UNIVERSITÀ
DEGLI STUDI
DELL'AQUILA

DISIM
Dipartimento di Ingegneria
e Scienze dell'Informazione
e Matematica

Informatica
@UnivAQ

Laboratorio di Algoritmi e Strutture Dati
a.a. 2024/2025

GENERICS: richiami

Giovanna Melideo
Università degli Studi dell'Aquila
DISIM

Generics: richiami

- La versione 1.5 ha introdotto un'importante funzionalità nel linguaggio: la programmazione parametrica, in Inglese anche detta ***generics***.
- Si tratta della possibilità di specificare il tipo di un elemento dotando classi, interfacce e metodi di **parametri di tipo**. Questi parametri hanno come **possibili valori i tipi del linguaggio**.
  - In particolare, possono assumere come valore qualsiasi tipo, esclusi i tipi primitivi (tipi base).

Vantaggi

- Questo meccanismo consente di scrivere **codice più robusto** dal punto di vista dei tipi di dato (fornisce una migliore gestione del **type checking** durante la compilazione), evitando in molti casi il ricorso al casting da `Object`
- La programmazione parametrica dimostra tutta la sua utilità nella realizzazione di **collezioni**

Generics: un esempio

```
interface ListOfStrings {
  boolean add(String element);
  Number get(int index);
}
interface ListOfIntegers {
  boolean add(Integer element);
  Integer get(int index);
}
      … ListOfOtherType …

// Tipo generico con parametro di tipo E
  interface List<E> {
  boolean add(E n);
  E get(int index);
}
```

Il tipo generico può essere istanziato usando qualunque tipo come parametro attuale

- `List<Integer>`
- `List<String>`
- `List<List<String>>`
- …

Dichiarazione di generics

```
class Name<TypeVar1, …, TypeVarN> {…}
interface Name<TypeVar1, …, TypeVarN> {…}
```

- All'interno di classi e interfacce, un parametro di tipo si comporta (tranne poche eccezioni) come un tipo di dati vero e proprio
  - In particolare, un parametro di tipo si può usare come tipo di un campo, tipo di un parametro formale di un metodo e tipo di ritorno di un metodo
- **Convenzioni**: T per Type, E per Element (generalmente per elementi di una collezione), K per Key, V per Value
- Istanziare una classe generica significa fornire un valore di tipo `Name<Type1, …, TypeN>`

Classi parametriche: uso

- Usare un tipo parametrico significa istanziare la classe per creare riferimenti ad oggetti. Es:

```
LinkedList<Integer> intList = new LinkedList<Integer>();
```

- Tutte le occorrenze dei parametri formali sono rimpiazzate dall'argomento (parametro attuale)
- Diversi usi generano tipi diversi
- Le classi parametriche sono compilate una sola volta e danno luogo ad un unico file `.class`

Interfacce parametriche

- Anche **le interfacce** possono essere dichiarate come parametriche o generiche, come:

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

- Ad esempio la classe **`String`** implementa **`Comparable<String>`**

Metodi parametrici

- Anche i singoli metodi e costruttori possono avere parametri di tipo, indipendentemente dal fatto che la classe cui appartengono sia parametrica o meno
  - Il parametro di tipo va dichiarato prima del tipo restituito, racchiuso tra parentesi angolari
  - Questo parametro è visibile solo all'interno del metodo
  - Notare che i metodi statici non possono utilizzare i parametri di tipo della classe in cui sono contenuti

Metodi parametrici: esempio

- Il seguente **metodo parametrico** restituisce l'elemento mediano (di posto intermedio) di un dato array

```
public static <T> T getMedian(T[] a) {
  int l = a.length;
  return a[l/2];
}
```

- In questo caso, il parametro di tipo permette di restituire un oggetto dello stesso tipo dell'array ricevuto come argomento

Metodi parametrici: uso

- Quando si invoca un metodo parametrico, è opportuno, ma non obbligatorio, specificare il parametro di tipo attuale per quella chiamata
- Ad esempio, supponendo che il metodo **`getMedian`** appartenga ad una classe **`Test`**, lo si può invocare così:

```
String[] x = {"uno", "due", "tre"};
String s = Test.<String>getMedian(x);
```

- Il parametro attuale di tipo va quindi indicato prima del nome del metodo

Type inference

- È possibile omettere il parametro attuale di tipo. In questo caso, il compilatore cercherà di dedurre il tipo più appropriato, mediante un meccanismo chiamato ***type inference*** (inferenza di tipo)
- La type inference cerca di individuare il tipo più specifico che rende la chiamata corretta
- L'algoritmo di type inference non è né corretto né completo

  (Le regole precise che il compilatore adotta nella type inference esulano dagli scopi di questo corso)

Costruttori parametrici

- Anche i **costruttori** possono essere parametrici, indipendentemente dal fatto che la loro classe sia parametrica o meno

```
Public class NomeClasse<T> {
        Public <U> NomeClasse(T x, U y) { ... }
 … }
```

- Il costruttore della classe parametrica **`NomeClasse`** ha a sua volta un **parametro di tipo chiamato `U`**
- Mentre il parametro **`T`** è visibile in tutta la classe, il parametro **`U`** è visibile solo all'interno di quel costruttore

Costruttori parametrici: uso

- Il costruttore in questione può essere invocato con la seguente sintassi

```
NomeClasse<String> a =
    new <Integer>NomeClasse<String>("ciao", Integer.valueOf(100));
```

- Il parametro di tipo del costruttore (**Integer**) va specificato prima del nome della classe
- Il parametro di tipo della classe va specificato dopo il nome della classe

Vincoli di tipo

```
<T extends SuperType>
```

- upper bound: va bene il supertype o uno dei suoi sottotipi

```
<T super SubType>
```

- lower bound: va bene il sottotipo o uno qualunque dei suoi supertipi

- Concettualmente questo corrisponde a forzare una sorta di precondizione sulla istanziazione del tipo
- Compile-time error se il tipo effettivo non rispetta i vincoli

Vincoli di tipo: esempio

```
interface List2<E extends Number> {…}

List2<Date>
// compile-time error
// Date non è sottotipo di Number
```

- Si possono effettuare tutte le operazioni compatibili con il limite superiore della gerarchia

Esempio

```
class List1<E> {   \\ come <E extends Object>
      void m(E arg) {
      arg.asInt( );
      // compiler error, E potrebbe non avere asInt}
}

class List2<E extends Number> {
      void m(E arg) {
      arg.asInt( );
// OK, Number e i suoi sottotipi supportano asInt }
}
```

Retro-compatibilità

- Per compatibilità con le versioni precedenti di Java, è possibile usare una classe (o interfaccia) parametrica come se non lo fosse
- Quando utilizziamo una classe parametrica senza specificare i parametri di tipo, si dice che stiamo usando la **versione grezza** di quella classe
  - Es: *l'interfaccia grezza Comparable*
- La versione grezza di queste classi permette alla nuova versione della libreria standard di essere compatibile con i programmi scritti con le versioni precedenti del linguaggio
  - Le classi grezze esistono solo per retro-compatibilità

Generics e sottotipi

- I meccanismi di subtyping si estendono alle classi generiche:

```
class C<T> implements / extends D<T> {…}
```

  - `C<T>` è sottotipo di `D<T>` per qualunque `T` (**`C<T>` ≤ `D<T>`**)
- Analogamente:

```
class C<T> implements / estends D {…}
```

  - **`C<T>` ≤ `D`** per qualunque `T`

Generics e sottotipi: esempi

- `ArrayList<E>` ≤ `List<E>`
- `List<E>` ≤ `Collection<E>`

Ma se …

- `Integer` ≤ `Number`
- ***`List<Integer>` ≤ `List<Number>` ?***

Meccanismi di subtyping

- In generale: se `Type1 ≤ Type2`, allora per ogni classe o interfaccia generica `GenType<>`, `GenType<Type1>` ***non*** è un sottotipo di `GenType<Type2>`
  - `Collection<T>` non è sottotipo di `Collection<Object>` per nessun `T != Object`
  - `Collection<Object>` non è supertipo di `Collection<T>` per nessun `T != Object`

Meccanismi di subtyping: esempio 1

- Sia `static void printSet(Set<Object> els) {…}` un metodo nella classe `Test` che stampa gli elementi di una qualunque collection

```
Set<String> s = new TreeSet<String>();
  …
printSet(s); //errore
```

  - *"The method printSet(Set<Object>) in the type Test is not applicable for the arguments (Set<String>)"*

Generics e sottotipi: invarianza

- Un operatore sui tipi F è:
  - **covariante** se `F<T'> ≤ F<T>` quando `T' ≤ T`
    - (F conserva la relazione di sottotipo).
  - **controvariante** se `F<T> ≤ F<T'>` quando `T' ≤ T`
    - (F inverte la relazione di sottotipo)
  - **invariante** se non è né covariante né controvariante
- Formalmente: la nozione di sottotipo usata in Java è **invariante** per le classi generiche

Wildcard

Una «Wildcard» è una variabile di tipo **anonima**

- ? indica un qualche tipo, non specificato
- ? notazione semplificata per `<? extends Object>`
- `<? extends Type>` indica un sottotipo non specificato di `Type`
- `<? super Type>` indica un supertipo non specificato di `Type`

Wildcard: esempio 1

Quale differenza c'è tra

1) `List<T>`

2) `List<? extends T> ?`

Nel caso 2) il tipo anonimo è un sottotipo sconosciuto di T

Wildcard: esempio 2

```
interface Set<E> {
     void addAll(Collection<? extends E> c);
}
```

- maggiormente flessibile rispetto a

```
void addAll(Collection<E> c);
```

- espressiva come

```
<T extends E> void addAll(Collection<T> c);
```

Wildcard e sottotipi

Per ogni classe o interfaccia generica **GenType, vale**

GenType<T> ≤ GenType<?>

T' ≤ T → GenType<T'> ≤ GenType<? Extends T>

- `Collection<T> ≤ Collection<?>`
- `Integer ≤ Number`
  - non implica `GenType<Integer> ≤ GenType<Number>`
  - ma `GenType<Integer> ≤ GenType<? extends Number>`

Wildcard: uso

Quando si usano le wildcard?

- si usa `<? extends T>` nei casi in cui si vogliono ottenere dei valori (da un produttore di valori)
- si usa `<? super T>` nei casi in cui si vogliono inserire valori (in un consumatore)
- non vanno usate (basta T) quando si ottengono e si producono valori

```
<T> void copy(List<? super T> dst, List<? extends T> src);
```

Meccanismi di subtyping: esempio 2

- Sia **`static void printSet(Set<?> els) {…}`** un metodo nella classe `Test` che stampa gli elementi di un qualunque collection

```
Set<String> s = new TreeSet<String>();
  …
printSet(s); //OK
```

- **Corretto perché Set<String> ≤ Set<?>**

Attenzione

```
ArrayList<?> c = new ArrayList<String>();

c.add(new String()); // errore
```

- Poiché non sappiamo esattamente quale tipo indica **?**, **non possiamo inserire elementi nella collezione**.
- In generale, non possiamo modificare valori che hanno tipo ?

Covarianza degli array

- Domanda: se `Type1 ≤ Type2`, che relazione esiste tra `Type1[]` e `Type2[]`?
  - Possiamo ipotizzare che se `Type1 ≤ Type2` i tipi `Type1[]` e `Type2[]` non dovrebbero essere correlati. Invece…

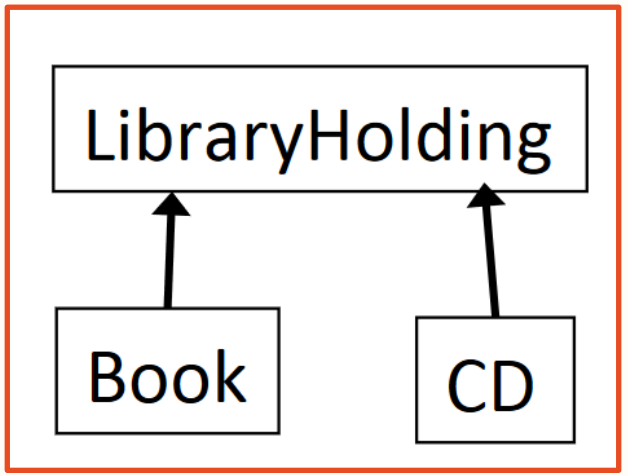
Peculiarità di Java:

- se `Type1 ≤ Type2`, allora `Type1[] ≤ Type2[]`

Gli array: esempio

```
void maybeSwap(LibraryHolding[ ] arr) {
  if(arr[17].dueDate( ) < arr[34].dueDate( ))
     // ... swap arr[17] and arr[34]
}

// cliente
Book[ ] books = ...;
maybeSwap(books); // uso covarianza array
```



Gli array: esempio (continua)

```
void replace17(LibraryHolding[ ] arr, LibraryHolding h)
{ arr[17] = h; }

// cliente
Book[] books = ...; //Notice that Book[] ≤ LibraryHolding[ ]
LibraryHolding theWall = new CD("Pink Floyd", "The Wall", …);
replace17(books, theWall);
Book b = books[17]; // contiene un CD
b.getChapters( );   // problema!!
```

- Attenzione agli array in Java!

UNIVERSITÀ
DEGLI STUDI
DELL'AQUILA

DISIM
Dipartimento di Ingegneria
e Scienze dell'Informazione
e Matematica

Informatica
@UnivAQ

Domande?

Giovanna Melideo
Università degli Studi dell'Aquila
DISIM