La lezione 01 è solo di introduzione al corso

L02 - 04/10/2024

- Un programma sorgente java viene compilato da Javac che lo trasforma in Bytecode. Questo bytecode viene eseguito dalla JVM, una sorta di hardware virtuale che interpreta il bytecode e che lo esegue.
- Librerie: JDK che contiene le librerie di Java e il compilatore Javaca (JRE che contiene le librerie e la JVM)
  - Classpath per includere librerie esterne a Java

Tutto ciò può essere fatto in automatico da un Build Automation Tool: noi useremo Gradle.

Java code
- segmento imperativo: accenno ai tipi e variabili locali + flow control
- uso degli oggetti

Un programma in Java è formato da una collezzione di oggetti (classi) che sono organizzati in pacchetti.

- nel corpo di questa classe scriviamo una serie di metodi. Il metodo più importante è quello main, ossia quello da cui l'esecuzione comincia e che sta dentro una classe nel pacchetto di default.

Useremo Java 21 (LTS) senza sfruttarne tutte le caratteristiche.

[...] *manca tutta la lezione lel*

L03 - 09/10/2024

Riassunto scorsa lezione

Tipi
primitivi: char, int, long, byte, boolean, ecc
riferimento: stringa, ecc

Espressioni
. utilizzato per invocare metodi: out.println("...")
+, *, %, / (+ utilizzato anche per concatenare due stringhe)

Statement (istruzioni)
Flow control:

- sequenza (le istruzioni vengono eseguite una dopo l'altra)
- selezione: if / if-else, switch
- iterazione: for (ciclo bounded), while e do-while (cicli unbounded)

Gerarchia *(approfondiremo successivamente)*
Supertipo e sottotipi

T supertipo, S sottotipo di T, U sottotipo di S

- ci si aspetta di poter fare T t = new U (ossia di attribuire ad un supertipo un sottotipo)
- t.mangia(...) -> **dispatching**: il comportamento dell'invocazione del metodo dipenderebbe dal tipo concreto della variabile, ma il compilatore guarda il tipo apparente.

Gerarchia dei Tipi in Programmazione

In una gerarchia di tipi, possiamo avere un **supertipo** e vari **sottotipi** che estendono o implementano il supertipo. Approfondiamo il concetto di gerarchia con un esempio basato sui sottotipi.

Supertipo e Sottotipi

Consideriamo i seguenti tipi:

- T: Supertipo
- S: Sottotipo di T
- U: Sottotipo di S

Nella gerarchia, si prevede che sia possibile creare un'istanza di un sottotipo e assegnarla ad una variabile di tipo supertipo. Questo è il principio della **sostituibilità**.

Per esempio, si può fare:

```
T t = new U();
```

dove un'istanza del sottotipo U viene assegnata ad una variabile di tipo T.

Esempio: Dispatching Dinamico

Quando invochiamo un metodo su una variabile di tipo T, il comportamento dell'invocazione dipende dal **tipo concreto** dell'oggetto a cui la variabile fa riferimento, non dal suo **tipo apparente**.

Per esempio:

```
t.mangia(...);
```

Anche se il tipo apparente di t è T, il **dispatching** del metodo mangia() dipenderà dal tipo concreto dell'oggetto a cui t fa riferimento, ossia U. Il compilatore, però, guarda solo il **tipo apparente** (in questo caso, T) per verificare che il metodo esista, mentre l'esecuzione vera e propria del metodo sarà determinata a runtime dal tipo concreto dell'oggetto.

Sintesi

In questa gerarchia:

- T è il supertipo.
- S è un sottotipo di T.
- U è un sottotipo di S.

La regola della **sostituibilità** ci consente di assegnare un'istanza di un sottotipo a una variabile di tipo supertipo, e il **dispatching** dinamico garantisce che il metodo invocato sia quello del tipo concreto dell'oggetto.

String Type in Java

A **String** represents a sequence of characters and is **immutable**. Once a `String` object is created, it cannot be changed.

Affolamento di Oggetti Intermedi (Intermediate Object Overload)

When concatenating multiple strings in a sequential manner, especially inside a loop, we might encounter the problem of **intermediate object overload** in the heap. For example, let's say we want to concatenate strings $s_1, s_2, s_3, \ldots, s_n$ using a `for` loop. At each iteration, a new string is created, leading to the following allocations in the heap:

- At iteration 1:
  s_1
- At iteration 2:
  $s_1 + s_2$
- At iteration 3:
  $s_1 + s_2 + s_3$
- At iteration (n):
  $s_1 + s_2 + \cdots + s_n$

Thus, at each step, a new concatenated string is allocated in the heap, resulting in **n growing strings**. This can lead to performance inefficiencies because of the repeated allocations *(allocazione quadratica n^2)*.

StringBuilder Type

The **StringBuilder** class provides a **mutable** sequence of characters. Unlike `String`, you can modify the content of a `StringBuilder` object without creating new objects. This reduces the overhead caused by intermediate object creation during string concatenation.

Example of Concatenation

Instead of using:

```
String result = "";
for (String s : strings) {
    result += s;
}
```

You would use:

```
StringBuilder builder = new StringBuilder();
for (String s : strings) {
    builder.append(s);
}
String result = builder.toString();
```

In this way, no new intermediate objects are created at each iteration.

CharSequence

https://docs.oracle.com/en/java/javase/21/docs/api/java.base/java/lang/CharSequence.html

Both String and StringBuilder are implementations *(sottotipi)* of the common interface *(supertipo)* **CharSequence**. This interface provides two key methods:

- `length()` : Returns the length of the character sequence.
- `charAt(i)` : Returns the character at index (i).

In addition to these methods, StringBuilder adds the `append()` method, allowing efficient concatenation and modification of the character sequence.

The CharSequence interface is general and can represent both mutable and immutable sequences of characters.

Summary of the append Method

- The `append()` method in `StringBuilder` appends data (e.g., strings, numbers, or other types) to the current sequence.
- It modifies the internal state of the `StringBuilder` object directly, avoiding the creation of intermediate objects.
- This makes `StringBuilder` the preferred choice for scenarios involving extensive string concatenation, especially inside loops.

```java
public class SequenzaDiCaratteri {
    public static void main(String[] args) {
        CharSequenze seq;

        if (args.length == 0) {
            System.out.println("ci vuole almeno un elemento")
            return
        }

        for (int i = 0; i < seq.length(); i++)
            System.out.println(i + ": " + seq.charAt(i));

        String totale = ""

        [...]
    }
}
```

- length è un metodo per le string ma non per gli array

forEach

Il ciclo for ha una versione forEach

```java
// Syntax of foreach loop in Java
for (Type item : collection) {
    // code to use item
}
```

- **Type**: The data type of elements in the collection.
- **Item**: Temporary variable for each element in the collection.
- **Collection**: The array or Iterable (like ArrayList) you're looping through.

L04 - 11/10/2024

I/O

https://prog2unimi-temi-svolti.netlify.app/intro/ifdm/io

Output
print / println / printf (stringa di formato con placeholders)

Input
se leggiamo dal flusso di ingresso si usa lo `Scanner`, da linea di comando si usa `args[]`

- `Scanner.hasNext`

Dynamic Data Structure

In Java, arrays are allocated but cannot be resized once their size is defined. This means that if you create an array with a fixed size, you cannot change its length during runtime.

However, older versions of Java provided the Vector class, which allows you to store an arbitrary number of objects. Although the Vector class could grow dynamically, it has been **deprecated** due to better alternatives introduced in newer Java versions.

Key Points

- **Arrays** in Java are fixed in size and cannot be resized dynamically after their creation.
- Vector was a data structure in older Java versions that allowed dynamic resizing. It would grow as you added elements. However, it is now considered outdated because of more modern and efficient alternatives.

- The List interface is a more flexible and modern approach to handling dynamic collections of objects. Unlike arrays, List provides dynamic resizing capabilities.

List Overview

The List interface is a **contract** for classes that implement it. It defines methods for adding, removing, and accessing elements, but doesn't specify how these operations are done. There are multiple implementations of the List interface, such as:

- ArrayList: Backed by an array that grows dynamically. It is generally faster for accessing elements (random access) but slower for inserting or removing elements in the middle of the list.
- LinkedList: A doubly-linked list structure. It is faster for insertions and deletions in the middle of the list but slower for random access compared to ArrayList.

Java's List interface does **not** allow primitive types like int, char, double, etc. Instead, it only works with **objects**, so you need to use wrapper classes *(wrapper types)* to store primitive values in a List.

- Wrapper types in Java (such as Integer, Double, Boolean, etc.) are effectively objects and are allocated on the heap. Unlike primitive types (like int, double, boolean), which are stored on the stack (or in registers depending on the situation), wrapper types are reference types. This means they hold references to objects, and these objects are created in the heap.
    - For that reason, they must be used in contexts where an object is required, such as in the lists.

Here are the corresponding wrapper classes for each primitive type:

- int → Integer
- char → Character
- double → Double
- boolean → Boolean
- etc

Example
If you want to store integers in a List, you would use the Integer class, not the primitive int:

```
// This will not work, as List does not allow primitives
// List<int> list = new ArrayList<>();

// Correct way: Use Integer instead of int
List<Integer> list = new ArrayList<>();
list.add(10); // autoboxing: converts int to Integer automatically
list.add(20);

// Accessing elements
int num = list.get(0); // autounboxing: converts Integer back to int
```

In this example, **boxing** and **unboxing** are used. Java automatically converts between primitives and their wrapper classes when adding or retrieving elements from a List.

```
// Array: Fixed size, cannot be resized
int[] arr = new int[10]; // size is fixed

// Vector: Deprecated, allows dynamic resizing
Vector<Object> vec = new Vector<>(); // grows dynamically, but outdated

// List: Interface, more modern and flexible approach
List<String> list = new ArrayList<>(); // or LinkedList<>

// ArrayList: Dynamic array, fast for random access
List<Integer> arrayList = new ArrayList<>();

// LinkedList: Doubly linked list, fast for insertions/deletions
List<Integer> linkedList = new LinkedList<>();
```

The use of List and its implementations (ArrayList, LinkedList) is the modern solution for working with dynamic collections in Java.

L05 - 16/10/2024

Astrazione procedurale [3.1]

- Astrazione: non ci si concentra sui dettagli irrilevanti, ma si considerano solo quelli che sono rilevanti al problema. Le sue realizzazioni poi dovranno adottare questi dettagli rilevanti,
- Si hanno tante implementazioni e si cerca di individuare ciò che hanno in comune, ri

Astrazione per parametrizzazione (dati)

In abstraction by parameterization, we abstract from the identity of the data being used. The abstraction is defined in terms of formal parameters; the actual data are bound to these formals when the abstraction is used. Thus, the identity of the actual data is irrelevant, but the presence, number, and types of the actuals are relevant. Parameterization generalizes abstractions, making them useful in more situations. A virtue of such generalizations is that they decrease the amount of code that needs to be written and, thus, modified and maintained.

Astrazione per specificazione (cose)

In abstraction by specification, we focus on the behavior that the user can depend on and abstract from the details of implementing that behavior. Therefore, the behavior—"what" is done—is relevant, while the method of realizing that behavior—"how" it is done—is irrelevant. For example, for an isPrimeprocedure, the fact that the procedure determines whether or not its argument is a prime is relevant, but the details of how this is determined are irrelevant.

Benefits:

- **Locality** (località): the implementation of an abstraction can be read or written without needing to examine the implementations of any other abstractions.
  - Posso capire il funzionamento del mio codice senza dover leggere il codice degli altri, in quanto basta la documentazione delle API.
- **Modifiability** (modificabilità): an abstraction can be reimplemented without requiring changes to any abstractions that use it.
  Due vantaggi della modificabilità:
  - **Cambiamento del software**: renderlo efficiente, aggiungere feature, etc... ma senza violare il contratto. Se si separa la descrizione di che cosa faccia il codice dal come effettivamente questa sia stata implementata, se in futuro si decide di cambiare il codice sorgente, ma senza violare il contratto (ossia mantenendo la promessa di cosa questo codice vada poi a fare), tutti i servizi che ne dipendono non ne risentiranno.
  - **Manutenzione del software**: se si è documentato un dettaglio irrilevante, che poi magari in futuro dovrò rimuovere, se qualcuno lo ha utilizzato si ritrova a perdere quel dettaglio: si viene meno al contratto. Bisogna garantire che le promesse del contratto vengano mantenute: bisogna mettere una barriera tra ciò che il mio codice promette di fare (e che gli altri dovrano utilizzare) dal come questa venga effettivamente implementata, e va fatto bene per garantire di mantenere il codice e di cambiarlo senza causare problemi.
    - Esempio: se nella mia funzione moltiplicazione, dove uso la somma, documento anche la somma, qualcuno magari utilizza sia la mia funzione moltiplicazione che somma. Se poi in futuro decide di non utilizzare le somme iterative ma la moltiplicazione alla russa, il mio codice non è più retrocompatibile in quanto vengo meno al contratto: avevo documentato la funzione somma (che seppur serviva per la moltiplicazione magari qualcuno ha usato per altri scopi) e poi l'ho rimossa.

Astrazione per specificazione [3.2]

La specificazione precede l'implementazione. La specificazione non deve essere un racconto dell'implementazione dopo averle fatte, ma invece prima si ragiona su cosa si debba fare.

Con che linguaggio scriviamo la specificazione?

- **Linguaggio formale**: idealmente un linguaggio formale, che consente una sintassi precisa, sembrerebbe adatto.
  Il software però non sempre lavora su oggetti che presentano già un linguaggio formale; inoltre il linguaggio formale è generalmente difficile da gestire.
  Infine nella specifica, se si utilizza un linguaggio formale, inoltre si tende a replicare l'implementazione: magari si riscrive il codice nel linguaggio formale.
- **Linguaggio informale**: si usa un linguaggio informale cercando di mantenere il più possibile la chiarezza.

Nella prima parte, ossia nell'**intestazione** (header) di una procedura, si utilizza proprio Java per specificare il tipo restituito, il suo nome e l'elenco dei parametri formali sul quale opera (parametrizzazione):

Intestazione: tipo_restituito NOME (p. formali)

Specification template for procedural abstractions: [3.3]

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

1. **Requires clause**
   The requires clause states the constraints under which the abstraction is defined.
   - Needed if the procedure is **partial** (behavior not defined for some inputs).
   - Can be omitted if the procedure is **total** (behavior defined for all type-correct inputs).
   - Restrictions on a legal call are implied by the header (number and types of arguments).
2. **Modifies clause**
   The modifies clause lists the names of any inputs (including implicit inputs) that are modified by the procedure.
   - A procedure with modified inputs has a side effect.
   - Can be omitted when no inputs are modified.
   - Absence of the modifies clause means none of the inputs are modified.
3. **Effects clause**
   The effects clause describes the behavior of the procedure for all inputs not ruled out by the requires clause.
   - Defines what outputs are produced and what modifications are made to the inputs listed in the modifies clause.
   - Written assuming the requires clause is satisfied.
   - Says nothing about the procedure's behavior when the requires clause is not satisfied.

Le post-condizioni sono valide solo se le pre-condizioni sono rispettate (le pre implicano le post)

- L'assenza di pre condizioni implica il vero, ossia implica sempre le post (logica di prima ordine): queste sono funzioni totali. Quelle che presentano delle pre condizioni sono dette invece funzioni parziali.
- Le funzioni sono dette **totali** se non presentano la clausola REQUIRES, mentre se la clausola REQUIRES è $\neq \varnothing$ allora la funzione è detta **parziale**.

Esempio con la procedura per la radice quadrata:

```
double SQRT(double x)
    // REQUIRES: x ≥ 0
    // MODIFIES: nothing
    // EFFECTS: returns y   R^+ t.c. |y^2| < 10^-6
```

L06 - 18/10/2024

Javadoc

Javadoc è uno strumento utilizzato per generare documentazione API in formato HTML a partire dai commenti presenti nel codice sorgente Java. I commenti Javadoc sono scritti all'interno di blocchi speciali `(/** ... */)` e possono includere tag specifici (come `@param, @return, @throws`) per descrivere in dettaglio classi, metodi e parametri. Il risultato è una documentazione leggibile per gli sviluppatori che utilizzano o mantengono il codice.

- Aspetti di sintassi
- Aspetto pragmatico: dove vado a includere la documentazione Liskoviana (require, modifies, effects)?

Le classi che implementano dei metodi statici sono solo dei contenitori e vengono dette di utilità, e hanno bisogno di un costruttore.

- Un costruttore privato indica che il costruttore implicito è visibile solo nella classe in cui è definito

[...]

Nella documentazione bisogna essere:

- **Minimamente vincolanti**: bisogna porsi in una condizione che sia nel giusto compromesso tra ciò che viene promesso all'utente e ciò che bisogna far fare all'implementatore (bisogna quindi stare attenti a quando si stipulano i contratti).
  - **Sottodeterminati** ($\neq$ non deterministico)
- **Massimamente generali**
- Funzioni **totali vs parziali**
  - Dipende dal contesto d'uso

[...] spiegare bene la parte di sopra

L07 - 23/10/2024

Se $REQUIRES \neq \varnothing \Rightarrow D \subseteq \mathcal{D} = \mathcal{P}_1 \times \mathcal{P}_2 \cdots \quad \longrightarrow \quad$ Funzione parziale

Eccezioni (vedere anche su EJ)

La procedura P o si comporta come atteso, altrimenti può sollevare un'eccezione.
Si specifica nel contratto le circostanze intese, e viceversa si indicano le condizioni eccezionali.

Specificazione

Si specifica utilizzando la clausola `THROWS (header)`

Effects $\begin{cases} \rightarrow \cdots \\ \rightarrow \text{solleva se } \cdots \end{cases}$

- Se negli effects si specificano le eccezioni, allora non bisogna dichiarare i requires poiché questi sono stati di fatto spostati nella clausola throws.

Modifies: **Atomicità**. Nel caso si sollevi l'eccezione, negli effects si specifica che **nessuna** delle cose elencate in modifies avviene.

Quando non è possibile o utile?

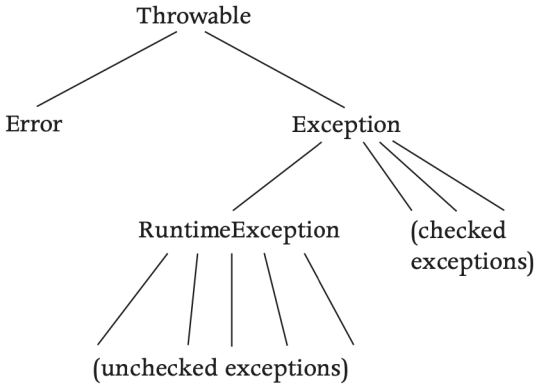
- quando si fa un metodo che compila una stringa, non è possibile prevedere in anticipo il comportamento di quel programma e se nel caso termini o meno (problema dell'arresto).
- se ci sono condizioni particolari di funzionamento (es nella ricerca dicotomica non sollevo un'eccezione nel caso di assenza, ma restituisco -1)

Java mette a disposizione molte eccezioni già definite, come per esempio l'`IllegalArgumentException`, ecc...

- di norma non c'è bisogno di crearne di nuove, cosa che comunque si può fare in quanto le eccezioni sono dei semplici oggetti

Uso

- Sollevare l'eccezione tramite `THROW newE(...)`
- Il come si gestiscono le eccezioni dipende dal loro tipo. Le eccezioni in Java si dividono in due famiglie: `checked` e `unchecked`



Se un'eccezione è **checked** allora va per forza gestita (si è obbligati dal compilatore): o la si cattura tramite **TRY-CATCH**, ma se si è in circostanze in cui questo non è possibile allora si **propaga l'eccezione** nelle throws della mia procedura (si scrive nell'HEADER).

Le eccezioni **unchecked** si propagano anche loro nelle throws delle procedure nested verso l'alto.

La Liskov dice che tutte le possibili eccezioni del metodo vanno elencate: ciò non è possibile per quelle unchecked (quelle checked si è forzati dal compilatore) poiché nessuno le elenca tutte. Se si usano librerie esterne, se tutte le eccezioni non sono documentate, noi a nostra volta non potremo documentarle

- L'approccio ragionevole è documentare tutte le eccezioni di cui il programmatore è al corrente al momento della costruzione del metodo: per esempio se si lancia una exception intenzionalmente sotto certe condizioni, va documentato quali siano queste condizioni che sollevano tale excpetion (es: IndexOutOfBoundsException)

Prassi

- (catch, se si vuole catturare l'eccezione): o la si risolve, o la si inghiotte (**masking** of exception, nel corpo di try-catch non si mette niente), oppure la si riflette (**reflect**, si riflette l'eccezione di librerie sottostanti non così come arrivano al mio metodo, ma lo traduco a chi utilizza questo metodo in maniera che sia a lui comprensibile. Si prende per esempio l'errore di basso livello e lo si trasforma a un errore sul livello di astrazione di chi utilizza il mio metodo).
    - Le eccezioni sono quindi diverse dagli errori

La Liskov dice che le eccezioni sono utili per il control flow (per esempio, nei cicli for non utilizzare i < n ma far andare il ciclo in eccezione indexOutOfBound e catturare l'eccezione):

- errore concettuale: i < n è un confronto velocissimo, catturare un'eccezione è invece estremamente inefficiente
- è estremamente da disgraziati per il debugging: se in un metodo di sorting, quando si riceve `null` si lancia un'exception del tipo "Vector must not be null" all'inizio del metodo, è estremamente facile capire dove questo accada e perché. Se invece non la lancio, e aspetto che sia Java a lanciare un'eccezione, questa verrà lanciata non più all'inizio del mio metodo, ma da qualche parte successivamente e risalirne all'origine e alle cause (nonostante sia una cosa talmente banale come il non passare un vettore `null`) potrà portare ad atteggiamenti blasfemi.

Quali eccezioni lanciare Checked e quali Unchecked:

- quando non c'è una facile evitabilità per la eccezione si fa una Checked Exception.

C4 - Exceptions

A procedural abstraction is a mapping from arguments to results, with possible modification of some of the arguments. The arguments are members of the domain of the procedure, and the results are members of its range. A procedure often makes sense only for arguments in a subset of its domain.

- For example, a procedure that computes the factorial makes sense only if its argument is positive. As another example, the search procedure can return the index of the element only if the element appears in the array.

The caller of a partial procedure must ensure that the arguments are in the permitted subset of the domain, and the implementor can ignore arguments outside this subset.

[...]

An exception mechanism provides what we want. It allows a procedure to terminate either normally, by returning a result, or exceptionally. There can be several different exceptional terminations. In Java, each exceptional termination corresponds to a different exception type. The names of the exception types are selected by the definer of the procedure to convey some information about what the problem is.

Specifications

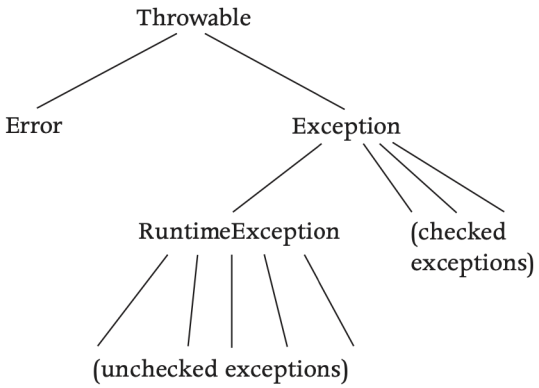
A procedure that can terminate exceptionally is indicated by having a throws clause in its header:
`throws < list_of_types >`

[...]

Java Exception Mechanism

Exeption Types

Exception types are subtypes of either Exceptionor RuntimeException, both of which are subtypes of type Throwable. The main point to note is that there are two kinds of exceptions: checked exceptions and unchecked exceptions. Unchecked exceptions are subtypes of RuntimeException; checked exceptions are subtypes of Exception but not of RuntimeException.



There are two differences in how checked and unchecked exceptions can be used in Java:

1. If a procedure might throw a checked exception, Java requires that the exception be listed in the procedure's header; otherwise, there will be a compile-time error. Unchecked exceptions need not be listed in the header.
2. If code calls a procedure that might throw a checked exception, Java requires that it handle the exception as explained in Section 4.2.4; otherwise, there will be a compile-time error. Unchecked exceptions need not be handled in the calling code.

L08 - 25/10/2024

Astrazione dati [C5]

Si cerca anche qui di passare dall'implementazione al comportamento.

Competenze fondamentali a cui pensare in fase di progettazione: capacità di **costruire**, di **osservare** (si deve poter osservare tutti i dati presenti all'interno della classe) e di **modificare** (bisogna poter modificare le informazioni già presenti)

- Astrazione per **parametrizzazione**, che ci concentra sui dati (e quindi sui parametri e sulle informazioni)
- Astrazione per **specificazione**, che serve per modellare i comportamenti (e le abilità)
  - si hanno una serie di procedure, le quali avranno i parametri formali a disposizione per lavorare più `this`, ossia la capacità di riferirsi ai dati dell'istanza. Si possono quindi specificare le cose nei metodi riferendosi ai dati.
  - Le procedure si dividono in due categorie: quelle per costruire l'oggetto dal niente (i **costruttori**) e quelle per modificare, osservare, ecc... (i **metodi**)

Bisogna costruire la propria classe quindi tenendo conto di tutti questi aspetti.

Specificazione per l'astrazione dei dati

Costruttori, fabbricatori e metodi di produzione

I **costruttori** hanno per nome del metodo esattamente il nome della classe, e possiamo averne tanti quanti ne vogliamo (ovviamente con lo stesso nome) a patto che abbiano una diversa lista dei parametri, e quindi una diversa signature.

Un **costruttore parziale** (che ha come promessa quella di costruire un oggetto) costruisce un oggetto malato

- a differenza dei metodi parziali (che rimangono comunque pericolosi) dovrebbero sempre essere evitati (da evitare per l'esame)

I **metodi di fabbricazione** sono meglio di fare un overload di costruttori. Un metodo di fabbricazione: un metodo statico, un metodo di classe, metodo di istanza che restituiscono l'istanza in quale stanno e possono essere utilizzati per rimpiazzare il costruttore nel caso di eccessivo overloading (non spiegati dalla Liskov credo, è un approccio migliore ma più recente al suo libro).

- I metodi di fabbricazione non sostituiscono il costruttore (il costruttore è l'unico che può costruire nuovi oggetti), ma lo invocano (con una New all'interno) tenendolo nascosto.

Metodi di produzione: ad un metodo passo un altro metodo e produco qualcosa.
Rimane il fatto che se questi metodi vorranno creare un nuovo oggetto dovranno comunque passare per il costruttore.

NB: ogni qual volta che un parametro dei nostri metodi è un oggetto, bisogna tener contro del fatto che questo possa essere `null`

Api eloquenti: [...]

Se una cosa è immutabile, ci deve essere un modo nel codice di raggiungere l'entità reale del mondo che ho in mente. Il contratto deve garantire di rappresentare tutte le entità tramite i suoi metodi: il contratto in questo caso si dice **pienamente popolabile**. [da vedere bene poiché è importante]

Nei successivi due esempi ci fermeremo all'aspetto contrattuale, non faremo nessuna implementazione.
Il primo esempio sarà mutabile (è pratico poter cambiare un insieme), mentre nel secondo caso si fa una scelta progettuale che consiste nel renderlo immutabile (i polinomi si lasciano perciò immutabili).

- Immutabilità e mutabilità non si riferiscono al fare delle variabili `FINAL` che quindi non si possono modificare, è un concetto che bisogna applicare ad un livello più astratto: quando si progettano i due dati, noi pensiamo agli insiemi come mutabili perché ci aggiungiamo e togliamo roba, mentre i polinomi sono univoci e se sommo due polinomi non ne cambio uno dei due ma se ne crea uno nuovo.
  - Questo è fondamentale perché in base a ciò andremo a costruire tutti i metodi relativi (per questo è un concetto che va ben oltre al semplice FINAL)

- È un qualcosa che comunque rimane verbale e che non si documenta formalmente, lo si può specificare a parole, o comunque in generale è chiaro dall'assenza di metodi di modificabilità

Specificazione di IntSet [C5.1.1]

Specificazione dei Polinomi [C5.1.2]

C5 - Data Abstraction

The new data types should incorporate abstraction both by **parameterization** and by **specification**. Abstraction by parameterization can be achieved in the same way as for procedures (by using parameters wherever it is sensible to do so). We achieve abstraction by specification by making the operations part of the type.

```
data abstraction =  objects, operations
```

We use data abstraction to avoid defining the structure immediately: we introduce the abstract type with its objects and operations. Implementations of using modules can then be designed in terms of the abstract type. Decisions about how to implement the type are made later, when all its uses are understood.

- Data abstraction is also valuable during program modification and maintenance. In this phase, data structures are particularly likely to change, either to improve performance or to accommodate changing requirements. Data abstraction limits the changes to just the implementation of the type; none of the using modules need be changed.

L09 - 30/10/2024

Le classi sono effettivamente molto utile per l'astrazione, poiché permettono di organizzare sia i dati (attributi) che le funzionalità (metodi) in un'unica entità. Questa divisione tra specificazione (cosa fa la classe) e implementazione (come la classe esegue il proprio lavoro) permette di lavorare in modo modulare e di garantire un certo grado di **incapsulamento**

[...
Data hiding is a technique of hiding internal object details, i.e., data members. It is an object-oriented programming technique. Data hiding ensures, or we can say guarantees to restrict the data access to class members. It maintains data integrity.
Data hiding means hiding the internal data within the class to prevent its direct access from outside the class.

If we talk about data encapsulation so, **Data encapsulation** hides the private methods and class data parts, whereas **Data hiding** only hides class data components. Both data hiding and data encapsulation are essential concepts of object-oriented programming. **Encapsulation** wraps up the complex data to present a simpler view to the user, whereas **Data hiding** restricts the data use to assure data security.
...]

Nel contesto di un contratto (ad esempio, un'interfaccia o una classe astratta), la barriera tra specificazione e implementazione è lieve poiché le classi che implementano il contratto si impegnano a rispettare una struttura specifica, mentre sono libere di decidere come implementare i dettagli interni. Questo approccio favorisce la flessibilità e la riusabilità del codice, oltre a facilitare il mantenimento e l'estensione dei programmi nel tempo.

Ora passeremo per gli esercizi della scorsa lezione dalla parte di specificazione alla parte di implementazione

IntSet implementazione

Ci serve una struttura dati dinamica, la Liskov utilizza i vector ma noi utilizzeremo i `List<Integer>`, ma questo è un contratto e di questo contratto utilizzeremo l'implementazione `ArrayList<...>`

Come implementiamo questi insiemi? Se ho l'insieme {1, 2}, come li metto in un array?

- [1, 2] liste ordinate senza duplicati?
- [2, 1] liste non ordinate senza duplicati?
- [1, 2, 1] liste che ammettono i duplicati? Semplifica l'inserimento, ma rende la rimozione di un elemento dal set computazionalmente complessa.

Queste non sono scelte di contratto, non vanno nel contratto. Queste scelte vanno nella **documentazione dell'implementazione** (e non in quella del contratto!!). L'importante è che ciascuna di queste implementazioni rispetti il contratto, il come poi lo faccia non è importante ai fini del contratto stesso, ma quanto più per rendere efficiente e pratico utilizzare i metodi su questi `IntSet`

Bisogna scegliere un'implementazione che ci renda pratico ed efficiente utilizzare questi insiemi.

- Noi utilizzeremo liste che non sono ordinate e che non ammettono duplicati

In una classe Java troviamo:

- **I campi/attributi (fields)** rappresentano i nomi assegnati alle variabili di istanza che contengono i dati specifici per ogni istanza della classe. Per esempio, un campo chiamato `elements` potrebbe contenere l'`ArrayList<Integer>` che rappresenta l'insieme degli interi nel nostro IntSet.

```
public class IntSet {
    // Fields
    private final List<Integer> elements

    // Constructors
    ...
    // Methods
    ...
}
```

`private` è un modificatore di accesso usato per rendere i campi (fields) e i metodi di una classe accessibili solo all'interno della classe stessa. Questo approccio è parte dell'incapsulamento, uno dei principi fondamentali della programmazione orientata agli oggetti, e protegge i dati della classe da accessi diretti e manipolazioni non autorizzate dall'esterno

- questo perché bisogna limitare ciò che viene implementato da ciò che viene promesso dal contratto: un utente esterno non deve avere accesso a ciò che sta dietro le quinte (alle mie informazioni) e che implementa ciò che lui invece deve andare a usare
  - L'utente deve usare l'interfaccia promessa dal mio contratto, ossia i metodi che io prometto funzionino come documentato: sono quindi loro che interagiscono con la mia implementazione, e non devo dare la possibilità all'utente di fare lo stesso; lui può interagire con gli attributi delle mie classi solo tramite i metodi.
- gli attributi delle classi devono quindi essere `private`, mentre i metodi e le competenze devono essere `public`, salvo rare eccezioni (non va bene non mettere niente poiché è una via intermedia, `package-private`)

Nel nostro caso, abbiamo scelto di dichiarare il campo elements come `final`, poiché non abbiamo bisogno di riassegnarlo o reinizializzarlo in futuro. Il modificatore final davanti a un campo impone che questo venga inizializzato esattamente una volta, rendendo impossibile cambiare il riferimento a quell'oggetto una volta assegnato.i

In Java, i campi `final` devono essere inizializzati esattemente una volta sola e o al momento della dichiarazione, o preferibilmente (quasi sempre) nel costruttore della classe. In questo modo, garantiamo che il campo `elements` sia associato a una specifica istanza di `List<Integer>` non appena l'oggetto `IntSet` viene creato, senza possibilità di riassegnarlo in seguito. Tuttavia, il modificatore final non impedisce di modificare il contenuto dell'oggetto List (possiamo quindi aggiungere o rimuovere elementi dalla lista), ma ne protegge solo il riferimento.

In questo modo, descriviamo chiaramente il significato di final applicato a elements e come questo contribuisca alla progettazione dell'oggetto IntSet nella classe.

```java
private int getIndex(int x) {
    return elements.indexOf(x); // Restituisce l'indice di x o -1 se non è presente
}
```

Il metodo `private int getIndex(int x)` rappresenta un metodo privato, accessibile solo all'interno della classe in cui è definito. Ecco cosa possiamo dire su di esso:

- **Modificatore private**: poiché getIndex è privato, non può essere chiamato direttamente da altre classi. Questo approccio è parte del principio di incapsulamento, in cui dettagli interni di implementazione (come il calcolo dell'indice di un elemento in questo caso) sono nascosti e non esposti all'esterno della classe. Il metodo è accessibile solo da altri metodi della classe IntSet, che possono utilizzarlo come strumento interno per realizzare altre operazioni.
- **Vantaggi dell'incapsulamento**: rendere getIndex privato significa che il dettaglio su come trovare l'indice di un elemento è nascosto e quindi flessibile per modifiche interne. Se in futuro si decidesse di cambiare la struttura dati o il modo in cui gli indici sono calcolati, nessun codice esterno alla classe IntSet verrebbe influenzato.

`getIndex` non è un metodo che io metto nel contratto per fornirlo all'utente: nel concetto di insieme non c'è l'indice di un elemento dell'insieme; l'utente non deve quindi poterlo utilizzare.

- È un metodo che serve a me poichè ho deciso di implementare `intSet` come una lista, e mi serve trovare l'indice dell'elemento per esempio quando devo rimuoverlo.
- Quindi è un metodo che serve agli altri metodi, in questo caso al metodo `removeElement`. Ed è proprio `removeElement` che è invece utilizzabile dal mio utente: lui deve avere la garanzia (dovuta dal contratto) che removeElement rimuova l'elemento dal Set, come poi lo faccia e su quali metodi miei nascosti faccia affidamento a lui non deve interessare.

Poly implementazione

Noi vogliamo rendere questi polinomi immutabili.

```java
public class Poly {
    private final int deg;
    private final int[] coeff;

    // Constructors
    public Poly() {
        degree = 0;
        coeff = new int[1];
    }

    // Methods
    ...
}
```

Mettere `final` non rende questi polinomi immutabili: semplicemente i riferimenti a questi campi final non potranno essere riassegnati, ma i valori all'interno, se questi campi sono per esempio degli array (come in questo caso), potrebbero essere modificati.

Ciò che rende veramente questi polinomi immutabili è l'assenza di metodi e di codice che vanno ad effettivamente modificarli: di conseguenza non ci devono neanche essere metodi promessi nel contratto (e definiti nella specificazione) che vadano quindi a modificarli.

- Non ci devono essere quindi metodi di modifica

```java
public Poly mul(Poly q) throws NullPointerException {
    final Poly result = new Poly(degree + q.degree);
    ...
}
```

- In Java, quando un campo è dichiarato come `private`, significa che può essere accessibile solo all'interno della stessa classe. Tuttavia, la classe Poly ha accesso ai suoi stessi campi private, quindi si possono utilizzare all'interno dei metodi della classe. Quindi limita la visibilità dei campi solo alla classe in cui sono definite.
- L'attributo `final` indica che il campo non può essere riassegnato una volta inizializzato. Tuttavia, non influisce sulla possibilità di accedere al campo; significa solo che il riferimento non può essere modificato.

Questo è il motivo per qui si può fare `q.degree`

Il modificatore private controlla solo la visibilità degli attributi e non influisce sull'immutabilità. Un campo può essere private e comunque modificabile da metodi all'interno della classe. Per rendere la classe veramente immutabile, è fondamentale garantire che non esistano metodi che alterino il suo stato, indipendentemente dal livello di visibilità dei campi.

[gli attributi costituiscono lo stato della classe] to check

Attributo qualificato vs non qualificato [...]

Spiegazione di GPT:

- Un **attributo qualificato** è un attributo che viene accesso specificando il contesto o la classe a cui appartiene. Ciò è particolarmente comune quando si lavora con classi o oggetti e aiuta a evitare ambiguità.
  - **Accesso a un attributo di un oggetto**

```java
public class Car {
    private String color;

    public Car(String color) {
        this.color = color;
    }

    public String getColor() {
        return color;
    }
}

public class Main {
    public static void main(String[] args) {
        Car myCar = new Car("red");
        String carColor = myCar.getColor(); // 'color' è qualificato dall'oggetto 'myCar'
    }
}
```

  - **Accesso a un attributo di classe**

```java
public class MathUtils {
    public static final double PI = 3.14159;
}

public class Main {
    public static void main(String[] args) {
        double area = MathUtils.PI * radius * radius; // 'PI' è qualificato dalla classe 'MathUtils'
    }
}
```

- Un **attributo non qualificato** è un attributo che viene accesso senza specificare il contesto o la classe a cui appartiene. Questo può avvenire quando l'attributo è presente nello stesso contesto in cui viene utilizzato, o quando viene importato in modo da evitare la necessità di qualificazione.
  - **Accesso a un attributo di un oggetto**

```java
public class Person {
    private String name;

    public Person(String name) {
        this.name = name;
    }

    public void printName() {
        System.out.println(name); // 'name' è non qualificato
    }
}
```

  - **Accesso a un attributo di classe**

```java
public class Main {
    public static void main(String[] args) {
        int result = 5 + 10; // Non c'è qualificazione necessaria per le variabili locali
    }
}
```

L10 - 06/11/2024

Additional Methods (cap 5.4 PDJ)

Fino ad adesso abbiamo ignorato dei metodi addizionali che tutti gli oggetti possiedono: questi metodi sono ereditati da `Object`. Tutte le classi sono dei sottotipi di `Object` e di conseguenza dovranno mettere a disposizione tutti i suoi metodi.

- Ereditare i metodi di Object va bene se l'implementazione ereditata è corretta per il nuovo tipo, altrimenti la classe deve predisporre la propria implementazione.

Alcuni metodi sono `equals`, `clone` e `toString` (non tutti andrebbero più utilizzati)

toString

`toString` restituisce una stringa che mostra il tipo e lo stato corrente del suo oggetto. Tutti i tipi devono implementare toString da loro.

- Questo perché di default tiene traccia solo del pacchetto, della classe e di altre informazioni non comprensibili e che non sono utili alle nuove classi che si creano

Bisogna di conseguenza fare un `@Override` di toString. `@Override` è un'annotazione (non un commento!), queste servono ad esprimere delle intenzioni dello sviluppatore (guardare meglio).

- Le annotazioni sono delle informazioni che si aggiungono al codice per segnalare delle intenzioni dello sviluppatore al compilatore, senza influenzare direttamente il suo funzionamento: queste servono per fornire informazioni che aiutino con il controllo del codice, con il debugging e la documentazione.
- `Override` indica che un metodo in una classe deriva da un metodo di una superclasse e che la sovrascrive, facendono quindi l'override.
  - Quindi si chiarisce a compilatore e lettori del codice che il metodo nella sottoclasse non è un nuovo metodo con lo stesso nome, ma uno che vuole sostituire quello dela superclasse

L'Override è importante per vari aspetti:

- **controllo degli errori**: se si sbaglia a scrivere il nome del metodo o della sua firma (i suoi parametri), il compilatore o l'IDE mostreranno un errore. Questo per evitare di creare un nuovo metodo pensando di starne sovrascrivendo uno nuovo
- **chiarezza**: indica in modo esplicito che il metodo è una ridefinizione

```java
@Override
public String toString() {
    ...
    return ...;
}
```

Quando si fa l'override di `toString`, bisogna fare attenzione alla problematica degli oggetti intermedi nell'heap.

- Se in un ciclo si crea la stringa in maniera progressiva tramite `s += p`, ci ritroveremo nell'heap tante stringhe intermedie quante le volte che viene eseguito il ciclo. Questo perché le stringhe sono immutabili e vanno quindi riassegnate.
- Per risolvere questo problema possiamo utilizzare `stringBuilder`, un'implementazione delle stringhe mutabili *(vedere Lezione 3)*

`toString` andrebbe quindi generalmente usato con `stringBuilder`

Clone

In generale la presenza del metodo `clone` è da escludersi (soprattutto al nostro livello) per varie ragioni:

- i nostri contratti sono troppo deboli, e clone richiede un contratto poco chiaro e non impone regole rigide su cosa significhi "copia" (questo è il motivo principale)
- cercatele **[...]**

La Liskov spende molte pagine per spiegarlo, ma noi al suo posto utilizzero i `copy constructor` che presentano molti più vantaggi (sono un'alternativa molto più valida di `clone`).

Copy constructor

```java
class MyClass {
    int value;

    // Default constructor
    public MyClass(int value) {
        value = value;
    }

    // Copy constructor
    public MyClass(MyClass other) {
        value = other.value;
    }

    public void display() {
        System.out.println("Value: " + value);
    }
}

public class Main {
    public static void main(String[] args) {
        MyClass original = new MyClass(10);
        MyClass copy = new MyClass(original);  // Using copy constructor

        original.display();  // Output: Value: 10
        copy.display();      // Output: Value: 10
    }
}
```

Copy constructor è un costruttore che riceve un altro oggetto e produce una copia che è quanto più possibile uguale all'altro (quanto più possibile poiché non esiste un contratto generale ma lo si implementa da soli)

- questo tipo di copia è generalmente più sicuro, soprattutto per evitare interferenze tra l'oggetto originale e la copia

Perché non costruiamo un nuovo oggetto vuoto e ci mettiamo dentro le informazioni? Perché di solito questo è un'operazione molto più onerosa (soprattutto nel caso di `intSet`, vedere lezioni precedenti e gli handouts)

```java
/**
 * Initializes this set so that it contains the same elements of the {@code other} set.
 *
 * @param other the other set.
 * @throws NullPointException if the other set is {@code null}.
 */
public IntSet(IntSet other) {
    els = new ArrayList<>(other.els);
}
```

Equals

Definition

"Due oggetti sono uguali se sono equivalenti dal punto di vista comportamentale. Gli oggetti mutabili sono uguali solo se sono lo stesso oggetto; tali tipi possono ereditare equals da Object. Gli oggetti immutabili sono uguali se hanno lo stesso stato; i tipi immutabili devono implementare essi stessi equals"

- Liskov

La Liskov dice che bisognerebbe verificare l'uguaglianza solo tra oggetti mutabili. Nella pratica però è utile verificare che due oggetti siano indistinguibili, sia che questi siano mutabili o immutabili, e ci serve quindi un metodo di comparazione.

`equals` prende come argomento l'altro elemento e restituisce un booleano. Questo metodo deve avere un contratto forte poiché l'uguaglianza è una relazione d'equivalenza, e bisogna rispettarne quindi tutte e 3 le proprietà: riflessività, simmetria e transitività.

Anche qui l'implementazione di default è inutile, e quindi `equals` va implementato tramite un `@Override`

```
@Override
public boolean equals(Object obj) {
    if (obj == null) return false;
    if (obj istanceof IntSet) { // controllare se l'oggetto è effettivamente un IntSet per evitare errori unchecked non presenti nel contratto
        IntSet other = (IntSet)obj; // si converte obj in un IntSet
        if (els.size() != obj.els.size()) return false;
        // ..
        return true;
    } else {
        return false;
    }
}

public int hashCode() {
    // manca hashCode
}
```

- *NB: `instanceof` va utilizzato, e non `getClass()`*

IMPORTANTE: `equals` e `hashCode` vanno implementate coerentemente (se non lo si fa è difficile capire questo errore anche facendo debugging perché fa delle cose veramente inaspettate)!

`equals` va sempre messo insieme con `hashCode` perché in Java, se due oggetti sono uguali secondo il metodo equals allora questi devono avere lo **stesso valore** di hashCode *(chiamare hashCode su entrambi gli oggetti deve produrre lo stesso intero come risultato)*

- questo è fondamentale perché le strutture di dati basate su hash usano la hashCode per localizzare potenziali corrispondenze e equals per verificarne l'uguaglianza esatta

Non vale il contrario: diversi oggetti possono avere lo stesso hashCode (questo si chiama **collisione**), anche se sono diversi. Tuttavia, una buona implementazione della funzione hashCode cerca di ridurre al minimo le collisioni.

[... da vedere bene questa hashCode ...]

[tutte le strutture dinamiche di java che useremo per il progetto ...]

L11 - 08/11/2024

AF & RI [cap 5.5]

Nella specificazione dei dati, le scelte di come implementiamo le informazioni negli attributi (nei campi) influisce profondamente su come implementeremo i comportamenti di questi dati come metodi:

- se implemento un attributo come lista di duplicati o come lista senza duplicati, quando per esempio dovrò utilizzare un metodo per trovarne la lunghezza, il come implementeremo questo metodo nei due casi cambia radicalmente.

Dal lato dell'implementazione, laddove ci sono delle scelte quando bisogna decidere il come implementare i dati, sarebbe utile appuntarsi e documentare da qualche parte queste scelte.

Due utili strumenti che ci fornisce la Liskov per fare ciò sono la **funzione di astrazione** e la **rep invariant**: sono strumenti utili al programmatore per esprimere le sue scelte fatte nell'implementazione.

→ *Queste funzioni sono molto importanti per il progetto*

Abstraction Function

Queste funzioni parlano degli attributi, e quindi il loro **dominio** è legato a questi: ciascun attributo ha un insieme di valori possibili *(gli interi a 32 bit da un min a un max, le stringhe da quella vuota a tutte le possibili stringe, ecc)* , e quindi il loro prodotto cartesiano ($D_1 \times D_2 \times D_3$) ci dà tutti i possibili modi di valorizzare questi attributi della classe -> è il dominio della AF.

Il **codominio** è l'insieme dei valori del mondo reale che voglio rappresentare

- Raramente questa funzione sarà iniettiva (perché a un valore del mondo reale quasi sempre possono corrispondere più implementazioni)

Questa AF va dall'implementazione al mondo perché, nella situazione opposta, non sarebbe più una funzione: se AF andasse dal mondo all'implementazione, esistendo varie possibili implementazioni, a un elemento del dominio corrisponderebbero più elementi del codominio ($\neq$ function)

Esempi

- Funzione di astrazione di `intSet`: $AF_{intSet} : \text{List} \rightarrow \text{insiemi di interi}$
  esempio: $[1,2,4] \rightarrow \{1,2,4\}; \quad [2,1,3] \rightarrow \{1,2,4\}$
- Questa AF non ci aiuta però a rappresentare, per esempio, `Poly`:
  `List<Int> coeff` 1 2 3
  `List<Int> degre` 4 5 7

  $$(c,d) \rightarrow \sum_{i=0}^{|c|} c \cdot get(i) \cdot x^{d \cdot get(i)}$$

  Cosa succederebbe se nella lista dei coeff andassi a mettere un elemento in più rispetto a quella dei gradi?

Invariante di rappresentazione

Il dominio rimane quello della funzione di astrazione, mentre il **codominio** è `true` / `false`
Più precisamente il **dominio** sarebbe $D \subseteq P_1 \times D_2 \times D_3 : \text{RI}(D) = \text{TRUE}$ dove RI è la funzione di *rep invariant*

È l'elencazione delle proprietà che rendono possibile il calcolo della AF.

Nel caso di **Poly** la funzione RI è:

```
// The rep invariant is
//    c.trms ≠ null && c.trms.length >= 1 && c.deg = c.trms.length-1
//    && c.deg > 0 ⇒ c.trms[deg] ≠ 0
```

[

vedere dal libro per approfondire come questa andrebbe legata per bene con la AF, magari anche riportando delle informazioni in comune nelle due funzioni (ci devono essere informazioni duplicate?)

]

IMPORTANTE: queste funzioni non fanno parte del contratto della classe, ma vanno dove bisogna mettere i `Fields`, ossia gli attributi.

- Questa informazione, essendo documentazione dell'implementazione, non deve comparire nella Javadoc

Esempio con `intSet`:

```
public class IntSet {

    // Fields

    /*-
     * RI:
     *
     * - els != null
     * - els non contiene null,
     * - els non contiene duplicati
     *
     * AF:
     *
     * - gli elementi dell'insieme sono tutti e soli gli elementi di els
     *
     */

    // Constructor
    ...
    // Methods
    ...
}
```

Definition

The abstraction function explains the interpretation of the rep. It maps the state of each legal representation object to the abstract object it is intended to represent. It is implemented by the toString method.

The representation invariant defines all the common assumptions that underlie the implementations of a type's operations. It defines which representations are legal by mapping each representation object to either true (if its rep is legal) or false (if its rep is not legal). It is implemented by the repOk method.

- Liskov

Come si implementano AF & RI?

- Liskov ci dice che la AF può essere implementata nella `toString`. In questa assunzione però si dà per scontato che `String` sia isoformo al mondo.
- Per la RI ci dice di scrivere una funzione `boolean repOk()`. Questa funzione però non è sostitutiva della documentazione di RI, è quindi davvero necessaria? Per capire le varie condizioni necessarie, è più facile leggere la documentazione piuttosto che capire cosa faccia la `repOk()` -> è anche più facile da fare dal punto di vista del programmatore, non solo dell'utente che deve capire l'implementazione.
  - È comoda come verifica, ma è estremamente onerosa dal punto di vista dell'efficienza. Se ogni volta che eseguo il metodo per `intSet`, devo chiamare repOk() che fa un controllo quadratico sull'insieme...
  - **IMPORTANTE:** inoltre `repOk()` non deve sostituire i controlli sui parametri, che servono a garantire la totalità dei metodi con un eventuale sollevamento di eccezione.

Nella pratica è meglio scrivere la documentazione di queste due funzioni e basta: scrivere che una lista non contiene duplicati è più facile di implementarla con due cicli for. Inoltre ciò non ci induce ad usare `repOk()` per funzioni per le quali non andrebbe usata.

Assert

Esiste un modo di annotare le mie aspettative nel codice, in modo tale da poter scegliere tra due strade:

1. Il mio codice è sano, sono sicuro che funzioni e non faccio più controlli
2. Si aggiungono dei controlli che sono veri solo in fase di sviluppo e di testing

C'è un modo per determinare che alcune parti di codice siano eseguite condizionalmente se si è nella fase di testing e non nella fase di produzione

- assert praticamente fa dei controlli in testing, ma nella fase di produzione spegne tutti i controlli e non li esegue
- questo risolve il problema dell'efficienza e toglie anche la possibilità di usare repOk() come controllo dei parametri

```
public class IntSet {
    ...
    assert repOk(): "repOk failed"; // il messaggio che deve essere incluso nella eccezione

    private boolean repOk() {
        // i vari controlli miei per la RI implementati come codice Java
        return true / false
    }
    // vedere codice lezione 11 per le modifiche appropriate che mi sono perso
}
```

`repOk()` viene eseguito quando invece di eseguire la classe con `java -cp ..` si mette la flag `java -ea -cp ...`

Importante: `assert` ci permette quindi di riciclare l'idea Liskoviana di `repOk()`. `repOk()` va quindi usato solo con `assert`

PUNTI IMPORTANTI PER IL PROGETTO

Punto 1

Che rapporto c'è tra RI e mutabilità?

Immutabile non significa che la rappresentazioni non cambi, ma che non è possibile osservare modifiche alla rappresentazione dall'esterno.

- la rappresentazione può cambiare, a patto che **rispetti la RI**.
- dal lato del muro contrattuale rimane immutabile, ma dietro le quinte la rappresentazione, più specificatamente la sua implementazione, può variare.

La mutabilità dietro le quinte ci permette quindi gli **effetti collaterali benevoli**: per esempio, quando devo ricercare tanti elementi tante volte in una lista, ordino la lista per fare una ricerca dicotomica. **[cap 5.6.1]**

- il metodo ricerca rispetta quindi il contratto, e già che c'è ha l'effetto di ordinare la lista (in questo caso per rendere la ricerca più efficiente)

In generale si ha la tendeza di salvarsi e di tenersi da parte un risultato intermedio: **caching** (è un effetto collaterale benevolo). Questo però va **specificato** nella RI.

- se quando faccio la somma di un insieme mi salvo questo valore e mi tengo una variabile `sum_valide = true` e rendo quest'ultima `false` quando per esempio aggiungo un nuovo elemento nell'insieme, beh questa cosa va documentata nella RI, vanno aggiunte delle nuove condizioni.

Consiglio: non fare caching per semplificarci la vita. Non aggiungere attributi calcolati da altri attributi.

Punto 2

La RI è vera a patto che la rappresentazione nostra, quella dietro le quinte, non venga **mai** data all'utente. Lui potrebbe usarla per rompere il contratto, e noi dietro le quinte, nell'implementazione, non abbiamo più modo di controllarlo. **Non bisogna esporre la rappresentazione!** (exposing representation)

Questo può succedere quando:

- **parte della nostra rappresentazione è mutabile** (l'implementazione eh, non la parte osservabile dall'utente)
- **in ingresso, generalmente nei costruttori** (a volte nei metodi mutazionali):
  per esempio in un metodo di fabbricazione noi prendiamo una lista e vi facciamo tutti i controlli. A partire da quella poi generiamo la nostra lista, nuova e da zero: qui siamo a posto.
  - se invece noi facciamo i nostri controlli e poi TENIAMO quella stessa lista, noi ci stiamo tenendo e stiamo poi utilizzando nella nostra rappresentazione un elemento fornitoci dall'esterno di cui l'utente ha un riferimento. Questo utente potrebbe variare poi la nostra lista, rompendo la RI.
- in uscita con i metodi osservazionali. Non devo mai restituire con un metodo osservazione nessun tipo di riferimento alla nostra rappresentazione o a parti di esse. Bisogna trovare dei metodi alternativi: restituire una copia, non dare direttamente quel metodo a disposizione, ecc...

L12 - 13/11/2024

Reasoning about Data Abstractions [cap 5.7]

Come usare AF ed RI per valutare se il codice che man mano stiamo scrivendo sia corretto?

- Come adoperarli per valutare la correttezza del nostro codice

Preservazione della RI [cap 5.7.1]

Quando si scrive RI bisogna analizzare i propri metodi accertandosi che questi preservino correttamente l'invariante di rappresentazione.

- si può ritoccare questa rappresentazione, a patto che dal punto di vista dell'osservatore questa rimani invariata (effetti collaterali benevoli)

Nel caso dell'**immutabilità** questo controllo va fatto semplicemente nei costruttori (non nei metodi di produzione, che tanto faranno lo stesso appoggio al costruttore)

- nel caso di effetti collaterali benevoli bisogna aggiungere i metodi dove questo avviene

Nel caso di oggetti **mutabili** questo controllo va fatto su tutti i metodi (costruttore incluso)

Noi, in un metodo, a metà dell'operazione di questo metodo, ossia nel suo corpo (mentre lo si sta svolgendo), ci sarà un momento in cui la RI non sarà vera, ma ciò non è importante (poichè ci troviamo nel bel mezzo del flusso di esecuzione e nessun altro metodo potrà esser invocato su quell'oggetto finchè non si esce dal metodo iniziale.)

- L'importante è ACCERTARSI che all'**uscita del metodo** la RI sia **vera**!
- Anche all'entrata nel metodo la RI dovrà essere ovviamente vera

Nel caso in cui la rappresentazione, o una sua parte, sia mutabile, bisogna accertarsi di non manifestare la nostra rappresentazione: una volta che la persona esterna ne ha un riferimento può usare l'oggetto come gli pare. Non abbiamo quindi più modo di difenderci da ciò, e **non possiamo più garantire la preservazione della RI**.

- **IMPORTANTE:** Non mostrare i dettagli implementativi significa anche non offrire metodi osservazionali ingenui. Le implementazioni non devono essere vincolanti per lo sviluppatore: se lui offre un metodo osservazionale che restituisce una lista, anche senza riferimenti alla sua rappresentazione, se in futuro lui decidesse di usare un database, sarebbe vincolato da questo metodo che restituisce una lista a fare un'implementazione che "traduce" dal database a questa lista, agghiacciante.
  - Lo sviluppatore non è più libero di implementare

Ragionare sulle Operazioni [cap 5.7.2]

Noi abbiamo il nostro metodo

```
pre condizioni
post condizioni (modifies)
metodo_T ... (...){
    // RI vera
    ...
}
```

Mentre la RI è un predicato degli attributi (i nostri parametri nel codice dell'oggetto), queste pre e post condizioni parlano del mondo reale.

Qualunque entità del mondo che soddisfa le pre condizioni, quando la opero tramite gli attributi e all'uscita dal metodo, quando poi la guardo nel mondo deve soddisfare le post condizioni!!

IntSet metodo insert(): dimostrazione per parti

- Se nel mondo un'entità S non contiene x, noi con l'AF siamo in grado di garantire qualunque `els` in cui non compare x ? Se x non c'era e ce lo aggiungo, si ottiene `els' = els + [x]` ed $S' = S \cup \{x\}$
  - non ci sono duplicati, e la RI rimane vera
- Se invece la contiene, AF produrrà una qualunque lista `els` con x dentro. `els` è immutato, $S' = S$ e la RI è vera.

In generale: si prende una qualsiasi entità che soddisfi le pre condizioni, con la AF (inversa, dal mondo mi trovo le rappresentazioni) mi trovo le sue possibili implementazioni, ragiono su ciascuna di esse che nel mio codice diventeranno delle altre implementazioni. Quando queste poi usciranno nel mondo tramite la AF, queste dovranno rispettare le post condizioni.

- per IntSet non solo deve funzionare per tutti gli insiemi possibili nel mondo reale, ma anche per ogni possibile implementazione di tutti questi insiemi. Esempio: per l'insieme del mondo reale $\{1, 2, 3\}$ le post condizioni dovranno essere rispettate per tutte le implementazioni che scaturiscono da: $[1, 2, 3], [1, 3, 2], [2, 1, 3], \cdots$
  - se per esempio quando aggiungo un elemento controllo dall'indice 1 e non 0, nel caso aggiungessi 2 mi ritroverei con le implementazioni precedenti: $[1, 2, 3], [1, 3, 2], [2, 1, 3, 2], \cdots$
    Qui scaturisce quindi il problema: le post condizioni non sono rispettate nella terza rappresentazione, e quindi non è rispettato per ogni possibile rappresentazione di ogni possibile elemento (insieme) del mondo reale

Importante: post condizioni devono essere valide per qualsiasi entità del mondo reale e per qualunque modo la si rappresenti!
Questo problema sorge quando la AF non è iniettiva: quando dalla AF si va dalla rappresentazione al mondo, questa avrà un'unico valore possibile. Ma se AF non è iniettiva e si passa invece dal mondo alla rappresentazione (la si percorre al contrario), questo significa che questa entità avrà tante possibili rappresentazioni, e io **devo considerarle tutte**.

- Questo problema può capitare nel **caching** *[il perché capiscilo da solo]*

handouts/src/main/java/it/unimi/di/prog2/h11/SparsePoly.java

Se si considera `handouts/src/main/java/it/unimi/di/prog2/h11/SparsePoly.java`, la AF alla linea di codice 76 è iniettiva perché la RI ci impone che la lista che rappresenta il polinomio sia in ordine crescente.

- normalmente $x^2 + 3x$ potrebbe corrispondere anche a $3x + x^2$, ma in questo caso la RI ce lo nega, e questo vincolo rende la AF iniettiva.

```
1  //handouts/src/main/java/it/unimi/di/prog2/h11/SparsePoly.java
2  ...
3  /** The array of terms (in increasing degree). */
4    private final List<Term> terms;
5
6    /*
7     * AF: associa a terms il polinomio dato dalla somma dei monomi in terms.
8     * RI: terms != null
9     * terms non deve contenere null
10    * terms non contiene due termini diversi col medesimo grado; detto
11    * altrimenti, 0 <= i < j < terms.size() allora
12    * terms.get(i).degree < terms.get(j).degree
13    */
14 ...
```

alla linea 147 abbiamo un metodo `findByDegree` che è parziale perché funziona solo in liste ordinate in maniera crescente dal punto di vista del grado. Questo però ha senso poiché questo metodo è privato (e non accessibile quindi dall'utente, ma serve a noi come supporto per altri metodi)

L13 - 15/11/2024 [cap 5.8]

Astrazione dei dati: riepilogo

specificazione	**implementazione**
contratto	rappresentazione
- informazioni	- attributi
- comportamenti	- metodi
pre/post condizioni	AF / RI

Metodo della classe: ???

Metodo statico: appartiene alla classe e non a una specifica istanza. Può essere chiamato senza creare un oggetto della classe, usando direttamente il nome della classe. Non può accedere direttamente ai membri di istanza (variabili o metodi non statici) perché non esiste un'istanza specifica associata al metodo.

```java
public class Calcolatrice {
    // Metodo statico
    public static int somma(int a, int b) {
        return a + b;
    }
}

public class Main {
    public static void main(String[] args) {
        // Chiamata al metodo statico senza creare un oggetto
        int risultato = Calcolatrice.somma(5, 3);
        System.out.println("Risultato: " + risultato);
    }
}
```

Metodo dell'istanza: appartiene a un'istanza specifica della classe. Deve essere invocato su un oggetto creato della classe. Usato per manipolare o accedere ai dati specifici di quell'oggetto.

```java
public class Persona {
    private String nome;

    // Metodo di istanza
    public void setNome(String nome) {
        this.nome = nome; // Modifica il dato di istanza
    }

    public String getNome() {
        return nome;
    }
}

public class Main {
    public static void main(String[] args) {
        // Creazione di un'istanza di Persona
        Persona persona = new Persona();

        // Chiamata a un metodo di istanza
        persona.setNome("Kevin");
        System.out.println("Nome: " + persona.getNome());
    }
}
```

Definition

- A data abstraction is mutable if it has any mutator methods; otherwise, the data abstraction is immutable.
- There are four kinds of operations provided by data abstractions: creators produce new objects “from scratch”; producers produces new objects given existing objects as arguments, mutators modify the state of their object; and observers provide information about the state of their object.
- A data type is adequate if it provides enough operations so that whatever users need to do with its objects can be done conveniently and with reasonable efficiency.

- Liskov

Mutabile o immutabile? [cap 5.8.1 - 5.8.2]

Le nostre competenze (**operazioni**) sono divise in diverse categorie:

- **creatori**: ?sono metodi della classe? (non si chiamano su un'istanza perché sono loro a crearla).
  Questi metodi creano un oggetto del loro tipo dal nulla, senza prendere in input altri oggetti dello stesso tipo. Tutti i creatori sono **costruttori**, però a volte i costruttori prendono argomenti del loro tipo (come il *copy constructor*), e questi non sono creatori.
  - **costruttore:** si tratta di metodi speciali che hanno lo stesso nome della classe e che non hanno un tipo di ritorno, ma hanno l'effetto di creare un nuovo oggetto dal nulla.
    - *Implicitamente poi inizializzano i parametri da passare alla* `new` *che dovrà allocare nell'heap, e poi la new ne restituisce il puntatore.*
  - **fabbricatori**: sono dei creatori che devono necessariamente fare affidamento sui costruttori: questo perché non possono creare un'oggetto dal nulla ma hanno bisogno di una `new` che quindi va a richiamare il costruttore.
    Possono essere un metodo statico o un ?metodo di istanza? **[guarda capitolo 15.1 PDJ]**
    - I fabbricatori statici si differiscono dai costruttori perché:
      1. hanno un nome
      2. non necessitano di creare un nuovo oggetto ogni volta che sono invocati
      3. possono ritornare un oggetto di ogni sottotipo del loro tipo di ritorno *(da capire bene cosa sia questo loro tipo di ritorno)*
      4. la classe dell'oggetto ritornato può variare da chiamata a chiamata in funzione dei parametri di input
      5. A fifth advantage of static factories is that the class of the returned object need not exist when the class containing the method is written *(da capire bene)*
- metodi di **produzione** o produttori: metodi dell'istanza, che quindi vengono richiamati su un'istanza già presente. Questi metodi prendono un oggetto del loro tipo in input e creano un altro oggetto sempre del loro tipo. Possono essere sia costruttori che metodi.
  - *Per esempio,* `add` *e* `mul` *sono produttori per* `Poly`: non potendo modificare l'istanza su cui vengono richiamati, hanno bisogno di crearne una nuova generalmente (questo perché vengono generalmente utilizzati su oggetti immutabili)
- metodi di **osservazione**: sono metodi che prendono un oggetto del loro tipo in input e ritornano risultati di altri tipi. Sono utilizzati per ottenere informazioni a riguardo degli oggetti.
- metodi **modificazionali**: sono metodi che modificano oggetti del loro tipo. Solo tipi mutabili possono avere metodi di modificazione.

Negli oggetti mutabili: sì creatori, sì di osservazione, sì di modificazione, di produzione non è generalmente necessario.
Negli oggetti immutabili: sì creatori, sì di osservazione, no metodi modificazionali, sì di produzione

Gli oggetti mutabili sono generalmente più efficienti, mentre quelli immutabili sono più sicuri. Bisogna bilanciare le due cose e scegliere quello che più si confà all'entità che andiamo a rappresentare.

Adeguatezza [cap 5.8.3]

È una proprietà importante che la nostra classe deve possedere ed è anche una delle più difficili da ottenere.

Quali e quanti metodi bisogna mettere in una classe per rendere il contratto adeguato.

- I metodi devono essere sufficienti per tutti gli **usi intesi**. Generalmente bisogna fare un bilanciamento tra la **convenienza** (quanto è più conveniente per l'utente che usa la classe adoperare quel metodo se è direttamente all'interno della classe), la **quantità di vincoli** che si pongono al programmatore e la **comprensibilità** (più una classe è complessa e più sarà difficile per l'utente capirne i metodi)
- **metodi per osservare**: non va bene utilizzare solo `toString` per visualizzare l'oggetto, serve un metodo di osservazione per visualizzarlo come oggetto vero e proprio nel caso ci sia bisogno di, per esempio, ordinarne i campi in maniera differente.
- **piena popolabilità**: soprattutto per gli immutabili, se c'è un'istanza del mondo reale che non riesco a rappresentare, non va bene. Il contratto che costruisco deve raggiungere tutti gli elementi del codominio (del mondo reale) in maniera plausibile. La AF deve essere suriettiva (questa proprietà implica la suriettività, ma non vale il viceversa: magari riesco a trasporre tutte le entità del mondo nei miei attributi, e questo è garantito dalla suriettività della AF, ma devo anche garantire che i metodi funzioni sulla rappresentazione di quell'entità, cosa che invece la AF non può garantire)
  - $\text{Piena popolabilità} \Rightarrow AF \text{ suriettiva}$

Benefici [cap 5.9]

Due sono i benefici cruciali che otteniamo seguendo questa metodica:

1. **Località** (beneficio dello sviluppatore): capacità di ragionare sul codice che si sta scrivendo osservando solo il codice che si sta scrivendo, e quindi solo localmente. Non devo preoccuparmi di guardare altre porzioni del codice. Questo perché il suo comportamento dipende solo dalle pre e post condizioni espresse lì, in quella parte di codice. (Nella rappresentazione ci aiutano invece AF e RI).
   - Possiamo presupporre per **induzione strutturale** che se le parti più semplici funzionano, allora anche la struttua più complessa che si basa su queste funzioni.
   - Poiché questo funzioni davvero è necessario che la mia rappresentazione sia invisibile a tutti gli altri se non attraverso i metodi del contratto. Se dall'esterno della classe è possibile agire sugli attributi della classe mediante altri mezzi che non siano i metodi messi a disposizione, non è più possibile ragionare localmente in quanto non si può più garantire la RI. **Le write devono avvenire solo con i metodi, altrimenti la rappresentazione potrebbe non rispettare la RI**.
2. **Modificabilità** (beneficio dell'utente): se si vuole fare manutenzione del codice (facendo un bugfix per correggere un errore oppure per fare un miglioramento), lo si può fare tranquillamente senza che questo influisca sul contratto e quindi sugli utenti che utilizzano la mia classe.
   - Per garantire ciò, **le read devono avvenire solo con i metodi**. Gli utenti non devono basarsi su proprietà che non sono presenti nel mio contratto. Altrimenti si crea un **vincolo**, perché gli utenti che utilizzano una proprietà della mia rappresentazione tramite non dei metodi che fornisco io, ma andando ad osservare mi vincolano a non poter più modificare quella cosa.

[**esercizio in classe**]

Record

```
  public record Term(int coefficient, int degree) {

    /*-
     * AF:
     *
     *  AF(coefficient, degree) = coefficient * x^degree
     *
     * RI:
     *
     *   - degree >= 0
     *   - coefficient != 0
     *
     */

    /**
     * Builds a term.
     *
     * @param coefficient the coefficient.
     * @param degree the degree.
     * @throws IllegalArgumentException if {@code n} &lt; 0 or {@code c} is 0.
     */
    public Term { // using the compact constructor, cannot declare throws
      if (degree < 0) throw new IllegalArgumentException("A term cannot have a negative
exponent.");
      if (coefficient == 0)
        throw new IllegalArgumentException("A term cannot have a zero coefficient.");
    }
  }

```

Records are a special kind of class in Java that are intended to be immutable data carriers. The compact constructor allows you to perform validation or other logic without having to explicitly declare the parameters again.

Record declaration: the upper example, the `Term` record is declared with two fields: `coefficient` and `degree`.
Compact constructor: it's defined using the record's name without parameter declarations. It allows you to add validation logic directly.
Validation logic: inside the compact constructor, validation checks are performed.

The compact constructor automatically uses the fields declared in the record, so you don't need to repeat them in the constructor's parameter list. This makes the code more concise and readable.

When using a record in Java, the compact constructor provides a concise and readable way to define a constructor with validation logic. If you were to achieve the same functionality using a traditional class, you would need to write more boilerplate code.

- A `private final` is attributed for each of its fields

L14 - 15/11/2024

Astrazione Iterazione [cap 6]

Si una collezione di oggetti omogenei e si vuole effettuare un'azione su tutti questi oggetti (la si vuole scorrere):

```
FOR x in "collezione"
    A(x)
```

Questa iterazione deve avere un suo luogo: bisogna capire dove farla.

- se si vuole farla **dentro** (implementando il metodo all'interno della classe, e quindi avendo a disposizione la rappresentazione) questa operazione è facile ed efficiente

- **fuori** è laborioso ed inefficiente

Mettere però questi metodi dentro risulta in un **problema di adeguatezza**:

- se ho un metodo per fare la somma di un insieme, si è costretti ad aggiungere dei metodi per fare la somma dei numeri pari, dei dispari, di un intervallo, ecc...
  - se aggiungo tutti questi metodi però mi pongo dei vincoli in futuro. Se questi metodi non mi servono più dopo qualche mese si è costretti a reimplementarli in quanto presenti nel contratto.

Bisogna quindi fare queste iterazioni fuori:

- **mostrare la propria rappresentazione** (passando l'elenco els) non va assolutamente bene (si ricade nell'esposizione della rappresentazione -> l'utente può rendere invalida la IR)
- si fa una **copia della nostra rappresentazione**, e di conseguenza non la si espone (non si da riferimento a nessuna parte mutabile della nostra rappresentazione). Due problematiche:
  - bisogna fare una copia, con oggetti grandi è un processo laborioso e che consuma memoria
    - problematiche tempistiche: il ciclo for dell'utente magari si ferma dopo 10 elementi, e io però ho copiato tutti i due milioni di elementi -> inefficienza
      Si hanno quindi dei limiti di spazio e di tempo
  - Inoltre si fa una forte assunzione passando una copia, per esempio, di una lista: da contratto noi promettiamo che saremo sempre capaci di restituire il nostro oggetto, comunque esso sia rappresentato dietro le quinte, sotto forma di lista. Si espone un contratto con un valore sofisticato come una lista; in futuro se si decidesse di reimplementare il tutto con magari non più una lista ma un database, andando quindi a cambiare la nostra rappresentazione, dobbiamo mantenere la promessa di restituirne una copia sottoforma di lista $\rightarrow$ **si pongono dei vincoli sul tipo di dato** (`unmodifiableList()`)
    - Si supponga che questa lista non contenga dei tipi primitivi ma sia una lista che contiene dei valori per riferimento, bisogna assicurarsi che questi oggetti nella lista non tengano i riferimenti degli oggetti nella nostra rappresentazione: bisogna quindi fare una copia anche di ciascuno di questi o bisogna assicurarsi che queste siano immutabili.
      $\rightarrow$ ** Bisogna fare una copia degli elementi della classe se questi non sono primitivi**
- **astrazione iterazione**

A.I.

Se si ha bisogno di iterare si aggiunge un **iteratore** che restituisce un **generatore**

- il `generatore <T>` è una qualunque classe che accetta un tipo T che implementa due abilità:
  1. prima abilità: metodo che restituisce un booleano `hasNext()`, ci dice quindi se c'è un elemento successivo
  2. seconda abilità: restituisce un oggetto del tipo T quando si chiama il metodo `next()`.
     restituisce quindi l'oggetto successivo

`next()` può sollevare un'eccezione: `T next() throws NoSuchElementException`

- questo perché ovviamente un utente magari richiede un elemento successivo senza assicurarsi con `hasNext()` che effettivamente ci sia.

```
GENERATORE <T>
    boolean hasNext()
    T next() throws NoSuchElementException
```

Come si implementa quindi questo ciclo for sulla collezione:

```
Gen g = s.elementi()
while (g.hasNext()) {
    int x = g.next():
        A(x)
}
```

- Questo è l'uso dell'iteratore ed è **esterno** (non c'è quindi nessun problema di adeguatezza)
- Unica cosa: `s` deve avere un iteratore interno `elementi` che restituisce un generatore che viene poi utilizzato esternamente

Iteratore per IntSet

IntSet.java

```java
public class IntSet implements Iterable<Integer> {
    ...
    /**
     * Returns a generator over the elements in this set.
     *
     * @returns the generator
     */
    public Iterator<Integer> iterator {
        return new IntGenerator();
    }
}
```

IntGenerator.java

```java
package ...

public class IntGenerator {

    /**
     * Generatore che restituisce tutti gli elementi di una lista
     */
    public class IntGenerator implements Iterator<Integer> {

        /** Lista di elementi dell'insieme. */
        private final List<Integer> els;

        /** Indice del prossimo elemento da restituire */
        private int index;

       /*-
        * AF:
        *   l'iteratore restituirà els.get(index) a meno che index >= els.size()
        * RI:
        *   - els non è null e non contiene null;
        *   - 0 <= index <= els.size()
        */

        /**
         * Costruisce un egenratore data una lista di interi.
         *
         * @param els la lista di interi.
         */
        public IntGenerator(List<Integer> els) {
            Objects.requireNonNull(els, "no null list")
            for (Integer e : els) if (e == null)
                throw new IllegalArgumentExcpetion("no null elements in list")
            this.els = els; // PUNTO 1: qui noi non stiamo esponendo la rappresentazione
        }

        @Override
        public boolean hasNext() {
            return index < els.size();
        }

        /**
         * Restituisce il prossimo intero nella lista.
         *
         * <p> si garantisce che il valore restituito non sia mai {@code null}
         * @see Iterator#next
         */
        @Override
```

```
48          public Integer next() {
49              if (!hasNext()) throw new NoSuchElementException("no more elements");
50              int next = els.get(index);
51              index += 1
52              ...
53          }
54      }
55
56  }
```

IntSetIteratorClient.java

```
1   package ...
2
3   public class IntSetIteratorClient {
4       public static void main(String[] args) {
5           IntSet s = new IntSet;
6           s.insert(3);
7           s.insert(1);
8           s.insert(2)
9
10          Iterator<Integer> g = s.iterator()
11
12          while (g.hasNext()) {
13              int x = g.next();
14              System.out.println(x);
15          }
16
17          for (int x : s)
18              System.out.println(x);
19      }
20  }
```

Invece di inventarci un generatore nuovo ogni volta, non ci forziamo noi a stare attenti che abbia quei due metodi lì e a farne il contratto ma facciamo uso delle API: usiamo l'interfaccia, che è un contenitore di contratti: nel nostro caso `Interface Iterator<T>`

- un generatore rispetta il suo contratto implementando un'interfaccia `Iterator`
- non doppiamo fare i contratti di `hasNext()` e `next()`, perché andiamo a fare gli override di quelli già presenti nell'interfaccia `Iterator`

PUNTO 1 riga 33 di **IntGenerator.java** noi non stiamo esponendo la rappresentazione, perché stiamo dando le chiavi di casa all'interfaccia `Iterator` (e non all'utente) che promette da contratto di fare quello che ci aspettiamo.

- se tutto coloro i quali con cui collaboro si attengono al loro contratto, allora il mio contratto è valido. Se io adopero uno che non rispetta il proprio contratto, anche noi non potremo garantire che il nostro sia rispettato. Il mio contratto funziona finché gli altri fanno funzionare i loro.

Ora abbiamo finalmente fornito ad `IntSet` un modo per osservare il suo contenuto tramite un iteratore.

Piccolo dettaglio da sistemare: preferiamo il ciclo for al while con `int x = g.next()`

- c'è un modo per specificare che un iteratore

Iteratore "naturale": nella classe che deve restituire un iteratore ce n'è uno speciale che è `iterator`. Se utilizziamo `iterator` al posto del nostro `elmenti` possiamo utilizzare anche una specie di for each sui nostri oggetti.

- Bisogna specificare che la nostra classe implementa `Iterable<T>` per rendere i nostri oggetti iterabili tramite un ciclo for each. Questa classe ha poi bisogno di un `Iterator<T>` per forza.
  - vedere interfaccia `Iterable<T>`, che è quella che implementa il metodo `Iterator<T>`, l'altra interfaccia che abbiamo utilizzato)

L'iteratore consuma quanto serve, consente di fare l'uscita anticipita, non fa copie. **TOP**

- Se devo iterare degli immutabili, faccio per ogni elemento una copia e poi restituisco quella con l'iteratore. Così evitiamo di esporre la nostra rappresentazione

Non è sempre detto che senza una collezione dalla quale pescare gli elementi non ci possa essere un iteratore (iteratore non è sinonimo di collezione):

```
package ...

public class IntRage implements Iterable<Integer> {

    @Override
    public Iterator<Integer>
        [...]
}
```

L15 - 15/11/2024 [to check]

Nester Static Classes & Inner Classes

Bisogna ora capire dove porre le classi in java

- Le classi possono essere messe **esternamente** (separate e in file diversi). Se la classe è `public` questa va messa in un file con il suo nome
- Possono essere messe dentro le classi creando delle **classi annidate** (opzione che abbiamo già esplorato con i record, una speciale tipo di classe)
  - **nested static classes**: [...] (dipendono dalla classe)
  - **inner classes** (dipendono dall'istanza). Vuol dire che per poterne parlare bisogna avere quella istanza lì. Sono state create attraverso il riferimento a quella istanza.
- Dentro nei metodi: **classe locale**. Sono sempre private nella misura in cui le variabili locali del metodo sono visibili solo all'interno del metodo.
  - Sono di un utilizzo abbastanza peculiare. Un modo per creare una classe locale è crearla al volo: **classi locali anonime**. È una classe che non ha un nome, ma che non ci serve effettivamente perché nasce e muore lì. Queste possono essere utilizzare per il generatore all'interno del metodo `Iterator<Integer> iterator()`. Zucchero sintattico. Essendo anonime non hanno un costruttore.

    ```
    @Override
    public Iterator<Integer> iterator() {
        return new Iterator<Integer>() {
            ...
        }
    }
    ```

    Queste sono utili perché le classi sono l'unico modo per definire comportamenti [...]

Quando si ha una classe alcuni suoi attributi, classi interne e metodi possono essere dichiarati `static` significando:

- per i metodi che appartengono alla classe e non ad una sua istanza (non possono quindi accedere a `this`)
- nel caso degli attributi significa che quelle variabili sono costanti (per esempio ti salvi il polinomio statico in un attributo statico se ti serve)
- se nella classe annidata, la dichiaro statica, ciò significa che quella classe è [...]

Non possono esistere classi esterne private, sono per forza public. Ciò non vale per le classi e per gli attributi interni.

- Le classi private non sono istanziabili dall'esterno

Non si può avere un `record` che non sia `static`.

Il costruttore fa il controllo sulle singole istanze che vengono create. Nel caso in cui però ci sia la necessità di fare controlli, mentre si costruiscono queste istanze, su tutte queste istanze create, in quel caso il costruttore non è in grado di farlo.

- Ciò che possiamo fare è rendere il costruttore privato ed utilizzare un metodo di fabbricazione **[to check]**

[...]

NS Classes

ContenitoreNS.java

```java
package ...

public class ContenitoreNS {
    public static final int NUMERO_PIU_IMPORTANTE = 42;

    public static class Contenuto {
        private final String contenuto;
        public Contenuto(String contenuto) {
            this.contenuto = Objects.requireNonNull(contenuto);
        }

        @Override
        public String toString() {
            return contenuto;
        }
    }

    public static void main(String[] args) {
        Contenuto c = new Contenuto("pippo");
        System.out.println(c);
        System.out.println(NUMERO_PIU_IMPORTANTE);

        /* non si fa come le tre righe sotto */
        ContenitoreNS contenitore = new ContenitoreNS();
        system.out.println(contenitore.NUMERO_PIU_IMPORTANTE);
        Contenuto contenuto2 = contenitore.new Contenuto("pippo")
    }
}
```

Se si vuole usarlo da esterno:

ContenitoreNSClient.java

```java
package

public class ContenitoreNSClient {
    public static void main(String[] args) {
        //Contenuto c = new Contenuto("pippo");
        //System.out.println(c);
        //System.out.println(NUMERO_PIU_IMPORTANTE);
        /* non è possibile fare ciò esternamente */
        ContenitoreNS.Contenuto c = new ContenitoreNS.Contenuto("pippo");
        System.out.println(c);

        System.out.println(ContenitoreNS.NUMERO_PIU_IMPORTANTE);
    }
}
```

Inner Classes

Così come esistono metodi e attributi di istanza, esistono anche classi annidate di istanza: **inner classes**. Sono classi nested non static.

ContenitoreI.java

```java
package ...

public class ContenitoreI {

   private final String contenitore;

   public ContenitoreI(String contenitore) { // costruttore outer class
        this.contenuto = Objects.requireNonNull(contenuto);
   }

    public class Contenuto {
        private final String contenuto;
        public Contenuto(String contenuto) { // costruttore inner class
            this.contenuto = Objects.requireNonNull(contenuto);
        }

        @Override
        public String toString() {
            return "Io sono il contenuto " + contenuto + " di " + contenitore;
        }
    }

    public static void main(String[] args) {
        ContenitoreI contenitore = new Contenitore("Scatola biscotti");
        Contenuto b1 = contenitore.new Contenuto("Biscotto cioccolato");
        Contenuto b2 = contenitore.new Contenuto("Biscotto integrale");

        System.out.println(b1 + ", " + b2);
        // Io sono il contenuto Biscotto cioccolato di Scatola biscotti, Io sono il contenuto
Biscotto integrale di Scatola biscotti
    }
}
```

Se in una classe ci sono solo metodi statici (ed è quindi una classe di utilità), non c'è bisogno allora di istanziarla. Per evitare che sia istanziabile dall'esterno si fa un costruttore senza argomenti che rendo privato, così nessuno dall'esterno potrà fare `new MyClass`.

- utile per esempio per una classe `math`: non devo invocare radice quadrata su un'istanza, io uso il metodo stati della classe radice quadrata su un numero che gli passo.

Classe interna privata: è visibile solo a me, utile per modellarmi le mie cose. Parallelismo con i metodi privati.

Il generatore va creato come una private **inner classes**. Così evitiamo di passargli gli argomenti, dato che il generatore diventa dipendente dall'istanza e può quindi vederne gli attributi. Ciascun iteratore vede gli attributi dell'istanza della classe.

- Ora il generatore non è più una classe esterna ed è inoltre privata, nascosta
- Dato che il suo obiettivo è ispezionare gli attributi della mia classe [...]

Meglio ancora integrarle come classi anonime.

Gli iteratori di default hanno il metodo `remove()`, che rende l'interfaccia non più immutabile.

- gaurdare file in handouts

```
@Override
public void remove() {
    trhow new UnsupportedOperationException("don't touch")
}
```

- risolviamo il problema facendo un override del metodo "importato"

Si può fare automaticamente così:

```
@Override
public Iterator<Integer> iterator() {
    return Collections.unmodifiableList(els).iterator();
}
```

IMPORTANTE PER PROGETTO:

1. In molti iteratori c'è un'**inversione della logica**: c'è uno "scambio dei ruoli" tra next/hasNext.
   - Utile quando non si può determinare se c'è un prossimo elemento se non cercandolo. Tanto vale farlo con l'hasNext e propagare questo valore poi nel next.
2. Iteratori che non derivano da "Collezione" (tipo iterare le cifre decimali di un `long`)

L16 - 27/11/2024 [cap 7.1/2]

gerarchia dei tipi [...] introduzione

L17 - 29/11/2024 [cap 7.3-5]

Gerarchia tipi

S < T: S è un sottotipo di T

S extends T: ne estende il comportamento (ha più competenze, ha uno stato più ampo). Le istanze di S rimangono però un sottoinsieme di T.

Questa cosa deve soddisfare l'**LSP**.

class S extends T

- **specificazione**: informazioni; competenze (aggiunge o "specializza")
- **implementazione**: stato/rappresentazione; metodi (+ costruttore)

Nella specificazione queste competenze, che siano di aggiunte o di specializzazione, ci vanno ad estendere il contratto.

Nei metodi che ereditiamo dal nostro supertipo (ossia dal padre) abbiamo due opzioni: o li si tengono così come sono, altrimenti bisogna sovrascriverli tramite un `@Override` (questo perché magari alcuni metodi necessitano appunto di essere specializzati per il sottotipo con cui stiamo lavorando).

Il figlio non vedrà gli attributi del padre in quando privati, ma potrà accedervi tramite i metodi del padre. La AF quindi aggiunge informazioni all'ouput del padre, e la IR dovrà solo esprimere vincoli solo sugli attributi visibili all'interno della sottoclasse (fa riferimento al padre).

- AF e RI quindi non hanno bisogno di fare controlli sugli attibuti del padre (non possono proprio farlo) e quindi si assume che il contratto del supertipo sia valido. Si va avanti quindi per riferimento.
- metodi e attributi privati non sono visibili al sottotipo. Vengono solo ereditati robe pubbliche. Discorso a parte per metodi pubblici statici: possono essere invocati nel sottotipo solo facendo riferimento al supertipo.

[Dal supertipo si ereditano metodi, ecc. È foriero di disastri]

MaxIntSet extends IntSet

Estende IntSet restituendo il numero più grande

src/main/java/it/unimi/di/prog2/h17/MaxIntSet.java

```java
package ...;
import ecc.IntSet;

/**
 * A {@link MaxIntSet} is a {@link IntSet} capable of returning its maximum value.
 */
public class MaxIntSet extends IntSet {

    /** An integer such that biggest >= x for every x in this set. */
    private int biggest = 0;

    /*-
     * AF(biggest) = is the set given by AF of the supertype.
     *
     * RI:
     *  - if size() > 0, then biggest is the maximum value in the set
     *    else biggest can be anything
     */

    /** Construct an empty {@code MaxIntSet}. */
    public MaxIntSet() {
        super();
    }
```

```
24
25      public MaxIntSet(MaxIntSet other) {
26          super(other);
27          biggest = other.biggest;
28      }
29
30      /**
31       * Returns the maximum value among the integers in the set
32       *
33       * @return the maximum integer
34       * @throws NoSuchElementException if the set is empty
35       */
36      public int max() {
37          if (size() == 0) throw new NoSuchElementException("an empty set has no max value")
38          return biggest;
39      }
40
41      @Override
42      public void insert(final int x) {
43          if (size() == 0) biggest = x;
44          super.insert(x);
45          if (x > biggest) biggest = x;
46      }
47
48      @Override
49      public void remove(final int x) {
50          super.remove(x);
51          if (x == biggest) {
52              biggest = Integer.MIN_VALUE;
53              for (int i : this)
54                  if (i > biggest) biggest = i;
55          }
56      }
57  }
```

- MaxIntSet non eredita i costruttori del supertipo IntSet. Si fa un costruttore vuoto che richiama `super()`
- Non bisogna riscrivere i contratti di `insert` e `remove` poiché questi li ereditano dal supertipo e non cambiano, sì vanno ad aggiornare biggest ma questo non deve interessare all'utente, non tocca il contratto. Va a cambiare solo l'implementazione.
- `x` viene dichiarato **final** sia nel metodo `insert` che `remove` perché non c'è bisogno di modificare il valore che viene passato in input. È utile per evitare errori in cui per esempio, per sbaglio si riassegna il valore di `x`

NB: Tutti i metodi i quali contratti non possono influenzare il valore dell'IR non hanno bisogno di essere ritoccati, viceversa, vanno ritoccati.

Attenzione: non invertire i rapporti di gerarchia: il quadrato va implementato come sottotipo del rettangolo e non viceversa [... guardare bene ...]

LSP (Liskov Substitution Process)

Esempio: tipo HIST

Si vuole costruire un tipo `hist` (istogramma) che raccoglie una collezione di rettangoli e dopodiché è in grado di rappresentarli in ordine decrescente di altezza.

HIST h: voglio avere come output i rettangoli ordinati in base all'altezza

Metodi: `add` per aggiungere i rettangoli, `toString` per metterli in questo ordine, `normalize(b)` per rendere tutte le basi dei rettangoli uguali.

Il metodo di normalizzazione altera la base dei rettangoli. Se poi però faccio un sottotipo quadrato del rettangolo, normalizzare questi quadrati significa alterarne anche l'altezza.

Progettare male i contratti di `rectangle` e `square` porta a problemi del genere.

L18 - 04/12/2024 [cap 7.6]

S < T: S è un sottotipo di T *(classe, interfacce)*

< non è IS-A. Più che una questione specificatoria, ci interessa un riuso del codice.

Si pone la classe astratta come via intermedia tra le classi e le astratte. Vengono introdotte per il code reuse.

Classe astratta

È una classe nella cui specificazione aggiungiamo `abstract`:

```
public abstract class {
    - costruttori
    - metodi: concreti, astratti
    - attributi
}
```

L'effetto principale di asbtract sono:

- non si possono istanziare oggetti di una classe astratta. Sono una sorta di template da cui derivare ed estendere altre classi. Se si vuole fare un new bisogna prima quindi estenderla in una classe.
  - la conseguenza di asbtract è che non si possa istanziare, **ma non è il fine per cui bisogna usarlo!** Se si vuole una classe non istanziabile si fa un costruttore privato, abstract lo si utilizza per estendere poi in futuro quella classe astratta in una classe 'concreta'

Dentro la classe astratta ci sono:

- costruttori
- metodi
  - concreti (quelli già visti, graffa codice graffa), che hanno specificazione + implementazione
  - astratti, che hanno solo la specificazione
- attributi, servono a rappresentare quelle informazioni che si pensa che nella nostra gerarchia siano comuni a tutte le implementazioni (tutti i sottotipi no?)

$S_n < S_{n-1} < \cdots < \mathcal{A}$

Prima o poi questa classe astratta $\mathcal{A}$ va implementata, e in S_n, ossia la nostra classe pubblica, dovrà fare un override dei metodi della classe astratta.

```
public class S_n {
    @Override dei metodi astratti
    - Attributi
}
```

I sottotipi possono andare a vedere i valori, e perciò lo stato, del supertipo da cui derivano. Questo ovviamente se gli attributi del supertipo non sono privati.

Parte degli attributi della classe astratta che sta sopra quindi devono essere visibili: si usa `protected`. Questo modificatore protected rendere visibile gli attributi nei sottotipi derivati da quel supertipo.

- Gli attributi protetti sono parzialmente visibili a tutti i suoi discendenti, ma non fuori da questi.
- C'è un **compromesso** tra l'apprezzabile idea di riusare il codice, e il fatto che se si scrive il codice con roba protetta in un qualche modo si aprono un po' le porte. Se un utente prova ad estendere la classe astratta, può 'romperla'.
- Protected espone al rischio che un eventuale altro implementatore non abbia ben colto la mia documentazione. La sottoclasse deriva e completa la specifica del padre, e quindi chiunque la usi ne comprende la funzionalità tramite la documentazione, ma chi deve implementare la sottoclasse deve anche sapere come funzioni la classe astratta dentro le quinte?

Parte dello stato deve quindi essere protetto, altrimenti si farebbe una classe normale direttamente

Il proposito delle classi astratta (che favorisce il code reuse), è un buon proposito. Ma ragionare per estensione è comunque un grande rischio e bisogna usarle con grande attenzione.

Noi vogliamo avere insiemi che consentono di avere varianti nella rappresentazione. Vogliamo che siano sottotipi di un tipo (e non due tipi diversi) perché:

1. pensiamo che due varianti abbiano gran parte del contratto in comune (comodo, si fa meno specificazione),
2. per il poliformismo: in un oggetto di tipo T possiamo mettere dentro oggetti di tipo s_1 ed s_2.

Il processo di portare parti di codici uguali dalle varianti nel loro supertitpo viene definito **fattorizzazione**.

La fattorizzazione non deve per forza rispecchiare una gerarchia del mondo reale: se delle varianti presentano del codice in comune, utilizzo una classe astratta per "fattorizzare" il codice (code reusing)

AbstractIntSet

Chiamiamo gli insiemi normali `IntSet` e quelli ordinati `OrderedIntSet`

Nella classe astratta vogliamo mettere i metodi dell'IntSet delle scorse lezioni, vogliamo che sia iterabile e metteremo un attributo `size` che sia protetto.

AbstractIntSet.java

```java
package ...
import ...

/**
 * An {@code AbstractIntSet} is a mutable, unbounded set of integers.
 *
 * <p>A typical {@code AbstractIntSet} is \( S = \{x_1, \ldots, x_n \} \).
 */
public abstract class AbstractIntSet implements Iterable<Integer> {

  /** The size of the set. */
  protected int size;

  /*-
   * AF(size) -> a set with size elements.
   * RI: size >= 0
   */

  /** Creates an empty set. */
  protected AbstractIntSet() {
    size = 0;
  }

  /**
   * Adds the given element to this set.
   *
   * <p>This method modifies the object, that is: \( S' = S \cup \{ x \} \).
   *
   * @param x the element to be added.
   */
  public abstract void insert(int x);

  /**
   * Removes the given element from this set.
   *
   * <p>This method modifies the object, that is: \( S' = S \setminus \{ x \} \).
   *
   * @param x the element to be removed.
   */
  public abstract void remove(int x);

  /**
   * Tells if the given element is in this set.
   *
   * <p>Answers the question \( x\in S \).
```

```
46      *
47      * @param x the element to look for.
48      * @return whether the given element belongs to this set, or not.
49      */
50     public boolean isIn(int x) {
51       for (int e : this) if (e == x) return true;
52       return false;
53     }
54
55     /**
56      * Returns the cardinality of this set.
57      *
58      * <p>Responds with \( |S| \).
59      *
60      * @return the size of this set.
61      */
62     public int size() {
63       return size;
64     }
65
66     /**
67      * Returns an element from this set.
68      *
69      * @return an arbitrary element from this set.
70      * @throws NoSuchElementException if this set is empty.
71      */
72     public int choose() throws NoSuchElementException {
73       if (size == 0) throw new NoSuchElementException("Can't choose from an empty set");
74       return iterator().next();
75     }
76
77     /**
78      * ...
79      */
80     @Override
81     abstract public Iterator<Integer> iterator();
82
83     @Override
84     public boolean equals(Object obj) {
85       if (this == obj) return true;
86       if (!(obj instanceof AbstractIntSet other)) return false;
87       if (size != other.size) return false;
88       for (int e : this) if (!other.isIn(e)) return false;
89       return true;
90     }
91
92     @Override
93     public int hashCode() {
94       int result = 0;
95       for (int e : this) result += e; // This is a very bad hash function!
96       return result;
97     }
98
99     @Override
100    public String toString() {
101      StringJoiner sj = new StringJoiner(", ", "{", "}");
102      for (Integer e : this) sj.add(e.toString());
103      return sj.toString();
104    }
105  }
```

Possiamo dare una 'prima' implementazione nella abstract function utilizzando gli iteratori perché supponiamo che nelle "foglie della gerarchia" gli iteratori siano effettivamente implementati. L'iteratore viene infatti definito qui come metodo astratto.

- vedere `public int choose` e `public boolean isIn`

Il costruttore è protetto

IntSet.java

```java
package ...
import ...

/** A concrete {@code IntSet}s. */
public class IntSet extends AbstractIntSet {

    /** The set elements. */
    private final List<Integer> elements;

    /*-
     * AF(elements, size) = {elements.get(0), elements.get(1), ..., elements.get(size - 1)}
     * RI:
     *   - super.RI is true
     *   - elements != null and does not contain nulls.
     *   - elements does not contain duplicates.
     *   - size == elements.size()
     */

    /** Creates an empty set. */
    public IntSet() {
        super();
        elements = new ArrayList<>();
    }

    @Override
    public Iterator<Integer> iterator() {
        return Collections.unmodifiableCollection(elements).iterator();
    }

    @Override
    public void insert(int x) {
        if (!elements.contains(x)) {
            elements.add(x);
            size++;
        }
    }

    @Override
    public void remove(int x) {
        if (elements.remove(Integer.valueOf(x))) size--;
    }
}

```

- Nel costruttore di IntSet il `super()` non è necessario in quanto fatto da lui implicitamente.
- Bisogna fare l'override di tutti i metodi astratti del supertipo
- Nella RI bisogna aggiungere `size == elements.size()`, non basta che sia solo maggiore di 0 (che deriva dalla RI del padre). Questo perché ovviamente abbiamo un attributo in più: non c'è solo la lista di elementi, ma ora abbiamo anche la size ereditata dal padre (non è più ricavata come prima direttamente dalla lista, ora è un attributo), e quindi bisogna assicurarsi che questa size sia vera anche dopo la chiamata di metodi che vanno a modificarla indirettamente (togliendo a aggiungendo per esempio elementi)
  - *(Non è un po' lo stesso discorso che si faceva per le variabili cachate?)*
- **Attenzione** al boxing e all'unboxing di `elements.remove()`!!

OrderedIntSet.java

```java
package ...
import ...

/**
```

```java
 * A concrete sorted {@code IntSet}.
 *
 * <p>The iterator of this set returns the elements in ascending order.
 */
public class OrderedIntSet extends AbstractIntSet {

    /** The set elements. */
    private final List<Integer> elements;

    /*-
     * AF(elements, size) = { elements.get(0), elements.get(1), ..., elements.get(size - 1) }
     * RI:
     *   - super.RI
     *   - elements != null and does not contain nulls,
     *   - elements is sorted in ascending order.
     *   - size == elements.size()
     */

    /** Creates an empty set. */
    public OrderedIntSet() { this.elements = new ArrayList<>(); }

    /**
     * Returns the maximum element of this set.
     *
     * @return the maximum element of this set.
     * @throws NoSuchElementException if this set is empty.
     */
    public int max() throws NoSuchElementException {
        if (size == 0) throw new NoSuchElementException("An empty set has no maximum element.");
        return elements.getLast();
    }

    /**
     * Returns the minimum element of this set.
     *
     * @return the maximum element of this set.
     * @throws NoSuchElementException if this set is empty.
     */
    public int min() throws NoSuchElementException {
        if (size == 0) throw new NoSuchElementException("An empty set has no minimum element.");
        return elements.get(0);
    }

    @Override
    public Iterator<Integer> iterator() {
        return Collections.unmodifiableList(elements).iterator();
    }

    @Override
    public void insert(int x) {
        final int index = Collections.binarySearch(elements, x);
        if (index < 0) {
            elements.add(-index - 1, x);
            size++;
        }
    }

    @Override
    public void remove(int x) {
        if (elements.remove(Integer.valueOf(x))) size--;
    }
}
```

- il metodo `insert` è delicato perché bisogna che mantenga l'ordine. Quindi si cerca tramite la ricerca dicotomica il punto in cui aggiungere l'elemento (così si va a pagare un costo di $\log n$)
  - assolutamente evitare di aggiungere l'elemento alla fine e poi sortare!! costo di $n \log n$

Se guardo le due classi che abbiamo implementato, notiamo che molte cose sono rimaste comunque in comune, come per esempio il metodo `remove`. Non vogliamo però mettere questo metodo, come anche l'attributo `List`, perché ciò creerebbe un vincolo se si volesse gestire l'insieme astratto con altre strutture dati. Creiamo una classe astratta in mezzo:

ListBasedAbstractIntSet.java *[h18 refactored]*

```java
package ...
import ...

/** An partial implementation of {@code AbstractIntSet} based on {@link List}. */
public abstract class ListBasedAbstractIntSet extends AbstractIntSet {

    /** The set elements. */
    protected final List<Integer> elements;

    /*-
     * AF(elements, size) = { elements.get(0), elements.get(1), ..., elements.get(size - 1) }
     * RI:
     *   - super.RI
     *   - elements != null and does not contain nulls.
     */

    /** Creates an empty set. */
    public ListBasedAbstractIntSet() { this.elements = new ArrayList<>(); }

    @Override
    public Iterator<Integer> iterator() {
        return Collections.unmodifiableCollection(elements).iterator();
    }

    @Override
    public void remove(int x) {
        if (elements.remove(Integer.valueOf(x))) size--;
    }
}
```

IntSet e OrderedIntSet poi deriveranno da questa altra classe astratta `ListBasedAbstractIntSet`, a sua volta sottoclasse della sottoclasse madre `AsbtractIntSet`.

L19 - 06/12/2024 [cap 7.7-8]

Chi mette mano nello stato protetto della classe astratta, non può più utilizzare il principio della località del codice ma deve andare a leggere la documentazione.

Interfacce

Si mette sopra alle classi un'interfaccia, una specificazione del tutto priva di implementazione. Poiché a noi importa stabilire dei comportamenti e ci interessa agire su famiglie di oggetti con comportamenti simili, noi spostiamo i contratti sulle interfacce: non ci interessa poi come questi oggetti diversi vengano implementati.

> **Definition**
>
> A class is used to define a type and also to provide a complete or partial implementation. By contrast, an interface defines only a type. It contains only nonstatic, public methods, and all of its methods are abstract. It does not provide any implementation. Instead, it is implemented by a class that has an implements clause in its header.

Eredità multipla

L'ereditarietà multipla non è in genere possibile a meno che il tipo che sta sopra non sia un'interfaccia. Quindi una classe può avere sotto altre classi, ma una classe non può avere sopra tante classi. Una classe però può avere sopra più interfacce: le interfacce sono quindi un'alternativa interessante.

Nelle classi si hanno quindi contratti semplici che catturano proprietà omogenee (contratti discendenti dalle interfacce padri)-> le sue competenze non derivano da un solo genitore ma da più

Problema del dispatching [...]

Spesso nella pratica è meglio evitare di ereditare da classi e classi astratte, e conviene utilizzare le interfacce.

- Se larga parte del codice è in comune (e quindi si ha una grande duplicazione del codice), ha comunque senso fattorizzare in una classe astratta.

Metodi di Default

Le interfacce hanno solo dei metodi astratti (è solo specificazione dei metodi), ma non hanno attributi (e quindi non hanno uno stato, non può essere istanziata).

- Ci sono delle circostanze nelle quali un metodo può essere implementato senza fare riferimento allo stato, ma facendo riferimento ad altri metodi.
  - Si rimbalza l'implementazione di un metodo su altri metodi, e non su un attributo
- Posso costruire dei metodi a partire da altri: *posso costruire il metodo contains(x) per una lista a partire dai suoi metodi esistenti get(index) e size()*

```
Interface L {
    m(...);
    DEFAULT f(...) { f·m };
    DEFAULT g(...) { g·f };
}
```

Le interfacce quindi adesso non possiedono solo dei contratti, ma hanno anche un accenno di implementazione [?]

È un modo per modificare i contratti senza romperli: io aggiungo un nuovo metodo e ti fornisco una maniera di default di implementarlo utilizzando i metodi che già c'erano, ma se te la senti puoi anche farne un @Override

È un metodo che storicamente ha consentito di aggiungere le stream, le lambda e altre funzionalità al linguaggio Java.

Cosa scegliere quindi tra classe concreta, classe astratta e interfaccia?

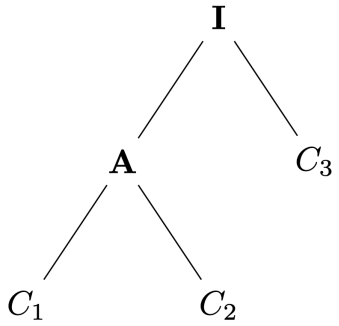
	C	A	I
stato	$privato$	$protected$	$non\ presente$
comportamenti	1. $Implementati_{stato}$: $pubblici$, $privati$	1. $Implementati_{stato}$: $pubblici$, $privati$, $protetti$ 2. $Astratti$: $pubblici$	1. $Astratti$ 2. $Default$

- **Stato**: una classe che sto facendo non ha bisogno di utilizzare lo stato, non ha senso utilizzare le prime due opzioni
- **Comportamenti**: una classe concreta i comportamenti possono essere sia pubblici (quelli che metto a disposizione), che privati, utili per fattorizzare il codice (sono metodi privati che servono a me implementatore per suddividere il codice, non devo pubblicarle)
  - In una classe astratta aggiungiamo dei comportamenti protetti. La classe astratta presenta anche dei metodi astratti (che sono pubblici).
    - Una classe astratta **deve** avere dei metodi astratti: non ha senso dichiarare che una classe sia parzialmente implementata e poi la si implementi tutta. Non bisogna utilizzarla come metodo per creare classi non istanziabili: in quel caso si usano delle classi con dei costruttori privati; servono per fornire delle classi parzialmente implementate.
  - Le prime due possono fare riferimento allo stato
  - Le interfacce, oltre ai metodi astratti, presentano anche dei metodi di default che però non fanno riferimento allo stato.

Cosa scegliere tra classe astratta e interfaccia con metodi di default?

- Se non ho bisogno di uno stato, non ha senso di utilizzare le classi astratte in quanto i suoi metodi dipendono dallo stato, privato o protected che sia. Si usano quindi le interfacce tramite i metodi di default.
  - **Avere lo stato protetto è un rischio per l'incapsulamento**: se do accesso ai miei figli al mio stato, c'è il rischio che loro possano fare casino. LA CLASSE ASTRATTA È PERICOLOSA: VA UTILIZZATA SE HA UN VANTAGGIO CONCRETO.

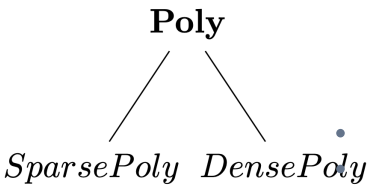
Spesso si raccoglie un'interfaccia il contratto (con anche metodi di default), sotto in una classe astratta si raccoglie un po' di stato e poi si implementano sotto la classe astratta i figli con quello stato in comune, e si implementa direttamente dall'interfaccia una classe che non condivide quello stato con la classe astratta.



Sovraccaricamento [...]

Sealed di classi astratte per evitare che vengano estese con altre classi concrete (+ metodi finali?)

Polinomi



L20 - 11/12/2024 [cap 7.9]

- Assegnamenti polimorfi

Dispatching dinamico, guardare appunti su

Questa relazione deve rispettare il principio di sostituzione della Liskov (LSP)

- Questo principio riguarda ciò che viene scritto nei contratti (e non l'implementazione): quando scriviamo il contratto di un sottotipo, se si fa l'override di un metodo del supertipo questo deve essere idoneo [...?]

Come verifichiamo il LSP? Se ho un tipo T con un suo contratto, come faccio prendendo un contratto di un metodo del tipo T a verificare che il suo override nel tipo S sia in sintonia?

LSP: Si supponga di avere un tipo T e un suo sottotipo S. Se nel testo del programma sostituisco ogni instanza t del tipo T con delle istanze s del tipo S, il programma deve funzionare e restituire il medesimo risultato.

- Questo deve essere garantito per fare in modo che i contratti del padre valgano quando sostituiti [?]

Si introducono tre lemmi (rules) che implicano l'LSP (poiché sono più facili da dimostrare dell'LSP direttamente):

1. **Regola delle segnature** (di natura sintattica): il sottotipo s deve presentare tutti i metodi che presenta t con la stessa segnatura. È talmente importante che è adottata automaticamente dal compilatore (quando manca un `@Override` ti avverte).
   - Non è fondamentale che la segnatura sia completamente identica: il sottotipo può restringere il tipo ritornato, ritornando un sottotipo (non può invece restituire il supertipo).
   - La segnatura del metodo del sottotipo può inoltre avere meno eccezioni oppure ritornare eccezioni più specifiche.
   - Non devono avere quindi segnature identiche ma compatibili.
2. **Regola dei metodi** (di natura semantica): si occupa di vedere metodo per metodo che il comportamento del figlio sia come quello del padre. Il compilatore ovviamente non può aiutare.
3. **Regola delle proprietà** (di natura semantica): qualunque proprietà venga manifestata dal supertipo deve essere vera se si sostituisce la 'sottoistanza' s alla 'superistanza' t. Queste proprietà non sono descritte nei contratti [?]
   1. Nell'LSP la quantificazione è su tutti i possibili sottotipi, questo è invece sulle proprietà: io ho i due contratti e devono garantire le stesse proprietà: queste due proprietà non dicono la stessa cosa.

Regola dei metodi

Si deve basare sui contratti del supertipo e del sottotipo. Si fa metodo per metodo. Si legge il contratto del padre, il suo corrispondente nel figlio e si cerca di capire se vadano bene. Se sono uguali il problema non si pone, se sono un po' diversi bisogna rifletterci sopra.

- Il tipo T ha un metodo m: questo metodo ha delle pre-condizioni_T e delle post-condizioni_T
- Il sottotipo S ha lo stesso metodo m: questo metodo ha delle pre-condizioni_S e delle post-condizioni_S

Se queste pre-condizioni e post-condizioni sono uguali, il problema non si pone. Ma se volessimo cambiare il metodo di s, come facciamo a lasciarlo più libero di t?

Pre-condizioni
Noi vorremmo lasciare s libero di essere più largo di t: se qualcosa funzione per il supertipo, deve funzionare anche per il sottotipo, ma questo deve essere libero di poter fare anche più cose di t.

- **Allargamento del comportamento**: è necessario che le precondizioni di T implicano le precondizioni di S; S è quindi più 'largo' sull'input: $PRE_T \Rightarrow PRE_S$

Post-condizioni
Una volta che il sottotipo S ha fatto il suo lavoro, sarebbe conveniente che quello vada anche bene per T:
$POST_S \Rightarrow POST_T$

- Se però S si allarga e prende più input di T, è difficile (se non possibile) che T abbia lo stesso risultato di S su quell'input che non potrebbe prendere. È troppo forte come implicazione.
- Allora se S prende in input istanze di T, si deve comportare come T, ma se prende in input istanze al di fuori di T, si può comportare diversamente: $PRE_T \wedge POST_S \Rightarrow POST_S$
  - Se il padre non ha una restrizione, allora anche il figlio deve poter lavorare su tutti i possibili input e restituire la stessa cosa: nei metodi totali $PRE_T = PRE_S = \varnothing;\ POST_S \Rightarrow POST_T$

[Nella pratica è la regola più evidentemente violata]

Esempio 1

Si supponga di avere una forma di collezione in cui un metodo `addZero()` abbia come contratto:

- nel tipo T: pre: se $s \neq \varnothing$ post: $s' = \{0\} \cup s$
- nel sottotipo S: pre: - post: $s' = \{0\} \cup s$

Questo soddisfa la regola dei metodi. Anche fare:

- nel sottotipo S: pre: - post: $s' = \{0\} \cup s$; se $s = \varnothing$ allora $s = \{1970\}$

rispetterebbe la regola dei metodi poiché il padre non fornisce indicazioni nelle pre-condizioni per l'insieme vuoto.

Bisogna controllare quindi che: $PRE_T \Rightarrow PRE_S,\ PRE_T \wedge POST_S \Rightarrow POST_T$

Esempio 2

Si supponga di avere un set in cui per il metodo `add(x)`:

- nel tipo `set` T: pre: - post: $s' = s \cup \{x\}$
- nel sottotipo `set+long` S: pre: - post: $s' = s \cup \{x\};\ L' = L + [x]$

Anche qui la regola dei metodi è rispettata. Questa regola non nega al figlio di poter fare più cose del padre. Semplicemente nelle competenze del padre si deve comportare allo stesso modo.

Immutabilità

IMPORTANTE: se si ha un tipo T immutabile e si ha un suo sottotipo S, questo sottotipo deve essere per forza immutabile? No, il fatto che T sia immutabile non implica che S lo sia per forza, **ma le mutazioni di S non devono essere osservabili tramite i metodi di T**, ma possono esserlo tramite i suoi metodi proprietari.

Equality [cap 7.9.3]

Che relazione c'è tra l'uguaglianza e l'idea di fare dei sottotipi?

Prima di tutto l'equals deve continuare ad essere una relazione d'equivalenza (proprietà riflessiva, simmetrica, transitiva)

IMPORTANTE: Se a T si aggiunge equals e ad S si aggiunge dello stato (che si vuole ovviamente esercitare), non c'è modo di aggiustare l'equals su S.

- non si riesce a garantire sia le 3 proprietà della relazione d'equivalenza che l'LSP

L'aggiunta di un equals previene l'estendibilità: non si può aggiungere nel sottotipo dello stato (stato che sia considerato nell'equals che si fa la comparazione, cosa che però è naturale)

- se ho un punto t bidimensionale, nel punto tridimensionale s si utilizza anche z per fare l'equals, se si implementa `equals` naturalmente.

Guardare qui: https://prog2-unimi.github.io/notes/UEE.html

Questo problema si risolve mettendo dentro S un T e aggiungendo poi quello che ci serve per S: **composizione**

L21 - 13/12/2024 [cap 7.10-11]

Composizione

Se l'obiettivo primario è il code reuse, l'ereditarietà (la gerarchia) non è la strada migliore: ci espone a tanti problemi tra cui l'LSP.

- Si usa l'ereditarietà se si crede davvero che questa serva come metodo di gerarchia. Se ci serve la sostituibilità è la scelta giusta.

Bisogna quindi evitare di usare extends nel nome della classe, non si vogliono utilizzare dei sottotipi per riutilizzare il codice.

Si fa il tipo T con i suoi metodi, si sviluppa poi il tipo S a parte, con nessuna relazione di sottotipo. Per mantenere il code reuse, possiamo fare in modo che il codice di S riutilizzi il codice di T assorbendo nei suoi attributi degli elementi $t_0, t_1, \cdots$

- La composizione inoltre ci permette di mettere in un'unica classe istanze di più classi, cosa che non si può fare tramite ereditarietà.

Un'esigenza: dentro S dove viene dichiarato T, se si vuole che S esibisca i metodi di T bisogna fare in modo che attraverso la **delega** S appunto deleghi il comportamento di certi suoi metodi ai metodi di T.

- non bisogna rilevare nel contratto che in S c'è T, rimane una scelta implementativa
- aumenta il numero di chiamate sì, ma l'efficienza di questo tipo non è l'obiettivo del linguaggio Java

Questa cosa noi la facciamo già: nelle nostre classi di norma usiamo tra i nostri attributi delle liste, e per esempio il metodo `size` di `intSet` delega il risultato a `list.size()`.

La documentazione Javadoc va però tutta rifatta. La parte contrattuale non viene riportata.

L'idea della delega is the way to go per fare code reuse

Esempio

https://prog2-unimi.github.io/notes/CED.html

Supponiamo di avere una classe `Adder` in grado di tenere traccia di somme tra interi (che possono essere svolte considerando un numero per volta oppure prelevando gli addendi da una lista), il cui codice è

```
package ...

/*** .. */
public class Adder {

    /** .. */
    private int result = 0;
    /*-
     * AF
     * RI
     */

    /** .. */
    public void add(int x) {
        result += x;
    }

    /** .. */
    public void add(List<Integer> l) {
        for (int x : l) add(x);
    }

    /** .. */
    public int result() {
        return result;
    }
}
```

Supponiamo ora di voler costruire una nuova classe in grado di tener traccia non solo della somma, ma anche dei valori di volta in volta sommati.

Dato che tale funzionalità sembra estendere quella della classe appena introdotta, sembra naturale procede per estensione, definendo la classe `LogAdderExt` come segue:

```
public class LogAdderExt extends Adder {
    private final List<Integer> seen = new ArrayList<>();

    ...

    /** POST_S sono POST_T + aggiunge al log */
    @Override
    public void add(int x) {
        seen.add(x);
        super.add(x);
    }

    /** POST_S sono POST_T + aggiunge al log */
    @Override
    public void add(List<Integer> l) {
        seen.addAll(l);
        super.add(l);
```

```
    }

    public List<Integer> log() {
        return List.copyOf(seen);
    }
}
```

[...] Problema di dispatching [...]

[guardare link o appunti lezione 21]

Interfacce e Deleghe

Lessicalmente l'`AdderLogger` non è un sottotipo di `Adder`, ma rispetta comunque l'LSP. Non si riesce più a sostituire `Adder` con `AdderLogger`. Questo problema si risolve tramite le interfacce.

- Catturiamo le proprietà e le competenze che vogliamo avere in un'interfaccia, e poi utilizziamo la delega nei derivati dell'interfaccia per riutilizzare il codice.

[...]

L22 - 18/12/2024 [cap 8]

Polimorfismo e generici

https://dev.java/learn/generics/

> **Definition**
>
> Polymorphism generalizes abstractions so that they work for many types. It allows us to avoid having to redefine abstractions when we want to use them for more types; instead, a single abstraction becomes much more widely useful.
>
> A procedure or iterator can be polymorphic with respect to the types of one or more arguments. A data abstraction can be polymorphic with respect to the types of elements its objects contain.
>
> - Liskov

Polymorphism is expressed through hierarchy. Certain arguments are declared to belong to some supertype, and then the actual arguments can be objects belonging to subtypes of that type. That supertype is often `object`.

Tre forme di poliformismo:

- **Subtyping**: se ho S sottotipo di T posso pensare di ragionare sulle competenze di T ma poi utilizzare l'implementazione di S
- **Ad hoc**: nel linguaggio si può definire un metodo con il medesimo nome di un altro, a patto che non abbiano la stessa signature (detta anche **overloading**) (esempio: metodo `add` che funziona per stringhe, insiemi, ecc -> in base al parametro Java capisce quale metodo add adoperare)
- **Generici**: $G<T>$

Devo fare una `IntList` e una `StringList`, come evito di duplicare il codice? Creo una `ObjectList`.

- Sorge il problema della type safety. Avere un contenitore poliformo significa perdere la capacità di avere un contenitore omogeneo.

In un codice generico, nella dichiarazione di un parametro forniamo antecedentemente il suo tipo. Quel codice è espresso parametricamente rispeto ad un tipo <T> *(esattamente come si fa in una funzione)*

Ciò ci libera dalla necessità di duplicare codice, e ci mette nelle circostanze che quando verrà adoperata, quel parametro per tipo viene sostituito da un tipo concreto.

Limiti di questo approcio: *[da controllare per bene ognuno di questi 4 punti]*

- T non può essere primitivo: non c'è modo di fare una lista di `Int`
  - Per risolvere questo problema si utilizza il **boxing/unboxing** (ad esempio con tipi wrapper come Integer al posto di int)
- Non si può fare `new T()` -> nei generics non è possibile istanziare direttamente T con new poiché il tipo concreto non è noto durante la compilazione.
- Non è possibile dichiarare oggetti di tipo generico: `T[] x = new T[10]; //error` -> questa cosa non si può fare.
  - Questo è legato al concetto di **type erasure** di Java: durante la compilazione, i generics vengono trasformati nel loro tipo grezzo (Object nella maggior parte dei casi).
    Di conseguenza, il compilatore non può garantire la **type safety** (sicurezza del tipo), poiché non è in grado di verificare il tipo reale in fase di runtime.
- Generici e subtyping sono complicati da comprendere.

  ```java
  List<Integer> integerList = new ArrayList<>();
  List<Number> numberList = integerList; // Errore!
  ```

  Anche se `Integer` è un sottotipo di `Number`, `List<Integer>` **non è un sottotipo di** `List<Number>`. Questo perché se il compilatore permettesse tale assegnazione, si potrebbero inserire valori in `numberList` che violerebbero la type safety:

  ```java
  numberList.add(3.14); // Questo sarebbe consentito, ma è un Double, non un Integer
  Integer value = integerList.get(0); // Errore a runtime!
  ```

  - Si usando quindi **wildcard** e **bound** per aggirare il problema. [...]

[Relazione di tipo dei generici] ??

Usare un generico: $G<T>$; $G<>$ Inferenza dei tipi
Srivere un generico: Nelle classi $G<T>$, nei metodi <T> $f(\cdots)$

Esempio metodi generici

Metodo `int countNull()` che restituisce un intero [...]

Esempio classe generica
[...]

L23 - 20/12/2024 [cap 8]

Metodi e classi generici.

Vogliamo scrivere funzioni che possono essere applicate su una famiglia di realizzazioni di quel tipo. Purtroppo anche se S è un sottotipo di T non è vero che se si fa un generico con S quello è un sottotipo di un generico fatto con T:
$S < T \quad \not\Rightarrow \quad G<S> < G<T>$ *(il motivo per cui non si può fare è spiegato sotto)*

Generici $<S> \not< \text{List}<T>$ però **Array** $S[\,] < T[\,]$

- L'array non solo è capace di fare un controllo di natura statica (come anche i generici), ma è anche capace di fare una serie di controlli di natura dinamica (controlli che il compilatore non può fare). Tra queste competenze di natura dinamica ci sono i controlli dei tipi, cosa che i generici non sono capaci di fare.
- I generici non hanno idea del tipo di cui sono effettivamente generici. In fase di compilazione qualunque tipo generico si realizza come tipo oggetto -> il meccanismo dei generici è stato realizzato in modo tale per cui i generici fanno controlli statici ma nessuna controparte a run-time. (Problema di backwards compatibility)

Si può invece fare: $S<U> \ < \ T<U>$

Bisogna trovare il modo di scrivere $f(T\ t)$ che ci permetta di fare la chiamata con qualcosa di diverso $f(\text{new } S())$

- Vogliamo la sostituibilità nella chiamata
- Per esempio da $List<\dots>$ fare $List < Numb >;\ List < float >,\ \cdots$

Si introducono quindi le **Wildcard ?**: mettendo ? al posto del parametro che indica che lì si possa mettere qualsiasi tipo.

- $G<?>$ è un supertipo di tutti i vari G: $G < int >;\ G < \cdots >$
- La wildcard consente di fare la sostituzione, ma a cosa serve in pratica? Se si vuole fare un metodo che cerca i null in una lista, si può fare un metodo che accetta una lista di un qualsiasi tipo

```java
static int countNull(List<?> list) {
    int num = 0;
    for (Object i : list) if (i == null) num++;
    return num;
}
```

  - Si elimina la necessità di definire nel metodo un tipo

Anche queste wildcard hanno i **bound**: $G<?\text{ extends } T>$

- $\forall S < T \quad G<S>$ è sottotipo di $G<?\text{ extends } T>$

```java
// interfaccia Figure

static void draw(List<? extends Figure> list) {
    for (Figure f : list) f.draw()
}
// Senza extends questa cosa non si può fare poiché Object non possiede la competenza draw()
```

Si può fare anche nell'altro verso: $G<?\text{ super } S>$ con $S < T \quad G<T>$ è sottotipo del primo generico *[to check]*

- In G si può mettere qualsiasi cosa che sia un supertipo di S

La logica può essere riassunta con **P E C S**: producer extends consumer super

`Pair <T, U>` **-> check example in lesson 23 handouts**

Ordinamenti

G extends Comparable<C>

L24 - 08/01/2025

Collections

Sono uno strumento di utilità. Si fanno con poco sforzo gestioni di strutture dati avanzate (-> Digraph, grafi orientati)

- Sono un buon esempio di utilizzo di generici.

La programmazione che fa uso delle stream e delle funzioni lambda noi non la copriremo perché: le lambda sono estremamente complesse ed è complesso cogliere gli aspetti di type inference legati alle lambda.

Le collection rispondono alla necessità di avere nelle API strutture dati per la memorizzazione e il reperimento di dati e per la trasmissione di informazioni -> `java.util.Hashtable`

Ci sono una serie di contratti che prescindono dall'implementazione.

Core Collection interfaces

Ci sono due famiglie di interfacce nel framework collection: `Collection` e `Map`

- Da `Collection` derivano `Set` (da cui deriva a sua volta `SortedSet`) e `List`
- Da `Map` deriva `SortedMap`

The `Set` interface adds no methods to `Collection`, but adds a stipulation: there are no duplicate elements. It also mandates equals and hashCode calculation.

- `public interface Set<E> extends Collection<E> {}`

The `Set` class has meaning only if the objects that we're putting into it have valid `equals` methods

The `List` interface adds an index to the `Collection` interface and various methods:

```
public interface List<E> extends Collection<E> {
    E get(int index);
    E set(int index, E element); // Optional
    ... methods
}
```

- List idioms [...]

The `Map interface`

```
public interface Map<K,V> {
    int size();
    boolean isEmpty();
    ... methods
}
```

Implementations

`List` -> `ArrayList` and `LinkedList`

`Map` -> `HashMap` and `TreeMap`

`Set` -> `HashSet` and `TreeSet`

A differenza di `Vector` and `HashTable` queste implementazioni non sono thread-safe; sia valori che chiavi possono contenere elementi `Null`; gli iteratori hanno un contratto per il quale se durante l'iterazione vengono fatte delle modifiche durante la nostra iterazione, questo contratto cerca di mandare un errore il prima possibile (fail-fast iterator).

Ci sono versioni unmodifiable che delegano tutti i modi osservazionali e che intercettano i metodi mutazionali

- Non fare le classi taggate