# Appunti di Programmazione Concorrente e Distribuita

(A.A 2018/2019)

Grigoras Valentin

Matricola: 1099561

# Contenuti

# Elenco delle figure

# 1 Metodo Equals()

L'operatore binario == viene utilizzato per stabilire se il contenuto di due variabili è identico. Per i tipi non primitivi viene confrontato il contenuto dei puntatori degli oggetti, ovvero l'indirizzo di memoria degli oggetti a cui i puntatori fanno riferimento. Spesso però potremmo avere oggetti che nonostante puntino ad indirizzi di memoria differenti possono essere considerati a tutti gli effetti uguali. La procedura standard per confrontare oggetti in Java è il metodo**equals()**, definito in **Object**, quindi è disponibile su tutti gli oggetti (ma non sui tipi primitivi):

```
/*segnatura*/
public boolean equals(Object x)

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

Il confronto viene fatto con l'operatore ==, dato che non è possibile conoscere in Object come confrontare qualsiasi oggetto Java. Per effettuare un confronto migliore non basato sugli indirizzi di memoria, è buona norma effettuare l'overriding del metodo equals in ogni classe che sviluppiamo.

## 1.1 Proprietà del metodo Equals()

Il linguaggio Java richiede che qualunque ridefinizione del metodo equals() rispetti determinate proprietà che forniscono una vera e propria **relazione di equivalenza**:

- **Riflessività**; per ogni oggetto x, avremo che x.equals(x) ritorna *true*;
- **Simmetria**: dati due oggetti x ed y, avremo che se x.equals(y) ritorna true, allora y.equals(x) ritorna *true*;
- **Transitività**: dati tre oggetti x, y e z, se x.equals(y) ritorna true e y.equals(z) ritorna true, allora x.equals(z) ritorna *true*.

## 1.2 Corretta implementazione di Equals()

Si devono eseguire 3 step nella classe più un quarto di verifica all'esterno:

1. Usare == per vedere se l'argomento è uguale a this;

   ```
   if(this == obj) return true;
   ```

2. Verificare se l'oggetto passato al metodo è del tipo desiderato, effettuando un controllo con l'operatore *instanceof*, in caso negativo restituiremo false;

   ```
   if(!(obj istanceof ClassName)) return false;
   ```

3. Essendo sicuri di avere un oggetto del tipo desiderato, potremo effettuare in sicurezza un downcast al tipo desiderato. In questo modo possiamo accedere ai campi e metodi prima nascosti a causa del riferimento di tipo Object utilizzato;

```
NomeClasse cp = (NomeClasse)ogj;
return cp.campo1.equals(this.campo1) &&
 cp.campoN.equals(this.campoN);
```

4. Confrontare se stiamo rispettando la riflessibilità, simmetria e transitività.

```
public class ColorPoint {
	private Point point;
	private Color color;

	public ColorPoint(...) {...}

	public Point asPoint() {
		return new Point(point.getX(), point.getY());
	}

	@Override
	public boolean equals(Object obj) {
		if(this == obj) return true;
		if(!(obj instanceof ColorPoint)) return false;
		ColorPoint cp = (ColorPoint)obj;
		return cp.point.equals(point) &&
			cp.color.equals(color);
	}
}
```

Figura 1: Implementazione corretta di equals

## 1.3 Binding dinamico di Equals()

Quando si effettua l'overriding di equals, è sconsigliato cambiare il tipo del parametro di equals Object , altrimenti non avremo un overriding ma un overloading. Questo cambiamento potrebbe provocare disastri, vediamo con un esempio in cui cambiando il tipo del parametro di equals cosa potrebbe accadere:

```
public boolean equals(Employee e){}
```

Inoltre supponeremo di avere:

```
Emplyee e1 = new Employee(...);
Emplyee e2 = new Employee(...);
Object o1 = e1;
Object o2= e1;
```

Analizziamo cosa succede ad ogni riga con l'utilizzo del Binding dinamico:

- **o1.equals(o2);** Le firme candidate sono cercate solo in Object perché il tipo dichiarato di o1 è Object. Di conseguenza non verrà considerata la specializzazione del metodo equals effettuata;

- **o1.equals(e2);** Nella fase 2 del Binding dinamico non c'è overriding , quindi verrà preso l'unico metodo equals esattamente uguale che sarà quello contenuto in Object. Anche in questo caso non verrà considerata la specializzazione del metodo equals che abbiamo effettuato;
- **e1.equals(o1);** Viene valutato ma Object non è assegnabile a Employee. Di conseguenza non verrà considerata la specializzazione del metodo equals effettuata;
- **e1.equals(e2);** La firma combacia con quella specializzata e sarà l'unico caso in cui verrà chiamato il "giusto" equals.

### 1.4 Metodo Equals() ed eredità

Quando il metodo equals viene sovrascritto bisogna prestare attenzione al funzionamento tra il metodo equals definito e il suo funzionamento nelle classi sottostanti. In definitiva bisogna decidere come si deve comportare il metodo equals con le sue sottoclassi. In particolare, bisogna porsi due domande:

1. **Il criterio di confronto tra oggetti di una sottoclasse è diverso da quello che vale tra oggetti della superclasse?**
   - No, allora bisogna definire equals della superclasse final in modo che nessuna sottoclasse possa sovrascriverlo;
   - Si, allora il metodo equals deve essere opportunamente ridefinito in ogni sottoclasse in cui il criterio di confronto è differente.
2. **Un oggetto di una sottoclasse può essere considerato uguale ad un oggetto di una superclasse?**
   - No, allora bisogna verificare di ammettere il confronto solo tra agli oggetti del tipo giusto;
   - Si, allora se gli oggetti di una sottoclasse possono essere considerati uguali a quelli di una superclasse dovremo ammettere al confronto tutti gli oggetti di quel tipo e tutte le sue sottoclassi.

### 1.5 Tipi primitivi vs oggetti

Si sente dire spesso l'affermazione *in Java tutto è un oggetto.* In realtà questa frase non è del tutto vera in quanto esistono i tipi primitivi che permettono di utilizzare numeri, caratteri e valori booleani senza ricorrere agli oggetti. I tipi primitivi sono riconoscibili anche perché iniziano con una lettera minuscola (mentre i nomi delle classi iniziano sempre con una maiuscola). I tipi primitivi disponibili in Java sono i seguenti:

- **boolean:** valore booleano, può assumere i valori true e false;
- **byte:** numero intero a 8 bit;
- **short:** numero intero a 16 bit;
- **int:** numero intero a 32 bit;
- **long:** numero intero a 64 bit;
- **float:** numero reale a 32 bit in virgola mobile (IEEE 754-1985);

- **double:** numero reale a 64 bit in virgola mobile (IEEE 754-1985);
- **char:** carattere unicode a 16 bit;

Gli oggetti in Java sono istanze di una classe. Una delle differenze importanti fra tipi primitivi e oggetti si ha se guardiamo la situazione della memoria a basso livello. Infatti nella locazione di memoria corrispondente a un tipo primitivo è presente il valore mentre in quella di un oggetto è presente il puntatore all'area di memoria che contiene l'oggetto. Un diagramma spiega meglio questa differenza:

```
int i1 = 5;
int i2 = 7;
Persona p1 = new Persona("Mario", "Rossi");
Persona p2 = new Persona("Mario", "Verdi");
```

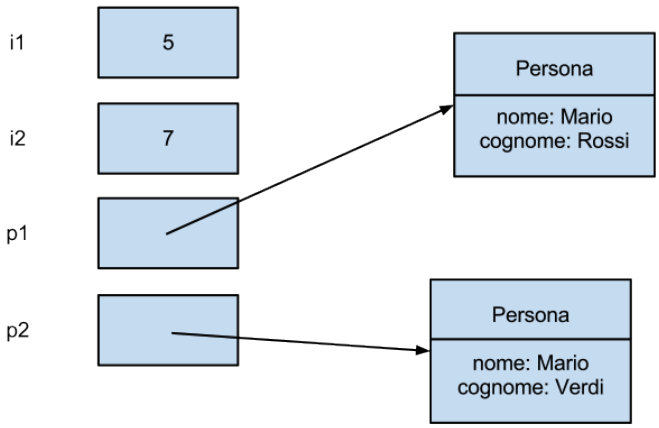


Figura 2: tipi primitivi e oggetti: rappresentazione in memoria

Fin qui sembra tutto semplice, andiamo avanti facendo un po' di assegnazioni:

```
i2 = i1;
p2 = p1;
```

A questo punto quale è la situazione? Ovviamente i1 e i2 assumo lo stesso valore, stessa cosa anche per p1 e p2. Ma se andiamo a vedere cosa è successo in memoria la situazione è un po' più complicata:

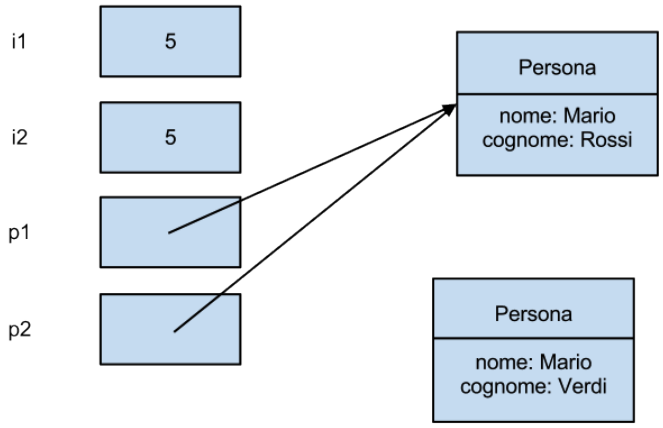


Figura 3: tipi primitivi e oggetti: rappresentazione in memoria dopo assegnazione

Adesso p1 e p2 puntano allo stesso oggetto: abbiamo una condivisione di memoria che può risultare pericolosa se non gestita adeguatamente. Infatti richiamando un metodo setter su uno dei due oggetti (per esempio p1.setNome("Fabio")) si modificherà l'oggetto condiviso dai due puntatori.

## 1.6 Equals() Vs ==

Iniziamo con l'operatore ==: invocando questo costrutto viene confrontato il valore contenuto nella variabile. Quindi, nel caso di tipi primitivi viene confrontato il valore vero, mentre nel caso di oggetti viene confrontato l'indirizzo di memoria a cui i puntatori fanno riferimento. Vediamo un esempio concreto. Il seguente codice stampa sulla console 4 volte il valore true:

```
int i1 = 5;
int i2 = 7;

System.out.println(i1 != i2);
i2 = i1;
System.out.println(i1 == i2);

Persona p1 = new Persona("Mario", "Rossi");
Persona p2 = new Persona("Mario", "Verdi");

System.out.println(p1 != p2);
p2 = p1;
System.out.println(p1 == p2);
```

p1 e p2 puntano allo stesso oggetto e quindi confrontando l'indirizzo di memoria corrispondente al puntatore si ha un esito positivo. Vediamo un altro esempio leggermente più complicato. Definiamo due volte lo stesso valore/oggetto e confrontiamolo con ==:

```
int i1 = 5;
int i2 = 5;

System.out.println(i1 == i2);
```

```
Persona p1 = new Persona("Mario", "Rossi");
Persona p2 = new Persona("Mario", "Rossi");

System.out.println(p1 == p2);
```

In questo caso la prima condizione sui tipi primitivi è vera mentre quella sugli oggetti è falsa! Il motivo è chiaro, le variabili p1 e p2 in questo caso puntano a locazioni di memoria diverse quindi confrontando gli indirizzi di memoria si avrà un risultato negativo. Per ovviare a questo problema è necessario usare il metodo equals che, a differenza dell'operatore ==, confronta gli oggetti puntati entrando quindi in merito al contenuto dell'oggetto. Riproviamo quindi a eseguire lo stesso codice usando il metodo equals per confrontare i due oggetti:

```
Persona p1 = new Persona("Mario", "Rossi");
Persona p2 = new Persona("Mario", "Rossi");

System.out.println(p1.equals(p2));
```

Eseguendo questo codice viene stampato sulla console ancora false! Come mai? Il motivo è che il metodo equals è definito dentro la classe Object con la seguente implementazione:

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

L'implementazione di Object del metodo equals utilizza l'operatore ==! Questa scelta può sembrare strana ma è l'unica che poteva essere fatta. infatti la classe Object è la classe padre di tutte le classi Java, non è possibile dentro questa classe sapere come confrontare qualunque oggetto Java!

## 2 Operatore istanceof

È un operatore che consente di determinare a run-time il tipo di un oggetto. Restituisce true o false a seconda che l'oggetto sia o no un'istanza della classe di confronto o di una delle sue superclassi. L'operatore si applica anche alle interfacce. Se un oggetto implementa un'interfaccia allora instanceof con il nome di quell'interfaccia restituisce true. La sintassi prevede la seguente dichiarazione:

```
<riferimento> instanceof <tipo_riferimento>
```

Esempi:

```
/*esempio 1*/
if ("prova" instanceof String) ... // restituisce true

/*esempio 2*/
Point pt = new Point(3,5);
if (pt instanceof String) ... // restituisce false
```

# 3 Il metodo String toString()

Si tratta di un metodo che restituisce una rappresentazione dell'oggetto di invocazione in formato stringa. L'implementazione di default restituisce una stringa contenente il nome della classe e l'indirizzo di riferimento di invocazione:

```
<classe>@<hashcode>
```

dove *classe* è il nome della classe dell'oggetto su cui il metodo è invocato, e *hashcode* è la rappresentazione esadecimale del codice hash dell'oggetto (indirizzo in memoria dell'oggetto). Questo accade perché la classe Object non può conoscere la struttura dell'oggetto. Invocazioni implicite del metodo *toString* sono inoltre inserite dal compilatore in ogni contesto dove sia usato un riferimento e si richiede invece un valore di tipo String, ad esempio il codice:

```
Data d = new Data();
System.out.println(d);
```

è tradotto dal compilatore in *System.out.println(d.toString());* chiamando cioè sul riferimento d il metodo toString() che traduce l'oggetto di tipo Data in una stringa, che può essere passata come parametro al metodo *println*.

## 3.1 Costruzione stringhe

Dato il seguente codice :

```
String nome= new String(''Roberto''); /*1*/
String nome = ''Roberto''; /*2*/
```

è importante ricordare che l'operazione più efficiente a lungo andare è la seconda. Nel primo caso ho un reference èerché alloco nello heap un oggetto, nel secondo caso NON ho un reference e costruisco la stringa cheè costante e so dove si trova in memoria.

## 3.2 Concatenazione stringhe

Dato il seguente codice:

```
String nomi = '' '';
for(var p : people)
nomi += p;

var nomi= new StringBuilder('' '');
for(var p : people)
nomi.append(p);
```

è importante ricordare che il tipo String è immutabile e ogni volta che si fa il += viene allocata una nuova stringa, quindi chiamare += spesso causa un continuo ri-alloco di stringhe. Lo StringBuilderinvece viene allocato una volta sola (al momento di creazione, col new) e poi con l'append si aggiunge testo in coda senza le riallocazioni (come accadeva prima).

# 4 Le classi wrapper

Sono dette **classi wrapper** (involucro) le classi che fanno da contenitore ad un tipo di dato primitivo, astraendo proprio il concetto di tipo primitivo. Le classi wrapper sono le seguenti: *Byte*, *Short*, *Integer*, *Long*, *Float*, *Double*, *Booleab*, *Character*, ognuna delle quali può contenere il relativo tipo primitvo. Tutte le class wrapper apparte la Boolean sono sottoclasssi di *Number* Come la classe *String*, anche le classi wrapper sono immutabili. Questo implica che un oggetto istanziato da una classe wrapper non potrà mai cambiare il suo valore interno tramite un suo metodo, anche perchè non esistono metodi setter. Quindi un oggetto istanziato cosi:

```
Boolean b = new Boolean(true);
```

non conterrà mai il valore false. Le classi wrapper sono utili soprattutto nei casi in cui dobbiamo utilizzare un tipo di dato primitivo laddove è richiesto un oggetto:

```
/*Errore! I tipi generici richiedono tipi parametro
solo complessi e non primitivi*/
ArrayList<int> list = new ArrayList<>();

/*sfrutta la regola dell'autoboxing-autounboxing*/
ArrayList<Integer> list ) = new ArrayList<>();
```

## 4.1 Autoboxing - Autounboxing

**Autoboxing**: conversione automatica (fatta dal compilatore) dei tipi primitivi nei rispettivi wrapper; **Autounboxing**: conversione automatica (fatta dal compilatore) dei tipi wrapper nei rispettivi tipi primitivi;

Esempio:

```
/* legale perche c é una coincidenza perfetta tra tipo
primitivo e wrapper*/
Double d = 2.2;

Double d = 2;  /*illegale perché 2 é intero*/
```

## 4.2 Cose da evitare

Il seguente codice compila ma è inefficiente:

```
Long sum = 0L;
var max = Integer.MAX_VALUE;
for(long i = 0; i < max; ++i)
   sum += i;
```

Il tipo di sum è Long, ovvrero un wrapper di long. Nel for invece ho longche è un tipo primitivo e quindi la chiamata a sum += i;effettua il boxing del tipo, quindi crea conversioni di continuo che impattano sulla performance del programma.

Soluzione: usare long sum;.

## 5 Il metodo Clone()

Il metodo *clone()* restituisce un nuovo oggetto il cui stato iniziale è una copia dell'oggetto su cui viene invocato.

Se si vuole che sia possibile copiare gli oggetti di una classe, la classe deve implementare l'interfaccia *Cloneable*. Il metodo clone() che viene ereditato dalla classe Object controlla prima di tutto che la classe dell'oggetto implementi l'interfaccia Cloneable e in caso contrario lancia l'eccezione *CloneNotSupportedException*, altrimenti crea un nuovo oggetto e lo inizializza con una copia degli attributi dell'oggetto originale; al termine restituisce un riferimento al nuovo oggetto.

Per garantire che il metodo Clone() non sia invocato accidentalmente, il metodo è dichiarato protected. Essendo dichiarato protected, per essere reso disponibile per l'invocazione nelle nostre classi bisognerà sottoporlo a override cambiandone il modificatore a public, come nel seguente esempio:

```
public class Quadrato implements Cloneable {
...
   public Object clone() {
      try {
         Object ogg = super.clone();
         return ogg;
      }
      catch(CloneNotSupportedException e ){
         return null;
      }
  }

}
```

È importante notare che la copia che viene fatta dal metodo *Clone()* della classe Object, si limita a copiare i valori delle variabili d'istanza dell'oggetto (si tratta di shalllow copy o copia superficiale).Quindi se le variabili d'istanza sono reference ad altri oggetti, allora verranno copiati i loro indirizzi, e non il contenuto del reference creandosi cosi una condivisione di memoria.

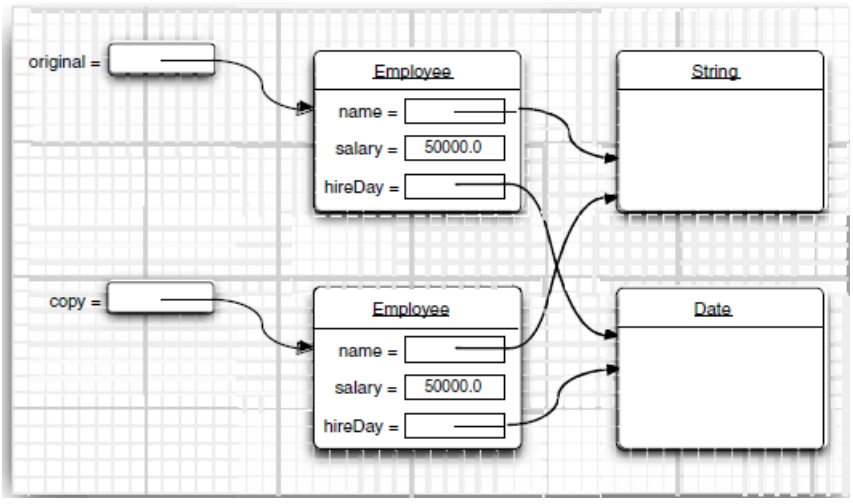


Figura 4: Shallow copy vs Deep copy

## 5.1 Clone() con copia profonda

```
public class Stack implements Cloneable {
   private Object[] elements;

   @Override public Stack clone() {
        try {
            Stack result = (Stack)super.clone();
            result.elements = elements.clone();
        }catch(CloneNotSupportedException cnsex)
            throw new AssertionError();
        }
        return results;
   }
}
```

Figura 5: Deep copy Clone()

1. chiamare il metodo clone della superclasse;
2. castare il risultato al tipo stesso della classe;
3. si fa una copia profonda dei campiche hanno delle reference mutabili. Notare che su elements è stato chiamato clone()perché è il metodo che fa una copia profonda di un array di Objectma volendo si sarebbe potuto fare manualmente un for e fare deep copy del contenuto dell'array.;
4. i ritorna il risultato.

La *clone()* comunque potrebbe essere un problema se un campo è final, in generale è meglio avere un metodo di cloning separato e non implementare Cloneable.Ad esempio, si può usare un costruttore di copia oppure un factory method, ovvero un metodo che costruisce per me l'oggetto. Esempio di factory:

```
public static Stack factoryNew(Stack source) {
Stack dest = new Stack(...);
   //faccio copie deep o altro prendendo i dati da source
   return dest;
}
```

L'idea del factory è: creo un metodo da usare tipo Stack copia = Stack.factoryNew(...)che ritorna un oggetto di tipo Stack che è una copia esatta dell'oggetto che passoin input.

# 6 La keyword final

La keyword *final* si può utilizzare in contesti diversi ma in generale identifica qualcosa di costante o immutabile. Il reference this, che in un metodo non statico si riferi-

sce all'oggetto di invocazione è sempre implicitamente *final*, ovvero non modificabile. Invece l'oggetto a cui this si riferisce è modificabile:

```
public class C {
   public int x;
   public void metodo(){
      this = new C(); /* ILLEGALE: cannot assign a
      value to final variable this*/
      this.x = 10; //ok
   }
}
```

## 6.1 Campi dati final

Un campo dati statico o no marcato *final* è un campo costante / immutabile, quindi ad ogni tentativo di modifica il compilatore segna un errore. I campi dati marcati final devono essere sempre marcati esplicitamente, non basta l'inizializzazione automatica:

```
public class E {
final int i; // ILLEGALE
}
public class E {
static final int k; // ILLEGALE
static final int v=0; //ok
}
public class E {
final int i;
E(){i=0;} // ok
}
public class E {
final int i;
E(){i=0; i=2;} /* ILLEGALE, non posso modificare
il valore di i dopo avergli assegnato 0*/
}
```

Se un riferimento *ref* viene marcato *final*, allora ref si riferisce ad uno ed un solo oggetto fissato, ma il contenuto dell'oggetto puntato può essere modificato. Java non fornisce un modo per rendere costanti gli oggetti:

```
class Z{
   int i=2;
}

public class D{
   Z z1 = new Z();
   final Z z2 = new Z();
   static final Z z3 = new Z();
   final int[] a = {1, 2, 3, 4, 5};

   public static void main(String[] args){
   D d1 = new D();
```

```
    d1.z2.i++; //ok
    d1.z1 = new Z();
    for(int i = 0; i < d1.a.lenght; i++){
    d1.a[i]++; /* ok, solamente il reference a é costante
    d1.z2 = new Z(); ILLEGALE, z2 é final
    D.z3 = new Z(); ILLEGALE, z3 é final
    d1.a = new int[3]; ILLEGALE, a é final
    }
  }
}
```

### 6.2 Metodi final

Un metodo marcato come *final* indica che tale metodo non può essere ridefinito in eventuali sottoclassi. I motivi per marcare un metodo final sono:

- **Sicurezza**: si può pensare che un metodo *boolean validatePassword()* può essere marcato final per evitare evenutali ridefinizioni maliziose;
- **Efficienza**;
- **Progettuali**: a volte si desidera mettere un blocco sul metodo per evitare che una sottoclasse cambi comportamento.

```
public class Classe{
    //attributi
    public final void MetodoNonRidefinibile(){
        //codice del metodo non ridefinibile
    }
    //codice
}
```

### 6.3 Classi final

Marcare una classe C come final significa che non possono essere definiti sottoclassi di C. Naturalmente, tutti i metodi di una classe final sono implicitamente final, mentre i campi dati possono o meno essere marcati final. Si noti che marcare una classe final si opera una notevole restrizione a suo utilizzo in quanto non sarebbe permesso l'estendibilità di quella classe.

## 7 Classe immutabile

Una classe è *immutabile* se lo stato di un oggetto di quella classe (detto anche oggetto immutabile) non può essere modificato dopo che è stato creato. Per esempio *String* è una classe immutabile e una volta istanziata con il suo valore non cambia più.

Prima infatti abbiamo visto che la concatenazione col += di stringhe non è molto efficiente perché le stringhe, essendo immutabili,non si possono modificare e quindi il += crea una nuova stringa ogni volta.

Una classe immutabile generalmente si crea se non ha alcun membro esposto e non ha alcun setter; diciamo che è in "read only mode" e quindi in quanto tale è anche thread safe. Più avanti vedremo che i problemi avvengono quando più processi vogliono simultaneamente scrivere su una risorsa condivisa ma se tale risorsa è di sola lettura, ognuno può leggere quando vuole e quanto vuole.

```
public final class Immutable {
    private final int value;
    public Immutable(int value) {
        this.value = value;
    }
    public int getValue() {
        return value;
    }
}
```

## 7.1 Regole per creare una classe immutabile

1. Non fornire alcun *mutatore* tipo setter;
2. Rendere la classe *final* o comunque impedire l'estensione al di fuori del package di appartenenza;
3. Rendere tutti i campi dati e i metodi *final*;
4. Mettere tutti i campi dati *privati*;
5. Ogni variabile membro che fa riferimento a oggetti mutabili (come ad esempio array, collezioni o la classe java.util.Date):
   - va resa privata;
   - non deve mai essere restituita all'esterno se non attraverso una sua copia;
   - deve avere reference solo all'interno della classe; quindi nel caso in cui la variabile membro sia inizializzata nel costruttore con un oggetto mutabile, va memorizza una copia dell'oggetto al posto dell'oggetto originale;
   - non deve essere variata dopo la costruzione dell'oggetto.

## 7.2 Esempio pratico

Esempio di classe mutabile:

```
package it.cosenonjaviste.classiimmutabili.mutabile;

import java.text.DecimalFormat;
import java.util.EnumSet;
import java.util.Set;

public class Pizza {

  public enum Formato {
    NORMALE, BABY, MAXI, CALZONE;
```

```
}

public enum Ingrediente {
  PROSCIUTTO, FUNGHI, PATATE, SALSICCIA, CARCIOFI, SPECK;
}

protected Formato formato;
protected Set ingredienti = EnumSet.noneOf(Ingrediente.class);

public Formato getFormato() {
  return formato;
}

public void setFormato(Formato formato) {
  this.formato = formato;
}

public void aggiungiIngrediente(Ingrediente i) {
  ingredienti.add(i);
}

public void rimuoviIngrediente(Ingrediente i) {
  ingredienti.add(i);
}

public Set getIngredienti() {
  return ingredienti;
}

public double getPrezzo() {
  double prezzo = 0.0;
  switch (formato) {
  case BABY:
    prezzo += 2.0;
    break;
  case NORMALE:
  case CALZONE:
    prezzo += 4.0;
    break;
  case MAXI:
    prezzo += 7.0;
    break;
  }
  prezzo += ingredienti.size() * 1.0;
  return prezzo;
}

public String toString() {
  return formato + " " + ingredienti + " " +
```

```
    new DecimalFormat("#,##").format(getPrezzo());
  }

}

/*nel main*/
Pizza prosciutto = new Pizza();
Pizzeria.pizze.add(prosciutto);
prosciutto.setFormato(Formato.CALZONE);
prosciutto.aggiungiIngrediente(Ingrediente.PROSCIUTTO);
System.out.println("Ho segnato l'ordine di " + prosciutto);
```

La stessa classe in versione immutabile:

```
public final class Pizza {

  public static final Pizza MARGHERITA =
   new Pizza(Formato.NORMALE, EnumSet.noneOf(Ingrediente.class));
  public static final Pizza PROSCIUTTO =
   new Pizza(Formato.NORMALE, EnumSet.of(Ingrediente.PROSCIUTTO));
  public static final Pizza CALZONE_PROSCIUTTO =
   new Pizza(Formato.CALZONE, EnumSet.of(Ingrediente.PROSCIUTTO));

  public enum Formato {
    NORMALE, BABY, MAXI, CALZONE
  }

  public enum Ingrediente {
    PROSCIUTTO, FUNGHI, PATATE, SALSICCIA, CARCIOFI, SPECK
  }

  private final Formato formato;
  private final Set<Ingrediente> ingredienti;
  private Double prezzo;
  private String toString;

  public Formato getFormato() {
    return formato;
  }

  public Set<Ingrediente> getIngredienti() {
    return EnumSet.copyOf(ingredienti);
  }

  public Pizza(Formato formato, Set<Ingrediente> ingredienti) {
    this.formato = formato;
    this.ingredienti = EnumSet.copyOf(ingredienti);
  }

  public double getPrezzo() {
```

```
    if (prezzo == null) {
      prezzo = 0.0;
      switch (formato) {
      case BABY:
        prezzo += 2.0;
        break;
      case NORMALE:
      case CALZONE:
        prezzo += 4.0;
        break;
      case MAXI:
        prezzo += 7.0;
        break;
      }
      prezzo += ingredienti.size() * 1.0;
    }
    return prezzo;
  }

  public String toString() {
    if (toString == null) {
      toString = formato + " " + ingredienti + " " +
       new DecimalFormat("#,##").format(getPrezzo());
    }
    return toString;
  }
```

Esempio sbagliato:

```
public class MyDateTime{
   private Date dateTime;
   private TimeZome tz;

   public MyDateTime(Date dateTime, TimeZone tz){
      this.dateTime = dateTime;
      this.tz = tz;
   }

   public Date getDateTime() {return dateTime;}
}
```

Non va bene qua perché sto inizializzando due classi (dateTime e tz) semplicemente facendo una copia di reference; non sto facendo una copia profonda ma sto solo copiando il puntatore. Inoltre sto ritornando una reference ad un campo oggetto privato della classe! In realtà, già solo con il costruttore sono in pericolo perché mi basta modificare l'oggetto da fuori la classe per modificarlo anche dentro dato che non ho fatto una copia profonda.

## 7.3 Vantaggi

- Nessun thread può corrompere lo stato interno di una classe immutabile;

- Le classi immutabili siano più semplici da programmare poiché le loro istanze possono assumere uno e un solo stato che viene mantenuto dalla costruzione in poi.

## 7.4 Svantaggi

- Dato che un oggetto immutabile non può essere variato, se si vuole effettuare una modifica sarà necessario costruire un nuovo oggetto che differisce dal primo solo per la variazione voluta;

## 7.5 Differenza tra classe immutabile e oggetto immutabile

Una **classe immutabile** genera oggetti immutabili per definizione, mentre un **oggetto immutabile** non necessariamente è instanziato da una classe anch'essa immutabile. Esempi di oggetti immutabili:

```
public final class Immutable2 {

        private final String text;

        public Immutable2(String text) {
            this.text = text;
        }

        public String getText() {
            return this.text;
        }
    }
```

```
public final class Immutable3 {

        private final String text;

        private Immutable3(String text) {
            this.text = text;
        }

        public String getText() {
            return this.text;
        }

        public static Immutable3 create(String text) {
            return new Immutable3(text);
        }
    }
```

## 7.6 Regole per creare oggetti immutabili

1. Tutte le proprietà devono essere settabili nel costrutture o in un metodo statico di init (vedi create sopra);

2. Nessun setter, se necessari per qualche motivo dovrebbero sollevare un eccezione;
3. tutte le proprietà devono essere marcate private e final;
4. La classe deve essere marcata final;
5. Caso in cui si riferenzi oggetti mutabili: usare deep copy per passare i valori al costruttore e per leggerli attraverso metodi di getter.

### 7.7 Perchè la classe String è immutabile

- **Sicurezza**: i parametri sono in genere rappresentati come String nelle connessioni di rete, url di connessione al database, nomi utente / password ecc. Se fosse mutabile, questi parametri potrebbero essere facilmente modificati;
- **Sincronizzazione e concorrenza**: rendere String immutable li rende automaticamente thread-safe, risolvendo così i problemi di sincronizzazione;
- **Caching**: quando il compilatore ottimizza gli oggetti String, vede che se due oggetti hanno lo stesso valore (a = "test" e b = "test") e quindi è necessario un solo oggetto stringa (per entrambi aeb, questi due punti allo stesso oggetto);
- **Caricamento della classe**: String viene utilizzato come argomento per il caricamento della classe. Se mutabile, potrebbe causare il caricamento di una classe errata (perché gli oggetti mutabili cambiano il loro stato);

## 8 Ereditarietà e composizione

Generalmentesi tende a favorire la composizione in favore dell'eredirarietà.L'ereditarierà e l'override di metodi può essere pericoloso e più difficile di quanto ci si aspetti, proprio come accade per il metodo equals. La cosiddetta tecnica di composizione ci può salvare da questi pericoli:

```
/*ereditarietÃ */
public class Test extends Another{
   //codice...
}

/*composizione*/
public class Test {
   private Another a = ... ;
   //codice...
}
```

In pratica al posto di ereditare e fare override di metodi si usa un campo privato interno alla classe e si usa quell'oggetto all'interno di alcuni metodi che andranno definiti nella classe contenitore, in questo caso la Test.Vediamo un esempiodove si crea una classe che ne estende un'altra e ha lo scopo di contare quanti elementi sono stati inseriti.

```
// This is very common, but broken
public class InstrumentedHashSet<E> extends HashSet<E>
  private int addCount = 0; // add() calls
  public InstrumentedHashSet() {}
  public InstrumentedHashSet(Collection<? extends E> c)
     { super(c); }
  public boolean add(E o) {
     addCount++; return super.add(o);
  }
  public boolean addAll(Collection<? extends E> c) {
     addCount += c.size(); return super.addAll(c);
  }
  // accessor method to get the count
  public int addCount() { return addCount; }
}
```

Figura 6: Ereditarietà esempio errato

Internamente la classe HashSet implementa addAll()basandosi su add(); in altre parole, la classe HashSet dentro al metodo addAll()chiama add(). Facendo così il metodo addCount() ritorna un numero sbagliato.

- La chiamata addAll() incrementa correttamente la variabile addCount ma poi fa una chiamata a super.addAll(c);questo chiama il metodo add()che oltre a chiamare (giustamente) super.add(c);chiama anche addCount++! Quindi il conteggio totale viene sballato perché incremento addCountdi c.size()e poi anche di 1 col ++ ma non dovrei.

Vediamo dunque che in questo caso possiamo “favor compisition over inheritance” per risolvere il tutto:

```
// Note that an InstrumentedSet IS-A Set
public class InstrumentedSet<E> implements Set<E> {
   private final Set<E> s;
   private int addCount = 0;

   public InstrumentedSet (Set<E> s) { this.s = s}
   public boolean add(E o) {
      addCount++; return s.add(o); }
   public boolean addAll (Collection<? extends E> c) {
      addCount += c.size();
      return s.addAll(c);
   }
   // forwarded methods from Set interface
}
```

Figura 7: Composizione esempio giusto

Adesso il problema non c'è più perché, avendo un oggetto interno, posso usare questo e non ho bisogno di fare override o fare chiamate a super. Implemento solamente l'interfaccia e faccio override dei contratti dei metodi specificati da quell'interfaccia (Set<E>).

# 9 La keyword abstract

## 9.1 Metodi astratti

È possibile definire dei metodi astratti all'interno delle classi astratte. Tali metodi, dovranno essere necessariamente ridefiniti (tramite override) nelle classi figlie.

Un metodo astratto non può essere invocato in quanto non è definito, ma potrà essere soggetto a riscrittura( override ) in una sottoclasse.

```
public abstract void Metodo();
```

## 9.2 Le classi astratte

In java una classe può essere dichiarata astratta usando il modificatore *abstract*.

Una classe astratta è una classe *incompleta*, perchè contiene dei metodi astratti, cioè metodi d'istanza senza una corrispondente implementazione.

Data una astratta *AbstractC* non posso creare istanze di quella classe tramite *new AbstractC()*, perchè il loro comportamento non sarebbe determinato nel caso si invocasse su di esse un metodo astratto.

```
public abstract class A {
double campoDati1;
/*una classe che contiene almeno un metodo astratto
dev'essere marcata abstract*/
   public abstract void metodo();
}
```

Una classe astratta può dichiarare dei campi (che ne descrivono lo stato) e dei metodi (non astratti) che ne specificano il funzionamento, oppure delle interfacce.

**Lo scopo** e l'utilità delle classi astratte è di gestire il comportamento di base delle classi che la derivano e che può essere ampliato e specializzato da queste ultime. E' possibile avere dei costruttori astratti soprattutto quando ci sono dei campi dati nella classe,

Una *classe concreta* che estende una classe astratta deve necessariamente implementare tutti i metodi astratti della superclasse, altrimenti il compilatore segnalerà un errore.

Se una classe concreta estende una classe abstract e in più è marcata abstract, allora tale sottoclasse è considerata una classe astratta. Esempio:

```
public abstract class A {
   private int a;
   public A(int x) {a=x;}
   public abstract int m();
   }

public class C extends A { // C e' una concretizzazione di A
```

```
  private int b;
  public int m() {return 4;} // implementazione di m()
  public C() {super(8); b=0;} // OK//
  public C(int x){}// ILLEGALE: No constructor matching A()
  //found in class A
  public static void main(String args[]) {
   // A r = new A(9); //ILLEGALE: Abstract class A can't be
   //instantiated
   A s = new C(); // OK
   }
  }
```

## 9.3 Cose permesse vs cose non permesse

**Non permesso**:

- Non è possibile avere metodi statici astratti, altrimenti potrebbero essere invocati tramite il nome della classe senza istanziare un oggetto di quella classe astratta;
- Non è possibile avere un metodo finalastratto: sarebbe un *no sense*;
- Non si possono creare oggetti di classi astratte;

**Permesso**:

- E' possibile dichiarare un reference il cui tipo (statico) è una classe astratta A: il tipo dinamico di un tale reference dovrà quindi essere una sottoclasse concreta di A. Quindi una classe astratta può essere solamente un tipo statico, mai un tipo dinamico;
- E' possibile definire una classe astratta ma senza alcun metodo astratto. E' anche possibile marcare astratta una classe che non definisce dei metodi (solo stato);
- In una classe astratta ci possono essere dei metodi implementati che invocano metodi astratti: a run-time verrà sempre invocato virtualmente l'opportuno metodo concreto