I Tipi di Dato Astratto (TDA) - II

Stefano Mizzaro

Dipartimento di matematica e informatica
Università di Udine
http://www.dimi.uniud.it/mizzaro
mizzaro@dimi.uniud.it
Programmazione, lezione 17
18 gennaio 2006

Oggi

- Comunicazioni di servizio, Provetta
- Riassunto
  - Programmazione strutturata
  - TDA (Tipo di dato astratto)
    - Evoluzione della programmazione strutturata
    - Occultamento delle informazioni
- Fine TDA
  - Aggiunte e puntualizzazioni
    - `this`, `toString`, procedurale vs. funzionale
  - Esempi

Orario

	Lun	Mar	Mer	Gio	Ven
I 9:00-10:45			Prog (E)		
II 11:00-12:45					
"II e ½"					
III 15:00-16:45	Lab A – ~M		Lab A – ~M		
IV 17:00-18:45	Lab ~M – Z		Lab ~M – Z	Prog (H)	

Prossime lezioni

- Aula (8 lez. con me)
  - 18/1, 19/1, 25/1, 26/1,
  - 1/2, 8/2, 15/2, 22/2
- Laboratorio (9 lez. con Lucio Ieronutti)
  - 16/1,
  - 23/1, 25/1, 30/1, 1/2,
  - 6/2, 13/2, 20/2, 27/2
- (salvo contrattempi/variazioni)
- Ricevimento (mio): Martedì 10:00 – 12:00

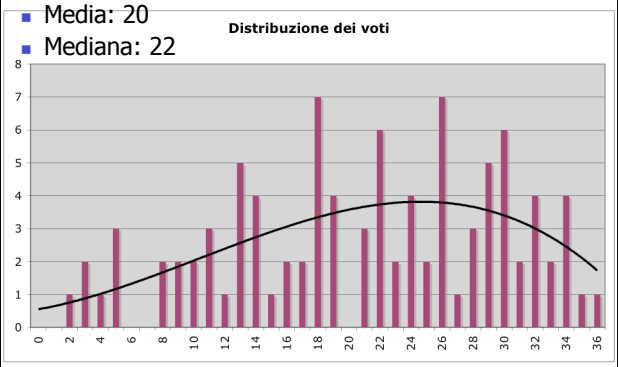
Provetta

	Iscritti	Presentati	Consegnati	Suff.	Suff. 2a provetta
qs. anno	130	110 (85%)	95 (73%)	64 (49%)	?!
anno scorso	140	~120 (~86%)	114 (81%)	80 (57%)	47 (34%)

- Ergo:
  - La 2a provetta è più difficile…
  - Studiate! Non "adagiatevi"!!

Troppo difficile?

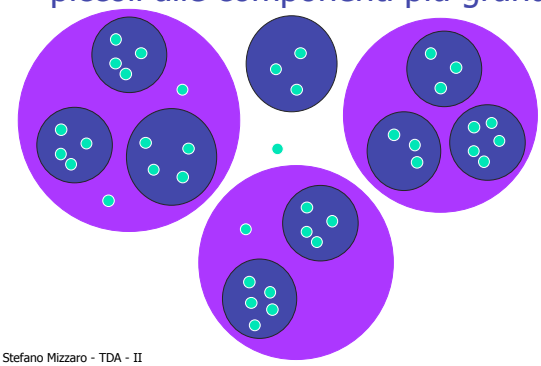
- Media: 20
- Mediana: 22



Scaletta

- Comunicazioni di servizio, Provetta
- Riassunto
  - Programmazione strutturata
  - TDA (Tipo di dato astratto)
    - Evoluzione della programmazione strutturata
    - Occultamento delle informazioni
- Fine TDA
  - Aggiunte e puntualizzazioni
    - `this`, `toString`, procedurale vs. funzionale
  - Esempi

Analisi sistematica: dai mattoni più piccoli alle componenti più grandi



Riassunto: fino alla programmazione strutturata

- "Mattoni"
- Strutture di controllo
  - Sequenza, Selezione, Iterazione
- Sviluppo incrementale
- Cicli annidati
- Array
  - Unidimensionali, multidimensionali
- Sottoprogrammi (metodi)
  - Parametri formali e attuali
- Ricorsione
- Pila dei record di attivazione, heap

Struttura programma Java?

```
class ... {
  static <tipo> <id> (<parametri>) {
    ...
  }
  static <tipo> <id> (<parametri>) {
    ...
  }
  public static void main (String[] args) {
    ...
  }
  static <tipo> <id> (<parametri>) {
    ...
  }
  static <tipo> <id> (<parametri>) {
    ...
  }
}
```


TDA: l'idea

- Tipo di Dato Astratto (TDA)
- Abstract Data Type (ADT)
- L'idea è:
  - Programma che gestisce numeri complessi => aggiungo il tipo di dato astratto **`NumeroComplesso`**
  - Programma dell'anagrafe => aggiungo i TDA **`Persona`**, **`Indirizzo`**, **`Data`**, …
- Non "aggiungo al Java i metodi che non ha" ma "aggiungo al Java i tipi che non ha"

TDA: i concetti (1/2)

- Astrazione: nuovo tipo (non nuova istruzione)
- Definire
  - Come sono fatti (attributi): implementazione
  - Operazioni (metodi, per ora `static`): interfaccia
- `class`
- Uso:
  - Dichiarazione
  - E allocazione, sullo heap
    - `new` (ricorda qualcosa?)
    - Costruttore
  - Notazione puntata (ricorda qcosa?)

TDA: i concetti (2/2)

- Occultamento delle informazioni (implementazione)
  - Information (implementation) hiding
  - Cambio implementazione ⇒ cambio uso? NO!!
  - `private`
  - Incapsulamento = unitarietà + inaccessibilità
  - Operazioni di accesso con metodi
  - Astratto = si astrae dall'implementazione quando si usa
- Metodi/costruttori sovraccarichi (overloaded)
- Terminologia:
  - Tipi, Classi, Variabili, (Istanze, Oggetti)

Esempio: il TDA `Ora`

```
class Ora {
  private int ore, minuti, secondi;
  Ora(int o, int m, int s) {
    ore = o; minuti = m; secondi = s;
  }
  static void setOra(Ora x, int o, int m, int s){
    x.ore = o; x.minuti = m; x.secondi = s;
  }
  static void setOra(Ora x, int o, int m) {
    setOra(x,o,m,0);
  }
  static int getOre(Ora x) {return x.ore;}
  static int getMinuti(Ora x) {return x.minuti;}
  static int getSecondi(Ora x){return x.secondi;}
}
```


Esempio: uso del TDA `Ora`

```
class UsaOra {
  public static void main(String[] args) {
    Ora x = new Ora(12,32,05);
    Ora y = new Ora(Ora.getOre(x), 20, 3);
    System.out.println(Ora.getMinuti(y));
    Ora.setOra(y, 20, 4);
    System.out.println(Ora.getOre(y) + ":" +
                       Ora.getMinuti(y) + ":" +
                       Ora.getSecondi(y));
  }
}
```

- Qs. è un caso banale, immaginarsi in generale..

Scaletta

- Comunicazioni di servizio, Provetta
- Riassunto
  - Programmazione strutturata
  - TDA (Tipo di dato astratto)
    - Evoluzione della programmazione strutturata
    - Occultamento delle informazioni
- Fine TDA
  - Aggiunte e puntualizzazioni
    - `this`, `toString`, procedurale vs. funzionale
  - Esempi

Visibilità

- Nei costruttori precedenti ho dovuto inventare nomi di parametri che fossero diversi da quelli delle variabili della classe.
- Cosa succede se usiamo gli stessi nomi?

```
class Coordinate {
  private int x, y;
  Coordinate(int x, int y) {
    x = x;
    y = y;
  }
}
```


Il `this`

- Regola di visibilità
  - Se nomi dei parametri = nomi degli attributi
  - Gli attributi vengono nascosti
  - (del costruttore, o di qualsiasi altro metodo)
- Per accedervi occorre specificare che sono le variabili di "questo oggetto", con la parola riservata `this`:

```
Coodinate(int x, int y){this.x=x; this.y=y;}
```

  - «assegno alla variabile `x` di questo oggetto il valore del parametro `x`, assegno alla variabile `y` di questo oggetto il valore del parametro `y`»

```
Coordinate(int x, int y) {x = x; y = y;}
```

  - «assegno al parametro `x` il valore del parametro `x`»... e quindi non modifico la variabile `x` della classe!

Il `this`

- `this` può essere usata anche per riferirsi ad un altro costruttore (sovraccarico). Esempio:

```
class Coordinate {
  private int x, y;
  Coodinate(int x, int y) {
    this.x = x; this.y = y;
  }
  /** costruttore che crea un oggetto con
   *  coordinate uguali */
  Coordinate(int a) {this(a,a);}
}
```

questo costruttore

Più costruttori

- A cosa servono?
- Semplificano l'uso (l'allocazione) delle variabili del TDA
- Può essere comodo, ma non esagerare…

```
Coordinate c = new Coordinate();
Coordinate c = new Coordinate(0);
Coordinate c = new Coordinate(0,0);
```


Riassunto visibilità

- Variabili locali a un blocco
  - `for (int i = …)`
- Variabili locali a un metodo
  - `static void m(int x){ int y...`
- Variabili locali a una classe
  - `class C { int x`
  - `public` (o niente): visibili anche all'esterno della classe
  - `private`: solo all'interno della classe

Scaletta

- Comunicazioni di servizio, Provetta
- Riassunto
  - Programmazione strutturata
  - TDA (Tipo di dato astratto)
    - Evoluzione della programmazione strutturata
    - Occultamento delle informazioni
- Fine TDA
  - Aggiunte e puntualizzazioni
    - `this`, `toString`, procedurale vs. funzionale
  - Esempi

Il metodo `toString`

- Chi deve avere la conoscenza/responsabilità di sapere come visualizzare il valore di un TDA?
- La classe! (cfr. `Ora` e `UsaOra`: scomodo!)
- Si potrebbe definire un metodo nel TDA
  - `static void print(...)`
- … ma si preferisce definire (vedremo perché)
  - `static String toString(…)`
- `toString` verrà invocato nell'istruzione `System.out.print` o `println`
- In questo modo è possibile specificare come verranno visualizzati gli oggetti di una certa classe
- (Vedremo l'invocazione automatica di `toString`)

`toString`

- `toString` restituisce la rappresentazione dell'oggetto in forma di stringa. Ad es. :

```
class Coordinate {
  private int x, y;
  ...
  static String toString(Coordinate c){
    return "(" + c.x + "," + c.y + ")";
  }
}
```

```
class UsoCoordinate {
  Coordinate a = new ...;
  ...
  System.out.println(Coordinate.toString(a));
}
```

Ora.java

```
class Ora {
  private int ore, minuti, secondi;
  Ora(int ore, int minuti, int secondi) {
    impostaOra(ore, minuti, secondi);
  }
  static void impostaOra(Ora x, int ore, int minuti,
                         int secondi) {
    x.ore = ore;
    x.minuti = minuti;
    x.secondi = secondi;
  }
  static int getOre(Ora x) {return x.ore;}
  static int getMinuti(Ora x) {return x.minuti;}
  static int getSecondi(Ora x) {return x.secondi;}
  static String toString(Ora x) {
    return x.ore+":"+x.minuti+"."+x.secondi;
  }
```

UsaOra.java

```
class UsaOra {
  public static void main(String[] args) {
    Ora x = new Ora(10,2,3);
    System.out.println(Ora.toString(x));
    x.impostaOra(12,2,0);
    System.out.println(Ora.getMinuti(x));
  }
}
```

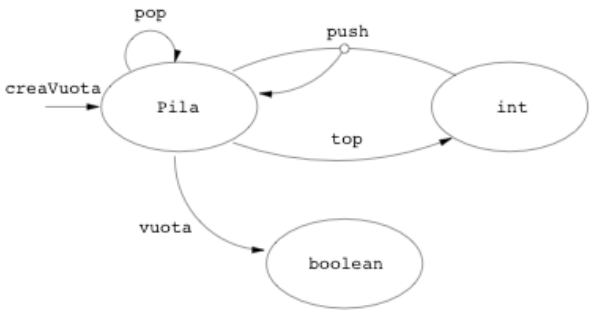
10:2.3

2

Pila (di interi limitata)

- Pila (stack): struttura dati a gestione LIFO (Last In First Out, il primo che entra è l'ultimo che esce)
- Operazioni:
  - push: aggiungi un elemento in cima
  - top: restituisce l'elemento in cima
  - pop: rimuove un elemento dalla cima
  - piena: dice se c'è ancora posta
  - vuota: dice se è vuota
- Vediamo una versione semplice: la pila limitata di interi
  - C'è un numero massimo di elementi
  - Quindi potremo implementarla con un array

Segnatura del TDA Pila



Pila.java (1/2)

```
/**Il TDA pila di interi */
class Pila {
  /** Il numero di elementi massimo */
  private static final int MAXELEMENTI = 10;
  /** Il vettore che contiene gli elementi*/
  private int[] elementi;
  /** Il numero di elementi effettivamente presenti */
  private int numElementi;
  /** Costruttore: costruisce (e restituisce
   * implicitamente) una pila vuota */
  public Pila() {
    numElementi = 0;
    elementi = new int[MAXELEMENTI];
  }
  /** Dice se la pila p e' vuota */
  public static boolean vuota (Pila p) {
    return (p.numElementi == 0);
  }
  /** Dice se la pila p e' piena */
  public static boolean piena (Pila p) {
    return (p.numElementi == p.MAXELEMENTI);
  }
```

Pila.java (2/2)

```
  /** push dell'elemento e sulla pila p */
  public static Pila push (Pila p, int e) {
    if (!piena(p))
      p.elementi[p.numElementi++] = e;
    return p;
  }

  /** Restituisce il top della pila p */
  public static int top(Pila p) {
    if (!vuota(p))
      return p.elementi[p.numElementi - 1];
    else return -1;
  }

  /** Elimina un elemento dalla cima della pila p */
  public static Pila pop(Pila p) {
    if(!vuota(p))
      p.numElementi--;
    return p;
  }
}
```

ProvaPila.java

```
/** Programma per la prova della classe Pila. */
class ProvaPila {
  public static void main (String[] args) {
    Pila p1 = new Pila();
    p1 = Pila.push(p1,1);
    p1 = Pila.push(p1,2);
    p1 = Pila.push(p1,3);
    p1 = Pila.push(p1,4);
    System.out.println(Pila.top(p1));
    p1 = Pila.pop(p1);
    p1 = Pila.pop(p1);
    System.out.println(Pila.top(p1));
    p1 = Pila.push(p1,5);
    System.out.println(Pila.top(p1));
  }
}
```


public o non public...

- Per ora non fa differenza
- Se non c'è il `private` attributi e metodi sono visibili dal resto del programma al di fuori della classe
  - (la differenza la vedremo più avanti: con il `public` sono "più visibili")
- L'unico che è necessario è quello del main
  - Esercizio: dato un programma funzionante a vostra scelta, provate a togliere il `public`, a compilare e ad eseguire
- Il `private` è importante

Funzionale e procedurale

- Nella versione precedente i metodi sono funzioni
- Alcuni potrebbero essere procedure
  - push, pop
- Lavorano per effetto collaterale (side effect) sui parametri
  - (Parametri che non sono tipi primitivi: viene passato il riferimento per valore => modifiche al parametro attuale si ripercuotono sul parametro formale)
  - Esercizio: rivedere la versione funzionale ed eliminare i side effect

PilaP.java (procedurale)

```
class PilaP {
  ...
  /** push dell'elemento e sulla pila p */
  static void push (PilaP p, int e) {
    if (!piena(p))
      p.elementi[p.numElementi++] = e;
  }

  /** Elimina un elemento dalla cima della pila
   * p */
  static void pop(PilaP p) {
    if(!vuota(p))
      p.numElementi--;
  }
}
```


ProvaPilaP.java (procedurale)

```
/** Programma per la prova della classe PilaP. */
class ProvaPilaP {
  public static void main (String[] args) {
    PilaP p1 = new PilaP();
    PilaP.push(p1,1);
    PilaP.push(p1,2);
    PilaP.push(p1,3);
    PilaP.push(p1,4);
    System.out.println(PilaP.top(p1));
    PilaP.pop(p1);
    PilaP.pop(p1);
    System.out.println(PilaP.top(p1));
    PilaP.push(p1,5);
    System.out.println(PilaP.top(p1));
  }
}
```

Funzionale – procedurale (1/2)

- Definizione metodo

```
public static Pila push (Pila p, int e) {
  if (!piena(p))
    p.elementi[p.numElementi++] = e;
  return p;
}
```

```
public static void push (PilaP p, int e) {
  if (!piena(p))
    p.elementi[p.numElementi++] = e;
}
```

Funzionale – procedurale (2/2)

- Invocazione metodo

```
class ProvaPila {
  public static void main (String[] args) {
    Pila p1 = new Pila();
    p1 = Pila.push(p1,1);
    ...
    p1 = Pila.push(Pila.push(p1,1),2);
    p1 = Pila.pop(p1);
  }
}
```

```
class ProvaPilaP {
  public static void main (String[] args) {
    PilaP p1 = new PilaP();
    PilaP.push(p1,1);
    ...
    PilaP.pop(p1);
  }
}
```


Riassunto

- Comunicazioni di servizio, Provetta
- Riassunto
  - Programmazione strutturata
  - TDA (Tipo di dato astratto)
    - Evoluzione della programmazione strutturata
    - Occultamento delle informazioni
- Fine TDA
  - Aggiunte e puntualizzazioni
    - `this`, `toString`, procedurale vs. funzionale
  - Esempi
- Prossima lezione:
  - Interazione fra TDA
  - TDA -> OO (Object Oriented, Orientato agli oggetti)