



Alma Mater Studiorum-Università di Bologna Scuola di Ingegneria

Riferimenti a oggetti

Corso di Laurea in Ingegneria Informatica
Anno accademico 2021/2022

Prof. ENRICO DENTI

Dipartimento di Informatica – Scienza e Ingegneria (DISI)

RIFERIMENTI

- Un **riferimento** è simile a un puntatore, ma rispetto ad esso costituisce *un'astrazione di più alto livello*
 - riduce i pericoli legati all'abuso (o uso errato) dei puntatori e dei relativi meccanismi (aritmetica dei puntatori)
- Un riferimento viene *dereferenziato automaticamente* quando serve, senza necessità di * o altri operatori
 - ciò elimina i rischi e gli errori relativi al dereferencing esplicito
 - l'oggetto è *accessibile direttamente con la notazione puntata*:
`c.inc()` ; **`x = c.getValue()`** ;
- Si conserva l'espressività dei puntatori, ma *controllandone e disciplinandone l'uso*.

RIFERIMENTI vs. PUNTATORI

Puntatore (C)

- contiene l'indirizzo di una variabile (ricavabile con &)...
 - ... *e permette di manipolarlo in qualsiasi modo*
 - incluso spostarsi altrove (aritmetica dei puntatori)
- richiede *dereferencing esplicito*
 - operatore * (o [])
 - rischio di errore
- rischio di invadere aree altrui

Potente ma pericoloso.

Riferimento (Java, C#, etc.)

- contiene l'indirizzo di un oggetto (non ricavabile con operatori)...
 - ... *ma non consente di vederlo né manipolarlo!*
 - non esiste alcuna aritmetica dei puntatori
- ha il *dereferencing automatico*
 - niente più operatore * (o [])
 - niente più rischio di errore
- impossibile invadere aree altrui

Mantiene la potenza del puntatore ma disciplinandone l'uso.



RIFERIMENTI vs. PUNTATORI

Puntatore (C)

- contiene l'indirizzo di una variabile (ricavabile con &)...
 - ... *e permette di manipolarlo in qualsiasi modo*
 - incluso spostarsi altrove (aritmetica dei puntatori)
- richiede *dereferencing esplicito*
 - operatore * (o [])
 - rischio di errore
- rischio di invadere aree altrui

Potente ma pericoloso.

L'operatore & viola la barriera di astrazione

- accesso *indiscriminato* al livello sottostante

L'aritmetica dei puntatori completa il vulnus

- ottenuto un indirizzo, *si può andare ovunque!!*

RIFERIMENTI vs. PUNTATORI

Il riferimento invece *impedisce* di violare la barriera di astrazione

- nessun accesso diretto al livello sottostante

Non esiste aritmetica dei puntatori

- l'indirizzo contenuto consente di *accedere solo a quell'oggetto*

Riferimento (Java, C#, etc.)

- contiene l'indirizzo di un oggetto (non ricavabile con operatori)...
- ... ma non consente di vederlo né manipolarlo!
 - non esiste alcuna aritmetica dei puntatori
- ha il *dereferencing automatico*
 - niente più operatore * (o [])
 - niente più rischio di errore
- impossibile invadere aree altrui

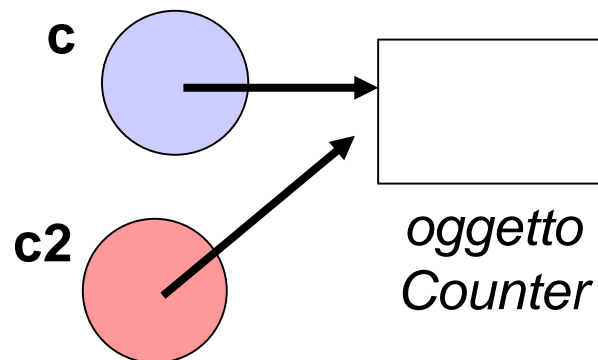
Mantiene la potenza del puntatore ma disciplinandone l'uso.

RIFERIMENTI A TIPI

- In C si possono definire, per ciascun tipo:
 - sia *variabili* (es. `int x;` ~~`Studente s;`~~)
 - sia *puntatori* (es. ~~`int *px;`~~ `Studente *s;`)
- In Java è il linguaggio a imporre le sue scelte:
 - *variabili* *per i tipi primitivi* (es. `int x;`)
→ passaggio dei parametri: PER VALORE
 - *riferimenti* *per gli oggetti* (es. `Counter c;`)
→ passaggio dei parametri: PER RIFERIMENTO
- In C# la situazione è "molto simile" a Java
- In Scala Kotlin, *everything is an object* (niente più tipi primitivi)

RIFERIMENTI: cosa ci si può fare

- Definirli senza inizializzarli: `Counter c;` Java C#
- Assegnare loro la costante `null`: `c = null;` (*non in Kotlin*)
Questo riferimento ora non punta a nulla
- Le due cose insieme: `Counter c = null;`
(*non in Kotlin*)
- Usarli per creare nuovi oggetti:
`c = new Counter();`
- Assegnarli uno all'altro:
`Counter c2 = c;`
c2 referencia lo stesso oggetto di c
- Confrontarli (nel senso di *identità*):
`c1 == c2` è vera se puntano allo stesso oggetto



ESPERIMENTO

```
public class Esempio2 {  
    public static void main(String[] args) {  
        Counter c1 = new Counter();  
        c1.reset(); c1.inc();  
        System.out.println(c1.getValue());  
  
        Counter c2 = c1;  
        c2.inc();  
        System.out.println(c1.getValue());  
        System.out.println(c2.getValue());  
    }  
}
```

Java

~C#

c1 vale 1

Principio di località

Le definizioni di variabile possono comparire ovunque, non solo all'inizio del programma

Ora c2 coincide con c1

Quindi, se si incrementa c2 ...

... risultano incrementati *entrambi*.

RIFERIMENTI.. NULLI ?

- Dunque, in Java e C#, i riferimenti possono essere:

Java

C#

- definiti senza inizializzarli:
- assegnati alla costante `null`:
- le due cose insieme:
- usati per creare nuovi oggetti:
- assegnati uno all'altro (alias):
- confrontati come aliasing:

```
Counter c;
```

```
c = null;
```

```
Counter c = null;
```

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

```
Counter c2 = c;
```

```
c1 == c2;
```

- Poiché però poter assegnare riferimenti a `null` è la premessa per le famigerate *NullPointerException*, *Scala e Kotlin non lo consentono in modo diretto*
 - il modo per impedirlo è diverso, ma il risultato ottenuto è lo stesso



RIFERIMENTI NON NULLI in Scala e Kotlin

- Java consente riferimenti *null*
 - passano la compilazione, ma facilmente esplodono a run-time

```
Counter c1 = null;  
Counter c2 = new Counter();  
System.out.println(c1.getValue());  
System.out.println(c2.getValue());
```

Java

NullPointerException
a run time!

- Scala consente riferimenti *null*, ma li considera istanze singleton della **classe `Null`**, che non ha metodi
 - eventuali accessi successivi causano quindi *errore di compilazione*

```
var c1 : Counter = null;  
var c2 = new Counter();  
println( c1.getValue() );  
println( c2.getValue() );
```

Scala

*getValue is not a
member of Null*

RIFERIMENTI NON NULLI in Scala e Kotlin

- Java consente riferimenti *null*
 - passano la compilazione, ma facilmente esplodono a run-time

```
Counter c1 = null;  
Counter c2 = new Counter();  
System.out.println(c1.getValue());  
System.out.println(c2.getValue());
```

Java

NullPointerException
a run time!

- Kotlin **non consente tout court** riferimenti *null*
 - ogni tentativo causa quindi *errore di compilazione*
 - (...a meno che non si usi una *speciale sintassi...*)

```
var c1 : Counter = null;  
var c2 = Counter();  
println( c1.getValue() );  
println( c2.getValue() );
```

Kotlin

*Null can not be a value of
a non-null type Counter*

*Only safe calls allowed
on nullable receiver*

RIFERIMENTI NULLI in Kotlin

- Kotlin accetta riferimenti *null* solo se
 - il **tipo** è **esplicitamente dichiarato** come ***tipo?***
 - il **metodo** è **esplicitamente chiamato** con l'operatore ***?.***

```
var c1 : Counter? = null;  
var c2 = new Counter();  
println( c1?.getValue() );  
println( c2.getValue() );
```

Kotlin

OK

- Il senso è che **un null non dev'essere frutto del caso**
 - nei rari casi in cui sia *specifico intento* del progettista inserire un null, lo si obbliga a *esplicitare tale intenzione* utilizzando operatori ad hoc, *volutamente più verbosi (e scomodi da usare...)*



MINI-TEST a CONFRONTO

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

Java

```
public fun main(args: Array<String>){  
    var c1 : Counter = Counter();  
    c1.reset(); c1.inc();  
    println(c1.getValue());  
    var c2 : Counter = c1;  
    c2.inc();  
    println(c1.getValue());  
    println(c2.getValue());  
}
```

Kotlin

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

Scala

MINI-TEST a CONFRONTO

```
public static void main(String[] args){  
    Counter c1 = new Counter();  
    c1.reset(); c1.inc();  
    System.out.println(c1.getValue());  
    Counter c2 = c1;  
    c2.inc();  
    System.out.pr  
    System.out.pr  
}
```

Java

In Scala e Kotlin, la specifica di tipo
:Counter può essere *omessa*
perché deducibile dal contesto

```
public fun main(args: Array<String>){  
    var c1 : Counter = Counter();  
    c1.reset(); c1.inc();  
    println(c1.getValue());  
    var c2 : Counter = c1;  
    c2.inc();  
    println(c1.getValue());  
    println(c2.getValue());  
}
```

Kotlin

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

Scala



MINI-TEST a CONFRONTO

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

Java

Scala 3: new opzionale

```
public fun main(args: Array<String>){  
    var c1 = Counter();  
    c1.reset(); c1.inc();  
    println(c1.getValue());  
    var c2 = c1;  
    c2.inc();  
    println(c1.getValue());  
    println(c2.getValue());  
}
```

Kotlin

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

Scala

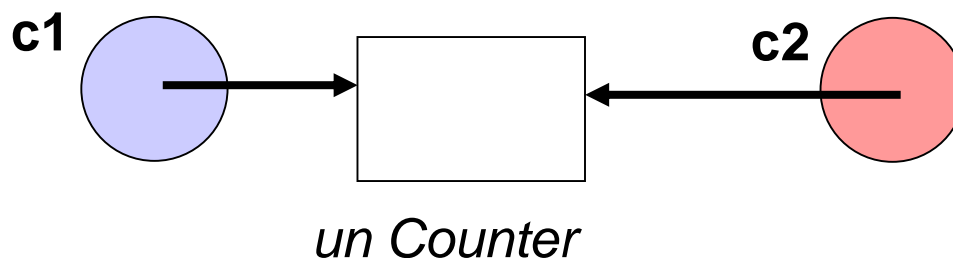


CONFRONTO DI IDENTITÀ (vs UGUAGLIANZA) DI OGGETTI

- In Java e C#, l'operatore `==` confronta l'**identità** di due oggetti, ossia verifica se siano *lo stesso* oggetto
 - è un confronto fra *riferimenti*
 - l'espressione `c1==c2` è vera solo se `c1` e `c2` puntano allo *stesso identico* oggetto, ossia se sono due *alias*

Java

C#



`c1 == c2` è *vero*

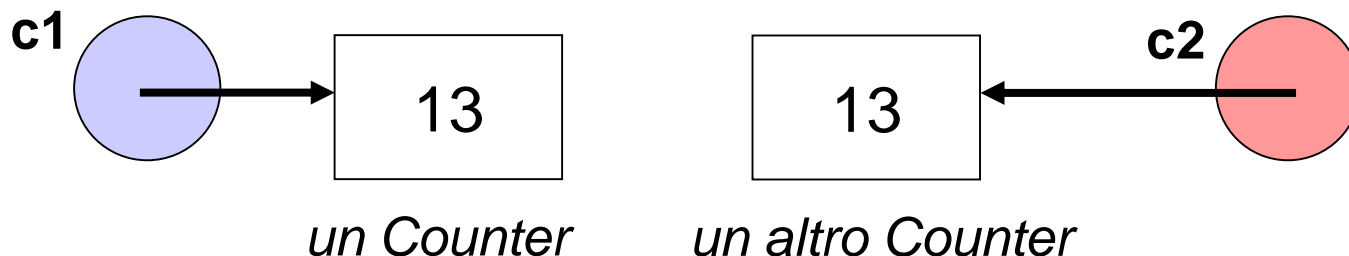


CONFRONTO DI IDENTITÀ (vs UGUAGLIANZA) DI OGGETTI

- Di conseguenza e *intenzionalmente*, l'operatore == in Java e C# *non* si basa sul **valore** dell'oggetto
 - dunque, in caso di due oggetti fotocopia ma *distinti*, `c1==c2` darà come risultato *falso*

Java

C#



`c1 == c2` è *falso*



CONFRONTO DI IDENTITÀ (vs UGUAGLIANZA) DI OGGETTI

```
public class Esempio3 {  
    public static void main(String[] args) {  
        Counter c1 = new Counter();  
        Counter c2 = new Counter();  
        System.out.println( c1 == c2 );  
    }  
}
```

Java

~C#

c1 e c2 sono qui
due oggetti distinti

FALSO, perché non puntano allo stesso oggetto.
In Java e C#, il *valore* dei contatori non rileva: il criterio è
l'uguaglianza dei *referimenti*, ossia l'*identità* degli oggetti

In Scala e Kotlin, in apparenza il risultato è identico
MA
come vedremo, il meccanismo in realtà è diverso...

Scala

Kotlin



MINI-TEST DI IDENTITÀ a CONFRONTO

```
public static void main(String[] args)
{
    Counter c1 = new Counter(13);
    Counter c2 = new Counter(13);
    System.out.println(c1==c2);
}
```

false

Java

```
public static void Main(string[] args)
{
    Counter c1 = new Counter(13);
    Counter c2 = new Counter(13);
    Console.WriteLine(c1==c2);
}
```

False

C#

```
public fun main(args: Array<String>)
{
    val c1 = Counter(13); println(c1)
    val c2 = Counter(13); println(c2)
    println(c1==c2)
}
```

false

Kotlin

```
def main(args: Array[String]) =
{
    val c1 = Counter(13);
    val c2 = Counter(13);
    println(c1==c2)
}
```

false

Scala



CONFRONTO DI UGUAGLIANZA DI OGGETTI

- Spesso però occorre confrontare oggetti non per identità, ma *in base a un qualche **criterio** legato al loro «valore»*
 - tale criterio deve necessariamente essere **personalizzabile**: *in base a cosa due oggetti dovrebbero essere considerati «uguali»?*
- A tal fine, tutti i linguaggi a oggetti introducono consentono al progettista di **specificare un criterio di uguaglianza** nella definizione della classe
- Lo si fa implementando lo **speciale metodo `equals`** (*in C, **Equals***)
 - l'esatta forma del metodo dipende dallo specifico linguaggio



ESEMPIO

UGUAGLIANZA DI Counter

Per specificare il criterio di uguaglianza di due **Counter**

- Signature del metodo da implementare (*per ora...*):
`boolean equals(Counter x)` (Java)
`bool Equals(Counter x)` (C#)
`equals(x:Counter) : Boolean` (Scala, Kotlin)
- POSSIBILE CRITERIO: considerarli uguali se hanno lo stesso *valore*
`return value == x.value`

A parole:

«I due Counter sono uguali se il valore dell'oggetto corrente («questo» oggetto) è uguale a quello dell'oggetto ricevuto come argomento" x («l'altro» oggetto)



ESEMPIO

UGUAGLIANZA DI Counter

Per specificare il criterio di uguaglianza di due **Counter**

- Signature del metodo da implementare (*per ora...*):

`boolean equals(Counter x)` (Java)

`bool Equals(Counter x)` (C#)

- PC...der... valore

Valore dell'oggetto corrente
(«questo» oggetto)

Valore dell' «altro» oggetto
(ricevuto come argomento)

`return value == x.value`

A parole:

I due Counter sono uguali se il valore dell'oggetto corrente («questo» oggetto) è uguale a quello dell'oggetto ricevuto come argomento, x («l'altro» oggetto)



Counter CON equals NEI VARI LINGUAGGI

```
public class Counter {  
    public Counter(int value){ this.value=value; }  
    private int value;  
    public int getValue() { return value; }  
    public boolean equals(Counter x) { return value == x.value; }  
  
    public static void main(String[] args){  
        Counter c1 = new Counter(13);  
        Counter c2 = new Counter(13);  
        System.out.println( c1.equals(c2) );  
    }  
}
```

Java

true

```
public class Counter {  
    public Counter(int value){ this.value=value; }  
    private int value;  
    public int GetValue() { return value; }  
    public bool Equals(Counter x) { return value == x.value; }  
  
    public static void Main(string[] args){  
        Counter c1 = new Counter(13);  
        Counter c2 = new Counter(13);  
        Console.WriteLine( c1.Equals(c2) );  
    }  
}
```

C#

True

```
fun main() {  
    val c1 = Counter(13);  
    val c2 = Counter(13);  
    println(c1.equals(c2))  
}  
  
public class Counter(private var value:Int) {  
    public fun getValue() : Int = value;  
    public fun equals(x:Counter) : Boolean { return value == x.value; }  
}
```

Kotlin

true

```
object Test{  
    def main(args:Array[String]) = {  
        val c1 = Counter(13);  
        val c2 = Counter(13);  
        println(c1.equals(c2))  
    }  
}  
  
class Counter(private var value:Int) {  
    def getValue() : Int = value;  
    def equals(x:Counter) : Boolean = { return value == x.value; }  
}
```

Scala

true

NB: in realtà, come vedremo, *non è proprio così* che si dovrebbe fare...



LA KEYWORD `this`

Per meglio evidenziare la simmetria fra i due oggetti del confronto («questo» e «l'altro»), conviene sfruttare la **parola chiave `this`** per denotare esplicitamente *l'oggetto corrente*

- Aniché scrivere

```
return value == x.value
```

- È più opportuno scrivere

```
return this.value == x.value
```

A parole:

*I due Counter sono uguali se il valore di «**questo**» oggetto è uguale a quello dell' «altro» oggetto ricevuto come argomento*



CONVENZIONE: *this* & *that*

Per completare «esteticamente» la simmetria, si usa chiamare *that* o *other* l'oggetto ricevuto come argomento

- NB: non sono parole chiave, sono solo nomi «convenienti»
- È quindi più opportuno scrivere

```
return this.value == that.value
```

- o, in Kotlin:

```
return this.value == other.value
```

A parole:

I due Counter sono uguali se il valore di «questo» oggetto è uguale a quello dell' «altro» oggetto



Counter CON this / that

Counter.java (o .cs)

```
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){  
        return this.value == that.value; }  
}
```

Java

~C#

~Scala

~Kotlin

DUBBIO: ma si può scrivere `that.value`,
essendo `value` privato...?

COLLAUDO

```
public class Esempio3b {  
    public static void main(String[] args) {  
        Counter c1 = new Counter();  
        Counter c2 = new Counter();  
  
        System.out.println(c1 == c2);  
  
        System.out.println(c1.equals(c2));  
    }  
}
```

Java

~C#

~Scala

~Kotlin

Sono oggetti distinti

Falso

VERO: sebbene non puntino allo stesso oggetto, sono
uguali secondo il criterio specificato dalla classe

Però, il criterio dell' "uguaglianza di contenuto" non è il solo possibile:
dipende dalla situazione.



UN CRITERIO ALTERNATIVO

- L'uguaglianza *di contenuto* non è l'unico criterio utile
 - ad esempio, se due contatori servissero *come orologi*, potrebbe essere utile un criterio di conteggio *modulo 24*
 - gli angoli sono spesso considerati uguali *modulo 360°*
 - per non parlare delle frazioni....!
- Criterio "per orologi": contatori uguali *se hanno lo stesso valore modulo 24*
`return this.value % 24 == that.value % 24`
- Generalizzando a tutto ciò che è uguale *modulo K* (es. contachilometri con $K=10^N$, angoli con $K=360$):
`return this.value % K == that.value % K`

UN ALTRO ESEMPIO: ANGOLI

- Un **angolo** può incapsulare il valore numerico in gradi

```
public class Angle {  
    private double value;  
  
    public Angle(double v) { value = v; }  
    public double getValue(){ return value; }  
    ...  
}
```

Java

~C#

~Scala

~Kotlin

- Ma tipicamente in matematica gli angoli si considerano uguali *modulo* 360°

```
public boolean equals(Angle that) {  
    return this.value % 360 == that.value % 360;  
}
```

..MA per angoli negativi..?

UN ALTRO ESEMPIO: ANGOLI

- La classe completa:

```
public class Angle {  
    private double value;  
    public Angle(double v) { value = v; }  
    public double getValue(){ return value; }  
    public boolean equals(Angle that){  
        return this.value % 360 == that.value % 360; }  
}
```

Java

~C#

~Scala

~Kotlin

Funziona anche per valori reali,
MA dà il remainder, non il modulo..

- Collaudo:

```
assert new Angle(30).equals(new Angle(390)); // OK  
assert new Angle(10).equals(new Angle(390)); // NO
```

- MA non funziona con mix di angoli positivi e negativi.. ☹
→ `Math.floorMod` risolverebbe, ma solo per angoli interi.. ☹



UN ALTRO ESEMPIO: ANGOLI

- L'operatore % restituisce il *remainder* (che può essere *negativo*), non il modulo (che si intenderebbe positivo)
→ coi valori negativi, non funziona come pensiamo
- Si può ovviare con un'espressione più sofisticata:
$$(x \% y + y) \% y$$
 - se x è negativo, anche $x \% y$ lo è, ma in valore assoluto è $< y$
quindi, $(x \% y + y)$ è sempre positivo
 - ergo, prendendolo $\%y$, si ottiene il risultato voluto
 - se x è positivo l'aggiunta di $+y$ non altera il risultato, poiché l'effetto sarà neutralizzato dall'operazione $\%y$ finale
- Con tale modifica, anche questo collaudo ha successo:

```
assert new Angle(31.5).equals(new Angle(-688.5));
```



UN TERZO ESEMPIO: FRAZIONI

Nel caso delle **frazioni**, definite come *coppia* (*num*, *den*),
quand'è che due oggetti si possono ritenere **uguali** ?

- **non solo** quando numeratore e denominatore sono identici
- **ma più in generale** quando le due frazioni sono **equivalenti**
1/2 è uguale a 3/6, a 9/18 ... e tante altre!

CRITERIO: **n/m** EQUIVALE A **p/q** SE **n × q = m × p**

CODIFICA

```
public class Frazione {  
    ...  
    public boolean equals(Frazione that) {  
        return this.num * that.den == this.den * that.num;  
    }  
}
```

Java

~C#

~Scala

~Kotlin



LA KEYWORD `this` (continua)

- Come si diceva, un altro uso tipico di questa keyword è quello di *disambiguare casi di potenziale omonimia*
- SCENARIO TIPICO: un *parametro di un metodo* è *omonimo a un campo-dati della classe*
 - può sembrare strano o assurdo, ma in realtà accade sempre!
 - lo si fa apposta, per sottolineare a cosa corrisponde quel certo argomento, evitando un'inutile moltiplicazione di nomi

L'ambiguità si risolve con la parola chiave `this`

```
public Counter(int value) { this.value = value; }
```

value (da solo) è il nome dell'argomento del metodo

this.value è il campo dati dell'oggetto corrente



PATTERN DI ASSEGNAMENTO

- I campi-dati di un oggetto sono tipicamente inizializzati nei costruttori con sequenze del tipo:

this.nome = nome

```
public class Counter {  
    private int value; // è this.value  
    public Counter() {this.value = 1; }  
    public Counter(int value){ this.value = value; }  
    public void reset() { this.value = 0; }  
    public void inc() { this.value++; }  
    public int getValue(){ return this.value; }  
    public boolean equals(Counter that){  
        return this.value == that.value; }  
}
```

Java

~C#

Pattern tipico



LA KEYWORD `this` (continua)

C'è un altro caso tipico di uso della keyword **this**:
richiamare un costruttore *da un altro*

- **this()** invoca il costruttore *senza argomenti*
- **this(...)** invoca il costruttore *corrispondente in numero e tipo alla lista di argomenti indicata*

Java

C#

Scala, Kotlin:
leggere differenze

MOTIVAZIONI

- Economia: si scrive solo il *costruttore più generale* (con più argomenti), poi si implementano i *costruttori ausiliari rimpalando la loro azione sul primo*, con opportuni *valori di default* per i parametri non specificati
- Buona pratica: promuove l'idea che una classe debba avere un *single point of entry*: il *costruttore primario*



this NEI COSTRUTTORI in JAVA

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

Senza **this**, il costruttore ausiliario *duplica* buona parte del codice del costruttore primario

Java

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

Con **this**, il costruttore ausiliario *rimpalla* l'azione sul costruttore primario, fornendogli i valori di default

Java

SCALA & KOTLIN: il costruttore primario è spostato a livello di dichiarazione della classe. I costruttori ausiliari hanno una forma particolare (→)



this NEI COSTRUTTORI in JAVA

```
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

Ulteriore vantaggio: TESTABILITY

Un singolo point of entry = un singolo punto d'ingresso da collaudare (bene)

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

Java

Anche qui, con **this** il costruttore ausiliario *rimpalla* l'azione sul costruttore primario, evitando duplicazioni



UN ALTRO ESEMPIO: Point

```
public class Point {  
    double x, y, z;  
  
    public Point(double x, double y, double z) {  
        this.x = x; this.y = y; this.z = z;  
    }  
  
    public Point(double x, double y) {  
        this(x, y, 0); // richiama costruttore precedente  
    }  
  
    public Point(double x) {  
        this(x, 0); // richiama costruttore precedente  
    }  
}
```

Costruttore a 3 argomenti
(il caso più generale)

Java

Costruttore a 2 argomenti:
richiama quello a 3 argomenti

Costruttore a 1 argomento:
richiama quello a 2 argomenti

ESEMPIO

Java

```
public class EsempioPoint {  
    public static void main(String[] args) {  
        Point p1 = new Point(3,2,1);  
        Point p2 = new Point(4,5); // (4,5,0)  
        Point p3 = new Point(7);   // (7,0,0)  
        ...  
    }  
}
```

L'argomento z viene posto a 0 dal costruttore

Gli argomenti y, z sono posti a 0 dal costruttore

Inizializzato da Point/1 che richiama Point/2 (dando come 2° argomento 0), che a sua volta richiama Point/3 (dando come 3° argomento 0)



this NEI COSTRUTTORI in C#

C#

```
public class Point {
```

```
    double x, y, z;
```

```
    public Point(double x, double y, double z) {
```

```
        this.x = x; this.y = y; this.z = z;
```

```
    }
```

```
    public Point(double x, double y) : this(x, y, 0) {
```

```
        // corpo vuoto
```

```
    }
```

```
    public Point(double x) : this(x, 0) {
```

```
        // corpo vuoto
```

```
    }
```

```
}
```

Costruttore a 3 argomenti
(il caso più generale)

Costruttore a 2 argomenti: richiama quello a 3 argomenti

Costruttore a 1 argomento:
richiama quello a 2 argomenti



this NEI COSTRUTTORI AUSILIARI in Scala e Kotlin

In Scala, i costruttori ausiliari si chiamano **this**:

Scala

```
class Point(val x: Double, val y: Double, val z: Double) {  
  def this(x: Double, y: Double) = this(x, y, 0);  
  def this(x: Double) = this(x, 0);  
}
```

Costruttore
primario a 3
argomenti

Costruttore a 1 argomento:
richiama quello a 2 argomenti

Costruttore a 2 argomenti:
richiama quello a 3 argomenti

In Kotlin si introduce invece la keyword **constructor**:

Kotlin

```
class Point(val x: Double, val y: Double, val z: Double) {  
  constructor(x: Double, y: Double) : this(x, y, 0.0);  
  constructor(x: Double) : this(x, 0.0);  
}
```

Necessaria una costante
Double (non Int)



Java: PRE-INIZIALIZZAZIONI

- Talora può accadere che vi siano inizializzazioni identiche per tutti i costruttori, o che debbano essere fatte *prima ancora che il (resto del) costruttore inizi ad operare*.
- In questi casi, è possibile **specificare l'inizializzazione direttamente nella dichiarazione del dato**
 - ad esempio, se il campo `z` di `Point` dovesse essere pre-inizializzato a 18, si potrebbe scrivere:

```
public class Point {  
    double x, y, z = 18; // pre-inizializzato  
    ...  
}
```

Java

- Non c'è differenza di efficienza, è solo questione di stile e leggibilità.

Problemi di incapsulamento



UN PROBLEMA COI CAMPI PRIVATI in Java e C#

- Riconsideriamo il codice di `Counter` e in `Frazione`.
- I campi dati (**value**, **num**, **den**) sono privati.. ma *DI CHI?*

```
public class Counter {  
    private int value; // ovvero this.value  
    ...  
    public boolean equals(Counter that) {  
        return this.value == that.value; }  
}
```

Java

~C#

```
public class Frazione {  
    private int num, den;  
    public boolean equals(Frazione that) {  
        return this.num * that.den == this.den * that.num;  
    }  
}
```

Java

~C#

Ma.. si può scrivere `that.den`,
essendo `den` privato?

Ma.. si può scrivere `that.num`,
essendo `num` privato?



UN PROBLEMA COI CAMPI PRIVATI in Scala e Kotlin

- Analogamente per il codice di **Counter** in Scala e Kotlin
- Il campo dati **value** è privato.. ma *DI CHI?*

```
class Counter(val value : Int) {  
  def equals(that : Counter) : Boolean = {  
    return this.value == that.value; }  
}
```

Scala

Keyword **return** non indispensabile

```
class Counter(val value : Int) {  
  fun equals(that : Counter) : Boolean {  
    return this.value == that.value; }  
}
```

Kotlin

```
object Main{  
  def main(args: Array[String]) : Unit = {  
    var c1 = new Counter(12);  
    var c2 = new Counter(12);  
    println(c1.equals(c2));  
  }
```

true

```
fun main(args: Array<String>) : Unit {  
  var c1 = Counter(12);  
  var c2 = Counter(12);  
  println(c1.equals(c2));  
}
```

true



UN PROBLEMA COI CAMPI PRIVATI

- A intuito, `private` dovrebbe voler dire *dell'oggetto..*
- *...ma allora non dovremmo poter scrivere `that.value`, che è un campo dell'oggetto `that`* (non di `this`, su cui è stato chiamato il metodo `equals`)

```
public class Counter {  
    private int value;    // ovvero this.value  
    ...  
    public boolean equals(Counter that) {  
        return this.value == that.value; }  
}
```

Java

~C#

Violazione di
incapsulamento

AARGH!!

Stiamo violando l'incapsulamento dell'oggetto `that`



UN PROBLEMA DI INCAPSULAMENTO

- Per rispettare l'incapsulamento dell'oggetto `that`,
non avremmo dovuto accedere direttamente ai suoi dati!
- Non era neanche necessario: bastava usare `getValue`!

```
public class Counter {  
    private int value;    // ovvero this.value  
    ...  
    public boolean equals(Counter that) {  
        return this.value == that.value; }  
}
```

Java

~C#

Violazione di
incapsulamento

```
public class Counter {  
    private int value;    // ovvero this.value  
    ...  
    public boolean equals(Counter that) {  
        return this.value == that.getValue(); }  
}
```

Java

~C#

Incapsulamento
rispettato



UN PROBLEMA DI INCAPSULAMENTO

- Ma allora.. perché funziona?
- *Perché non ci è stata impedita la compilazione?*

```
public class Counter {  
    private int value;    // ovvero this.value  
    ...  
    public boolean equals(Counter that) {  
        return this.value == that.value; }  
}
```

Java

~C#

Violazione di
incapsulamento

- Perché di base l'incapsulamento è *enforced solo a livello di classe, non di singolo oggetto*
- MOTIVO: non forzare il progettista ad aggiungere metodi di accesso *pubblici*, non previsti dal progetto, «solo per» rispondere a un'esigenza pratica.



UN PROBLEMA DI INCAPSULAMENTO

Una prima conclusione

- Nel progetto dei meccanismi di un linguaggio si devono spesso considerare esigenze contrastanti
 - l'enforcing dell'incapsulamento è uno di queste
- Anche se l'incapsulamento è enforced solo a livello di classe, ciò *non è un buon motivo per violarlo a livello di oggetto se si può evitare di farlo*
- Quindi: *se è presente un metodo accessor, è molto meglio usare quello che non accedere direttamente*
 - al dato di un oggetto ricevuto come argomento (**that**)
 - ma anche ai propri dati (**this**)



INCAPSULAMENTO A LIVELLO DI SINGOLO OGGETTO IN Scala

Tuttavia, questo non è l'unico approccio possibile

- ad esempio, in Scala in cui si può *imporre incapsulamento anche a livello di singolo oggetto*
 - notazione **private[oggetto]**
 - in particolare: **private[this]**

```
class Counter(private[this] val value : Int) {  
  def get() : Int = { return this.value; }  
  def equals(that : Counter) : Boolean = { return this.value = that.value; }  
  ✖ value value is not a member of Counter  
}
```

Incapsulamento violato, ma ok sull'oggetto **this**

Scala

Violazione di incapsulamento dello specifico oggetto **that**



INCAPSULAMENTO: DIFFERENZE

- Questa caratteristica viene talvolta presentata (W. Cook) come *differenza fra*:
 - Astrazioni di dato **con Abstract Data Types (ADT)**
 - Java classico, Scala default, Kotlin
 - Astrazioni di dato **propriamente Object-Oriented (OO)**
 - Scala con `private[this]`

REFACTORING DEL CODICE

Codice riorganizzato rispettando l'incapsulamento (fase 1):

```
public class Counter {  
    private int value;    // this.value  
    ...  
    public boolean equals(Counter that) {  
        return this.value == that.getValue();  
    }  
}
```

Java

~C#

Incapsulamento
rispettato per that

```
public class Frazione {  
    private int num, den; // this.num, this.den  
    public boolean equals(Frazione that) {  
        return  
            this.num * that.getDen() == this.den * that.getNum();  
    }  
}
```

Java

~C#

Incapsulamento
rispettato per that

Incapsulamento
rispettato per that



REFACTORING DEL CODICE

Codice riorganizzato rispettando l'incapsulamento (fase 2):

```
public class Counter {  
    private int value;    //  
    ...  
    public boolean equals(Counter that) {  
        return this.getValue() == that.getValue();  
    }  
}
```

Incapsulamento rispettato
anche per this

Java

~C#

```
public class Frazione {  
    private int num, den; // this.num, this.den  
    public boolean equals(Frazione that) {  
        return  
            this.getNum() * that.getDen()  
            == this.getDen() * that.getNum();  
    }  
}
```

Incapsulamento rispettato anche per this

Java

~C#

Overloading di funzioni

OVERLOADING

- Già sappiamo che in Java e C# possono esistere *costruttori omonimi*, purché distinguibili dalla lista degli argomenti
- Il caso dei costruttori non è l'unico: una classe può contenere *funzioni omonime*, purché distinguibili dalla lista argomenti

– NB: il tipo di ritorno non distingue da solo due signature

Questa possibilità si chiama **OVERLOADING**

- Obiettivo: evitare la proliferazione di nomi per operazioni "molto simili" (tipicamente, varianti della stessa operazione)

Esempio: incremento del valore del contatore di 1 o di K

- Perché usare nomi diversi, come `inc1()` e `incK(int k)`, per quella che è sostanzialmente la stessa azione "incremento"?
- ANZI: se è la stessa, è bene che il nome sia lo stesso!

OVERLOADING: ESEMPIO IN Java

Un `Counter` con due metodi di incremento

```
public class Counter {  
    ...  
    public void inc() { this.value++; }  
    public void inc(int k) {this.value += k; }  
}
```

Metodo `inc` senza argomenti

Java

~C#

Metodo `inc` con un argomento (intero)

Questo invece sarebbe errato:

```
public int    getValue() { return this.value; }  
public String getValue() { return ""+this.value; }
```

Non distinguibili dalla lista argomenti



OVERLOADING: ESEMPIO IN Scala e Kotlin

var perché deve poter essere modificato

In Scala non esiste ++

Scala

```
class Counter(var value : Int) {  
  def inc() : Unit = { this.value +=1 ; }  
  def inc(k : Int) : Unit = {this.value += k; }  
  def equals(that : Counter) : Boolean = { return this.value == that.value; }  
}
```

var perché deve poter essere modificato

MA così tutti potranno modificarlo!
Implementazione troppo naif

Kotlin

```
class Counter(var value:Int) {  
  fun equals(that : Counter) : Boolean { return this.value == that.value; }  
  fun inc() : Unit { this.value ++ ; }  
  fun inc(k : Int) : Unit {this.value += k; }  
}
```



OVERLOADING: ESEMPIO IN Scala e Kotlin

E infatti.. ☹️

```
class Counter (var value : Int){  
  def inc() : Unit = { value +=1; }  
}
```

Scala

```
object MyMain {  
  def main(args: Array[String]) : Unit = {  
    var c1 : Counter = new Counter(7);  
    var c2 : Counter = new Counter(10);  
    println("c1 = " + c1.value + ", c2 = " + c2.value);  
    c1.inc(); c1.inc(); c1.inc();  
    println("c1 = " + c1.value + ", c2 = " + c2.value);  
    c1.value = 18;  
    println("c1 = " + c1.value + ", c2 = " + c2.value);  
  }  
}
```

```
public fun main(args: Array<String>) {  
  var c1 : Counter = Counter(7);  
  var c2 : Counter = Counter(10);  
  println("c1 = " + c1.value + ", c2 = " + c2.value);  
  c1.inc(); c1.inc(); c1.inc();  
  println("c1 = " + c1.value + ", c2 = " + c2.value);  
  c1.value = 18;  
  println("c1 = " + c1.value + ", c2 = " + c2.value);  
}
```

Kotlin



OVERLOADING: ESEMPIO IN Scala e Kotlin

Per impedire accessi indesiderati a `value` basta etichettarlo **private**

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

Scala

```
object MyMain {  
  def main(args: Array[String]) : Unit = {  
    var c1 : Counter = new Counter(7);  
    var c2 : Counter = new Counter(10);  
    println("c1 = " + c1.getValue() + ", c2 = " + c2.getValue());  
    c1.inc(); c1.inc(); c1.inc();  
    println("c1 = " + c1.getValue() + ", c2 = " + c2.getValue());  
    c1.value = 18;  
  }  
}
```

✗ variable value in class Counter cannot be accessed as a member of Counter from object MyMain



OVERLOADING: ESEMPIO IN Scala e Kotlin

Per impedire accessi indesiderati a `value` basta etichettarlo **private**

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

Kotlin

```
! public fun main(args: Array<String>) {  
    var c1 : Counter = Counter(7);  
    var c2 : Counter = Counter(10);  
    println("c1 = " + c1.getValue() + ", c2 = " + c2.getValue());  
    c1.inc(); c1.inc(); c1.inc();  
    println("c1 = " + c1.getValue() + ", c2 = " + c2.getValue());  
    ! c1.value = 18;
```

! Cannot access 'value': it is private in 'Counter'