Programmazione Orientata agli Oggetti

Code4Fun: Project *Lombok*

A cura di Valerio Cetorelli

Sommario

- Le annotazioni in Java
- Introduzione a Project Lombok
- Integrazione con Eclipse
- Aggiunta della libreria al progetto
- Le annotazioni più comuni
  - **`@Getter @Setter @Value @ToString @Data`**
- Alcune più complesse
  - **`@EqualsAndHashCode`**
  - **`@Builder`**
  - **`@With`**

Annotazioni

- Le **annotazioni** Java consentono di specificare delle (meta-)informazioni riguardanti il codice ai tool che lo processano. Ad es.
  - Al compilatore: **`@Override`**
    - controlla che un metodo sia effettivamente una *override* di un metodo in una superclasse
  - A JUnit: **`@Test`**
    - Segnala al *runner* di JUnit un metodo deve essere considerato come test-case

- Ne esistono molte altre!
  - ✔ È possibile anche definirne di nuove
  - ✔ Parte di librerie/tool di supporto

Project Lombok

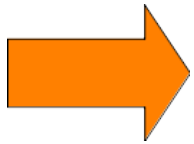
- Libreria open source basata sulla definizione di una serie di annotazioni *https://projectlombok.org/*
- ✔ Motivazione: *ridurre la verbosità di Java*
- ✔ Utilizza *annotazioni* ed *introspezione* per la generazione automatica di codice (ridondante)
- Evita di creare codice ripetitivo e prolisso
  - da manutenere
- Mantiene “pulite” le nostre classi

- Fa risparmiare molto tempo nella scrittura del codice

- ✔ N.B. un certo livello di verbosità favorisce l’apprendimento!

Project Lombok in Action

```java
public class Punto {
    private int x,y;
    public Punto (int x, int y) {
        this.x = x;
        this.y = y;
    }
    public void setX(int x) { this.x = x; }
    public void setY(int y) { this.y = y; }
    public int getX() { return this.x; }
    public int getY() { return this.y; }
    @Override
    public boolean equals(Object obj) {
        if (obj == null)
                return false;
        if (this.getClass() != obj.getClass())
            return false;
        Punto that = (Punto) obj;
        return this.getX() == that.getX()
                && this.getY() == that.getY();
    }
    @Override
    public int hashCode() {return x + y; }
    @Override
    public String toString() {
        return "Punto(x="+x+",y="+y+")";
    }
}
```



```java
import lombok.AllArgsConstructor;
import lombok.Data;

@AllArgsConstructor
@Data
public class Punto {

    private int x,y;

}
```

Integrazione con Eclipse (1)

- Lombok interagisce direttamente con il compilatore
- Dobbiamo quindi integrarlo nel nostro IDE (Eclipse)

1) Scaricare il `.jar` di Lombok

https://projectlombok.org/download

✔ trovare una posizione “definitiva” al file (>>)

2) Avviare il `.jar` (basta un doppio click)

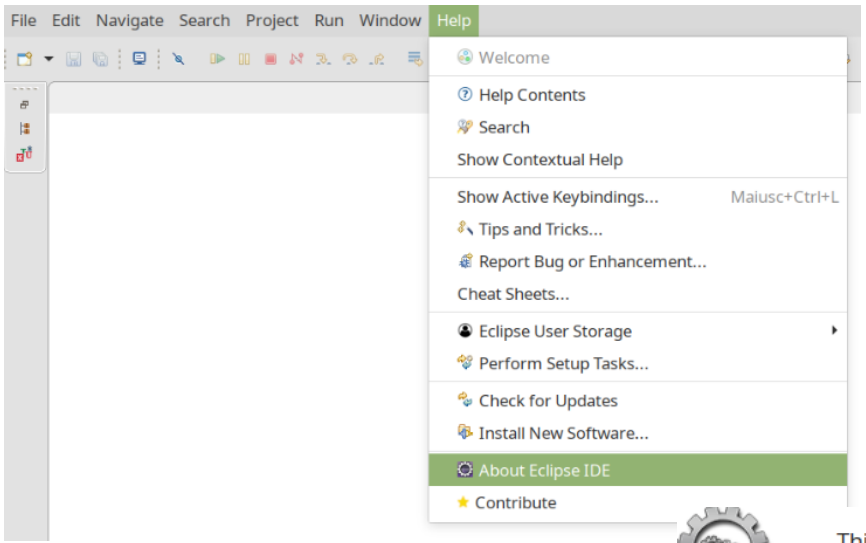
3) Selezionare la propria installazione di Eclipse

Integrazione con Eclipse (2)

4) Premere *Install/Update*

5) Controllare che *Eclipse* abbia riconosciuto Lombok come segue:

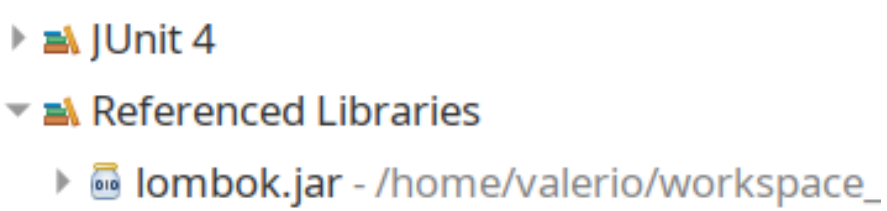
- ✔ Selezionare*: Help → About Eclipse IDE*
- ✔ Controllare il copyright: deve essere presente la versione di Lombok



Importare la Libreria `lombok.jar`

- Ora che il compilatore sa come gestire le annotazioni di Lombok dobbiamo aggiungere la libreria al nostro progetto
  - ➢ Va fatto per ogni progetto in cui vogliamo usarla

1) Click dx sulla cartella del progetto, vista *Package Explorer*
2) Seguire il percorso *Build Path* → *Configure Build Path...*
3) Click su *Add External JARs...*
4) Selezionare il file **`lombok.jar`** scaricato per l'installazione
   - ✔ Se ci limitiamo a spostare il file dentro la cartella del progetto, Eclipse smetterà di compilare il progetto
5) Click su *Apply and Close*
6) La libreria apparirà tra *Referenced Libraries* del progetto



@Getter/@Setter (1)

- Le annotazioni **@Getter** e **@Setter** su una variabile di istanza di nome *X* generano i metodi **getX()** e **setX()**
  - **isX()** invece di **getX()** se **boolean**
- I metodi generati sono sempre **public**
  - Può però essere specificato un **AccessLevel** alternativo:
  - **PUBLIC**, **PROTECTED**, **PACKAGE**, **PRIVATE**, e **NONE**

```
@Setter(AccessLevel.PROTECTED)
```

- Annotando l'*intera* classe vengono generati *getter&setter* per *tutte* le variabili d'istanza non statiche

```
@Getter
@Setter
public class Punto {
    …
```

@Getter/@Setter (2)

Senza Lombok

```
public class Cerchio {

  final private Punto centro;
  private int raggio;

  public Cerchio(Punto centro) {
    this.centro = centro;
  }

  public Cerchio(Punto centro, int raggio) {
    this(centro);
    this.raggio = raggio;
  }

  public int getRaggio() {
    return raggio;
  }

  public void setRaggio(int raggio) {
    this.raggio = raggio;
  }

  public Punto getCentro() {
    return centro;
  }

}
```

Con Annotazioni `@Getter` e `@Setter`

```
import lombok.Getter;
import lombok.Setter;

public class Cerchio {

  @Getter final private Punto centro;
  @Getter @Setter private int raggio;

  public Cerchio(Punto centro) {
    this.centro = centro;
  }

  public Cerchio(Punto centro, int raggio) {
    this(centro);
    this.raggio = raggio;
  }

}
```

Prova sul Campo: *Forme Geometriche*

- Per apprezzare come l'utilizzo delle annotazioni **`@Getter`** e **`@Setter`** riduca la verbosità del codice scritto provarle nelle classi **`Punto`**, **`Cerchio`** e **`Rettangolo`**
  - (<<) Riprendere l'esercitazione

    `POO-polimorfismo-esercitazione`

Annotazioni per Costruttori (1)

- Tre annotazioni per generare costruttori
  - **`@NoArgsConstructor`**
    - ✔ Costruttore vuoto e senza argomenti
  - **`@RequiredArgsConstructor`**
    - ✔ Costruttore con tutti i parametri necessari per una completa inizializzazione dei nuovi oggetti, ovvero, un parametro per ciascun variabile d'istanza:
      - ✔ dichiarata **`final`** e non inizializzata
      - ✔ marcata con l'annotazione **`@NonNull`** (>>)
  - **`@AllArgsConstructor`**
    - ✔ Costruttore con un parametro per ogni variabile d'istanza

Annotazioni per Costruttori (2)

Senza annotazioni per i costruttori

```
import lombok.*;
public class Cerchio {

  @Getter final private Punto centro;
  @Getter @Setter private int raggio;

  public Cerchio(Punto centro) {
    this.centro = centro;
  }

  public Cerchio(Punto centro, int raggio) {
    this(centro);
    this.raggio = raggio;
  }
}
```

Con annotazioni per creare costruttori

```
import lombok.*;

@RequiredArgsConstructor
@AllArgsConstructor
public class Cerchio {

  @Getter final private Punto centro;
  @Getter @Setter private int raggio;

}
```

@NonNull

- L'annotazione **`@NonNull`** può essere utilizzata per annotare:
  - Parametri di metodi o costruttori
    - Solleva una **`NullPointerException`** se il parametro passato è **`null`**

Senza Lombok

```
public class Cerchio {
  private Punto centro;
  public Cerchio(Punto centro) {
    if (centro == null)
      throw new NullPointerException(
        "centro cannot be null");
    this.centro = centro;
  }
}
```

Con annotazione `@NonNull`

```
import lombok.*;

public class Cerchio {
  private Punto centro;
  public Cerchio(@NonNull Punto centro) {
    this.centro = centro;
  }
}
```

  - Variabili d'istanza

    N.B. Nel caso in cui *Lombok*, per la presenza anche di altre annotazioni, deve generare dei metodi che coinvolgono una variabile d'istanza (ad es. **`@Setter`**, annotazioni per i costruttori, ecc. ecc.) annoterà automaticamente con **`@NonNull`** quei metodi

```
import lombok.*;

@AllArgsConstructor
public class Cerchio {

  @NonNull private Punto centro;

}
```

`@ToString` (1)

- L'annotazione **`@ToString`** genera automaticamente il metodo **`toString()`** utilizzando tutte le variabili d'istanza non statiche
  - I parametri dell'annotazione
  - **`@ToString.Exclude`** serve per specificare quali variabili di istanza *utilizzare* nella stampa finale
  - **`@ToString.Include`** serve per specificare quali variabili di istanza *escludere* dalla stampa finale

@ToString (2)

Senza Lombok

```
public class Borsa {

  //non viene stampato dal metodo toString()
  public final static int
    DEFAULT_PESO_MAX_BORSA = 10;
  private Map<String, Attrezzo> nome2attrezzo;
  private int pesoMax;
  private int pesoAttuale;

  @Override
  public String toString() {
    return "Borsa(nome2attrezzo="
      + this.nome2attrezzo.toString()
      +",pesoMax="+this.pesoMax
      +",pesoAttuale="+this.pesoAttuale"+")";
  }
  ...
}
```

Con annotazione @ToString

```
import lombok.ToString;


@ToString
public class Borsa {

  //non viene stampato dal metodo toString()
  public final static int
    DEFAULT_PESO_MAX_BORSA = 10;
  private Map<String, Attrezzo> nome2attrezzo;
  private int pesoMax;
  private int pesoAttuale;
  ...
}
```

`@EqualsAndHashCode` (1)

- Una classe può essere anche annotata con **`@EqualsAndHashCode`** per generare automaticamente i metodi **`equals()`** e **`hashCode()`**
- In automatico vengono utilizzati tutte le variabili d'istanza non **`static`** e non **`transient`** (cfr. **`java.io.`*`Serializable`***)
  - Si può precisare quali variabili di istanza utilizzare nella definizione dei metodi automaticamente generati specificandoli con
    - **`@EqualsAndHashCode.Include`**
    - **`@EqualsAndHashCode.Exclude`**

@EqualsAndHashCode (2)

Senza Lombok

```
public class Stanza {
  private String nome;
  protected Map<String, Attrezzo> nome2attrezzo;
  private Map<String, Stanza> direzione2stanzaAdiacente;
  @Override
  public int hashCode() {
    return this.getDirezione2stanzaAdiacente().hashCode() + this.getNome().hashCode()
           + this.getNome2attrezzo().hashCode(); }
  @Override
  public boolean equals(Object obj) {
    if (obj == null) return false;
    if (getClass() != obj.getClass()) return false;
    Stanza that = (Stanza) obj;
    return this.getDirezione2stanzaAdiacente().equals(that.getDirezione2stanzaAdiacente())
           && this.getNome().equals(that.getNome())
           && this.getNome2attrezzo().equals(that.getNome2attrezzo());
  }  ... }
```

Con Annotazione `@EqualsAndHashCode`

```
@EqualsAndHashCode
public class Stanza {
  private String nome;
  protected Map<String, Attrezzo> nome2attrezzo;
  private Map<String, Stanza> direzione2stanzaAdiacente;
… }
```

`@EqualsAndHashCode` & Estensione (1)

- Utilizzando l'annotazione **`@EqualsAndHashCode`** su una classe derivata bisogna prestare particolare attenzione
  - Per l'interazione con i metodi che definiscono il criterio di equivalenza nella classe base
- Di default i metodi **`equals()`** e **`hashCode()`** generati considerano solo le variabili d'istanza della classe annotata e *non* quelle della sua superclasse
- È necessario indicare esplicitamente (utilizzando **`callSuper=true`**) che si vogliono anche chiamare i metodi **`equals()`** e **`hashCode()`** della superclasse
  - Errore di compilazione se **`callSuper=true`** ma la classe annotata non estende alcuna classe (ovvero estende implicitamente **`java.lang.Object`**)

@EqualsAndHashCode & Estensione (2)

Senza Lombok

```
public class StanzaMagica extends Stanza {
  private int contatoreAttrezziPosati;
  private int sogliaMagica;
  @Override
  public int hashCode() {
    return super.hashCode() + this.getContatoreAttrezziPosati() + this.getSogliaMagica();
  }
  @Override
  public boolean equals(Object obj) {
    if (obj == null) return false;
    if (getClass() != obj.getClass()) return false;
    if(!super.equals(obj)) return false;
    StanzaMagica that = (StanzaMagica) obj;
    return this.getContatoreAttrezziPosati() == that.getContatoreAttrezziPosati()
           && this.getSogliaMagica() == that.getSogliaMagica();
  } ... }
```

Con annotazione @EqualsAndHashCode

```
@EqualsAndHashCode(callSuper=true)
public class StanzaMagica extends Stanza {
  private int contatoreAttrezziPosati;
  private int sogliaMagica;

 ...
}
```

`@Data`

- L'annotazione **`@Data`** è una abbreviazione per includere automaticamente le seguenti annotazioni:
  - **`@Getter`**/**`@Setter`**
  - **`@RequiredArgsConstructor`**
  - **`@EqualsAndHashCode`**
  - **`@ToString`**

✔Riconsiderare il primo esempio (<<)

`@Value`

- L'annotazione **`@Value`** è una variante *immutabile* di **`@Data`**
  - Tutte le variabili d'istanza vengono dichiarate **`private`** e **`final`**
  - I metodi **`setter`** non vengono generati
  - La classe stessa viene dichiarata **`final`** (non può essere estesa)

`@Builder` (1)

- L'annotazione **`@Builder`** genera il *builder* di una classe
- Viene generata una nuova classe chiamata **`NomeClasseAnnotata`**`Builder`
  - Es: **`Labirinto`** → **`Labirinto`**`Builder`
- Per ogni variabile d'istanza della classe annotata con **`@Builder`** Lombok genera un metodo del builder che si occupa di impostare il valore di quella specifica variabile
  - È anche possibile modificare i metodi del builder (>>)
- La classe annotata offrirà un metodo statico **`builder()`** per creare il proprio builder
- A sua volta il builder offrirà un metodo **`build()`** per ottenere l'oggetto che si è costruito

@Builder (2)

- Il builder della seguente classe **`Labirinto`** si ottiene con **`Labirinto.builder()`**
- La **`stanzaIniziale`** e la **`stanzaVincente`** si impostano con i metodi **`stanzaIniziale(Stanza stanzaIniziale)`** e **`stanzaVincente(Stanza stanzaVincente)`**

```
@Builder
public class Labirinto {

  private Stanza stanzaIniziale;
  private Stanza stanzaVincente;

  ...

}
```

Creazione tramite builder

```
Labirinto.builder()
  .stanzaVincente(stanzaVincente)
  .stanzaIniziale(stanzaIniziale)
  .build();
```

- Le stanze presenti nei labirinti potrebbero prevedere la presenza di attrezzi al loro interno
- Come gestirli tramite il builder generato da Lombok?
  - Modifichiamo il codice del builder in modo da esporre un metodo **`addAttrezzo()`**, che aggiunga un attrezzo nell'ultima stanza inserita

`@Builder`: Modifica del Builder (1)

- Creiamo all'interno della classe **`Labirinto`** una classe statica **`LabirintoBuilder`**
  - Al suo interno predisponiamo il metodo **`addAttrezzo()`**
  - ✔ Il builder di Lombok esporrà anche i metodi definiti in questo modo

```
@Builder
public class Labirinto {

    public static class LabirintoBuilder {

        public LabirintoBuilder addAttrezzo(Attrezzo attrezzo) {
            //Logica per inserire un attrezzo...
        }
    }
    ...
}
```

- Come gestiamo l'aggiunta di tutte le altre stanze nel labirinto?
  - Non solo iniziale e vincente
- Come impostiamo le stanze adiacenti?
- Eventuali stanze "particolari" (magica, buia, bloccata)?
- I personaggi?
- …

`@Builder`: Modifica del Builder (2)

- Otteniamo la seguente classe **`Labirinto`**

```
@Builder
public class Labirinto {
  private Stanza stanzaIniziale;
  private Stanza stanzaVincente;

  public static class LabirintoBuilder {
    private Stanza ultimaAggiunta; //ultima stanza aggiunta
    private Map<String,Stanza> nome2stanza; //le stanze del labirinto

    public LabirintoBuilder addAdiacenza(Stanza partenza, Stanza adiacente,
      String direzione) {
      //imposta adiacenza tra due stanze
    }
    public LabirintoBuilder addAttrezzo(Attrezzo attrezzo) {
      //aggiunge un attrezzo nell'ultima stanza aggiunta
    }
    public LabirintoBuilder addStanza(Stanza stanza) {
      //aggiunge una stanza al labirinto e la salva come ultima aggiunta
    }
    public LabirintoBuilder addStanzaMagica(Stanza stanza) {
      //aggiunge una stanza magica al labirinto e la salva come ultima aggiunta
    }
    ...
  }
  ...
}
```

`@Builder` e `diadia`

- Forti ed evidenti legami con la classe **`LabirintoBuilder`** sviluppata nel gioco **`diadia`** (vedi *HW3* e *HW4*)
- Ma la segnatura dei metodi è diversa!
  - I metodi di **`LabirintoBuilder`** in **`diadia`** ricevono stringhe
  - Quelli del *builder* qui costruito con Lombok degli oggetti **`Stanza`**
    - Ovvero il tipo delle variabili d'istanza
- Potremmo rifattorizzare il codice presente in **`diadia`** per
  - Annotare la classe **`Labirinto`** di **`diadia`** con l'annotazione **`@Builder`**
  - Rimuovere la classe introdotta contestualmente allo svolgimento degli homework **`LabirintoBuilder`**

@Builder e diadia: Confronto

Classe LabirintoBuilder senza Lombok (come negli HW)

```
public class LabirintoBuilder {

  private Labirinto labirinto;
  private Stanza ultimaAggiunta;
  private Map<String,Stanza> nome2stanza;

  public LabirintoBuilder() {
    this.labirinto = new Labirinto();
    this.nome2stanza = new HashMap<>();
  }

  public LabirintoBuilder addAdiacenza(
    String  partenza, String adiacente,
    String direzione) {...}

  public LabirintoBuilder addAttrezzo(
    String nome, int peso) {...}

  public LabirintoBuilder addStanza(String nome)
    {...}

  public LabirintoBuilder addStanzaMagica(
    String nome) {...}

  public Labirinto getLabirinto() {
    return this.labirinto;
  }
  ...
}
```

Classe Labirinto con Annotazione @Builder

```
@Builder
public class Labirinto {

  private Stanza stanzaIniziale;
  private Stanza stanzaVincente;

  public static class LabirintoBuilder {

    private Stanza ultimaAggiunta;
    private Map<String,Stanza> nome2stanza;

    public LabirintoBuilder addAdiacenza(
      Stanza partenza, Stanza adiacente,
      String direzione) {...}

    public LabirintoBuilder addAttrezzo(
      Attrezzo attrezzo) {...}

    public LabirintoBuilder addStanza(
      Stanza stanza) {...}

    public LabirintoBuilder addStanzaMagica(
      Stanza stanza) {...}
    ...
  }
  ...
}
```

`@Builder` e `diadia`: Conclusioni

- Finiamo per dover modificare e reimplementare praticamente tutti i metodi del nostro builder
  - Effettivo risparmio utilizzando l'annotazione **`@Builder`**?
  - Nel caso di **`diadia`** il codice di **`LabirintoBuilder`** ha una semantica troppo specifica e legata al dominio
  - ✔ Spesso costruire ex-novo *builder* con l'annotazione **`@Builder`** finisce per essere molto più sensato che cercare di rifattorizzare classi builder preesistenti
- Proviamo ad utilizzarla per una nuova classe di **`diadia`**

@Builder e diadia: Configurazione (1)

- L'annotazione **@Builder** è ottima per creare una classe che centralizza e gestisce gli elementi "configurabili" del gioco **diadia**
  - La classe **Configurazione** contiene informazioni come ad esempio il peso massimo della borsa, i CFU iniziali e non solo...
- La classe **Configurazione** deve comportarsi come la classe **IOConsole**
  - Deve essere creata una sola volta nel metodo **main()**
  - Il suo riferimento deve essere passato in tutto il codice

```
@Getter
@Builder
public class Configurazione {

    private int pesoMaxBorsa;
    private int cfuIniziali;
    //Altre configurazioni a seguire...

}
```

@Builder e diadia: Configurazione (2)

```
public static void main(String[] argc) {
    Configurazione config = Configurazione.builder()
        .cfuIniziali(20)
        .pesoMaxBorsa(30)
        .build();
    IO ioConsole = new IOConsole();
    Labirinto labirinto = new LabirintoBuilder()
        ... //creazione del labirinto
        .getLabirinto();
    DiaDia gioco = new DiaDia(ioConsole, labirinto, config);
    gioco.gioca();
}
```