Astrazioni sui dati : Specifica di Tipi di Dato Astratti in Java

Specifica ed Implementazione di Tipi di Dato Astratti in Java

- ✓ cos'è un tipo di dato astratto
- ✓ specifica di tipi di dati astratti
  - un tipo modificabile (`IntSet`)
  - qualche tipo primitivo
  - un tipo non modificabile (`Poly`)
- ✓ implementazione di tipi di dati astratti
  - definire la rappresentazione
    - le restrizioni nell'uso di Java
    - un record type (l'eccezione!)
  - l'implementazione di `IntSet` e `Poly`
- ✓ alcuni metodi "ereditabili"

Perché l'astrazione sui dati

- ✓ il più importante tipo di astrazione
- ✓ il nucleo (non esclusivo) della programmazione orientata ad oggetti
- ✓ lo scopo è quello di estendere il linguaggio
  - nel nostro caso, Java

  con nuovi tipi di dato
- ✓ quali?
  - dipende dall'applicazione
    - interprete: stacks e tabelle di simboli
    - applicazione bancaria: conti
    - applicazioni numeriche: matrici

Cos'è un'astrazione sui dati

- ✓ in ogni caso è
  - un insieme di oggetti
    - stacks, conti, matrici
  - + un insieme di operazioni
    - per crearli e manipolarli
- ✓ i nuovi tipi di dato dovrebbero incorporare i due meccanismi di astrazione
  - parametrizzazione
    - simile al caso delle astrazioni procedurali
    - quella più importante si realizza con il polimorfismo
      - che vedremo più avanti
  - specifica
    - porta a vedere necessariamente insieme oggetti ed operazioni

Astrazione sui dati via specifica

- ✓ con la specifica, astraiamo dall'implementazione del tipo di dato
  - dalla sua rappresentazione
- ✓ se avessimo solo la rappresentazione degli oggetti non sarebbe possibile
  - o l'utente opera direttamente sulla rappresentazione
    - non ci sarebbe astrazione
    - se la rappresentazione viene modificata, la modifica si ripercuote su tutti gli utenti
  - oppure non si rende possibile la manipolazione degli oggetti del nuovo tipo
- ✓ 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 della specifica di un tipo di dato astratto

- ✓ Java (parte sintattica della specifica)
  - classe o interfaccia
    - per ora solo classi
  - nome per il tipo
    - nome della classe
  - operazioni
    - metodi di istanza, incluso il(i) costruttore(i)
- ✓ la specifica del tipo
  - fornita dalla clausola OVERVIEW

  descrive ***i valori astratti*** degli oggetti

  ed alcune loro proprietà
  - per esempio la modificabilità
- ✓ per il resto la specifica è una specifica dei metodi
  - strutturata come già abbiamo visto per le astrazioni procedurali
  - l'oggetto su cui i metodi operano è indicato nella specifica da this

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
}
```

L'insieme di interi 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, altrimenti false
  ...}
```

L'insieme di interi 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`: commenti 1

```
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
  - in questo caso, gli insiemi matematici
- ✓ gli stessi concetti sono anche utilizzati nella specifica dei metodi
  - aggiungere, togliere elementi
  - appartenenza, cardinalità

`IntSet`: commenti 2

```
public class IntSet {
 // OVERVIEW: ...
 // 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`: commenti 3

```
public class IntSet {
...
 // metodi
   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 nè `insert` nè `remove` sollevano eccezioni
    - se si inserisce un elemento che c'è già
    - se si rimuove un elemento che non c'è

`IntSet`: commenti 4

```
public boolean isIn (int x)
    // EFFECTS: se x appartiene a this ritorna
    // true, altrimenti false
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...}
```

- ✓ 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 sottodeterminata (implementazioni corrette diverse possono dare diversi risultati)

Specifica di un tipo “primitivo”

- ✓ le specifiche sono ovviamente utili per capire ed utilizzare correttamente i tipi di dato “primitivi” di Java
- ✓ vedremo, come esempio, il caso dei vettori
  - `Vector`
  - arrays dinamici che possono crescere e accorciarsi
  - 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>= 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>= 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                    }
```

`Vector`: commenti 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
   ....
}
```

- ✓ gli oggetti della classe sono descritti nella specifica in termini di concetti noti
  - in questo caso, gli arrays
- ✓ gli stessi concetti sono anche utilizzati nella specifica dei metodi
  - indice, elemento identificato dall'indice
- ✓ il tipo è modificabile
  - come l'array
- ✓ notare che gli elementi sono di tipo Object
  - per esempio, non possono essere int, bool e char

`Vector`: commenti 2

```
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) ad un "array" vuoto

`Vector`: commenti 3

```
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 primitiva unchecked

`Vector`: commenti 4

```
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

I polinomi 1

```
public class Poly {
 // OVERVIEW: un Poly è un polinomio a
 // cofficienti interi non modificabile
 // esempio: c0 + c1*x + c2*x^2 + ...

 // costruttori
  public Poly ()
    // EFFECTS: inizializza this al polinomio 0
  public Poly (int c, int n) throws
     NegativeExponentExc
    // EFFECTS: se n<0 solleva NegativeExponentExc
    // altrimenti inizializza this al polinomio cx^n

 // metodi

  ...}
```

I polinomi 2

```
public class Poly {
 ...
 // metodi
  public int degree ()
     // EFFECTS: ritorna 0 se this è il polinomio
     // 0, altrimenti il più grande esponente con
     // coefficiente diverso da 0 in this
  public int coeff (int d)
     // EFFECTS: ritorna il coefficiente del
     // termine in this che ha come esponente d
  public Poly add (Poly q) throws
      NullPointerException
    // EFFECTS: q=null solleva NullPointerException
    // altrimenti ritorna this + q
...}
```

I polinomi 3

```
public class Poly {
 ...
 // metodi
  ...
  public Poly mul (Poly q) throws
     NullPointerException
   // EFFECTS: q=null solleva NullPointerException
   // altrimenti ritorna this * q
  public Poly sub (Poly q) throws
     NullPointerException
   // EFFECTS: q=null solleva NullPointerException
   // altrimenti ritorna this - q
  public Poly minus ()
   // EFFECTS: ritorna -this
}
```

`Poly`: commenti 1

```
public class Poly {
 // OVERVIEW: un Poly è un polinomio a
 // cofficienti interi non modificabile
 // esempio: c0 + c1*x + c2*x^2 + ...
 ...}
```

- ✓ gli oggetti della classe sono descritti nella specifica in termini di concetti noti
  - in questo caso, i polinomi
- ✓ gli stessi concetti sono anche utilizzati nella specifica dei metodi
  - operazioni di +, *, e -

Poly: commenti 2

```
public class Poly {
 // OVERVIEW: ...
 // costruttori
  public Poly ()
    // EFFECTS: inizializza this al polinomio 0
  public Poly (int c, int n) throws
     NegativeExponentExc
    // EFFECTS: se n<0 solleva NegativeExponentExc
    // altrimenti inizializza this al polinomio cx^n
...}
```

- ✓ due costruttori overloaded
  - stesso nome (quello della classe)
  - diverso numero o tipo di parametri
    - se no, errore di compilazione
  - la scelta tra metodi overloaded viene effettuata in base al numero e tipo di parametri
    - eventualmente a run time scegliendo il più specifico

Poly: commenti 3

```
public class Poly {
 // OVERVIEW: ...
 // costruttori
  public Poly ()
    // EFFECTS: inizializza this al polinomio 0
  public Poly (int c, int n) throws
     NegativeExponentExc
    // EFFECTS: se n<0 solleva NegativeExponentExc
    // altrimenti inizializza this al polinomio cx^n
...}
```

- ✓ l'eccezione `NegativeExponentExc` non è definita qui
  - nello stesso package di `Poly`?
  - può essere unchecked, perché non è probabile che l'utente usi esponenti negativi

Poly: commenti 4

```
// metodi
  public int degree ()
     // EFFECTS: ritorna 0 se this è il polinomio
     // 0, altrimenti il più grande esponente con
     // coefficiente diverso da 0 in this
  public int coeff (int d)
     // EFFECTS: ritorna il coefficiente del
     // termine in this che ha come esponente d
  public Poly add (Poly q) throws
      NullPointerException
   // EFFECTS: q=null solleva NullPointerException
   // altrimenti ritorna this + q
...
```

- ✓ non ci sono modificatori
  - il tipo è non modificabile!
  - `degree` e `coeff` sono osservatori
  - `add`, `mul`, `sub` e `minus` ritornano nuovi oggetti di tipo `Poly`

Proprietà della Specifica

- per prima cosa si definisce la specifica
  - "scheletro" formato da headers, overview, pre e post condizioni di tutti i metodi
  - mancano la rappresentazione degli oggetti ed il codice dei corpi dei metodi
    - che possono essere sviluppati in un momento successivo ed indipendentemente dallo sviluppo dei "moduli" che usano il nuovo tipo di dato
    - è molto importante riuscire a differire le scelte relative alla rappresentazione
  - se aggiungiamo corpi contenenti return 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)

Implementazione di un metodo che usa `IntSet`

```
public static IntSet getElements (int[] a) throws
    NullPointerException
     // EFFECTS: 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
  - non esiste ancora
  - anche se ci fosse non potrebbe "vederla"
- costruisce, accede e modifica l'oggetto solo attraverso i metodi (incluso il costruttore)

Verifiche nel metodo che usa `IntSet`

```
public static IntSet getElements (int[] a) throws
    NullPointerException
     // EFFECTS: 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;
}
```

✓ insert non ha precondizione

- se ci fosse una precondizione c bisognerebbe
  - inserire in getelements il codice che verifichi c (run time check), oppure
  - dimostrare che c è sempre verificata (verifica "statica")

Implementazione di un metodo che usa `Poly`

```
public static Poly diff (Poly p) throws
    NullPointerException
     // EFFECTS: p=null solleva NullPointerException
     // altrimenti, restituisce il polinomio che risulta
     // dalla differenziazione di p
{Poly q = new Poly();
 for (int i = 0; i <= p.degree(); i++)
       q = q.add(new Poly(p.coeff(i)*i, i-1));
 return q;          }
```

- ✓ scritta solo conoscendo la specifica di `Poly`
  - non accede all'implementazione
    - non esiste ancora
    - anche se ci fosse non potrebbe "vederla"
  - costruisce ed accede l'oggetto solo attraverso i metodi (incluso il costruttore)
  - nessuna precondizione -> nessuna verifica