

Appunti di Programmazione Concorrente e Distribuita

(A.A 2018/2019)

Grigoras Valentin

Matricola: 1099561

Contenuti

1	Metodo <code>Equals()</code>	5
1.1	Proprietà del metodo <code>Equals()</code>	5
1.2	Corretta implementazione di <code>Equals()</code>	5
1.3	Binding dinamico di <code>Equals()</code>	6
1.4	Metodo <code>Equals()</code> ed eredità	7
1.5	Tipi primitivi vs oggetti	7
1.6	<code>Equals()</code> Vs <code>==</code>	9
2	Operatore <code>instanceof</code>	10
3	Il metodo <code>String toString()</code>	10
3.1	Costruzione stringhe	11
3.2	Concatenazione stringhe	11
4	Le classi wrapper	11
4.1	Autoboxing - Autounboxing	12
4.2	Cose da evitare	12
5	Il metodo <code>Clone()</code>	12
5.1	<code>Clone()</code> con copia profonda	13
6	La keyword <code>final</code>	14
6.1	Campi dati <code>final</code>	14
6.2	Metodi <code>final</code>	15
6.3	Classi <code>final</code>	16
7	Classe immutabile	16
7.1	Regole per creare una classe immutabile	16
7.2	Esempio pratico	17
7.3	Vantaggi	19
7.4	Svantaggi	20
7.5	Differenza tra classe immutabile e oggetto immutabile	20
7.6	Regole per creare oggetti immutabili	20
7.7	Perchè la classe <code>String</code> è immutabile	21
8	Ereditarietà e composizione	21
9	La keyword <code>abstract</code>	23
9.1	Metodi astratti	23
9.2	Le classi astratte	23
9.3	Cose permesse vs cose non permesse	24
10	Interfacce	24
10.1	Pre Java 8	24
10.1.1	La dichiarazione di interfacce	25
10.2	Post Java 8	25
10.2.1	Metodi statici	25
10.2.2	Metodi di default	26
10.2.2.1	Regole per l'utilizzo dei metodi default	27

11	Classi interne	28
11.1	Classi ed interfacce statiche innestate	28
11.2	Classi innestate non statiche	29
11.2.1	Proprietà	29
11.2.2	Quando usare le classi innestate	30
11.3	Classi anonime	31
12	Dipendenze	33
12.1	Dependency Injection	34
12.1.1	Constructor Injection	34
12.1.2	Setter Injection	34
12.1.3	Method Injection	35
12.2	Inversion of control	35
13	Generics e Collection	36
13.1	Classi generiche	37
13.2	Metodi generici	38
13.2.1	Prima delle Generics	39

Elenco delle figure

1	Implementazione corretta di equals	6
2	tipi primitivi e oggetti: rappresentazione in memoria	8
3	tipi primitivi e oggetti: rappresentazione in memoria dopo assegnazione	8
4	Shallow copy vs Deep copy	13
5	Deep copy Clone()	13
6	Ereditarietà esempio errato	22
7	Composizione esempio giusto	22
8	Classe anonima	32
9	Traditional Program Execution	35
10	Inversion of Control Execution	36
11	Generics - ArrayList	37
12	Prima dei Generics - Vector	38
13	Generics - classe generica	38

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 restituiamo `false`;

```
if (!(obj instanceof 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) obj;
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:

```

Employee e1 = new Employee (...);
Employee 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;
 - Sì, 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;
 - Sì, 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");
```

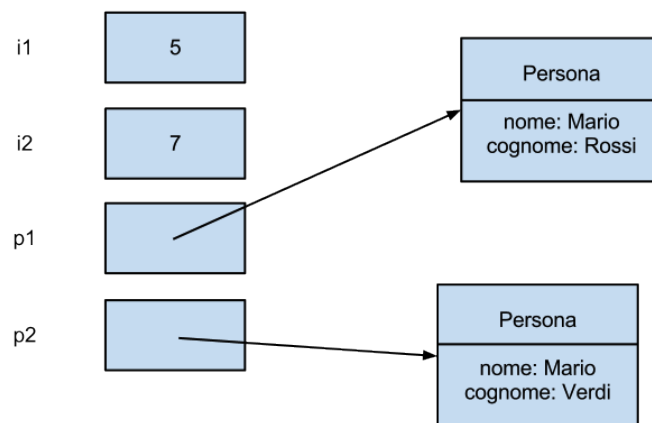


Figura 2: tipi primitivi e oggetti: rappresentazione in memoria

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:

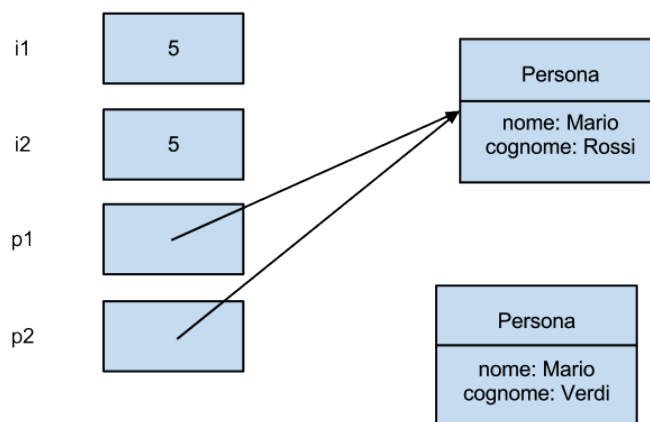


Figura 3: tipi primitivi e oggetti: rappresentazione in memoria dopo assegnazione

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 instanceof

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:

```
<referimento> instanceof <tipo_referimento>
```

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

è importante ricordare che l'operazione più efficiente a lungo andare è la seconda. Nel primo caso ho un reference perché alloco nello heap un oggetto, nel secondo caso NON ho un reference e costruisco la stringa che è 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 StringBuilder invece 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, astruendo proprio il concetto di tipo primitivo. Le classi wrapper sono le seguenti: *Byte*, *Short*, *Integer*, *Long*, *Float*, *Double*, *Boolean*, *Character*, ognuna delle quali può contenere il relativo tipo primitivo. Tutte le class wrapper appartengono alla *Boolean* sono sottoclassi 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 perché non esistono metodi setter. Quindi un oggetto istanziato così:

```
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, ovvero 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 shallow 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 così una condivisione di memoria.

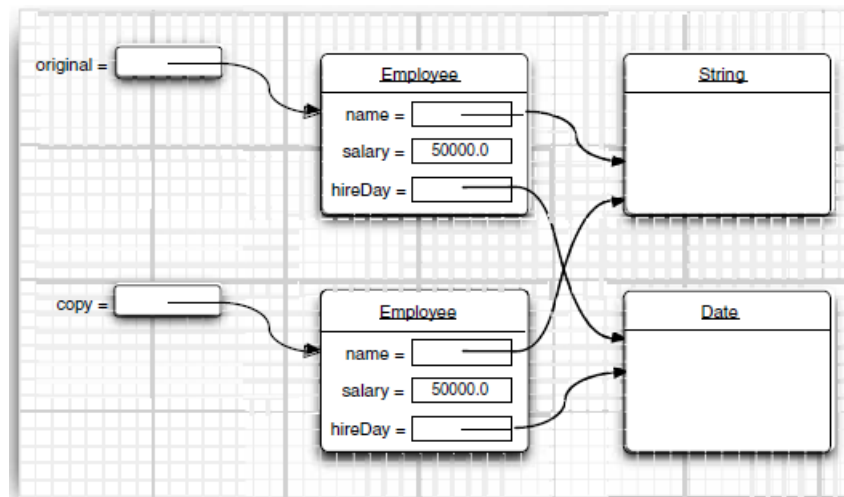


Figura 4: Shallow copy vs Deep copy

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 result;
    }
}

```

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 campi che 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 `Object` ma 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 passo in 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 riferisce 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.length; 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 eventuali 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 memorizzata 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 += 6.0;
                break;
        }
        return prezzo;
    }
}
```

```
        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 TimeZone 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 istanziato 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 costruttore 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 Perché 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* immutabile 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 a e b, 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'ereditarietà. L'ereditarietà 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:

```
/*ereditarietà*/
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 esempio dove 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 `addCount` di `c.size()` e poi anche di 1 con `++` ma non dovrei.

Vediamo dunque che in questo caso possiamo “favor composition 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

10 Interfacce

Le interfacce possono essere intese come un'evoluzione delle classi astratte e permettono, di fatto, di simulare l'ereditarietà multipla.

10.1 Pre Java 8

Sino a Java 7 un'interfaccia per definizione poteva possedere solo metodi dichiarati *public* e *abstract* e variabili dichiarate *public*, *static*, *final*. Esempio di interfaccia:

```
public interface Saluto{
    public static final String CIAO = "Ciao";
    public static final String BUONGIORNO = "Buongiorno";

    public abstract void saluta ();
}
```


Ommettendo i modificatori, il programma funziona comunque in quanto vengono aggiunti automaticamente da Java. Viene aggiunto *public abstract* per i metodi e *public static final* per le variabili.

Non posso dichiarare i modificatori *private*, *protected*.

10.1.1 La dichiarazione di interfacce

Una dichiarazione di interfaccia crea un nuovo tipo riferimento che avrà come istanze tutte le istanze delle classi che implementano l'interfaccia.

La definizione di un'interfaccia avviene tramite la chiave **interface**. Un'interfaccia può contenere solo dichiarazioni di classi, interfacce, costanti e metodi statici. Si possono usare i seguenti modificatori:

- **Abstract**: ogni interfaccia è implicitamente astratta;
- **Public**: tutti i membri di un'interfaccia sono dichiarati implicitamente come *public*;
- **Static**: le classi membro, le interfacce membro e i campi sono dichiarati implicitamente *static*;
- **Protected e Private**: si applica solo a interfacce membro a una dichiarazione di classe;
- **Final**: i campi sono implicitamente *final*, cioè definiscono delle costanti.

Un'interfaccia non può estendere una classe ma può estendere una o più interfacce dette *superinterfacce*, ereditando i suoi metodi.

La differenza tra implementare un'interfaccia ed estendere una classe, consiste nel fatto che possiamo estendere una sola classe alla volta. Possiamo invece implementare un numero indefinito di interfacce simulando l'ereditarietà multipla.

Riassumendo quindi una classe può implementare un'interfaccia e un'interfaccia può estendere un'altra interfaccia.

10.2 Post Java 8

10.2.1 Metodi statici

Dalla versione 8 di Java in poi è possibile definire all'interno delle interfacce anche **metodi statici**.

```
public interface MyInterface{
    static void metodoStatico(){
        System.out.println("Metodo Statico Chiamato!");
    }
}
```

È possibile chiamare i metodi statici direttamente dalle interfacce:

```
public interface TestMyInterface{
    public static void main (String args[]){
        MyInterface.metodoStatico();
    }
}
```

Un'interfaccia contenente metodi statici viola la natura contrattuale delle interfacce. Un'interfaccia simile non ha bisogno di essere implementata perchè i metodi statici di un'interfaccia non vengono ereditati. Interfacce di questo tipo serve solamente per definire funzioni.

```
public class MyClass implements StaticMethodInterface {}

public class TestMyInterface {
    MyClass.metodoStatico(); /*errore: cannot find symbol
                             perché il metodo statico non viene ereditato.*/

    StaticMethodInterface.metodoStatico(); /*ok, metodoStatico
                             presente in StaticMethodInterface*/
}
```

Si possono creare anche dei metodi statici privati, chiamati solamente dalle interfacce che gli contengono. Agiscono da metodi ausiliari per i metodi statici.

10.2.2 Metodi di default

Un'altra novità di Java 8 è la possibilità di dichiarare metodi concreti all'interno delle interfacce grazie alla keyword *default*.

```
interface TestInterface {
    //square() é implicitamente public ed abstract
    void square(int a);
    default public void show(){
        System.out.print("i-test");
    }
}

class Test implements TestInterface {
    public void square(int a)
    { ... }
    public static void main(...) {
        Test d = new Test();
        d.square(5); //ok, esegue square
        d.show();    //stampa: i-test
    }
}
```

In pratica è possibile aggiungere la segnatura *default* davanti alla dichiarazione del metodo e dargli un'implementazione; così facendo tutte le classi che implementano quell'interfaccia avranno a disposizione quel metodo senza doverlo definire per forza.

Notare però che *square(int a)* non è default e quindi, solito, serve definire il contratto nella classe che implementa l'interfaccia. Tuttavia il metodo di default "scompare" quando viene ridefinito all'interno della classe che lo implementa:

```
interface TestInterface {
    //square() é implicitamente public ed abstract
    public void square(int a);
    default public void show(){
        System.out.print("i-test");
    }
}
```

```

class Test implements TestInterface {
    public void square(int a)
    { ... }

    public void show(){
        System.out.print("i-test-over");
    }

    public static void main(...) {
        Test d = new Test();

        d.show();    //stampa: i-test-over
    }
}

```

Qui vediamo che è stato fatto l'override di show() (che è comunque un metodo virtuale!) e quindi la classe mostra i-test-over. in output. La regola è che la classe "vince sempre" perché se non appare nulla si usa il metodo default, altrimenti vale l'override.

Può capitare questo:

```

interface TestInterface1 {
    default public void show() {
        System.out.print("test1");
    }
}

interface TestInterface2 {
    default public void show() {
        System.out.print("test2");
    }
}

class Test implements TestInterface1, TestInterface2 {
    public void show() {
        TestInterface1.super.show();
        TestInterface2.super.show();
    }
}

public static void main(...) {
    ...
    d.show(); // stampa: i-test-over
}

```

In questo caso c'è il problema che due interfacce diverse danno la stessa definizione del metodo show; il problema nasce perché Test implementa da entrambe le interfacce. Si crea un conflitto dato che Java non sa fra quale dei due scegliere. Per risolvere è necessario fare l'override e specificare cosa fare; in particolare la sintassi **NomeInterfaccia.super.metodo()** richiama il relativo metodo specificato di default.

La keyword default è stata introdotta per estendere le Collection API, sottoposta a vari cambiamenti, per aggiungere nuove funzionalità fornite da Java 8 quali lambda e Stream API, preservando la compatibilità con le librerie già esistenti.

10.2.2.1 Regole per l'utilizzo dei metodi default

1. Le classi vincono sulle interfacce. Quindi, se vi è una dichiarazione di un metodo concreto o astratto nelle catene dei sottotipi con la stessa segnatura di un metodo di default, possiamo ignorare le interfacce completamente. Valgono le stesse regole di invocazione dinamica già viste;
2. Il figlio (sottotipo) vince sul padre (sovratipo). Se si presenta una situazione nella quale due interfacce forniscono un metodo di default con stessa segnatura e un'interfaccia estende l'altra, allora il sottotipo vince.
3. No rule3. Se applicate le prime due regole i conflitti non sono stati risolti, la sottoclasse dovrebbe (1) implementare il metodo oppure (2) dichiararlo astratto.

11 Classi interne

11.1 Classi ed interfacce statiche innestate

Le classi e le interfacce statiche innestate sono definite all'interno del corpo di una classe o di un'interfaccia e precedute da uno specificatore d'accesso (opzionale) e dalla parola chiave `static`.

```
public class Persona {
    public static class NomeAnagrafico {
        private String nome;
        private String cognome;

        public NomeAnagrafico(String n, String c) {
            nome = n;
            cognome = c;
        }

        public String getNome() {
            return nome;
        }

        public String getCognome() {
            return cognome;
        }
    }

    private NomeAnagrafico nomeAnagrafico;
    public Persona(String n, String c) {
        nomeAnagrafico = new NomeAnagrafico(n, c);
    }

    public NomeAnagrafico getNomeAnagrafico() {
        return nomeAnagrafico;
    }
}
```

La classe `NomeAnagrafico` è interna, definita nel corpo della classe di primo livello `Persona`. La compilazione del sorgente `Persona.java` porta alla nascita di due distinti bytecode: `Persona.class` e `Persona$NomeAnagrafico.java`. La norma che fa corrispondere un bytecode ad ogni classe o interfaccia non è infranta.

Le classi statiche innestate fanno uso di quattro specificatori d'accesso:

- **public**: visibile ovunque mediante la notazione *ClasseContenitrice.ClasseInterna*;

- **private**: accessibile solo dal codice della classe che la contiene;
- **protected**: accessibile dal codice della classe che la contiene e dalle sue eventuali classi figlie, oltre che dal codice delle classi presenti nello stesso package;
- **Priva di specificatore**: disponibile solo all'interno del package che ospita la classe che la contiene.

11.2 Classi innestate non statiche

Una **classe innestata** (o **classe interna** nel caso non sia dichiarata statica, o classe membro) non è altro che una classe definita all'interno di un'altra classe. La limitazione delle classi interne statiche o delle interfacce statiche è che non possono invocare membri non statici della classe che li contiene. Esempio:

```
public class Outer{
    public String messaggioOuter = "ciao";

    private void StampaMessaggio(){
        System.out.println(messaggioOuter);
    }

    public class Inner{
        public void Stampa(){
            StampaMessaggio();
        }
    }
}
```

La classe interna (Inner) ha accesso ai membri della classe esterna anche se dichiarati *private*. Nell'esempio precedente la classe inner richiama il metodo StampaMessaggio() presente nella classe Outer.

11.2.1 Proprietà

- Deve avere un identificatore differente dalla classe che la contiene;
- Per istanziare una classe interna al di fuori della classe in cui è definita bisogna seguire alcuni passi:
 - istanziare la classe esterna in cui è dichiarata la classe interna : `Outer outer = new Outer();`
 - dichiarare l'oggetto che si vuole istanziare dalla classe interna tramite la classe interna: `Outer.Inner oggettoInner;`
 - istanziare l'oggetto della classe interna tramite l'oggetto istanziato dalla classe esterna: `oggettoInner = out.new Inner();`
- ha accesso sia alle variabili d'istanza sia a quelle statiche della classe in cui è contenuta;
- se viene dichiarata statica diventa automaticamente **top-level class**, quindi non potrà accedere alle variabili d'istanza nella classe che la contiene;
- solo se dichiarata statica può dichiarare membri statici;

- può essere dichiarata astratta;
- può utilizzare il reference `this` per accedere ai suoi membri interni (non a quelli della classe contenitrice);
- per accedere ai membri esterni a partire da una classe interna `:NomeClasseEsterna.this.NomeMembroDellaClasseEsterna`;
- se 2 classi hanno lo stesso membro verrà preso quello della classe interna;

11.2.2 Quando usare le classi innestate

Potrebbe essere desiderabile creare una classe innestata solo in casi in cui ci sia una forte relazione tra la classe Inner e Outer. Per esempio una classe `Auto` potrebbe avere bisogno di una classe `Meccanico` che effettui delle riparazioni. La soluzione senza classi innestate potrebbe essere questa:

```
public class AutoNoInner{
    private String statoMotore;
    private MeccanicoNoInner meccanico;

    public AutoNoInner(){
        meccanico = new MeccanicoNoInner(this);
    }

    public void setStatoMotore(String statoMotore){
        this.statoMotore = statoMotore;
    }
    public String getStatoMotore(){
        return statoMotore;
    }
}

public class MeccanicoNoInner{
    private AutoNoInner auto;

    public MeccanicoNoInner(AutoNoInner auto){
        this.auto = auto;
    }

    public void aggiustaMotore(){
        auto.setStatoMotore("buono");
    }
}
```

Invece la versione con le classi interne è la seguente:

```
public class Auto{
    private String statoMotore;

    public class Meccanico{
        public void aggiustaMotore(){
            statoMotore = "buono";
        }
    }
}
```

Il lato negativo di utilizzare la classe interna è che se si vuole utilizzare la classe meccanico al di fuori della classe Auto, bisognerà sempre istanziare un'auto per istanziare un meccanico.

Altro esempio:

```
class Outer {
    private int a;
    public Outer(int value) {
        a = value;
    }

    public class Inner {
        private int b;
        public Inner(int value) {
            b = a + value;
        }

        public void showValue() {
            System.out.println(b);
        }
    }

    public void printValue() {
        System.out.println(a);
    }
}

public class Main {
    public static void main(...) {
        Outer o = new Outer(10);
        o.printValue(); // stampa 10
        Outer.Inner i = o.new Inner(20);
        i.showValue(); // stampa 30
    }
}
```

Notare che *Inner i = new Inner(20)* non va bene perché le classi interne possono essere create solo se legate ad istanze di oggetti. Occorre quindi creare prima un oggetto della classe outer (fatto prima con *new Outer(10)*) e poi usare la sintassi *Outer.Inner i = o.new Inner(20)*.

11.3 Classi anonime

Una classe anonima è una classe **locale** senza un nome assegnato, si tratta di una classe definita e istanziata un'unica volta attraverso una singola espressione caratterizzata da una versione estesa della sintassi dell'operatore new.

```

public class E {
    public I i() {
        return new I() { // anonymous inner class
            private int i = 5;
            public int value() { return i; }
        }; // Note the ";"
    }
    public static void main(String[] args) {
        E e = new E();
        I ref = e.i();
        ref.value();
    }
}

```

```

interface I {
    int value();
}

```

Figura 8: Classe anonima

La definizione della classe è integrata nell'espressione di return o di dichiarazione di variabile, basta che ci sia l'operatore new. Una classe anonima è una notazione abbreviata per creare un semplice oggetto piccolo "inline" che racchiude del codice da eseguire. Di solito il tipo è una interfaccia. Nell'esempio sopra la classe è castata ad una interfaccia; sarebbe equivalente a:

```

class Anonima implements I { ... };

return new Anonima();

```

A differenza delle classi interne, una classe anonima richieda che:

- al momento della dichiarazione della classe venga istanziato un suo oggetto;
- ci sia l'esistenza di una classe o di una superinterfaccia di cui sfrutterà il costruttore (solo virtualmente nel caso di un'interfaccia). Se una classe non ha nome non può avere un costruttore.

Se compiliamo una classe che contiene delle classi anonime verranno creati tanti file quante sono le classi anonime.

```

public class Outer{
    private String messaggio = "Nella classe";

    /*Definizione della classe anonima e sua istanza*/
    ClasseAnonima ca = new ClasseAnonima(){
        @Override
        public void metodo(){
            System.out.println(messaggio + "anonima");
        }
    };

    /*superclasse della classe anonima*/
    public class ClasseAnonima{
        public void metodo(){
            System.out.println("Nella classe anonima");
        }
    }
}

```



```
}
}
```

La classe anonima dell'esempio usa il reference `ca` di tipo `ClasseAnonima` per referenziare un oggetto della sua sottoclasse anonima, e implicitamente come tutte le sottoclassi chiama sempre il costruttore della superclasse.

Una classe anonima viene dichiarata con lo scopo di fare override di uno o più metodi della classe che la estende. Se infatti nell'esempio di prima avessimo definito un nuovo metodo (non ereditato) nella classe anonima, non saremmo stati in grado di invocarlo senza un reference del tipo della classe anonima (che però non può esistere).

```
/*Definizione della classe anonima e sua istanza*/
ClasseAnonima ca = new ClasseAnonima() {
    @Override
    public void metodo() {
        System.out.println(messaggio + "anonima");
    }
};

public void metodoNonInvocabile() {
    System.out.println("Per una classe anonima non esistono
reference. Impossibile chiamare questo metodo!");
}
}
```

12 Dipendenze

Un buon sistema OO dovrebbe basarsi sull'interazione fra gli oggetti e non fra la dipendenza/parentela fra di essi; si vuole avere un'alta coesione ed un basso accoppiamento. Conviene quindi evitare l'uso dell'ereditarietà, che è il tipo più forte di dipendenza, a discapito delle interfacce. Vogliamo che le classi siano enti più "isolati" possibili e che non dipendano da altri pezzi.

Favor composition over inheritance

L'ereditarietà comporta il fatto che, in caso di cambiamenti radicali ad una classe che sta in alto nella gerarchia, si può rompere tutta la gerarchia perché le figlie devono adattarsi al cambiamento della madre. Se ci sono molte sottoclassi, potrebbe rompersi tutto. Usare le interfacce è buona cosa perché dicono cosa fare ma lasciano libertà di implementazione. Vediamo un esempio:

```
/*classe che elenca una serie di film*/
class ListaFile {
    /*classe con metodi di ricerca in una lista*/
    private TrovaFilm f;

    public ListaFilm() {
        f = new CSVMovieFinder("movies.csv");
    }
}
```

In questo codice molto semplice sto facendo new CSVMovieFinder, sottoclasse di TrovaFilm, e ho vari problemi. Intanto, se dovesse cambiare l'implementazione del costruttore dovrei aggiornare **tutti** i costruttori di quella classe, ovunque la abbia dichiarata. Poi, se dovessi pescare i dati no da un CSV ma da un json/SQL dovrei cambiare il tipo di sottoclasse perché la new crea il tipo sbagliato. Se cambiassi TrovaFilm? Insomma, la classe ListaFilm **dipende** da TrovaFilm tantissimo; un cambiamento a TrovaFilm potrebbe rompere tutto.

Risolvero il problema usando la Dependency Injection.

12.1 Dependency Injection

L'idea alla base della Dependency Injection è quella di avere un componente esterno (assembler) che si occupi della creazione degli oggetti e delle loro relative dipendenze e di assemblarle mediante l'utilizzo dell'injection. Questa tecnica mira a minimizzare le dipendenze fra classi. Questa, può essere realizzata principalmente con tre tecniche che vanno usate in base al contesto:

12.1.1 Constructor Injection

La dipendenza viene iniettata tramite l'argomento del costruttore

```
class ListaFilm {  
    private TrovaFilm f;  
  
    public ListaFilm (TrovaFilm film) {  
        f = film; // Constructor injection  
    }  
  
    public void qualcosa () { f. stampaListaAvideo (); }  
}
```

Per risolvere il problema della dipendenza e "staccare" le due classi, posso "iniettare" la dipendenza via costruttore. In altre parole, faccio la new FUORI E NON DENTRO alla classe che necessita di usare l'altra classe. Passo via costruttore una reference ad un oggetto che vive fuori. In questo modo ListaFilm non crea più nulla e non ha più l'ownership dell'oggetto; la classe ListaFilm ha solo una reference che viene gestita altrove; non interessa altro. Notare quindi che in generale, quando è necessario (come in questo caso) che ci sia dipendenza fra classi (ListaFilm deve usare TrovaFilm) è meglio usare i principi di dependency injection per ridurre l'accoppiamento tra classi.

12.1.2 Setter Injection

La dipendenza viene iniettata attraverso un metodo "set".

```
class ListaFilm {  
    private TrovaFilm f;  
  
    public void setMovie (TrovaFilm film) {  
        f = film;  
    }  
  
    public void qualcosa () { f. stampaListaAvideo (); }  
}
```

Stessi discorsi di prima solo che qui anziché passare al costruttore la dipendenza, la passo via metodo. Nel costruttore creo l'oggetto ma non ho la necessità di usare TrovaFilm, quindi non la passo; piuttosto creo un metodo che mi permette di settare tale variabile. Diciamo quindi che la Constructor Injection mi indica che ho una dipendenza obbligatoria che va usata per forza, la Setter Injection invece mi dice che sì ho una dipendenza ma non è così "forte", posso anche farne a meno.

12.1.3 Method Injection

```
class ListaFilm {  
    private TrovaFilm f;  
  
    public void qualcosa(TrovaFilm film) {  
        return film.printAll();  
    }  
}
```

Si tratta del tipo più "soft" di injection perché utilizza temporaneamente un'altra dipendenza (classe o interfaccia); la classe non tiene alcuna reference al suo interno di tale dipendenza. Quindi si fa il minimo indispensabile proprio, toccata e fuga via metodo e basta. Il più basso livello.

12.2 Inversion of control

Il principio di IoC (detto anche DIP – Dependency Inversion Principle) dice che ogni modulo ad alto o basso livello non deve dipendere da altri moduli ma piuttosto deve dipendere da astrazioni. Si fa inversione di controllo perché si inverte il ruolo di chi gestisce l'applicazione; invece di essere l'app che controlla il flusso di esecuzione, è il framework che controlla il flusso di esecuzione.

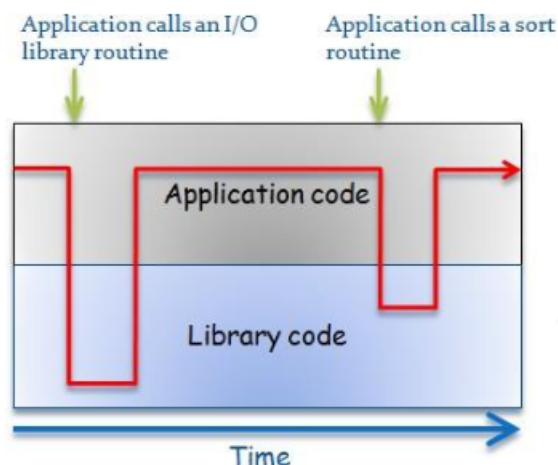


Figura 9: Traditional Program Execution

L'applicazione ha la maggior parte del controllo del flusso di esecuzione (in rosso); questo è un esempio di ciò che normalmente accade. Un'applicazione usa una libreria qualsiasi (libreria numerica, di database...) solo quando ne ha di bisogno ma poi è

l'applicazione (scritta dal programmatore) che gestisce. Quindi è il programmatore che tramite il codice dell'applicazione controlla il flusso.

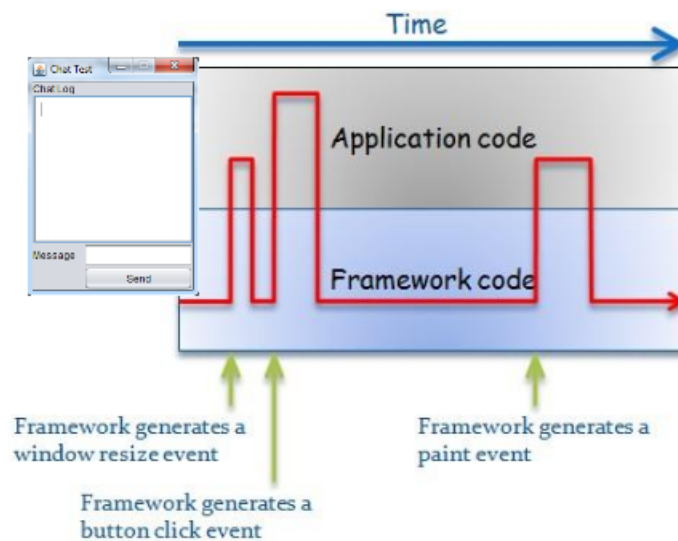


Figura 10: Inversion of Control Execution

In caso di inversione di controllo, c'è un framework che controlla l'intero flusso di una applicazione. Infatti si vede che il flusso appartiene molto alla libreria e poco all'app. Questo accade con le app android ad esempio. Il framework chiama, quando ne ha di bisogno il codice scritto dal programmatore; prima invece era il programmatore che tramite il programma chiamava le funzioni quando voleva lui!

Meglio fare un esempio. Una applicazione Android funziona nell'ambiente Android; quando premo sull'icona di una app c'è il framework (sistema operativo Android) che fa una serie di chiamate a metodi, fa delle new e altre cose per far partire l'app. Il ruolo del programmatore è quello di scrivere una Activity android, cioè il punto di ingresso dell'app (Activity android = il main del C++ per capirci).

Quindi quando l'app parte, il controllo del flusso lo ha in mano il framework (SO Android) perché è lui che chiama i metodi di inizializzazione, gestisce la memoria e gestisce la vita dell'app. Non c'è l'utente che manualmente clicca sull'app, la fa partire invocando i metodi e scegliendo cosa fare. Se ne occupa tutto il sistema operativo.

La dependency injection è una delle tecniche con le quali si può fare inversione di controllo; essa prende il controllo di tutte le azioni base (creazione di oggetti e dipendenze). Famosa è la libreria Spring di java che usa molto questo fatto dell'inizializzazione automatica e cerca di passare il flusso di controllo alla libreria il più possibile.

13 Generics e Collection

I generics sono stati introdotti in Java 5 e sono simili ai template di classe e di funzione del C++. In Java si usa la terminologia **classe generica** e **metodo generico**. I generics consentono di scrivere codice meno verboso evitando pericolosi cast di tipi, e permettendo di creare algoritmi generici (ovvero basati sui generics).

La definizione utilizza un tipo parametrico T che deve essere specificato quando si vuole usare un'istanza di quel template di classe/ interfaccia, mentre per i metodi viene

dedotto dal compilatore. Il tipo T non può essere un primitivo; vengono accettati solo tipi classe e interfaccia. Per definire un generic si usano le parentesi angolari:

```
public class Classe <T,Z> { }  
  
public interface Interfaccia <T> { }  
  
public <T> void f(T[] array ) { }
```

Un metodo statico di una classe generica che utilizza un tipo parametrico deve essere un metodo generico o si ha un errore di compilazione.

```
public class Classe <T >{  
    public static <T> void m(T[] arg ) { }  
}
```

Se un generic viene esteso o implementato, deve essere indicato il tipo esplicitamente nella segnatura.

13.1 Classi generiche

La classe `ArrayList` della libreria, ad esempio, è generica, cioè è stata dichiarata usando un parametro di tipo *T*.

```
public class ArrayList<T>  
    extends AbstractList<T>  
    implements List<T>, RandomAccess, Cloneable, Serializable {  
    public boolean add(E obj) {...}  
    public T elementAt(int index) {...}  
    ...  
}  
  
ArrayList<String> v = new ArrayList<>();  
ArrayList<Dipendente> w = new ArrayList<Dipendente>(5);  
v.add(new String("pippo"));  
String s = v.elementAt(0);  
// w.add(new String("pippo")); // NON COMPILA
```

Figura 11: Generics - ArrayList

Qui vediamo che la classe `ArrayList` è generica perché accetta un parametro chiamato *T* e questo verrà usato al suo interno per definire metodi e variabili. Come si può vedere, se *T* fosse stata di tipo `String`, allora ci sarebbe un *tipaggio forte* e nessun altro tipo sarebbe stato accettato. In pratica, una volta definito il tipo del parametro, bisognerà sempre tenere presente che verrà usato quel tipo. Se la classe sopra, cioè `Dipendente`, fosse stata una superclasse, non sarebbe stato comunque accettabile fornire una sua sottoclasse al metodo `add()`. Non centra l'ereditarietà, bisogna che il tipo sia uguale.

I parametri di tipo di un generics possono essere istanziati a classi ed interfacce ma non a tipi primitivi. Prima dei generics, la classe *Vector* (thread-safe *ArrayList*) sfruttava l'Object:

```
Vector v = new Vector();  
v.add(new String("pippo")); v.add(new Integer(1));  
String s = (String)v.elementAt(0);  
Integer i = (Integer)v.elementAt(1);
```

Figura 12: Prima dei Generics - Vector

Posso inserire qualsiasi tipo di oggetto ma devo sapere cosa si va a pescare (cast specifico a run-time) per poter sfruttare l'oggetto. **Potenziale disastro a run-time!**

Esempio di una semplice classe generica *Coppia<T,S>*:

```
public class Coppia<T, S> {  
    private T first;  
    private S second;  
    public Coppia(T x, S y) {first = x; second = y;}  
    public T getFirst() {return first;}  
    public S getSecond() {return second;}  
}
```

Figura 13: Generics - classe generica

Ecco un esempio di classe con due parametri generici. Quando si istanzia il tipo, basta fornire i tipi al posto dei parametri e il compilatore dedurrà automaticamente i tipi da sostituire alle variabili. Quindi:

```
var c = new Coppia<Integer, Double>(3, 7)();
```

Importante notare che i parametri generici possono essere istanziati a classi e interfacce ma **NON** a tipi primitivi. Per questo motivo è stata usata la classe wrapper Integer al posto di un int nel costruttore. Grazie al boxing del tipo, è stato possibile passare un int all'Integer. Notare che questo codice non compila:

13.2 Metodi generici

Un metodo generico è un metodo che ha una o più variabili di tipo. Se non vi fosse la possibilità di scrivere il metodo in forma generica, per adempiere allo stesso scopo si dovrebbe ricorrere al meccanismo dell'overloading dei metodi, che comporta la necessità di scrivere tanti metodi che eseguono lo stesso compito su tipi di dato differenti. Quando si invoca un metodo generico non si devono specificare i tipi effettivi da usare al posto dei parametri di tipo: si invoca semplicemente il metodo con i parametri

appropriati e il compilatore dedurrà corrispondentemente i tipi da usare per i parametri. Bisogna usare metodi generici ogniqualvolta si può. Se è possibile, conviene utilizzare un metodo generico al posto dell'intera classe. Se un metodo è statico non avrà alcun accesso ai parametri generici della classe. Se serve che esso sia generico, serve dichiararlo esplicitamente usando un altro tipo.

```
public class C<T>{
    public static void print(T[] a) {

    }
}
//Non compila: Cannot make a static reference to non-static type T
```

```
public class C<T>{
    public static <E>void print(E[] a) {
        ...
    }
} //OK!
```

13.2.1 Prima delle Generics

Vediamo alcuni esempi di come venivano scritti le classi senza l'utilizzo delle generics: Il primo esempio che vediamo è stato preso dal libro Java 8 Guida Completa di Pellegrino Principe Questa è la versione senza Generics:

```
package com.pellegrinoprincipe;
class PrintArray
{
    public PrintArray() {}
    public void printArray(Integer el[]) // stampa un array di interi
    {
        for (Integer i : el)
            System.out.print(i + " ");
    }
    public void printArray(Double el[]) // stampa un array di double
    {
        for (Double i : el)
            System.out.print(i + " ");
    }
    public void printArray(Character el[]) // stampa un array di caratteri
    {
        for (Character i : el)
            System.out.print(i + " ");
    }
}
public class PrintArrayClient
{
    public static void main(String[] args)
    {
        PrintArray pa = new PrintArray();
        Double d[] = { 11.1, 11.2 };
        Integer i[] = { 12, 13 };
        Character c[] = { 'a', 'b' };
        System.out.print("[ ");
    }
}
```

```
        pa.printArray(d);
        pa.printArray(i);
        pa.printArray(c);
        System.out.print("]");
    }
}

//output : [ 11.1 11.2 12 13 a b ]
```

Il codice sopra crea una classe denominata `PrintArray` con tre metodi in overloading che permettono di stampare tutti gli elementi di un array di tipo differente.

Nella classe `PrintArrayClient` si creano un oggetto di tipo `PrintArray` e tre riferimenti di oggetti array che contengono rispettivamente valori di tipo `double`, `int` e `char`. Nell'assegnamento dei valori agli array viene sfruttato *autoboxing* e *autounboxing* (vedi §4.1).

L'output sarà sempre diverso, perchè a seconda del tipo di valore passato come argomento, il compilatore invocherà il corrispettivo metodo, grazie all'overloading.

Vediamo adesso quanto semplice diventa il codice con l'utilizzo delle Generics:

```
package com.pellegrinoprincipe;
class PrintArrayGeneric
{
    public PrintArrayGeneric() {}
    public <E> void printArray(E el[])
    {
        for (E i : el) // stampa in modo generico gli elementi
                        //dell'array di differente tipo
            System.out.print(i + " ");
    }
}

public class PrintArrayGenericClient
{
    public static void main(String[] args)
    {
        PrintArrayGeneric pag = new PrintArrayGeneric();
        Double d[] = { 11.1, 11.2 };
        Integer i[] = { 12, 13 };
        Character c[] = { 'a', 'b' };
        String s[] = { "sono", "una", "stringa" };
        System.out.print("[ ");
        pag.printArray(d);
        pag.printArray(i);
        pag.printArray(c);
        pag.<String>printArray(s); // sintassi alternativa di invocazione
                                  //di un metodo generico
        System.out.print("]");
    }
}

//output: [ 11.1 11.2 12 13 a b sono una stringa ]
```

Come possiamo notare il codice si riduce drasticamente. Abbiamo infatti scritto un solo metodo, rispetto ai 3 del codice precedente.

In dettaglio: il metodo `printArray` ha, nella sezione di dichiarazione dei parametri di tipo, il tipo `E`, che è utilizzato poi come tipo del parametro formale nella lista dei parametri del metodo (`E el[]`) e poi come tipo della variabile locale nel ciclo `for` (`E i`). Notiamo, inoltre, che nella classe `PrintArrayGenericClient` abbiamo aggiunto un array di oggetti `String` e l'abbiamo fatto stampare sempre dal metodo `printArray`. Quest'aggiunta, tuttavia, non ha invalidato la corretta compilazione del programma. Infatti, poiché `printArray` è un metodo generico, non è stato necessario scrivere nella classe `PrintArrayGeneric` il corrispondente metodo in overloading che accettasse un argomento di tipo array di `String` (cosa che, invece, avremmo dovuto fare se il linguaggio non avesse supportato i generici).

Infine, è interessante evidenziare l'istruzione `pag.<String>printArray(s)`, che mostra un modo alternativo per invocare un metodo generico che si concretizza nel passare il tipo effettivo tra le parentesi angolari `< >` prima del nome del metodo stesso. Tuttavia tale sintassi non è necessaria; infatti si può tranquillamente omettere di scrivere il tipo tra le parentesi angolari, poiché il compilatore è in grado di "capirlo" autonomamente analizzando il tipo passato come argomento, come mostrato per le altre invocazioni del metodo `printArray` (type inference).

type inference : è la capacità del compilatore di determinare automaticamente il tipo di argomento di un metodo generico. Il compilatore è in grado di inferire (dedurre) in autonomia il tipo di argomento quando il risultato di un'invocazione di metodo è passato come argomento a un altro metodo (*inference in argument position* o *inference in method context*)

13.3 Vincoli sui tipi delle variabili e Wildcards

Spesso è necessario specificare quali tipi possano essere usati in una classe generica oppure in un metodo generico. Ad esempio, in un metodo generico `T min(T[] a)` che trova il valore minimo presente in un array:

```
public static <T> T min (T[] a) {
    T x = a[0];
    for(int i=1; i<a.length; ++i)
        if(a[i] < x)
            x=a[i];

    return x;
}
```

Il compilatore non può applicare l'operatore `<` perché non sa nulla su `T`, sa solamente che è un tipo che verrà istanziato. Non può sapere l'effetto dell'operatore sul tipo `T` perché non può sapere cosa diventerà `T` (`Integer`, `Double`, una classe, un'interfaccia...). Per risolvere il problema si può fare così:

```
public static <T extends Comparable<T>> T min (E[] a){
    E x = a[0];
    for(int i = 1; i<a.length; ++i)
        if(a[i].compareTo(x) < 0)
            x=a[i];
    return x;
}
```