16.
Parametri di tipo con limiti

Marco Faella

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

Corso di Linguaggi di Programmazione II

Esempio getTotalSalary

- Supponiamo di voler scrivere un metodo statico che accetta una lista di Employee e restituisce la somma dei loro salari

- Iniziamo con una versione non parametrica:

```
public static int getTotalSalary(List<Employee> l) {
     int tot = 0;
     for (Employee e: l)
         tot += e.getSalary();
     return tot;
}
```

V.1

- **Domanda**: possiamo passare a getTotalSalary una **List<Manager>?**

Esempio getTotalSalary

- Supponiamo di voler scrivere un metodo statico che accetta una lista di Employee e restituisce la somma dei loro salari

- Proviamo con questa versione non parametrica:

```
public static int getTotalSalary(List<Employee> l) {
     int tot = 0;
     for (Employee e: l)
         tot += e.getSalary();
     return tot;
}
```

V.1

- **Domanda**: possiamo passare a getTotalSalary una **List<Manager>?**

- **No**, perché **List<Manager> non è sottotipo di List<Employee>**

- Le prossime slide illustrano le regole di sottotipo tra tipi parametrici

Tipi parametrici e la relazione di sottotipo

- Una delle regole di sottotipo afferma che se A è sottotipo di B, array di A è sottotipo di array di B
- Questo principio non vale per i tipi parametrici
- Ovvero, se X è una classe con un parametro di tipo, ed A e B sono due tipi tali che A è sottotipo di B, **non** è vero che X<A> è sottotipo di X<B>
- A questo proposito, è interessante confrontare il comportamento di **array** e **collezioni**, rispetto ai sottotipi

Array e la relazione di sottotipo

- Consideriamo il seguente frammento di codice:

```
Manager[] managers = new Manager[10];
Employee[] employees = managers;
employees[0] = new Employee(...);
```

Compila, cioè supera il type-checking?
In caso affermativo, cosa succede a tempo di esecuzione?

Array e la relazione di sottotipo

- Consideriamo il seguente frammento di codice:

```
Manager[] managers = new Manager[10];
Employee[] employees = managers;
employees[0] = new Employee(...);
```

- Questo frammento **compila correttamente**, grazie alle regole di sottotipo per array
- Tuttavia, il terzo rigo provoca un'**eccezione a run-time**
- Infatti, non è possibile aggiungere un Employee ad un array che è stato dichiarato come Manager[]

- La prossima slide illustra cosa succede in una situaziona analoga, che utilizza ArrayList invece di semplici array

Tipi parametrici e la relazione di sottotipo

- Possiamo tentare di sviluppare un esempio simile a quello della slide precedente, sostituendo gli array con oggetti di tipo ArrayList

```
ArrayList<Manager> managers = new ArrayList<Manager>();
ArrayList<Employee> employees = managers;
employees.add(new Employee(...));
```

Tipi parametrici e la relazione di sottotipo

- Possiamo tentare di sviluppare un esempio simile a quello della slide precedente, sostituendo gli array con oggetti di tipo ArrayList

```
ArrayList<Manager> managers = new ArrayList<Manager>();
ArrayList<Employee> employees = managers;
employees.add(new Employee(...));
```

- Questa volta, il frammento di codice provoca un **errore di compilazione**
- In particolare, l'errore si verifica al secondo rigo, perché ArrayList<Manager> **non** è sottotipo di (e quindi non è assegnabile a) ArrayList<Employee>

- La situazione è quindi *migliore* della precedente, perché quello che prima era un errore al run-time ora è diventato un errore di compilazione, più facile da individuare e quindi correggere

Tipi parametrici e la relazione di sottotipo

- Concludiamo che il sistema dei tipi parametrici è **più robusto** di quello degli array, relativamente agli errori di tipo

- Infatti, le regole dei tipi parametrici garantiscono che, *se il programma viene compilato senza warning e se non utilizza array e cast, non possono verificarsi errori di tipo al run-time*

Limiti superiori

- Tornando al nostro esempio, per poter passare a getTotalSalary una lista di Manager, possiamo utilizzare un **parametro di tipo con limite superiore**

- Quando si dichiara un parametro di tipo, appartenente ad una classe oppure ad un metodo, si può specificare che quel parametro potrà assumere come valore **solo sottotipi di un tipo dato**

- Questo tipo viene chiamato *limite superiore* per il parametro in questione

- Un parametro di tipo può anche avere **più limiti superiori simultaneamente**

    - In questo caso, il parametro *attuale* di tipo dovrà essere sottotipo di ciascuno dei limiti superiori

Sintassi dei limiti superiori

- La sintassi per indicare uno o più limiti superiori è la seguente:

<T extends U1 & U2 & ...>

- U1 può essere una classe, mentre i successivi (eventuali) limiti devono essere interfacce
- Non avrebbe senso pretendere che il parametro attuale estendesse simultaneamente due classi

Esempio getTotalSalary

- Quindi, otteniamo la seguente versione dell'esercizio:

```
public static <T extends Employee> int getTotSalary(List<T> l) {
     int tot = 0;
     for (T e: l)
         tot += e.getSalary();
     return tot;
}
```

V.2

- E' possibile invocare questa versione passando una lista di **qualsiasi sottotipo di Employee**
- I riferimenti di tipo T (come "e" nell'esempio) sono trattati dal compilatore come se fossero di tipo Employee

La versione parametrica di Comparable e Comparator

- Le due interfacce Comparable e Comparator, già oggetto di una lezione precedente, sono in realtà parametriche:

```
public interface Comparable<T> {
     public int compareTo(T x);
}

public interface Comparator<T> {
     public int compare(T x, T y);
}
```

- Come si vede, il parametro di tipo permette di specificare che tipo di oggetti tali metodi sono in grado di confrontare
- Ad esempio, se la classe Employee intende fornire un ordinamento naturale tra impiegati (ad esempio, ordinamento alfabetico per nome), essa si presenterà come segue:

```
public class Employee implements Comparable<Employee>
```

Esempio getMin

- Supponiamo di voler scrivere un metodo statico che accetta una lista di oggetti dotati di ordinamento naturale, e restituisce l'elemento **minimo** della lista

Esempio getMin

- Supponiamo di voler scrivere un metodo statico che accetta una lista di oggetti dotati di ordinamento naturale, e restituisce l'elemento **minimo** della lista

- Proviamo con questa versione non parametrica:

```
public static Object getMin(List<Comparable<Object>> l) {
     Object min = null;
     for (Comparable<Object> x: l)
         if (min==null || x.compareTo(min) < 0)
              min = x;
     return min;
}
```

V.1

- **Problema**: possiamo passare a getMin solo una List<Comparable<Object>>

- Anche se Employee implementasse Comparable<Object> (che già sarebbe strano), comunque List<Employee> non sarebbe sottotipo di List<Comparable<Object>>

Esempio getMin

- Nella nuova versione di getMin, chiamiamo T il tipo degli elementi della lista...
- ...e stabiliamo che T debba essere sottotipo di Comparable
- Inoltre, siccome Comparable è a sua volta parametrica, bisogna specificare anche il suo parametro di tipo

- Otteniamo quindi la seguente soluzione:

```
public static <T extends Comparable<T>> T getMin(List<T> l) {
     T min = null;
     for (T x: l)
         if (min==null || x.compareTo(min) < 0)
              min = x;
     return min;
}
```

V.2

- Questa versione ha anche il vantaggio che il tipo restituito è lo stesso di quello della lista

Esempio

- Se Employee implementa Comparable<Employee>, è possibile passare al metodo precedente una lista di Employee

- Supponiamo poi che la classe **Manager** estenda Employee, e non implementi direttamente nessuna versione di Comparable
- Indirettamente, Manager implementerà anch'essa Comparable<Employee>

- In questa situazione, **non è possibile** passare al metodo getMin una lista di Manager, perché la classe Manager non implementa Comparable<Manager>, neanche indirettamente

- Le prossime slide mostrano come ovviare a questa limitazione

Il parametro di tipo jolly

- Le regole appena viste per la relazione di sottotipo tra tipi parametrici comportano, tra l'altro, che List<Object> *non* sia il supertipo comune a tutte le liste
- Quindi, sembrerebbe che, se un metodo intende accettare liste di ogni tipo, esso *debba necessariamente essere parametrico*
- Invece, il **parametro di tipo jolly**, rappresentato sintatticamente da un **punto interrogativo**, permette di raggiungere lo stesso scopo senza utilizzare parametri di tipo

Ovvero:

List<?> è il supertipo comune a tutte le versioni di List

Il parametro di tipo jolly

- Il parametro di tipo jolly si può usare solamente come **parametro attuale di tipo**, ed intuitivamente rappresenta un *tipo sconosciuto*

- Non c'è nessuna relazione tra due diverse occorrenze del parametro jolly

- Ad esempio, il seguente metodo accetta **due liste dello stesso tipo**

```
public static <T> void foo(List<T> l1, List<T> l2) { ... }
```

- mentre il seguente metodo accetta **due liste qualsiasi**

```
public static void bar(List<?> l1, List<?> l2) { ... }
```

Il parametro di tipo jolly

- Supponiamo che A<T> sia una classe parametrica
- Dichiarando un riferimento di tipo A<?>, è come se le occorrenze di T all'interno di A diventassero "?", cioè, "tipo sconosciuto"
- Questo pone delle limitazioni sull'uso che si può fare di quel riferimento

- In particolare, se un metodo della classe A<T> **accetta un argomento di tipo T**, usando un riferimento di tipo A<?> potremo passare a quel metodo solamente *null*
- Infatti, non sapendo qual è il tipo concreto che quel metodo si aspetta, *null* è l'unico valore che è lecito passargli

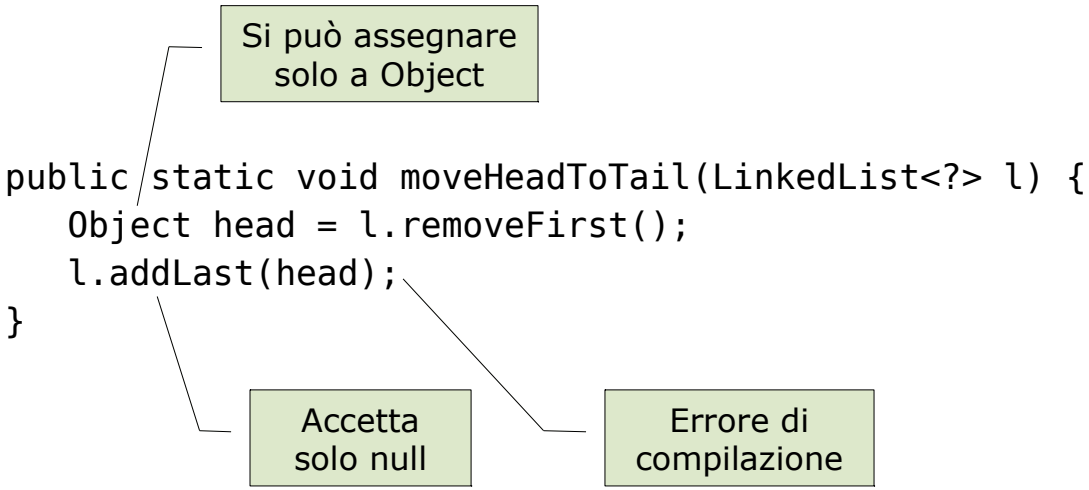
- Se invece un metodo della classe A<T> **restituisce un valore di tipo T**, chiamandolo con un riferimento di tipo A<?> potremo assegnare il valore restituito solamente ad un riferimento di tipo Object
- Infatti, Object è l'unico tipo che sicuramente accetta qualunque tipo sconosciuto (non primitivo)

Esempio: limitazioni del jolly

- Il seguente metodo estrae la testa di una LinkedList e la inserisce in coda:

```
public static <T> void moveHeadToTail(LinkedList<T> l) {
   T head = l.removeFirst();
   l.addLast(head);
}
```

- Se sostituiamo il parametro T con ?, *non* possiamo svolgere lo stesso compito:



Parametro di tipo jolly con limiti superiori e inferiori

- Come i normali parametri di tipo, **anche il parametro di tipo jolly può avere un limite superiore** (ma non più di uno)

- Ad esempio, List<? extends Employee> è una lista di tipo sconosciuto che estende Employee
- Precisamente, List<? extends Employee> è il supertipo comune a tutte le liste il cui parametro di tipo estende Employee

- A differenza dei normali parametri di tipo, il tipo jolly può anche avere un ***limite inferiore***
- Ad esempio, List<? super Employee> rappresenta una lista di un tipo sconosciuto che è supertipo di Employee (come Person o Object)

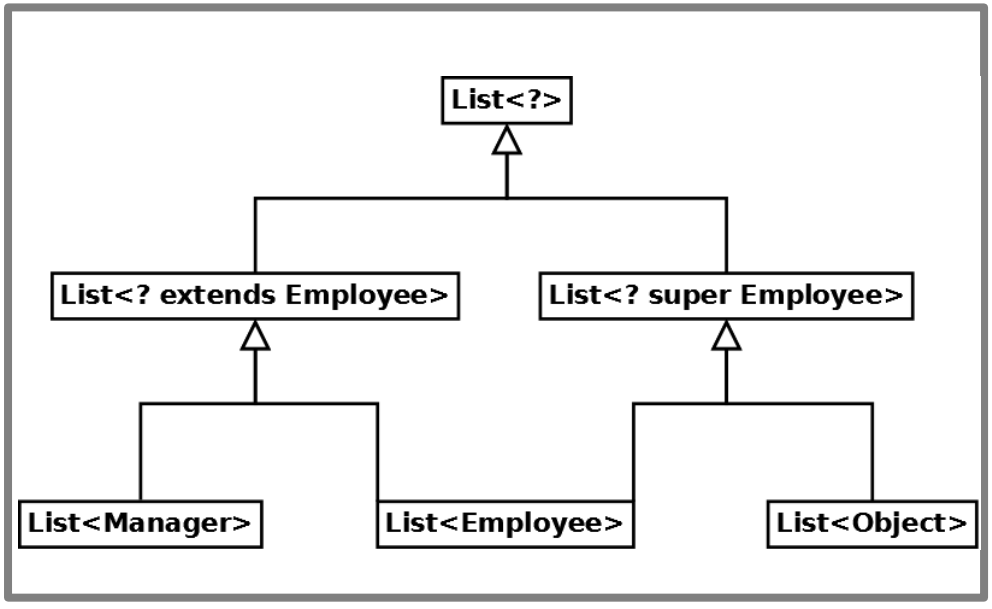


Figura 1: La relazione di sottotipo tra diverse versioni di List.

Esempio getMin

- Consideriamo nuovamente il metodo getMin

- Il parametro di tipo jolly permette di dichiarare il metodo in modo tale da accettare una lista di Manager, anche se Manager implementa Comparable<Employee>:

```
public static <T extends Comparable<? super T>> T getMin(List<T> l) { ... }
```

V.3

- In effetti, un metodo molto simile si trova nella classe java.util.Collections della libreria standard:

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

- I dettagli di questa firma verranno esaminati nella lezione successiva

Versioni di getMin

Riassumendo, consideriamo le seguenti intestazioni per getMin:

`Object getMin(List<Comparable<Object>> l)`	non accetta List<Employee>
`<T> T getMin(List<Comparable<T>> l)`	non accetta List<Employee>
`<T extends Comparable<T>> T getMin(List<T> l)`	non accetta List<Manager>
`<T extends Comparable<? super T>> T getMin(List<T> l)`	ok

Nota: Quest'ulteriore proposta, che usa un secondo parametro di tipo invece di un jolly, non compila (fino a Java 17) a causa di una limitazione del sistema dei tipi:

```
<T, S extends T & Comparable<T>> S getMin(List<S> l)
```

Messaggio di errore: *a type variable may not be followed by other bounds*

Esempio getSum

- Realizziamo un metodo che accetta una collezione di valori numerici e ne restituisce la somma
- Le collezioni non possono contenere tipi primitivi, quindi ci riferiamo alle classi wrapper di tipo numerico
- Tutte queste classi estendono Number

Esempio getSum

- Realizziamo un metodo che accetta una collezione di valori numerici e ne restituisce la somma
- Le collezioni non possono contenere tipi primitivi, quindi ci riferiamo alle classi wrapper di tipo numerico
- Tutte queste classi estendono Number
- Number offre il metodo “doubleValue”, che converte in double questo valore numerico, qualunque sia il suo tipo

```
public static double getSum(Collection<? extends Number> c)
{
     double sum = 0.0;
     for (Number n: c)
          sum += n.doubleValue();
     return sum;
}
```

- Il parametro jolly ci ha consentito di scrivere un metodo getSum che non è parametrico
- Il metodo getSum accetterà una collezione di qualunque tipo wrapper numerico

Esempio getMinMax

- Realizziamo un metodo che accetta una collezione di oggetti confrontabili e una **coppia di oggetti** (di una ipotetica classe Pair<T>) e modifica la coppia in modo che contenga l'oggetto minimo e quello massimo della collezione

Esempio getMinMax

- Realizziamo un metodo che accetta una collezione di oggetti confrontabili e una **coppia di oggetti** (ipotetica classe Pair<T>) e modifica la coppia in modo che contenga l'oggetto minimo e quello massimo della collezione

```
public static <T extends Comparable<? super T>>
     void getMinMax(Collection<T> c, Pair<? super T> p)
{
     T min = null, max = null;
     for (T x: c) {
          if (min==null || x.compareTo(min)<0)
               min = x;
          if (max==null || x.compareTo(max)>0)
               max = x;
     }
     p.setFirst(min);
     p.setSecond(max);
}
```

- Il primo argomento potrebbe anche essere dichiarato di tipo Collection<? extends T>, per esprimere la garanzia che la collezione non verrà modificata (si veda lezione successiva)

Limitazioni imposte dal parametro jolly

- Come abbiamo visto per il tipo jolly senza limiti, l'uso del parametro jolly con limiti superiori o inferiori impone determinate condizioni sulle chiamate ai metodi
- Supponiamo che A<T> sia una classe parametrica
- La seguente tabella riassume le limitazioni che valgono per un riferimento di tipo A<?>, A<? extends B>, oppure A<? super B>, rispetto ad un metodo di A che accetta un argomento di tipo T, oppure restituisce un valore di tipo T

tipo del riferimento	**cosa si può passare a f(T)**	**a cosa si può assegnare T f()**
A<?>	solo null	solo ad Object
A<? extends B>	solo null	a B e ai suoi supertipi
A<? super B>	B e suoi sottotipi	solo ad Object

La sigla PECS

- Il comportamento del jolly con limiti superiori e inferiori è stato riassunto dalla sigla **PECS**:

  Producer Extends, Consumer Super

- Secondo questo acronimo:

  - Un oggetto che **fornisce** oggetti di tipo T *(produttore)* dovrebbe essere parametrizzato con <? extends T>
  - Esempio: Set<? extends T>, se usato per leggerne il contenuto

  - Un oggetto che **accetta** oggetti di tipo T *(consumatore)* dovrebbe essere parametrizzato con <? super T>
  - Esempio: Comparator<? super T>
  - Esempio: Set<? super T>, se usato per inserire, quindi consumare, oggetti

Ulteriori informazioni nella lezione su "Scelta dell'interfaccia migliore per un metodo".

Si veda anche:
Effective Java, 3ª edizione, di Joshua Bloch, Item 31

Trovare gli errori di compilazione nel seguente frammento:

```
LinkedList<Employee> l = new LinkedList<Employee>();
l.add(new Employee(...));
l.add(new Manager(...));
l.add(new Person(...));

LinkedList<?> l1 = l;
LinkedList<? extends Person> l2 = l;
LinkedList<Employee> l3 = l1;

l1.addFirst(l1.removeLast());
List<?> l4 = l1;
System.out.println(l1.getClass() == l.getClass());
l2.add(new Employee(...));
```

Trovare gli errori di compilazione nel seguente frammento:

```
LinkedList<Employee> l = new LinkedList<Employee>();
l.add(new Employee(...));
l.add(new Manager(...));
l.add(new Person(...));                          // errore

LinkedList<?> l1 = l;
LinkedList<? extends Person> l2 = l;
LinkedList<Employee> l3 = l1;                    // errore

l1.addFirst(l1.removeLast());                    // errore
List<?> l4 = l1;
System.out.println(l1.getClass() == l.getClass()); // stampa true
l2.add(new Employee(...));                       // errore
```