1 Lezione del 07-10-25

1.1 Classi final

All'opposto delle classi astratte ci sono le classi `final`: queste sono classi che non possono essere estese. I metodi definiti nelle classi `final` sono necessariamente `final`, cioè non sovrascrivibili.

In Java, le classi `final` sono poche: principalmente si usano metodi `final` per impedire a chi estende la classe che contiene il metodo di modificarne il comportamento.

1.2 Interfacce

Le **interfacce** sono un paradigma simile a quello delle classi astratte, ma solitamente più usate in Java.

Dal punto di vista sintattico si definiscono come una raccolta di metodi astratti: in pratica ci permettono di definire modelli di comportamento standard che più classi possono implementare. Quando una classe implementa un interfaccia, chi sa interagire con quell'interfaccia sa automaticamente interagire con quella classe. In particolare, si possono creare *riferimenti* di tipo interfaccia, ma non *oggetti* di tipo interfaccia. I riferimenti di tipo interfaccia riferiranno a un qualsiasi oggetto che implementa l'interfaccia.

Per definire un'interfaccia si usa la parola chiave `interface` e una sintassi simile a quella delle classi. Automaticamente tutti i metodi definiti sono `abstract` e `public`.

1.2.1 Implementare interfacce

Per una classe, implementare l'interfaccia (attraverso la parola chiave `implements`) significa definire *tutti* i metodi dell'interfaccia, a meno che la classe sia astratta. Una classe può implementare tutte le interfacce che vuole: solo le `extends` sono forzatamente unarie. Si può quindi avere.

```
class A extends B implements C, D, ... { /* ... */ }
```

1.3 Tipi enumerazione

I **tipi enumerazione** sono sostanzialmente interfacce che definiscono variabili costante intere (i campi sono automaticamente `public`, `static` e `final` nelle interfacce), che si possono poi usare come identificatori costanti (e semanticamente coerenti con lo scopo del tipo enumerazione).

Si può usare la parola chiave `enum` per non dover definire esplicitamente gli interi, ad esempio:

```
enum Colori {
  ROSSO,
  VERDE,
  GIALLO
}
```

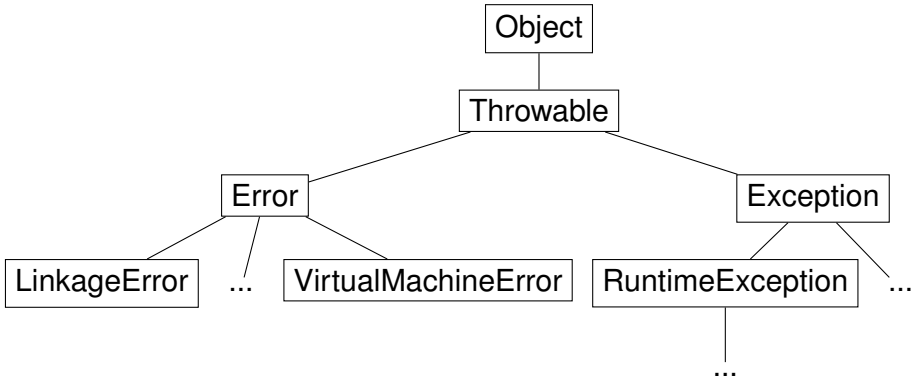
Abbiamo che gli `enum` in Java sono effettivamente definiti come una classe, la classe `Enum<E>` implementata in `java.lang.Enum`.

Sono per questo dotati di alcune funzioni di utilità, tra cui `toString()` e `name()` per ottenere i nomi dei campi dell'`enum` a tempo di esecuzione (impossibile in linguaggi come il

C++, a meno di non definire macchinose conversioni esplicite in stringhe). In particolare, la differenza fra `toString()` e `name()` è che `name()` è `final`, mentre `toString()` è ridefinibile da ogni classe `enum`.

1.4 Eccezioni

Il Java permette la gestione delle eccezioni secondo il meccanismo noto ad esempio dal C++. In particolare, la tassonomia delle classe di eccezione è la seguente:



L'oggetto Throwable può essere *lanciato*, cioè mandato al chiamante attraverso la parola chiave `throw`. Questo definirà il classico blocco `try { /* ... */} catch(Exception e) { /* gestore */ }` per implementare il gestore di eccezioni.

Esistono eccezioni di tipo **checked** e **unchecked**:

- **Checked**: fanno parte della *firma* della funzione che le lancia, e bisogna quindi che queste lo dichiarino (con la parola chiave `throws`) e il chiamante definisca un gestore (o a sua volta dichiari di poter inoltrare l'eccezione). Sono di questo tipo tutte le classi che estendono Exception tranne RuntimeException;
- **Unhecked**: rappresentano errori di programmazione o comunque non rimediabili a tempo di esecuzione, e non è necessario gestirne un gestore (ci aspettiamo che sia il gestore di default a stampare informazioni ed arrestare il programma). Sono di questo tipo le specializzazioni di:
  - **Error**: rappresentano errori interni alla JVM, o comunque specifici del linguaggio Java. Non ci aspettiamo che il progammatore possa gestirli, e quindi terminano il programma;
  - **RuntimeException**: rappresentano errori di programmazione irrecuperabili come accessi fuori buond ad array, ecc...

1.4.1 Blocchi `try-catch-finally`

Vediamo nello specifico come si trattano le chiamate a metodi che possono lanciare eccezioni, e il modo in cui si definiscono gestori di eccezioni. Un classico blocco `try-catch-finally` in Java ha l'aspetto:

```
try {
  // statement
} catch(Exception1 e1) {
  // gestisci Exception1
} catch(Exception2 e2) {
```

```
  // gestisci Exception2
}
// ...
finally {
  // blocco finally
}
```

Vediamone le componenti:

- Il blocco `try` contiene il codice che vogliamo eseguire, cioè probabilmente una chiamata a metodo che lancia Exception1 e Exception2;
- Il blocco `catch` cattura una certa eccezione (specificata negli argomenti, può catturare più eccezioni se si usa il separatore `|`). A questo punto dovrà definire codice che si occupa di gestire tale eccezione;
- Il blocco `finally` contiene codice che viene eseguito dopo che tutta l'elaborazione precedente, sia stata dalla `try` o dalle `catch`, è stata eseguita. Ha principalmente lo scopo di permetterci di fare pulizia se un'operazione su risorse (come un accesso a file) fallisce.

I blocchi `catch` vengono valutati sequenzialmente quando viene lanciata un eccezione: è inutile definire un gestore di eccezione se in seguito se ne definisce uno per un suo sottotipo, in quanto il primo verrà sempre eseguito al posto del secondo quando possibile (e in questo caso è possibile sempre).

1.4.2 Clausola `throws`

Abbiamo introdotto come la parola chiave `throws` specifica per il *contratto* della funzione che questa può lanciare una certa eccezione *checked*: approfondiamo questo aspetto. Inserire le eccezioni lanciate costringe il programmatore che usa il metodo a capire cosa potrebbe succedere, e risolvere correttamente le situazioni critiche. Proprio per questo motivo, a dispetto del polimorfismo è opportuno che le eccezioni specificate siano il più specifiche possibile.

I blocchi di inizializzazione statici non possono lanciare eccezioni *checked* in quanto non possiedono clausola `throws`: di contro i blocchi di inizializzazione non statici prendono direttamente la clausola `throws` del costruttore di classi (e infatti ne sono considerati parte).

Nel caso dell'ereditarietà, abbiamo che:

- Se si sovrascrive un metodo con clausola `throws`, bisogna specificare una causola `throws` del metoodo sovrascritto con la stessa eccezione o un suo sottotipo (a meno che il metodo sovrascritto non lanci eccezioni di qualsiasi tipo). Sostanzialmente, vogliamo assicurarci che chiunque vede soltanto il metodo base (ad esempio da un riferimento a superclasse) possa gestire l'eccezione;
- Se un metodo esiste in più interfacce, e ha clausole `throws` diverse in ognuna, dovrà implementarle tutte nella classe che implementa le interfacce.