Java Generics

Introduzione

Con le **Java Generics** è possibile implementare il concetto di tipo parametrizzato, che permette di creare componenti (spesso dei contenitori) che sono semplici da utilizzare con più tipi.

Containers e Java Generics...

Una delle motivazioni principali delle Java Generics è rivolta alla possibilità di creare classi contenitori.

Consideriamo una semplice classe che contiene un solo oggetto di classe **Object**:

Containers e Java Generics...

```
class Automobile{}
public class Holder2 {
  private Object a;
  public Holder2(Object a) { this.a = a; }
  public void set(Object a) { this.a = a; }
  public Object get() { return a; }


public static void main(String[] args) {

Holder2 h2 =new Holder2(new Automobile());
Automobile a = (Automobile)h2.get();
h2.set("Not an Automobile");
String s = (String)h2.get();
h2.set(1); // Autoboxes to Integer
Integer x = (Integer)h2.get();
  }
}
```

Containers e Java Generics...

La classe Holder2 è in grado di contenere oggetti di qualsiasi classe.

L'istruzione `h2.set(1)` converte automaticamente un **int** in **Integer** (*autoboxing*)

L'istruzione `for` permette di iterare su tutti gli oggetti del contenitore.

...Containers e Java Generics.

Ci sono alcuni casi nei quali è necessario inserire all'interno di un contenitore più oggetti omogenei. In questi casi è possibile ricorrere alle Java Generics che:

- Permettono di specificare il tipo degli oggetti contenuti
- Demandano al compilatore la possibilità di effettuare controlli di tipo al “compile time”.

Nell'esempio precedente, è possibile ricorrere alle generics:

...Containers e Java Generics.

```
public class Holder3<T> {
  private T a;

  public Holder3(T a) { this.a = a; }

  public void set(T a) { this.a = a; }

  public T get() { return a; }

  public static void main(String[] args) {
        Holder3<Automobile> h3 =      new Holder3<Automobile>(new Automobile());
        Automobile a = h3.get(); // No cast needed
         //h3.set("Not an Automobile"); // Error
          // h3.set(1); // Error
          Holder3<String> h4 = new Holder3<String>("Not an Automobile");
  }
} ///:~
```

Containers e Java Generics: definizione di Tuple...

In alcuni casi è possibile che sia necessario definire una funzione che restituisca non un singolo valore ma una coppia di valori o una tripla, ecc...

```
public class TwoTuple<A,B> {
  public final A first;
  public final B second;
  public TwoTuple(A a, B b) { first = a; second = b; }
  public String toString() {
    return "(" + first + ", " + second + ")";
  }
}
```

...Containers e Java Generics: definizione di Tuple.

```
class Amphibian {}
class Vehicle {}

public class TupleTest {
  static TwoTuple<String,Integer> f() {
    // Autoboxing converts the int to Integer:
    return new TwoTuple<String,Integer>("hi", 47);
  }

  public static void main(String[] args) {
    TwoTuple<String,Integer> ttsi = f();
 }
}
```

Containers e Java Generics

- Tutte le classi contenitore sono definite come classi generiche (ArrayList<T>, LinkedList<T>,…)

```
//Cat.java
public class Cat {
private int catNumber;
Cat(int i) { catNumber = i; }
void print() {
System.out.println("Cat #" + catNumber);
}
}

//Dog.java
public class Dog {
private int dogNumber;
Dog(int i) { dogNumber = i; }
void print() {
System.out.println("Dog #" + dogNumber);
}
}
```

```
//CatsAndDogs.java

import java.util.*;
public class CatsAndDogs {
public static void main(String[] args) {
ArrayList<Cat> cats = new ArrayList();
for(int i = 0; i < 7; i++)
   cats.add(new Cat(i));
Iterator<Cat> it=cats.iterator();
while(it.hasNext())
    it.next().print();

// cats.add(new Dog()) // errore di
compilazione
}
```

Java Generics e Interfacce

- Le Java Generics possono essere anche utilizzate per parametrizzare la dichiarazione di interfacce:
- Esempi di interfaccie generiche

Iterator<T>, Comparable<T>, Comparator<T>, ….

for each

- Per tutte le classi che implementano l'interfaccia Collection (per le quali è definita l'operazione iterator()) si può usare la struttura di controllo for-each

for (<variabile>:<collection>){…}

```
Iterator<Cat> it=cats.iterator();
while(it.hasNext())
   it.next().print();
```

equivale a

```
for(Cat o: cats)
    o.print();
```

Metodi Generici...

Oltre a parametrizzare la dichiarazione di intere classi, è possibile parametrizzare la dichiarazione di metodi all'interno di una classe.

Un metodo può essere definito generico indipendentemente dalla fatto che la classe sia generica oppure no.

Per di più, se un metodo definito in una classe parametrizzata è statico, tale metodo non accederà al parametro di tipo della classe.

...Metodi Generici...

Per definire un metodo come generico è sufficiente parametrizzare la sua dichiarazione:

```
public class GenericMethods {
  public <T> void f(T x) {
    System.out.println(x.getClass().getName());
  }
  public static void main(String[] args) {
    GenericMethods gm = new GenericMethods();
    gm.f("");
    gm.f(1);
    gm.f(1.0);
    gm.f(1.0F);
    gm.f('c');
    gm.f(gm);
  }
}
```

```
/* Output:
java.lang.String
java.lang.Integer
java.lang.Double
java.lang.Float
java.lang.Character
GenericMethods
*/
```

...Metodi Generici...

```
public class New {
  public static <T> LinkedList<T> lList() {
    return new LinkedList<T>();
}

  public static void main(String[] args) {
        LinkedList<String> lls = New.lList();
        lls.<String>add("map"); // inutile
        lls.add("corso A");

  }
}
```

...Metodi Generici...

- In Java SE 7 List<Object> non può essere convertito in List<String>
- Da Java SE 8 l'inferenza dei tipi è estesa all'argomento dei metodo

```
import java.util.*;
public class New {
  public static <T> LinkedList<T> lList() {
    return new LinkedList<T>();
}
  public static void processList(LinkedList<String> lList) {
  for (String obj:lList)
  System.out.println(obj);
  } public static void main(String[] args) {
       processList(New.<String>lList());  // obligatorio in Java 7
       processList(New.lList());  // solo da Java 8 in poi
  }
}
```

...Metodi Generici...

```
import java.util.*;
public class GenericVarargs {
  public static <T> List<T> makeList(T... args) {
    List<T> result = new ArrayList<T>();
    for(T item : args)
      result.add(item);
    return result;
  }
  public static void main(String[] args) {
    List<String> ls = makeList("A");
    System.out.println(ls);
    ls = makeList("A", "B", "C");
    System.out.println(ls);
    ls = makeList("ABCDEFFHIJKLMNOPQRSTUVWXYZ".split(""));
    System.out.println(ls);
  }
}
```

```
/* Output:
[A]
[A, B, C]
[, A, B, C, D, E, F, F, H, I, J, K, L,
M, N, O, P, Q, R, S, T, U, V, W, X,
Y, Z]
*/
```

Il problema dell'erasure

Le Java Generics lasciano comunque alcune questioni poco chiare. Per esempio, mentre è possibile ricorrere al letterale di classe per la classe ArrayList:

```
ArrayList.class
```

non è possibile ricorrere al letterale di classe per la classe ArrayList ottenuta parametrizzando il tipo del contenuto:

```
// ArrayList<Integer>.class
```

Per comprendere meglio questo aspetto, consideriamo il seguente esempio:

Il problema dell'erasure

```
import java.util.*;

public class ErasedTypeEquivalence {
  public static void main(String[] args) {
    Class c1 = new ArrayList<String>().getClass();
    Class c2 = new ArrayList<Integer>().getClass();
    System.out.println(c1 == c2);
  }
}

/* Output:
true
*/
```

Il problema dell'erasure

Il risultato è true, benché ArrayList<String> ed ArrayList<Integer> sono di tipo diverso (ad esempio, non si può memorizzare una istanza di Integer in ArrayList<String>)

Com'è possibile?

La ragione sta nel seguente fatto:

Non è disponibile alcuna informazione sui tipi di parametri generici all'interno del codice generico.

Le generics del Java sono implementate usando l'*erasure* (cancellazione). Questo significa che ogni informazione di tipo è cancellata quando si ricorre all'astrazione generica.

Così ArrayList<String> e ArrayList<Integer> sono, di fatto, lo stesso tipo (ArrayList) al run time.

```
import java.util.*;
class Frob {}
class Fnorkle {}
class Quark<Q> {}
class Particle<POSITION,MOMENTUM> {}

public class LostInformation {
  public static void main(String[] args) {
    List<Frob> list = new ArrayList<Frob>();
    Map<Frob,Fnorkle> map = new
           HashMap<Frob,Fnorkle>();
    Quark<Fnorkle> quark = new Quark<Fnorkle>();
    Particle<Long,Double> p = new
           Particle<Long,Double>();
    System.out.println(Arrays.toString(
      list.getClass().getTypeParameters()));
    System.out.println(Arrays.toString(
      map.getClass().getTypeParameters()));
    System.out.println(Arrays.toString(
      quark.getClass().getTypeParameters()));
    System.out.println(Arrays.toString(
      p.getClass().getTypeParameters()));
  }
}
```

```
/* Output:
[E]
[K, V]
[Q]
[POSITION, MOMENTUM]
*/
```

Il problema dell'erasure

Nell’esempio precedente, è usato il metodo di istanza *Class.getTypeParameters()* che, secondo i javadoc, ”returns an array of TypeVariable objects that represent the type variables declared by the type specification...”

Tuttavia, tale metodo non restituisce i tipi, ma i nomi dei parametri.