



AA 2016-2017

## **8. Astrazioni sui dati: implementazione di tipi di dato astratti in Java**

# Abstract Data Type



- ✎ Insieme di **valori**
- ✎ Un insieme di **operazioni** che possono essere applicate in modo uniforme ai valori
- ✎ **NON** è caratterizzato dalla rappresentazione dei dati
  - La rappresentazione dei dati è **privata**, senza effetto sul codice che utilizza il tipo di dato
  - La rappresentazione dei dati è **modificabile** nel caso di ADT **mutabili**

# Specificare un ADT



- ✎ La specifica di un ADT è un contratto che definisce
  - **valori, operazioni** in termini di nome, parametri tipo, effetto osservabile
- ✎ *Separation of concerns*
  - Progettazione e realizzazione del ADT
  - Progettazione applicazione che utilizza ADT

# Specifiche con Javadoc



- ✎ Javadoc: non esiste una notazione formale, ma solamente un insieme di convenzioni
- ✎ Javadoc: le principali convenzioni
  - Segnatura (tipo) dei metodo
  - Descrizione testuale del comportamento atteso (astrazione sul comportamento)
  - Dalla documentazione di Java
  - **@param**: description of what gets passed in
  - **@return**: description of what gets returned
  - **@throws**: exceptions that may occur

# Esempio: String.contains



*public boolean* **contains**(CharSequence *s*)

**Returns true if and only if this string contains the specified sequence of char values.**

*Parameters:*

**s- the sequence to search for**

*Returns:*

**true if this string contains s, false otherwise**

*Throws:*

**NullPointerException**

*Since:*

**1.5**

# Il nostro formato della specifica



```
public class NuovoTipo {  
    // OVERVIEW: Gli oggetti di tipo NuovoTipo  
    // sono collezioni modificabili di ...  
  
    // costruttori  
    public NuovoTipo()  
        // EFFECTS: ...  
  
    // metodi  
    // specifiche degli altri metodi  
}
```

# Alternativa equivalente



- 🦋 La *precondizione del metodo*: dichiara i vincoli che devono valere prima della invocazione del metodo (se i vincoli non sono soddisfatti allora il contratto non vale)
  - **@requires**: l'obbligo del cliente
- 🦋 La *postcondizione del metodo*: dichiara quali sono le proprietà che devono valere al termine dell'esecuzione del metodo (nell'ipotesi che la precondizione sia valida)
  - **@modifies**: descrive la portata delle modifiche effettuate durante l'esecuzione. Solo le entità descritte nella clausola "modifies" sono effettivamente modificate.
  - **@throws**: le eccezioni che possono essere sollevate (come Javadoc)
  - **@effects**: proprietà che valgono sullo stato modificato
  - **@return**: il valore che viene restituito (come Javadoc)

# Esempio (con Vector)



int change(Vector vec, Object oldelt, Object newelt)

@requires      v, oldelt, e newelt sono valori non-null  
oldelt appartiene a vec

@modifies      vec

@effects      la prima occorrenza del valore oldelt in vec viene  
modificata con il valore newelt  
& gli altri elementi di vec non sono modificati

@returns      l'indice della posizione in vec che conteneva il  
valore oldelt e che ora contiene il valore newelt



# Osservazione



- Supponiamo che il cliente dell'astrazione invochi il metodo **ma** che le precondizioni del metodo non siano verificate. Il codice del metodo è libero di fare qualunque cosa visto che non è vincolato dalla precondizione
  - È opportuno generare un *fallimento* piuttosto che generare dei comportamenti misteriosi

# IntSet



```
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, altrimenti false
    ...
}
```

# IntSet



```
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
}
```

# Esempi di uso



```
myIntSet = new IntSet( );
```

```
:
```

```
If myIntSet.IsIn(50) then System.out.println(...)
```

```
// Uso corretto
```

```
:
```

```
myIntSet = 50;
```

```
// Uso non corretto
```



# Specifiche e implementazioni

Supponiamo che *Impl* sia una possibile implementazione della specifica dell'astrazione *S*

*Impl* soddisfa *S* se e solo se

- ogni comportamento di *Impl* è un comportamento permesso dalla specifica *S*
- “i comportamenti di *Impl* sono un sottoinsieme dei comportamenti specificati da *S*”

Se *Impl* non soddisfa *S*, *Impl* oppure *S* (o entrambi) non sono “corretti”

- pragmaticamente è meglio cambiare l'implementazione piuttosto che la specifica

# Specifiche



- ✎ Potremmo avere due specifiche differenti dello stessa astrazione
- ✎ Magari vorremmo anche confrontarle!!

# Esempio



```
int find(int[ ] a, int value) {  
    for (int i = 0; i < a.length; i++)  
        if (a[i] == value) return i;  
    return -1;  
}
```

## 🦋 Specifica A

- **requires:** value è un valore memorizzato nell'array a
- **return:** indice i tale che  $a[i] = \text{value}$

## 🦋 Specifica B

- **requires:** value è un valore memorizzato nell'array a
- **return:** il più piccolo indice i tale che  $a[i] = \text{value}$

## 🦋 Specifica C

- **return:** l'indice i tale che  $a[i] = \text{value}$ , oppure -1 nel caso in cui value non sia memorizzato nell'array a

# Valutazione



- ✎ Una specifica “forte”
  - difficile da soddisfare (maggiore numero di vincoli sull’implementazione)
  - facile da usare (il cliente dell’astrazione può fare maggiori assunzioni sul comportamento)
- ✎ Una specifica “debole”
  - facile da verificare (facile da implementare, molte implementazioni la possono soddisfare)
  - difficile da usare per il minor numero di assunzioni



# Una visione formale



- ✎ Una specifica è una formula logica
  - $S1$  è più forte di  $S2$  se  $S1$  implica  $S2$
- ✎ Lo avete visto formalmente a LPP (riguardate il materiale didattico)
  - trasformate la specifica in una formula logica e poi verificate l'implicazione ( $S1 \wedge S2 \Rightarrow S2$ )

# Astrazioni sui dati: implementazione



- ✎ Scelta fondamentale è quella della rappresentazione (rep)
  - come i valori del tipo astratto sono implementati in termini di altri tipi
    - ✓ tipi primitivi o già implementati
    - ✓ nuovi tipi astratti che facilitano l'implementazione del nostro
      - tali tipi vengono specificati
      - metodologia: iterazione del processo di decomposizione basato su astrazioni
  - la scelta deve tener conto della possibilità di implementare in modo efficiente i costruttori e gli altri metodi
- ✎ Poi viene l'implementazione dei costruttori e dei metodi



# La rappresentazione

- ✎ I linguaggi che permettono la definizione di tipi di dato astratti hanno meccanismi molto diversi tra loro per definire come
  - i valori del nuovo tipo sono implementati in termini di valori di altri tipi
- ✎ In Java, gli oggetti del nuovo tipo sono semplicemente collezioni di valori di altri tipi
  - definite (nella implementazione della classe) da un insieme di variabili di istanza private
    - ✓ accessibili solo dai costruttori e dai metodi della classe

# Definire un tipo in Java



- ✎ Un insieme di variabili di istanza (devono essere dell'oggetto e non della classe)
  - **private**: devono essere accessibili solo dai costruttori e dai metodi della classe
- ✎ I valori espliciti che si vedono sono solo quelli costruiti dai costruttori
  - più o meno i casi base di una definizione ricorsiva
- ✎ Gli altri valori sono eventualmente calcolati dai metodi
  - rimane nascosta l'eventuale struttura ricorsiva

# Usi “corretti” delle classi in Java



- ✎ Nella definizione di astrazioni sui dati
  - le classi contengono essenzialmente metodi di istanza e variabili di istanza private
    - ✓ eventuali variabili statiche possono servire (ma è “sporco”!) per avere informazione condivisa fra oggetti diversi
    - ✓ eventuali metodi statici non possono comunque vedere l’oggetto e servono solo a manipolare le variabili statiche

# I tipi record in Java



- ✎ Java non ha un meccanismo primitivo per definire tipi record (le struct di C)
  - ma è facilissimo definirli
  - anche se con una deviazione dai discorsi metodologici che abbiamo fatto
    - ✓ la rappresentazione non è nascosta (non c'è astrazione!)
    - ✓ non ci sono metodi
    - ✓ di fatto non c'è una specifica separata dall'implementazione

# Un tipo record



```
class Pair {
    // OVERVIEW: un tipo record
    int coeff, exp;
    // costruttore
    Pair(int c, int n) {
        // EFFECTS: inizializza il "record" con i
        // valori di c ed n
        coeff = c; exp = n;
    }
}
```

- La rappresentazione non è nascosta
  - dopo aver creato un'istanza si accedono direttamente i "campi del record"
- La visibilità della classe e del costruttore è ristretta al package in cui figura
- Non ci sono metodi diversi dal costruttore



# Implementazione di IntSet

```
public class IntSet {
    // OVERVIEW: un IntSet è un insieme modificabile
    // di interi di dimensione qualunque
    private Vector els; // la rappresentazione
    // costruttore
    public IntSet( ) {
        // EFFECTS: inizializza this all'insieme vuoto
        els = new Vector( );
    }
    ...
}
```

- ⌚ Un insieme di interi è rappresentato da un **Vector**
  - più adatto dell'array, perché ha dimensione variabile
- ⌚ Gli elementi di un Vector sono di tipo **Object**
  - non possiamo memorizzarci valori di tipo **int**
  - usiamo oggetti di tipo **Integer**
    - ✓ interi visti come oggetti





# Implementazione di IntSet

```
public void insert(int x) {
    // EFFECTS: aggiunge x a this
    Integer y = new Integer(x);
    if (getIndex(y) < 0) els.add(y); }
private int getIndex(Integer x) {
    // EFFECTS: se x occorre in this ritorna la
    // posizione in cui si trova, altrimenti -1
    for (int i = 0; i < els.size( ); i++)
        if (x.equals(els.get(i))) return i;
    return -1; }
```

- ⌚ Non abbiamo occorrenze multiple di elementi
  - si semplifica l'implementazione di `remove`
- ⌚ Il metodo privato ausiliario `getIndex` ritorna un valore speciale e non solleva eccezioni
  - va bene perché è privato
- ⌚ Notare l'uso del metodo `equals` su `Integer`



# Implementazione di IntSet

```
public void remove(int x) {
    // EFFECTS: toglie x da this
    int i = getIndex(new Integer(x));
    if (i < 0) return;
    els.set(i, els.lastElement( ));
    els.remove(els.size( ) - 1);
}
public boolean isIn(int x) {
    // EFFECTS: se x appartiene a this ritorna
    // true, altrimenti false
    return getIndex(new Integer(x)) >= 0;
}
```

- 🐞 Nella rimozione, se l'elemento è presente, ci scrivo sopra l'ultimo corrente ed elimino l'ultimo elemento



# Implementazione di IntSet

```
public int size( ) {
    // EFFECTS: ritorna la cardinalità di this
    return els.size();
}
public int choose( ) throws EmptyException {
    // EFFECTS: se this è vuoto, solleva
    // EmptyException, altrimenti ritorna un
    // elemento qualunque contenuto in this
    if (els.size() == 0) throw
        new EmptyException("IntSet.choose");
    return
        ((Integer) els.lastElement()).intValue();
}
```

🐞 Anche se `lastElement` potesse sollevare un'eccezione, in questo caso non può succedere. Come mai?