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

Metodi e tipi di astrazione

- **Metodi:**
  - ***Astrazione per parametrizzazione***
  - ***Astrazione per specifica***
- **Tipi di astrazione:**
  - ***Astrazione procedurale***: separazione delle proprietà logiche di una azione computazionale dai dettagli implementativi
  - ***Astrazione di dati***: separazione delle proprietà logiche dei dati dai dettagli della rappresentazione dei dati.
  - ***Iterazione astratta***: consente di iterare su elementi di una collezione, ignorando dettagli su struttura e accesso

Astrazione Procedurale

- Si realizza con funzioni, procedure, macro, metodi, subroutines, …
- Supportata da tutti i linguaggi di programmazione
- ***Astrazione per parametrizzazione*** realizzata con meccanismo di passaggio di parametri
- ***Astrazione per specifica***:
  - Descrizione precisa del comportamento
  - Compatta, non ambigua
  - Formale (logica, es: Triple di Hoare) o informale (linguaggio naturale, anche con notazione matematica)

Astrazione Procedurale: Specifica

Struttura di una specifica:

- Intestazione (definita dal linguaggio di programmazione): informazione sintattica, facilita la pametrizzazione. Es: `int gcd(int x, int y)`
- Parte semantica (in linguaggio naturale) comprende le clausole
  - **REQUIRES** [opzionale]: vincoli sugli input
    - ✓ Presente per procedure parziali
  - **MODIFIES** [opzionale]: descrive modifiche sugli input
    - ✓ Input: parametri, variabili globali, risorse (file), ….
    - ✓ Assente se non ha side-effects
  - **EFFECTS**: definisce il comportamento
    - ✓ Quali output, per input legali
    - ✓ Quali modifiche
- La specifica deve essere scritta prima del codice

Esempio di specifica (javadoc)

```
/** inserts new characters into a string * at the indicated
location<br>
* REQUIRES: base's length >= index <br> * MODIFIES: base <br>
* EFFECTS: inserts chars after index characters in base, leaving
remaining part of base (if any) after chars
* @param base original string
* @param index where to insert
* @param chars the part being inserted
*/

public static void insert(StringBuffer base,
                          int index,String chars)
```


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

Specificare un ADT

- Un ADT è un esempio di **astrazione di dati per specifica**
  - La specifica è costituita dalla segnatura (firma) delle operazioni (nome, tipo dei parametri e del risultato), e inoltre può comprendere:
    - ✓Pre- e post-condizioni
    - ✓Assiomi / equazioni
    - ✓Semantica formale

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

Ingredienti della specifica di un ADT

- ✓ 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 ed alcune loro proprietà per esempio la modificabilità
- ✓ per il resto la specifica è una specifica dei metodi
  - strutturata tramite precondizioni (clausola REQUIRE) e postcondizioni (clausola EFFECT)

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
}
```


Precondizioni e Postcondizioni

- Per ogni costruttore o metodo,
- la precondizione (REQUIRES) è una formula logica che caratterizza le proprietà e il valore dei parametri attuali
- la postcondizione (EFFECTS) è un formula logica che caratterizza il risultato calcolato dall'operazione rispetto al valore degli argomenti e allo stato dell'oggetto

Esempio: 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 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
  - uso di una notazione matematica (in seguito)
- Gli stessi concetti sono anche utilizzati nella specifica dei metodi
  - aggiungere, togliere elementi
  - appartenenza, cardinalità

IntSet: analisi

```
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: analisi

```
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: analisi

```
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 risultati)

Specifica di un tipo "primitivo"

- le specifiche sono ovviamente utili **anche** per capire ed utilizzare correttamente i tipi di dato "primitivi" di Java
- vedremo, come esempio, il caso dei vettori
  - Vector (o ArrayList)
  - arrays dinamici che possono crescere e accorciarsi
  - sono definiti nel package java.util

La specifica di Vector

```
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 un array 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: 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 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
  - non possono essere int, bool e 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) ad 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 primitiva 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

Un altro esempio: i polinomi

```
public class Poly {
 // OVERVIEW: un Poly è un polinomio a
 // cofficienti interi non modificabile
 // esempio: c0 + c1*x + c2*x2 + ...

 // 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 cxn

 // metodi

  ...}
```


I polinomi

```
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

```
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`: Analisi

```
ublic 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 cxn
..}
```

- due costruttori overloaded
  - stesso nome (quello della classe)
  - diverso numero o tipo di parametri
  - la scelta tra metodi overloaded viene effettuata a tempo di compilazione in base al numero e tipo di parametri (eventualmente scegliendo il più specifico)

`Poly`: analisi

```
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 cxn
...}
```

- 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`: analisi

```
// 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: se 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

Aspetti metodologici

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 e 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

Aspetti metodologici

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** e' basato su questa idea.

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 nell'array 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
    - ✓ora non esiste ancora. Ma anche se ci fosse non potrebbe "vederla"
  - costruisce, accede e modifica l'oggetto solo attraverso i metodi (incluso il costruttore)

Verifiche

```
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 nell'array 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 bisognerebbe
    - ✓inserire in *getElements* il codice che la verifichi (run time check), oppure
    - ✓dimostrare che la precondizione è sempre verificata (verifica "statica")

Specifica di ADT con assiomi

- Nome: *Stack (of Item)*
- *Item* (parametro) il tipo degli elementi che possono essere inseriti nello stack
- Informalmente, uno stack (pila) è una sequenza finita di *items* gestita con politica LIFO (Last In First Out)
- Usa anche **astrazione per parametrizzazione**, oltre che **per specifica**
  - Realizzabile con *variabili di tipo* sia in Ocaml (es: **'a stack**) che in Java (es: **Stack<Item>**)

ADT STACK: Operazioni

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

ADT STACK: Assiomi e proprietà

- size(s), empty(s), push(s,t) sono sempre definite (operazioni totali)
- pop(s) e top(s) sono definite iff empty(s) = False
- empty(s), size(s), top(s) lasciano S immutato (*accessors* o *observers*)
- size(create()) = 0
- size(push(s,t)) = size(s) + 1
- size(pop(s)) = size(s) - 1
- empty(s) = True iff size(s) = 0
- empty(push(s,x)) = False
- pop(push(s,t)) = s
- top(push(s,t)) = t // LIFO

A cosa servono gli assiomi?

- Possono essere usati facilmente come test per verificare la correttezza di un'implementazione (p.e. come asserzioni)
- Possono essere usati per dimostrare proprietà di programmi
  - **Esempio:** Assumiamo che (n = size(s)) e che s' sia lo stack ottenuto dopo avere eseguito k operazioni di push su s.
  - Allora size(s') = size(s) + k

Concludendo, qual è il punto?

- *Client—Supplier*
- *Supplier: fornisce il servizio con un ADT*
- *Client: utente del servizio (a sua volta fornitore di altri servizi)*
- *Visione moderna che si ritrova in*
  - *Service oriented computing*
  - *Cloud computing*

Aspetti significativi (ACM SIG Soft. Eng)

- Supplier:
  - efficient and reliable algorithms and data structures
  - convenient implementation
  - easy maintenance
- Clients:
  - using the supplier services without effort to understand its internal details
  - having a sufficient, but not overwhelming, set of operations