

# **Programmazione Orientata agli Oggetti**

---

Generics: Concetti Base

# Obiettivi della Lezione

- I generics sono uno strumento per scrivere classi (e metodi) ***parametriche rispetto ad un tipo***
- Ci concentriamo soprattutto su come usare classi generiche
  - Al termine del corso lo studente dovrà essere in grado di usare classi generiche (in particolare quelle del package `java.util` relative alla gestione di collezioni di oggetti)
- La progettazione di classi generiche va oltre gli obiettivi del corso
  - Anche se, da un punto di vista puramente didattico, risulta invece utile introdurre i *Generics* progettando una semplice classe contenitrice di oggetti: **Coppia**

# Introduzione

- Notare che le coppie sono una forma molto rudimentale di *collezione*
- In effetti i *Generics* furono introdotti in Java 5 proprio per migliorare il JCF, la libreria dedicata alle collezioni già presente sin da Java 2 (ovvero 1.2)
- Obiettivo: una classe generica rispetto al tipo dei due oggetti ospitati, purché sia lo *stesso* per entrambi. Ad esempio la classe dovrà gestire:
  - Coppie di stringhe (istanze della classe **String**)
  - Coppie di attrezzi (istanze della classe **Attrezzo**)
  - Coppie di URL (istanze della classe **URL**)
  - ...

# La Classe Generica Coppia

- La classe **Coppia** deve offrire dei servizi per gestire una coppia di oggetti del **medesimo** tipo:
  - Metodi per ottenere/cambiare
    - il primo elemento della coppia
    - il secondo elemento della coppia
  - Un costruttore che riceve come parametri due riferimenti ad oggetti del medesimo tipo

# Una Possibile Soluzione Basata sul Polimorfismo

- Una possibile soluzione (l'unica possibile prima dell'introduzione dei *Generics* in Java 5) consiste nello sfruttare il polimorfismo, ed in particolare il principio di sostituzione
- Definiamo una classe che gestisce una coppia di oggetti istanza di **Object**
  - per il principio di sostituzione (e per la gerarchia dei tipi Java a singola radice in **Object**) la nostra classe può gestire coppie di oggetti istanza di qualsiasi classe (in quanto sottotipo di **Object**)

# La Coppia di Object

```
public class Coppia {  
    private Object primo;  
    private Object secondo;  
  
    public Coppia() {}  
  
    public Coppia(Object primo, Object secondo) {  
        this.primo = primo;  
        this.secondo = secondo;  
    }  
  
    public Object getPrimo() {  
        return this.primo;  
    }  
  
    public Object getSecondo() {  
        return this.secondo;  
    }  
  
    public void setPrimo(Object primo) {  
        this.primo = primo;  
    }  
  
    public void setSecondo(Object secondo) {  
        this.secondo = secondo;  
    }  
}
```

# Tipo degli Elementi Ospitati

- Considereremo di seguito del codice che fa riferimento alla semplice classe **Persona**

```
class Persona {  
    private String nome;  
  
    public Persona(String nome) {  
        this.nome = nome;  
    }  
  
    public String getNome() {  
        return this.nome;  
    }  
}
```

# Controllo sui Tipi Senza Generics: Scomodi (ed Inutili?) Downcast

- Il seguente codice compila, e funziona correttamente:

```
public class CoppiaSenzaGenericsTest {  
    @Test  
    public void testCheCompilaEdEsegue() {  
        Coppia coppia = new Coppia();  
        String pippo = new String("Pippo");  
        String pluto = new String("Pluto");  
        Persona p1 = new Persona(pippo);  
        coppia.setPrimo(p1);  
        Persona p2 = new Persona (pluto);  
        coppia.setSecondo(p2);  
        Persona persona = (Persona)coppia.getPrimo();  
        assertSame(pippo, persona.getNome());  
    }  
}
```

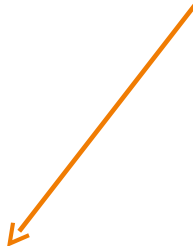


# Controllo sui Tipi Senza Generics: non Compila Quando Vorremmo...

- Per un utilizzo più semplice delle coppie, vorremmo poter scrivere il seguente codice:

```
public class CoppiaSenzaGenericsTest {  
    @Test  
    public void testCheNonCompila() {  
        Coppia coppia = new Coppia();  
        String pippo = new String("Pippo");  
        String pluto = new String("Pluto");  
        Persona p1 = new Persona(pippo);  
        coppia.setPrimo(p1);  
        Persona p2 = new Persona(pluto);  
        coppia.setSecondo(p2);  
        assertSame(pippo, coppia.getPrimo().getNome());  
    }  
}
```

**NON COMPILA!**  
Il tipo statico  
`Object`  
non possiede il  
metodo `getNome()`



# Controllo sui Tipi Senza Generics: non Vorremmo che Compilasse!

- Al contrario, il seguente codice compila correttamente ma l'esecuzione fallisce:

```
public class CoppiaSenzaGenericsTest {  
    @Test  
    public void testCheCompilaMaNonEsegue() {  
        Coppia coppia = new Coppia();  
        String pippo = new String("Pippo");  
        String pluto = new String("Pluto");  
        Persona p1 = new Persona(pippo);  
        Persona p2 = new Persona(pluto);  
        coppia.setPrimo(pippo);  
        coppia.setSecondo(pluto);  
        assertSame(pippo, ( (Persona) coppia.getPrimo() ).getNome() );  
    }  
}
```

**ClassCastException a tempo di esecuzione!**

# Controllo sui Tipi Statici vs Dinamici

- Il problema nasce dal controllo a tempo dinamico operato dal downcast esplicito:
  - nell'oggetto **coppia** è atteso come primo elemento un oggetto di tipo dinamico **Persona**
- Invece vi si trova un riferimento ad un oggetto di tipo dinamico **String**
- Tutto perfettamente lecito per il compilatore che effettua verifiche solo sul tipo statico:
  - l'istruzione **coppia.setPrimo(pippo)** riceve come parametro attuale la variabile locale **pippo**, di tipo statico **String**, sottotipo del tipo **Object** atteso

# Tipizzazione Lasca

- A ben vedere sono tutte conseguenze di una tipizzazione *lasca*
  - ✓ **obiettivo:** *coppie di oggetti dello stesso tipo*
- A differenza di come originariamente desiderato, per ovviare alla scarsa espressività del sistema dei tipi Java, siamo finiti per progettare una classe che può ospitare un coppia di oggetti qualsiasi
  - ✓ **risultato:** *coppie di oggetti non necessariamente dello stesso tipo!*
- Solo un'approssimazione del tipo desiderato
  - pratica frequentemente utilizzata precedentemente all'introduzione dei *Java Generics* nella piattaforma Java

# Conseguenze della Tipizzazione Lasca

- Un controllo lasco dei tipi a tempo di compilazione comporta almeno le seguenti conseguenze:
  - Costringe ad inserire downcast ogni volta che accediamo ad un elemento della coppia
    - ✓ Prima di Java 5 molti sviluppatori consideravano i downcasting quantomeno *ineleganti*, ma pochi lo consideravano un sostanziale limite, nella pratica, del linguaggio
  - Rimanda a tempo di esecuzione alcuni errori che risulterebbero facilmente rilevabili già a tempo di compilazione con una migliore analisi dei tipi statici
    - ✓ Questo è il vero problema! riconsiderare il costo dei bug rispetto al costo degli errori di compilazione>>
- In fase di definizione della coppia, si vorrebbe poter specificare un unico tipo per *entrambi* i riferimenti ad oggetti che la coppia è destinata a memorizzare

# Tipi e Metodi Generici

- I generics sono uno strumento per scrivere classi, ed interfacce, il cui tipo diventa *parametrico rispetto ad uno o più tipi*
- Si applica anche ai metodi per renderne parametrica la segnatura
- Nella definizione di un tipo generico, il codice viene scritto in maniera parametrica rispetto ad un *tipo formale*
- Nell'uso di un tipo generico, il tipo *formale* deve essere istanziato con un tipo *attuale* per renderlo effettivamente utilizzabile e completamente definito

# La Classe Generica Coppia<T> (1)

- La definizione di una classe generica prevede la dichiarazione del parametro formale di tipo racchiuso tra parentesi acute

```
public class Coppia<T> {  
    ...  
}
```

- In questo modo si specifica che all'interno della definizione della classe **Coppia**, il simbolo **T** indica il tipo sulla base del quale la definizione di classe è parametrica
- Convenzione sul nome del parametro formale di tipo
  - Singola lettera maiuscola: **T**, **E**, **V** ...

# La Classe Generica Coppia<T> (2)

- Nella definizione di campi e metodi all'interno della classe **T** viene usato come una dichiarazione di tipo:

```
public class Coppia<T> {  
    private T primo;  
    private T secondo;  
  
    public Coppia(T primo, T secondo) {  
        this.primo = primo;  
        this.secondo = secondo;  
    }  
  
    public T getPrimo() {  
        return this.primo;  
    }  
  
    public T getSecondo() {  
        return this.secondo;  
    }  
    ...  
}
```



# Oggetti Coppia<T> *Mutabili*

```
public class Coppia<T> {  
    private T primo;  
    private T secondo;  
  
    public Coppia(T primo, T secondo) {  
        this.primo = primo;  
        this.secondo = secondo;  
    }  
  
    public T getPrimo() {  
        return this.primo;  
    }  
  
    public T getSecondo() {  
        return this.secondo;  
    }  
  
    public void setPrimo(T primo) {  
        this.primo = primo;  
    }  
  
    public void setSecondo(T secondo) {  
        this.secondo = secondo;  
    }  
}
```

# Usare una Classe Generica (T=Persona)

- Quando usiamo una classe generica, dobbiamo *istanziarne completamente il tipo* fornendo il tipo *attuale* di tutti i tipi *formali* di cui fa uso
- Ad esempio, usiamo la nostra classe generica **Coppia<T>**, per gestire coppie di oggetti **Persona**

```
public class CoppiaTest {  
    @Test  
    public void testCoppiaDiPersone() {  
        Coppia<Persona> coppia;  
        Persona p1 = new Persona("Stanlio");  
        Persona p2 = new Persona("Olio");  
        coppia = new Coppia<Persona>(p1, p2);  
        assertEquals(p1, coppia.getPrimo());  
        assertEquals(p2, coppia.getSecondo());  
    }  
}
```

# Usare una Classe Generica (T=Color)

- Vediamo la classe parametrica `Coppia<T>` istanziata su un altro tipo qualsiasi (ad es. `java.awt.Color`)

```
import java.awt.Color;
```

```
...
```

```
public class CoppiaTest {
```

```
    @Test
```

```
    public void testCoppiaDiColori() {
```

```
        Coppia<Color> colori;
```

```
        Color rosso = new Color(255,0,0);
```

```
        Color blue = new Color(0,0,255);
```

```
        colori = new Coppia<Color>(rosso, blue);
```

```
        assertEquals(rosso, colori.getPrimo());
```

```
        assertEquals(blue, colori.getSecondo());
```

```
    }
```

```
}
```

# Controllo sui Tipi, con Generics

- Riconsideriamo il codice di prima, quello che avremmo voluto *non* compilasse affatto:

```
public class CoppiaTest {  
    @Test  
    public void testCheSmetteDiCompilare() {  
        Coppia<Persona> coppia = new Coppia<Persona>();  
        String pippo = new String("Pippo");  
        String pluto = new String("Pluto");  
        Persona p1 = new Persona(pippo);  
        Persona p2 = new Persona(pluto);  
        coppia.setPrimo(pippo);  
        coppia.setSecondo(pluto);  
        assertSame(pippo,  
                    ((Persona) coppia.getPrimo()).getNome());  
    }  
}
```

NON COMPILA!

# Tipo Formale - Tipo Attuale

- Non è difficile trovare una similitudine tra
  - il concetto di parametro formale/attuale inerente l'invocazione dei metodi
  - il concetto di tipo formale/attuale inerente la tipizzazione di classi generiche
- Solo superficialmente sono concetti simili: tra le tante differenze, non dimenticare mai la prima e più importante:
  - il legame tra parametri formali/attuali è operato dalla JVM a tempo di esecuzione
  - il legame tra tipi formali/attuali è operato dal compilatore a tempo di compilazione *solo* sulla base dei tipi *statici*

# Generics a più Parametri

- È possibile definire classi, interfacce e metodi generici con più parametri di tipo
  - Sintatticamente, si separano i vari parametri con una virgola

```
public class Esempio<T, S> {...}
```

# Generics e Tipi Primitivi

- Java è un linguaggio ibrido ci sono informazioni che si rappresentano senza utilizzare oggetti, ad es. `int`
- Non è possibile istanziare i tipi di una classe, di una interfaccia o di un metodo generico con tipi primitivi
- Per ovviare al problema è possibile rappresentare i tipo primitivi mediante le cosiddette *classi wrapper*

```
public class TestCoppia {  
    @Test  
    public void testCheNonCompila() {  
        Coppia<int> coppia;           // ERRORE: NON COMPILA!  
        int i1 = 100;  
        int i2 = 200;  
        coppia = new Coppia<int>(i1, i2); // ERRORE  
    }  
}
```

# Classi *Wrapper* (1)

- Per ogni tipo primitivo esiste una corrispondente classe *wrapper* che consente di «oggettificare» il dato primitivo, costruendoci un oggetto «attorno», per ospitarlo:
  - **int** → **Integer**
  - **double** → **Double**
  - **float** → **Float**
  - **char** → **Character**
  - **boolean** → **Boolean**



# Classi *Wrapper* (2)

```
int i;  
i = 18;  
Integer iwrap = new Integer(i);
```

...

```
int value = iwrap.intValue();
```

...

"scarto" il  
valore

il valore della variabile  
`int` è "avvolto" in un  
oggetto `Integer`



# Classi *Wrapper* (3)

- Le classi wrapper sono definite nel package `java.lang`
  - quindi non è necessario importarle esplicitamente
  - per approfondimenti sui dettagli dei loro metodi è sufficiente vedere la documentazione
- Metodi più frequentemente usati:
  - metodi `xxxValue()`
  - metodi `valueOf()` e `parseXxx()`
  - metodo `equals()`

# Generics e Tipi Primitivi

- Esempio:

```
public class TestCoppia {  
    @Test  
    public void testCheCompila() {  
        Coppia<Integer> coppia;           // OK  
        Integer i1 = new Integer(100);  
        Integer i2 = new Integer(200);  
        coppia = new Coppia<Integer>(i1, i2); // OK  
    }  
}
```

✓ Decisamente prolisso

# Boxing/Unboxing

- Per ovviare alla eccessiva verbosità, dalla versione 5 di Java (non a caso la stessa dell'introduzione dei *generics*), la gestione di oggetti wrapper è semplificata dalle funzionalità di *boxing* e *unboxing*
- In sostanza una tecnica di conversione automatica di tipi primitivi nei corrispondenti oggetti di un tipo *wrapper* e viceversa
  - per certi aspetti simile alla promozione di tipi che già si opera ad es. da `int` a `float`
  - per altri ancora, differente: sono coinvolti sia tipi che non sono oggetti, sia tipi che invece lo sono

# Boxing

- *Boxing*: è possibile assegnare direttamente tipi primitivi a oggetti wrapper
- Le seguenti istruzioni sono equivalenti:

```
int i = 0;  
Integer iWrap;  
iWrap = i;  
iWrap = 5;
```

```
int i = 0;  
Integer iWrap;  
iWrap = new Integer(i);  
iWrap = new Integer(5);
```

- È il compilatore che inserisce automaticamente le istruzioni per gestire boxing/unboxing!

# Unboxing

- *Unboxing*: è possibile assegnare direttamente oggetti wrapper a tipi primitivi
- Le seguenti istruzioni sono equivalenti:

```
int i = 0;  
Integer iWrap;  
iWrap = 5;  
i = iWrap;
```

```
int i = 0;  
Integer iWrap;  
iWrap = new Integer(5);  
i = iWrap.intValue();
```

- È il compilatore che inserisce automaticamente le istruzioni per gestire boxing/unboxing!

# Boxing, Unboxing e Collezioni

- Grazie a boxing e unboxing, anche la gestione di collezioni che memorizzano informazioni riconducibili a tipi primitivi risulta semplificata
- Le seguenti operazioni sono lecite (sempre grazie a boxing e unboxing):

```
Coppia<Integer> c;  
c = new Coppia<Integer>();  
int i = 4;
```

```
c.setPrimo(i);  
c.setSecondo(5);
```



```
c.setPrimo(new Integer(i));  
c.setSecondo(new Integer(5));
```

The diagram illustrates the process of boxing. A dashed box on the left contains the code `c.setPrimo(i);` and `c.setSecondo(5);`. A solid line with a downward arrow points from this box to another dashed box on the right, which contains the equivalent code using boxed `Integer` objects: `c.setPrimo(new Integer(i));` and `c.setSecondo(new Integer(5));`.

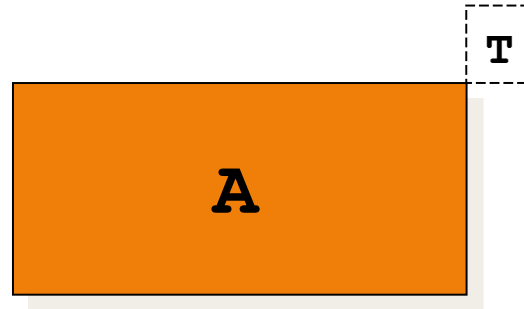
# Attenzione allo Zucchero (Sintattico)

- Le versioni più recenti del linguaggio hanno semplificato la gestione dei tipi primitivi nascondendo alcune conversioni di tipo
- Tuttavia, è necessario comprendere a fondo
  - la differenza tra il concetto di tipo primitivo e la loro controparte ad oggetti, i wrapper
  - quali operazioni non sono necessarie solo grazie ai servizi offerti dalle ultime versioni del compilatore Java (sarebbero necessarie con versioni precedenti)
  - quali operazioni il compilatore inserisce per conto nostro
- Perché conviene avere queste competenze?
  - per stimare meglio il numero di oggetti creati dalle nostre applicazioni
  - per migliorare la nostra capacità di ricerca delle origine degli errori sia a tempo di compilazione che di esecuzione
  - per riuscire ad usare versioni precedenti del compilatore

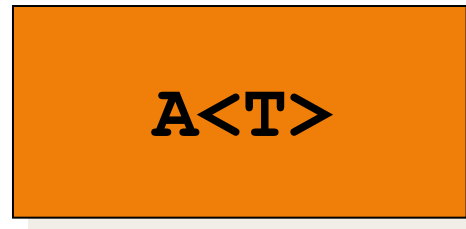


# Generics: Rappresentazione Diagrammatica

- Rappresentazione diagrammatica di una classe generica



- oppure



# Generics: Wildcard (1)

- A seguire approfondiamo alcuni concetti più avanzati relativi ai generics per mezzo di esempi
- N.B. In questo corso ci concentriamo sull'uso di classi generiche e non sulla loro progettazione
  - Ma l'utilizzo di alcune librerie pressuppone la capacità di comprendere alcun tipi e signature di metodi generici
  - ✓ sviluppare la propria capacità di astrazione è comunque sempre importante nella programmazione

# Generics: Wildcard (2)

- Aggiungiamo alla classe `Coppia<T>` il metodo `copyAll()`
  - copia nella coppia corrente gli elementi di un'altra coppia che viene passata come parametro
- La coppia che passiamo come parametro per «fornire» gli elementi da copiare deve essere istanziata su un qualunque sottotipo degli oggetti della coppia corrente che finirà per ospitarli
- Questa particolarità si esprime con il carattere jolly `?` e (di nuovo) con la parola chiave **`extends`**

# Generics: Upper-Bounded Wildcard (1)

- Vediamone la segnatura del metodo di istanza contenuto della classe generica `Coppia<T>`:

```
public class Coppia<T> {  
    ...  
    public void copyAll(Coppia<? extends T> c)  
    ...  
}
```

- ✓ Cosa significa questa segnatura?
- ✓ Di che tipo deve essere il parametro?

# Generics: Upper-Bounded Wildcard (2)

**Coppia<? extends T> c**

- Significa: un riferimento ad un oggetto (qui usato come parametro di un metodo) del tipo generico **Coppia** istanziato sullo *stesso* tipo **T**, o su un suo sottotipo, su cui è istanziata la coppia/oggetto *corrente* **Coppia<T>**

- Esempio di utilizzo:

```
Coppia<Strumento> strumenti;
```

```
Coppia<Chitarra> chitarre;
```

```
...
```

```
strumenti.copyAll(chitarre); // OK! T = ???
```

- Definizione del metodo **copyAll()** (nella classe **Coppia<T>**):

```
public void copyAll(Coppia<? extends T> coppia) {
```

```
    this.setPrimo(coppia.getPrimo());
```

```
    this.setSecondo(coppia.getSecondo());
```

```
}
```

# Metodi Statici Generici

- È possibile definire anche metodi statici generici (cioè parametrici rispetto ad un tipo formale)
- Un metodo generico definisce i tipi formali nella segnatura del metodo, subito prima del tipo restituito. Ad es.:

```
public static <T> int metodoGenerico(  
                                Coppia<T> c,  
                                T e  
                                )
```

- ✓ Anche in questo caso, come già per le classi generiche, è possibile invocare il metodo solo dopo aver fornito (a tempo statico, in fase di compilazione) tutti i tipi attuali necessari a completare definitivamente la segnatura

# Metodi Generici: Wildcard

- Definiamo ora la classe **Coppia****s**, classe contenitrice per ospitare metodi generici e di utilità generale per manipolare oggetti di tipo **Coppia**
- In particolare la classe **Coppia****s** offre alcuni metodi (statici), di cui stiamo per definire le signature, perseguendo la loro generalità rispetto al tipo degli oggetti ospitati nelle coppie:
  - **??? reverse(??? coppia)**  
prende come parametro una coppia e ne inverte gli elementi (il primo elemento diventa il secondo e viceversa)
  - **??? fill(??? coppia, ??? e)**  
prende due parametri: una coppia ed un elemento; riempie entrambi gli elementi della coppia con l'elemento ricevuto

# Metodi Generici: Esempio

```
public class Coppias {  
    public static <T> void reverse(Coppia<T> c) {  
        T tmp;  
  
        tmp = c.getPrimo();  
        c.setPrimo(c.getSecondo());  
        c.setSecondo(tmp);  
    }  
    ...  
}
```



# Generics: Lower-Bounded Wildcard (1)

- Il metodo `fill(??? coppia , ??? e)`
  - imposta entrambi gli elementi della coppia che viene passata come primo parametro, con lo stesso riferimento ad oggetto nel secondo parametro
- E' un metodo parametrico: il tipo del secondo parametro deve essere un qualunque sottotipo del tipo istanziato dalla coppia
  - ✓ ovvero: il tipo su cui la coppia è istanziata deve essere un supertipo del tipo del parametro
- Si esprime così:

```
static <T> void fill(Coppia<? super T> coppia,  
                    T elemento)
```

# Generics: Lower-Bounded Wildcard (2)

`Coppia<? super T>`

- ✓ Significa: un riferimento ad un oggetto `Coppia` istanziato su `T` o su un qualsiasi supertipo di `T`

- Esempio di utilizzo:

```
Coppia<Strumento> strumenti;
```

# Generics: Bounded Wildcard

```
static <T> void copy(Coppia<? super T> dest,  
                    Coppia<? extends T> src)
```

✓ Attenzione a non confonderlo con `copyAll()`

- Esempio di utilizzo:

```
Coppia<Strumento> strumenti = new Coppia<Strumento>();
```

```
Coppia<Chitarra> chitarre = new Coppia<Chitarra>();
```

```
...
```

```
Coppias.copy(strumenti, chitarre); // OK T = ???
```

- Definizione:

```
static <T> void copy(Coppia<? super T> dest,  
                    Coppia<? extends T> src) {
```

```
    dest.setPrimo(src.getPrimo());
```

```
    dest.setSecondo(src.getSecondo());
```

```
}
```

# La Regola Mnemonica *PECS*

- Semplice regola per ricordare il tipo dei parametri formali nelle signature di collezioni

*Producer Extends Consumer Super*

- Utilizzare `<? extends T>` per i parametri di collezioni che “producono” elementi
  - ad es. il parametro del metodo `copyAll()`
- Utilizzare `<? super T>` per i parametri di collezioni che “consumano” elementi
  - ad es. il parametro del metodo `fill()`
- Esempio di utilizzo contestuali di entrambi: i due parametri del metodo statico `copy()`


# Personaggi ed Interpreti

- Ogni riferimento ad API esistenti o a metodi realmente esistenti *NON* è affatto puramente casuale
  - Coppia nel ruolo di `Collection`, `List`...
  - Coppias nel ruolo di `Collections` (>>)
- Dentro `java.util.Collections` si trovano:

```
static <T> void fill(List<? super T> list, T obj)
static <T> boolean addAll(Collection<? super T> c,
                        T... elements)
```

```
static <T> void copy(List<? super T> dest,
                    List<? extends T> src)
```

```
static void reverse(List<?> list)
```



Preferibile quando non è strettamente necessario dare un nome al tipo formale

# Esercizi

- Scrivere il codice della classe generica **Coppia<T>**
- Scrivere il codice della classe **Coppias**
- Scrivere, utilizzando JUnit, classi di test per le classi **Coppia<T>** e **Coppias**
- Cercare di capire sino al dettaglio quante più possibili signature generiche dei metodi statici fornite nella classe **java.util.Collections**

# Esercizi

- Supponendo che **Studiante** sia sottotipo di **Persona**, è vero che **Coppia<Studiante>** risulta essere sottotipo di **Coppia<Persona>**?
- Ripetere l'esercizio di sopra, nel caso di coppie *immutabili*, ovvero prive dei metodi **setXXX()**
- Il tipo di **Coppia<T>** è *covariante* o *controvariante* rispetto al tipo **T**?
  - Nel caso di coppie *mutabili* ?
  - Nel caso di coppie *immutabili* ?

# Riferimenti ed Approfondimenti

- Alcuni articoli spiegano *molti* altri dettagli:

<http://www.oracle.com/technetwork/java/javase/generics-tutorial-159168.pdf>

- Per sapere (quasi) tutto sui java generics:

<http://www.angelikalanger.com/GenericsFAQ/JavaGenericsFAQ.html>