Programmazione II

Marco Pasini

March 2021

Contents

1 Introduzione **3**
1.1 Info . . . 3
1.2 Astrazione . . . 3
1.3 Struttura dei programmi . . . 3
1.4 Packages . . . 4
1.5 Variabili . . . 4
1.6 Invocazione dei metodi . . . 4
1.7 Hello World! . . . 4

2 Lezione 2 **5**
2.1 Controllo dei tipi . . . 5
2.2 Gerarchia dei tipi . . . 5
2.3 Conversioni e Overloading . . . 5
2.4 Dispatching . . . 6

3 Lezione 3 **7**
3.1 Tipi pirmitivi . . . 7
3.2 Boxing - Unboxing . . . 7
3.3 Collezioni . . . 7
3.4 Input - Output . . . 7
3.5 Argomenti da linea di comando . . . 8

4 Lezione 4 - Astrazione procedurale **9**
4.1 Utilità . . . 9
4.2 Com'è fatta la specificazione? . . . 9
4.2.1 Clausola Requires (o precondizione) . . . 10
4.2.2 Clausola Effects (o postcondizione) . . . 10
4.2.3 Clausola Modifies (o effetti collaterali) . . . 10
4.3 Implementazione . . . 10
4.4 Come si progetta e valuta una buona astrazione procedurale . . . 10

5 Eccezioni **12**
5.1 Tipologie di eccezioni . . . 12
5.2 Definire eccezioni . . . 13
5.3 Gestire eccezioni . . . 13
5.4 Programmare con le eccezioni . . . 14

6 Lezione 6 - Astrazione sui dati **15**
6.1 Specificazione per i tipi di dati . . . 15
6.2 Metodi . . . 15
6.3 IntSet . . . 16
6.4 Poly . . . 16

7 Lezione 7 - Metodi aggiungitivi **17**
7.1 Distinzione di due oggetti con equals . . . 17
7.2 hashCode . . . 17
7.3 Rapporto tra rappresentazione ed astrazione . . . 18
7.3.1 Funzione di astrazione (AF) . . . 18
7.3.2 Invariante di rappresentazione (RI) . . . 18
7.4 Mutabilità ed effetti collaterali . . . 19

8 Lezione 8 - Verifica mantenimento astrazione **23**
8.1 Preservamento RI . . . 23
8.2 Correttezza delle operazioni . . . 23
8.3 Preservazione degli AI . . . 24
8.4 Mutabilità . . . 24
8.5 Adeguatezza . . . 24
8.6 Località e modificabilità nell'astrazione dei dati . . . 24

9 Astrazione iterazione **25**
9.1 Come viene specificata l'iterazione? . . . 25
9.2 Nested class . . . 26
9.3 Implementazione in IntSet . . . 27
9.4 Classi anonime . . . 27
9.5 Generatori Standalone . . . 27
9.6 Caching . . . 28

10 Ereditarietà **29**
10.1 Dispatching . . . 29
10.2 Come si implementano i tipi nella gerarchia? . . . 29
10.3 IR e AF nella gerarchia . . . 30
10.3.1 Esempio . . . 30

11 Sottotipi **31**
11.1 Regola dei metodi e specificazioni . . . 31

12 Polimorfismo **31**

13 Generici **31**
13.1 Collezioni . . . 31
13.2 Interfacce . . . 31

14 Laboratorio **33**

15 Metodi utili **34**
15.1 Esercitazione 6 . . . 35
15.1.1 Memoria . . . 37
15.1.2 Registri . . . 37
15.1.3 VM . . . 37
15.1.4 Modalità di accesso . . . 37
15.1.5 Opcode . . . 37

1 Introduzione

1.1 Info

> Per fare piccoli "esperimenti" con java senza creare un nuovo file ogni volta è possibile invocare `JSHELL` da terminale

Il corso non tratta il linguaggio di programmazione Java, bensì la metodologia di sviluppo di **sistemi** SW, che possono avere dimensioni non indifferenti, è importante avere programmi affidabili ed efficienti in cui il sistema deve essere descritto in modo semplice per permettere la comprensione, la modifica e la manutenzione.
Java è un WORM (Write Once Run Many) che ha risolto il problema della portabilità tra diversi sistemi operativi grazie al compilatore **JAVAC** che traduce il file sorgente `.java` in un bytecode `.class`. La macchina virtuale viene invocata con il comando `java`, che esegue il bytecode eseguendo quindi il codice.
La chiave è la modularizzazione, suddividendo il problema in sottoproblemi più piccoli e più semplici da risolvere. I sotto-problemi (o moduli) devono

- Avere lo stesso livello di dettaglio
- Essere tutti indipendenti
- Essere componibili

1.2 Astrazione

Per modularizzare si utilizza l'astrazione, che permette una maggiore comprensione di un'entità mediante la riduzione del dettaglio, separando gli aspetti rilevanti da quelli irrilevanti.
Astrazione per parametrizzazione: i dati vengono astratti e sono sostituiti con dei parametri generalizzando la procedura per poterla applicare su moltissimi valori
Astrazione per specificazione: se vengono rispettate certe condizioni iniziali (*precondizioni*), allora il programma garantisce di fornire un oggetto che soddisfa determinate proprietà (*postcondizioni*). Ad esempio, se fornisco un intero positivo, il programma mi restituirà la sua radice.

1.3 Struttura dei programmi

Java è orientato agli oggetti (una collezione di informazioni), che contengono:

- Stati: informazioni, possono essere:
  1. Mutabili: negli stati mutabili devo porre attenzione ai metodi che invoco su di essi, infatti potrebbero modificarlo, dato che Java passa sempre gli oggetti per riferimento
  2. Immutabili: come le stringhe. Quando accodo due stringhe, ne viene creata una terza il cui stato è la concatenazione degli stati delle prime due.
- Metodi: consentono di modificare o osservare lo stato

Le classi fanno "vivere" l'oggetto e vengono utilizzate per la definire una collezione di **procedure** o di **nuovi tipi**. Una classe definisce quindi un metodo per ogni procedura.
Tanti metodi → classe
. L'intestazione o prototipo di un metodo è il seguente:

$$int\ somma\ \underbrace{(int\ x, int\ y)}_{\text{Parametri formali}}$$

1.4 Packages

Java è organizzato secondo una gerarchia. I pacchetti permettono di risolvere due problemi:

1. Incapsulamento: permette all'utente di accedere ai metodi senza però accedere (ad esempio) direttamente al codice. Per questo motivo ogni classe ha una certa visibilità.
2. Naming: permette di fare riferimento ad una classe all'interno di un pacchetto indicando solamente il metodo all'interno di quel pacchetto. Se si vuole utilizzare un metodo di un altro pacchetto si deve indicare il *fully qualified name*, cioè il percorso specifico a partire dalla radice del file system. Per evitare di utilizzare il fully qualified name è possibile utilizzare `import`.

1.5 Variabili

Le. Sono indicate da:

1. Nome
2. Tipo:
   - Primitivi: *int, float, bool, etc....* Sono variabili locali che esistono all'interno del metodo, quindi vengono allocate nello stack, quando il metodo termina la variabile viene rimossa dallo stack
   - Riferimento agli oggetti: array e oggetti, sono allocati dinamicamente nello heap e devono essere inizializzate prima di utilizzarle, altrimenti il compilatore darà errore. Possono sorgere dei problemi nel momento in cui faccio un assegnamento `s=r` quando gli stati sono mutabili, infatti al modificare `r` modifico anche `s`. In Java gli oggetti e i vettori vengono automaticamente inizializzati a `NULL`, mentre gli interi viengono inzializzati a `0`.
     Quando non è più presente un riferimento per degli oggetti allocati, il garbage collector si occupa della loro eliminazione.

1.6 Invocazione dei metodi

Per invocare un metodo:

$$\underbrace{oggetto}_{\text{espressione riferimento}} .nome \underbrace{(arg_1, arg_2)}_{\text{parametri concreti (o attuali)}}$$

I parametri concerti vengono passati per valore ai parametri formali (1.3), quindi **NON viene fatta la copia**.

1.7 Hello World!

`java.lang` è il pacchetto prediletto ed è automaticamente disponibile all'interno del codice.

```
1 public class HelloWorld{
2     public static void main(String [] args{
3         int x, y;
4         x=1;
5         y=3;
6         System.out.println("Risultato: " +(x+y));
7         }
8     }
```

2 Lezione 2

2.1 Controllo dei tipi

Java è un linguaggio fortemente tipato, ciò significa che il compilatore controlla il tipo di ogni assegnamento. Il controllo dei tipi avviene in:

- Letterali, cioè quando scrivo `3` Java capisce che è un `int`, così come quando scrivo `"Pippo"` capisce che è `String`.
- Dichiarazione di variabili, quando scrivo `String s`
- Segnatura dei metodi, che consente di capire il tipo del codominio (valore restituito) e il dominio
- Mettendo insieme i punti precedenti si determina il tipo delle espressioni grazie all'**induzione strutturale**

2.2 Gerarchia dei tipi

I tipi vivono in una gerarchia in cui ci può essere un sottotipo `a` e un supertipo `b`, che verrà così indicato $a \prec b$. La gerarchia gode della proprietà transitiva e in testa alla gerarchia c'è il tipo `Object`.
La relazione di sottotipo $\prec$ deve godere del Principio di sostituzione di Liskov:

- se $a \prec b$, a deve avere tutti i metodi di b, quindi in qualsiasi punto del codice posso sostituire un'istanza di a con un'istanza di b $\rightarrow$ **a extends b**. Questa è una proprietà **sintattica** che viene controllata dal compilatore
- Devono avere lo "stesso" comportamento e preservare la **semantica**.

Il principio di sostituzione, se $a \prec b$, permette di scrivere `b=a`, dato che ad un supertipo è sempre possibile assegnare un sottotipo, dato che quest'ultimo è più specializzato (a è più speciale di b). Il controllo del tipo avviene in due momenti:

1. In fase di compilazione: il tipo **apparente** (formale) viene dedotto dal compilatore dalle informazioni presenti nelle sue dichiarazioni. Questo permette ai programmi Java che vegnono compilati di essere type safe. La type safety è garantita da tre meccanismi
   - compile-time checking
   - automatic storage management
   - array bounds checking, rende impossibile accedere all'indice 6 se l'array ha solo 3 elementi
2. In fase di esecuzione: il tipo concreto è il tipo che riceve al momento della sua creazione e può essere diverso dal tipo apparente

Il tipo apparente vale per tutte le espressioni e per tutte le sottoespressioni. Ad esempio, in `((B)a).l(...,...)`:
`~~((B)~~a).l(...,...)` $\rightarrow$ Tipo apparente di a = A
`((B)a).l(...,...)` $\rightarrow$ Tipo apparente di ((B)a) = B
`((B)a).l(...,...)` $\rightarrow$ Tipo apparente di ((B)a).l() = T (assumento che il metodo restituisca un tipo T)

2.3 Conversioni e Overloading

La conversione implicita avviene nei tipi primitivi, ad esempio un `char` può essere esteso a tipi numerici.
L'overloading consiste in un sovraccaricamento che può riguardare:

- il nome degli operatori, ad esempio + viene utilizzato sia come operatore per la somma, sia per concatenare due stringhe

- la segnatura, quindi è possibile avere due metodi con lo stesso nome ma con tipi differenti. Quando ci sono più metodi con lo stesso nome il compilatore invoca il metodo più specifico, ovvero quello che può essere invocato con l numero minore di conversioni implicite possibili. Quando il compilatore non è in grado di determinare il metodo più specifico (cioè quando è necessario fare lo stesso numero di conversioni implicite) restituirà un errore.

2.4 Dispatching

Come spiegato precedentemente, in fase di compilazione viene determinato il tipo apparente e per questo motivo potrebbe chiamare un altro metodo (a causa dell'overloading(?)) che ha un comportamento differente rispetto a quello che ci si aspetta. Il dispatching dinamico interviene proprio per risolvere questo problema, a run-time la JVM assegnerà il metodo in base al tipo concreto.
nel **casting** indico al compilatore che il tipo apparente è diverso, quindi il dispatching deve essere fatto a partire da `b` e non da `a`:

```
9  ((B)a).l(...,...);
10 boolean equals (Object other);
11 String toString();
12 s==t;  //in questo modo indica che s e t hanno lo stesso identico oggetto nello heap
13 s.equals(t); //equals indica se il contenuto di due oggetti    uguale
```

3 Lezione 3

3.1 Tipi pirmitivi

I tipi primitivi non sono dei riferimenti, quindi non sono oggetti e dato che può essere comodo invocare metodi su di essi si è introdotto il **wrapping** in cui, a ciascun primitivo, viene associato un tipo oggetto:

- costruttore `new T(x)`
- metodo statico `T.valueof(x)`, che costruisce l'oggetto del tipo primitivo

Per fare l'inverso, quindi per passare da un oggetto ad un tipo primitivo si utilizza `x=i.intValue()`.

3.2 Boxing - Unboxing

Si tratta di un metodo più moderno rispetto al creare oggetti nei due metodi precedenti in cui il compilatore si rende conto automaticamente che in quella porzione di codice è necessario un oggetto anzichè un tipo primitivo (e viceversa) ed esegue *automaticamente il wrapping*:

```
Integer i = ...;
4+i; //il compilatore esegue automaticamente il wrapping
//ed esegue 4+i.intValue
```

3.3 Collezioni

Le collezioni sono delle raccolte di oggetti omogenei e si differenziano dagli array dato che per quest'ultimi la dimensione deve essere nota al momento della compilazione. Si sono introdotti i **vector**, anche se seguono due problemi:

- L'api è deprecato, il che significa che non è consigliabile utilizzarlo dato che potrebbero essere eliminati in futuro
- il Vector è una collezione di `Object`, quindi è molto generale ed è necessario fare attenzione che nel `vector` ci siano effettivamente oggetti omogenei. Inoltre, essendo il tipo apparente sempre un `object` era sempre necessario fare il casting

Per questi motivi sono stati introdotti i generici e al posto dei generici sono state introdotte le liste:

```
    // (interfaccia) permette di definire una collezione di stringhe
    List <String> l;
    //(implementazione) permette di inserire un oggetto nella collezione
    List <String> l = new ArrayList<>();
```

3.4 Input - Output

Output: `system.out.println(...)`
Per l'input formattato si utilizza lo scanner, una classe che ha la competenza di leggere un flusso e restituire dei valori di tipo opportuno:

- Per istanziare lo scanner: `s = new Scanner(System.in)`
- Per consumarlo utilizzo l'iteratore:
  - `s.hasNextT()` → return bool
  - `s.NextT()` → return t. Nel caso della lettura delle stringe `s.Next()` restituisce ogni volta una nuova stringa, quindi se faccio un ciclo nello heap ci saranno molte stringhe, ma solamente l'ultima ha un riferimento.

3.5 Argomenti da linea di comando

Quando metto dei comandi dopo il comando `java` nel terminale, java riempie automaticamente l'array `args`. Se si tratta di stringhe però, il comando

```
21 System.out.println(args);
```

da problemi, infatti rappresenta l'array come viene "rappresentato" nella memoria di Java. Per evitare questo problema si può usare un metodo del pacchetto `java.util`:

```
22 System.out.println(Arrays.toString(args));
```

Per tradurre un intero in una stringa:

```
23 int n = Integer.parseInt(args[0]);
```

4 Lezione 4 - Astrazione procedurale

L'astrazione permette di astrarre dagli aspetti irrilevanti e di dimenticare dei dettagli, concentrandosi solamente su quelli rilevanti → si introducono delle procedure.

4.1 Utilità

L'astrazione procedurale permette di dividere il codice in procedure (cioè porzioni di codice) per mettere in evidenza l'astrazione in due punti di vista:

- Astrazione per parametrizzazione: astrazione rispetto ai dati, vengono introdotti dei parametri formali in modo che il programma funzioni indipendentemente dai valori che vengono inseriti → Viene eliminato il riferimento rispetto ai dati concreti
- Astrazione per specificazione: astrazione rispetto alla computazione, descrivendo cosa restituisce la funzione se vengono forniti certi input. Non mi importa come faccio una certa cosa, ma che **cosa** faccio. L'astrazione per specificazione viene utilizzata ogni qualvolta venga associato un commento abbastanza esplicativo che permetta ad un utente di capire che cosa fa la procedura senza guardare il codice

La **modularizzazione funzionale** rende più facile la:

1. Modificabilità, cioè la manutenzione, l'estensione/ottimizzazione e il testing dato che si hanno più parti di codice replicate nel programma. È possibile modificare un'astrazione senza modificare tutte le altre astrazioni che la utilizzano (e il loro codice)
2. Comprensione (o località): avere moduli separati permette di comprendere il funzionamento di un modulo indipendentemente dagli altri, riducendo il numero di effetti collaterali perché è possibile sapere nel dettaglio che cosa fa ogni modulo, rendendo più facile la previsione di tutti i comportamenti possibili (compresi quelli indesiderati). Grazie alla località un programma può essere anche realizzato da persone che lavorano su moduli indipendenti → un utente può implementare un'astrazione che utilizza un'astrazione di un altro utente

4.2 Com'è fatta la specificazione?

La specificazione viene fatta prima della scrittura del codice, viene prima descritto che cosa farà la funzione e quale dovrà essere l'input. La specificazione viene fatta dal punto di vista:

1. Sintattico → astrazione rispetto ai dati: nello header (o intestazione) viene indicato il tipo restituito e i parametri formali $nome({}_0P_{0,1}\, P_1,)$
2. Semantico → che cosa fa la funzione?: per spiegarlo viene utilizzato un linguaggio informale ma preciso, infatti con un linguaggio formale si rischierebbe di descrivere il *come*, che è proprio ciò che l'astrazione per specificazione vuole evitare. Verranno quindi utilizzati degli accorgimenti linguistici → 3 clausole della Liskov (o JavaDoc)

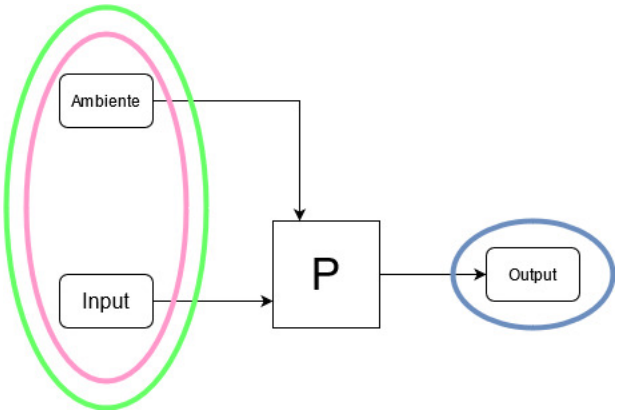


Figure 1: L'input e l'ambiente sono dati in pasto al programma che generano l'output

```
return_type pname (...)
    // REQUIRES: This clause states any constraints on use
    // MODIFIES: This clause identifies all modified inputs
    // EFFECTS: This clause defines the behavior
```

Figure 2: Le clausole della Liskov

4.2.1 Clausola Requires (o precondizione)

Indica che cosa si aspetta la funzione dall'input e dall'ambiente, **andando ad escludere alcuni valori dal dominio**. Se le precondizioni sono soddisfatte allora il funzionamento della funzione è corretto. Se questa clausola non è presente nelle specifiche si parla di **funzione totale** quando il comportamento è specificato per tutti gli input *ammissibili*. Le funzioni totali vengono utilizzate in qualcosa di pubblico quando non si è in grado di prevedere che cosa farà l'utilizzatore, in modo da assicurare che non avvengano effetti collaterali. Le procedure parziali sono tutte le procedure che non sono totali. In esse viene posto un controllo sull'input e sono più efficienti/semplici perchè non è necessario tenere in considerazioni valori particolari.
Javadoc: `@param`

4.2.2 Clausola Effects (o postcondizione)

Indica cosa la funzione restituisce in output e eventuali modifiche all'ambiente, quale è l'effetto che si ottiene. Javadoc: `@return`.

4.2.3 Clausola Modifies (o effetti collaterali)

Come la clausola Requires l'ambiente e l'input, ma indica anche le possibili modifiche dopo che la funzione è stata eseguita, ad esempio se un oggetto viene passato per riferimento è necessario indicare che cosa fa. Negli effetti collaterali indico le modifiche di una struttura anche se è l'"obbiettivo" scritto nella postcondizioni.
Javadoc: inclusa nel testo.

4.3 Implementazione

Se certe procedure sono standalone (funzionano da sole) viene utilizzato il modificatore `static`. Se una classe viene dichiarata come `public` significa che può essere da tutte le altre classi, anche al di fuori della libreria in cui è stata definita. I metodi `private` possono essere invocati solamente nella classe in cui sono definiti, permettendo così allo sviluppatore un grado di libertà maggiore, dato che non possono essere invocati all'esterno e che vengono invocati su un numero di variabili minore.
Nell'implementazione è necessario garantire che, se le precondizioni sono rispettate, la funzione abbia l'effetto dichiarato nelle postcondizioni. Se le precondizione non sono rispettate il comportamento corretto non è garantito.

4.4 Come si progetta e valuta una buona astrazione procedurale

I criteri da rispettare sono due:

1. Minimalità dei vincoli che si stanno specificando: in questo modo si dà più libertà allo sviluppatore, vincolando meno l'implementazione ed aumentando l'efficienza. Lo svantaggio è che si potrebbe non specificare qualcosa e avere dei dettagli mancanti → **sottospecificazione** (ad esempio indicando che la radice è una funzione che *approssima*).

2. La procedura deve implementare qualcosa di deterministico: sottospecificando potrebbe capitare che la funzione restituisca output differenti eseguendola sugli stessi input. Una funzione **deterministica**, su invocazioni successive con il medesimo input producano il medesimo output.

Ricorda

Non deterministico $\neq$ nondeterministico. Quello che interessa a noi è il primo, il secondo viene utilizzato nell'informatica teorica.

3. Generalità: le funzioni devono essere utili in un ampio insieme di contesti
4. Semplicità: il comportamento di una funzione deve essere ben preciso e ben specificato

5 Eccezioni

Errori $\neq$ Eccezioni. L'eccezione indica un comportamento particolare del codice di cui l'utente deve essere informato e non sempre coincide con un errore. Se voglio fare la ricerca in un array e passo un elemento che non è presente è più comodo fare un'eccezione che indicare un errore e fermare l'esecuzione del programma.

Nell'intestazione di una funzione il dominio è specificato con i parametri formali e non è possibile indicare un sottoinsieme proprio dei possibili valori sui quali la funzione è definita. Potrei indicarlo nelle precondizioni, ma con valori al di fuori di quel dominio la funzione potrebbe fare di tutto, anche compromettere altre porzioni della memoria $\rightarrow$ **funzioni parziali** = la funzione viene eseguita correttamente solo su determinati valori, per gli altri valori il comportamento non è determinato. In questo caso la funzione non è robusta (un programma robusto continua a comportarsi in modo adeguato anche se vengono rilevati degli errori) e per avere del codice che rispetti la **grateful degradation** (cioè che venga indicato che ci sono stati degli errori senza però danneggiare i dati. Se l'input non è valido allora la procedura non deve finire a quel punto o continuare il lavoro distruggendo tutto, ma finire senza modificare o non distruggere nulla) si può agire in due modi, indicando che c'è stato un problema:

1. Vengono scelti alcuni valori per indicare che è avvenuto un errore, restituendo `NULL` oppure `-1`. Non è possibile fare sempre questa cosa se ho già utilizzato tutto il dominio Θ, ad esempio la somma può essere qualsiasi valore, quindi è impossibile trovare un valore che indichi un problema. Un altro problema si verifica quando utilizzo la funzione in una somma. In questo caso potrei non accorgermi se c'è stato un errore: `y=3+sqrt(x)` restituisce un numero, come faccio a sapere se all'interno la somma è stata fatta con il valore dell'errore?

   Soluzione estendere il codominio: $\Theta \cup \{\perp\}$, come viene fatto da GO, che introduce `v, err=f(...)`.

2. Eccezioni: meccanismo dell'astrazione funzionale che indica che la funzione non è potuta proseguire ed il flusso di esecuzione si è dovuto interrompere prematuramente sollevo un'eccezione

5.1 Tipologie di eccezioni

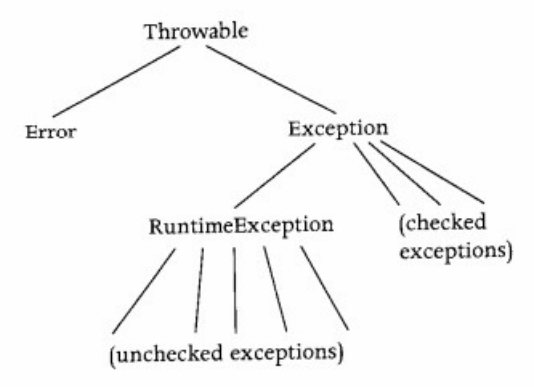


Figure 3: Gerarchie delle eccezioni

Le eccezioni sono oggetti che appartengono ad una gerarchia di classi. Il tipo `Exception` è un sottotipo sia di `Exception` e di `RunTime Exception`, entrambi sottotipo di `Throwable`. Le eccezioni unchecked sono sottotipi di `RunTime Exception` e di `Error`, mentre le checked sono sottotipi di `Exception`:

- checked (checked *a compile time*): sono gestite esplicitamente e devono seguire il *catch or specify* ossia avere nell'intestazione il *throw-catch* oppure il *throws*, altrimenti il compilatore restituisce un errore. Si usano le eccezioni checked quando i metodi sono `public` e se il *check* è oneroso.
- unchecked: non vengono esplicitamente gestite, non sono nè nell'intestazione né nel corpo del codice. Di questa categoria fanno parte gli errori e le eccezioni a runtime. Possono essere utilizzate nei metodi `private` dove si è sicuri che nessuno altro utente oltre lo sviluppatore possa chiamare la funzione con valori problematici. Vengono inoltre utilizzate quando è facile fare un check.

	Checked	Unchecked
Gestione	`catch - specify`	-
Dichiarazione	Specificazione + Throws	Specificarle solamente nella specificazione per non appesantire il codice
Quando è meglio utilizzarla?	- Recoverable Conditions: quando ci sono delle condizioni nelle quali il programma può non terminare in maniera brutale - Quando sono dovute da anomalie esterne al programma (e al programmatore) che non possono essere previste con una assoluta certezza, come un errore di input da parte dell'utilizzatore	- Usualmente per la violazione delle precondizioni - Anomalie interne al programma, come errori di implementazione - "Programming Errors": anomalie che possono essere risolte se il programmatore prestasse più attenzione
Pro (se ben utilizzate)	- Separazione dal flusso di esecuzione - Defensive Programming - Maggiore robustezza del codice - "Reminder" per il programmatore - Maggiore leggibilità	- Separazione dal flusso di esecuzione - Defensive Programming - Maggiore robustezza del codice
Contro (se si abusa)	- Non devono essere utilizzate per modificare il flusso di esecuzione (come usare le eccezioni per uscire da dei loop) - Non rendere difficile l'utilizzo del codice quando lo stiamo condividendo con altri programmatori	Non devono essere utilizzate per modificare il flusso di esecuzione (come usare le eccezioni per uscire da dei loop)

5.2 Definire eccezioni

Una procedura che può lanciare un'eccezione ha un header di questo tipo:

```
24    public static int fact (int n)
25        throw new NullPointerException, NotFoundException;
26    //posso anche creare delle nuove eccezioni
27    public static int fact (int n)
28        throw new NuovaEccezione; //NuovaEccezione.java devo definirla io
```

Il tipo eccezione ha due costruttori (overload), uno è vuoto e il secondo inizializza l'oggetto eccezione in modo da contenere una stringa.

```
29    Exception NuovaEcc1 = new NewKindOfExc(); //costruttore vuoto
30    Exception NuovaEcc2 = new NewKindOfExc("Eccezione");
```

5.3 Gestire eccezioni

La Liskov dice che tutte le eccezioni che potrebbero essere lanciate vanno specificate nell'intestazione di ogni funzione, tuttavia è un'operazione molto difficile da fare e nemmeno le api sono realizzate in questo modo.
Per gestire un'eccezione è necessario racchiudere tra `try` una porzione di codice che sospettiamo che sollevi un'eccezione e, se nella porzione di codice si verifica l'eccezione, l'esecuzione del blocco cessa istantaneamente e viene trasferita a dei blocchi `catch` che sono definiti per l'eccezione che è occorsa oppure per un supertipo dell'eccezione che è occorsa. Dopo i blocchi `catch` è anche possibile definire dei blocchi `finally` che vengono **sempre** eseguiti, sia che l'eccezione avvenga che no. Solitamente in questi blocchi si chiudono le risorse allocate facendo delle *azioni di cleanup*.
Per fare in modo che una procedura lanci un'eccezione:

```
31    if (n<0) throw new NonPositiveException("Num.fact");
```

Questa porzione di codice lancerà un oggetto del tipo `NonPositiveException`

5.4 Programmare con le eccezioni

Le eccezioni possono essere gestite **specificamente**, in cui il catch risponde alla situazione specifica, oppure **generalmente**, dove il catch fa un'azione generica, come ad esempio riavviare il programma o ripristinare una versione precedente senza risolvere nello specifico il problema che ha lanciato l'eccezione. Per gestire un'eccezione ci sono due modi:

1. Reflecting, le eccezioni vengono riportate al giusto livello di astrazione, quindi viene propagata, ovvero quando una certa procedura P solleva un'eccezione senza avere un blocco `try` in P. In questo caso Java propaga l'ecezione al chiamante di P.

2. Masking, è presente il blocco `catch` ma in qualche modo l'errore viene annullato, riprendendo il normale flusso di esecuzione → può essere molto pericoloso

Quando si usano le eccezioni?
Per rendere le funzioni parziali delle funzioni totali, in modo da eliminare i `requires` indicando negli `effects` quali sono e come si comportano le eccezioni. È importante descrivere anche come esse vadano a modificare l'ambiente, quando ad esempio viene eseguita una porzione di codice, l'ambiente viene modificato e viene lanciata l'eccezione.
Ecco alcune eccezioni utilizzate in laboratorio:

- `IllegalArgumentException`, quando ad esempio viene inserito come argomento un numero non valido

- `NullPointerExceprtion`, quando come argomento viene passato un NULL

6 Lezione 6 - Astrazione sui dati

Se si vogliono rappresentare delle informazioni che non possono essere rappresentati in modo fedele dai tipi provvisti dal linguaggio è necessario astrarre e costruire una rappresentazione semplificata → *si vuole "estendere" i tipi elementari del linguaggio.*
Si riflette sul **comportamento** e il sul **significato** delle entità, ovvero il **cosa** senza preoccuparsi del *come*.
I nuovi tipi creati dovranno incorporare l'astrazione:

- per parametrizzazione, nelle procedure si ottiene utilizzando i parametri. Il dato viene trasformato in un parametro che svolge il ruolo di dato nella computazione.
- per specificazione, che si ottiene facendo diventare le operazioni parte del tipo stesso. Allontana dal dettaglio implementativo, vengono descritti i comportamenti ma non la tecnicalità su di essi.

Ricorda

Astrazione dei dati = Oggetti + Operazioni. Grazie all'astrazione sui dati è possibile preoccuparsi successivamente di come implementare le strutture dati. Prima si ragiona suii tipi astratti e sulle operazioni che è possibile fare con, poi si implementa il tutto.

6.1 Specificazione per i tipi di dati

L'astrazione per specificazione permette di descrivere il comportamento dei tipi ed è definita dalla *sintattica* e dal *"testo"*, cioè dei commenti appropriati che descrivono che cosa fa la classe. Classe: descrizione di come sono costruiti gli oggetti, contengono la specificazione e l'implementazione. Ad esempio la classe persona definisce come è fatta la persona e che cosa può fare. Ogni classe ha dei costruttori che sono utilizzati per inizializzare nuovi oggetti di un tipo e dei metodi di istanza che permettono di accedere e modificare gli oggetti. Nella classe viene utilizzato `.this` che viene utilizzata per parlare dell'oggetto corrente.
I costruttori devono avere lo stesso nome della classe e vengono invocati. Gli **attributi** sono un elenco di variabili dichiarate con il loro tipo, così indicati T `nome` che formano la **rappresentazione**. Queste variabili sono valorizzate quando viene costruita un'istanza della classe. **Ogni oggetto fa riferimento ad uno spazio nello heap.**

```
32 visibility class dname{
33     //OVERVIEW: descrizione dell'astrazione sui dati
34
35     //attributi o campi (fields)
36     //costruttori
37     //metodi
38
39 }
```

6.2 Metodi

I metodi di istanza permettono di descrivere il comportamento dell'oggetto e sono suddivisi in quattro categorie (anche se sintatticamente non è possibile farlo, posso farlo solamente lungo una descrizione testuale):

- Metodi di creazione, che creano un oggetto o un tipo "da zero". Quasi tutti i costruttori sono metodi di creazione. (5.8.2 PDJ)
- Metodi di mutazione, alterano l'entità sulla quale vengono invocati
- Metodi di osservazione, non alterano lo stato dell'oggetto
- (Metodi di produzione/fabbricazione): producono altri oggetti dello stesso tipo, ad esempio quando creo una nuova matrice copiandone un'altra

I metodi e i costruttori hanno a che fare con gli oggetti, non con la classe, per questo non serve `STATIC`

6.3 IntSet

Insieme mutabile di `Integer` che può essere modificata e interrogata in vari modi. Il costruttore di `IntSet` inizializza un nuovo Set vuoto su cui è possibile utilizzare i metodi. Quando viene aggiungo un elemento al Set che è già presente oppure quando si cerca di rimuovere un elemento che non è presente, i metodi **non** restituiscono un'eccezione. **Ogni istanza di IntSet fa riferimento ad uno spazio nello heap.**
class IntSet

6.4 Poly

Insieme immutabile per polinomi con coefficienti interi che una volta creati non possono essere modificati (non esistono metodi di mutazione), ma su cui possono essere addizionati, sottratti e moltiplicati. Possono essere visti come degli array in cui l'*i-esimo* elemento è il coefficiente dell'*i-esimo* esponente.
class Poly

7 Lezione 7 - Metodi aggiungitivi

Come già spiegato, alla radice della gerarchia c'è `Object` e ogni oggetto della gerarchia deve essere in grado di avere tutte le competenze dei genitori seguendo il **Liskov Sustitution Principle** A volte però il comportamento del genitore non è abbastanza specifico per i sottotipi.

== viene utilizzato per confrontare i tipi primitivi, mentre il confronto tra tipi riferimento è più complicato dato che == indica se i due oggetti occupano lo stesso spazio di memoria → viene introdotto il metodo `equals` che permette di identificare due oggetti come "uguali" quando il loro *comprtamento è uguale*. È però rischioso avere una nozione di uguaglianza che dipende dal tempo, quindi si agisce in modo differente a seconda degli oggetti:

- Oggetti mutabili: Secondo Liskov tutti gli oggetti mutabili non possono essere indistinguibili perchè possono sempre mutare quindi la Liskov introduce il concetto di similarità, che però non è così diffuso nel mondo reale. Per questo motivo utilizzeremo ancora `equals` considerandoli uguali finchè non viene invocato un metodo mutazionale.
- Oggetti immutabili: posso scrivere il metodo `equals`.

7.1 Distinzione di due oggetti con equals

```
41 boolean equals (Object o)
```

Come prima cosa è necessario verificare che i due oggetti siano dello stesso tipo → `o.istanceOf` T. Poi si eseguono dei controlli sulle cose che sono ritenute uguali. Si esegue un casting T `v =` (T)`o` e si fanno i controlli `this.x  v.x`.

Nella documentazione di Object è specificato che `equals` deve soddisfare alcune proprietà:

- Riflettività
- Simmetria
- Transitività
- Consistenza: multiple invocazioni di `equals` devono sempre ritornare sempre o `true` o `false` almeno che non vengano chiamati metodi mutazionali.

7.2 hashCode

Dentro `Object` c'è un metodo simile ad `equals` chiamato `hashCode` (che quindi viene ereditato) che serve a mappare gli oggetti sugli interi → **Problema della piccionaia**, ci sono più oggetti che interi, quindi la funzione $h : O \to int$ non può essere iniettiva.

È necessario implementare `equals` in modo che l'uguaglianza di `equals` equivalga a quella di `hashCode`. Non vale però il contrario: `x.equals() == y.equals()` ⇒ `x.hashCode()==y.hashCode()` ma **non** vale `x.hashCode()==y.hashCode()x.equals() == y.equals()`.

Implementazione di hashCode:
Creo una variabile `result` e per ogni campo sommo l'hashCode:

```
42     @Override
43     public int hashCode(){
44         int result = Integer.hashCode(numerator);
45         //a result devo aggiungere tutti gli altri campi in questo modo:
46         result = 31 * result + Integer.hashCode(denominator);
47         return result;
48         // cio   return 31 *  Integer.hashCode(denominator) + Integer.hashCode(
               numerator);
49     }
```

Ricorda

Le collezioni permettono di immagazzinare oggetti omogenei per tipo. Nelle collezioni `hashCode` e `equals` devono essere implementate correttamente. Nelle collezioni è necessario aggiornare `result` ad ogni membro della collezione.

7.3 Rapporto tra rappresentazione ed astrazione

Ci sono delle situazioni in cui i rapporti tra rappresentazione (sintattica) e astrazione devono essere gestiti con attenzione, in un'istanza di IntSet il numero di elementi potrebbe non coincidere con il numero di elementi di size, oppure negli elementi ci sono più valori ripetuti → non tutti i possibili valori che vengono messi in una rappresentazioni corrispondono ad un'astrazione oppure diverse valorizzazioni nella rappresentazioni coincidono con la stessa astrazione.

```
50 IntSet
51     int []elem; //{1, 3, 5, 3}
52     int size; //size=12
53     //astrazione e rappresentazione non coincidono
```

Dei "ponti" che permettono di passare dall'implementazione verso l'astrazione sono la **funzione di astrazione** (AF) e l'**invariante di rappresentazione** (RI).

7.3.1 Funzione di astrazione (AF)

Se ho una serie di attributi $a_0, a_1, ..., a_n$ potranno assumere un valore qualsiasi dei loro domini $A_0, A_1, ..., A_n$ e facendo il prodotto cartesiano dei domini si ottiene lo spazio di tutti i valori che può assumere lo stato dell'oggetto: $A = A_0 \times A_1 \times ... \times A_n$.
La funzione di astrazione **permette di associare ad un certo possibile insieme di valori (dominio), quindi un certo A, un oggetto nell'astrazione (codominio)**: $AF : A \to \epsilon$. AF è una funzione non iniettiva, dato che dimentica i dettagli, ad esempio mappando $\{5, 7, 3\}$ e $\{5, 7, 3\}$ nella stessa entità $\{3, 5, 7\}$. Un esempio di AF è il metodo `toString`.

7.3.2 Invariante di rappresentazione (RI)

L'invariante di rappresentazione indica, per un qualunque insieme di valori, quali corrispondono oppure no ad un'entità valida nell'astrazione: $RI : A \to \{V, F\}$
Il metodo `boolean repOk()` (*idea della Liskov, non è nelle API*) è un modo per implementare l'RI che esplora i valori negli attributi e restituisce `TRUE` se i valori vanno bene (ad esempio se non ci sono duplicati quando rappresento un insieme). Non è però attiva di default nella macchina virtuale ma bisogna esplicitarla da terminale quando eseguo il programma `java -ea test`, dove `-ea` sta per `Enable Assertions`. Le **asserzioni** sono un meccanismo linguistico grazie al quale si può decorare il codice in modo che durante l'esecuzione del codice vengano controllati alcuni predicati.

Le asserzioni sono una sorta di `repOk` più granulare, sono dei predicati che si annotano nel codice per i quali la macchina virtuale che esegue i controlli di validità.

`assert e`: se l'espressione non è vera la macchina virtuale solleva l'eccezione `AsserException`.

```
54   public void hit() {
55     hits++;
56     assert repOk();
57   }
58
59   public void removeHit() {
60     if (hits > 0) hits--;
61     assert repOk();
62   }
63
```

```
64    public int hits() {
65      return hits;
66    }
67  //se repOk restituisce false il compilatore d   errore
68    boolean repOk() {
69      return hits >= 0;
70    }
```

7.4 Mutabilità ed effetti collaterali

L'implementazione può essere resa immutabile con il modificatore `final`.

Un'entità mutabile ha un'implementazione mutabile ma un'entità immutabile potrebbe avere un'implementazione mutabile. Una rappresentazione mutabile non è un problema finchè non viene esposta la rappresentazione. Immaginiamo di avere un polinomio ed una funzione che restituisca il suo grado: quando non è ancora stata chiamata il valore del grado è -1, quando viene chiamata per la prima volta il grado viene modificato e memorizzato in modo tale che possa essere restituito all'utente senza scorrere nuovamente tutta la lista. Il polinomio è immutabile ma la sua implementazione è mutabile → **effetto collaterale benevolo (B. Side Effect)**: le modifiche non sono visibili al di fuori dell'implementazione.
Ad esempio, nei numeri razionali, il metodo `equals` dovrebbe implementare `reduce`, che riduce ai minimi termini. *Questo è un esempio di BSE.* Sempre in questo esempio (numeri razionali) RI è così costituito:

- La lista dei coefficienti deve essere un riferimento non nullo
- Le coppie (coefficiente, grado) devono essere non nulle
- Le coppie (coefficiente, grado) non devo essere duplicate
- Il valore dentro il grado è -1 oppure coincide con l'esponente più grande di quelli presenti nella lista.

Ricorda

Per implementare la data abstraction è necessario ottenere il **local reasoning**, ovvero essere in grado di garantire che una classe è corretta solamente esaminando il suo codice. Il local reasoning è valido solamente se la rappresentazione degli oggetti astratti non può essere modificata al di fuori della loro implementazione. Se non c'è la il local reasoning la rappresentazione si dice **esposta**, quindi i componenti della rappresentazione sono accessibili dall'esterno della classe.

L'esposizione si verifica quando vengono dichiarate delle variabili *non private*. L'esposizione della rappresentazione è errata per:

1. Se la rappresentazione è mutabile è possibile alterarla
2. Espongo un dettaglio implementativo, quindi sono legato per sempre a mantenere quella rappresentazione, ad esempio se ho un vettore non potrò mai usare una base di dati

```
71
72  /**
73   * {@code IntSet}s are mutable, unbounded sets of integers.
74   * <p>A typical IntSet is \( S = \{x_1, \ldots, x_n \} \).
75   */
76  public class IntSet {
77
78    // Fields
79        //INVARIANTE DI RAPPRESENTAZIONE (scritta esplicitamente)
80        //els non    null
81        //els non contiene elementi nulli
```

```
82        //els non  deve contenere due interi uguali (perch     un insieme)
83
84    /** The {@link List} containing this set elements. */
85    private final List<Integer> els;
86
87    // Constructors
88
89    /**
90     * Initializes this set to be empty.
91     *
92     * <p>Builds the set \( S = \varnothing \).
93     */
94
95    public IntSet() {
96
97       //L'INVARIANTE DI RAPPRESENTAZIONE    INVARIATO AL TERMINE DEI COSTRUTTORI?
98       //ArrayList quando viene creata    vuota, non    NULL dato che new non
             restituisce null, non contiene elementi nulli e non contiene elementi
             duplicati dato che    vuota
99      els = new ArrayList<>();
100   }
101
102   /**
103    * A *copy constructor*, provided to implement {@link #clone()}.
104    *
105    * @param other the {@code IntSet} to copy from.
106    */
107    // COSTRUISCE UN INSIEME DI INTERI A PARTIRE DA UN ALTRO INSIEME DI INTERI COPIANDO
         IL COSTRUTTORE COPIA DI ARRAYLIST
108    //PER DIMOSTRARE IL PRESERVAMENTO DI RI DEVO FARE DUE IPOTESI INDUTTIVE
109    //1. ARRAY LIST SI COMPORTA IN MODO CORRETTO
110    //2. OTHER SIA STATO COSTRUITO IN MODO CORRETTO (cio  non    null, non contiene
         elementi nulli o elementi duplicati).
111
112   private IntSet(IntSet other) {
113     els = new ArrayList<Integer>(other.els);
114   }
115
116   // Methods
117 // GUARDO TUTTI I METODI CHE HANNO A CHE FARE CON L'INVARIANTE
118   /**
119    * Looks for a given element in this set.
120    *
121    * @param x the element to look for.
122    * @return the index where {@code x} appears in {@code els} if the element belongs
         to this set, or
123    *     -1
124    */
125    //NON MODIFICA ELSE
126   private int getIndex(int x) {
127     return els.indexOf(x);
128   }
129
130   /**
131    * Adds the given element to this set.
132    *
133    * <p>This method modifies the object, that is: \( S' = S \cup \{ x \} \).
134    *
135    * @param x the element to be added.
```

```
136     */
137     // MODIFICA ELSE. DEVO CONTROLLARE L'RI. IL BOXIN NON PU  PRODURRE UN NULL, QUINDI
            ELS NON    NULL, NON CI SONO DUPLICATI E ELS NON CONTIENE NULL, DATO CHE GLI
            AGGIUNGO UN ELEMENTO NON NULL
138     public void insert(int x) {
139       if (getIndex(x) < 0) els.add(x);
140           //se aggiungo un elemento che era gi  presente nella lista allora non lo
                aggiungo,
141           //cos  viene rispettato l'invariante di rappresetazione
142     }
143
144     /**
145      * Removes the given element from this set.
146      *
147      * <p>This method modifies the object, that is: \( S' = S \setminus \{ x \} \).
148      *
149      * @param x the element to be removed.
150      */
151     public void remove(int x) {
152       int i = getIndex(x);
153       if (i < 0) return;
154       int last = els.size() - 1;
155       els.set(i, els.get(last));
156       els.remove(last);
157     }
158
159     /**
160      * Tells if the given element is in this set.
161      *
162      * <p>Answers the question \( x\in S \).
163      *
164      * @param x the element to look for.
165      * @return whether the given element belongs to this set, or not.
166      */
167     public boolean isIn(int x) {
168       return getIndex(x) != -1;
169     }
170
171     /**
172      * Returns the cardinality of this set.
173      *
174      * <p>Responds with \( |S| \).
175      *
176      * @return the size of this set.
177      */
178
179     //COME VERIFICO LA CORRETTEZZA DI QUESTO METODO? Controllo operazioni AF. Se assumo
            per induzione che RI sia valido quando inizia il metodo e i parametri siano
            corretti (qui non ci sono), allora il valore restituito dal metodo (cio  l'
            implementazione) corrisponde ad una certa propriet  dell'entit  che sto
            studiando (in questo caso la cardinalit )
180      AF cosa fa? els -> {els[0], els[1],...,els.get(els.size()-1)}
181     public int size() {
182       return els.size();
183     }
184
185     /**
186      * Returns an element chosen at random from this set.
187      *
```

```
188     * @return an arbitrary element from this set.
189     * @throws EmptyException if this set is empty.
190     */
191    public int choose() throws EmptyException {
192      if (els.size() == 0) throw new EmptyException("Can't choose from an empty set");
193      return els.get(els.size() - 1);
194    }
195
196    @Override
197    public String toString() {
198      if (els.size() == 0) return "IntSet: {}";
199      String s = "IntSet: {" + els.get(0);
200      for (int i = 1; i < els.size(); i++) s = s + ", " + els.get(i);
201      return s + "}";
202    }
203
204    @Override
205    public IntSet clone() {
206      return new IntSet(this);
207    }
208  }
```

8 Lezione 8 - Verifica mantenimento astrazione

Una descrizione **formale** è una descrizione precisa di una certa realtà, caratterizzata da:

- Linguaggio (+grammatica)
- Semantica (ovvero i simboli significano qualcosa)
- "Calcolo": assiomi e teoremi

In Java l'implementazione è formale e il linguaggio di programmazione è uno, mentre l'astrazione non lo è, non esiste un unico "quadro di riferimento" → nell'astrazione si ragiona in modo informale.

Voglio prendere AF e RI per verificare la correttezza. L'implementazione è costruita in maniera plausibile?

1. Preservamento delle invarianti di rappresentazione, cioè che RI sia valido per qualunque oggetto della classe ed in qualunque momento. È importante perché tramite RI si sono costruiti i fondamenti
2. Correttezza delle operazioni: si utilizza la funzione di astrazione, che permette da passare dall'implementazione al mondo dell'astrazione
3. Invariante di astrazione (Abstraction Invariant), cioè delle proprietà che valgono solo nel mondo dell'astrazione, ad esempio un elemento di un insieme non può essere negativo. La parte di verifica dell'invariante di astrazione riguarda solamente il sottoinsieme del codice che muta lo stato dell'oggetto. È come se fosse l'analogo dell'invariante di rappresentazione per l'oggetto astratto

Nell'implementare il codice si sceglie una rappresentazione riflettendo su AF e RI.

La tecnica dimostrativa è una tecnica induttiva: se $\pi(e)$ vale, dimostro che vale anche $\pi(e+1)$. Ad esempio se $\pi(e)$ vale e $e < f \Rightarrow \pi(f)$, dove π è la dimostrazione di correttezza ed `e` e `f` sono gli oggetti.

8.1 Preservamento RI

L'invariante di rappresentazione deve essere vera ogni volta che l'oggetto viene utilizzato al di fuori della sua classe. Per verificare che l'invariante di rappresentazione sia invariato:

- È necessario che RI sia vero al termine dei costruttori
- Analizzare i metodi che alterano la rappresentazione (metodi di Mutazione + BSE + Produttori). Non bisogna controllare solo i metodi mutazionali, dato che ci potrebbero essere dei metodi con degli effetti collaterali benevoli.

8.2 Correttezza delle operazioni

Controllo molto più delicato perché le operazioni accedono anche alle informazioni dell'oggetto. Non controllo solo i metodi di mutazione ma tutti, anche quelli di osservazione.

Ricorda

Il salto tra implementazione ed astrazione è realizzato grazie a RI e AF

Le specificazioni sono scritte in termini di oggetti astratti mentre l'implementazione manipola una rappresentazione concreta. È quindi necessario guardare l'implementazione e l'entità, il salto tra uno e l'altro,è fatto da AF: $Implementazione \xrightarrow{AF} Entità$
//COME VERIFICO LA CORRETTEZZA DI QUESTO METODO? Controllo operazioni AF. Se assumo per induzione che RI sia valido quando inizia il metodo e i parametri siano corretti (qui non ci sono), allora il valore restituito dal metodo (cioè l'implementazione) corrisponde ad una certa proprietà dell'entità che sto studiando (in questo caso la cardinalità).

- Nei metodi di osservazione si fa un'ipotesi induttiva sul fatto che `this` sia corretto

- Nei metodi di mutazione si fa un'ipotesi induttiva sul fatto che siano scorretti `this` e i parametri
- Nei metodi di produzione si fa un'ipotesi induttiva sul fatto che siano scorretti `this`, i parametri e su parametri `other`, ossia altri oggetti dello stesso tipo

8.3 Preservazione degli AI

Vengono controllati solo i metodi che alterano la rappresentazione.

Un invariante di rappresentazione potrebbe essere la cardinalità di un insieme, che è sempre maggiore di zero: $|S| \geq \emptyset$. Per dimostrarlo devo farlo per induzione, verificare che valga in costruzione e nei metodi che lo alterano, come il metodo che lo altera.
Quando commento

8.4 Mutabilità

La mutabilità è una proprietà dell'implementazione ma non dell'astrazione. Possono esserci entità mutabili per loro natura ma per le quali può convenire utilizzare un'astrazione immutabile. I contenitori sono mutabili (`SET`, `LIST`,...) ma non sempre le loro implementazioni devono essere mutabili, ad esempio unendo due insiemi posso creare un nuovo insieme formato dall'unione dei due. Bisogna tenere in considerazione un giusto tradeoff tra sicurezza ed efficienza:

1. Sicurezza: con entità immutabili è più semplice ragionare sul codice, non può accadere che un metodo mutazionale comprometta l'invariante di rappresentazione. Vengono semplificate le dimostrazioni ma ci potrebbero essere molte allocazioni di memoria, come quando scrivo `s+="..."`
2. Efficienza: con entità mutabili l'efficienza è maggiore

Il punto di riferimento è l'entità che voglio modellare → si cerca di costruire l'astrazione tenendo conto che la sicurezza è più importante dell'efficienza, per questo si tende a preferire implementazioni immutabili.

8.5 Adeguatezza

Quando progettiamo un'astrazione è necessario fornire all'entità tutti i metodi per poter operare su di essa nei modi intesi. Una definizione rudimentale di adeguatezza può essere raggiunta fornendo all'astrazione:

- Costruttore
- Metodi di osservazione
- Metodi di mutazione, se l'oggetto è mutabile
- Metodi di produzione, necessari se l'oggetto è immutabile, infatti il costruttore potrebbe generare insiemi vuoti
- Il "tipo" deve essere **completamente popolato**, il che significa che è possibile ottenere/costruire ogni stato nell'astrazione tramite i metodi sopracitati.

8.6 Località e modificabilità nell'astrazione dei dati

La località richiede che la rappresentazione sia modificabile solamente attraverso l'implementazione. La modificabilità richiede che l'accesso alla rappresentazione, anche ai componenti immutabili, avvenga all'interno dell'implementazione del tipo, infatti se l'accesso avvenisse in un altro modulo non si potrebbe modificare l'implementazione senza avere ripercussioni sull'altro modulo.

9 Astrazione iterazione

Alcune classi che abbiamo costruito ci permettono di conservare entità omogenee, come `IntSet`, `Poly`, `Queue`...
Questi contenitori vengono sviluppati insieme ad una serie di comportamenti adeguati, ma ci potrebbero essere delle operazioni aggiuntive che sono plausibili eseguire sul contenitore che però non ricadono nelle loro competenze naturali. Ad esempio,in un insieme, potrei voler calcolare la somma dei suoi elementi oppure il valore massimo anche se non sono competenze naturale degli insiemi. Queste funzioni aggiuntive possono essere implementate:

- Internamente: viene creato un nuovo `IntSet` in cui vengono implementate le funzioni aggiuntive
- Esternamente: si crea una funzione che restituisce il vettore di `IntSet` ma è una soluzione critica perchè si espone la rappresentazione. Un altro modo è quello di restituire una copia del contenuto, nel caso di `IntSet` si potrebbe creare una copia del vettore e lavorare su di esso. In questo caso il problema è che l'operazione di copia è onerosa dal punto di vista temporale e computazionale.

Il metodo migliore per aggiungere queste funzioni è quello di utilizzare **un'astrazione iterazione esterna**. Il contenitore viene dotato di un **generatore** che produce gli elementi prelevandoli uno alla volta da `IntSet`.

9.1 Come viene specificata l'iterazione?

Per utilizzare l'astrazione iterazione si utilizza l'**iteratore**, una procedura speciale che restituisce un oggetto **generatore**, che tiene traccia dello stato dell'iterazione nella rappresentazione. Per implementare un iteratore è necessario scrivere il codice e implementare una classe per il suo generatore. Per ogni iteratore è necessaria un generatore (classe).

- Un iteratore è un metodo che restituisce un generatore. Un'astrazione può avere più iteratori. Le specifiche dell'iteratore definiscono il comportamento del generatore
- La classe che implementa l'iteratore (`implements iterator`) è chiamata generatore. Un generatore è un oggetto che produce gli elementi utilizzati nell'iterazione e ha i metodi `hasNext` e `next`. Questi due metodi devono essere sempre implementati dato che l'iteratore deve soddisfare il contratto dell'interfaccia

Tutti i generatori sono sottotipi dell'interfaccia `Iterator`. Per costruire un iteratore è necessario creare una nuova classe:

```
210     //generatore
211     public interface Iterator{
212         public boolean hasNext();
213         public Object next();
214     }
215
216     class SetElements implements iterator<Integer>??
```

```
217     public class IntAdder{
218     public static void main (String[] args){
219         List<Integer> lst = List.of(1,2,3,4);
220
221         int sum=0;
222         Iterator<Integer> it= lst.iterator(); //questo metodo restituisce un iteratore
223         while (it.hasNext()){
224             int x=it.next();
225             sum+=x;
226         }
227     }
228 }
```

Questa scrittura, troppo lunga, è stata sostituita con una versione più compatta e veloce, il `for each`. Per utilizzarlo è necessario indicare nella classe `implements Iterator<Integer>`.

```
229     iterator it = o.
            iterator();
230     while (it.hasNNext()){
231         E x=it.next();
232 }
```

```
233     for (E x : o) //viene estratto automaticamente l
            'iteratore da o
234     //con i due punti viene ciclato
235     List <Integer> lst = List.of(1, 2, 3, 4);
236     for (Integer x : lst){
237         sum+=x;
238         }
```

Per esseri sicuri che si sia il metodo `iterator` (nell'esempio sopra `o.iterator`) è necessario implementare l'iteratore con un'interfaccia, come ad esempio nell'`IntSet`. L'interfaccia permette di fornire ulteriori competenze ad un oggetto ed in questo caso deve avere un metodo che restituisce un iteratore che si chiami "`iterator`".

```
239     interface Iterable <E>{
240         iterator<E> iterator();
241         //il for-each funziona su tutti gli oggetti iterabili, questo permette che
                funzioni
242     }
```

Ricorda

Se gli oggetti di tipo T hanno almeno un iteratore di nome `iterator`, allora la classe T è una classe iterabile (`T implements iterable`). Iterabile = c'è esattamente un metodo di nome `iterator` che restituisce un Iterator (cioè un **generatore** per Liskov) → in questo caso T implements iterabile) → https://docs.oracle.com/javase/8/docs/api/java/lang/Iterable.html

9.2 Nested class

In un generatore è necessario rappresentare:

- Lo stato del contenitore, che deve essere privato
- Lo stato del generatore deve comunicare con lo stato del contenitore, che è però `private` e non potrebbe essere acceduto. Per risolvere il problema si utilizzano le **Nested Class**.

Le classi interne vengono utilizzate per non esporre la rappresentazione, in modo che sia possibile accedere anche alle istanze `private` della classe "superiore". A queste istanze private non è possibile accedere con un pezzo di codice che è fuori da quella classe, ma è possibile accederci tramite le classi interne. I vantaggi delle classi interne sono il **naming** e la **visibilità** (dentro essa c'è una visibilità degli attributi privati della classe esterna.
Le classi interne possono essere:

- Static nested classes: implementate per ragioni di naming. Non hanno nessuna relazione con gli oggetti della classe "superiore" quindi non è possibile riferirsi a `this` della superclasse. Il metodo statico è globale per tutta la classe, non esiste nessuna istanza `this`, allo stesso modo la classe statica vale per tutte le istanze quindi non è possibile riferirsi ad una particolare istanza → si utilizza `this`
- Inner classes (non static) →: possono essere locali oppure anonime. Noi implementeremo quelle anonime, ossia delle classi implementate dentro ad un metodo. Nelle Inner Classes per creare un oggetto nella classe interna devo scrivere `y=x.new Y()`, in questo modo y viene costruito solo a partire da una istanza di x:

```
243     Class X{
244         Class Y{
```

```
245             y=x.new Y();
246             }
247         }
```

9.3 Implementazione in IntSet

```
248     //il codice di elementGenerator, per il prinicipo di sostituziome, dovrebbe avere
            i metodi dentro iterator
249     //ma iterator    un'interfaccia, quindi ha solo le signature e non il codice.
            Prima devo implementare il codice
250     //devo implementare hasNext e Next
251
252     //QUESTO    UN ITERATORE, ossia un metodo che restituisce un generatore
253     //EFFECTS: ritorna l'iteratore
254     public Iterator<Integer> iterator(){
255         return new ElementsGenerator(this); //restituisco un iterator = restituisco l'
                istanza di una classe che implementa iterator
256     }
257
258
259     //QUESTO E UN GENERATORE (classe che implementa iterator)
260 //   static nested perch  cos  posso avvedere ai membri privati di els
261     static class ElementsGenerator implements Iterator<Integer>{
262         private IntSet set;
263         private int idx; //indice dell'ultimo elemento che ha restituito
264         //costruttore
265         ElementsGenerator (IntSet set){
266             this.set=set;
267         }
268
269         @Override
270         public boolean hasNext(){
271             return idx<set.els.size();
272         }
273
274         @Override
275         public Integer next(){
276             if (!hasNext()) throw NoSuchElementException();
277             return set.els.get(idx++);
278         }
279     }
```

9.4 Classi anonime

Le classi anonime sono delle espressioni che corrispondono ad un'istanza, è possibile crearne solamente una. Concettualmente sono identiche alle Inner Class ma sono più comode dato che viene risparmiato molto codice.

9.5 Generatori Standalone

Gli iteratori standalone vengono utilizzati quando l'iteratore è slegato dal contenitore e si vuole un iteratore che permetta di iterare su un range di interi, da `start` a `end` con un numero di passi pari a `step`. Un altro esempio è un iteratore che fa l'**avvolgimento (wrap)** di un altro iteratore, come un iteratore che itera sui numeri primi prendendo un iteratore che itera sui numeri.

9.6 Caching

In alcuni casi il contenitore non è facile da osservare e in `hasNext` l'unico modo per sapere se esiste l'elemento successivo è produrre proprio `next` (se non già esistente). Nel `next` si richiama `hasNext` e restituisce l'elemento. In questo caso è `hasNext` che modifica lo stato del generatore quindi è conveniente utilizzare un meccanismo di caching che permette di ricordare gli elementi.

10 Ereditarietà

L'ereditarietà consente di introdurre dei sottotipi in una gerarchia. La scrittura $S \prec T$ indica che S è sottotipo di T. È importante sottolineare che per il **LSP** è possibile sostituire ad ogni istanza di T il sottotipo S. **Perchè viene introdotta?**

1. Specializzare il comportamento, quando è necessario rappresentare comportamenti diversi ma con una certa logica comune
2. Estendere il comportamento, aggiungendo ulteriori metodi
3. Per contenere diverse implementazioni, ad esempio può essere necessario avere dei polinomi e dei sottotipi per la rappresentazione densa e sparsa
4. Ricalcare una disposizione della realtà ontologica, quando un elemento è dentro ad un altro come nella gerarchia delle eccezioni.

Come funziona la gerarchia in Java? Gli assegnamenti e i passaggi di parametro alla funzione sono tali per cui, per qualsiasi assegnamento di T è possibile passare come parametro qualsiasi istanza che sia sottotipo di T. Il controllo dei tipi viene eseguito:

1. In fase di compilazione viene controllato il tipo apparente, cioè il tipo dichiarato nel codice sorgente. È il tipo che il compilatore si aspetta che abbiano i riferimenti attribuiti ad una certa variabile → `Number n = new Double(3);`
2. In fase di esecuzione viene controllato il tipo concreto. Può essere determinato solamente a runtime dato che il compilatore non può prevedere che cosa l'utente vada ad inserire. Ad esempio in `number n` ci potrebbe essere un double, un integer, un number, ecc...

10.1 Dispatching

Quando viene chiamata una funzione `x.f(e0, e1,...)` non sempre è facile capire quale deve essere il codice da eseguire perchè ci potrebbero essere metodi con lo stesso nome e gli stessi argomenti in più punti della gerarchia. Java utilizza:

- La determinazione a *compile time* di quale sia il tipo apparente di `x` e poi verifica se nel tipo apparente di `x` o in uno dei sui supertipi esiste il metodo `f`. Infine determina la segnatura corretta con il **tipo più specifico**.
- A runtime è necessario determinare il metodo corretto da invocare dato che ci possono essere più metodi nei supertipi. Avviene quindi il **dispatching**, ossia viene scelto quale è il codice da eseguire sul tipo concreto.

10.2 Come si implementano i tipi nella gerarchia?

Il supertipo potrebbe avere implementazioni molto diverse rispetto ai sottotipi:

1. **Interfaccia (supertipo di grado 0)**: una collezione di metodi con le relative segnature che non contengono nessuna implementazione, non ha nessuna rappresentazione, nessun attributo, nessun codice e solo metodi pubblici. Lo scopo è quello di esibire una base comune di comportamenti. Ad esempio, in un supertipo che rappresenta l'area delle figure geometriche potrei avere un metodo che calcola l'area e nei sottotipi implementare questo metodo con la formula corretta per quella specifica figura geometrica. *Non può essere instanziata. Non contengono nessun codice ma implementano il supertipo.*
2. **Classe astratta**: alcuni metodi contengono il codice mentre altri metodi specificano solamente un **contratto** come le interfacce. Una classe astratta può avere un metodo astratto, che non sono implementati nel supertipo e devono essere specificati in un sottotipo. Non hanno oggetti e hanno costruttori che le sottoclassi possono chiamare, a differenza dell'utente che non può invece falro.

3. **Classe concreta**: è l'opposto dell'interfaccia, è raccolta di metodi e di attributi che realizzano completamente un certo insieme di comportamenti, può essere istanziata con una `new`.

La reazione di sottotipo è realizzata da una:

- Estensione: le classi astratte e concrete estendono i loro supertipi → `extends`
- Implementazione: i sottotipi delle classi implementano le interfacce → `implements`

```
class C {...}
abstract class A {...}
interface I {...}

class S extends C {...}
class S extends A {...}
class S implements I {...}
```

Nelle graffe ci sono:

Metodi

- ~~Statici~~ che sono legati alla classe e non all'istanza
- Di istanza: possono essere *"nuovi"*, ossia non presenti nel supertipo, oppure ereditati dal supertipo (quindi il loro codice è in T e non in S). I metodi sovrascritti sono già presenti in T ma è necessario riscriverli dato che S deve specializzare il comportamento.
- Metodi *final* che sono in T e non possono essere riscritti in S.

Attributi

- ~~statici~~, non vengono ereditati dalla superclasse
- Ereditati, permettono di accedere ai membri di T
- Nuovi

10.3 IR e AF nella gerarchia

È sempre necessario considerare la *funzione di astrazione* e *l'invariante di rappresentazione* quando si parla di implementazione.
IR deve riguardare i nuovi attriuti ed eventualmente rifarsi all'invariante del genitore. IR tipicamente è una congiunzione tra le proprietà dello stato della sottoclasse unito con le proprietà dello stato della superclasse. Ad esempio `repOk` prima chiama `repOk` del padre.
Il principio di sostituzione deve sempre valere, quindi la funzione di astrazione deve rimanere sostanzialmente invariata.

10.3.1 Esempio

Vorrei creare un sottotipo di IntSet che restituisca il massimo → `MaxIntSet`. È necessario introdurre un attributo `bigger` nella sottoclasse per memorizzare il massimo, che è da aggiornare per ogni elemento aggiunto o rimosso. I metodi mutazionali potrebbero infatti modificare questo attributo. Nel sottotipo posso utilizzare `super()` per riferirsi al supertipo immediato.

11 Sottotipi

Se creiamo dei sottotipi è necessario fare alcune considerazioni:

1. LSP: Il tipo concreto può sempre essere quello di un supertipo
2. Regola delle segnature: Per ogni metodo nel supertipo è necessario avere un metodo con la stessa segnatura del sottotipo. Essendo però il sottotipo più specifico potrebbe anche avere un valore restituito *più specifico*.
3. Regola dei metodi: Ciascun metodo nel sottotipo deve comportarsi come nel supertipo.
4. Regola delle proprietà: Se delle proprietà valgono nel supertipo devono valere anche nel sottotipo

11.1 Regola dei metodi e specificazioni

- Il sottotipo potrebbe **indebolire le precondizioni** e potrebbe accettare più input rispetto al supertipo (rilasso le precondizioni del supertipo): $PRE_{super} \Rightarrow PRE_{sub}$. Se le precondizioni del supertipo sono vere devono essere vere anche quelle del sottotipo. Le precondizioni del sottotipo includono quelle del supertipo.
- Il sottotipo potrebbe **rafforzare le postcondizioni**, gli oggetti del sottotipo devo essere compatibili anche con il supertipo: $PRE_{super} \wedge POST_{sub} \Rightarrow POST_{super}$

12 Polimorfismo

1. Polimorfismo per sottotipo: le espressioni hanno più forme grazie ai tipi concreti e ai tipi apparenti. Un esempio sono le `List` ed `ArrayList`. In questo modo una singola astrazione può lavorare con più tipi.
2. Polimorfismo ad hoc: realizzato grazie a *overload*(1) che permette di dare una forma o comportamento più specifici andando a cambiare il tipo di argomento e all'*overriding*(2) che permette di definire comportamenti diversi.
3. Polimorfismo parametrico: realizzato con i generici

13 Generici

13.1 Collezioni

Famiglie di strutture dati + algoritmi che sono di comodità generale nella programmazione, come `List`, `Set`, `Map`.

13.2 Interfacce

Una interfaccia definisce solamente un tipo e contiene solamente metodi astratti senza nessuna implementazione. Tutte le classi astratte sono implementate nelle classi che hanno `implements` nell'intestazione. La sintassi `<E>` indica che l'interfaccia è generica, devo indicare il tipo di oggetti contenuto nell'interfaccia in modo da permettere al compilatore di verificare a *compile time* se gli oggetti inseriti nella collezione sono corretti.
Varianti di collezioni:

1. `Collection`: radice della gerarchia, contiene segnature ma non contiene nessuna implementazione, che verrà invece definita nei sottotipi
2. `Set`: Collezione che non può contenere elementi duplicati

3. `List`: collezione ordinata
4. `Queue`:
5. `Map`: Contiene coppie chiave - valore

14 Laboratorio

Nella specifica della classe è necessario specificare se gli oggetti sono mutabili oppure no. Quando gli effetti collaterali modificano l'oggetto attuale è bene indicare `this`.
Quando viene creata una nuova classe e **non** viene dichiarato `extends`, la classe estende automaticamente la classe `Object`, che è la radice dei tipi. Tutti i figli sanno fare almeno tutto quello che sa fare il genitore quindi `toString()` il comportamento viene sovrascritto e viene automaticamente richiamato il metodo **più specifico**, ossia quello della nuova classe che stiamo creando. La sovrascrittura non avviene quando dichiariamo dei parametri: `toString(int n)`, in questo caso avviene un **overloading**. Se però prima dell'implementazione viene scritto `@Override` il compilatore si assicurerà che venga la sovrascrittura **anche se ci sono dei parametri**: **Non so se è vero**

```
287     @Override
288     public static String toString(int x){}
289     //ho dei parametri ma il compilatore effettua la sovrastrittura di toString
```

Se ho due metodi sovraccaricati ma con parametri con tipi diversi il dispatcher da priorità al tipo che gli passo. Ad esempio `elements.remove(x)`, dove x è un int, rimuoverà un elemento in posizione x. Per rimuovere invece l'intero x da intset devo trasformarlo in integer `elements.remove(Integer.valueOf(x))`

1. Nelle specifiche devo anche indicare AF e IR. AF è una funzione che nel dominio ha gli oggetti concreti, nel codominio gli oggetti astratti.
2. IR è implementato da `RepOk` mentre AF da `toString`
3. `RepOk` deve essere privata. Controllo `RepOK` all'inizio di ogni metodo oppure prima di restituire un valore nei metodi mutazionali. Quando ho un sottotipo devo prima controllare RepOk del padre: `if(!super.RepOK()) return false;`
4. `Integer.valueOf(x)` anzichè il parsing. Con il parsing potrebbero esserci problemi.
5. Quando in un costruttore metto `super()` significa che eredita il costruttore della superclasse, cioè il padre.
6. `final` indica che la varibile, una volta che è stata inizializzata non potrà più cambiare
7. Devo sempre chiedermi **come rappresento l'oggetto**. Ad esempio se sto rappresentano un numero razionale devo chiedermi: come rappresento la frazione? Riduco ai minimi termini oppure no; scrivo $\frac{5}{10}$ oppure $\frac{1}{2}$? Il segno dove lo metto?
8. in `hashCode` e in `equals` non devo fare casting o conversioni, è meglio operare direttamente sui campi dell'oggetto.
9. la comparazione tra variabili in virola mobile è una pessima idea.
10. Quando implemento `hashCode` se ho delle variabili riferimento posso utilizzare direttamente `hashCode` di quell'oggetto, mentre se ho un tipo primitivo posso utilizzare la classe involucro `Integer.hashCode`
11. Nei metodi mutazionali e di produzione è necessario commentare sempre la preservazione dell'invariante di rappresentazione. Per ipotesi induttiva devo considerare che gli oggetti rispettino l'IR quando sto per richiamare un metodo
12. il metodo `o.remove()` in una arraylist **non** diminuisce la lunghezza, ma rimane sempre la stessa.
13. Nei metodi non mutazionali non devo indicare gli effetti collaterali

15 Metodi utili

1. Iterare sulle chiavi di una mappa: costrusico un iteratore che itera sul set delle chiavi:

```
290     final List<Giocattolo> giocattoli = new ArrayList<>(inventario.keySet());
```

2. Stampare coda con moduli

```
291     for (i = 0; i < size() - 1; i++) r += elements[(head + i) % elements.length] +
            ", ";
292     r += elements[(head + i) % elements.length];
```

3. Leggere da Stdin

```
293     Scanner input = new Scanner(System.in);
294         while (input.hasNext()){
295             String line = input.nextLine();
296             String tkns[] = line.split(" ");
297         }
298         input.close();
```

4. Lunghezza di una lista

```
299        public int size (){
300     if (isEmpty()) return 0;
301     if (isFull()) return els.length;
302     return (tail - head + els.length) % els.length;
303    }
```

5. `String.valueOf(n).length()`: restituisce la lunghezza dell'array n

6. `Objects.requireNonNull` controlla che il riferimento non sia null e lancia automaticamente una eccezione NullPointerException: `Objects.requireNonNull(k, "k must be not null");`

7. Calcolare MCM:

```
304         private int gdc(int a, int b){
305         if (a<0 || b < 0) throw new IllegalArgumentException("A e B devono essere
                > 0");
306         while (b!=0){
307             int tmp = b;
308             b = a % b;
309             a = tmp;
310         }
311         return a;
312     }
```

8. `istanceof` per controllare che `Object o` sia effettivamente un'istanza di questa classe:

```
313         if (!(o instanceof SimpleMap)) return false;
314         //se mi sono accertato che sia un'istanza il casting diventa lecito
315         SimpleMap other = (SimpleMap) o;
```

9. Iterare una mappa

```
316             for (Map.Entry<Giocattolo, Integer> entry : items.entrySet())
317         //come fa getKey a prendere il toString di giocattolo?
318             str = str + entry.getValue() + " " + entry.getKey() + "\n";
```

10. Iteratore e hashcode:

```
319         @Override
320     public int hashCode(){
321         int result = 17;
322         //devo fare l'iteratore e mettere tutto in un array ordinato
323         List<Integer> inorder = new ArrayList<>();
324         Iterator<Integer> g = iterator();
325         while (g.hasNext())
326             inorder.add(g.next());
327         Collections.sort(inorder);
328         for (int i = 0; i<this.els.size(); i++)
329             result = result * 31 + Integer.hashCode(inorder.get(i));
330         return result;
331     }
332
333   @Override
334   public Iterator<Integer> iterator() {
335     return els.iterator();
336   }
```

15.1 Esercitazione 6

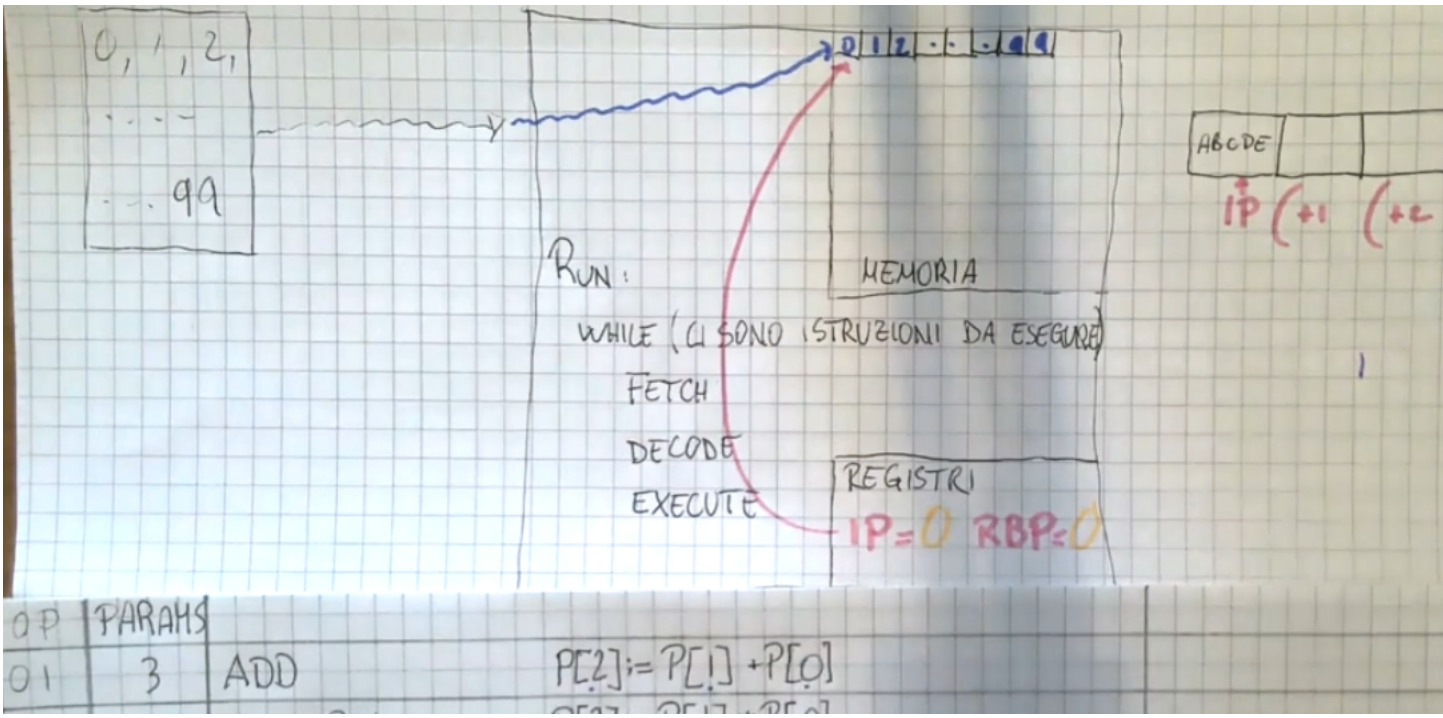


Figure 4: Virtual Machine

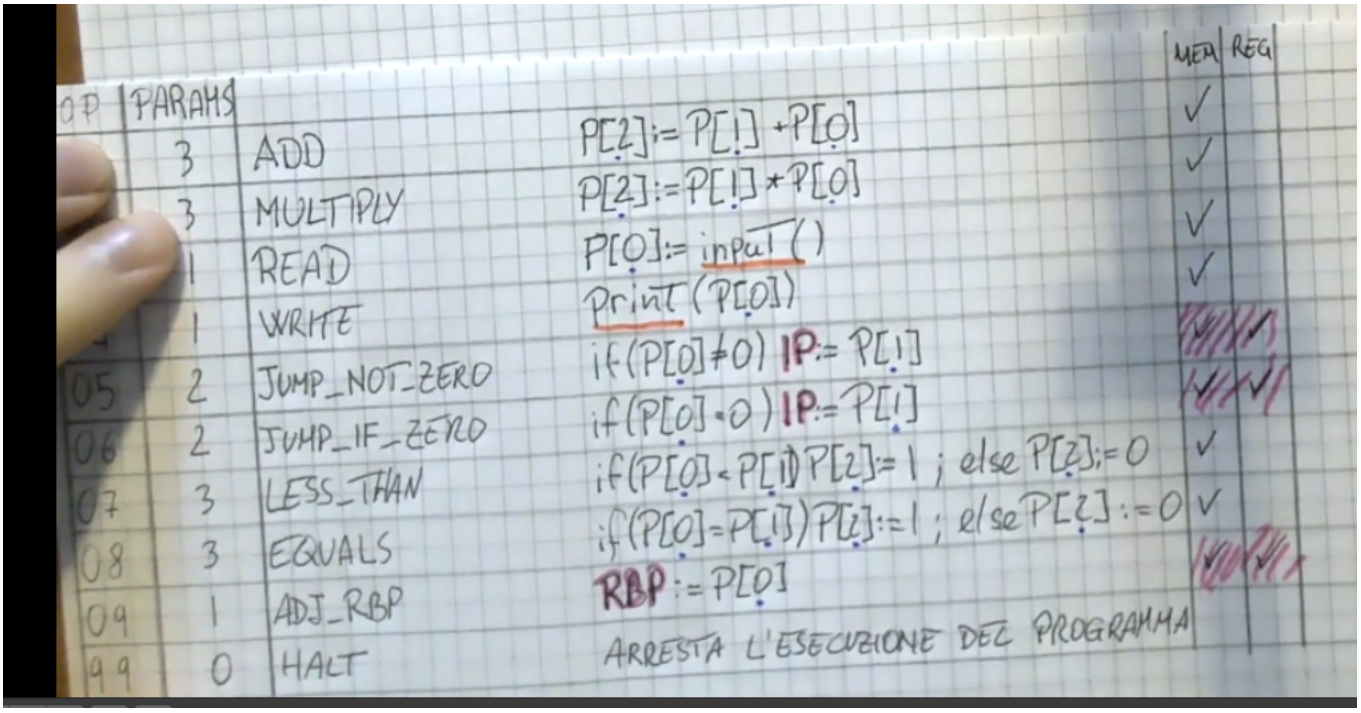
Rappresento con delle classi:

- Memoria
- Istruzioni
- Tipi di istruzioni

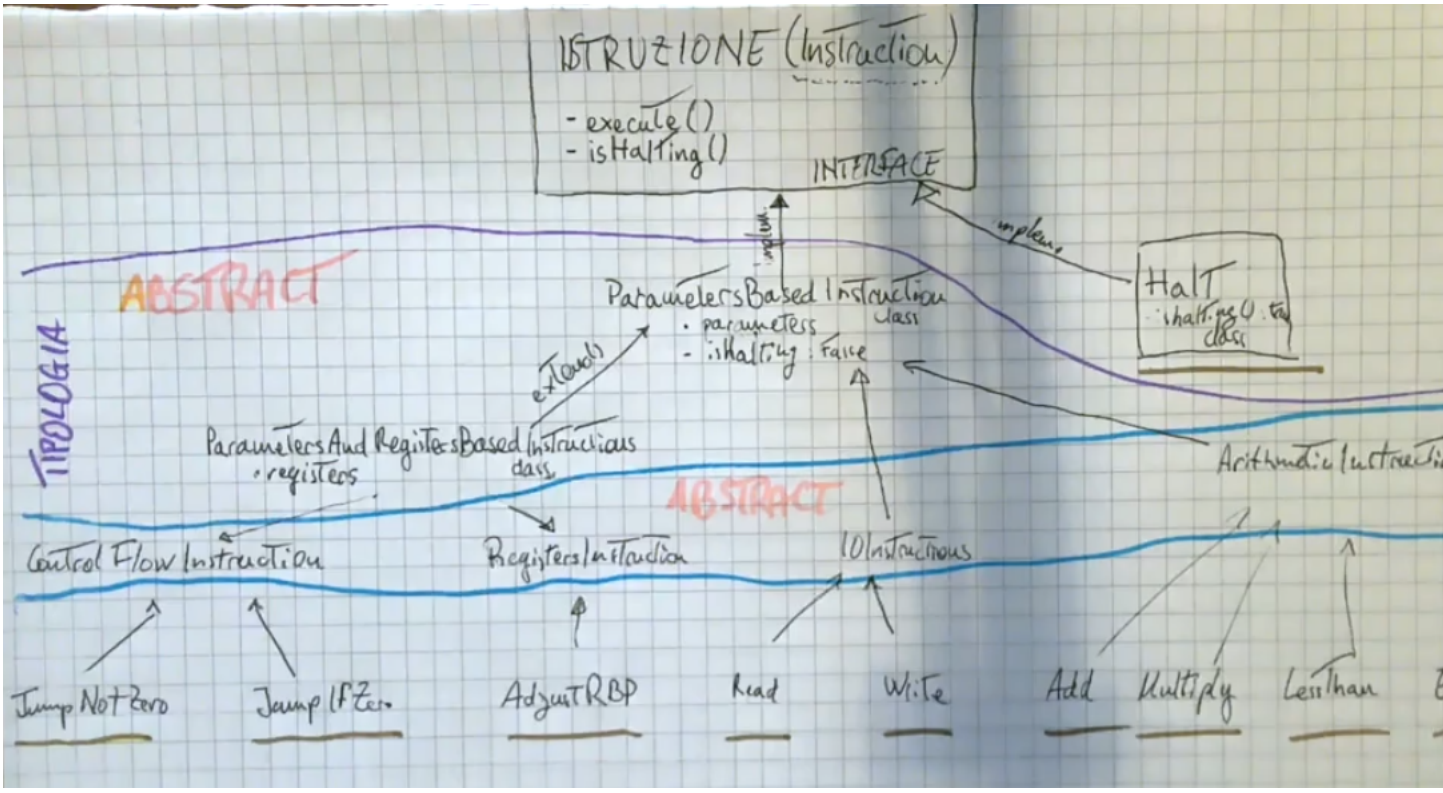
- VM
- Registri

Le informazioni del OPCODE e della modalità di accesso è possibile rappresentarle con delle costanti, in particolare dei tipi **enum**. Enum è un particolare tipo di Java che permette di definire collezioni di costanti, in modo da potermi riferire alle istanze di questa classe facendo `mode.POSITION`.

Come definisco l'istruzione? Le istruzioni possono essere in esecuzione (`exec`) oppure arrestate (`isHalting`). Per definire questi comportamenti comuni utilizzo delle **interfacce**. A seconda del tipo di operazione si dovrà lavorare sui registri e/o sulla memoria. Noto che quando opero sui registri opero anche sulla memoria.



I parametri devono essere salvati da qualche parte → uso una classe per rappresentare `ParametersBased Instruction`.
In `ParametersAndRegistersBased Instructions` devo utilizzare i registri.
Le `ControlFlowInstruction` comprendono le istruzioni di `JUmpNotZero` e `JumpIfZero`. Le uniche classi concrete sono quelle sottolineate.



15.1.1 Memoria

Le celle sono rappresentate con un attributo. Deve dare la possibilità di leggere `get ()` e scrivere `set()`. La memoria al suo interno ha la capacità di fornire delle locazioni di memoria pronte all'uso, sulle quali le istruzioni possano fare direttamente read e write.

15.1.2 Registri

Attributi: IP e RBP

15.1.3 VM

Registri e la memoria, cioè i punti precedenti, e un comando `run()` che la faccia partire.

15.1.4 Modalità di accesso

Tipo numerativo

15.1.5 Opcode

Ciascun opcode deve costruire qualcosa. Li rappresento ancora come `enum`, come attributi hanno il codice operativo e il numero di parametri.