1 Lezione del 04-11-25

1.1 Java moderno

Vediamo quindi alcune funzionalità introdotte e in uso nelle versioni più recenti di Java.

Non notiamo alcuni dettagli più piccoli, come gli switch che supportano stringhe e le condizioni multiple nei blocchi `try` (usando l'operatore `|`).

1.1.1 Annotazioni

Le **annotazioni** sono uno strumento introdotto in Java 5 che si affianca ai commenti. La differenza fra commenti e annotazioni è che le annotazioni sono strutturate.

Esistono alcune annotazioni predefinite:

- *@Deprecated*, significa che un metodo è deprecato e quindi bisognerebbe cercare di non usarlo;
- *@Override*, significa che un metodo ridefinisce uno della superclasse;
- *@SuppressWarnings*, sopprime avvisi dal compilatore.

Le annotazioni possono essere definite dal programmatore, sfruttando una sintassi simile alla dichiarazione di interfacca (ma usando la chiocciola) ad esempio:

```
import java.lang.annotation.*;
[...]

// visibile a runtime (attraverso reflection)
@Retention(RetentionPolicy.RUNTIME)
// applicabile a metodi
@Target(ElementType.METHOD)
public @Interface Info {
  String info1();
  String info2();
}
```

Vediamo come si prevedono due annotazioni preliminari alla definizione di annotazione:

- *@Retention*, indica dove l'annotazione è visibile fra *SOURCE* (solo nel sorgente, scartata dal compilatore), *CLASS* (presente nel bytecode ma non visibile a runtime), *RUNTIME* (accessibile tramite reflection). Nell'esempio è `RUNTIME`;
- *@Target*, indica l'obiettivo dell'annotazione (metodi, classi, ecc...). Nell'esempio è *METHOD* (metodo).

Si può quindi usare l'annotazione dichiarata come segue:

```
class MiaClasse {
  @Info(info1 = "qualcosa", info2 = "qualcos'altro")
  public void foo() {
    // ...
  }
}
```

A questo punto si può, attraverso reflection, accedere all'annotazione di `foo()`:

```java
import java.lang.reflect.*;
[...]

Method method = MiaClasse.class.getMethod("foo");
if (method.isAnnotationPresent(Info.class)) {
    Info annotation = method.getAnnotation(Info.class);

    // stampa i campi dell'annotazione
    System.out.println(annotation.info1());
    System.out.println(annotation.info2());
}
```

1.1.2 Reflection

La **reflection** (*riflessione*) è il meccanismo che permette ai programmi Java di ispezionare altri programmi Java (incluso il programma stesso).

Questo avviene attraverso classi disponibili in `java.lang.reflect`:

- **Class**: è una classe le cui istanze rappresentano altre classi;
- **Method**: è una classe le cui istanze rappresentano metodi di altre classi;
- **Field**: è una classe le cui istanze rappresentano campi di altre classi.

Esistono poi altre classi che rappresentano costruttori, ecc...

Abbiamo già visto un esempio di reflection nella scorsa sezione riguardante le annotazioni. Vediamone un'altro:

```java
import java.lang.reflect.*;
[...]

Object cane = new Cane();
Class<?> clazz = cane.getClasse();

// da qui possiamo accedere alla struttura della classe Cane, e.g.:
Field[] fields = clazz.getDeclaredFields();
List<String> actualFields = getFieldNames(fields);
System.out.println(actualFields.get(0)); // magari "nome"
```

1.1.3 Import statici

Java permette, sempre dalla versione 5, di fare i cosiddetti **import statici**, cioè importare tutti i metodi e campi statici di una classe. Questo si fa semplicemente aggiungendo la parola chiave `static` a le direttive `import`. Una volta fatto lo static import da una classe, si possono usare gli identificatori statici di quella classe senza prefiggere il nome della classe.

1.1.4 Classi innestate

Approfondiamo il discorso delle classi innestate.

Le classi statiche possono essere innestate all'interno di altre classi. In questo caso:

- Possono accedere a tutti i campi statici della classe esterna, anche privati;
- Possono accedere a tutti i campi della classe esterna, anche privati, se ne hanno un riferimento.

Le interfacce e gli enum annidati a classi sono implicitamente static.

Le classi innestate non statiche sono dette *classi interne*.

- Ogni istanza di classe interna è associata ad una classe esterna, ed ha accesso ai riferimenti `this` e `Esterna.this` (cioè accesso alla classe esterna qualificando il `this` con il nome della classe esterna);
- Le classi interne non possono avere membri statici;
- Possono invece accedere ai campi privati della classe esterna: se non ci sono ambiguità, infatti, il riferimento this sia a sé stesse che alla classe esterna può essere omesso.

1.1.5 Argomenti variabili

Java permette di avere metodi con numero di argomenti variabile (*varargs*) usando la solita sintassi coi tre puntini. Gli argomenti risultano quindi accessibili come vettori. Ad esempio, si può avere:

```
void metodo(String...a) {
  for(String x : a) {
    // stampa tutti gli argomenti forniti
    System.out.println(x);
  }
}

[...]

// entrambi validi
metodo("Ciao");
metodo("Ciao", "mondo");
// ecc...
```

1.1.6 Try-with-resources

Una soluzione introdotta in Java 7 è quella delle **try-with-resources**, usata quando bisogna accedere a risorse, gestire delle eccezioni e quindi liberare tali risorse in presenza di eccezioni.

La sintassi storica per questo tipo di operazione era la seguente:

```
void read(String n) throws IOException {
  FileReader f = new FileReader(n);
  BufferedReader b = new BufferedReader(f);
  try {
    System.out.println(b.readLine());
  } finally {
    b.close();
    f.close();
  }
}
```

Notando che andiamo subito al blocco `finally`, in quanto per qualsiasi eccezione vogliamo comunque fare le stesse cose: chiudere `b` ed `f` (nell'ordine inverso a quello in cui vengono creati). Questo approccio può però portare a leak di risorse: poniamo ad esempio che la `b.close()` generi a sua volta un'eccezione: in questo caso la `f.close()` non viene mai eseguita ed il file non viene liberata.

Da Java 7 si può risolvere questo problema come segue

```
void read(String n) throws IOException {
  try(FileReader f = new FileReader(n); BufferedReader b = new
    BufferedReader(f)) {
    System.out.println(b.readLine());
  }
  // qui f e b sono stati chiusi!
}
```

In questo caso il meccanismo delle try-with-resources ci assicura che le risorse verranno chiuse automaticamente, e in ogni caso, nell'ordine inverso a quello in cui vengono acquisite (cioè dichiarate nella condizione del `tr`).

1.2 Tipi generici

I **tipi generici** sono simili ai template del C++. Usare i tipi raw (come ad esempio *Object*) può portare a problemi non rilevati in fase di compilazione come ad esempio conversioni invalide, ecc... Ad esempio, potremmo avere:

```
public class Casella {
  private Object o;

  public Casella(Object o) {
    this.o = o;
  }

  public Object prendi() {
    return o;
  }

  public void copia(Casella c) {
    this.o = c.o;
  }
}

[...]

Casella c1 = new Casella("ABC");
Casella c2 = new Casella(Integer.valueOf(123));
c1.copia(c2);
String s = (String) c1.prendi(); // errore! int non puo' essere convertito
    in stringa
```

La soluzione è rappresentata dai tipi generici. Un tipo generico è una classe o un'interfaccia parametrizzata in termini di tipi. Ad esempio, si può avere:

```
public class Casella<T> {
  private T t;

  public Casella(T t) {
    this.t = t;
  }

  public Object prendi() {
    return t;
  }

  public void copia(Casella<T> c) {
    this.t = c.t;
  }
}
```

In questo caso, attraverso la notazione <>, si riesce a discriminare fra classi *Casella* sulla base del tipo di dati che trasportano. Si avrà quindi rilevamento di errori come quello riportato prima, ma a tempo di compilazione:

```
Casella c1 = new Casella("ABC");
Casella c2 = new Casella(Integer.valueOf(123));
c1.copia(c2); // errore! Casella<Integer> non puo' essere convertito in
    Casella<String>
```

Per i tipi generici si può usare la wildcard <?>, già vista in 10.1.2 quando stavamo parlando della riflessione. Un riferimento con tipi generici wildcard non può essere istanziato, ma può essere usato per riferire a classi con tipi generici reali.

Ad esempio:

```
w = Casella<?>;

cs = new Casella<>("ABC");
ci = new Casella<>(Integer.valueOf(123));

w = ci;
w = cs; // permesso

String s = (String) c1.prendi();
```

Questo chiaramente ci riporta alla situazione con cui abbiamo aperto la sezione: questo non ci turba, in quanto con gli wildcard dobbiamo esplicitamente specificare che vogliamo riferirci a classi con tipi generici diversi fra di loro (quindi ci portiamo in una situazione dove siamo soggetti ad errori).

Notiamo poi che le wildcard possono essere anche più specifiche: in particolare, si possono usare wildcard con classi base per riferirsi a tipi generici classi specializzate.

1.2.1 Metodi default

I **metodi di default** servono a coprire un buco lasciato dalle interfacce. Quando si definisce un metodo all'interno di un'interfaccia, infatti, server che ogni classe che implementa tale interfaccia implementi la sua versione di tale metodo. Un metodo dichiarato come `default` all'interno di un'interfaccia, invece, permette di specificare un'implementazione predefinita, che viene quindi usata da tutte le interfacce (può eventualmente essere ridefinita).

Il problema è chiaramente la sovrapposizione che si ha data dalla definizione e implementazione in più interfacce (o anche solo in una classe base e quindi in interfacce) di una classe. In questo caso si deve usare il nome completamente qualificato del metodo (cioè riferirsi esattamente alla classe o all'interfaccia la cui versione del metodo vogliamo).