



Alma Mater Studiorum-Università di Bologna Scuola di Ingegneria

Polimorfismo & equals: un problema

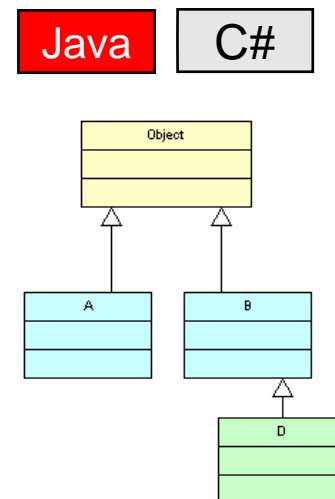
Corso di Laurea in Ingegneria Informatica
Anno accademico 2021/2022

Prof. ENRICO DENTI

Dipartimento di Informatica – Scienza e Ingegneria (DISI)

IL METODO "PREDEFINITO" `equals`

- Abbiamo visto che la relazione di ereditarietà determina la nascita di *gerarchie (tassonomie)* di ereditarietà.
- In particolare, in Java e C# **ogni classe deriva implicitamente da `Object`**
 - in Scala e Kotlin, da `Any`
- `Object` dichiara alcuni metodi, *ereditati* quindi *da ogni altra classe*:
 - `public String toString()`
 - `public boolean equals(Object that)`
 - `public int hashCode()`
 - `protected void clone()`



Concentriamoci su questa



equals: UN PROBLEMINO

- A suo tempo, definimmo un metodo `equals` in `Counter` e `Frazione` per incapsulare il *nostro personale criterio* di uguaglianza:

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

Java

~C#

~Scala

~Kotlin

- Tali definizioni sembrarono all'epoca del tutto logiche, ma ora è il caso di confrontarle con quella di `Object`.



equals: UN PROBLEMINO

- La equals di Counter:

```
public boolean equals(Counter that) {  
    return (this.val==that.val);  
}
```

- La equals di Frazione:

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

- La equals di Object:

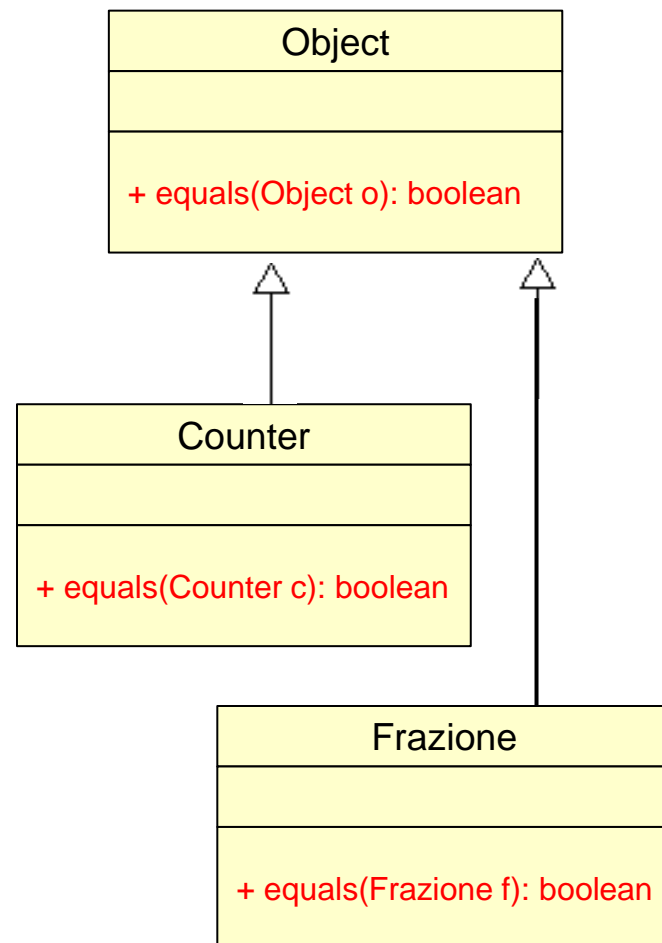
```
public boolean equals(Object obj) {  
    return this==obj;  
}
```

MA.. sono tre
signature diverse!

*Non può essere la
stessa funzione!*

MA QUANTE `equals` CI SONO ?

- La diversa lista di argomenti nella signature dimostra che *non è la stessa funzione*
 - la `equals` definita in `Counter` o `Frazione` non ha sostituito quella ereditata da `Object`
 - si è *affiancata* ad essa
- È *overloading*, non *overriding*
 - `Counter` e `Frazione` contengono in realtà *due diverse equals* (distinguibili dal tipo dell'argomento)
 - quella definita da loro (più specifica)
 - quella ereditata da `Object` (più generale)





UN PRIMO ESPERIMENTO (1/5)

Per capire cosa può succedere, consideriamo il seguente frammento di codice:

```
Counter c1 = new Counter(13);  
Counter c2 = new Counter(13);  
Counter c3 = new Counter(8);  
  
System.out.println(c1.equals(c2)); // true  
System.out.println(c2.equals(c1)); // true  
System.out.println(c1.equals(c3)); // false  
System.out.println(c3.equals(c1)); // false
```

Java

~C#

- con la nostra `equals` «classica» (con argomento `Counter`), fin qui tutto procede normalmente – l'output è quello atteso

```
true  
true  
false  
false
```



UN PRIMO ESPERIMENTO (2/5)

Ora però cambiamo il tipo nominale del riferimento con cui si punta a **c1** e ripetiamo il test:

```
Object obj = c1;  
System.out.println(obj.equals(c2)); // false  
System.out.println(c2.equals(obj)); // false
```

Java

~C#

Errore: il test che prima aveva successo, ora fallisce!! ☹ ☹

```
false  
false
```

Cosa è accaduto?? Non doveva pensarci il polimorfismo?

In realtà, il polimorfismo sta facendo il suo mestiere:
la colpa è proprio di quella **equals** «duplicata»



UN PRIMO ESPERIMENTO (3/5)

INTERPRETAZIONE

Nel primo caso, il tipo degli oggetti è ovunque **Counter**

- quindi, il target della frase **c1.equals(c2)** (e viceversa) è sicuramente un oggetto *di classe Counter*
- è in quella classe che viene cercata una **equals** «adatta»
- **dato che l'argomento è anch'esso un Counter**, la **equals** adatta è quella *con argomento Counter*, ossia **la nostra** → tutto funziona

```
Counter c1 = new Counter(13);  
Counter c2 = new Counter(13);  
  
System.out.println(c1.equals(c2)); // true  
System.out.println(c2.equals(c1)); // true
```

Java

~C#

UN PRIMO ESPERIMENTO (4/5)

INTERPRETAZIONE

Nel secondo caso, il tipo di uno dei due oggetti è **Object**

- nella frase **obj.equals(c2)** stavolta il target è un **Object** mentre l'argomento resta un **Counter**
- nella frase duale **c2.equals(obj)** il target resta un **Counter** mentre è l'argomento che diventa un **Object**

```
Object obj = c1;  
System.out.println(obj.equals(c2)); // false  
System.out.println(c2.equals(obj)); // false
```

Java

~C#

- nel primo caso una **equals** «adatta» viene cercata **nella classe Object**, dove l'unica possibile è quella ereditata
- il polimorfismo ne cerca una versione più specifica in **Counter** (il tipo effettivo dell'oggetto target **obj**) *ma non la trova*, quindi usa quella ereditata → non funziona!



UN PRIMO ESPERIMENTO (5/5)

INTERPRETAZIONE

Nel secondo caso, il tipo di uno dei due oggetti è **Object**

- nella frase **obj.equals(c2)** stavolta il target è un **Object** mentre l'argomento resta un **Counter**
- nella frase duale **c2.equals(obj)** il target resta un **Counter** mentre è l'argomento che diventa un **Object**

```
Object obj = c1;  
System.out.println(obj.equals(c2)); // false  
System.out.println(c2.equals(obj)); // false
```

Java

~C#

- nel secondo caso una **equals** «adatta» viene cercata **nella classe Counter** (il tipo di **c2**) dove ce ne sono due
- per scegliere la «più adatta», la risoluzione dell'overloading guarda il tipo nominale dell'argomento, che è un **Object**
→ sceglie quella «sbagliata»! → non funziona



OVERLOADING equals

- MORALE: avere due diverse `equals` dentro a `Counter` e `Frazione` *non è affatto una situazione desiderabile*
 - noi ci aspettiamo che venga eseguita la nostra..
 - ma l'altra *non è cancellata né disattivata*, è ancora là sotto...
 - .. e può apparire "a tradimento" nei momenti più inaspettati!
- Dentro `Frazione` ci sono *due diverse equals*

```
public boolean equals(Frazione that)  
public boolean equals(Object obj)
```
- Anche dentro `Counter` ci sono *due diverse equals*

```
public boolean equals(Counter that)  
public boolean equals(Object obj)
```

Java

~C#

~Scala

~Kotlin

LE «VERE» equals

- La «vera» `equals` è quella definita al top level

- Java, C#: nella classe `Object`

```
public boolean equals(Object obj)
```

Java

```
public bool Equals(Object obj)
```

C#

- Scala, Kotlin: nella classe `Any`

```
def equals(other: Any): Boolean
```

Scala

```
fun equals(other: Any?): Boolean
```

Kotlin

- Per personalizzare le nostre classi in modo polimorfo, dobbiamo **sovrascriverla**, non affiancargliene un'altra!
 - C#, Scala, Kotlin: ciò è reso esplicito dalla keyword `override`, obbligatoria in tutte le ridefinizioni
 - in Java c'è invece solo una annotation `@Override`, facoltativa



OVERLOADING equals

POSSIBILI PROBLEMI

- Perché non ce ne siamo mai accorti?
 - perché abbiamo sempre chiamato **equals** *in modo omogeneo*, ossia fra due frazioni o fra due counter
f1.equals(f2) **c1.equals(c2)**
 - in tali casi veniva sempre scelta quella più adatta, cioè la nostra: l'altra non entrava mai in gioco, quindi non ne abbiamo mai percepito la sotterranea presenza
 - MA la **equals** ereditata da **Object** non è stata disattivata: è ancora là sotto, che aspetta di saltar fuori "a tradimento" .. ☹
- Come e quando emerge?
 - *in tutte le situazioni polimorfe*, dove l'unica cosa che si sappia dell'argomento è che sia «un qualche oggetto»
 - in Java: l'**assertEquals** di JUnit, che infatti *non funzionava!*



OVERLOADING equals

PROBLEMI IN SCALA & KOTLIN

- In Scala e Kotlin, va anche peggio!
 - a differenza di Java, **Scala e Kotlin considerano l'operatore == come una scorciatoia per la chiamata di equals**
f1.equals(f2) ↔ c1 == c2
 - ma ciò si riferisce alla «giusta» equals, non alla nostra «mal fatta»!!

E infatti, provandoli, i due danno risultati diversi!! ☹☹

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

```
true  
false
```

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

```
true  
false
```

MOTIVO: la definizione di == si basa sulla «giusta» equals che noi non abbiamo in realtà modificato!



OVERLOADING `equals` IN JAVA E C#

UN ALTRO ESPERIMENTO (1/4)

Tempo fa avevamo scritto una funzione (statica) `idem` per confrontare due array (prima di `int`, poi di `Counter`)

```
public static boolean idem( int[] a, int[] b ){  
    if (a.length != b.length) return false;  
    for (int i=0; i<a.length; i++){  
        if (a[i] != b[i]) return false;  
    }  
    return true;  
}
```

Java

~C#

```
public static boolean idem( Counter[] a, Counter[] b){  
    if (a.length != b.length) return false;  
    for (int i=0; i<a.length; i++){  
        if (!a[i].equals(b[i])) return false;  
    }  
    return true;  
}
```

Java

~C#

Si basa sulla *`equals`* di *`Counter`*



OVERLOADING equals IN JAVA E C#

UN ALTRO ESPERIMENTO (2/4)

Il corrispondente `main` era:

```
System.out.println(  
    idem(new Counter[]{  
        new Counter(2), new Counter(3), new Counter(4) },  
        new Counter[]{  
            new Counter(2), new Counter(3) }  
    )  
);
```

Java

~C#

Analogamente si può fare per un array di **Frazione**:

```
System.out.println(  
    idem(  
        new Frazione[]{ new Frazione(2,4), new Frazione(3,5) },  
        new Frazione[]{ new Frazione(3,6), new Frazione(6,10) }  
    )  
);
```




UN ALTRO ESPERIMENTO (3/4)

con `idem(Counter[], Counter[])`

```
System.out.println(idem(  
    new Counter[]{new Counter(2),new Counter(3),new Counter(4)},  
    new Counter[]{new Counter(2),new Counter(3)}  
));
```

Java

~C#

```
System.out.println(idem(  
    new Counter[]{new Counter(2),new Counter(3),new Counter(4)},  
    new Counter[]{new Counter(2),new Counter(3),new Counter(5)}  
));
```

```
System.out.println(idem(  
    new Counter[]{new Counter(2),new Counter(3),new Counter(4)},  
    new Counter[]{new Counter(2),new Counter(3),new Counter(4)}  
));
```

false
false
true

Poiché scatta *la idem specifica per i Counter*,
tutto va bene, come previsto nei tre casi.



UNA FUNZIONE “QUASI” GENERICA

- In realtà, la logica di funzionamento di `idem` è *indipendente dallo specifico tipo dell'array*
- Perciò, ha senso chiedersi se non si possa scriverne una *versione generica valida per ogni tipo*
- La risposta è sì: visto che tutto deriva da `Object`, basta *definire gli argomenti come array di `Object`* !

```
public static boolean idem(Object[] a, Object[] b) {  
    if (a.length != b.length) return false;  
    for (int i=0; i<a.length; i++){  
        if (!a[i].equals(b[i])) return false;  
    }  
    return true;  
}
```

Java

~C#

Grazie al polimorfismo, il confronto avviene in base alla `equals` dell'effettivo tipo dell'argomento `a[i]`



UNA FUNZIONE “QUASI” GENERICA

- Infatti,
 - nominalmente, `a[i]` è un `Object`
 - concretamente, `a[i]` può essere un `Counter`, una `Frazione`...
 - il polimorfismo attiva quindi *la `equals` definita nella classe `Counter` o `Frazione`*
- Peccato che in quelle classi ci siano *due `equals`*
 - quella definita nella classe `Counter` o `Frazione`
`public boolean equals(Frazione that)`
 - quella ereditata da `Object`
`public boolean equals(Object obj)`
 - come sempre in presenza di overloading, viene scelta la funzione con signature più simile → ATTENZIONE...



PECCATO CHE.. NON FUNZIONI !

```
public static boolean idem(Object[] a, Object[] b) {  
    if (a.length != b.length) return false;  
    for (int i=0; i<a.length; i++){  
        if (!a[i].equals(b[i])) return false;  
    }  
    return true;  
}
```

Gli argomenti di `idem` sono **nominalmente** array di `Object`:
quindi, anche ogni cella `a[i]` **nominalmente** è un `Object`

- Il polimorfismo sceglie sempre *la **equals** più adatta agli argomenti ricevuti, ma per quel che ne sa lui gli argomenti sono tutti (array di) **Object***
- Conclusione: basandosi su quel che sa, il polimorfismo ***finisce per scegliere la **equals** sbagliata!*** ☹️☹️
 - MOTIVO: quella ereditata da `Object` ha il tipo dell'argomento in effetti *più simile* all'argomento ricevuto, che è un `Object`

PECCATO CHE.. NON FUNZIONI !

```
public static boolean idem(Object[] a, Object[] b) {  
    if (a.length != b.length) return false;  
    for (int i=0; i<a.length; i++){  
        if (!a[i].equals(b[i])) return false;  
    }  
    return true;  
}
```

Java

~C#

Gli argomenti di `idem` sono **nominalmente** array di `Object`:
quindi, anche ogni cella `a[i]` **nominalmente** è un `Object`

RIPETIAMO L' ESPERIMENTO:

```
System.out.println(idem(  
    new Counter[]{new Counter(2), new Counter(3), new Counter(4)},  
    new Counter[]{new Counter(2), new Counter(3), new Counter(4)}  
));
```

false
false
false

INFATTI, ripetendo l'esperimento
con la `idem` generica, *il terzo caso
fallisce!* ☹ ☹ ☹



IL NOCCIOLO DEL PROBLEMA

```
public static boolean idem(Object[] a, Object[] b) {  
    if (a.length != b.length) return false;  
    for (int i=0; i<a.length; i++){  
        if (!a[i].equals(b[i])) return false;  
    }  
    return true;  
}
```

Java

~C#

IL PROBLEMA è che la **equals** chiamata *non è la nostra*

- è quella originale di **Object**, che ha come argomento *un Object* e quindi è a tutti gli effetti *una funzione diversa*
- **MOTIVO: **b[i]** formalmente è un **Object** (non un **Counter**)** *anche se l'argomento **effettivo** è un **Counter***
- ergo, nulla funziona come ci aspettiamo.



REFACTORING

Correggere la equals di Counter

- Occorre che la nostra `equals` in `Counter` *rimpiazzi quella ereditata da `Object`* anziché affiancarla.
- A tal fine occorre che abbia *la stessa signature*.

Java

~C#

Vecchia versione (errata)

```
public boolean equals(Counter that)  
    return this.val==that.val;  
}
```

Signature DIVERSA dalla
`equals` ereditata da `Object`
→ non la sovrascrive ☹

Nuova versione (corretta)

```
public boolean equals(Object that)  
    return val==((Counter) that).val;  
}
```

Signature IDENTICA alla
`equals` ereditata da `Object`
→ la sovrascrive ☺

Serve un CAST, perché `Object`
non ha un campo `val`

..ma il CAST di Java (C#, etc.) è
sicuro e verificato a run time ☺



UN ALTRO ESPERIMENTO (4/4)

con `idem(Object[], Object[])`

```
System.out.println(idem(  
    new Counter[]{new Counter(2),new Counter(3),new Counter(4)},  
    new Counter[]{new Counter(2),new Counter(3)}  
));
```

Java

~C#

```
System.out.println(idem(  
    new Counter[]{new Counter(2),new Counter(3),new Counter(4)},  
    new Counter[]{new Counter(2),new Counter(3),new Counter(5)}  
));
```

```
System.out.println(idem(  
    new Counter[]{new Counter(2),new Counter(3),new Counter(4)},  
    new Counter[]{new Counter(2),new Counter(3), new Counter(4)}  
));
```

false
false
true

FINALMENTE FUNZIONA !
Counter ridefinisce la `equals` ereditata da
Object, *specializzandola per il suo caso.*



CORREGGERE equals per Frazione

- Anche qui occorre che la nostra equals *rimpiazzi quella ereditata da Object* anziché affiancarla → *stessa signature*

Vecchia versione (errata)

Java

~C#

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

Nuova versione (corretta)

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



RIASSUNTO: LE GIUSTE equals

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

true
false

Java

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

True
False

In Java e C#, == è solo
un confronto di identità

C#

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

true
true

Kotlin

In Scala & Kotlin, invece,
equals è sinonimo di ==

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

true
true

Scala



UN PUNTO DI ATTENZIONE

- OCCHIO, però: poiché queste `equals` contengono un cast, *è cruciale passare loro il «giusto» tipo di oggetto*, non un `Object` qualsiasi (*altrimenti, esplosione!*)

Java

~C#

```
public boolean equals(Object obj) {  
    Frazione that = (Frazione)obj;  
    return (this.num*that.den == this.den*that.num) ;  
}  
  
public boolean equals(Counter obj) {  
    Counter that = (Counter)obj;  
    return this.val == that.val;  
}
```



L'ESPERIMENTO CON *idem* : SCELTE OPERATIVE

- Operativamente, bisogna stabilire *dove mettere* le varie funzioni statiche *idem*
- Quelle specifiche per **Counter** o **Frazione** potrebbero essere messe:
 - in una libreria ad hoc → **IdemLib**, **Holder**, etc.
 - nella stessa classe del main → **Main**, **Test**, etc.
 - **nelle rispettive classi Counter o Frazione**
- La *idem* generica, con argomenti **Object[]**, invece può essere messa solo:
 - **in una libreria ad hoc** → **IdemLib**, **Holder**, etc.
 - nella stessa classe del main → **Main**, **Test**, etc.

Ovviamente, le chiamate nel main vanno adeguate alla scelta che si fa



L'ESPERIMENTO CON `idem` : SCELTE OPERATIVE

- Ipotesi:
 - quelle specifiche per `Counter` o `Frazione` nelle rispettive classi
 - la `idem` generica in una libreria ad hoc → classe di utilità `Holder`
- In Java e C#:
 - le varie `idem` hanno la forma di *funzioni statiche* nelle varie classi
- In Scala e Kotlin:
 - le varie `idem` hanno la forma di *funzioni in un singleton object*
 - se tale singleton si chiama come la classe, ne costituisce *l'oggetto compagno (companion object)* che gode di visibilità reciproca
 - in Scala, il companion object deve avere *lo stesso nome della classe* ma resta definito *esternamente ad essa*
 - in Kotlin, esso è invece definito *dentro alla classe* di cui è il compagno, tramite l'apposita **keyword companion**

L'ESPERIMENTO IN C#

C#

```
class Frazione{
    int num, den;
    public Frazione(int num, int den) { this.num=num; this.den=den; }
    public int GetNum() { return num; }
    public int GetDen() { return den; }
    /*
    public bool Equals(Frazione that){
        return (this.num*that.den == this.den*that.num);
    }
    */
    public override bool Equals(Object obj){
        Frazione that = (Frazione)obj;
        return (this.num*that.den == this.den*that.num);
    }
    public static bool idem( Frazione[] a, Frazione[] b){
        if (a.Length != b.Length) return false;
        for (int i=0; i<a.Length; i++){
            if (!a[i].Equals(b[i])) return false;
        }
        return true;
    }
}
```

```
class Counter {
    int value;
    public Counter(int value) { this.value=value; }
    public int GetValue() { return value; }
    /*
    public bool Equals(Counter that){
        return (this.value==that.value);
    }
    */
    public override bool Equals(Object obj){
        Counter that = (Counter)obj;
        return (this.value==that.value);
    }

    public static bool idem( Counter[] a, Counter[] b){
        if (a.Length != b.Length) return false;
        for (int i=0; i<a.Length; i++){
            if (!a[i].Equals(b[i])) return false;
        }
        return true;
    }
}
```

Prima con Equals (Counter)
poi con Equals (Object)

```
True
True
True

False
True
False -> True

False
True
False -> True
```

```
class Holder{
    public static bool idem( Object[] a, Object[] b){
        if (a.Length != b.Length) return false;
        for (int i=0; i<a.Length; i++){
            if (!a[i].Equals(b[i])) return false;
        }
        return true;
    }
}
```

```
Console.WriteLine( Counter.idem(new Counter[]{ new Counter(2),new Counter(3),new Counter(4)},
                                new Counter[]{ new Counter(2),new Counter(3)} ));
Console.WriteLine( Counter.idem(new Counter[]{ new Counter(2),new Counter(3),new Counter(4)},
                                new Counter[]{ new Counter(2),new Counter(3),new Counter(4)} ));
Console.WriteLine( Holder.idem( new Counter[]{ new Counter(2),new Counter(3),new Counter(4)},
                                new Counter[]{ new Counter(2),new Counter(3),new Counter(4)} ));

Console.WriteLine( Frazione.idem( new Frazione[]{ new Frazione(2,4) },
                                new Frazione[]{ new Frazione(3,6), new Frazione(6,10)} ));
Console.WriteLine( Frazione.idem( new Frazione[]{ new Frazione(2,4), new Frazione(3,5) },
                                new Frazione[]{ new Frazione(3,6), new Frazione(6,10)} ));
Console.WriteLine( Holder.idem( new Frazione[]{ new Frazione(2,4), new Frazione(3,5) },
                                new Frazione[]{ new Frazione(3,6), new Frazione(6,10)} ));
```

L'ESPERIMENTO IN SCALA

```
class Frazione(val num:Int, val den:Int){
  /*
  def equals(that: Frazione) : Boolean = {
    return (this.num*that.den == this.den*that.num);
  }
  */
  override def equals(obj:Any) : Boolean = {
    var that = obj.asInstanceOf[Frazione];
    return (this.num*that.den == this.den*that.num);
  }
}

object Frazione {
  def idem(a: Array[Frazione], b:Array[Frazione]) : Boolean = {
    if (a.size != b.size) return false;
    for (i <- 0 until a.size){
      if (!a(i).equals(b(i))) return false;
    }
    return true;
  }
}
```

Companion object

```
class Counter(val value:Int) {
  /*
  def equals(that:Counter) : Boolean = {
    return (this.value==that.value);
  }
  */
  override def equals(obj:Any) : Boolean = {
    val that = obj.asInstanceOf[Counter];
    return (this.value==that.value);
  }
}

object Counter {
  def idem(a: Array[Counter], b:Array[Counter]) : Boolean = {
    if (a.size != b.size) return false;
    for (i <- 0 until a.size){
      if (!a(i).equals(b(i)))return false;
    }
    return true;
  }
}
```

Companion object

```
object Holder {
  def idem( a: Array[Any], b:Array[Any]) : Boolean = {
    if (a.size != b.size) return false;
    for (i <- 0 until a.size){
      if (!a(i).equals(b(i))) return false;
    }
    return true;
  }
}
```

Prima con `equals(Counter)`
poi con `equals(Any)`

True
True
True

False
True
False -> True

False
True
False -> True

```
println( Counter.idem(Array(new Counter(2),new Counter(3),new Counter(4)),
  Array(new Counter(2),new Counter(3)) )); //false
println( Counter.idem(Array(new Counter(2),new Counter(3),new Counter(4)),
  Array(new Counter(2),new Counter(3),new Counter(4)) )); // true
println( Holder.idem( Array(new Counter(2),new Counter(3),new Counter(4)),
  Array(new Counter(2),new Counter(3),new Counter(4)) )); // true??? No, FALSE!

println( Frazione.idem( Array(new Frazione(2,4)),
  Array(new Frazione(3,6),new Frazione(6,10)) )); // false
println( Frazione.idem( Array(new Frazione(2,4),new Frazione(3,5)),
  Array(new Frazione(3,6),new Frazione(6,10)) )); // true

println( Holder.idem( Array(new Frazione(2,4),new Frazione(3,5)),
  Array(new Frazione(3,6),new Frazione(6,10)) )); // true?? NO, false!
```

Scala

L'ESPERIMENTO IN KOTLIN

```
class Frazione(val num:Int, val den:Int){
    fun equals(that: Frazione) : Boolean {
        return (this.num*that.den == this.den*that.num);
    }
    /*
    override fun equals(obj:Any?) : Boolean {
        var that = obj as Frazione;
        return (this.num*that.den == this.den*that.num);
    }*/
    companion object {
        public fun idem(a: Array<Frazione>, b:Array<Frazione>) : Boolean {
            if (a.size != b.size) return false;
            for (i in 0..a.size-1){
                if (!a[i].equals(b[i])) return false;
            }
            return true;
        }
    }
}
```

Companion object

```
class Counter(val value:Int) {
    public fun equals(that:Counter) : Boolean {
        return (this.value==that.value);
    }
    /*
    public override fun equals(obj:Any?) : Boolean {
        val that = obj as Counter;
        return (this.value==that.value);
    }*/
    companion object {
        public fun idem( a: Array<Counter>, b:Array<Counter>) : Boolean {
            if (a.size != b.size) return false;
            for (i in 0..a.size-1){
                if (!a[i].equals(b[i])) return false;
            }
            return true;
        }
    }
}
```

Companion object

```
object Holder{
    public fun idem( a: Array<Any>, b:Array<Any>) : Boolean {
        if (a.size != b.size) return false;
        for (i in 0..a.size-1){
            if (!a[i].equals(b[i])) return false;
        }
        return true;
    }
}
```

Prima con `equals (Counter)`
poi con `equals (Any?)`

```
True
True
True

False
True
False -> True

False
True
False -> True
```

```
println( Counter.idem( arrayOf(Counter(2),Counter(3),Counter(4)),
                        arrayOf(Counter(2),Counter(3)) ));           //false
println( Counter.idem( arrayOf(Counter(2),Counter(3),Counter(4)),
                        arrayOf(Counter(2),Counter(3),Counter(4)) )); // true
println( Holder.idem(   arrayOf(Counter(2),Counter(3),Counter(4)),
                        arrayOf(Counter(2),Counter(3),Counter(4)) )); // true??? No, FALSE!

println( Frazione.idem( arrayOf(Frazione(2,4)),
                           arrayOf(Frazione(3,6),Frazione(6,10)) )); // false
println( Frazione.idem( arrayOf(Frazione(2,4),Frazione(3,5)),
                           arrayOf(Frazione(3,6),Frazione(6,10)) )); // true

println( Holder.idem(   arrayOf(Frazione(2,4),Frazione(3,5)),
                        arrayOf(Frazione(3,6),Frazione(6,10)) )); // true?? NO, false!
```

Kotlin



RIASSUNTO & PROBLEMI

- Usando come **tipo formale Object** (Any in Scala e Kotlin) si possono scrivere funzioni e metodi *generici*
 - si parla di *polimorfismo verticale* perché si segue "a scendere" la gerarchia di ereditarietà
- PRO: funziona!
- CONTRO: *equivale ad abolire il controllo di tipo*
 - si potrebbero passare argomenti (o array) di tipi diversissimi e il compilatore non se ne accorgerebbe...
 - .. salvo poi veder esplodere tutto a run time! ☹

UN APPROCCIO NON «TYPE SAFE»



ESPERIMENTO "EXPLOSION"

con `idem(Object[], Object[])`

```
System.out.println(idem(  
    new Counter[]{  
        new Counter(2), new Counter(3), new Counter(4) },  
    new String[]{  
        "Pippo", "Pluto", "Paperino"}  
));
```

Java

~C#

ORRORE! Due array diversissimi!

Però passa tranquillamente la compilazione, perché
sia Counter che String derivano da Object
e quindi la chiamata è formalmente corretta.

Peccato che poi a run time... **DISASTRO !**

```
Exception in thread "main"  
java.lang.ClassCastException:  
java.lang.String cannot be cast to Counter
```



REVISIONE CRITICA di equals

- Problema: la nostra implementazione naif di `equals` **presuppone** di ricevere il «giusto» tipo di oggetto, *ma non c'è modo di garantire che sia davvero così*: se non lo è, *il cast esplode!*

```
public boolean equals(Object obj) {  
    return val == ((Counter) obj).val;  
}
```

Java

~C#

CURA: *condizionare il cast* a una preventiva verifica
→ in Java, con l'*apposito operatore instanceof*

```
public boolean equals(Object obj) {  
    if (obj instanceof Counter)  
        return this.val == ((Counter) obj).val;  
    else return false;  
}
```

Se falso, NON tentiamo neppure di fare il cast: semplicemente, in tal caso i due oggetti sono diversi.



IL CHECK DINAMICO DI TIPO

- Tutti i linguaggi OO offrono un qualche operatore o metodo per verificare a run time il tipo di un'istanza
 - Java: **instanceof**
 - C#: **is**
 - Scala: **isInstanceOf[Tipo]**
 - Kotlin: **as**
- Spesso, alla verifica si può unire la conversione di tipo (cast):
 - Java 16: **instanceof esteso**
 - C#: **is esteso**
 - Scala: **asInstanceOf[Tipo]**
 - Kotlin: *smart cast – il tipo è già convertito dentro l'if che lo verifica*



IL CHECK DINAMICO DI TIPO

```
public boolean equals(Object obj){  
    if (obj instanceof Counter)  
        return this.value==(Counter)obj.value;  
    else return false;  
}
```

Java

```
public override bool Equals(Object obj){  
    if (obj is Counter)  
        return this.value==(Counter)obj.value;  
    else return false;  
}
```

C#

```
override def equals(obj: Any): Boolean = {  
    if (obj.isInstanceOf[Counter])  
        return this.value==obj.asInstanceOf[Counter].value;  
    else return false;  
}
```

Scala

```
override fun equals(obj: Any?): Boolean {  
    if (obj is Counter)  
        return this.value==(obj as Counter).value;  
    else return false;  
}
```

Kotlin

Cast non necessario,
grazie allo *smart cast*

IL CHECK DI TIPO CON CAST

- Alcuni linguaggi offrono una versione *estesa* dell'operatore o un metodo che *include il cast automatico* in caso di esito positivo del test
 - Java 16: *instanceof esteso*
 - C#: *is esteso*
 - Kotlin: *smart cast – il tipo è già convertito dentro l'if che lo verifica*
- In C# e Java 16+ una estensione del costrutto *instanceof / is* permette di *specificare un identificatore da associare all'oggetto già convertito nel «giusto» tipo*

Java: `if(obj instanceof Counter that)...`

C#: `if(obj is Counter that)...`

Se *instanceof* è vera, *that* rappresenta lo stesso oggetto già convertito in tipo (nel solo ramo *true* dell'*if*)



instanceof ESTESO

- Grazie al costrutto esteso, all'interno dell'if non è più necessario alcun cast

```
public boolean equals(Object obj) {  
    if (obj instanceof Counter that)  
        return this.value==that.value;  
    else return false;  
}
```

Java

Questa feature è disponibile da Java 16;
in Java 15 solo con **--enable-preview**

```
public override bool Equals(Object obj) {  
    if (obj is Counter that)  
        return this.value==that.value;  
    else return false;  
}
```

C#

Caratteristica supportata da .NET 5 in poi

In Kotlin il costrutto **is** include sempre automaticamente tale **smart cast**

Kotlin



VERSIONI MIGLIORATE

```
public boolean equals(Object obj){  
    if (obj instanceof Counter that)  
        return this.value==that.value;  
    else return false;  
}
```

Java

```
public override bool Equals(Object obj){  
    if (obj is Counter that)  
        return this.value==that.value;  
    else return false;  
}
```

C#

```
override def equals(obj: Any): Boolean = {  
    if (obj.isInstanceOf[Counter])  
        return this.value==obj.asInstanceOf[Counter].value;  
    else return false;  
}
```

Scala

Scala:
nessuna variazione

```
override fun equals(obj: Any?): Boolean {  
    if (obj is Counter)  
        return this.value==obj.value;  
    else return false;  
}
```

Kotlin



RIPETENDO L'ESPERIMENTO..

```
System.out.println(idem(  
    new Counter[]{  
        new Counter(2), new Counter(3), new Counter(4) },  
    new String[]{  
        "Pippo", "Pluto", "Paperino"}  
));
```

Java

~C#

RIMANE UN ORRORE,
passa ancora tranquillamente la compilazione,
ma almeno non esplode più a run time!

false



LAST BUT NOT LEAST..

```
System.out.println(idem(  
    new String[]{  
        "Pippo", "Pluto", "Paperino"},  
    new Counter[]{  
        new Counter(2), new Counter(3), new Counter(4) }  
));
```

Java

~C#

Invertendo i due array, non esplode perché scatta la `equals` di `String`, *che prevede già questo controllo*

false



RIPRENDENDO IL PRIMO ESPERIMENTO (1/2)

Per verifica, riconsideriamo il primo esperimento:

- con la `equals` «originale» (con argomento `Counter`), i primi due test davano il risultato atteso, mentre gli altri due fallivano:

```
System.out.println(c1.equals(c2)); // true
System.out.println(c2.equals(c1)); // true
Object obj = c1;
System.out.println(obj.equals(c2)); // false
System.out.println(c2.equals(obj)); // false
```

Java

~C#

```
true
true
false
false
```



RIPRENDENDO IL PRIMO ESPERIMENTO (2/2)

Per verifica, riconsideriamo il primo esperimento:

- invece, con la **equals** riformulata con argomento **Object**, che **sovrascrive realmente** quella ereditata, i test hanno ora successo:

```
System.out.println(c1.equals(c2)); // true
System.out.println(c2.equals(c1)); // true
Object obj = c1;
System.out.println(obj.equals(c2)); // true
System.out.println(c2.equals(obj)); // true
```

Java

~C#

```
true
true
true
true
```

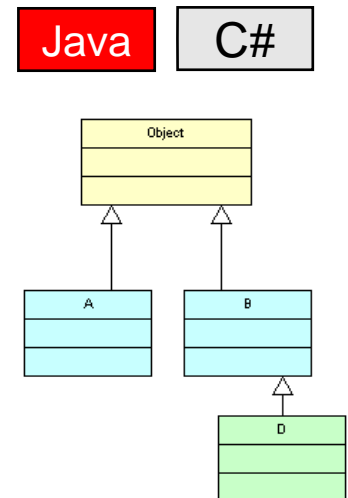
RITORNANDO ALLA RADICE ..

- Se ogni classe deriva implicitamente da `Object`, i metodi lì dichiarati sono automaticamente ereditati *ovunque*
- Principali metodi di `Object` in Java:
 - `public String toString()`
 - `public boolean equals(Object obj)`
 - `public int hashCode()`
 - `protected void clone()`

In realtà, **`equals`** non opera da sola, ma in tandem con un'altra funzione: **`hashCode`**

Perciò, non basta occuparsi di **`equals`**:
bisogna *sempre* occuparsi di *entrambe*

C#: `Equals`, `GetHashCode`





CHI È hashCode?

- Il metodo **hashCode** genera un intero che «identifica il meglio possibile» una data istanza di una data classe
 - si basa su aritmetica modulare secondo algoritmi standardizzati
 - **REGOLA AUREA**: «oggetti uguali secondo equals devono avere anche hashcode uguali»
- È usato internamente
 - per gestire velocemente i test di uguaglianza
 - per fare ricerche in alcune strutture dati (es. HashMap, HashSet)
- Per non avere inconsistenze di comportamento è quindi **essenziale ridefinirla sempre insieme a equals** secondo una *logica unica e coerente*
 - ..ma come si fa?? cosa ci si scrive..?



ESPERIMENTO hashCode

Nelle nostre classi non abbiamo mai considerato **hashCode**
– che valori si ottengono provando per vari **Counter**?

```
Counter c1 = new Counter(13);  
Counter c2 = new Counter(13);  
Counter c3 = new Counter(8);  
  
System.out.println(c1.hashCode());  
System.out.println(c2.hashCode());  
System.out.println(c3.hashCode());
```

Java

~C#

```
1523554304  
1175962212  
918221580
```

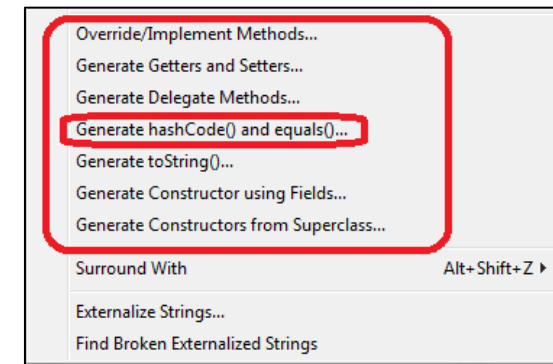
Non va bene: viola la regola aurea!

Perché tutto sia coerente, occorre ridefinire sempre
anche **hashCode** «in tandem» con **equals**



COME RIDEFINIRE hashCode?

- Per ridefinire `hashCode` vi sono varie **ricette**
 - si sfruttano numeri primi
 - si «compongono» i valori dei campi dati «moltiplicandoli e sommandoli» con tali numeri primi, così da ottenere *valori unici*
- Possiamo cercarle sul web...
 - per un valore intero: $prime + valore$
 - per due valori interi: $(prime + valore1) * prime + valore2$
 - per oggetti non primitivi: `oggetto.hashCode()`
- ... oppure **farlo fare a Eclipse**, che ha un'apposita funzionalità di generazione già inclusa 😊 😊



UNA RICETTA PER hashCode

- Una buona ricetta per **hashCode** prevede di usare un numero primo come moltiplicatore di ogni campo dati:

```
@Override
public int hashCode() {
    final int prime = 31;
    int result = 1;
    result = prime * result + ((cognome == null) ? 0 : cognome.hashCode());
    result = prime * result + ((nascita == null) ? 0 : nascita.hashCode());
    result = prime * result + ((nome == null) ? 0 : nome.hashCode());
    return result;
}
```

Java

```
}class Frazione(val num:Int, val den:Int){
    override def equals(obj: Any): Boolean = {
        if (obj.isInstanceOf[Frazione]) {
            var that = obj.asInstanceOf[Frazione];
            return this.num*that.den == this.den*that.num;
        } else return false;
    }

    override def hashCode(): Int = 41 * (41 + num) + den;
}

class Counter(val value:Int){
    override def equals(obj: Any): Boolean = {
        if (obj.isInstanceOf[Counter])
            return this.value==(obj.asInstanceOf[Counter]).value;
        else return false;
    }

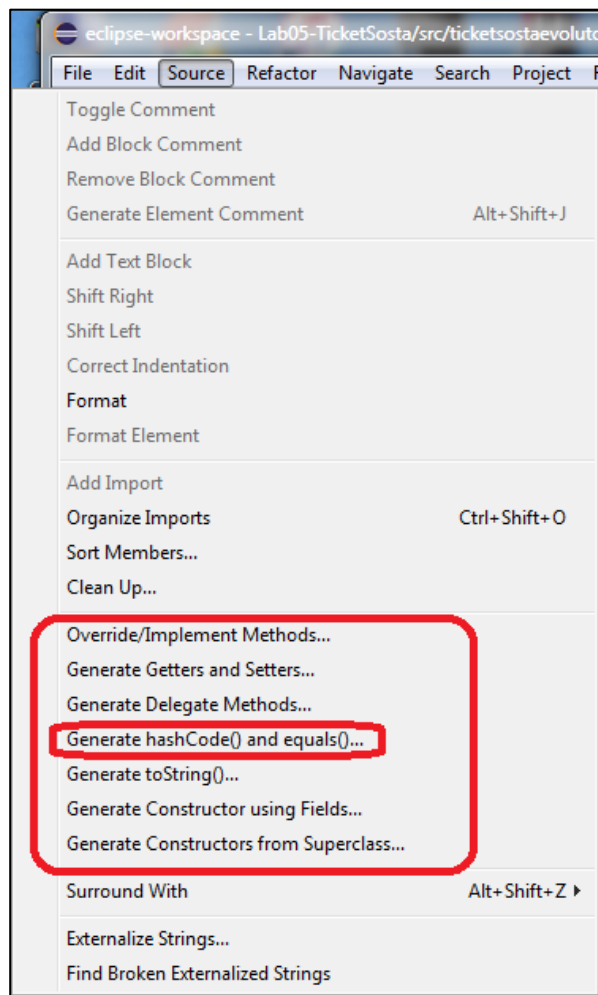
    override def hashCode(): Int = 41 * value;
}
```

Scala

Ma NON lo faremo noi!
In Java lo fa gratis Eclipse ☺
Generate → hashCode & equals

GENERAZIONE AUTOMATICA DI `equals` & `hashCode`

Java



- Eclipse può generare automaticamente la coppia `equals` & `hashCode` *a partire dai campi dati della classe*
 - che, quindi, devono essere già definiti
- NB: se si cambiano / aggiungono dati *dopo aver generato `equals` & `hashCode`, occorre rigenerarle*
- Come vedete, Eclipse può generare automaticamente, *sempre a partire dai campi dati della classe*, tante altre belle cose... 😊



ESEMPIO: hashCode per Counter e Point

- Possibile hashCode per Counter
 - ha un solo valore intero $\rightarrow prime + val$
- Possibile hashCode per Point
 - ha due valori interi $\rightarrow (prime + x) * prime + y$

```
class Counter {  
    private int val;  
    ...  
    public int hashCode() { return 41 + val; }  
}
```

```
class Point {  
    private int x,y;  
    ...  
    public int hashCode() { return (31 + x)*31 + y; }  
}
```



ESPERIMENTO hashCode BIS

Ripetendo l'esperimento precedente con la classe con la nuova ridefinizione di **hashCode** :

```
Counter c1 = new Counter(13);  
Counter c2 = new Counter(13);  
Counter c3 = new Counter(8);  
  
System.out.println(c1.hashCode());  
System.out.println(c2.hashCode());  
System.out.println(c3.hashCode());
```

Java

~C#

54

54

49

Ora la regola aurea è rispettata



hashCode: LINEE GUIDA (1)

- I valori interi si usano (di base) così come sono
- I valori reali vanno in qualche modo «resi interi» estraendo da essi un «intero significativo»
 - non si può arrotondare, altrimenti ne verrebbero molto uguali
 - meglio sfruttare la rappresentazione interna a livello di bit, ottenibile con le funzioni di libreria `Float.floatToIntBits(x)` e `Double.doubleToLongBits(x)`

Esempio: `hashCode`
per un `Point` con
coordinate reali

```
@Override
public int hashCode() {
    final int prime = 31;
    int result = 1;
    long temp;
    temp = Double.doubleToLongBits(x);
    result = prime * result + (int) (temp ^ (temp >>> 32));
    temp = Double.doubleToLongBits(y);
    result = prime * result + (int) (temp ^ (temp >>> 32));
    return result;
}
```



hashCode: LINEE GUIDA (2)

- Per gli altri valori si richiama **hashCode** sui componenti

Esempio: **hashCode**
per una **Persona**
con nome ed età

```
private String nome;  
private int eta;  
  
@Override  
public int hashCode() {  
    final int prime = 31;  
    int result = 1;  
    result = prime * result + eta;  
    result = prime * result + ((nome == null) ? 0 : nome.hashCode());  
    return result;  
}
```

OPPURE

- Si usano per tutti le **funzioni statiche di libreria *hash(...)*** presenti nella classe-utility **Objects**

Esempio: **hashCode**
per un **Point** con
coordinate reali

```
@Override  
public int hashCode() {  
    return Objects.hash(x, y);  
}
```

Esempio: **hashCode**
per **Counter**

```
public class Counter {  
    private int val;  
  
    @Override  
    public int hashCode() {  
        return Objects.hash(val);  
    }  
}
```



PROBLEMA: hashCode per Frazione

- A differenza di **Counter**, **Point** e **Persona**, la **equals** di **Frazione** incorpora un criterio «non banale», basato sul principio di equivalenza
 - una frazione *non* è solo una mera coppia interi
 - la ricetta base considererebbe diversi $3/2$ e $6/4 \rightarrow$ non va bene!
- La relativa **hashCode** dovrà essere **coerente**
 - come farla?
 - POSSIBILE IDEA: ridurre la frazione ai minimi termini e usare la frazione ridotta (che è unica per la classe di equivalenza) per calcolare **hashCode** 😊

```
@Override  
public int hashCode() {  
    Fraz reduced = minTerm();  
    return Objects.hash(reduced.den, reduced.num);  
}
```



REGOLE PER UNA CLASSE BEN FATTA

`equals` & `hashCode`

Una classe ben fatta:

Java

- ridefinisce `equals` con la giusta signature in modo da riflettere un corretto principio di equivalenza fra istanze in Java: `public boolean equals(Object that) {...}`
- ridefinisce coerentemente anche `hashCode`, in modo che *istanze «uguali» abbiano anche hashCode uguali*
 - o lo si fa fare a Eclipse (MEGLIO)
 - oppure si sfruttano le funzioni statiche di libreria `Objects.hash`

C#

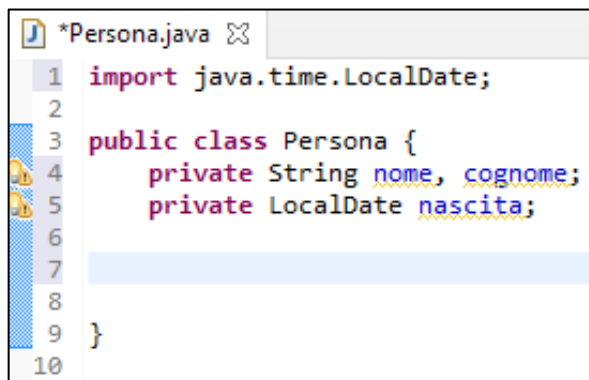
Scala

Kotlin

Minime modifiche

ESEMPIO in ECLIPSE (1/6)

Dallo scheletro della classe Persona in Java:



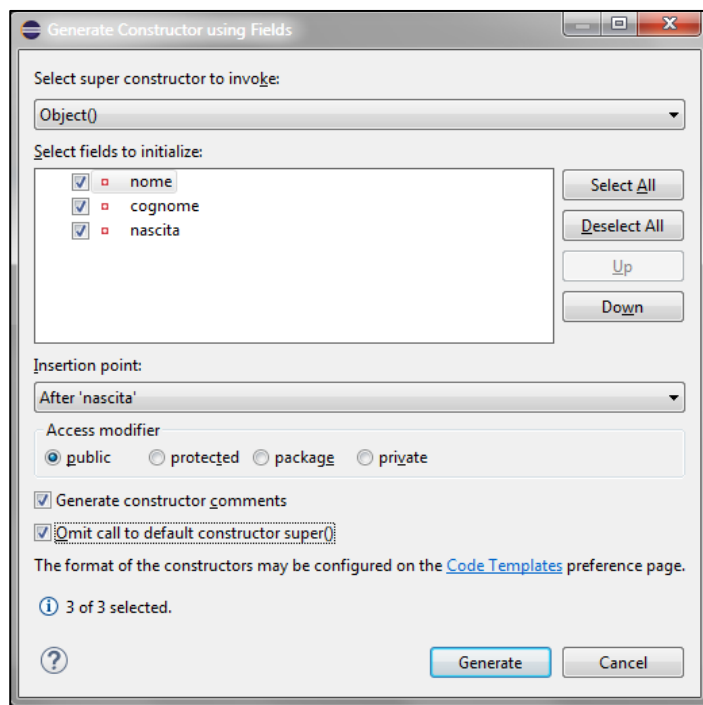
```
*Persona.java ✕
1  import java.time.LocalDate;
2
3  public class Persona {
4      private String nome, cognome;
5      private LocalDate nascita;
6
7
8
9  }
10
```

facciamo generare:

- i costruttori
- gli accessor in lettura (*getters*)
- `toString`
- `equals` & `hashCode` in diversi modi

ESEMPIO in ECLIPSE (2/6)

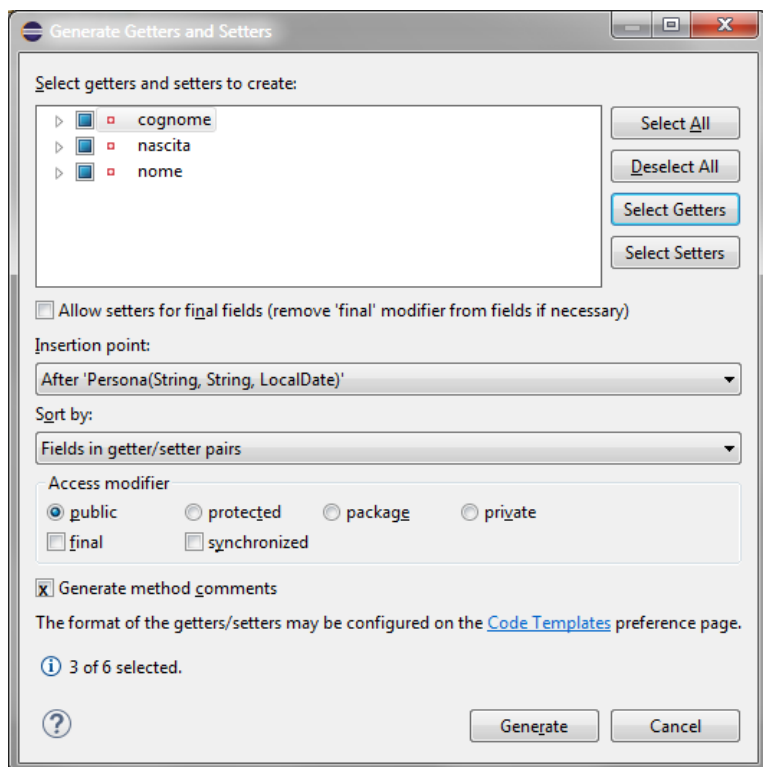
Generazione dei costruttori (con commenti):



```
*Persona.java
1  import java.time.LocalDate;
2
3  public class Persona {
4      private String nome, cognome;
5      private LocalDate nascita;
6
7      /**
8       * @param nome
9       * @param cognome
10      * @param nascita
11      */
12     public Persona(String nome, String cognome, LocalDate nascita) {
13         this.nome = nome;
14         this.cognome = cognome;
15         this.nascita = nascita;
16     }
17
18 }
19
```

ESEMPIO in ECLIPSE (3/6)

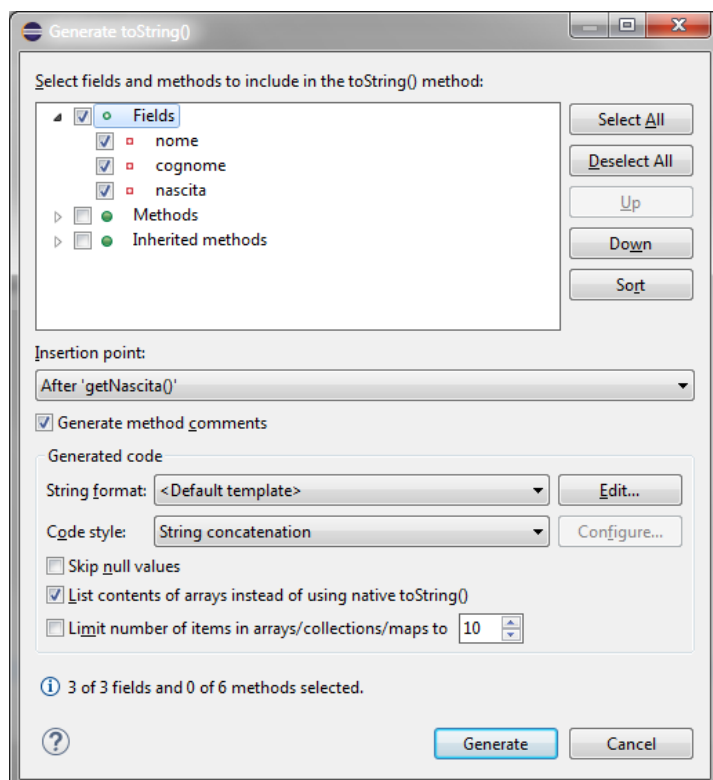
Generazione degli accessor in lettura (*getters*, con commenti):



```
1 import java.time.LocalDate;
2
3 public class Persona {
4     private String nome, cognome;
5     private LocalDate nascita;
6
7     /**
8      * @param nome
9      * @param cognome
10     * @param nascita
11     */
12     public Persona(String nome, String cognome, LocalDate nascita) {
13         this.nome = nome;
14         this.cognome = cognome;
15         this.nascita = nascita;
16     }
17
18     /**
19      * @return the nome
20      */
21     public String getNome() {
22         return nome;
23     }
24
25     /**
26      * @return the cognome
27      */
28     public String getCognome() {
29         return cognome;
30     }
31
32     /**
33      * @return the nascita
34      */
35     public LocalDate getNascita() {
36         return nascita;
37     }
38 }
39
40
```

ESEMPIO in ECLIPSE (4/6)

Generazione di toString

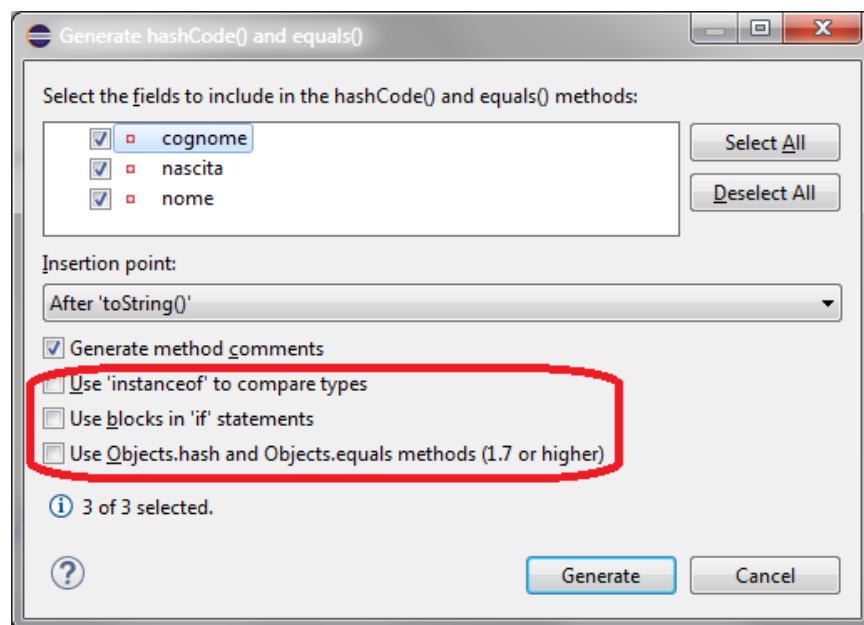


```
@Override
public String toString() {
    return "Persona [nome=" + nome + ", cognome=" + cognome +
        ", nascita=" + nascita + "]";
}
```

ESEMPIO in ECLIPSE (5/6)

Generazione di equals & hashCode

- ci sono diverse opzioni
- senza opzioni, si genera codice per qualunque versione di Java ma *molto verboso*



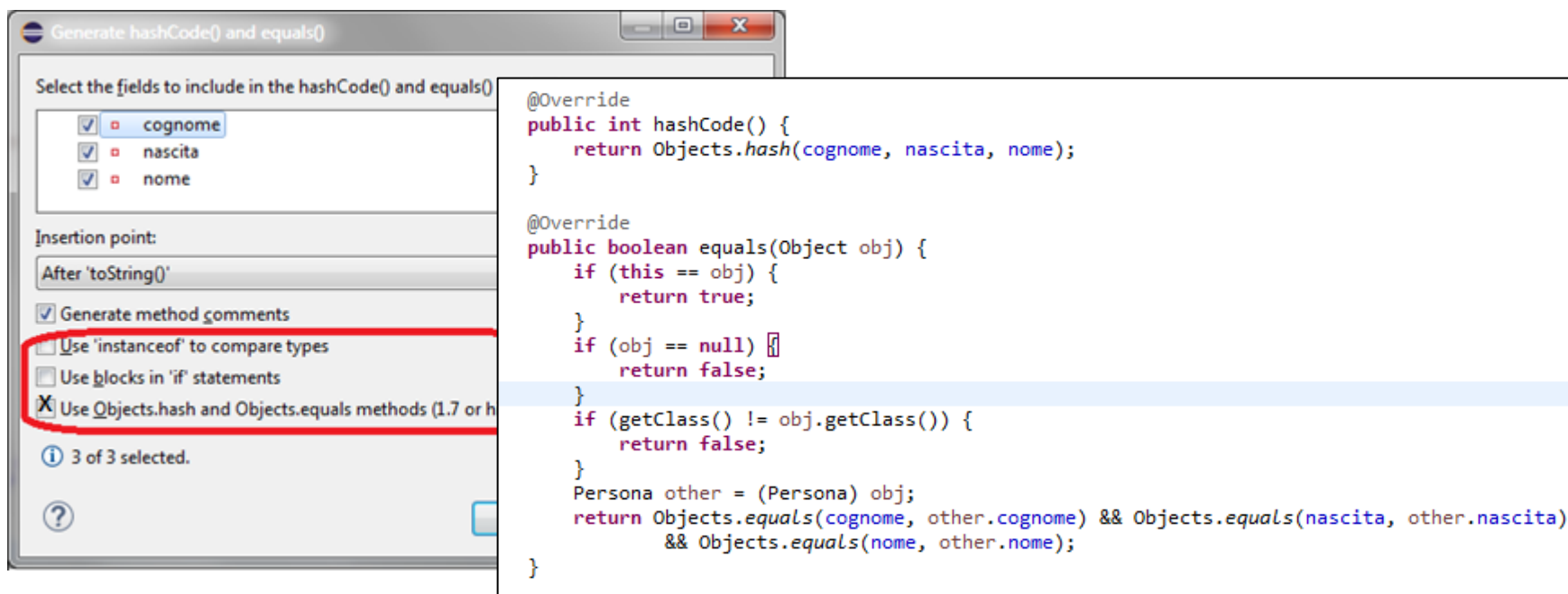
```
@Override
public int hashCode() {
    final int prime = 31;
    int result = 1;
    result = prime * result + ((cognome == null) ? 0 : cognome.hashCode());
    result = prime * result + ((nascita == null) ? 0 : nascita.hashCode());
    result = prime * result + ((nome == null) ? 0 : nome.hashCode());
    return result;
}

@Override
public boolean equals(Object obj) {
    if (this == obj)
        return true;
    if (obj == null)
        return false;
    if (getClass() != obj.getClass())
        return false;
    Persona other = (Persona) obj;
    if (cognome == null) {
        if (other.cognome != null)
            return false;
    } else if (!cognome.equals(other.cognome))
        return false;
    if (nascita == null) {
        if (other.nascita != null)
            return false;
    } else if (!nascita.equals(other.nascita))
        return false;
    if (nome == null) {
        if (other.nome != null)
            return false;
    } else if (!nome.equals(other.nome))
        return false;
    return true;
}
```

ESEMPIO in ECLIPSE (6/6)

Generazione di equals & hashCode

- selezionando la terza opzione, si genera codice molto più conciso e chiaro, usando metodi di libreria disponibili da Java 7



The screenshot shows the Eclipse IDE with the 'Generate hashCode() and equals()' dialog box open. The dialog has a section 'Select the fields to include in the hashCode() and equals()' with three checked items: `cognome`, `nascita`, and `nome`. Below this, the 'Insertion point' is set to 'After \'toString()\'' and 'Generate method comments' is checked. At the bottom, three options are listed: 'Use \'instanceof\' to compare types' (unchecked), 'Use blocks in \'if\' statements' (unchecked), and 'Use `Objects.hash` and `Objects.equals` methods (1.7 or h...)' (checked, highlighted with a red box). A status bar at the bottom of the dialog indicates '3 of 3 selected'.

The resulting code is shown in a separate window, displaying the generated `hashCode()` and `equals()` methods. The `hashCode()` method uses `Objects.hash(cognome, nascita, nome)`. The `equals()` method uses `Objects.equals()` to compare the fields, which is more concise and clearer than using `instanceof` and `String.equals()`.

```
@Override
public int hashCode() {
    return Objects.hash(cognome, nascita, nome);
}

@Override
public boolean equals(Object obj) {
    if (this == obj) {
        return true;
    }
    if (obj == null) {
        return false;
    }
    if (getClass() != obj.getClass()) {
        return false;
    }
    Persona other = (Persona) obj;
    return Objects.equals(cognome, other.cognome) && Objects.equals(nascita, other.nascita)
        && Objects.equals(nome, other.nome);
}
```

hashCode IN ALTRI LINGUAGGI

ESEMPIO: Scala

- secondo la ricetta, si può ad esempio moltiplicare via via per un numero primo (es. 41), sommando i contributi dei vari campi uno dopo l'altro

```
class Frazione(val num:Int, val den:Int){  
  override def equals(obj: Any): Boolean = {  
    if (obj.isInstanceOf[Frazione]) {  
      var that = obj.asInstanceOf[Frazione];  
      return this.num*that.den == this.den*that.num;  
    } else return false;  
  }  
  override def hashCode(): Int = 41 * (41 + num) + den;  
}  
  
class Counter(val value:Int){  
  override def equals(obj: Any): Boolean = {  
    if (obj.isInstanceOf[Counter])  
      return this.value==(obj.asInstanceOf[Counter]).value;  
    else return false;  
  }  
  override def hashCode(): Int = 41 * value;  
}
```

```
20 object Foo{  
21   def main(args:Array[String]) :Unit = {  
22     val c1= new Counter(12)  
23     val c2= new Counter(14)  
24     val c3= new Counter(12)  
25     print(c1.equals(c2)); print(", "); println(c1==c2)  
26     print(c2.equals(c3)); print(", "); println(c2==c3)  
27     print(c1.equals(c3)); print(", "); println(c1==c3)  
28     var myCset = new scala.collection.mutable.HashSet[Counter](); myCset += c1;  
29     println(myCset.contains(c3))  
30     val f1 = new Frazione(3,4);  
31     val f2 = new Frazione(3,5);  
32     val f3 = new Frazione(3,4);  
33     print(f1.equals(f2)); print(", "); println(f1==f2)  
34     print(f2.equals(f3)); print(", "); println(f2==f3)  
35     print(f1.equals(f3)); print(", "); println(f1==f3)  
36     var myFset = new scala.collection.mutable.HashSet[Frazione](); myFset += f1;  
37     println(myFset.contains(f3))  
38   }  
39 }
```

Scala

```
false, false  
false, false  
true, true  
true  
false, false  
false, false  
true, true  
true
```

OSSERVA:

- senza hashCode, **contains** fallisce;
- con hashCode, ha successo



hashCode IN ALTRI LINGUAGGI

ESEMPIO: Kotlin (main analogo alla versione Scala, non riportato)

```
class Frazione(val num:Int, val den:Int){
    ! override fun equals(obj: Any?): Boolean {
        if (obj is Frazione) {
            ! val that = obj as Frazione;
            return this.num*that.den == this.den*that.num;
        } else return false;
    }
    //override fun hashCode(): Int = 41 * (41 + num) + den;
}

class Counter(val value:Int){
    ! override fun equals(obj: Any?): Boolean {
        if (obj is Counter)
            ! return this.value==(obj as Counter).value;
        else return false;
    }
    //override fun hashCode(): Int = 41 * value;
}
```

```
false, false
false, false
true, true
false
false, false
false, false
true, true
false
```

**Senza hashCode,
contains fallisce**

```
class Frazione(val num:Int, val den:Int){
    ! override fun equals(obj: Any?): Boolean {
        if (obj is Frazione) {
            ! val that = obj as Frazione;
            return this.num*that.den == this.den*that.num;
        } else return false;
    }
    override fun hashCode(): Int = 41 * (41 + num) + den;
}

class Counter(val value:Int){
    ! override fun equals(obj: Any?): Boolean {
        if (obj is Counter)
            ! return this.value==(obj as Counter).value;
        else return false;
    }
    override fun hashCode(): Int = 41 * value;
}
```

Kotlin

```
false, false
false, false
true, true
true
false, false
false, false
true, true
true
```

**Con hashCode,
contains ha successo**