



---

# **PROGRAMMAZIONE 2**

## **7. Astrazioni sui dati: specifica di tipi di dato astratti in Java**

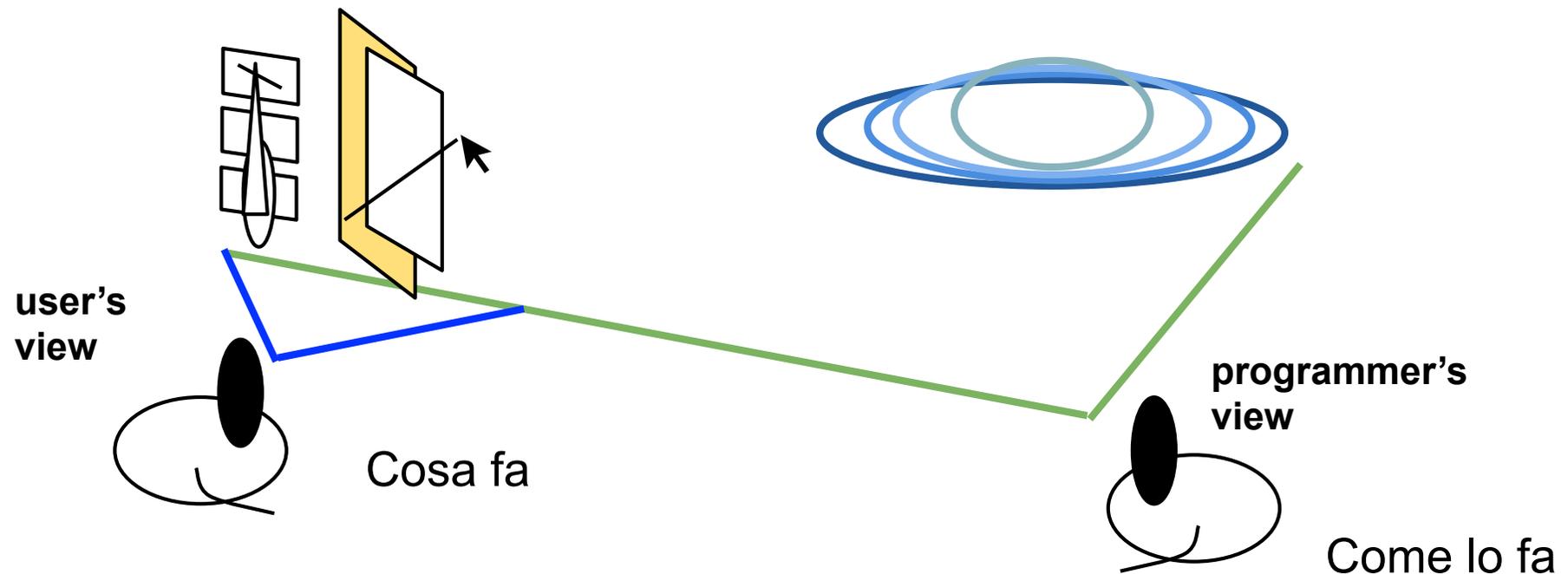
# Forme di astrazione

---

- ***Procedural abstraction***: separazione delle proprietà logiche di una azione computazionale dai suoi (della azione) dettagli implementativi
- ***Data abstraction***: separazione delle proprietà logiche dei dati dai dettagli della loro (dei dati) rappresentazione

# Modularità

---



Quando si implementa un programma  
bisogna considerarsi un utente delle  
altre parti dalle quali il programma  
dipende

# Contratti di uso

---

- Un “**contratto**” è l’insieme dei vincoli di uso concordati tra l’utente di un modulo software e chi implementa effettivamente il modulo
  - è una descrizione delle aspettative delle due parti
- Perché i contratti sono utili?
  - *separation of concern*: i dettagli implementativi sono mascherati all’utente che vede solo le funzionalità offerte
  - **facilitano la manutenzione e il ri-uso del software**

# Interfacce come contratti?

---

- In Java la nozione di **interfaccia** permette di definire esplicitamente il “confine” tra cliente e implementatore

```
public interface IntSet {  
    public void insert(int elem);  
    public void remove(int elem);  
    public boolean isIn(int elem);  
    public boolean isEmpty( );  
    public void add(int elem);  
    ...  
}
```

- Le interfacce in Java definiscono la sintassi e il tipo dei metodi ma non definiscono il comportamento e l’effetto atteso dell’esecuzione di un metodo
  - *sintassi e tipi forniscono una informazione limitata ai clienti*

# Java interface++

---

- Estendiamo la **nozione delle interfacce in Java inserendo informazioni sul codice effettivo di implementazione dei metodi**
- Ma il codice...
  - è troppo complicato: un cliente non deve sapere nel dettaglio come il metodo è **implementato** ma deve solamente avere una **astrazione del comportamento**
  - può cambiare nel tempo e il cliente non è interessato alle modifiche puntuali

# Abstract data type

---

- Un *abstract data type* (ADT) è una collezione di elementi il cui comportamento logico è definito da un dominio di valori e da un insieme di operazioni su quel dominio
- ADT
  - Nome
  - Valori
  - Operazioni
  - Semantica delle operazioni



# Esempio (ADT via Assiomi)

---

- Nome: **Stack (of Item)**
- *Item* (parametro) il tipo degli elementi che possono essere inserito nello stack

# Stack: operatori

---

- **Constructors**
  - *create: unit -> Stack*
- **Mutators**
  - *push: (Stack, Item) -> Stack*
  - *pop: Stack -> Stack*
- **Accessors**
  - *top: Stack -> Item*
  - *empty: Stack -> boolean*
  - *size: Stack -> int*
- **Destructors**
  - *destroy: Stack -> unit*

[visione imperativa]

# Stack: assiomi

---

Assiomi: regolano il comportamento delle operazioni (semantica intesa della astrazione)

- $\text{top}(\text{push}(s, x)) = x$
- $\text{pop}(\text{push}(s, x)) = s$
- $\text{empty}(\text{create}()) = \text{true}$
- $\text{empty}(\text{push}(s, x)) = \text{false}$

[visione imperativa]

# Stack: assiomi

---

- $s.size()$ ,  $s.empty()$  e  $s.push(t)$  sono sempre definite
- $s.pop()$  e  $s.top()$  sono definite sse  $s.empty() = false$
- $s.empty()$ ,  $s.size()$  e  $s.top()$  lasciano  $s$  immutato
- $s.empty() = true$  sse  $s.size() = 0$
- se  $s1$  è lo stack dopo  $s.push(t)$ 
  - lo stack dopo  $s1.pop()$  è  $s$
  - $s1.top() = t$
  - $s1.size() = s.size() + 1$
- se  $s1$  è lo stack dopo  $s.pop()$ 
  - $s1.size = s.size() - 1$

# A cosa servono gli assiomi?

---

- A provare teoremi!!

se  $s1$  è lo stack ottenuto dopo avere eseguito  $k$  operazioni di push a partire da uno stack  $s$ , allora  
 $s1.size() = s.size() + k$



# Operazioni parziali

---

- Precondition of `s.pop( )`
  - `s.empty( ) = false`
- Precondition of `s.top( )`
  - `s.empty( ) = false`

# ADT: visione costruttiva

---

- Le operazioni sono caratterizzate da una **pre-condizione** e da una **post-condizione**
- **Precondition**: *formula logica che caratterizza le proprietà e il valore degli argomenti*
- **Postcondition**: *formula logica che caratterizza il risultato calcolato dall'operazione rispetto al valore degli argomenti*

# Esempio

---

- `s.push(t)` porta a uno stack `s1`  
**Precondition:** `s` è una istanza valida di `Stack`  
&& `s` non è full
- **Postcondition:** `s1` è una istanza valida di `Stack`  
&& `s1` coincide con `s` esteso con `t`  
come elemento top

# Data abstraction **via specifica**

---

- **Con la specifica, astraiano dall'implementazione del tipo di dato**
- Avendo gli oggetti insieme alle operazioni, l'astrazione diventa possibile
  - la rappresentazione è nascosta all'utente esterno, mentre è visibile all'implementazione delle operazioni
  - se una rappresentazione viene modificata, devono essere modificate le implementazioni delle operazioni, ma non le astrazioni che la utilizzano
    - ✓ è il tipo di modifica più comune durante la manutenzione

# Gli ingredienti di una specifica

---

- **Java (parte sintattica della specifica)**
  - classe o interfaccia
    - ✓ per ora solo classi
  - nome per il tipo (la classe)
  - operazioni
    - ✓ metodi di istanza, incluso il/i costruttore/i
- **la specifica del tipo**
  - fornita dalla clausola **OVERVIEW** che descrive i valori astratti degli oggetti e alcune loro proprietà
    - ✓ per esempio la modificabilità
- per il resto la specifica è una specifica dei metodi
  - strutturata tramite **pre-condizioni (clausola requires)** e **post-condizioni (clausola effects)**

# Formato della specifica

---

```
public class NuovoTipo {
    // OVERVIEW: Gli oggetti di tipo NuovoTipo
    // sono collezioni modificabili di ...

    // costruttori
    public NuovoTipo( )
        // REQUIRES
        // EFFECTS: ...

    // metodi
    // specifiche degli altri metodi
}
```

# IntSet 1

---

```
public class IntSet {
    // OVERVIEW: un IntSet è un insieme modificabile
    // di interi di dimensione qualunque
    //costruttore
    public IntSet( )
        // EFFECTS: inizializza this all'insieme vuoto
    // metodi
    public void insert(int x)
        // EFFECTS: aggiunge x a this
    public void remove (int x)
        // EFFECTS: toglie x da this
    public boolean isIn(int x)
        // EFFECTS: se x appartiene a this ritorna true,
        // false altrimenti
    ...
}
```



# IntSet 2

---

```
public class IntSet {
    ...
    // metodi
    ...
    public int size( )
        // EFFECTS: ritorna la cardinalità di this
    public int choose( ) throws EmptyException
        // EFFECTS: se this è vuoto, solleva
        // EmptyException, altrimenti ritorna un
        // elemento qualunque contenuto in this
}
```

# IntSet: analisi

---

```
public class IntSet {  
    // OVERVIEW: un IntSet è un insieme modificabile  
    // di interi di dimensione qualunque
```

- I **valori astratti** degli oggetti della classe sono descritti nella specifica in termini di concetti noti
  - gli insiemi matematici
  - l'uso di una notazione matematica (in seguito)
- Gli stessi concetti sono usati nella specifica dei metodi
  - aggiungere, togliere elementi
  - appartenenza, cardinalità

# IntSet: analisi

---

```
public class IntSet {  
    // OVERVIEW: un IntSet è un insieme modificabile  
    // di interi di dimensione qualunque  
    //costruttore  
    public IntSet( )  
        // EFFECTS: inizializza this all'insieme vuoto
```

- Un solo **costruttore (senza parametri)**
  - inizializza this (l'oggetto nuovo)
  - non è possibile vedere lo stato dell'oggetto tra la creazione e l'inizializzazione

# IntSet: analisi

---

```
public void insert(int x)
    // EFFECTS: aggiunge x a this
public void remove (int x)
    // EFFECTS: toglie x da this
```

- **Modificatori**
  - modificano lo stato del proprio oggetto
  - notare che **né insert né remove sollevano eccezioni**
    - ✓ se si inserisce un elemento che c'è già
    - ✓ se si rimuove un elemento che non c'è

# IntSet: analisi

---

```
public int size( )
    // EFFECTS: ritorna la cardinalità di this
public int choose( ) throws EmptyException
    // EFFECTS: se this è vuoto, solleva
    // EmptyException, altrimenti ritorna un
    // elemento qualunque contenuto in this
public boolean isIn(int x)
    // EFFECTS: se x appartiene a this ritorna true,
    // false altrimenti
```

- Osservatori
  - non modificano lo stato del proprio oggetto: choose può sollevare un'eccezione (se l'insieme è vuoto)
    - EmptyException può essere unchecked, perché l'utente può utilizzare size per evitare di farla sollevare
    - choose è sotto-determinata (implementazioni corrette diverse possono dare risultati diversi)



# Specifica di un tipo “primitivo”

---

- Le specifiche sono ovviamente utili **anche** per capire e usare correttamente i tipi di dato “primitivi” di Java
- Vedremo, come esempio, il caso dei **vettori**
  - Vector
  - array dinamici che possono crescere e ridursi
  - sono definiti nel package `java.util`

# Vector 1

---

```
public class Vector {  
    // OVERVIEW: un Vector è un array modificabile  
    // di dimensione variabile i cui elementi sono  
    // di tipo Object: indici tra 0 e size - 1  
    // costruttore  
    public Vector( )  
        // EFFECTS: inizializza this a vuoto  
    // metodi  
    public void add(Object x)  
        // EFFECTS: aggiunge una nuova posizione a this  
        // inserendovi x  
    public int size( )  
        // EFFECTS: ritorna il numero di elementi di this  
    ...  
}
```

# Vector 2

---

```
public Object get(int n) throws IndexOutOfBoundsException
    // EFFECTS: se  $n < 0$  o  $n \geq this.size$  solleva
    // IndexOutOfBoundsException, altrimenti ritorna
    // l'oggetto in posizione n in this
public void set(int n, Object x) throws
IndexOutOfBoundsException
    // EFFECTS: se  $n < 0$  o  $n \geq this.size$  solleva
    // IndexOutOfBoundsException, altrimenti modifica this
    // sostituendovi l'oggetto x in posizione n
public void remove(int n) throws IndexOutOfBoundsException
    // EFFECTS: se  $n < 0$  o  $n \geq this.size$  solleva
    // IndexOutOfBoundsException, altrimenti modifica this
    // eliminando l'oggetto in posizione n
```

# Vector: analisi

---

```
public class Vector {  
    // OVERVIEW: un Vector è un array modificabile  
    // di dimensione variabile i cui elementi sono  
    // di tipo Object: indici tra 0 e size - 1  
    ...  
}
```

- Gli oggetti della classe sono descritti nella specifica in termini di concetti noti: gli array
- Gli stessi concetti sono anche usati nella specifica dei metodi
  - indice, elemento identificato dall'indice
- Il tipo è modificabile come l'array
- Notare che gli elementi sono di tipo Object
  - non possono essere int, bool o char

# Vector: analisi

---

```
public class Vector {  
    // OVERVIEW: un Vector è un array modificabile  
    // di dimensione variabile i cui elementi sono  
    // di tipo Object: indici tra 0 e size - 1  
    // costruttore  
    public Vector( )  
        // EFFECTS: inizializza this a vuoto  
    ...  
}
```

- Un solo costruttore (senza parametri)
  - inizializza this (l'oggetto nuovo) a un "array" vuoto

# Vector: analisi

---

```
public void add(Object x)
    // EFFECTS: aggiunge una nuova posizione a this
    // inserendovi x
public void set(int n, Object x) throws
    IndexOutOfBoundsException
    // EFFECTS: se n<0 o n>= this.size solleva
    // IndexOutOfBoundsException, altrimenti modifica
    // this sostituendovi l'oggetto x in posizione n
public void remove (int n) throws
    IndexOutOfBoundsException
    // EFFECTS: se n<0 o n>= this.size solleva
    // IndexOutOfBoundsException, altrimenti modifica
    // this eliminando l'oggetto in posizione n
```

- Sono modificatori
  - modificano lo stato del proprio oggetto
  - set e remove possono sollevare un'eccezione unchecked

# Vector: analisi

---

```
public int size( )
    // EFFECTS: ritorna il numero di elementi di this
public Object get(int n) throws IndexOutOfBoundsException
    // EFFECTS: se n<0 o n>= this.size solleva
    // IndexOutOfBoundsException, altrimenti ritorna
    // l'oggetto in posizione n in this
public Object lastElement()
    // EFFECTS: ritorna l'ultimo oggetto in this
```

- Sono osservatori
  - non modificano lo stato del proprio oggetto
  - get può sollevare un'eccezione primitiva unchecked

# Aspetti metodologici

---

- Per prima cosa si definisce la **specificità**
  - “scheletro” formato da header, overview, pre- e post-condizioni di tutti i metodi
  - mancano la **rappresentazione degli oggetti e il codice dei corpi dei metodi...**
    - ✓ che possono essere sviluppati in un momento successivo e indipendentemente dallo sviluppo dei “moduli” che usano il nuovo tipo di dato
    - ✓ ed è molto importante riuscire a differire le scelte relative alla rappresentazione

# Un cliente di IntSet

---

```
public static IntSet getElements(int[ ] a) throws  
    NullPointerException {  
    // EFFECTS: se a=null solleva NullPointerException,  
    // altrimenti restituisce un insieme che contiene  
    // tutti e soli gli interi presenti in a  
  
    IntSet s = new IntSet( );  
    for (int i = 0; i < a.length; i++)  
        s.insert(a[i]);  
    return s;  
}
```

- Scritta solo conoscendo la **specifica di IntSet**
  - non accede all’implementazione
    - magari non esiste ancora, ma pure se ci fosse non potrebbe “vederla”
  - costruisce, accede e modifica l’oggetto solo attraverso i metodi

# Concludendo

---

Se aggiungiamo il codice dei metodi ben tipati alla specifica dei metodi

- la **specifica può essere compilata**
- possono essere compilate implementazioni di moduli che la utilizzano (errori rilevati subito dall'analisi statica)
- possono essere progettati i test di analisi
- **esempio: Junit è basato su questa idea**