



Tiger 5, evoluzione in atto

Giovanni Marigi
KTECH

www.gmarigi.it

g.marigi@k-tech.it





Step del talk

- Generics
- Autoboxing Unboxing
- Varargs
- Static Import
- Enum
- Annotations
- Format/Printf
- Riferimenti e conclusioni



Il talk si rivolge a tutti gli sviluppatori Java neofiti o meno del linguaggio.

Il talk ha l'obiettivo di presentare le principali novità introdotte nell'ultima versione della JDK, 1.5, comunemente chiamata Tiger



Generics

I generics sono stati introdotti in Java con il duplice obiettivo:

- permettere la creazione di oggetti e/o metodi con un controllo esplicito sui parametri (type-safe objects)
- permettere la creazione di oggetti e/o metodi altamente parametrizzati

I generics sono realizzati con un meccanismo di espansione/sostituzione detto **Erasure and Translation**

Dopo gli opportuni controlli di consistenza tra tipi, il compilatore cancella (Erasure) tutte le informazioni relative ai tipi parametrici, e sostituisce (Translation) le relative occorrenze con il tipo di oggetto specifico



Generics *oggetti type-safe*

Java è un linguaggio fortemente tipizzato:

✓ tutte le classi in modo implicito estendono la classe **Object**.

Moltissime classi nelle librerie accettano come parametro di ingresso ai loro metodi degli Object, e/o restituiscono come valore di ritorno un Object:

Le classi e le interfacce nelle API usate come collezioni o mappe “**Java Collection Framework**” presentano moltissimi metodi con queste caratteristiche



Generics *oggetti type-safe*

Nell'uso delle collezioni il programmatore deve gestire situazioni di **ambiguità** e deve sapere con certezza il tipo specifico di oggetto che sta manipolando.

Il compilatore garantisce che il metodo get dell'interfaccia List sia un Object ma la gestione del corretto uso dell'oggetto e dell'esplicita conversione dello stesso in un Object type-safe è a carico del programmatore attraverso l'uso di un'operazione di **casting**.

Problemi connessi con le operazioni casting:

- ✓ in ogni situazione possiamo sapere con certezza il tipo su cui operiamo l'operazione di casting?
- ✓ il nostro codice risulta essere portabile?
- ✓ introduciamo overhead e controlli a run-time nella JVM?



Generics *oggetti type-safe*

Le operazioni di casting non sono più necessarie se utilizziamo la sintassi dei generics, limitando ad un **tipo di oggetto specifico** il parametro di ingresso o il valore di ritorno di un metodo "ambiguo".

```
List<String>stringhe = new ArrayList<String>();  
  
stringhe.add("Pippo");  
stringhe.add("Pluto");  
  
String x = stringhe.iterator().next();
```

- ✓ non c'è più ambiguità nelle operazioni sulla collezione!
`stringhe.add(new Integer(5))` darà errore in fase di compilazione.
- ✓ in fase di estrazione non è stata effettuata alcuna operazione di casting: la collezione è di fatto esplicitamente parametrizzata a tempo di istanziazione



Generics *oggetti type-safe*

Osservando nelle API la definizione della interfaccia List:

```
public interface List<E> extends Collection<E>
{
    boolean add(E o)
    E get(int index)
}
```

Al posto di Object abbiamo un “oggetto” E, che in realtà è nient'altro che un **placeholder**.

L'oggetto viene tipizzato in fase di definizione della collection:

- ✓ il compilatore sostituisce ognuno dei placeholder con l'oggetto specifico che caratterizza la Collection .



Generics esempi

Creazione di un oggetto generic

```
import java.util.*;
public class VectorType<T> {

    private Vector<T>vettore = new Vector<T>();

    public void add(T elemento) {
        vettore.add(elemento);
    }
    public void printAll() throws Exception {
        Enumeration<T>en = vettore.elements();
        while(en.hasMoreElements()) {
            System.out.println(en.nextElement());
        }
    }
    public void printAllNewLoop() throws Exception {
        for(T elemento:vettore) {
            System.out.println(elemento);
        }
    }
}
```

Enumeration deve essere generic



nuova sintassi per i loop ciclo foreach





Generics *esempi*

Uso di una mappa K,V

```
Map<Integer,Libro>mappa =  
new Hashtable<Integer,Libro>();
```

```
mappa.put(new Integer(1),new Libro("Satyricon"));  
mappa.put(new Integer(2),new Libro("Iliade"));
```

```
Libro libro = mappa.get(new Integer(1));
```



Generics *wildcards*

Nella sintassi dei generics viene usata la keyword ? come wildcard.

Se infatti volessimo definire un metodo che accetti collezioni di un qualsiasi oggetto non possiamo usare `Collection <Object> coll`, in quanto **Object non è più supertype** di tutti gli oggetti.

La sintassi corretta è con l'uso del wildcard ?

`Collection <?> coll`

(collezione di oggetti sconosciuti la cui definizione è al momento della creazione)

Altri usi di wildcard (bounded wildcards):

- ✓ `<? extends type>` indica tutti i tipi che ereditano da `type`.
- ✓ `<? super type>` tutte le superclassi di `type`



Generics *wildcards*

```
import java.util.*;
public class GenericsUtility {
    public static void printAllCollections(Collection<?>collezione)
        throws Exception
    {
        for(Object oggetto:collezione) {
            System.out.println(oggetto);
        }
    }
    public static void printAllExtendsLibro(List<? extends Libro>libri)
        throws Exception
    {
        for(Object oggetto:libri) {
            System.out.println(oggetto.toString());
        }
    }
    public static void merge2Collection(
        Collection<? extends Libro>coll1,Collection<? extends Libro>coll2)
        throws Exception
    {
        ArrayList<Libro>dest = new ArrayList<Libro>();
        dest.addAll(coll1);
        dest.addAll(coll2);
        for(Object oggetto:dest) {
            System.out.println(oggetto.toString());
        }
    }
}
```



Autoboxing Unboxing

In Java esistono tipi primitivi (int, boolean, float) che non sono considerati Object: Java è un linguaggio orientato agli oggetti!

Per passare un tipo primitivo ad un metodo che accetti come parametro di ingresso un Object, occorre trasformare il tipo primitivo nell'oggetto corrispondente attraverso l'uso di una classe **wrapper** (Integer, Boolean, Float).

Con Java 5 la trasformazione del primitivo nell'oggetto wrapper, così come il procedimento inverso, viene fatto automaticamente dal compilatore, sollevando di fatto lo sviluppatore da questo compito.

```
Vector <Integer> vettore = new Vector<Integer>();  
for(int i=0;i<10;i++) {  
    vettore.add(i);    //autoboxing  
}  
for(Integer oggetto:vettore) {  
    System.out.println(oggetto);    //unboxing  
}
```



Varargs

E' possibile in Java 5 definire metodi e/o costruttori che ricevano un numero arbitrario di argomenti.

L'unica limitazione è che gli argomenti devono essere dello stesso tipo.

```
public void metodoVar(int intero, String...argomenti)
```

Il compilatore tratterà gli argomenti variabili come se fossero un array

✓ metodo equivalente considerato dal compilatore:

```
public void metodoVar(int intero, String[]argomenti)
```

✓ si può fare l'overloading di un metodo che ha come parametri varargs:

```
public void metodoVar(int intero, String...argomenti)
```

```
public void metodoVar(int intero, Object...argomenti)
```



Varargs

Se abbiamo un metodo nella forma:

```
public void metodoVar(int intero, String...argomenti)
```

non è possibile fare l'overload del metodo utilizzando un array dello stesso tipo del parametro varargs

```
public void metodoVar(int intero, String[]argomenti)
```

ci sarà un errore in compilazione in quanto il compilatore tratta i parametri varargs come un array

- ✓ abbiamo definito un metodo perfettamente identico al precedente



Static import

L'operatore **import** ora permette di poter usare all'interno delle proprie classi anche le porzioni statiche, sia che esse siano variabili o metodi.

- ✓ Molto comodi per poter usare nelle classi interfacce di costanti.
- ✓ Nel caso di membri e/o metodi con lo stesso nome possono esserci conflitti.

Se si importano membri omonimi (anche di tipo differente) da classi diverse, vi è un errore di compilazione.

L'import statico è possibile nelle forme:

```
import static package_name.class_name.member;  
import static package_name.class_name.method;  
//import di tutti i membri e metodi statici di una classe  
import static package_name.class_name.*;
```



Static import *wildcards*

```
package tiger.esempi.classi;
import static java.lang.System.out;
public class Utility {
    public static final String MESSAGGIO = "Tiger 5.0";

    public static void stampa(String...args) throws Exception {
        for(String stringa:args) {
            out.println(stringa);
        }
    }
    public void stampaNonStatica(String...args) throws Exception {
        for(String stringa:args) {
            out.println(stringa);
        }
    }
}
```

```
package tiger.esempi.runner;
import static tiger.esempi.classi.Utility.MESSAGGIO;
import static tiger.esempi.classi.Utility.stampa;
public class RunnerUtility {
    public static void main(String...args) throws Exception {
        stampa("Giovanni", "Marigi", "JavaPortal");
        stampa(MESSAGGIO);
    }
}
```



Enum

Permettono la definizione di nuovi tipi che accettano solo un set di valori predefiniti:

```
private enum Squadra { INTER, MILAN, JUVENTUS, ROMA};
```

Gli enum sono delle vere e proprie classi Java (non tipo primitivi) che estendono `java.lang.Enum`

```
private Squadra vincitore;  
public void setVincitore(Squadra vincitore) {  
    this.vincitore = vincitore;  
}  
public Squadra getVincitore() {  
    return this.vincitore;  
}
```

- ✓ Prevedono l'override del metodo `toString()`, implementano l'interfaccia `java.lang.Comparable`
- ✓ Per iterare sui valori di un enum si usa il metodo `values()`;



Enum esempio

```
public class Campionato {
    private String nome;
    private enum Squadra { INTER, MILAN, JUVENTUS};
    private Squadra vincitore;

    public void setVincitore(Squadra vincitore) {
        this.vincitore = vincitore;
    }
    public Squadra getVincitore() {
        return this.vincitore;
    }
    public void infoVincitore(Squadra vincitore) {
        switch(vincitore) {
            case INTER:
                System.out.println("Colori sociali: nero azzurro");
                break;
            case MILAN:
                System.out.println("Colori sociali: rosso nero");
                break;
            case JUVENTUS:
                System.out.println("Colori sociali: bianco nero");
                break;
        }
    }
}
```



Annotations

Le annotazioni permettono di specificare delle informazioni aggiuntive **metadati** al codice Java senza influenzarne la semantica.

Esempi di annotazioni già presenti nelle versioni precedenti di Java:

- ✓ javadoc
- ✓ transient (specifica che il campo non deve venir serializzato)

Con Tiger è possibile creare delle proprie classi di annotazioni per poter arricchire con meta informazioni metodi specifici, classi, campi

Le annotazioni non influenzano il comportamento del codice ma possono venir esplorate tramite reflection



Annotations *creazione*

Per definire una nostra interfaccia di annotazioni dobbiamo osservare queste semplici regole:

- ✓ creare metodi con segnatura e senza corpo che ritornano tipi standard, come int, boolean, String
- ✓ la definizione avviene tramite la sintassi
public @interface nome_annotazione {
- ✓ non usare variabili

Una volta definita la nostra annotazione la possiamo inserire prima di un metodo nel nostro codice riferendoci ad essa con il simbolo @ seguito dal nome dell'annotazione

Per specificare i valori associati ai metodi di un'annotazione si usa la sintassi metodo=valore



Annotations *esempio*

```
import java.lang.annotation.*;
@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.METHOD)
public @interface InfoMetodo {
    int versione();
    String creatoreMetodo();
    String autoreRevisione();
    int statoAvanzamento();
}

@InfoMetodo(versione=1.0f, creatoreMetodo="Paolo Rossi",
    autoreRevisione="Giovanni Marigi", statoAvanzamento=1)
public static void printAllCollections(
    Collection<?>collezione) throws Exception
{
    for(Object oggetto:collezione) {
        System.out.println(oggetto);
    }
}
```



Annotations *built in*

Esistono delle annotazioni base (built-in) già definite:

- ✓ Override il metodo ridefinisce quello della classe da cui eredita
- ✓ Deprecated il metodo dovrebbe non essere utilizzato più
- ✓ Inherited l'annotazione viene estesa anche nelle sottoclassi della classe in cui è stata applicata
- ✓ SuppressWarnings disabilita i messaggi di warning

@Retention – annotazione relativa solo a:

- ✓ RetentionPolicy.SOURCE // no JVM e compiler
- ✓ RetentionPolicy.CLASS // solo compiler
- ✓ RetentionPolicy.RUNTIME // solo JVM



Annotations esempio

Se si vuole esaminare a livello operativo le annotations create occorre ricorrere alla reflection:

```
public static void estrazioneAnnotazioni() throws Exception {
    Method metodi[] = GenericsUtility.class.getMethods();

    for(Method metodo: metodi){

        if(metodo.isAnnotationPresent(InfoMetodo.class)) {
            Annotation annotazioni[] = metodo.getAnnotations();
            for(Annotation a: annotazioni){
                System.out.println("\nMetodo: "+metodo.getName());
                System.out.println("Annotazione "+a);
                if(a instanceof InfoMetodo){
                    InfoMetodo info = (InfoMetodo)a;
                    System.out.println("Versione: "+info.versione());
                    System.out.println("Avanzamento: "+info.statoAvanzamento());
                    System.out.println("Autore: "+info.creatoreMetodo());
                }
            }
        }
    }
}
```



Format/Printf

E' stata introdotta la possibilità di formattare l'output in maniera "personalizzata" grazie all'introduzione dei metodi **format e printf** per gli oggetti `java.io.PrintStream`, `java.io.PrintWriter` e `java.lang.String`

```
format( String format, Object... args);  
printf( String format, Object... args);  
format( Locale locale, String format, Object... args);  
printf( Locale locale, String format, Object... args);
```

Il parametro `format` è una stringa che può includere uno o più elementi di formattazione, ognuno dei quali deve iniziare con il carattere `%`. Ogni elemento di formattazione viene poi applicato agli oggetti in entrata al metodo `format/printf`.



Format/Printf

Va notato come il metodo `format/printf` accetta un **varargs** come secondo argomento, il che ci permette di passare un numero arbitrario di oggetti da formattare.

Grazie al meccanismo dell'autoboxing/unboxing possiamo passare al metodo dei tipi primitivi i quali verranno automaticamente convertiti nella loro classe wrapper.



Format/Printf

Il formato che segue ogni elemento di formattazione è il seguente:

✓ %[argument_index\$][flags][width][.precision]conversion

```
System.out.printf("Valori: %2$f - %1$f.2", Math.PI, Math.E);
```

- ✓ `argument_index$`: specifica la posizione dell'oggetto da formattare
nell'esempio %2 si riferisce all'oggetto Math.E e %1 all'oggetto Math.PI
- ✓ `width`: minimo numero di caratteri che devono venir stampati
- ✓ `precision`: minimo numero di cifre decimali da stampare
nell'esempio %1\$f.2 specifica la stampa di due soli cifre decimali per il valore di Math.PI
- ✓ `conversion`: il tipo di oggetto che deve essere formattato
f float, t time, d decimal



Format/Printf *formattazione date*

Le date possono venire formattate attraverso la conversione %t o %T.

Introducendo un'altra opzione alla conversione %t possiamo specificare quale porzione di data vogliamo formattare e stampare.

tH, tI, tK, tL → per estrarre l'ora

tM → per estrarre i minuti

tr → tH:tM

tA → giorno della settimana

tB → nome del mese

te → numero del giorno del mese

tY → anno

```
System.out.printf("Ora %tr del " + "%<tA, %<tB %<te,  
%<tY.%n", Calendar.getInstance  
( ));
```

Per conoscere tutti gli elementi di formattazione consultare
le API dell'oggetto Formatter



Conclusioni

Quelle presentate sono alcune delle moltissime nuove features introdotte con Tiger

Esiste una sezione nel sito della Sun con articoli, how-to su tutte le nuove caratteristiche del linguaggio

<http://java.sun.com/reference/tigeradoption>