ASTRAZIONI SUI DATI : IMPLEMENTAZIONE DI TIPI DI DATI ASTRATTI IN JAVA

Abstract Data Types

- Un Tipo di Dati Astratto (ADT) è costituito da:
  - un **nome**
  - un insieme di **valori**
  - un insieme di **operazioni** su tali valori (con **nome**, lista di **parametri**, tipo del **risultato**)
  - tipicamente, una **specifica** (più o meno formale, più o meno esplicita) delle operazioni
- Un ADT **non** fissa una **rappresentazione** dei valori, quindi sui valori di un ADT si può operare solo con le operazioni fornite

Specifica come contratto

- La specifica di un ADT è un contratto che garantisce una *Separation of Concerns:*
  - Progettazione e realizzazione dell'ADT
    - ✓Si fissa una rappresentazione per i valori
    - ✓Si implementano le operazioni in base alla rappresentazione scelta, in modo da soddisfare la specifica
  - Progettazione delle applicazioni che usano l'ADT
    - ✓Usano valori dell'ADT tramite le operazioni
    - ✓Non accedono alla rappresentazione
    - ✓Cambi di implementazione dell'ADT non le influenzano

3

Esempio di specifica: `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
  ...}
```

4

Esempio di specifica: 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
}
```

5

Esempi di uso

```
myIntSet = new IntSet();
:
//Uso corretto
if (myIntSet.IsIn(50))
     System.out.println(...)
:
//Uso scorretto
myIntSet.elements.add(new Integer(50));
```

6

Astrazioni sui dati: implementazione

- Scelta fondamentale è quella della rappresentazione (**rep**)
  - cioè come i valori del tipo astratto sono implementati in termini di altri tipi
    - ✓tipi primitivi o già implementati
    - ✓nuovi tipi astratti, che facilitino l'implementazione del nostro
      - Si fornisce la specifica di tali tipi
      - Metodologia: iterazione del processo di decomposizione top-down basato su astrazioni
  - la scelta della rep deve tener conto della possibilità di implementare in modo efficiente costruttori e metodi
- Poi viene l'implementazione dei costruttori e dei metodi

7

La rappresentazione

- Come si implementano i valori di un nuovo tipo in termini di valori di altri tipi?
- I linguaggi che consentono la definizione di tipi di dati astratti hanno meccanismi molto diversi tra loro
- 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**
    - ✓quindi accessibili solo dai costruttori e dai metodi della classe
  - I valori del TDA sono generati dai *costruttori*, o dai *produttori* (metodi che generano nuovi valori), e modificati dai *modificatori*

8

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 (di classe) possono servire per avere informazione condivisa fra oggetti diversi (sconsigliabili…)
    - ✓eventuali metodi statici non possono comunque vedere le variabili di istanza di un oggetto, e servono solo a manipolare le variabili statiche

9

I tipi record in Java

- Java non ha un meccanismo primitivo per definire **tipi record** (le `struct` del C), cioè
  - collezioni di valori anche eterogenei, indicizzati da parole chiave (*campi*)
- Ma è facile definirli usando le classi, anche se con una eccezione rispetto ai 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

10

Un esempio di tipo record

```
class Pair {
 // OVERVIEW: un tipo record
  int coeff;
  int 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", es:
    - ✓`int n = myPair.coeff; myPair.n = 5`
- la visibilità della classe e del costruttore è ristretta al package in cui figura
- non ci sono metodi diversi dal costruttore

11

`IntSet`: scelta di rappresentazione

```
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 l'insieme 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

12

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`
- NB: 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`

13

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

14

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, qui non può succedere

15

I polinomi: specifica

```
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
    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
...
```

16

I polinomi: specifica

```
 ...
// produttori
   public Poly add (Poly q) throws NullPointerException
   // EFFECTS: q=null solleva NullPointerException
   // altrimenti ritorna this + q

   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
}
```


Prima implementazione di Poly

```
public class Poly {
 // OVERVIEW: un Poly è un polinomio a
 // cofficienti interi non modificabile
 // esempio: c0 + c1*x + c2*x^2 + ...
 private int[] termini; // la rappresentazione
 private int deg;
```

- I polinomi non cambiano la dimensione
  - Array invece che Vector
  - l'elemento in posizione *i* contiene il coefficiente del termine che ha esponente *i*. Quindi, p.es., $c_0 + c_1*x + c_2*x^2$ è rappresentato come l'array $[c_0, c_1, c_2]$
  - va bene per polinomi non sparsi
- per comodità (efficienza) teniamo traccia nella rep del grado (degree) del polinomio:
  - variabile di tipo int

Prima implementazione di Poly

```
// costruttori
   public Poly ()
   // EFFECTS: inizializza this al polinomio 0
      {termini = new int[1]; deg = 0; }
   public Poly (int c, int n) throws NegativeExponentExc
   // EFFECTS: se n<0 solleva NegativeExponentExc
   // altrimenti inizializza this al polinomio cx^n
      {if (n < 0) throw new
      NegativeExponentExc("Poly(int,int) constructor");
      if (c == 0){
            termini = new int[1];
            deg = 0; return; }
      termini = new int[n+1];
      for (int i = 0; i < n; i++) termini[i] = 0;
      termini[n] = c; deg = n; }
   private Poly (int n)
      {termini = new int[n+1]; deg = n; }
```

- Il polinomio vuoto è rappresentato da un array di un elemento contenente 0
- Si noti il costruttore privato

19

Prima implementazione di 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
      {return deg; }
   public int coeff (int d)
   // EFFECTS: ritorna il coefficiente del
   // termine in this che ha come esponente d
      {if (d < 0 || d > deg) return 0;
            else return termini[d];}
   public Poly minus ()
   // EFFECTS: ritorna –this
      {Poly y = new Poly(deg);
      for (int i = 0; i < deg; i++)
            y.termini[i] = - termini[i];
      return y;}
   public Poly sub (Poly q) throws NullPointerException
   // EFFECTS: q=null solleva NullPointerException
   // altrimenti ritorna this – q
      {return add(q.minus()); }
```

20

Prima implementazione di Poly

- le implementazioni di **add** e **mul** sono più complesse
  - ma solo negli aspetti algoritmici che non mostriamo
- se i polinomi sono sparsi, questa implementazione non è efficiente
  - arrays grandi e pieni di 0
  - un'implementazione alternativa in termini di Vector i cui elementi sono coppie (coefficiente, esponente)
    - ✓esattamente il record type che abbiamo visto

```
class Pair {
int coeff; int exp;
Pair (int c, int n)
    { coeff = c; exp = n;}}
```

21

Seconda implementazione di Poly

```
public class Poly {
 // OVERVIEW: un Poly è un polinomio a
 // cofficienti interi non modificabile
 // esempio: c0 + c1*x + c2*x^2 + ...
 private Vector termini; // la rappresentazione
 private int deg; // la rappresentazione
```

- gli oggetti contenuti in termini sono Pair che rappresentano i termini con coefficiente diverso da 0
- un esempio di operazione

```
public int coeff (int d)
 // EFFECTS: ritorna il coefficiente del
 // termine in this che ha come esponente d
 {for (int i = 0; i < termini.size(); i++)
   {Pair p = (Pair) termini.get(i);
   if (p.exp == d) return p.coeff;}
 return 0;}
```

  - notare il casting

22