ALTRI PARADIGMI DI PROGRAMMAZIONE

A.A. 2020/2021

Laurea triennale in Informatica

3: Classi e Tipi (2/2)

NON SOLO CLASSI

In Java, l'unità principale di organizzazione del codice è la *Classe*.

Ma non è l'unica. Altri costrutti arricchiscono il sistema dei tipi di Java, o sono stati aggiunti nel tempo in risposta all'evoluzione del mercato e della pratica.

INTERFACCE

Una Interfaccia dichiara le caratteristiche di un *Tipo* senza fornire una sua implementazione.

Le classi possono dichiarare di *implementare* una interfaccia fornendo l'implementazione richiesta.

```java
package it.unipd.app2020;

interface Baz {
  int TEST = 1;
  void bar();
  String desc(boolean b);
}

class Foo extends App implements Baz {
  private String c;
  void bar() {};
  String desc(boolean b) { return ""; }
}
```

Speaker notes

la classe `Foo` è **obbligata** a fornire un'implementazione di `desc()`, a meno di non essere astratta.

Le interfacce in Java permettono di avere una sorta di ereditarietà multipla mitigando il Diamond Problem. Una classe può estendere una sola altra classe, ma implementare molte interfacce.

Il loro uso principale è quello di stabilire un *contratto* fra le implementazioni e l'utilizzatore, consentendo di sostituire implementazioni differenti senza impatti sul

Speaker notes

E' comune, per esempio, che una libreria fornisca un insieme di interfacce per affrontare un certo problema, ed un insieme più ampio di implementazioni con caratteristiche differenti, che possono essere selezionate al runtime a seconda delle circostanze. Il codice utilizzatore non dipende dalla implementazione, ma può usare quella che gli viene fornita. L'assimmetria con l'ereditarietà di classi consente di risolvere le ambiguità.

Una interfaccia può essere estesa solo da un'altra interfaccia.

Una interfaccia può avere visibilità pubblica o di package.

Tutti i membri di una interfaccia sono pubblici, senza necessità di indicarlo.

Speaker notes

Cercare di indicare altrimenti non viene accettato dal compilatore. Per completezza, una interfaccia può essere `static`, `private` o `protected` quando è definita all'interno di un'altra classe.

Una interfaccia può contenere:

- dichiarazioni di costanti
- metodi astratti
- metodi statici

Speaker notes

ogni variabile dichiarata in una interfaccia è implicitamente `public static final`, vale a dire costante. Non è necessario indicare un metodo come `abstract`. Tutti i metodi sono implicitamente `public`. Una interfaccia può contenere anche la definizione di interfacce interne.

Dopo Java 8, una interfaccia può contenere anche:

- metodi di default (da Java 8)
- metodi privati (da Java 9)

Speaker notes

i metodi privati non creano Diamond Problem perché non sono visibili dalle implementazioni. I metodi di default si.

METODI DI DEFAULT

Nella realizzazione di Java 8, Brian Goetz e gli altri autori si sono trovati a dover affrontare un problema molto complesso:

da un lato, la libreria standard necessitava di un profondo aggiornamento, sia per correggere errori storici che per includere innovazioni stilistiche e cambiamenti nel modo di programmare che si erano accumulati in oltre 15 anni

dall'altro, per le caratteristiche delle Interfacce e del linking dinamico di Java, non era possibile modificare una interfaccia senza "rompere" tutto il codice che la utilizzava. Nel caso della libreria standard, questo era un problema enorme.

Speaker notes

Aggiungere un metodo ad una interfaccia richiedeva di modificare **tutte** le implementazioni, aggiungendo a ciascuna il relativo corpo del nuovo metodo.

La soluzione è diventata parte della JSR335 che introduce il concetto di *default method*.

Speaker notes

Sebbene avesse come focus l'introduzione delle Lambda Expressions, il *default method* ne è un indispensabile abilitatore. La JSR indica tutta una serie di regole per risolvere la compatibilità fra codice scritto prima della modifica di una interfaccia e codice scritto dopo la sua estensione, in modo da rendere più permissive le regole di compatibilità binaria al momento del caricamento.

Alle interfacce viene permesso di avere dei *default method*, cioè dei metodi implementati che si comportano in modo simile a quello delle superclassi.

Questo di fatto re-introduce l'ereditarietà multipla in Java.

Speaker notes

Un metodo di default viene prefissato dalla parola chiave `default` e viene fornita la sua implementazione.

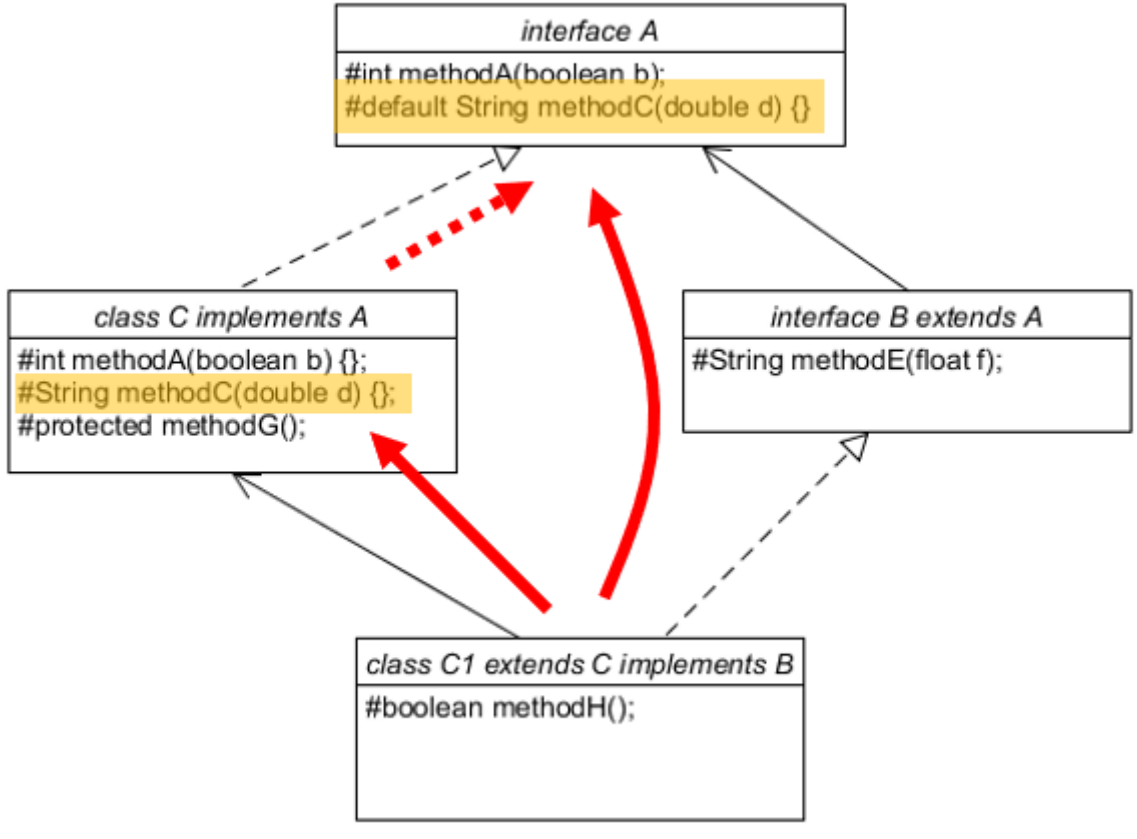
```
interface Top {
  default String name() { return "unnamed"; }
}

interface Left extends Top {
  default String name() { return getClass().getName(); }
}

interface Right extends Top {}

interface Bottom extends Left, Right {}
```

Speaker notes

In questo caso, `Bottom` usa l'implementazione di `Left` in quanto più vicina in senso ereditario. Le regole per le interfacce sono separate da quelle delle classi.

Per mezzo dei *default method* una interfaccia può essere modificata con nuovi metodi senza che le implementazioni debbano essere toccate; se il metodo nuovo non è gestito dalla classe, viene usato quanto dichiarato nell'interfaccia.

Tuttavia, il Diamond Problem si ripresenta:



Il Diamond Problem viene rilevato al momento della compilazione: se la gerarchia di ereditarietà ed implementazioni di una classe porta ad una ambiguità nella selezione di un metodo, il compilatore segnala un errore, che può essere risolto solo modificando il codice

Speaker notes

La gerarchia dei tipi deve essere rivista per conservare uno solo dei percorsi che portano al metodo ora ambiguo. Questo può costituire un problema molto laborioso da risolvere. Quello che si osserva, nella pratica, è che più il codice è specifico, più bassa dev'essere mantenuta la gerarchia delle classi; solo codice particolarmente generico, e accuratamente studiato, può permettersi una gerarchia molto profonda.

ANNOTAZIONI

Una interfaccia il cui nome inizia con `@` diventa di un tipo particolare, detto *annotazione*.

Le annotazioni si possono applicare sintatticamente a classi, metodi e variabili, e sono disponibili al momento dell'esecuzione.

Speaker notes

Le annotazioni sono state introdotte in Java 5.

Il loro scopo è arricchire di *metadati* la struttura a cui sono applicate, in modo da consentirne la rilevazione e l'uso durante la compilazione o l'esecuzione.

Alcune annotazioni della libreria standard:

Tipo	Uso
`@Deprecated`	Segnala un metodo che verrà rimosso in futuro
`@Override`	Segnala un membro che sostituisce o implementa un membro di superclasse o interfaccia

Tipo	Uso
`@SuppressWarnings`	Istruisce il compilatore a sopprimere gli avvisi nel costrutto annotato
`@FunctionalInterface`	Indica al compilatore che l'interfaccia può essere usata in modo "funzionale"

Le annotazioni possono avere parametri (solo di determinati tipi) e anche metodi (che possono ritornare lo stesso limitato insieme di tipi).

Il compilatore può eseguire degli *Annotation Processors* che, durante la compilazione, possono produrre nuovi file (nuovi sorgenti, nuove classi, o qualsiasi altro tipo di file) a partire da annotazioni

Speaker notes

Questa API rende relativamente semplice un insieme di tecniche di metaprogrammazione. Inoltre, essendo l'API standard, gli strumenti di sviluppo possono interagire in modo definito con i Processor.

```
class Foo extends Bar {
  @Override
  String methodB() { return ""; }

  @Deprecated
  @Override
  int methodOld { return 0; }

  @SuppressWarnings("unused")
  private boolean b = false;
}
```

TIPI GENERICI

Il primo importante aggiornamento di Java fu la versione 5 del 2004, tanto che cambiò la numerazione della versione da 1.5 a 5.0.

Uno dei cambiamenti radicali introdotti nel linguaggio in quella versione è stato il supporto per i cosiddetti *Generics*, ovvero *1-kind parametric types*.

Speaker notes

uno dei principali autori del nuovo compilatore per Java 5 è stato Martin Oderski, che dagli esperimenti realizzati nello studio di questa modifica al linguaggio ha tratto il linguaggio Scala.

Una classe (o una interfaccia, o un metodo) *generica* dichiara uno o più parametri di tipo che possono essere specificati in seguito:

```
interface MappableList< T > {
  void add(T element);
  T head();
  List< T > tail();
  < M > List< M > map(Function< T, M > xform);
}
```

Speaker notes

Una eccezione non può essere una classe generica. In una classe o interfaccia, il parametro di tipo va indicato dopo il nome. In un metodo, prima del tipo di ritorno: nell'esempio, la dichiarazione del metodo `map` indica il parametro di tipo `M` nella sua firma.

All'interno della definizione, il parametro `T` può essere usato come un tipo. Al momento della creazione di una istanza della classe, dell'implementazione dell'interfaccia o della chiamata del metodo, è necessario specificare un tipo concreto al posto del parametro.

```
class StringList implements MappableList< String > {
  public void add(String element) {};
  public String head() { return ""; };
  public List< String > tail() {
    return Collections.emptyList();
  };
  public < M > List< M > map(Function< String, M > xform) {
    List< M > res = new ArrayList<>();
    for (String s : Arrays.asList(""))
      res.add(xform.apply(s));
    return res;
  };
}
```

Speaker notes

sopratutto con i metodi, e nelle versioni di Java più recenti, sono sempre di meno i casi in cui è realmente necessario specificare il tipo del parametro concreto; il compilatore è sempre più evoluto nel riconoscere il tipo desiderato dal contesto, quando possibile.

E' possibile esprimere alcuni vincoli sui parametri per specificare meglio il loro tipo, se necessario:

```
interface SortableList< T extends Comparable < T > > {
  void add(T element);
  T head();
  List< T > tail();
}
```

Speaker notes

in questo esempio, una implementazione di questa interfaccia può avvenire solo specificando un tipo che implementi l'interfaccia `Comparable`. In questa situazione, `extends` vale sia per le classi che per le interfacce.

E' anche possibile non esprimere vincoli sui parametri, ma solo sul tipo parametrizzato:

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

  public static void main(String[] args) {
    Collection< > cs = new ArrayList< String >();
    cs.add("hello");
    cs.add("world");
```

Speaker notes

in questo caso, non ho necessità di condizionare il contenuto della collezione. Mettendo la *wildcard* `?` esprimo al compilatore questa mia intenzione. Avere un tipo specifico non sarebbe la stessa cosa, perché il metodo sarebbe usabile esclusivamente nelle espressioni di quel tipo. Cfr: https://docs.oracle.com/javase/specs/jls/se15/html/jls-4.html#jls-4.5.1

L'uso più comune di questo costrutto avviene nella libreria standard delle *Collezioni*, ovvero tutte le implementazioni dei tipici contenitori:
liste, insiemi, code.

L'uso dei generici permette di evitare di duplicare il codice, e di semplificare l'uso delle classi contenitore fornendo al compilatore informazioni riguardo al tipo di ritorno di alcuni metodi.

Purtroppo, a causa di problematiche di compatibilità con il codice precedente, le informazioni sui tipi generici vengono cancellate al runtime e non sono più disponbili dopo la compilazione. Questo è stato un grosso cruccio per molte librerie e strumenti che cercavano di implementare algoritmi generalizzati su più tipi

Speaker notes

nonostante i loro limiti, i tipi generici permettono di realizzare in Java codice particolarmente espressivo in modo molto più semplice che, per esempio, con l'approccio dei template in C++. Con il continuo miglioramento delle capacità di inferenza del compilatore, inoltre, è stata via via ridotta la sintassi necessaria per usarli efficacemente.

VALORI PRIMITIVI

Non tutto in Java è un oggetto.

Come retaggio del suo iniziale focus verso le piattaforme *embedded*, i tipi di dato fondamentali non sono oggetti ma vengono detti *valori primitivi*.

Universo	Tipo Corto	Tipo Lungo
Interi	`byte`, `short`, `int`	`long`
Decimali	`float`	`double`
Caratteri	`char`	
Booleani	`boolean`	

Speaker notes

una stringa di caratteri non è un tipo primitivo, ma un oggetto (per quanto con una gestione molto particolare). `byte` è lungo 8 bit,`short` è lungo 16, `int` e `float` sono lunghi 32 e `long` e `double` 64; tutti hanno segno, e non c'è una versione senza segno. `char` non si può considerare di 8 bit, perché un carattere in una stringa Java è UTF-16 (https://docs.oracle.com/javase/specs/jls/se14/html/jls-3.html#jls-3.1). Un booleano ovviamente è un bit solamente.

Un valore primitivo non è un oggetto: nella sintassi non è trattato allo stesso modo ed un tipo primitivo non può essere indicato come parametro di tipo.

Ogni tipo primitivo ha quindi un corrispettivo Tipo non primitivo che può essere usato per trasportare lo stesso valore nelle situazioni in cui sia necessario.

Universo	Tipo Corto	Tipo Lungo
Interi	`Byte`, `Short`, `Integer`	`Long`
Decimali	`Float`	`Double`
Caratteri	`Char`	
Booleani	`Boolean`	

Prima di Java 5 la conversione doveva essere fatta manualmente (con grande dispendio di codice).

In Java 5 è stato introdotto il concetto di *autoboxing*: il compilatore riconosce il contesto in cui il valore primitivo (o del corrispondente tipo) viene usato e applica la trasformazione necessaria

Speaker notes

ovviamente, nei primi tempi (ed anche oggi, in situazioni estreme) questo introduceva a sorpresa delle penalità di performance in codice apparentemente innocuo. Inoltre, ci sono delle differenze semantiche che non possono essere tralasciate e che a volte portano a bug molto subdoli.

Gli array invece, sono oggetti. O meglio, come dice la specifica del linguaggio:

> *An object is*
> *a class instance or an array.*

Speaker notes

https://docs.oracle.com/javase/specs/jls/se14/html/jls-4.html#jls-4.3.1

La classe di un array discende direttamente da Object, viene creata dinamicamente quando necessario, e si comporta come un oggetto in ogni altro rispetto.

```
class Foo {
  int[] array;

  Foo() {
    array = new int[10];
    array[0] = 42;
  }
}
```

ENUMERAZIONI

Una `enum` è una particolare categoria di classi: rappresenta un tipo contenente un numero determinato di elementi, definiti alla dichiarazione.

```
enum Category {
  A, B, C, D;
}

Category cat = Category.A;
```

Una classe `enum` ha automaticamente alcuni metodi di utilità per interrogare l'insieme dei suoi valori.

Formalmente, una classe `E` di questo tipo deriva da `Enum< E >`.

Non è possibile istanziare una enumerazione, solo usare i suoi valori.

Gli elementi dell'enumerazione sono le uniche istanze della classe che li rappresenta.

Se una enumerazione è una classe interna, è implicitamente `static`.

L'enumerazione può dichiarare variabili, metodi, o implementare interfacce, e queste caratteristiche si riflettono sugli elementi in quanto istanze della classe.

Se nessun elemento dichiara una implementazione specifica, l'intera `enum` si considera `final`. Se uno o più ce l'hanno, l'intera enumerazione non lo è.

DETTAGLI

MODIFICATORI DI CLASSE

Riassumiamo i modificatori che possono essere applicati ad un classe, per elencare anche alcuni che abbiamo tralasciato:

Speaker notes

i modificatori vanno sempre prima della parola chiave `class` nella dichiarazione della classe; sono essi stessi parole chiave. Solitamente si mettono dopo eventuali modificatori di visibilità.

Parola Chiave	Significato
`abstract`	Deve essere estesa da un'implementazione.
`final`	Non può essere estesa da un'implementazione.
`strictfp`	Il codice della classe usa semantica FP

Speaker notes

In una classe `strictfp` tutte le espressioni in virgola mobile devono essere strettamente omogenee nel tipo (float o double). Altrimenti, l'implementazione ha la libertà di introdurre conversioni per gestire eventuali possibili underflow o overflow.

MODIFICATORI DI METODO

Riassumiamo i modificatori che possono essere applicati ad un metodo, per elencare anche alcuni che abbiamo tralasciato:

Speaker notes

i modificatori vanno sempre prima del tipo di ritorno nella dichiarazione del metodo; sono essi stessi parole chiave. Solitamente si mettono dopo eventuali modificatori di visibilità.

Parola Chiave	Significato
`abstract`	Deve essere implementato da una classe di estensione.
`static`	Legato alla classe e non ad una istanza.
`final`	Non può essere reimplementato da una classe di estensione.
`native`	Implementato da una libreria nativa.

Parola Chiave	Significato
`strictfp`	Il codice del metodo usa semantica FP *restrittiva*.
`synchronized`	Il metodo può essere usato da un solo thread per volta.

Speaker notes

Java ha da sempre avuto necessità di una interfaccia verso codice nativo, trattandosi di un caso d'uso comunissimo nel mondo embedded. Vedremo `synchronized` parlando di concorrenza.

RECORDS

https://www.infoq.com/news/2020/08/java16-records-instanceof/

Records and Pattern Matching for Instanceof Finalized in JDK 16

LIKE DISCUSS BOOKMARKS

AUG 12, 2020 • 3 MIN READ

by

Johan Janssen FOLLOW

Final releases of records and the new pattern matching functionality for instanceof are planned for JDK 16. With these new features, boilerplate code can be reduced and code becomes less error-prone.

Both features were already available as a preview in JDK 14 and 15. They are slightly improved and should be available in JDK 16 whose release is planned for March 2021. If all goes to plan then the new features will be available in JDK 17, the next long term support version, expected in September 2021.

RELATED CONTI

The InfoQ eMag - Innovations That Way
JUN 26, 2020

Micronaut 2.0 Enhance Support for Serverless
AUG 09, 2020

Spinto dalle esperienze di altri linguaggi (sulla JVM: Scala, Kotlin) anche in Java è stato introdotto il concetto di *named tuple*, lontano parente della *struct* di C.

In Java prende il nome di *Record*, e diventa l'occasione per importare dal mondo della programmazione funzionale ulteriori funzionalità che hanno già dimostrato in quell'ambito la loro utilità.

La sintassi del record è in realtà già disponibile da Java 14, ma solo abilitando esplicitamente la *preview* della feature. Inoltre, come annunciato, solo in Java 16 (Mar 2021) raggiungerà la piena efficacia.

Speaker notes

Il meccanismo delle preview si sta dimostrando di grande efficacia nell'ottenere feedback sulle modifiche, anche profonde che si stanno applicando al linguaggio. Il risultato è una evoluzione sempre controllata, ma molto più rapida, cadenzata dalle release semestrali. Anche se non corrisponde, spesso, ad una reale adozione, consolida l'immagine della piattaforma ed evidenzia come molto lavoro sia in corso sul suo ammodernamento.

La dichiarazione di un Record è molto semplice:

```
record Name(String firstName, String lastName){}
```

Nota importante: il record è *immutabile*.

Speaker notes

Il Record è anche implicitamente `final`, quindi non può essere astratto nè essere esteso. Un record interno ad un'altra classe è anche implicitamente `static`.

Un record ottiene automaticamente:

- membri privati con metodi di accesso pubblici
- un costruttore con tutti gli elementi del record
- `equals`, `hashCode`, `toString` generati automaticamente dallo stato del record

Speaker notes

Il costruttore così ottenuto è detto il costruttore *canonico*. Attenzione al fatto che i metodi di accesso hanno lo stesso nome dei membri del record, senza nessun prefisso `get`. Questo è contrario ad una consolidata (ma sempre meno sopportata) tradizione di Java, detta *Java Bean*.

E' possibile definire metodi in un Record, o reimplementare uno dei metodi generati automaticamente. E' possibile dichiarare un record generico o implementare una interfaccia.

In questo modo, il Record diventa una scorciatoia per definire tutte le classi che normalmente modellano valori di dominio che vengono trasportati da un posto all'altro del sistema.

Speaker notes

Il Record è una approssimazione di un *product type*.

Anche se ora come ora il Record non è molto più di una scorciatoia per realizzare una certa tipologia di Classe, in realtà è un passo intermedio per arrivare ad aggiungere in Java una sintassi di decostruzione degli oggetti, per arrivare all'implementazione di un vero e proprio *Pattern Matching*.

Riferimenti:

- Motivazione e attuale implementazione: https://www.infoq.com/articles/java-14-feature-spotlight/
- JEP359: https://openjdk.java.net/jeps/359
- Pattern Matching in Java: http://cr.openjdk.java.net/~briangoetz/amber/patte match.html