

Java: lambda expressions



Classe interna anonima

Sottinteso: AnonymousClass implements

- `c.setOnMouseEntered(new EventHandler<MouseEvent>() {`
- `public void handle(MouseEvent event) {`
- `System.out.print("Entered");`
- `c.setFill(Color.RED);`
- `});`
-



Sostituzione con una lambda expression

- `c.setOnMouseEntered(new EventHandler<MouseEvent>() {`
- `public void handle(MouseEvent event) {`
- `System.out.print("Entered");`
- `c.setFill(Color.RED);`
- `});`

- `c.setOnMouseEntered((MouseEvent event) -> {`
- `System.out.print("Entered");`
- `c.setFill(Color.RED);`
- `});`



Inferring the functional interfaces

- Does the interface have only one abstract (unimplemented) method?
- Does the parameters (types) of the lambda expression match the parameters (types) of the single method?
- Does the return type of the lambda expression match the return type of the single method?



```
c.setOnMouseEntered((MouseEvent event) -> {  
    System.out.print("Entered");  
    c.setFill(Color.RED);  
});
```

- D: Cosa si aspetta il metodo `setOnMouseEntered`?
- R: Un `EventHandler<MouseEvent>`
- D: L'Interfaccia `EventHandler<MouseEvent>` ha un solo metodo? Quale e con che firma?
- R: si, `handle(MouseEvent event)`
- D: che valore di ritorno restituisce il metodo `handle`?
- R: `void`
- D: La lambda expression è coerente con le attese?
- R: si.



Supporto da Netbeans

49
50
51
52
53
54
55
56

This anonymous inner class creation can be turned into a lambda expression.

(Alt-Enter shows hints)

```
c.setOnMouseEntered(new EventHandler<MouseEvent>() {  
    public void handle(MouseEvent event) {  
        System.out.print("Entered");  
        c.setFill(Color.RED);  
    }  
});
```

45
46
47
48
49
50

Anonymous class can be used

(Alt-Enter shows hints)

```
c.setX(200);  
c.setY(200);  
c.setOnMouseEntered((MouseEvent event) -> {  
    System.out.print("Entered");  
    c.setFill(Color.RED);  
});
```



Generics





Cast a volontà...

```
public class MazzoBase {  
    protected List carte = new LinkedList();  
    public Carta pescaCarta() {  
        return (Carta) carte.remove(0);  
    }  
}
```

Strutture dati polimorfe
richiedono conversioni di tipo
al momento dell'estrazione

```
...  
carte.add("C,1");  
...
```

Ciò può generare
errori di tipo a runtime!

Generics

Consentono di specificare un
tipo come parametro ...

```
public class MazzoBase {  
    protected List<Carta> carte =  
        new LinkedList<Carta>();  
    public Carta pescaCarta() {  
        return (Carta) carte.remove(0);  
    }  
}
```

... evitando la necessità di cast espliciti ...

```
...  
carte.add("C,1");  
...
```

... e permettendo di identificare
gli errori *a compilation time*

```
error: no suitable method found for add(String)  
  carte.add("C,1");  
  method Collection.add(Carta) is not applicable  
    (argument mismatch; String cannot be converted to Carta)  
  method List.add(Carta) is not applicable  
    (argument mismatch; String cannot be converted to Carta)
```



Generics in Java

- Presenti a partire da Java 5
- Analoghi concetti in altri linguaggi
 - Es., template in C++
- Consentono la definizione di:
 - ***tipo generico***: una classe o interfaccia la cui definizione include uno o più tipi come parametro
 - ***metodo generico***: include la dichiarazione di uno o più tipi usati come parametro



Tipi generici

- **class name<T1, T2, ..., Tn>**
dove **T1...Tn** sono i tipi con cui la classe è parametrizzata
 - Analogo per le interfacce
- Il tipo «attuale» deve essere specificato all'atto della dichiarazione

```
class Group<T> { ...           //definizione  
Group<Student> gs = ... //uso  
Group<Tourist> gt = ... //uso
```





Esempio

```
class Pair<X,Y> {  
    private X first;  
    private Y second;  
    public Pair(X a1, Y a2) {  
        first = a1;  
        second = a2;  
    }  
    public X getFirst() { return first; }  
    public Y getSecond() { return second; }  
    public void setFirst(X arg) { first = arg; }  
    public void setSecond(Y arg) { second = arg; }  
}
```

tipo generico

I «parametri tipo» sono visibili
nell'intera definizione
della classe

... dove sono usati come un
qualsiasi altro tipo
(a parte alcuni casi particolari)

«argomenti tipo»: rimpiazzano i
parametri all'atto della dichiarazione

«diamond operator» (Java 7): gli
argomenti possono essere omessi e
inferiti dal compilatore

```
Pair<String,Long> = new Pair<String,Long>("PI", 3.14L);  
12Pair<String,Long> = new Pair<>("PI", 3.14L);
```

Esempio

Set può contenere un qualsiasi Object:
nessuna garanzia sul fatto che i due insiemi
contengano oggetti dello stesso tipo

```
public static Set union(Set s1, Set s2) {  
    Set result = new HashSet(s1);  
    result.addAll(s2);  
    return result;  
}
```

dichiarazione dell'argomento tipo, locale
al metodo

```
public static <E> Set<E> union(Set<E> s1,  
                               Set<E> s2) {  
    Set<E> result = new HashSet<>(s1);  
    result.addAll(s2);  
    return result;  
}
```

metodo generico



Collections e generics

- Il Java Collection Framework fa ampio uso dei generics:
 - tutte le classi, interfacce e metodi che abbiamo visto finora sono in realtà generici

```
Collection<E>, Set<E>  
List<E>  
Map<K, V>  
Comparable<T>  
Comparator<T>
```

```
add(E e)  
add(int index, E e)  
put(K key, V value)  
int compareTo(T o)  
int compare(T o1, T o2)
```

Usate sempre le versioni generiche!



Generics e sottotipi

- Dati due tipi generici **G<A>** e **G** dove **B** è una sottoclasse di **A**, **non** è vero che **G** è una sottoclasse di **G<A>**
- In altre parole: fra tali tipi generici non vale il principio di sostituzione

- Esempio:

`Person s = new Student();` //ok

`Group<Person> g = new Group<Student>();` //no!

- **G<A>** e **G** sono sottoclassi di **Object**
 - In realtà sono la **stessa** sottoclasse **G**



... perché?

- I generics in Java sono implementati mediante ***type erasure***
 - L'informazione sui parametri tipo viene eliminata dopo i controlli statici
 - Vengono inseriti gli opportuni cast per mantenere i vincoli sul tipo
- Questa scelta mantiene compatibilità con API che non usano generics ...
 - Es. Java Collection framework pre-Java 5
- ... ma genera una serie di limitazioni



Esempio

```
public class Pair {
    private Object first;
    private Object second;
    public Pair(Object a1,
                Object a2) {
        first = a1;
        second = a2;
    }
    public Object getFirst() { return first; }
    public Object getSecond() { return second; }
    public void setFirst(Object arg) { first = arg; }
    public void setSecond(Object arg) { second = arg; }
}

final class Test {
    public static void main(String[] args) {
        Pair pair = new Pair("PI", 3.14L);
        String s = (String) pair.getFirst();
        Long l = (Long) pair.getSecond();
        Object o = pair.getSecond();
    }
}
```

```
class Pair<X,Y> {
    private X first;
    private Y second;
    public Pair(X a1, Y a2) {
        first = a1;
        second = a2;
    }
    public X getFirst() { return first; }
    public Y getSecond() { return second; }
    public void setFirst(X arg) {first = arg;}
    public void setSecond(Y arg) {second = arg;}
}

class Test {
    public static void main(String[] args) {
        Pair<String,Long> pair =
            new Pair<>("PI", 3.14L);
        String s = pair.getFirst();
        Long l = pair.getSecond();
        Object o = pair.getSecond();
    }
}
```



Generics e sottotipi: *wildcard*

- È possibile specificare una o più *wildcard*
 - Rappresentano un tipo non noto

```
Group<?> g = new Group<Student>() ;
```

- Non possono essere usate per creare oggetti

```
Group<?> g = new Group<?>() ; // no!
```

- Utile ad esempio per realizzare metodi generici...



Esem

L'intento è quello di stampare una lista contenente qualsiasi tipo di oggetto

```
static void printList(List<Object> list) {  
    for (Object elem : list)  
        System.out.println(elem + " ");  
}
```

In realtà consente di stampare solo liste di Object

```
List<Person> lp = new LinkedList<>();  
... // popola la lista  
printList(lp); // errore in compilazione!
```

La wildcard risolve il problema consentendo subtyping

```
static void printList(List<?> list) {
```



Generics e sottotipi: *bounded wildcards*

- È possibile «limitare» le wildcard specificando che sono accettati solo sottotipi di un tipo dato

```
Group<? extends Persona> g  
    = new Group<Student>();
```

- Si usa **extends** anche con le interfacce

```
Group<? extends Comparable> g  
    = new Group<Student>();
```

- È possibile specificare più di un tipo:
se c'è una classe deve essere la prima

```
Group<? extends Persona & Comparable> g  
    = new Group<Student>();
```



Esempio

Somma il contenuto di una lista i cui elementi sono vincolati a essere numeri

```
static double aggregate(List<? extends Number> list) {
    double s = 0.0;
    for (Number n : list)
        s += n.doubleValue();
    return s;
}
```

Può invocare metodi di Number!

```
List<Integer> li = Arrays.asList(1, 2, 3);
System.out.println(aggregate(li)); // 6.0
List<Double> ld = Arrays.asList(1.2, 2.3, 3.5);
System.out.println(aggregate(ld)); // 7.0
List<String> ls = Arrays.asList("1", "2", "3");
System.out.println(aggregate(ls)); // compilation error
```

- Le wildcard sono usate nel Collection Framework!
es. `addAll(Collection<? extends E> c)`



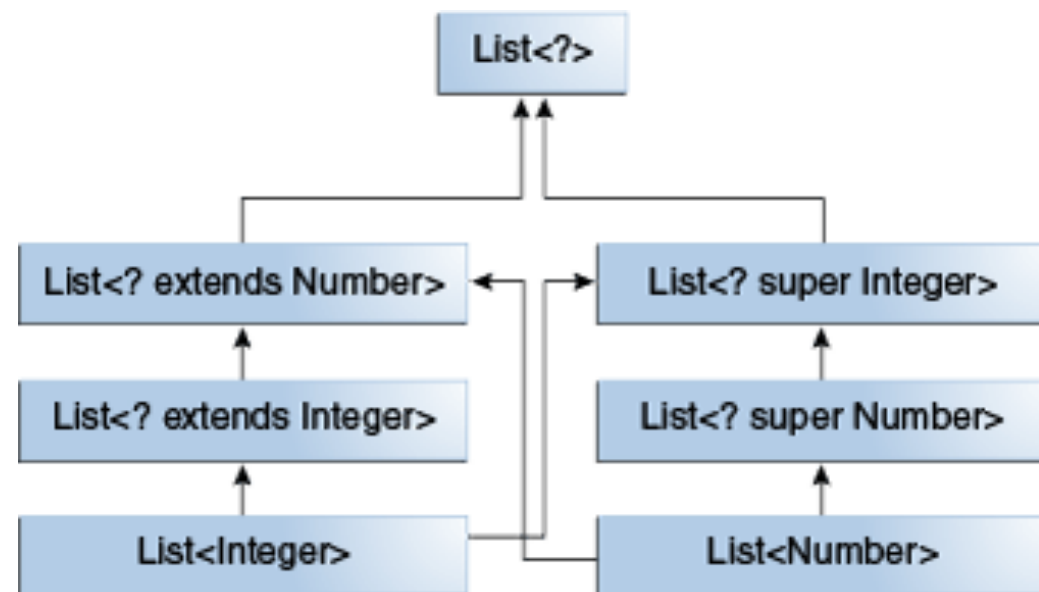
Generics e sottotipi: *bounded wildcards*



- È possibile «limitare» le wildcard anche verso le superclassi

```
Group<? super Studente> g  
    = new Group<Persona>();
```

- Le regole di tipo diventano complicate...



Generics e array

- Non si può creare un array generico:
 - Es.: tutte queste espressioni sono illegali:
new List<E>[]
new List<String>[]
new E[]
- Motivo: non sarebbe type safe
 - minerebbe il vantaggio principale dei generics
- Se necessario, è possibile usare un cast
 - Ma non è desiderabile, per lo stesso motivo
- In generale, meglio usare **List**
 - Più flessibili e generiche





Esempio

```
public class Chooser {  
    private final Object[] choiceArray;  
    public Chooser(Collection choices) {  
        choiceArray = choices.toArray();  
    }  
    public Object choose() {  
        return choiceArray[new Random().nextInt(choiceArray.length)];  
    }  
}
```

Diagram annotations:

- `T[]` points to the `choiceArray` field.
- `Collection<T>` points to the `choices` parameter.
- `(T[])` (circled in green) points to the `toArray()` method.
- `T` points to the `choose()` return type.

Chooser.java:4: **warning**: [unchecked] unchecked cast
choiceArray = (T[]) choices.toArray();

```
public class Chooser<T> {  
    private final List<T> choiceList;  
    public Chooser(Collection<T> choices) {  
        choiceList = new ArrayList<>(choices);  
    }  
    public T choose() {  
        return choiceList.get(new Random().nextInt(choiceList.size()));  
    }  
}
```


Alcune limitazioni dei generics

- I parametri tipo non possono essere tipi primitivi
 - Es. **ArrayList<int>**; ma con wrapper e autoboxing solitamente non è un problema

- Generics non si possono usare con **instanceof**

```
Object o = new LinkedList<Long>();  
obj instanceof List  
obj instanceof List<?>  
obj instanceof List<Long>           //error  
obj instanceof List<? extends Number> //error  
obj instanceof List<? super Number>  //error
```

- Svariati altri casi particolari ...
- Per chi vuole approfondire: <http://www.angelikalanger.com/GenericsFAQ/JavaGenericsFAQ.html>

