Java (J2SE) 1.5 (5.0)

Motivi per le modifiche

- “The new language features all have one thing in common: they take some common idiom and provide linguistic support for it. In other words, they shift the responsibility for writing the boilerplate code from the programmer to the compiler.”

  --Joshua Bloch, senior staff engineer, Sun Microsystems

Nuove caratteristiche

- Generics
  - Utilizzo di collezioni senza uso di casting esplicito (Compile-time type safety)
- Enhanced for loop

  Gestione automatica di un iterators
- Autoboxing/unboxing
  - Consente di evitare la conversione esplicita fra tipi primitivi Avoids (come int) e classi wrapper (come Integer)
- Typesafe enums
  - Fornisce tutti I benefici di un tipo Enum che abbia la caratteristica di essere Typesafe
- Scanner class API per semplificare l'input del testo
  - Funzionalita di input di base
- Etc …
- Miglioramento delle performance

Generics

- Un generic è un metodo che è ricompilato con differenti tipi secondo le necessità (simile ai template C++)
- Svantaggi:
  - Invece di : `List words = new ArrayList();`
  - Si deve definire:

    `List<String> words = new ArrayList<String>();`
- Vantaggi:
  - Fornisce una migliore gestione del type checking durante la compilazione
  - Evita il casting da Object. I.e., invece di

    `String title = ((String) words.get(i)).toUppercase();`

    utilizzaremo

    `String title = words.get(i).toUppercase();`

```
public interface List<E> {
   void add(E x);
   Iterator<E> iterator();
}

public interface Iterator<E> {
   E next();
   boolean hasNext();
}
```

```
List myIntList = new LinkedList(); // 1
myIntList.add(new Integer(0)); // 2
Integer x = (Integer) myIntList.iterator().next(); // 3

List<Integer> myIntList = new LinkedList<Integer>(); // 1'
myIntList.add(new Integer(0)); //2'
Integer x = myIntList.iterator().next(); // 3'
```

Tipi parametrizzati

Tutte le occorenze del parametro "formal type" (E in questo caso) sono sostituite dagli argomenti i "*actual type*" (in questo caso Integer).

```
List<String> ls = new ArrayList<String>(); //1
List<Object> lo = ls; //2
```

L'istruzione 2 genera un errore in compilazione

In generale, se C2 è una classe derivata da C1 e G è una dichiarazione generica, non è vero che G<C2> è un tipo derivato da G<C1>.

Wildcards

```
void printCollection(Collection c) {
        Iterator i = c.iterator();
        for (k = 0; k < c.size(); k++) {
                System.out.println(i.next());
}}
```

```
void printCollection(Collection<Object> c) {
for (Object e : c) { System.out.println(e);}}
```

Wildcards

```
void printCollection(Collection<?> c) {
        for (Object e : c) { System.out.println(e);}}
```

```
public abstract class Shape {
    public abstract void draw(Canvas c);}

public class Circle extends Shape {
    private int x, y, radius;
    public void draw(Canvas c) { ... }
}

public class Rectangle extends Shape {
    private int x, y, width, height;
    public void draw(Canvas c) { ... }
}

public void drawAll(List<Shape> shapes) {
    for (Shape s: shapes) { s.draw(this);}}

public void drawAll(List<? extends Shape> shapes) { ... }
```

- List<? **extends** Shape> è un esempio di *bounded wildcard.*
- *Il simbolo*? Sta per un tipo sconosciuto
- Sappiamo che in questo caso tale tipo sconosciuto e un subtype di Shape.
- Diremo che Shape è un *upper bound di una* wildcard.

Generic Methods

- Supponiamo di voler scrivere un metodo che prende un array di objects è una collection e pone gli oggetti dell'array in una collection.
- Soluzione

```
static void fromArrayToCollection
        (Object[] a, Collection<?> c)
        { for (Object o : a) { c.add(o); // compile
    time error
}}
```

- I metodi generici consentono di superare un tale problema.
- Cosi come in una dichiarazione di tipo, la dichiarazione di un metodo può essere generica cioè parametrizzata rispeotto ad uno o più parametri

```
static <T> void fromArrayToCollection
                (T[] a, Collection<T> c)
    { for (T o : a) { c.add(o); // correct}
}
```

Enhanced for loop

- Invece di

```
void cancelAll(Collection c) {
    for (Iterator i = c.iterator(); i.hasNext(); ) {
        TimerTask tt = (TimerTask) i.next();
        tt.cancel();
    }
}
```

- Si può utilizzare:

```
void cancelAll(Collection<TimerTask> c) {
    for (TimerTask task : c)
        task.cancel();
}
```

```
void cancelAll(Collection<TimerTask> c) {
    foreach TimerTask task of c)  //C# notation
        task.cancel();
}
```

  - Cosa fa automaticamente JDK 1.5?
  - Non ogni cosa come la seguente sintassi.

Autoboxing

- Java distingue fra tipi primitivi e oggetti
  - I tipi primitivi come int, double, sono compatti e supportano gli operatori aritmetici
  - Le classi (Integer, Double) hanno più metodi: Integer.toString()
  - Noi abbiamo bisogno di un "wrapper" per utilizzare I metodi:
    Integer ii = new Integer(i); ii.hashCode()
  - Allo stesso modo noi abbiamo bisogno di un tipo "unwrap" per utilizzare gli operatori aritmetici
    int j = ii.intValue() * 7;
- Java 1.5 fa queste operazioni automaticamente:
  - **Prima:**

```
ArrayList<Integer> list = new ArrayList<Integer>();
list.add(0, new Integer(42));
int total = (list.get(0)).intValue();
```

  - **Java 1.5:**

```
ArrayList<Integer> list = new ArrayList<Integer>();
list.add(0, 42);
int total = list.get(0);
```

Enumerations

- Una enumerazione, o “enum,” è semplicemente un insieme di costanti che rappresentano differneti valori
- Prima

```
public final int SPRING = 0;
public final int SUMMER = 1;
public final int FALL = 2;
public final int WINTER = 3;
```

- Questa soluzione è scomoda e soggetta ad errori
- Java 1.5

```
enum Season { winter, spring, summer, fall }
```

Vantaggi del nuovo enum?

- They provide compile-time type safety
  - int enums don't provide any type safety at all
- They provide a proper name space for the enumerated type
  - With int enums you have to prefix the constants to get any semblance of a name space.
- They're robust
  - int enums are compiled into clients, and you have to recompile clients if you add, remove, or reorder constants.
- Printed values are informative
  - If you print an int enum you just see a number.
- Because they're objects, you can put them in collections.
- Because they're essentially classes, you can add arbitrary fields and methods

Formatted output

- Simile al C/C++ printf e scanf:

```
System.out.printf("name count%n"); //newline
System.out.printf("%s %5d%n", user,total);
```

- Consultare la classe java.util.Formatter per ulteriori informazioni
- Formati supportati per le date, etc:

```
System.out.format("Local time: %tT",
Calendar.getInstance()); // -> "Local time: 13:34:18"
```

- Le stringhe possono essere formattate utilizzando String.format:

```
import java.util.Calendar;
import java.util.GregorianCalendar;
import static java.util.Calendar.*;
Calendar c = new GregorianCalendar(1995, MAY, 23);
String s = String.format("Duke's Birthday: %1$tm %1$te,%1$tY", c);
// -> s == "Duke's Birthday: May 23, 1995"
```

Classe Scanner

- Fornisce un meccanisco di input per leggere dati dalla console
- Per esempio per leggere una stringa dallo standard input e leggere un valore int:

```
Scanner s=Scanner.create(System.in);
String param= s.next();
int value=s.nextInt();
s.close();
```

  - Il metodo next e nexInt di Scanner bloccano se non sono disponibili dati
  - Forniscono il supporto per le regular expression
- Per processare input più complessi, algoritmi di pattern matching sono disponibili nella classe java.util.Formatter