

17.

Implementare i generics

Marco Faella

Dip. Ing. Elettrica e Tecnologie dell'Informazione
Università di Napoli “Federico II”

Corso di Linguaggi di Programmazione II

Parametri di tipo in altri linguaggi

- Il primo linguaggio a supportare parametri di tipo è stato ML (Meta-Language) nel 1973
- Parametri di tipo = *polimorfismo parametrico*
- **Java** supporta i generics dal 2005 (Java 5)
- Il **C++** ha un meccanismo simile alla programmazione generica, chiamato *template*, a partire dal 1990
- Il **C#** supporta i generics dal 2005 (C# 2.0)
- Ciascuno di questi linguaggi implementa i generics in modo diverso!

- La sintassi per le classi e i metodi template in C++ è simile a quella Java, ma l'implementazione è molto diversa:

```
template<class T1, class T2>
struct pair {
    T1 first;
    T2 second;

    pair(const T1& a, const T2& b) : first(a), second(b) { }
    ...
};
```

- Quando il compilatore C++ trova un riferimento ad una versione concreta di un template, come `pair<string, employee>`, esso *istanzia* una nuova copia della classe `pair`, con `string` al posto di `T1` ed `employee` al posto di `T2`
- Questo approccio, detto *reificazione a tempo di compilazione*, è diametralmente opposto a quello di Java

Riassumendo:

- C++
 - **reificazione** a tempo di compilazione
 - il compilatore crea una versione specifica del codice per ogni parametro attuale di tipo
- C#
 - **reificazione** a tempo di esecuzione
 - come in C++, ma a tempo di esecuzione (*on demand*)
- Java
 - **cancellazione** (*erasure*)
 - il compilatore usa i generics per fare un type checking più accurato, poi *scarta* l'informazione

- Approfondiamo il meccanismo implementativo scelto da Java e le limitazioni che questo meccanismo comporta sull'uso dei parametri di tipo
- Il meccanismo che supporta la programmazione generica prende il nome di *cancellazione* (in inglese, *erasure*)
- Il principio di funzionamento è il seguente:
 - 1) I parametri di tipo vengono usati dal compilatore per effettuare i dovuti controlli di tipo (type checking)
 - 2) Poi, tutti i parametri di tipo vengono **rimossi** (cancellati, appunto) e sostituiti da `Object`, oppure dal *primo limite superiore* del parametro in questione, se presente
 - 3) Il parametro di tipo jolly viene semplicemente rimosso
 - 4) In conseguenza della cancellazione, vengono inseriti degli opportuni cast, per ripristinare la coerenza tra tipi
- Di conseguenza, nel **bytecode** risultato della compilazione **non c'è più traccia dei parametri di tipo**
- Ovvero, in fase di esecuzione i tipi parametrici sono scomparsi
 - Fanno eccezione alcune funzionalità di riflessione (argomento di una lezione successiva), che sono in grado di recuperare a run-time alcuni parametri di tipo specificati nel sorgente

Limitazioni dei generics

Come conseguenza dell'erasure:

I parametri attuali di tipo non possono produrre effetti a runtime

A causa di questa limitazione, il linguaggio limita *a tempo di compilazione* quello che si può fare con i parametri formali di tipo

Non è possibile utilizzare un parametro formale di tipo per istanziare oggetti

`new T()` *(errore di compilazione)*

- Se fosse consentita, quest'istruzione **produrrebbe effetti a runtime** dipendenti dal parametro attuale che sostituisce T, cosa impossibile a causa dell'erasure
- In altri termini, a runtime tale istruzione diventerebbe "`new Object()`", che sicuramente non è quello che il programmatore intendeva ottenere
- D'altronde, è possibile utilizzare un parametro di tipo per istanziare una classe concreta, come in:

`new LinkedList<T>()`

- Infatti, quest'istruzione non produce effetti a runtime dipendenti dal parametro attuale che sostituisce T

Non è possibile utilizzare il jolly per istanziare oggetti

(questa regola è indipendente dall'erasure)

- Il parametro di tipo jolly non può essere utilizzato neanche per istanziare una classe concreta

`new LinkedList<?>()` *(errore di compilazione)*

- Quest'istruzione corrisponderebbe alla richiesta di creare una lista di tipo sconosciuto
- Ricordiamo che il parametro di tipo jolly serve per avere riferimenti in grado di puntare a diverse versioni di una classe parametrica
- Se il nostro intento è di creare una lista che possa contenere qualsiasi oggetto, il tipo giusto è `LinkedList<Object>`

Non è possibile istanziare un array di tipo parametrico

`new T[10]` (*errore di compilazione*)

- Gli array ricordano il tipo con il quale sono stati creati
- Se fosse consentita, quest'istruzione produrrebbe effetti a runtime dipendenti dal parametro attuale che sostituisce T
- Una possibile soluzione consiste nell'istanziare una lista di tipo T, invece di un array
- Invece, è possibile dichiarare un riferimento di tipo array di tipo parametrico

`T[] a;`

- Questo costrutto è utile, ad esempio, come parametro formale di un metodo, per accettare array di qualsiasi tipo ed associare il tipo dell'array a quello di altri parametri, oppure al tipo di ritorno

- I parametri di tipo presentano una serie di problemi legati all'overloading
- Per prima cosa:

Non è possibile usare un parametro di tipo per distinguere due versioni di un metodo

- Consideriamo la classe `Pair<S,T>`, che rappresenta una coppia di elementi di due tipi diversi
- Si potrebbe essere tentati di realizzarla secondo il seguente schema:

```
public class Pair<S, T> {  
    private S first;  
    private T second;  
    public void setValue(S x) { first = x; }  
    public void setValue(T x) { second = x; }    (errore di compilazione)  
    ...  
}
```

- Questa soluzione non funziona, perché entrambi i parametri di tipo, dopo il type checking, diventeranno `Object`, e quindi l'overloading diventerà non valido

Dopo l'erasure, un metodo parametrico potrebbe andare in conflitto con uno non parametrico

- Ad esempio, non possiamo avere i seguenti metodi nella stessa classe:

```
public <T> void f(T t)      { ... }  
public      void f(Object o) { ... } (errore di compilazione)
```

- Invece, i seguenti possono coesistere:

```
public <T> void f(T t)      { ... }  
public      void f(String s) { ... }
```

Non è possibile utilizzare un parametro di tipo
per selezionare una determinata versione di un metodo in overloading

- Ad esempio, consideriamo i seguenti metodi:

```
public void f(String s) { ... }  
public void f(Object o) { ... }
```

```
public <T> void g(T x) { f(x); }
```

- Indipendentemente dal valore assunto dal parametro di tipo T, il metodo **g chiamerà sempre f(Object)**
 - in particolare, anche se g sarà chiamato con T=String
- Infatti, la risoluzione dell'overloading avviene a tempo di compilazione, quando ancora non si conosce il valore che T assumerà
 - il compilatore non può che assumere T=Object

I parametri di tipo non vanno usati per effettuare conversioni esplicite (**cast**)

- In particolare, abbiamo:

`(T) x;` *(warning)*

a run-time diventa un cast verso `Object` (o verso il primo limite superiore di `T`, se presente)

- Qualunque cast verso un tipo parametrico (un parametro di tipo oppure una classe o interfaccia parametrica) produce un warning in compilazione:

`(LinkedList<String>) x` *(warning)*

- Se è necessario effettuare un cast (es., metodo `equals`), lo si può fare usando il parametro jolly:

`(LinkedList<?>) x`


1) `(List<String>) new Object()`

Warning, poi eccezione a run-time perché il tipo effettivo (`Object`) non è sottotipo di `List`

2) `(List<String>) new LinkedList<Integer>()`

Errore di compilazione perché `LinkedList<Integer>` non è sottotipo di `List<String>` (né viceversa)

3) `(List<String>) (List<?>) new LinkedList<Integer>()`



Warning, nessuna eccezione a run-time;

con questa sequenza di conversioni possiamo inserire una stringa in una lista di interi (peggio per noi...)

**Non si può applicare instanceof a un parametro formale di tipo
o a una classe parametrica**

Esempio:

`x instanceof T` *(errore di compilazione)*

A run-time diventerebbe "x instanceof Object"

Quindi, il risultato sarebbe sempre true (tranne che per null)

Non si può applicare instanceof a un parametro di tipo o a una classe parametrica

Inoltre:

```
x instanceof List<T>           (errore di compilazione)  
x instanceof List<String>      (errore di compilazione)
```

La JVM non può controllare se x è sottotipo di List<T> o List<String>, perché a runtime x sarà semplicemente una List (o una LinkedList, ArrayList, etc.)

Questo contesto è uno dei pochi casi in cui è opportuno usare la versione grezza:

```
x instanceof List
```

oppure:

```
x instanceof List<?>
```

Vantaggi della reificazione (C#, C++):

- Espressività: si può fare con un parametro di tipo tutto quello che si può fare con un tipo concreto

Vantaggi della cancellazione (Java):

- Evita il *code bloating* (ripetizione, nell'eseguibile o in memoria, di codice simile)
- Supporta la compilazione separata

- Nella lezione precedente, abbiamo esaminato il seguente metodo della classe java.util.Collections:

```
public static <T extends Object & Comparable<? super T>> T min(Collection<? extends T> l)
```

- A questo punto, è possibile analizzare in dettaglio questa firma
- Per individuare l'elemento minimo di una collezione, il metodo min ha bisogno che gli elementi della collezione siano mutuamente confrontabili tramite l'interfaccia Comparable
- Quindi, il parametro di tipo T, che rappresenta il tipo degli elementi della collezione, ha come limite superiore Comparable<? super T>
- Inoltre, prima che esistessero i tipi parametrici in Java, questo metodo era già presente, con la firma

```
public static Object min(Collection l)
```

- Il limite superiore <T extends Object & ...> fa sì che, dopo la cancellazione, la nuova firma coincida con la vecchia, a vantaggio della compatibilità con il codice pre-esistente
- Infine, il tipo del parametro Collection<? extends T> offre la garanzia che la collezione passata al metodo min non sarà modificata