

# Programmazione II

---

## 5. Astrazione dei Tipi (Definizione e Specifica) (PDJ 5-5.2, 5.8)

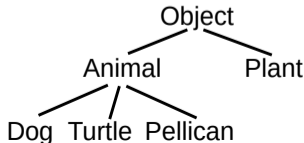
# Tipi di Dati nei linguaggi orientati a oggetti

Tipi di dati modellati da classi - un oggetto di una classe ha:

- **Uno stato**, ovvero delle informazioni che caratterizzano lo specifico oggetto di quella classe
- **Dei comportamenti** che tutti gli oggetti di quella classe sanno fare e che possono:
  - Restituire informazioni sullo stato dell'oggetto
  - Manipolare lo stato dell'oggetto

## Gerarchia delle classi

- Una classe  $A$  può **estendere** una classe  $B$ , aggiungendo attributi di stato o comportamenti
  - $A$  è **sottotipo** di  $B$  ( $A \prec B$ ). È una proprietà **transitiva**: se  $A \prec B$  e  $B \prec C$  allora  $A \prec C$
  - In Java, tutte le classi sono sottotipo di **Object** ed ereditano alcuni metodi basilari
  - A una var.  $b$  di tipo  $B$  è possibile assegnare un valore di tipo  $A \prec B$  (**Principio di Sostituzione di Liskov**)
  - Si dice che il **Tipo Apparente** di  $b$  è  $B$  mentre il **Tipo Concreto** è  $A$



## Sintassi delle Classi

La sintassi di una classe contiene sia dati che procedure

- **Definizione della classe:** ``(public) class NomeClasse {...}``
- **Attributi (fields):** ``<TipoDato> nome;``
  - Sono le variabili il cui valore rappresenta lo stato dell'oggetto
  - Saranno la **rappresentazione** del dato che andrà a finire in memoria
  - Si impostano quando si crea una nuova **istanza della classe (oggetto)**
- **Metodi di istanza** - servono per accedere agli attributi di un oggetto della classe:
  - **costruttori:** ``(public) NomeClasse() {...}`` per creare nuovi oggetti del tipo
  - **altri metodi:** ``(public) <TipoRitorno> nomeMetodo(...) {...}`` che agiscono sullo **stato**
- **metodi di classe (statici):** ``(public) static <TipoRitorno> nomeMetodo() {...}``

## Metodi di istanza

In che cosa **differiscono** i **metodi di istanza** dai **metodi statici**?

- I metodi di istanza **agiscono sugli attributi** dell'oggetto
- Quindi si possono usare solo su oggetti specifici (**istanze**) es:
  - `s.charAt(0)` restituisce il primo carattere di `s` che è una `String`
  - `sc.nextInt()` restituisce l'ultimo intero letto da `sc` che è di tipo `Scanner`
  - `l.size()` restituisce la dimensione di `l` che è di tipo `List`
- I metodi di istanza possono essere:
  - **Metodi di costruzione** (Creators) - **creano** un nuovo oggetto e ne **assegnano gli attributi**
  - **Metodi di mutazione** (Mutators) - **cambiano** lo stato dell'oggetto (modificando gli **attributi**)
  - **Metodi di osservazione** (Observers) - **restituiscono informazioni** sugli attributi dell'oggetto
  - **Metodi di produzione** (Producers) - restituiscono un altro oggetto del loro stesso tipo

## Utilizzo degli oggetti

Per utilizzare un nuovo oggetto del tipo  $T$  devo prima crearlo e poi interagire coi suoi metodi

- Si assegna ad una variabile del tipo  $T$  un oggetto restituito dal suo **costruttore**.

- ``Persona giovanni = new Persona("Giovanni", "M", 23, 1.74, "biondo");``

- Si possono usare i suoi **metodi di osservazione** per leggerne gli attributi

- ``double altezzaDiGiovanni = giovanni.altezza();``

- Si possono usare i suoi **metodi di mutazione** per cambiare gli attributi

- ``giovanni.siTinge("rosso");``

- **Non si accede** direttamente agli **attributi** perché l'implementazione può cambiare!

- **Si può** accedere direttamente agli attributi di altri **oggetti della stessa classe**
- Questo perché una classe ha sempre accesso completo al suo stesso codice

# Utilità dell'astrazione dei dati in OOP

Perchè ci serve astrarre sui dati?

Altrimenti sono vincolato su come il dato è implementato.

- Se in fase di implementazione stabilisco una **determinata implementazione** del tipo di dato
- **Qualsiasi modifica al tipo di dato** richiederebbe di:
  - Modificare **come implemento il tipo** dato (ovviamente)
  - Modificare **tutte le procedure** che **utilizzano questo tipo di dato**
  - Incluso le procedure fatte da altri che utilizzano il mio tipo di dato!
- Questo rende in pratica **difficile il riuso** dei tipi di dati e **limita la modularità**

# Utilità dell'astrazione dei dati in OOP

Esempio:

```
1 public class Studente {
2     int matricola;
3     String nome;
4 }
5
6 public class Corso {
7     String nome;
8     Year annoAccademico;
9     ArrayList<Studente> listaStudenti;
10 }
11
12 public class Segreteria {
13     public static ArrayList<Corso> corsiStudente(int matricola, ArrayList<Corso> corsi) {
14         ArrayList<Corso> result = new ArrayList<Corso>();
15
16         for(Corso c : corsi)
17             for(Studente s : c.listaStudenti)
18                 if s.matricola == matricola
19                     result.add(c); break;
20
21         return result;
22     }
23 }
```

# Utilità dell'astrazione dei dati in OOP

## Esempio:

```
1 public class Studente {
2     int matricola;
3     String nome;
4 }
5
6 public class Corso {
7     String nome;
8     Year annoAccademico;
9     HashMap<Integer, Studente> listaStudenti;
10 }
11
12 public class Segreteria {
13
14     //Devo cambiare il metodo dato il cambiamento alla RAPPRESENTAZIONE di Corso
15     public static ArrayList<Corso> corsiStudente(int matricola, ArrayList<Corso> corsi) {
16         ArrayList<Corso> result = new ArrayList<Corso>();
17         for(Corso c : corsi)
18             if(i.listaStudenti.containsKey(matricola))
19                 result.add(i);
20
21         return result;
22     }
23 }
```

# Utilità dell'astrazione dei dati in OOP

## Perchè è utile il paradigma a oggetti

Permette di fare l'astrazione sui dati, ovvero ignorare la rappresentazione effettiva

- Posso concentrarmi su cosa voglio **rappresentare** e sui **comportamenti** che il dato può avere
- Devo definire **cosa** è il dato e **cosa** fa, ma **non come** è implementato e **non come** lo fa
- Ci concentreremo su come specificare l'astrazione sui dati mediante:
  - **Astrazione per parametrizzazione** - metodi e parametri dei metodi
  - **Astrazione per specifica** - specifiche della classe e dei suoi metodi

# Specifica dei tipi di dati

## Specifiche

Per astrarre un tipo di dati prima decido che cosa voglio modellare e che cosa deve saper fare

- Definisco l'`//OVERVIEW` della classe, descrivendo il tipo che voglio modellare.
  - Voglio anche stabilire se le istanze saranno **mutabili** o no (ci torneremo dopo)
- Specifico i metodi della classe
  - Definisco le **istestazioni degli costruttori** sotto `//constructors`
  - Definisco il comportamento del tipo **specificando** tutti i **metodi** necessari sotto `//methods`
  - Per metodi necessari si intende tutti quelli che servono per operare sul tipo
- Solo quando avrò idea di tutto quello che il tipo dovrà fare penso alla **rappresentazione**
  - stabilisco gli attributi della classe sotto `//attributes`
- Posso finalmente iniziare a scrivere l'implementazione del tipo

# Specifica dei tipi di dati

## Esempio specifica Studente:

```
1 public class Studente {
2 //OVERVIEW: modella un tipo studente definito da matricola e nome
3
4 //constructors
5 public Studente(String nome, String cognome, int matricola)
6 //MODIFIES: this
7 //EFFECTS: inizializza this con i valori dati di nome, cognome, e matricola
8
9 //methods
10 public String getNome()
11 //EFFECTS: restituisce nome completo dello studente
12
13 public String getMatricola()
14 //EFFECTS: restituisce matricola dello studente
15
16 public boolean sostenutoEsame(String nomeEsame)
17 //EFFECTS: restituisce true se esamiDati contiene nomeEsame, false altrimenti
18
19 public int esameVoto(String esame) throws EsameNonDatoException
20 //EFFECTS: restituisce voto se esame in esamiDati, se no throws EsameNonDatoException
21
22 public void inserisciVoto(String esame, int voto) throws VotoEsisteException
23 //MODIFIES: this
24 //EFFECTS: inserisci esame, voto in this.esamiDati, se esiste throws VotoEsisteException
25 }
```

# Specifica dei tipi di dati

## Scelta degli attributi

Dopo la fase di specifica, prima di implementare un Tipo, devo decidere come **rappresentarlo**

- Devo scegliere quali attributi servono per memorizzare il tipo
  - Questi attributi saranno la **rappresentazione** del tipo
  - Devono essere **sufficientemente espressivi** per rappresentare **completamente** il tipo
  - La scelta deve considerare tutti i comportamenti che il tipo deve avere
  - Bisogna considerare anche l'**efficienza** e **semplicità d'uso** del tipo
- La visibilità degli attributi deve essere il più restrittiva possibile (**Encapsulation**)
  - se possibile **private** (ci sono altri tipi di visibilità utili nelle gerarchie di classi)
  - Così solo il codice della classe stessa potrà leggerli e scriverli (non devo esporre la rappresentazione)
- Tutte le interazioni esterne con gli attributi dovranno avvenire attraverso i metodi
  - Sarà compito della classe e dei suoi metodi assicurarsi che le eventuali modifiche siano valide
  - (argomento principale delle prossime lezioni)

# Specifica dei tipi di dati

## Esempio attributi Studente:

```
1 public class Studente {
2 //OVERVIEW: modella un tipo studente definito da matricola e nome
3
4 //attributes
5 private String nome;
6 private String cognome;
7 private int matricola;
8 private HashMap<String,Integer> esami;
9
10 //constructors
11 public Studente(String nome, String cognome, int matricola)
12 //MODIFIES: this
13 //EFFECTS: inizializza this con i valori dati di nome, cognome, e matricola
14
15 //methods
16 public String getName()
17 //EFFECTS: restituisce nome completo dello studente
18
19 public String getMatricola()
20 //EFFECTS: restituisce matricola dello studente
21
22 public boolean sostenutoEsame(String nomeEsame)
23 //EFFECTS: restituisce true se esamiDati contiene nomeEsame, false altrimenti
24
25 public int esameVoto(String esame) throws EsameNonDatoException
26 //EFFECTS: restituisce voto se esame in esamiDati, se no throws EsameNonDatoException
27
28 public void inserisciVoto(String esame, int voto) throws VotoEsisteExeption
29 //MODIFIES: this
30 //EFFECTS: inserisci esame, voto in this.esamiDati, se esiste throws VotoEsisteExeption
31 }
```

# Specifica dei tipi di dati

## Tipi mutabili vs immutabili

È possibile creare tipi i cui oggetti, una volta creati, non si possono modificare

- Questa scelta (**mutabilità** vs **immutabilità**) è da fare in fase di specifica
- Dipende dalle caratteristiche del dato che voglio modellare:
  - Gli oggetti **mutabili** sono preferibili se devono essere modificati **frequentemente**
  - Gli oggetti **immutabili** sono preferibili perché **più robusti** (non devo stare attento a errori di modifica)
- La scelta di creare un **tipo immutabile** si traduce in una serie di **vincoli**:
  - Non deve essere possibile **accedere direttamente** agli attributi (metterli **private**)
  - Gli attributi sono resi immodificabili una volta inizializzati (metterli **final**)
  - I metodi **non devono modificare** lo stato dell'oggetto (ma possono **restituire un oggetto nuovo**)
  - Anche la classe può essere dichiarata **final**. In questo modo non è possibile estenderla.
- È possibile rendere solo alcuni attributi immutabili (es: nome dello studente)
  - Il tipo è da considerarsi sempre mutabile
  - Gestione della mutabilità più facile -> considerare sempre se possibile rendere gli attributi immutabili

# Specifica dei tipi di dati

## Esempio Punto mutabile:

```
1 public class Punto {
2   //OVERVIEW: modella un punto mutabile sul piano cartesiano (a coordinate decimali)
3
4   //attributes
5   private double x;
6   private double y;
7
8   //constructors
9   public Punto()
10    //MODIFIES: this
11    //EFFECTS: inizializza this con x=0 e y=0
12
13   public Punto(double x, double y)
14    //MODIFIES: this
15    //EFFECTS: inizializza this con x e y
16
17   //methods
18   public String getX()
19    //EFFECTS: restituisce x
20
21   public String getY()
22    //EFFECTS: restituisce y
23
24   public void setX()
25    //MODIFIES: this
26    //EFFECTS: imposta x
27
28   public void setY()
29    //MODIFIES: this
30    //EFFECTS: imposta y
31
32   public double distanza(Punto p)
33    //EFFECTS: restituisce distanza euclidea tra this e p
34 }
```

# Specifica dei tipi di dati

## Esempio Punto immutabile:

```
1 public class Punto {
2 //OVERVIEW: modella un punto immutabile sul piano cartesiano (a coordinate decimali)
3
4 //attributes
5     private final double x;
6     private final double y;
7
8 //constructors
9     public Punto()
10        //MODIFIES: this
11        //EFFECTS: inizializza this con x=0 e y=0
12
13     public Punto(double x, double y)
14        //MODIFIES: this
15        //EFFECTS: inizializza this con x e y
16
17 //methods
18     public String getX()
19        //EFFECTS: restituisce x
20
21     public String getY()
22        //EFFECTS: restituisce y
23
24     public Punto copia()
25        //EFFECTS: crea una nuova copia di this
26
27     public double distanza(Punto p)
28        //EFFECTS: restituisce distanza euclidea tra this e p
29 }
```

# Programmazione II

---

## 5. Astrazione dei Tipi (Definizione e Specifica) (PDJ 5-5.2, 5.8)

Dragan Ahmetovic