI: PROGETTO

- Dalla definizione dell'ADT:
  - una frazione è caratterizzata da due proprietà, *numeratore* e *denominatore*, *impostate inizialmente* e poi *mai più modificate*
  - numeratore e denominatore devono essere *recuperabili singolarmente* → due metodi *accessor,* **`getNum`** e **`getDen`**
- Dalle scelte precedenti:
  - se il cliente crea, ma *delega l'inizializzazione,* l'ADT deve prevedere una funzione di inizializzazione, **`init`**, che, dati due interi, inizializzi con tali valori la frazione
  - la riduzione ai minimi termini, **`minTerm`**, *restituisce una nuova frazione-risultato*, senza alterare la frazione corrente
  - il test di uguaglianza, **`equals`**, si basa sulla *condizione di equivalenza,* non sulla mera uguaglianza di numeratore e denominatore; l'argomento è *l'altra frazione,* con cui confrontare quella attuale

ADT FRAZIONE in Java & C#

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 other){
       return … // condizione di equivalenza
  }
  public Frazione minTerm(){
       return … // la nuova frazione ridotta
  }
}
```

Java

~C#

Inizializzazione

Accessor

FRAZIONI: USO

- Esperimento interattivo con Jshell:

```
jshell> Frazione f = new Frazione();
f ==> Frazione@55d56113

jshell> f.init(3,4);

jshell> f.getNum()
$4 ==> 3

jshell> f.getDen()
$5 ==> 4
```

Java

  - DUBBIO: se non l'avessimo inizializzata, cosa avrebbero restituito **`getNum`** e **`getDen`**…?

FRAZIONI: STAMPA

- E se volessimo stampare una frazione sotto forma di stringa del tipo «3/4», «2/3», etc. ?
  - non abbiamo ancora approfondito le stringhe, ma intanto…
- L'operatore + concatena stringhe
  - ESEMPIO: `"ciao" + "mondo"` → `"ciaomondo"`
- In Java e C# esso *concatena anche stringhe con numeri,* convertendo automaticamente in stringa questi ultimi
  - ESEMPIO: `31 + ":" + "4"` → `"31:4"`
- Sfruttandolo, è facile stampare frazioni:

```
jshell> f.getNum() + "/" + f.getDen()
$10 ==> "3/4"
```

FRAZIONI: USO

- Un possibile main che manipola frazioni:

Java

~C#

```
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");
 }
}
```

Riferimenti

`init` inizializza

Notazione uniforme, senza più *

Concatenazione stringhe con +

Come sarà `equals` ?

Deallocazione automatica ed implicita

ADT FRAZIONE in Scala & Kotlin

```
public class Frazione {
  private var num:Int = 0;
  private var den:Int = 1;
  def init(n:Int, d:Int) = { num = n; den = d; }
  def getNum():Int = { return num; }
  def getDen():Int = { return den; }
  def equals(other:Frazione):Boolean = {…}
  def minTerm():Frazione = {…}
}
```

Inizializzazione obbligatoria

Scala

~Kotlin

Kotlin: sostituire **def** con **fun**

Possibile uso:

```
def main(args: Array[String]) = {
  val f = new Frazione();
  f.init(3,4);
  println(f.getNum()); println(f.getDen());
  println(f.getNum().toString() + "/" + .getDen().toString());
}
```

Kotlin: togliere **new**

Kotlin: sostituire **def** con **fun**

Scala

~Kotlin

Necessaria conversione esplicita dei numeri in stringhe

UN POSSIBILE PROBLEMA

- Lo schema d'uso delineato prevede che, *per avere una frazione pronta per l'uso*, si debba *agire in due tempi*:
  - *prima*, creando il nuovo oggetto, con **`new`**
  - *poi*, inizializzandolo tramite **`init`**

```
f1 = new Frazione(); f1.init(3,4);
f2 = new Frazione(); f2.init(6,8);
```

- È palese il rischio di creare un oggetto *dimenticandosi poi di inizializzarlo:* un problema antico, causa di moltissimi bug
- Non è saggio contare sulla memoria delle persone: meglio prevedere un automatismo, il *costruttore*

CREAZIONE DI OGGETTI vs. COSTRUZIONE DI OGGETTI

- Finora, abbiamo *creato* oggetti:
  - definendo un riferimento del tipo opportuno
  - e creando l'oggetto tramite **`new`**

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

- Ma così facendo gli oggetti creati *non sono inizializzati*
  - rischio di operare su valori casuali → causa di molti bug!

- In un linguaggio moderno è opportuno *evitare ciò*: si devono costruire *solo* oggetti *compiutamente configurati*

COSTRUIRE è più che CREARE

COSTRUZIONE DI OGGETTI

- Più in generale, molti errori nel software sono causati tradizionalmente da *mancate inizializzazioni* di variabili.
- Per ovviare a questo problema, 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#

In Java e C#, il costruttore:

Java

~C#

- 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"

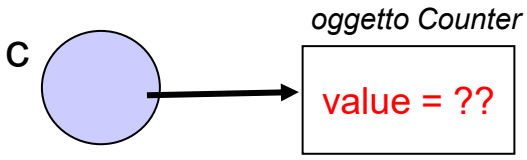
COSTRUZIONE DI DEFAULT

- Gli oggetti degli esempi precedenti sono stati:
  - creati *esplicitamente* da noi tramite **`new`**
  - costruiti *implicitamente* dal costruttore di defaut

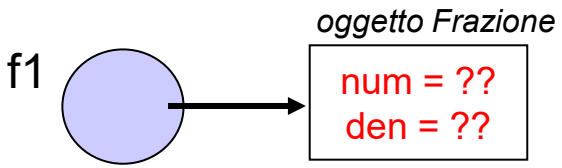
Java

~C#

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




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



- Il costruttore di default:
  - viene inserito dal compilatore *solo se la classe non prevede esplicitamente alcun costruttore esplicito*
  - dovrebbe essere sempre *definito esplicitamente* (salvo che la costruzione senza argomenti non abbia senso), 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 naturale a *un argomento*
- per la **`Frazione`**, ha senso poter specificare all'atto della costruzione *numeratore e denominatore*
  → costruttore naturale a *due argomenti*

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

+ COSTRUTTORI AUSILIARI per *casi particolari di uso frequente*

COSTRUTTORI PERSONALIZZATI

Costruttore primario (caso generale)

- per il **`Counter`**, costruttore a un argomento
  riceve il valore iniziale desiderato
- per la **`Frazione`**, costruttore a due argomenti
  riceve *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 in Java & C#

```
public class Counter {
   private int value;
   public Counter(int v){ value = v; }
   public Counter(){ value = 1; }

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

Costruttore primario a un argomento

Costruttore ausiliario a zero argomenti

Java

~C#

NB: il metodo **`init`**, previsto nelle versioni precedenti di questa classe, non è quindi più necessario e viene eliminato.

Frazione CON COSTRUTTORI in Java & C#

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;
  }
  ...
}
```

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

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 in Java & C#: ESEMPIO D'USO

Questo main istanzia e manipola contatori:

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 in Java & C#: 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 }
}
```

GENERAZIONE AUTOMATICA..?

scrivi una classe Java "Frazione" che rappresenti una frazione caratterizzata da numeratore e denominatore. Deve esserci anche un metodo equals per verificare che una frazione sia equivalente a un'altra.

```java
public class Frazione {
    private int numeratore;
    private int denominatore;

    // Costruttore che inizializza numeratore e denominatore
    public Frazione(int numeratore, int denominatore) {
        if (denominatore == 0) {
            throw new IllegalA
        }
        this.numeratore = nume
        this.denominatore =
        semplifica();
    }

    // Metodo per semplificare la frazione ai minimi termini
    private void semplifica() {
        int mcd = mcd(numeratore, denominatore);
        numeratore /= mcd;
        denominatore /= mcd;
    }

    // Metodo per calcolare il massimo comune divisore
    private int mcd(int a, int b) {
        while (b != 0) {
            int temp = b;
            b = a % b;
            a = temp;
        }
        return a;
    }
```

Problema: la frazione viene semplificata di default!

```java
    // Metodo per ottenere il numeratore
    public int getNumeratore() {
        return numeratore;
    }

    // Metodo per ottenere il denominatore
    public int getDenominatore() {
        return denominatore;
    }

    // Metodo equals per confrontare due frazioni
    @Override
    public boolean equals(Object obj) {
        if (this == obj)
            return true;
        if (obj == null || getClass() != obj.getClass())
            return false;
        Frazione other = (Frazione) obj;
        return numeratore == other.numeratore && denominatore == other.denor
    }

    // Metodo toString per rappresentare la frazione come stringa
    @Override
    public String toString() {
        return numeratore + "/" + denominatore;
    }
}
```

GENERAZIONE AUTOMATICA..?

scrivi una classe Java "Frazione" che rappresenti una frazione caratterizzata da numeratore e denominatore. Deve esserci anche un metodo equals per verificare che una frazione sia equivalente a un'altra.

```java
// Metodo equals per confrontare due frazioni
@Override
public boolean equals(Object obj) {
    if (this == obj)
        return true;
    if (obj == null || getClass() != obj.getClass())
        return false;
    Frazione other = (Frazione) obj;
    return numeratore == other.numeratore && denominatore == other.denominatore;
}
```

Inoltre, **`equals`** confronta solo numeratori e denominatori: il principio di equivalenza non è stato considerato!

Riproviamo:

Non devi semplificare la frazione. Inoltre il metodo equals non è corretto, dovrebbe esprimere l'equivalenza, non la semplice uguaglianza dei membri.

GENERAZIONE AUTOMATICA..?

Assolutamente! Hai ragione a sottolineare l'importanza di queste considerazioni. Ecco una nuova versione della classe `Frazione` che tiene conto delle tue osservazioni:

```
public Frazione(int numeratore, int denominatore) {
    if (denominatore == 0) {
        throw new IllegalArgumentException("Il denominato
    }
    this.numeratore = numeratore;
    this.denominatore = denominatore;
}
```

La semplificazione è stata tolta: bene.

```
@Override
public boolean equals(Object obj) {
    if (this == obj)
        return true;
    if (obj == null || getClass() != obj.
        return false;
    Frazione other = (Frazione) obj;
    return numeratore * other.denominatore == denominatore * other.numerato
}
```

Ora è corretto

Morale:

- la generazione automatica può sgrossare (talora anche bene)
- MA dobbiamo sempre e comunque *guardarci noi* e *giudicare noi !*

Frazione in Java & C#: ESEMPIO D'USO

Questo main istanzia e manipola frazioni:

```
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(…);
 }
}
```

Java

~C#

Costruttori: il cliente crea, il costruttore inizializza

COSTRUTTORI 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 qualificatori (**`val`**/**`var`**/nessuno) per gli argomenti

Scala

```
class Frazione(val num:Int, val den:Int = 1) {
  // Niente più costruttori espliciti
  // Niente più accessor espliciti
  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){
  // Niente più costruttori espliciti
  // Niente più accessor espliciti
  public fun equals(f:Frazione) : Boolean {condizione di equivalenza }
  public fun minTerm() : Frazione { return la nuova frazione ridotta }
}
```

COSTRUTTORI in Scala e Kotlin

- Conglobare il costruttore primario nell'intestazione:
  - rende «inevitabile» il costruttore principale e lo evidenzia
  - evita gli assegnamenti ripetitivi, tipici dei costruttori Java & C#
  - permette la *generazione automatica degli accessor* **`getXXX`** contribuendo così a una drastica «asciugatura» del codice
    - nessun qualificatore → *nessun accessor generato* (variabile locale al costruttore)
    - qualificatore **`val`** → *generato accessor in sola lettura* (campo pubblico ma read-only)
    - qualificatore **`var`** → *generati accessor in lettura e scrittura* (campo pubblico read/write)
  - permette infine anche di *specificare valori di default* per i singoli argomenti, catturando così tipici casi d'uso dei costruttori ausiliari → ulteriore «asciugatura» del codice

Frazione in Scala & Kotlin: ALL TOGETHER

Scala

```
class Frazione(val num:Int, val den:Int = 1) {
  // Niente più costruttori esplic
  // automaticamente dai parametri
  // 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 }
}
```

Cattura già il caso d'uso del costruttore ausiliario

Si potrebbe asciugare ulteriormente con la sintassi compatta di Scala 3

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 }
}
```

Cattura già il caso d'uso del costruttore ausiliario

Frazione in Scala & Kotlin: ESEMPIO D'USO

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`** /**`val`** e la specifica di tipo postfissa (l'operatore **`new`** è opzionale in Scala, abolito del tutto in Kotlin):

```
var f1 : Frazione = Frazione(3,4)
```

  Inoltre, in questi linguaggi la specifica di tipo può essere omessa se è deducibile dal contesto:

```
var f1 = Frazione(3,4)
```

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;
|            ^
```