

Prato – via Pomeria,90

# Java & Generics

*Alcune novità introdotte a partire dalla versione 5.0*

**Flavio Casadei Della Chiesa**  
fcasadei@gmail.com



PRATO LINUX USER GROUP

# Obiettivi

- Presentare i generics a programmatori *old-fashion*
- Elencare le funzionalità di base dei generics
- Studiare le “trappole” dei generics
- Non verranno fornite spiegazioni o motivazioni sul perché classi e metodi generici siano migliori o peggiori della controparte “legacy”



PRATO LINUX USER GROUP

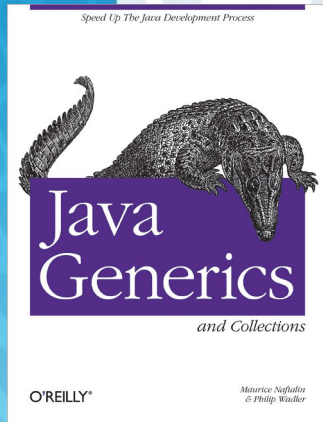
# Argomenti trattati

- 1) Alcune novità rispetto a java 1.4
- 2) Subtyping & Wildcards
- 3) Confronti tra elementi
- 4) Dichiarazioni di classi generiche
- 5) Evoluzione, non rivoluzione
- 6) Reification

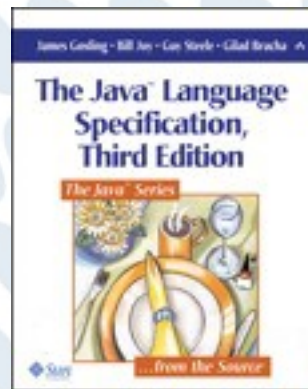


PRATO LINUX USER GROUP

# Riferimenti ...



Java Generics and Collections – O'Reilly



Java Language Specification - SUN



PRATO LINUX USER GROUP

# Cominciamo

Generics e Collections vanno a braccetto con altre nuove *feature* (introdotte da Java5):

- *boxing* e *unboxing*;
- nuovo ciclo **for**;
- funzioni che accettano un numero variabile di argomenti.

La loro combinazione è *sinergica*: l'intero è maggiore della somma delle parti!



PRATO LINUX USER GROUP

# Un primo esempio 1/

```
List<Integer> ints = Arrays.asList(1,2,3);  
int s = 0;  
for (int n : ints) {  
    s += n;  
}  
assert s == 6;
```

Probabilmente il codice è comprensibile anche senza alcuna spiegazione ...



PRATO LINUX USER GROUP

# Un primo esempio 2/

List ed ArrayList sono parte del CollectionFramework.

List è *generico*: è possibile scrivere List<E> per indicare liste contenenti elementi di tipo E.

List<Integer> indica liste contenenti elementi di tipo Integer.

```
List<Integer> ints =  
Arrays.asList(1,2,3);
```

```
int s = 0;
```

```
for (int n : ints) {
```

```
    s += n;
```

```
}
```

```
assert s == 6;
```



PRATO LINUX USER GROUP

# Un primo esempio 3/

*boxing ed unboxing  
automagicamente applicati*

```
List<Integer> ints =  
Arrays.asList(1,2,3);
```

```
int s = 0;
```

```
for (int n : ints) {
```

```
    s += n;
```

```
}
```

```
assert s == 6;
```



PRATO LINUX USER GROUP



# Un primo esempio 4/

Il metodo statico `Arrays.asList` prende un numero variabile di argomenti.

```
List<Integer> ints =  
Arrays.asList(1,2,3);
```

```
int s = 0;
```

```
for (int n : ints) {
```

```
    s += n;
```

```
}
```

```
assert s == 6;
```



PRATO LINUX USER GROUP

# Un primo esempio 5/

Ciclo *foreach*: permette di associare ad una variabile i valori contenuti in un oggetto *iterabile*

Ad ogni iterazione si prende l'elemento successivo

```
List<Integer> ints =  
Arrays.asList(1,2,3);
```

```
int s = 0;
```

```
for (int n : ints) {
```

```
    s += n;
```

```
    }
```

```
assert s == 6;
```



PRATO LINUX USER GROUP

# Un primo esempio 6/

Lo statement `assert` permette di controllare che l'asserzione fatta sia corretta ...

- Asserzione vera
- `AssertionError`
- ... disabilitate di default ...

```
List<Integer> ints =  
Arrays.asList(1,2,3);
```

```
int s = 0;
```

```
for (int n : ints) {
```

```
    s += n;
```

```
}
```

```
assert s == 6;
```



PRATO LINUX USER GROUP

# Un primo esempio 7/

(codice old fashion)

```
List ints = Arrays.asList(new Integer [] {  
    new Integer(1),  
    new Integer(2),  
    new Integer(3) }  
);  
int s = 0 ;  
for (Iterator it = ints.iterator() ; it.hasNext() ; ){  
    int n = ((Integer)it.next()).intValue();  
    s += n;  
}  
assert(s == 6);
```



PRATO LINUX USER GROUP

# Un primo esempio 8/

Il precedente codice è meno leggibile del primo :-)

Senza i *generics* non è possibile indicare che tipo di elementi vogliamo inserire nella lista

- sono necessari dei cast :-)

Senza *boxing* ed *unboxing* è necessario eseguire manualmente la conversione

- esempio: `.intValue()`



PRATO LINUX USER GROUP

# Un primo esempio 9/

Senza funzioni ad argomenti variabili è necessario passare alla *asList* un *array* “preimpostato” :-)

Senza il ciclo *foreach* è necessario istanziare un iteratore per scandire la lista :-)

*Si c'è anche un altro modo.*



PRATO LINUX USER GROUP

# Pausa

Domande?  
Osservazioni?



PRATO LINUX USER GROUP

# Generics: base 1/

Una classe o interfaccia può dichiarare di ricevere uno o più parametri di tipo:

- Sono scritti tra parentesi angolate (<T>)
- I tipi attuali devono essere forniti
  - quando si dichiara una variabile
  - quando si istanzia un oggetto



PRATO LINUX USER GROUP



# Generics: base 2/

```
List <String> words = new ArrayList<String>();  
words.add("Hello ");  
words.add("world !");  
String s = words.get(0) + words.get(1);  
assert s.equals("Hello world !");
```



PRATO LINUX USER GROUP

# Generics: base 3/

```
List<String> words = new  
    ArrayList<String>();  
  
words.add("Hello ");  
words.add("world !");  
  
String s = words.get(0) +  
    words.get(1);  
  
assert s.equals(  
    "Hello world !");
```

La classe `ArrayList<E>`  
implementa l'interfaccia  
`List<E>`

*words* è una lista  
contenente stringhe

Vengono inserite due  
stringhe e  
successivamente vengono  
recuperate

... tutto senza cast!



PRATO LINUX USER GROUP

# Generics: base 4/

Codice senza l'ausilio dei *generics* ...

```
List words = new ArrayList();  
words.add("Hello ");  
words.add("world !");  
String s = (String)words.get(0) + (String)words.get(1);  
assert s.equals("Hello world !");
```



PRATO LINUX USER GROUP

# Type Erasure 1/

Il *bytecode* compilato dei due precedenti esempi è (*grossomodo*) identico → retro compatibilità

I generics sono implementati tramite la *type erasure*



PRATO LINUX USER GROUP

# Type Erasure 2/

Compile time	Run Time
List<integer>	List
List<String>	List
List<List<String>>	List
...	...

I generics eseguono implicitamente il cast che deve essere esplicitato nella versione senza generics



PRATO LINUX USER GROUP

# Type Erasure 3/

*Cast-iron guarantee:*

*I cast impliciti aggiunti dalla compilazione dei generics non falliscono mai! (\*)*

(\*) si applica esclusivamente al caso in cui non vengano inviati dal compilatore dei "unchecked warnings"



PRATO LINUX USER GROUP

# Type Erasure come mai?

*Semplicità* → il bytecode è identico

*Dimensioni* → c'è solo una classe List

*Evoluzione* → il bytecode (\*) è retro compatibile e le librerie con e senza generics possono coesistere

E' possibile *evolvere* il proprio codice “con calma e con pazienza”

*(\*) Solo se ricompilato per versioni vecchie di java*



PRATO LINUX USER GROUP

# Generics VS Template C++

Java Generics	C++ Template
List<List<String>>	List< List<String> > (con lo spazio!) :-( (?ottimizzazioni?)
<i>Type erasure</i> : una sola versione della classe	<i>Expansion</i> : <i>n</i> versioni della solita classe; una per ogni tipo definito a compile-time → <i>code bloat</i> :-( (?ottimizzazioni?)





# Tipi *reference*

Classi

Istanze

Array (tutti)

Possono assumere il valore *null*

Sono tutti figli di Object



PRATO LINUX USER GROUP

# Tipi primitivi

8 tipi primitivi hanno un corrispondente “tipo reference” nel package java.lang

Primitivo	Reference
byte	Byte
short	Short
int	<u>Integer</u>
long	Long
float	Float
double	Double
bool	<u>Boolean</u>
char	<u>Character</u>



PRATO LINUX USER GROUP

# Boxing e unboxing

*Boxing* → conversione da *primitivo* a *reference*

*Unboxing* → conversione da *reference* a *primitivo*

La conversione viene fatta in automatico

`int e` → `new Integer(e)` // *boxing*

`Integer e` → `e.intValue()` // *unboxing*



PRATO LINUX USER GROUP

# Esempi boxing/unboxing

```
// OK
public static int somma(List<Integer> ints){
    int s = 0;
    for ( int n :ints) { s += n ; }
    return s;
}
```

```
//Troppe conversioni! Performance :- (
public static Integer sommaInteger(List<Integer> ints)
{
    Integer s = 0;
    for ( int n :ints) { s += n ; }
    return s;
}
```



PRATO LINUX USER GROUP

# Binary Numeric Promotion

Se uno degli operandi è un *reference type* viene applicato l'unboxing

Poi

- Se uno degli operandi è **double** anche l'altro viene *promosso* a **double**
- Se uno degli operandi è **float** anche l'altro viene *promosso* a **float**
- Se uno degli operandi è **long** anche l'altro viene *promosso* a **long**
- Altrimenti entrambi vengono *promossi* a **int**

**Si applica a vari operatori binari (tra cui ==)**



PRATO LINUX USER GROUP



# Pericolo boxing/unboxing == 1/

Per i primitivi == significa “uguaglianza dei valori”

Per i reference == significa “stessa identità”

```
List<Integer> bigs =  
    Arrays.asList(100,200,300);  
assert sommaInteger(bigs) == somma(bigs);  
assert sommaInteger(bigs) !=  
    sommaInteger(bigs); // non raccomandato
```

I generics funzionano solo con i *reference*



PRATO LINUX USER GROUP



# Pericolo boxing/unboxing == 2/

Interi da -128 a 127, caratteri da 0 a \u007f, Byte e Boolean possono essere *cachati*

```
List<Integer> smalls = Arrays.asList(1,2,3);
```

```
assert sommaInteger(smalls)
```

```
== somma(smalls); // 6
```

```
assert sommaInteger(smalls)
```

```
== sommaInteger(smalls); // 6, non raccomandato
```



PRATO LINUX USER GROUP



# Pericolo boxing/unboxing == 3/

Posso assegnare *null* ad un primitivo? → NO



PRATO LINUX USER GROUP



# Due paroline sul ciclo *foreach*

For (Pippo p: pippi) ...

- Applicabile a istanze di *java.lang.Iterable*<T>
- Applicabile ad *array*[]
- Esegue in automatico eventuali *boxing* ed *unboxing*

```
int[] ints = {1,2,3,4};  
for (Integer i :ints)  
    System.out.println(i);
```



PRATO LINUX USER GROUP

# Pausa

Domande?  
Osservazioni?



PRATO LINUX USER GROUP

# Metodi Generici 1/

```
class Lists {  
    public static <T> List<T> toList(T[] arr){  
        List<T> list = new ArrayList<T>();  
        for (T elem: arr) list.add(elem);  
        return list;  
    }  
    ....  
}
```



PRATO LINUX USER GROUP

# Metodi Generici 2/

```
class Lists {  
    public static <T> List<T>  
    toList(T[] arr){  
        List<T> list = new  
        ArrayList<T>();  
        for (T elem: arr)  
            list.add(elem);  
        return list;  
    }  
}
```

...

Il metodo toList accetta un array di tipo T[] e ritorna un List<T> *per ogni* tipo T

Si deve indicare <T> all'inizio della firma del metodo *statico*

T è un parametro di tipo

Ogni metodo che dichiara un parametro di tipo è un *metodo generico*



PRATO LINUX USER GROUP

# Metodi Generici 3/

```
List<Integer> ints = Lists.toList(  
    new Integer[] {1,2,3});
```

```
List<String> strings = Lists.toList(  
    new String[] {"ciao", "mondo"});
```

Boxing e unboxing gratuiti!



PRATO LINUX USER GROUP

# Varargs 1/

Che noia inserire gli elementi nell'array!

```
public static <T> List<T> toList(T ... arr) {  
    List<T> list = new ArrayList<T>();  
    for (T elem: arr) list.add(elem);  
    return list;  
}  
...  
List<Integer> ints = Lists.toList( 1,2,2);  
List<String> strings = Lists.toList( "ciao", "mondo");
```



PRATO LINUX USER GROUP

# Varargs 2/

Abbiamo sostituito

- T[] con T ...
- L'array[] con valori separati da virgola

Qualsiasi numero di argomenti può precedere il *vararg*  
*niente deve seguire il vararg*



PRATO LINUX USER GROUP

# Varargs 3/

Attenzione! Il tipo T non viene sempre dedotto dal compilatore, a volte è necessario esplicitarlo

```
Lists.<Object>toList( 1, "mondo");
```

Non è detto che Integer e String abbiano in comune solo Object! (Serializable, Comparable, ...)



PRATO LINUX USER GROUP



# Asserzioni

Possono essere abilitate tramite i flag della JVM **-ea** o **-enableassertions**

Altrimenti stanno a dormire ...



PRATO LINUX USER GROUP

# Parte II

## Subtyping & Wildcards



PRATO LINUX USER GROUP

# Subtyping (alcune nozioni)

In Java un tipo A è un sottotipo di un altro tipo B se questi sono legati da una clausola *extends* o *implements*

*A extends B* o *A implements B*

(Integer è sottotipo di Number, List<E> è sottotipo di Collection<E>)



PRATO LINUX USER GROUP

# Subtyping (alcune nozioni)

Subtyping è riflessiva e transitiva

*Se A è sottotipo di B allora B è supertipo di A*

Ogni tipo *reference* è sottotipo di Object ed Object è supertipo di ogni *reference type*



PRATO LINUX USER GROUP

# Principio di sostituzione

- I) Ad una variabile di un certo tipo T può essere assegnato un valore di qualsiasi sottotipo del tipo T
- II) Un metodo che ha un parametro di tipo T può essere invocato con un argomento il cui tipo è un sottotipo di T



PRATO LINUX USER GROUP

# Principio di sostituzione 1/

```
interface Collection<E>{  
    ...  
    public boolean add(E elem);  
    ...  
}
```

Principio di sostituzione:  
possiamo aggiungere *Integer*  
e *Double* a collezioni di  
*Numbers*

(*Integer* e *Double* sono  
sottotipi di *Number*)



PRATO LINUX USER GROUP

# Principio di sostituzione 2/

```
List<Number> numeri = new  
    ArrayList<Number>();  
numeri.add(2);  
  
numeri.add(3.14);  
assert  
    numeri.toString().equals(  
        "[2, 3.14]");
```

OK List<Number> è sottotipo  
di Collection<Number>

OK 2 ha tipo\* Integer  
che è sottotipo di Number

OK 3.14 ha tipo\* Double  
che è sottotipo di Number

Per ora tutto OK



Eccoci al dunque ...



PRATO LINUX USER GROUP



# Wildcards con extends 1/

```
interface Collection<E>{  
...  
public boolean addAll(  
    Collection<? extends E> c);  
...  
}
```

OK, posso inserire in una collezione di tipo E elementi di tipo E

? *extends* E → OK, posso inserire in una collezione di tipo E elementi appartenenti ad una collezione di *qualsiasi sottotipo* di E



PRATO LINUX USER GROUP

# Wildcards con extends 2/

```
List<Number> numeri = new  
ArrayList<Number>();  
  
List<Integer> interi =  
Arrays.asList(1,2);  
  
List<Double> doubles =  
Arrays.asList(2.78,3.14);  
  
numeri.addAll(inter);  
numeri.addAll(doubles);
```

OK Integer e Double sono sottotipi di Number

*List<Integer> è sottotipo di List<? extends Number> (idem per Double)*



PRATO LINUX USER GROUP

# Wildcards con extends 3/

```
List<Integer> interi =  
Arrays.asList(1,2);
```

```
List<? extends Number> numeri  
= interi;
```

```
numeri.add(3.14); → Errore di  
compilazione
```

```
assert interi.toString().equals(  
"[1, 2, 3.14]");
```

List<? extends  
Number> può essere  
una lista di qualsiasi  
tipo di numero! Non è  
detto che sia una lista  
di Double!

:-)



PRATO LINUX USER GROUP

# Wildcards con super 1/

```
public static <T> void  
copia(List<? super T> dest,  
List<? extends T> src){  
  
    for (int i=0; i < src.size();  
i++) {  
  
        dest.set(i, src.get(i));  
  
    }  
}
```

? super T → lista  
destinazione di *supertipo* di T

← Attenzione



PRATO LINUX USER GROUP

# Wildcards con super 2/

```
List<Object> oggetti = Arrays.<Object>asList(2,3.14, "four");  
List<Integer> interi = Arrays.asList(5,6);  
Collections.copia(oggetti, interi);  
assert oggetti.toString().equals("[5, 6, four]");
```



PRATO LINUX USER GROUP

# Get and Put Principle

GET: Utilizzare il wildcard *extends* quando si deve solamente recuperare valori da una struttura

PUT: Utilizzare il wildcard *super* quando si deve solamente inserire dati in una struttura

GET and PUT: Non utilizzare i wildcard quando si deve sia prendere che inserire



PRATO LINUX USER GROUP

# Eccezioni al Get/Put principle

- ? **extends** E → è possibile inserire null
- ? **super** T → è possibile prelevare Object



PRATO LINUX USER GROUP

# Sporco trucco di programmazione

```
public void rebox(  
    Box<? extends Object> box) {  
    reboxHelper(box);  
}
```

Viene effettuata una chiamata ad un “helper” senza wildcard

```
private static <V> void  
reboxHelper(Box<V> box) {  
    box.put(box.get());  
}
```



PRATO LINUX USER GROUP



# Pausa

Domande?  
Osservazioni?



PRATO LINUX USER GROUP

# Array 1/

Il subtyping degli array in java è *covariante*:

S sottotipo di T → S[] sottotipo di T[]

Il codice sulla destra viene compilato

```
Integer[] interi =  
    new Integer[] {1,2,3};  
Number[] numeri = interi;  
numeri[2] = 3.14;
```



PRATO LINUX USER GROUP

# Array 2/

```
Integer[] interi = new  
Integer[] {1,2,3};  
Number[] numeri = interi;  
Numeri[2] = 3.14; → Oops  
// numeri--> 1 2 3.14
```

Exception in thread "main"  
java.lang.ArrayStoreException:  
java.lang.Double



PRATO LINUX USER GROUP

# Array 3/

```
List<Integer> interi=  
Arrays.asList(1,2,3);
```

```
List<Number> numeri  
= interi; → non compila!  
numeri.add(3.14);
```

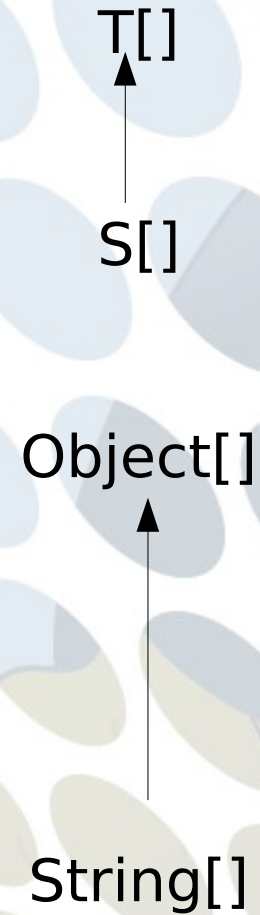
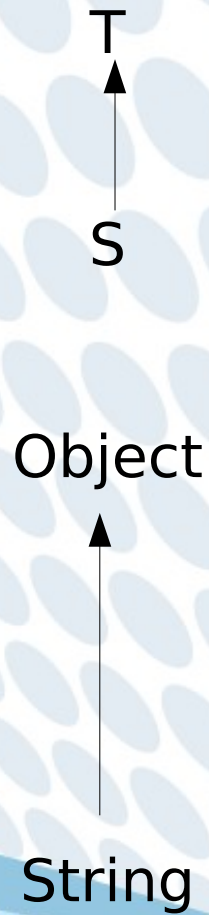
Subtyping per i generics è *controvariante*:

S supertipo di T → List<S> è sottotipo di List <? super T>



PRATO LINUX USER GROUP

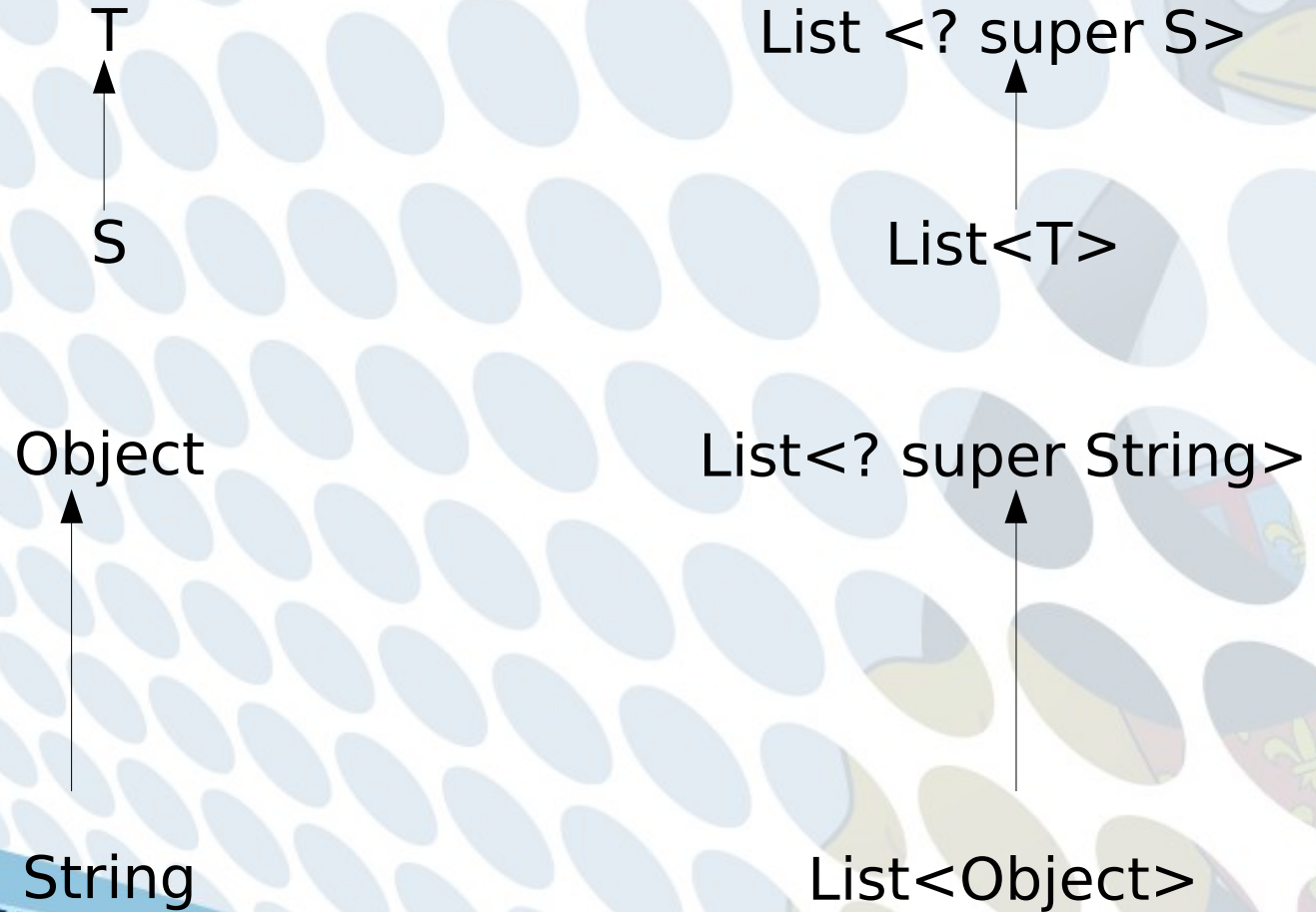
# Covariante



PRATO LINUX USER GROUP



# Controvariante



PRATO LINUX USER GROUP

# Cattura del wildcard 1/

Quando viene invocato un metodo generico il parametro di tipo deve essere scelto in modo da “combaciare” con il tipo rappresentato dal wildcard → *wildcard capture*



PRATO LINUX USER GROUP

# Cattura del wildcard 2/

```
public static <T> void  
reverse(List<T> list)
```

<?> è un sinonimo di  
*? extends Object*

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



PRATO LINUX USER GROUP



# Cattura del wildcard 3/

```
public static <T> void  
reverse(List<T> list){  
  
List<T> tmp = new ArrayList<T>(list);  
  
for (int i = 0 ; i < list.size() ; i+  
+){  
  
list.set(i, tmp.get(list.size() -i -  
1) );  
  
}  
  
}
```

```
public static void reverse(List<?>  
list){  
  
List<Object> tmp = new  
ArrayList<Object>(list);  
  
for (int i = 0 ; i < list.size() ;  
i++){  
  
list.set(i, tmp.get(list.size() -i -  
1) );  
  
}  
  
}
```



PRATO LINUX USER GROUP

# Cattura del wildcard 3/

```
public static <T> void
reverse(List<T> list){

List<T> tmp = new ArrayList<T>(list);
for (int i = 0 ; i < list.size() ; i+
++){
list.set(i, tmp.get(list.size() -i -
1) );
}
}
```

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

List<Object> tmp = new
ArrayList<Object>(list);
for (int i = 0 ; i < list.size() ;
i++){
list.set(i, tmp.get(list.size() -i -
1) );
}
}
```

--- ERRORE DI COMPILAZIONE ---

The method `set(int, capture#3-of ?)` in the type `List<capture#3-of ?>` is not applicable for the arguments `(int, Object)`



PRATO LINUX USER GROUP

# Restrizioni dei wildcard 1/

Creazione dell'istanza

Chiamata a metodo generico

Supertipo



PRATO LINUX USER GROUP

# Restrizioni dei wildcard: creazione

```
List<?> l = new ArrayList<?>();
```

NO! Errore di compilazione

```
List<? super Number > s =  
    new ArrayList<Number>();
```

OK



PRATO LINUX USER GROUP

# Restrizioni dei wildcard: chiamata e metodi generici

```
public class Lista {  
    public static <T> List<T>factory() { return new  
        ArrayList<T>(); }  
} .. OK  
  
List<?> l1 = Lista.factory(); OK  
List<?> l2 = Lista.<Object>factory(); OK  
List<?> l3 = Lista.<?>factory(); → ERRORE  
List<?> l4 = Lista.<List<?>>factory(); OK
```



PRATO LINUX USER GROUP

# Restrizioni dei wildcard: supertipi

Class Lista extends ArrayList<?> {...} → NO

Class Lista2 implements List<?> {...} → NO

Class List3 implements ArrayList<List<?>> {...} → OK



PRATO LINUX USER GROUP

# Parte III

## Confronti tra elementi (e limiti sui tipi)



PRATO LINUX USER GROUP

# Confronti

Parleremo di Comparable<T>

- Confrontare elementi
- Trovare massimo e minimo in una collezione
- Metodi *bridge*



PRATO LINUX USER GROUP



# Comparable

```
public interface Comparable<T>
{
    public int compareTo(T elem);
}
```

Restituisce un valore che è negativo, zero o positivo a seconda che il parametro fornito sia rispettivamente minore, uguale o maggiore del parametro implicito (this)

Quando una classe implementa *Comparable* l'ordine specificato dalla sua interfaccia è chiamato ordine naturale per la classe



PRATO LINUX USER GROUP

# Mele con mele ...

```
Integer i0 = 0;  
Integer i1 = 1;  
assert i0.compareTo(i1) < 0;
```

```
String s0 = "zero";  
String s1 = "uno";  
assert s0.compareTo(s1) > 0;
```

*(Di norma)* un oggetto di una classe può essere confrontato solo con oggetti della stessa classe

*(pere con pere, mele con mele)*



PRATO LINUX USER GROUP

# ... ma non mele con pere

```
Number n1 = 1;
```

```
Number np1 = 3.14;
```

```
assert n1.compareTo(np1) < 0;
```

Posso confrontare mele con pere?



PRATO LINUX USER GROUP

# ... ma non mele con pere

```
Number n1 = 1;
```

```
Number np1 = 3.14;
```

```
assert n1.compareTo(np1) < 0;
```

Posso confrontare mele con pere?

NO ← errore di compilazione



PRATO LINUX USER GROUP

# ... ma non mele con pere

```
Number n1 = 1;
```

```
Number np1 = 3.14;
```

```
assert n1.compareTo(np1) < 0;
```

Notare il boxing

Il codice non compila!

Number non implementa Comparable



PRATO LINUX USER GROUP

# Consistente con l'uguaglianza

`x.equals(y) ↔ x.compareTo(y) == 0`

Attenzione all'inserimento in `SortedList` ecc ...

`compareTo` VS `equals`

- `x.equals(null) → true, false`
- `x.compareTo(null) → NullPointerException`

`BigDecimal` non è consistente con l'uguaglianza



PRATO LINUX USER GROUP

# Contratto per Comparable

- Antisimmetrica

$$\mathit{sgn}(x.\mathit{compareTo}(y)) = - \mathit{sgn}(y.\mathit{compareTo}(x))$$

- Transitiva

$$x.\mathit{compareTo}(y) < 0 \ \& \ y.\mathit{compareTo}(z) < 0 \ \rightarrow \ x.\mathit{compareTo}(z) < 0$$

- Congruenza

$$x.\mathit{compareTo}(y) == 0 \ \rightarrow \ \mathit{sgn}(x.\mathit{compareTo}(z)) == \mathit{sgn}(y.\mathit{compareTo}(z))$$

- Riflessiva

$$x.\mathit{compareTo}(x) == 0$$

*sgn(x) è il segno di x: -1 negativo, 0 zero, 1 positivo*



PRATO LINUX USER GROUP

# Attenzione!

```
public class Integer implements  
Comparable<Integer>{
```

```
public int compareTo(Integer o) {  
return this.value < o.value ? -1 :  
this.value == o.value ? 0 : 1 ;  
}
```

OK

E' il modo giusto di  
confrontare interi



PRATO LINUX USER GROUP



# Attenzione!

```
public int compareTo(Integer o) {  
    return this.value - o.value;  
}
```

**NO!**

Può generare overflow:

confrontando un numero  
negativo grande in  
modulo con un grande  
numero positivo → Si può  
superare  
Integer.MAX\_VALUE



PRATO LINUX USER GROUP

# Overflow/Underflow

*Overflow*: Numero troppo grande in valore assoluto

Maggiore di `Integer.MAX_VALUE` o minore di `Integer.MIN_VALUE`

*Underflow*: numero, diverso da zero, troppo piccolo in valore assoluto per essere rappresentato dalla macchina



PRATO LINUX USER GROUP

# Trovare il massimo di una collezione

```
public static <T extends Comparable<T>> T max (Collection<T> coll){  
    T cand = coll.iterator().next();  
    for (T elem: coll){  
        if (cand.compareTo(elem) < 0) cand = elem;  
    }  
    return cand;  
}
```



PRATO LINUX USER GROUP

# Limiti

$\langle T \text{ extends Comparable}\langle T \rangle \rangle$  si dice che  $T$  è *limitato* da  $\text{Comparable } T$

Si è quindi posto un limite sul tipo di  $T$

Il limite può essere ricorsivo

$\langle T \text{ extends } C\langle T, U \rangle, U \text{ extends } D\langle T, U \rangle \rangle$



PRATO LINUX USER GROUP

# Esempi

```
List<String> stringhe =  
Arrays.asList("ciao", "mondo");
```

OK

```
assert  
Collections.max(stringhe).equals("mondo")  
);
```

OK

```
List<Integer> interi =  
Arrays.asList(1,2,3);
```

```
assert Collections.max(inter) == 2;
```

```
List<Number> numeri = Arrays.asList(1 ,2  
,3 ,4 ,5 ,3.14);
```

KO

```
assert Collections.max(numeri) == 5;
```

Il codice non compila



# Utilizzare firme il più possibile generiche

```
public static <T extends Comparable<T>> T max (Collection<T> coll)
```



```
public static <T extends Comparable<? super T>> T max (  
    Collection<? extends T > elements)
```



```
public static <T extends Object & Comparable<? super T>> T max(  
    Collection<? extends T> coll ) {
```



PRATO LINUX USER GROUP

# Limiti Multipli

```
public static <S extends Readable & Closeable,  
    T extends Appendable & Closeable>  
    void copy(S src, T dest, int dim) throws IOException {  
    CharBuffer buff = CharBuffer.allocate(dim);  
    int i = src.read(buff);  
    while (i >= 0 ) {  
        buff.flip(); // prepara per la scrittura  
        dest.append(buff);  
        buff.clear();  
        i = src.read(buff);  
    }  
    src.close(); dest.close();  
}
```



# Metodi Bridge 1/

Generics sono implementati mediante *type erasure*

- Il *bytecode* è piuttosto simile (e compatibile) con la versione senza Generics

Classi che implementano interfacce generiche (es: `Comparable<T>` )

- Vengono aggiunti dal compilatore alcuni metodi detti metodi *bridge*



PRATO LINUX USER GROUP



# Metodi Bridge 2/

```
public class Foo1 implements Comparable {  
    private final String value ;  
    public Foo1(String value) {  
        super();  
        this.value = value;  
    }  
    public int compareTo(Object o) {  
        return compareTo((Foo1)o);  
    }  
    public int compareTo(Foo1 o){  
        return value.compareTo(o.value);  
    }  
}
```

Esempio di Comparable senza Generics

Il metodo classico chiama il metodo ridefinito dopo un cast (*double dispatch*)

Attenzione! Java supporta il binding dinamico solamente sull'argomento implicito (*this*) e non sui parametri formali (quelli tra parentesi)



PRATO LINUX USER GROUP

# Metodi Bridge 3/

```
public class Foo2 implements
Comparable<Foo2> {

    private final String value;

    public Foo2(String value) {
        this.value = value;
    }

    public int compareTo(Foo2 o) {
        return value.compareTo(o.value);
    }
}
```

C'è solo un metodo nel codice sorgente

Vediamo cosa succede nel *bytecode* generato dal compilatore ...



PRATO LINUX USER GROUP

# Bytecode con metodo bridge

```
for (Method m :  
    Foo2.class.getMethods()){  
    if (m.getName()  
        .equals("compareTo")) {  
        System.out.println(  
            m.toGenericString());  
    } }  
-----
```

```
public int Foo2.compareTo(Foo2)
```

```
public bridge int Foo2.compareTo(java.lang.Object)
```



PRATO LINUX USER GROUP

# Override covariante (Finalmente!)

In java  $\leq 1.4$  un metodo può sovrascrivere un altro  $\leftrightarrow$  le due firme coincidono esattamente

In java  $\geq 5$  un metodo può sovrascrivere un altro se gli argomenti sono identici ed il tipo di ritorno del metodo riscrivente è un sottotipo del tipo di ritorno del metodo riscritto



PRATO LINUX USER GROUP

# Override <= 1.4

```
public class OldPunto {  
    private final int x;  
    private final int y;  
    public OldPunto(int x, int y) {  
        this.x = x;  
        this.y = y;  
    }  
    public Object clone() {  
        return new OldPunto(x, y);  
    }  
}
```

Object ha un metodo clone()  
che restituisce un Object



PRATO LINUX USER GROUP

# Override $\geq$ 5.0

```
public class NewPunto {  
    private final int x;  
    private final int y;  
    public NewPunto(int x, int y) {  
        this.x = x;  
        this.y = y;  
    }  
    public NewPunto clone() {  
        return new NewPunto(x, y);  
    }  
}
```

NewPunto è sottotipo di Object quindi non ci sono errori di compilazione



PRATO LINUX USER GROUP

# Vediamo se ci sono dei bridge

```
public NewPunto NewPunto.clone()  
public bridge java.lang.Object  
NewPunto.clone() throws  
java.lang.CloneNotSupportedException
```



PRATO LINUX USER GROUP

# Bridge

A seconda del compilatore la dicitura bridge:

- Può non comparire!
- Può essere sostituita da volatile! (bug)
- Altro ...
- Nel compilatore java6 di Apple non compare bridge



PRATO LINUX USER GROUP



# Parte IV

## Dichiarazioni di variabili (brevissimo)



PRATO LINUX USER GROUP

# Costruttori

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

In una classe generica i parametri di tipo appaiono nell'intestazione che dichiara la classe

*non nel costruttore!*



PRATO LINUX USER GROUP

# Costruttori

```
Coppia<String, Integer> c = new  
Coppia<String, Integer>("Ciao", 1);
```

OK

```
assert c.getPrimo().equals("Ciao") &&  
c.getSecondo().equals(1);
```

```
Coppia<String,Integer> c = new  
Coppia("Ciao",1);
```

Warning! UncheckedWarning

L'istruzione è legale ma  
genera un warning

(non è detto che valga la Cast Iron Guarantee)



PRATO LINUX USER GROUP

# Static 1/

```
List<Integer> interi =  
Arrays.asList(1,2,3);  
  
List<String> stringhe =  
Arrays.asList("ciao", "mondo");  
  
assert interi.getClass() ==  
stringhe.getClass();
```

I generics sono implementati  
tramite *type erasure*

List<String> e List<Integer>  
a tempo di compilazione sono  
List



PRATO LINUX USER GROUP

# Static 2/

*Type erasure* →

- variabili e metodi statici di classi generiche sono **condivise tra tutte le istanze** della classe
- le variabili e metodi statici di una classe generica **non possono** far riferimento ai parametri di tipo della classe
- accedendo ad un metodo statico di una classe generica il nome della classe **non deve** essere parametrizzato (con la variabile di tipo)



PRATO LINUX USER GROUP

# Static: esempio 1/

```
public class Cella<T> {  
    private final int id;  
    private final T valore;  
    public static int count = 0;  
    public static synchronized int getCount() { return count;}  
    public static synchronized int nextId() { return count++;}  
    public int getId() { return id; }  
    public T getValore() { return valore;}  
    public Cella( T valore) { this.id = nextId(); this.valore = valore;}
```



PRATO LINUX USER GROUP

# Static: esempio 2/

```
Cella<String>cs = new  
Cella<String>("ciao");  
  
Cella<Integer> ci = new  
Cella<Integer>(1);  
  
assert cs.getId() == 0  
    && ci.getId() == 1  
    && Cella.getCount() == 2;
```

Il contatore è condiviso tra tutte le istanze della classe Cella



PRATO LINUX USER GROUP

# Utilizzo di metodi statici

Cella.getCount() **OK**

Cella<Integer>.getCount() **NO**

Cella<?>.getCount() **NO**



PRATO LINUX USER GROUP



# Inner Class 1/

Java permette di annidare una classe dentro un'altra

Se la classe esterna ha un parametro di tipo T:

- Classe interna non statica → **può** accedere direttamente al parametro di tipo T
- Classe interna statica → **non può** accedere (*direttamente*) al parametro di tipo T *ma esiste un escamotage ...*



PRATO LINUX USER GROUP

# Inner class non statica

```
public class Esterna<G> {  
    ...  
    private class Interna1{  
        G g = null;  
        ...  
    }  
    ...  
}
```



PRATO LINUX USER GROUP

# Inner class statica: *escamotage*

```
public class Esterna<G> {  
    ...  
    public G getPippo() {  
        return new Interna2<G>().getPippo();  
    }  
    public static class Interna2<H>{  
        public H getPippo() {  
            ...  
        }  
        ...  
    }  
    ...  
}
```



PRATO LINUX USER GROUP

# Parte V

Evoluzione: SI



Rivoluzione: NO



PRATO LINUX USER GROUP

# Evoluzione, non rivoluzione

Migrare gradualmente il codice all'utilizzo dei *generics* (*evoluzione*) senza modifiche radicali (*rivoluzione*)



PRATO LINUX USER GROUP

# Evoluzione

L'implementazione dei *generics* permette che il vecchio codice venga compilato ed eseguito anche con l'utilizzo delle nuove librerie

L'evoluzione è più forte della retro compatibilità: non esistono *più versioni* delle classi e delle librerie *ma una sola versione* (grazie alla *type erasure*) compatibile



PRATO LINUX USER GROUP

# Evoluzione

Non sempre è possibile mettere mano a tutto il codice

- È costoso
- Non abbiamo accesso ai sorgenti
- Possiamo mettere le mani solo sul client o sulla libreria
- ...

E' quindi necessario procedere per passi



PRATO LINUX USER GROUP

# Raw type e generic type

I *raw types* sono la controparte *legacy* dei *generic types*

Generic	Raw
Stack<E>	Stack
List<E>	List
ArrayStack<E>	ArrayStack
...	...



PRATO LINUX USER GROUP



# Casi di evoluzione

Dovendo utilizzare *client* con *librerie* abbiamo 4 casi

Client / Libreria	Libreria Generic	Libreria Legacy
Client Generic	CG,LG	<b>CG,LL</b>
Client Legacy	<b>CL,LG</b>	CL,LL



PRATO LINUX USER GROUP

# Client Generic, Libreria Generic

Non pone grossi problemi (è il punto di arrivo)

Per arrivare a questo punto di norma tutte le occorrenze di Object vengono sostituite con opportuni parametri di tipo (solo dove ha senso)



PRATO LINUX USER GROUP

# Client Legacy, Libreria Legacy

Di norma è il punto di partenza dell'evoluzione in cui sono presenti solo *raw type*

```
public static void fai(List l){  
    // ...  
}
```

List is a raw type. References to generic type List<E> should be parameterized

3 quick fixes available:

- [Add type arguments to 'List'](#)
- [Infer Generic Type Arguments...](#)
- [Add @SuppressWarnings 'unchecked' to 'fai\(\)'](#)

Press 'F2' for focus

Writable Smart Insert 17 : 6



PRATO LINUX USER GROUP

# Client Legacy, Libreria generica

E' il caso più importante di retro compatibilità: il Collection framework di java 5 deve funzionare con i client di java 1.4

Per supportare l'evoluzione, per ogni parametro generico java riconosce anche il suo *raw type*

*Ogni tipo parametrico è sottotipo del corrispondente raw type*

Un oggetto "parametrico" può quindi essere utilizzato quando ci si aspetta un *raw type*



PRATO LINUX USER GROUP

# Unchecked conversion warning 1/

Di norma è un errore utilizzare un supertipo di T quando ci si aspetta un oggetto di tipo T

Eccezione alla regola: è possibile utilizzare un *raw type* al posto di uno dei corrispondenti tipi parametrici

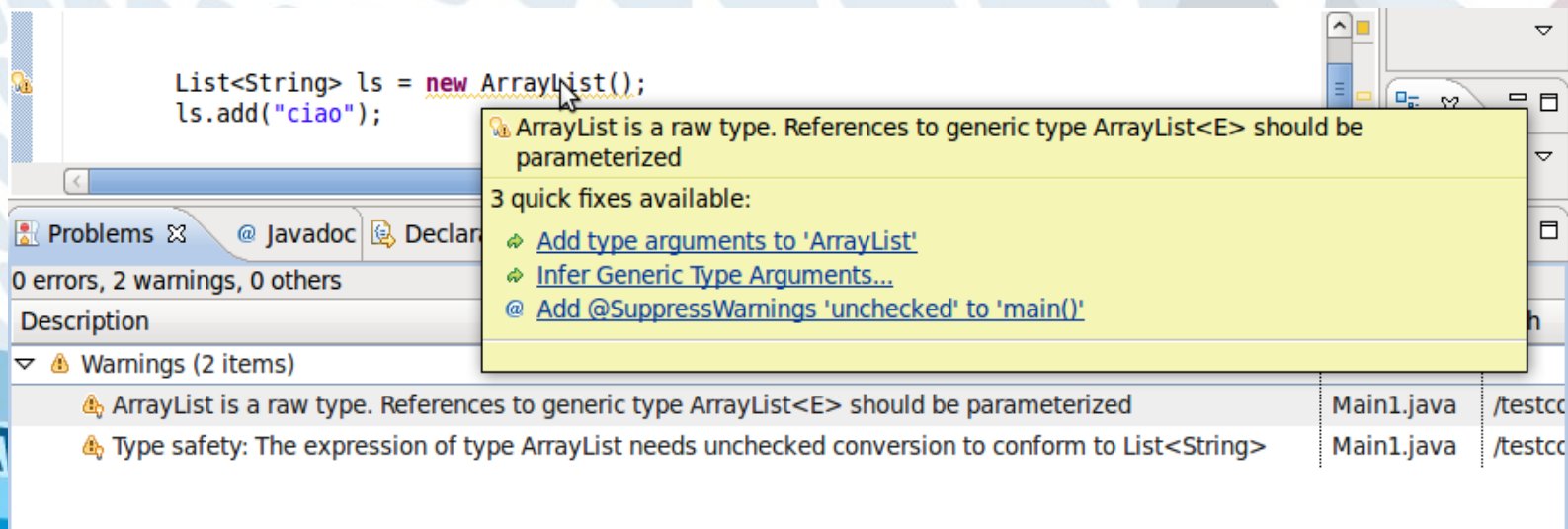
*Viene tuttavia generato un “unchecked conversion warning”*



PRATO LINUX USER GROUP

# Unchecked conversion warning 2/

List a = OK, normale  
`new ArrayList<String>();` sottotipizzazione  
List<String> x = OK ... ma genera un *Unchecked Conversion Warning*  
`new ArrayList();`



The screenshot shows an IDE window with the following code:

```
List<String> ls = new ArrayList();  
ls.add("ciao");
```

A tooltip is displayed over the `new ArrayList()` line, containing the message: "ArrayList is a raw type. References to generic type ArrayList<E> should be parameterized". Below the message, it lists "3 quick fixes available":

- [Add type arguments to 'ArrayList'](#)
- [Infer Generic Type Arguments...](#)
- [Add @SuppressWarnings 'unchecked' to 'main\(\)'](#)

The IDE's Problems view at the bottom shows two warnings:

- ArrayList is a raw type. References to generic type ArrayList<E> should be parameterized (Main1.java /testco)
- Type safety: The expression of type ArrayList needs unchecked conversion to conform to List<String> (Main1.java /testco)



# Unchecked conversion warning 3/

Indica che il compilatore non è in grado di offrire le stesse garanzie che sono possibili quando i generics vengono utilizzati in maniera uniforme

Vengono comunque assicurate le stesse (???) garanzie di quando non si utilizzano per niente i generics (!!!)

Non vale la *cast iron guarantee*



PRATO LINUX USER GROUP

# Client generico, Libreria Legacy

```
List l = new ArrayList<String>();
```

```
l.add(elem);
```

Il codice compila ma viene generato un *Unchecked Call Warning*

---

```
Pippo.java:7: warning: [unchecked] unchecked call to add(E) as a member of the raw type java.util.List
```

```
l.add(elem);
```



PRATO LINUX USER GROUP



# Se i warning vi danno doia

...

```
@SuppressWarnings("unchecked")
```



PRATO LINUX USER GROUP

# Parte VI

Reificazione (si è una parola Italiana)



PRATO LINUX USER GROUP

# Definizioni di reificazione

Processo mentale mediante il quale a concetti astratti viene assegnata consistenza di cose concrete

Nella filosofia di Marx, processo secondo cui nell'economia capitalistica l'uomo e il suo lavoro sono ridotti al valore della merce che producono

La reificazione è una fallacia o un'ambiguità quando un'astrazione (una credenza astratta o un costrutto ipotetico) viene trattata come se fosse un concreto evento reale o un'entità fisica

Nell'ambito dell'Ingegneria della conoscenza, e in particolare della definizione di ontologie, la reificazione è una rappresentazione indiretta che prevede l'utilizzo di determinate espressioni del linguaggio per descrivere entità del mondo reale intuitivamente associate a espressioni di tipo diverso. L'esempio tipico di reificazione è costituito dall'utilizzo di individui per rappresentare classi.

Nel contesto della programmazione orientata agli oggetti si definisce reificazione il procedimento di creazione di un modello di dati basato su un concetto astratto predefinito. Mediante la reificazione, un computer può compiere elaborazioni riguardanti un'entità astratta come se si trattasse di un insieme qualsiasi di dati di altro tipo



PRATO LINUX USER GROUP

# La reificazione è ...

...



PRATO LINUX USER GROUP

# In java

In java un tipo è *reificabile* se il tipo è completamente identificabile a *runtime*, ovvero se la *type erasure* non rimuove informazioni utili



PRATO LINUX USER GROUP

# Tipi reificabili

- Primitivi es: int
- Classi o interfacce non parametriche es: String, Comparable
- Tipi parametrici in cui tutti gli argomenti di tipo sono wildcard non limitati es: List<?> o Map<?,?>
- Raw type es: List, Map
- Array di elementi di tipo reificabile es: int[], List<?>[]



PRATO LINUX USER GROUP

# Tipi non rieficabili

- Variabili di tipo es: `T in Xxx<T>`
- Tipi parametrici con parametri attuali es: `List<String>`, `Comparable<Integer>`
- Tipi parametrici con un limite es: `List<? extends String>`, `Comparable <? super Number>`



PRATO LINUX USER GROUP

# Note

- In java il tipo di un array viene *reificato* con il tipo dei suoi componenti
- Un tipo parametrico non viene *reificato* con il suo parametro di tipo
- `List<?>` è equivalente a `List<? extends Object>` (nel senso di passaggio di parametri) ma il primo è reificabile il secondo no



PRATO LINUX USER GROUP



# Instance test e cast

I test *instanceof* ed i *cast* dipendono dall'esame *a runtime* dei tipi degli oggetti coinvolti e quindi dalla *reificazione*

Test di istanza verso un tipo *non reificabile* → errore (di compilazione)

Cast ad un tipo *non reificabile* → genera (di norma) un warning



PRATO LINUX USER GROUP

# Esempio OK

```
public class Myinteger {  
    private final int value;  
    public boolean equals(Object o) {  
        if (o instanceof Myinteger){  
            return this.value ==  
                ((Myinteger)o).value;  
        } else return false;  
    }  
}
```

...

```
}
```

Mynteger è *reificabile* → compila

Myinteger è *reificabile* → il cast non genere warning

PRATO LINUX USER GROUP



# Equals su liste

```
public abstract class MiaLista<E>
    extends AbstractCollection<E>
    implements List<E>{
    public boolean equals(Object o){
        if ( o instanceof List<E>){
            Iterator<E> it1 = iterator();
            Iterator<E> it2 = ((List<E>)o ).iterator();
            while (it1.hasNext() && it2.hasNext()){
                E e1 = it1.next();
                E e2 = it2.next();
                if (! ( e1 == null ? e2== null: e1.equals(e2) ) ) {
                    return false;
                }
            }
            return !it1.hasNext() && !it2.hasNext();
        } else return false;
    }
}
```



# Equals su liste: analisi

```
public abstract class MiaLista<E>
    extends AbstractCollection<E>
    implements List<E>{
    public boolean equals(Object o){
    if ( o instanceof List<E>){
    Iterator<E> it1 = iterator();
    Iterator<E> it2 =
        ((List<E>)o ).iterator();
    while (it1.hasNext()
        && it2.hasNext()){
```

Errore di compilazione!

List<E> non è reificabile

Unchecked cast warning!



PRATO LINUX USER GROUP

# Equals su liste: versione *fixata*

```
public abstract class MiaListaOK<E>
    extends AbstractCollection<E>
    implements List<E>{
    public boolean equals(Object o){
        if ( o instanceof List<?>){
            Iterator<E> it1 = iterator();
            Iterator<?> it2 = ((List<?>)o ).iterator();
            while (it1.hasNext() && it2.hasNext()){
                E e1 = it1.next();
                Object e2 = it2.next();
                if (! ( e1 == null ? e2== null: e1.equals(e2)  )){
                    return false;
                }
            }
            return !it1.hasNext() && !it2.hasNext();
        } else return false;
    }
}
```



PRATO LINUX USER GROUP

# Analisi

```
public abstract class MiaListaOK<E>
```

```
    extends AbstractCollection<E>
```

```
    implements List<E>{
```

```
    public boolean equals(Object o){
```

```
        if (o instanceof List<?>){
```

```
            Iterator<E> it1 = iterator();
```

```
            Iterator<?> it2 = (
```

```
                (List<?>)o ).iterator();
```

```
            while (.....
```

```
                Object e2 = it2.next();
```

OK List<?> è reificabile

List<?> è reificabile e non vengono generati warning

Da una List<?> posso prelevare Object senza problemi



PRATO LINUX USER GROUP

# Cast non reificabili

Test di istanza verso tipi *non reificabili* generano sempre errore

In alcune circostanze un cast ad un tipo *non reificabile* può non generare warning



PRATO LINUX USER GROUP

# Nessun warning

```
public static <T> List<T>
asList(Collection<T> c )
    throws IllegalArgumentException { List<?> è reificabile → compila
if ( c instanceof List<?>){
    return (List<T>)c;
} else throw new
    IllegalArgumentException("Il
parametro passato non e' un sottotipo
di List<T>");
```

Il cast non genera warning in quanto la sorgente del cast ha tipo `Collection<T>` e ogni oggetto di questo tipo che implementa `List` deve avere come tipo `List<T>`



PRATO LINUX USER GROUP



# Unchecked cast

Non sempre il compilatore è in grado di capire se un cast verso un tipo non reificabile avrà successo

I sistemi di tipi (dei linguaggi di programmazione orientati agli oggetti) non sono perfetti e non possono individuare situazioni di successo come può fare un (buon) programmatore

Per questo motivo un cast verso un tipo non reificabile non genera un errore ma un warning



PRATO LINUX USER GROUP

```

public static List<String>
converti(List<?> oggetti){
for (Object obj: oggetti){
if (!(obj instanceof String)){
throw new
IllegalArgumentException( ...
} }
return
(List<String>)(List<?>)oggetti;
}

```

Il cast “a logica” non fallirà mai

OK String è *reificabile*

Unchecked Cast ← il compilatore non è in grado di capire se il cast avrà successo o meno

Nota: è illegale “castare” una lista di oggetti ad una lista di stringhe, quindi serve il doppio cast



PRATO LINUX USER GROUP

# Array e reificazione

Gli array *reificano* i loro componenti ovvero conservano a *runtime* i tipi dei loro componenti



PRATO LINUX USER GROUP

# Un vecchio esempio

```
Integer[] interi = new  
    Integer[] {1,2,3};  
Number[] numeri = interi;  
numeri[2] = 3.14;
```

OK, gli array sono *covarianti*

ArrayStoreException

Double non è compatibile con  
il tipo *reificato* dell'array

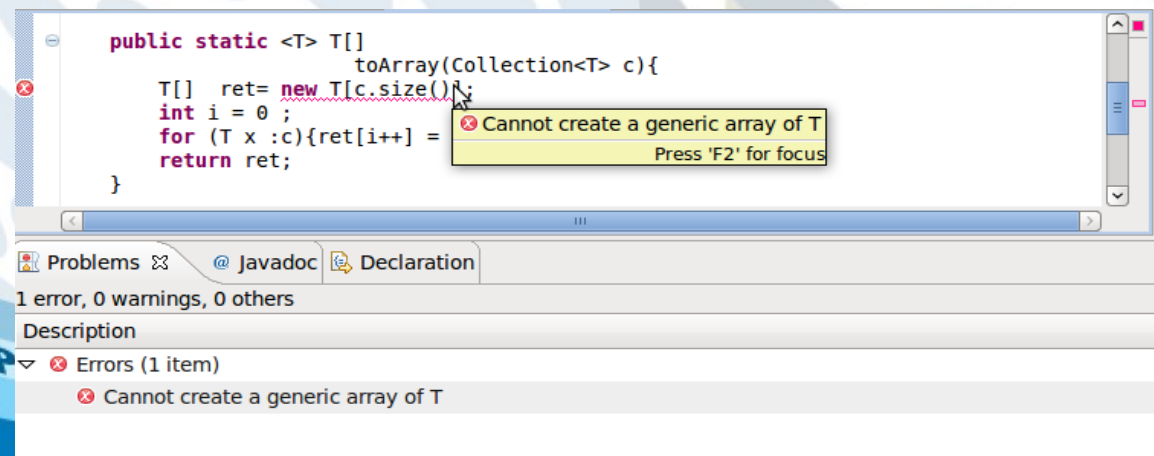


# Creazione di array

```
public static <T> T[]  
    toArray(Collection<T> c){  
    T[] ret= new T[c.size()];  
    int i = 0 ;  
    for (T x :c){ret[i++] = x;}  
    return ret;  
}
```

NO! errore di compilazione

Le variabili di tipo non sono reificabili → viene generato un *generic array creation error*



```
public static <T> T[]  
    toArray(Collection<T> c){  
    T[] ret= new T[c.size()];  
    int i = 0 ;  
    for (T x :c){ret[i++] = x;}  
    return ret;  
}
```

Cannot create a generic array of T  
Press 'F2' for focus

Problems @ Javadoc Declaration  
1 error, 0 warnings, 0 others  
Description  
Errors (1 item)  
Cannot create a generic array of T



PRATO LINUX USER GROUP

# Difetti ...

Non poter creare array di generics è una limitazione di java  
Array generici sono problematici a causa della *erasure*  
(tuttavia) *erasure* facilita l'evoluzione

In generale ... dire “no!” ad array ed utilizzare il Collection Framework



PRATO LINUX USER GROUP

# Reflection ...

Tramite reflection è possibile ovviare al problema della creazione di array generici



PRATO LINUX USER GROUP

# Don't try this at home

```
public static <T>
```

```
    T[] toArray(Collection<T> coll){
```

```
        T[] ret = (T[]) new Object[coll.size()];
```

Unchecked Cast!

```
        int i = 0 ;
```

```
        for (T x :coll){ret[i++] = x;}
```

```
        return ret;
```

```
    }
```

```
---
```

```
List<String> stringhe = ...
```

```
String[] a = toArray(stringhe);
```

Class Cast Exception

Exception in thread "main"

java.lang.ClassCastException:

[Ljava.lang.Object; cannot be cast to  
[Ljava.lang.String;

at generics.cap6.Ex1.main(Ex1.java:23)



PRATO LINUX USER GROUP



# Messaggio oscuro

```
Exception in thread "main"  
java.lang.ClassCastException:  
[Ljava.lang.Object; cannot be cast to  
[Ljava.lang.String;  
at generics.cap6.Ex1.main(Ex1.java:23)
```

[L significa *array* di *reference type*

[LObject significa array di Object, Object è il *component type* dell'array

L'errore non viene segnalato nel punto in cui viene generato (!) ma in un altro punto (!!!) ovvero nel main



PRATO LINUX USER GROUP

# Versione generata dall'erasure: analisi

```
public static Object[] toArray1(
```

```
    Collection coll){
```

```
    Object[] ret = (Object[])
```

```
        new Object[coll.size()];
```

```
    int i = 0 ;
```

```
    for (Object x :coll){ret[i++] = x;} 
```

```
    return ret;
```

```
}
```

```
---
```

```
List<String> stringhe =
```

```
    Arrays.asList("ciao", "mondo");
```

```
String[] a = (String[])toArray(stringhe);
```

Unchecked Cast sparito!

ClassCastException



PRATO LINUX USER GROUP

# Versione generata dall'erasure: analisi

```
public static Object[] toArray1(
    Collection coll){
    Object[] ret = (Object[])
        new Object[coll.size()];
    int i = 0 ;
    for (Object x :coll){ret[i++] = x;}
    return ret;
}
```

---

```
List<String> stringhe =
    Arrays.asList("ciao", "mondo");
String[] a = (String[])toArray(stringhe);
```

Erasure: converte il cast a T[] in un  
Cast a Object[]

Erasure: inserisce il cast a String[]



# Attenzione!

Nonostante l'array contenga solamente stringhe, il suo tipo reificato è un array di Object!



PRATO LINUX USER GROUP

# Cast iron guarantee (ancora)

I cast inseriti dall'*erasure* non falliscono mai a parte quando viene generato un *unchecked cast warning*

Quando viene generato un *unchecked cast warning* allora i cast inseriti dalla *erasure*

- Possono fallire
- Possono essere segnalati in parti del codice differenti da quella che ha generato l'errore



PRATO LINUX USER GROUP

# Adoro i soldi che generano i soldi

A volte un modo per fare soldi è tramite altri soldi

Lo stesso si può applicare agli array: generare array tramite un altro array



PRATO LINUX USER GROUP

# Array genera array

```
public static <T> T[] toArray(  
    Collection<T> coll, T[] arr){  
    T[] ret = null;  
    ret =  
        (T[])java.lang.reflect.Array.newInstance  
            (arr.getClass().getComponentType(),      unchecked cast  
            coll.size());  
    int i = 0;  
    for (T elem: coll) ret[i++] = elem;  
    return ret;  
}
```

PRATO LINUX USER GROUP



# Esecuzione del codice

```
List<String> a=  
    Arrays.asList("ciao","a");  
String[] st =  
    toArray(a, new String[0]);  
for (String s: st)  
    System.out.println(s);
```

Ok! Funziona!

Il metodo `getComponentType()` restituisce un "Class Object" che rappresente il tipo dei componenti dell'Array T

```
Class<?> ctype =
```

```
arr.getClass().getComponentType();
```

Nel nostro caso è String





# La versione “di classe”

```
public static <T> T[] toArray(  
Collection<T> coll, Class<T> classe){  
T[] ret = null;  
ret =  
(T[])java.lang.reflect.Array.newInstance  
(  
Classe, coll.size() );  
int i = 0;  
for (T elem: coll) ret[i++] = elem;  
return ret;  
}
```

```
List<String> a=  
Arrays.asList("ciao","a");  
String[] st2 =  
toArray(a, String.class);
```



PRATO LINUX USER GROUP

**Fine**

**DOMANDE?**



**PRATO LINUX USER GROUP**

# Licenza Creative Commons 2.5 BY NC SA

## Tu sei libero:

- di riprodurre, distribuire, comunicare al pubblico, esporre in pubblico, rappresentare, eseguire e recitare quest'opera
- di modificare quest'opera

## Alle seguenti condizioni:

- **Attribuzione.** Devi attribuire la paternità dell'opera nei modi indicati dall'autore o da chi ti ha dato l'opera in licenza e in modo tale da non suggerire che essi avallino te o il modo in cui tu usi l'opera.
- **Non commerciale.** Non puoi usare quest'opera per fini commerciali.
- **Condividi allo stesso modo.** Se alteri o trasformi quest'opera, o se la usi per crearne un'altra, puoi distribuire l'opera risultante solo con una licenza identica o equivalente a questa.

<http://creativecommons.org/licenses/by-nc-sa/2.5/deed.it>



# Appendice

Note prese durante lo svolgimento della  
sessione J&G



PRATO LINUX USER GROUP

# Subtyping e controvarianza

- List estende Collection
  - List<E> estende Collection<E>
  - Se B estende A allora
    - List<B> **estende** Collection<B>
    - List<B> **non estende** List<A>
    - ... ripeto ...
- x List<B> **non estende** List<A>



PRATO LINUX USER GROUP

# Wildcard Capture 1/

```
Map<String,?> map;
```

```
Iterator<Entry<String,?>> it = map.entrySet().iterator();
```

----

MyClass.java:9: incompatible types

found : java.util.Iterator<java.util.Map.Entry<java.lang.String,capture of ?>>

required: java.util.Iterator<java.util.Map.Entry<java.lang.String,?>>

```
Iterator<Entry<String,?>> it = map.entrySet().iterator();
```



PRATO LINUX USER GROUP

# Wildcard Capture 2/

`Iterator<Entry<String,capture of ?>>` non è compatibile con `Iterator<Entry<String,?>>`

"capture of ?" non è un sottotipo di "?"?

SI! Ma sappiamo che in questo caso abbiamo bisogno che gli argomenti siano esattamente dello stesso tipo.

Sappiamo già che `List<String>` non è un sottotipo di `List<Object>` ad esempio (generics non sono covarianti)



PRATO LINUX USER GROUP

# Wildcard Capture 3/

"capture of ?" significa che un *wildcard* deve essere convertito (prima o poi) in una variabile di tipo usando la *capture conversion*.

In "Map<String,?>" ( Map da String ad un tipo ignoto) il secondo parametro è ignoto, ma, ad un certo punto il compilatore dovrà ragionare sul tipo specifico del parametro (java è un linguaggio fortemente tipato\*)

Il *wildcard* non può rimanere *wild* (o meglio, ignoto) per sempre, così il il compilatore "catturerà" una snapshot di "?" in una variabile di tipo anonimo ad un certo punto del programma.

La differenza tra "?" e "capture of ?" è che il primo si riferisce a "tutti i possibili tipi", mentre il secondo si riferisce ad un particolare tipi in un particolare punto del programma.

In parole povere, prima o poi al posto del wildcard ci andrà messo qualcosa di concreto

Esempi presi da [http://blogs.sun.com/ahe/entry/why\\_is\\_the\\_capture\\_of](http://blogs.sun.com/ahe/entry/why_is_the_capture_of)





# Capture

Durante l'intervento ho utilizzato il termine “catturare” (corretto) ... poi ho notato che Bruce Eckel (o meglio il traduttore di Thinking in Java) utilizza il termine “intercettare”



PRATO LINUX USER GROUP

# Capture Conversion 1/

Ricordiamoci che prima o poi il *wildcard* dovrà sparire:

- Istanziamento di oggetti
- Passaggio di parametri

Sarà quindi necessaria prima o poi una “conversione” di tipo

Alcune conversioni di tipo

- Subtyping (già visto)
- Numeric promotion (già visto)
- Widening/Narrowing (esempio da interi a decimali)
- ... capture conversione ...



PRATO LINUX USER GROUP

# Capture Conversion 2/

```
public void rebox(Box<?> box) {  
    reboxHelper(box);  
}
```

```
private<V> void reboxHelper(Box<V> box) {  
    box.put(box.get());  
}
```



PRATO LINUX USER GROUP

# Capture Conversion 3/

- La chiamata da `rebox()` a `reboxHelper()` è sicura, ma **non** è giustificabile tramite una relazione di subtyping tra `Box<?>` e `Box<V>`.
- La chiamata è sicura perché l'argomento in ingresso è sicuramente un "Box" di qualche tipo concreto (seppure ignoto). Se possiamo *catturare* questo tipo ignoto in una variabile di tipo `X` allora possiamo dedurre che `V` sia `X`.
- Questa è l'essenza della capture conversion!
- ... tratto da Java Language Specifications ...



PRATO LINUX USER GROUP