



# Programmazione II

---

## 6. Astrazione dei Tipi (Implementazione e Metodi default) (PDJ 5.2-5.4)

# Implementazione dei metodi d'istanza

## Come implementare i metodi d'istanza?

L'implementazione dei metodi deve rispettare le specifiche e interagire con gli attributi

- Implementare i costruttori
  - Hanno il compito di inizializzare gli attributi, definendo lo stato dell'istanza
- Implementare gli altri metodi
  - Possibile strategia bottom-up: partire da quelli che saranno utilizzati da altri metodi
- Sarà possibile creare anche altri metodi, se ulteriore decomposizione utile
  - Saranno da specificare opportunamente
  - Se utili solo internamente devono essere **private**
- Modifiche alla rappresentazioni (attributi) sono ancora possibili
  - Durante l'implementazione se trovo una rappresentazione più semplice / performante
  - In un momento successivo alla implementazione se diventano disponibili soluzioni migliori

# Implementazione dei metodi d'istanza

## Costruttori

Sono dei **metodi di istanza** speciali che creano un nuovo oggetto **assegnando i suoi attributi**

- Il costruttore è il metodo che si invoca quando si **inizializza** un nuovo oggetto
  - Sintassi: 'Persona giovanni = new Persona("Giovanni Storti");'
  - È possibile l'**overloading** dei costruttori se devo passare dei parametri diversi
  - es: 'Persona giacomo = new Persona("Giacomo", "Poretti");'
- L'implementazione del costruttore solitamente **inizializza e assegna** gli attributi dell'oggetto
  - es: 'this.nomecompleto = nome + " " + cognome;'
  - la keyword 'this' è disponibile solo nei **metodi di istanza** e si usa per riferirsi all'oggetto stesso
  - Non è obbligatoria ma torna utile per **disambiguare tra nomi dei parametri e attributi**
  - es: 'this.matricola = matricola;'
- Metodi **factory**: metodi statici che fanno da wrapper ai Costruttori per creare nuove istanze
  - sono dei metodi di costruzione (come i costruttori), ma non sono dei costruttori
  - vantaggio: stesso nome del metodo da invocare anche nelle sottoclassi
  - vantaggio: aggiunta di ulteriore logica (es: logging), senza "sporcare il costruttore"

# Implementazione dei metodi d'istanza

Esempio overload costruttori:

```
1 public class Studente {  
2     //OVERVIEW: modella un tipo studente definito da matricola e nome  
3     private String nome;  
4     private String cognome;  
5     private int matricola;  
6     private boolean attivo = true;  
7     private HashMap<String, Integer> esami;  
8  
9     //constructors  
10    public Studente(String nome, String cognome, int matricola) {  
11        //MODIFIES: this  
12        //EFFECTS: inizializza this con nome, cognome e matricola  
13        this.nome = nome;  
14        this.cognome = cognome;  
15        this.matricola = matricola;  
16        this.esami = new HashMap<String, Integer>();  
17    }  
18  
19    public Studente(String nome, String cognome, int matricola, HashMap<String, Integer> esami) {  
20        //MODIFIES: this  
21        //EFFECTS: inizializza this con nome, cognome, matricola e esami trasferiti  
22        this.nome = nome;  
23        this.cognome = cognome;  
24        this.matricola = matricola;  
25        this.esami = esami;  
26    }  
27    ...
```

# Implementazione dei metodi d'istanza

## Metodi di Osservazione

I **metodi di osservazione** sono quelli che **ispezionano gli attributi** dell'oggetto

- **Getters:** metodi di osservazione che restituiscono lo stato di uno specifico attributo singolo
  - es: 'public String getNome()'
  - in molti IDE esiste la funzionalità di generazione automatica dei Getter
- Metodi di osservazione più complessi potrebbero richiedere delle computazioni
  - es: 'public boolean sostenutoEsame(String nomeEsame)'
- Possono anche lanciare eccezioni
  - es: 'public int esameVoto(String nomeEsame) throws EsameNonDataException'

# Implementazione dei metodi d'istanza

Esempio Metodi di Osservazione:

```
1 ...
2 //methods
3     public String getNomeCompleto() {
4         return this.nomeCompleto;
5     }
6
7     public boolean sostenutoEsame(String nomeEsame) {
8         //EFFECTS: restituisce true se esamiDati contiene nomeEsame, false altrimenti
9         if(this.esamiDati.get(nomeEsame) != null)
10             return true;
11
12         return false;
13     }
14
15     public int esameVoto(String esame) throws EsameNonDatoException {
16         //EFFECTS: restituisce voto se esame in esamiDati, se no throws EsameNonDatoException
17         if(this.esamiDati.get(esame) != null)
18             throw new EsameNonDatoException("Lo studente non ha sostenuto: " + esame);
19
20         return this.esamiDati.get(esame);
21     }
22 ...
```

# Implementazione dei metodi d'istanza

## Metodi di Manipolazione (o Mutazione)

I **metodi di manipolazione** servono a **cambiare gli attributi** dell'oggetto

- **Setters** impostano il valore di uno specifico attributo al valore del parametro fornito
  - es: `public void setMatricola(int matricola)`
- Possono esserci metodi di manipolazione senza parametri
  - es: `public void rinunciaStudi()`
- Se la modifica **non avviene** dovrebbero **lanciare eccezioni**
  - es: `public void inserisciVoto(String esame, int voto) throws VotoEsisteException`
- Possono anche ritornare un valore
  - es: `public int incrementaVoto(String esame, int incremento) throws VotoNonEsisteException`
- La modifica può dipendere dallo stato dell'oggetto

# Implementazione dei metodi d'istanza

## Esempio Metodi di Manipolazione:

```
1 ...
2     public String setMatricola(String matricola) {
3         this.matricola = matricola;
4     }
5
6     public void rinunciaStudi() {
7         //MODIFIES: this
8         //EFFECTS: rendi studente inattivo
9         this.attivo = false;
10    }
11
12    public void inserisciVoto(String esame, int voto) throws VotoEsisteException {
13        //MODIFIES: this
14        //EFFECTS: inserisci esame, voto in this.esamiDati, se esiste throws VotoEsisteException
15        if(this.esamiDati.get(esame) != null)
16            throw new VotoEsisteException("Lo studente ha sostenuto: " + esame)
17
18        this.esamiDati.put(esame, voto);
19    }
20
21    public int incrementaVoto(String esame, int incremento) throws VotoNonEsisteException {
22        //MODIFIES: this
23        //EFFECTS: applica incremento a esame e restituisci totale, se non esiste lancia VotoNonEsisteException
24        if(this.esamiDati.get(esame) == null)
25            throw new VotoNonEsisteException("Lo studente non ha sostenuto: " + esame)
26
27        int res = Math.max(33, this.esamiDati.get(esame)+incremento);
28        this.esamiDati.set(esame, res);
29
30        return res;
31    }
32 }
```

## Metodi di Produzione

I **metodi di produzione** generano dei nuovi oggetti partendo da uno esistente

- es: `public Matrice trasposta();`
  - Unico modo per operare con tipi immutabili senza creare esplicitamente nuove istanze
- **Copy Constructors**
  - Costruttori che prendono in ingresso un oggetto dello stesso tipo e lo copiano
  - es: `public static Matrice(Matrice m)`

# Implementazione dei metodi d'istanza

Esempio metodo di Produzione vs metodo factory:

```
1 public class Matrice { //OVERVIEW: crea e manipola matrici quadrate di numeri interi
2 //attributes
3 int[][] matrice;
4
5 //constructors
6 public Matrice(int d) {
7 //EFFECTS: inizializza this a una matrice di dimensione d
8 this.matrice = new int[d][d];
9 }
10
11 //methods
12 public void setValue(int x, int y, int valore) {
13 //REQUIRES: x e y < d
14 //MODIFIES: this
15 //EFFECTS: inserisci valore alla posizione x,y di this
16 this.matrice[x][y] = valore;
17 }
18
19 public Matrice trasposta() { //Questo e' un metodo di Produzione
20 //EFFECTS: restituisce matrice trasposta di this
21 Matrice res = new Matrice(this.matrice.length);
22 for(int i = 0; i < this.matrice.length; i++)
23   for(int j = 0; j < this.matrice[i].length; j++)
24     res[i][j] = this.matrice[j][i];
25 return res;
26 }
27
28 public static Matrice identita(int d) { //Questo invece e' un metodo di Costruzione (factory)
29 //EFFECTS: crea e restituisce una nuova matrice identita
30 Matrice id = new Matrice(d);
31 for(int i = 0; i < id.matrice.length; i++)
32   id.matrice[i][i] = 1;
33 return id;
34 }
35 }
```

# Metodi Default

## Ridefinizione (Overriding) dei metodi Default

- Tutte le classi in Java **ereditano alcuni metodi** basilari di 'Object'
  - `equals` - verifica l'**uguaglianza** tra due oggetti
  - `hashCode` - mappa gli oggetti a dei valori interi corrispondenti
  - `clone` - serve per creare una **copia** di un oggetto
  - `toString` - serve per dare una **rappresentazione** dell'oggetto come String
- Questi metodi possono essere **ridefiniti** per le esigenze dei tipi definiti
- ATTENZIONE: questa è ridefinizione (Overriding), NON sovraccaricamento (Overloading)
  - L'intestazione deve essere identica\* (tipi parametri, tipo output, eccezioni)
- Non serve la specifica per questi metodi - il loro significato è chiaro

## Equals

Si può ridefinire il concetto di **uguaglianza** tra due elementi.

- Liskov: uguaglianza possibile solo per oggetti **immutabili**:
  - Oggetti **immutabili** sono considerati uguali se si **comportano ugualmente**
  - Oggetti **mutabili** sono **simili** se esibiscono lo stesso stato (mediante osservatori)
- Nella prassi (e doc ufficiali) di Java si **considerano uguali in entrambi i casi**.
- Come implementare `boolean equals(Object o)`:
  - Controllare se non sono lo **stesso riferimento**: `if(this == o) return true;`
  - Controllare se `o` non è **null**: `if(o == null) return false;`
  - Controllare se `o` è dello **stesso tipo** di `this`: `if(getClass() != obj.getClass()) return false;`
  - Se sì, fare cast di `o` al tipo `T`: `T r = (T)o;`
  - Controllare se lo stato (attributi considerati) è uguale. es: `if(this.x != r.x) return false;`
  - Questo metodo dovrebbe essere **riflessivo, simmetrico, transitivo e coerente**
- Come verificare l'uguaglianza nell'ereditarietà? (questione principale della prossima lezione)

# Metodi Default

Esempio equals:

```
1 ...
2     @Override
3     public boolean equals(Object o) {
4         if(this == o) //se i due riferimenti puntano allo stesso oggetto in memoria
5             return true;
6
7         if(o == null) //se l'oggetto passato per fare il paragone == null
8             return false;
9
10        //if(o instanceof Studente) //se o istanza di Studente (o sottoclassi)
11        if(getClass() != o.getClass()) //se sono della stessa classe
12            return false;
13
14        Studente t = (Studente)o; //casting
15
16        if(this.matricola != t.matricola) //controllare lo stato d'interesse
17            return false;
18
19        //ATTENZIONE: ci possono essere altri controlli, anche complessi
20        //es: che una list contenga gli stessi elementi ma non per forza nello stesso ordine
21
22        return true;
23    }
24 ...
```

## hashCode

**hashCode** associa a un oggetto un intero che ne **funge da chiave nelle HashMap** o simili

- Quando si inserisce o si interroga la HashMap viene usato il hashCode della chiave
- Due oggetti uguali (stesso risultato di equals) **devono** avere lo stesso hashCode
- Però possono esserci delle **collisioni**, cioè oggetti diversi che producono lo stesso hashCode
- Come implementare `int hashCode()` (vecchio modo):
  - Partire da un numero primo e, a ogni passaggio:
  - Moltiplicare per un altro primo e sommare hashCode di ciascun attributo d'interesse
  - Il risultato viene distribuito su tutto il range degli interi
- Come implementare `int hashCode()` (nuovo modo):
  - Restituire il risultato di `int Objects.hash(Object ...)`
  - Autoboxing dei valori primitivi

# Metodi Default

## Esempio hashCode (old):

```
1 ...
2     @Override
3     public int hashCode() {
4         int result = 17; //risultato da ritornare, in genere si parte da un numero primo
5         int prime = 31; //ogni passaggio si moltiplica per un primo per spargere i risultati
6
7         result = prime * result + matricola; //se int si somma il valore, per altri serve cast
8
9 // Se volessi aggiungere controlli su oggetti dovrei richiamare il loro hashCode
10 //     if(nomeCompleto != null)
11 //         result = prime * result + nomeCompleto.hashCode();
12
13     return result;
14 }
15 ...
```

## Esempio hashCode (new):

```
1 ...
2     @Override
3     public int hashCode() {
4         return Objects.hash(matricola) //autoboxing
5         //return Objects.hash(matricola, nomeCompleto, esamiDati); //numero arbitrario di Object
6     }
7 ...
```

## clone

**clone** crea una copia dell'oggetto e serve per **evitare di modificare l'oggetto originale**.

- Per creare la copia **non basta** copiare tutti gli attributi (shallow copy - Default)
- Perché per gli attributi di tipo riferimento entrambi gli oggetti punterebbero allo stesso dato
- Questo è un **problema** per i tipi **mutabili** ma non lo è per quelli immutabili
- Attenzione, clone crea un **Object**, bisogna **castarlo**
- Utile anche nel Costruttore per non assegnare ad attributi dei parametri mutabili
- Come implementare **Object clone()**:
  - segnalare che la classe è clonabile es: `public Studente implements Clonable`
  - Usare il metodo **clone** di **Object** per fare una copia shallow
  - Assegnare a ciascun attributo mutabile una copia clonata similmente
  - Non serve per primitivi o immutabili
  - Se un attributo mutabile non è clonabile bisogna creare una copia con il costruttore

# Metodi Default

Esempio clone:

```
1 ...
2     @Override
3     public Object clone() {
4         Studente s = null;
5         try {
6             s = (Studente) super.clone();
7         } catch (CloneNotSupportedException e) {
8             s = new Studente();
9             s.matricola = this.matricola;
10            s.nomeCompleto = this.nomeCompleto;
11            s.attivo = this.attivo;
12        }
13
14        s.esamiDati = (HashMap<String, Integer>)this.esamiDati.clone(); //clone di esamiDati
15        //se non 'Cloneable' bisogna creare nuovo attributo con il Costruttore
16    }
17 ...
```

# Metodi Default

## toString

Utile fornire una **rappresentazione stringata** dell'oggetto.

- Viene usato quando si stampa l'oggetto via `System.out.println()`.
- L'implementazione default stampa il nome e l'**hashCode** del dato - poco informativo
- La rappresentazione dovrebbe essere **univoca**, diversa per oggetti che non sono equal.
- Dovrebbe specificare la classe e mostrare lo stato completo dello stato dell'oggetto
- es `Studente:{Nome: Mario Rossi, Matricola: 66543}`.
- Liskov usa questo metodo oltre alla sua applicazione normale (prossima volta)

## Esempio:

```
1 ...
2     @Override
3     public String toString() {
4         String res = "Studente: " + nomeCompleto + " matricola: " + matricola + "\n";
5
6         for(String key : esamiDati.keySet())
7             res += key + ": " + esamiDati.get(key);
8     }
9 ...
```

# Programmazione II

---

## 6. Astrazione dei Tipi (Implementazione e Metodi default) (PDJ 5.2-5.4)

Dragan Ahmetovic