UNIVERSITÀ DEGLI STUDI DELL'AQUILA

Laboratorio di Programmazione ad Oggetti

Ph.D. Juri Di Rocco
juri.dirocco@univaq.it
https://jdirocco.github.io/

Sommario

Interfacce

- Sintassi
- Ereditarietà
- Costanti (static final)
- Metodi di default
- Ereditarietà metodi default
- Metodi Statici
- Metodi Privati

Interfacce (1)

- Permette di stabilire la *forma* per una classe
  - Nomi metodi, elenchi argomenti, e tipi restituiti
  - Non sono definiti i corpi dei metodi
- Può essere considerata come una classe astratta pura
  - Classe che contiene **solo** metodi astratti e **non** contiene variabili di istanza e metodi (tranne quelli di default)
- Può contenere campi ma sono soltanto `static` e `final`
- Definisce un protocollo di comportamento che può essere implementato da una qualsiasi classe che si trova nella gerarchia

Interfacce (2)

- Classe che implementa un'interfaccia si impegna ad implementare **tutti** i metodi definiti nell'interfaccia ovvero accetta di rispondere a determinati comportamenti
- Per dichiarare una interfaccia si utilizza la parola chiave `interface`
- Classe che si adatta ad una particolare interfaccia/e utilizza `implements`
- E' possibile dichiarare variabili di tipo interfaccia
- **Non** è possibile creare oggetti

Interfacce (3)

```
[ public ] interface <nome dell'interfaccia>
        [extends interfaccia1, interfaccia2, ..] {


[<dichiarazione di costanti>]

[<dichiarazione dei metodi>]


}
```

Corpo dell'interfaccia

Interfacce (4): Esempio 1

```
public interface Saluto {
    String CIAO ="Ciao";
    String BUONGIORNO ="Buongiorno";
    //. . .
    /* protected */ void saluta();  // Questo esempio non compila
                                       con protected!
}

public class SalutoImpl implements Saluto {
    public void saluta(){
        System.out.println(CIAO);
    }
}
Saluto s = new SalutoImpl();
```

Interfacce (5): Esempio 2

```
public interface StrumentoMusicale {

        void suona(Nota n); // public automatico

        String quale(); // public automatico

        void accorda(); // public automatico

}
```

Interfacce (6): Esempio 2

```
public class Nota {
	private String nome;
	private Nota(String nome) {
		this.nome = nome;
	}

	public String toString() {
		return nome;
	}
	public static final Nota DO = new Nota("Do"), RE = new Nota("Re"),
			MI = new Nota("Mi"), FA = new Nota("Fa"),
			LA = new Nota("La"), SI = new Nota("Si"),
			SOL = new Nota("Sol"),
			LA_DIESIS = new Nota("Do Diesis"),
			DO_DIESIS = new Nota("Do Diesis");
}
```

Interfacce (7): Esempio 2

```
public abstract class StrumentoFiato implements StrumentoMusicale {
      public String quale() {
            return "StrumentoFiato";
      }
}

public abstract class StrumentoPercussione implements StrumentoMusicale {
      public String quale() {
            return "StrumentoPercussione";
      }
}
```

Interfacce (8): Esempio 2

```
public abstract class StrumentoCorda implements StrumentoMusicale {
	public String quale() {
		return "StrumentoCorda";
	}
}

public class Ottone extends StrumentoFiato {
	public String quale() {
		return super.quale() + ": Ottone";
	}
	public void suona(Nota n) {
		System.out.println("Ottone.suona() " + n);
	}
	public void accorda() {
		System.out.println("Ottone.accorda()");
	}
}
```

Interfacce (9): Esempio 2

```
public class Clarinetto extends StrumentoFiato {
      public String quale() {
            return super.quale() + ": Clarinetto";
      }
      public void suona(Nota n) {
            System.out.println("Clarinetto.suona() " + n);
      }
      public void accorda() {
            System.out.println("Clarinetto.accorda()");
      }
}
```

Interfacce (10): Esempio 2

```
public class Chitarra extends StrumentoCorda {
      public String quale() {
            return super.quale() + ": Chitarra";
      }
      public void suona(Nota n) {
            System.out.println("Chitarra.suona() " + n);
      }
      public void accorda() {
            System.out.println("Chitarra.accorda()");
      }
}
```

Interfacce (11): Esempio 2

```
public class Batteria extends StrumentoPercussione {
      public String quale() {
            return super.quale() + ": Batteria ";
      }
      public void suona(Nota n) {
            System.out.println(" Batteria.suona() " + n);
      }
      public void accorda() {
            System.out.println(" Batteria.accorda()");
      }
}
```

Interfacce (12): Esempio 2

```
public class TestStrumentoMusicale {
	private static void accorda(StrumentoMusicale i) {
		i.suona(Nota.LA);
	}
	private static void accordaOrchestra(StrumentoMusicale[] e) {
		for (StrumentoMusicale sm : e) {
			accorda(sm);
		}
	}
	public static void main(String[] args) {
		StrumentoMusicale[] orchestra = { new Chitarra(),
		                                  new Ottone(),
		                                  new Clarinetto(),
		                                  new Batteria() };
		accordaOrchestra(orchestra);
	}
}
```

Interfacce (13): Esempio 2 rivisitato

```
public class TestStrumentoMusicale {
	private static void accordaOrchestra(StrumentoMusicale[] e) {
		for (StrumentoMusicale sm : e) {
			sm.accorda();
		}
	}
	public static void main(String[] args) {
		StrumentoMusicale[] orchestra = { new Chitarra(),
						  new Ottone(),
						  new Clarinetto() };
		accordaOrchestra(orchestra);
	}
}
```

Interfacce (14)

› E' possibile che una classe implementi diverse interfacce (simile ereditarietà multipla)

```
interface CanFight {
  void fight();
}
interface CanSwim {
  void swim();
}
interface CanFly {
  void fly();
}
class ActionCharacter {
  public void fight() {}
}
```

Interfacce (15)

```
class Hero extends ActionCharacter implements CanFight, CanSwim, CanFly {
  public void swim() {}
  public void fly() {}
}
public class Adventure {
  public static void t(CanFight x) { x.fight(); }
  public static void u(CanSwim x) { x.swim(); }
  public static void v(CanFly x) { x.fly(); }
  public static void w(ActionCharacter x) { x.fight(); }
  public static void main(String[] args) {
    Hero h = new Hero();
    t(h); // Treat it as a CanFight
    u(h); // Treat it as a CanSwim
    v(h); // Treat it as a CanFly
    w(h); // Treat it as an ActionCharacter
  }
}
```

Interfacce (16)

- Se si implementano diverse interfacce è possibile avere problemi con i metodi dichiarati
- Esempio

```
interface I1 {
     void f();
}
interface I2 {
     int f(int i);
}
interface I3 {
     int f();
}
```

Interfacce (17)

```
class C {
    public int f() {
         return 1;
    }
}

class C2 implements I1, I2 {
  public void f() {} //metodo di I1
  public int f(int i) { return 1; } //Metodi di I2
}

class C3 extends C implements I2 {
  public int f(int i) { return 1; } // I2: int f(int i)
                                    //metodo int f() di C ereditato e overload
}
```

Interfacce (18)

```
class C4 extends C implements I3 {
  // I3: int f(), C: int f()
  public int f() { return 2; }
}

class C5 extends C implements I1 {
      // I1: void f(), C: int f() → errore in compilazione
      public void f() {}
}
interface I4 extends I1, I3 {
  //I1: void f(), I3: int f() → errore in compilazione
}
```

Interfacce (19)

- E' possibile dichiarare un'interfaccia estendendola da un'altra/e (ereditarietà multipla)
- Esempio

```
interface Monster {
  void menace();
}

interface DangerousMonster extends Monster {
  void destroy();
}

interface Lethal {
  void kill();
}
```

Interfacce (20)

```
class DragonZilla implements DangerousMonster {
  public void menace() {}
  public void destroy() {}
}

interface Vampire extends DangerousMonster, Lethal {
  void drinkBlood();
}

class VeryBadVampire implements Vampire {
  public void menace() {}
  public void destroy() {}
  public void kill() {}
  public void drinkBlood() {}
}
```

Interfacce (21)

```
public class HorrorShow {
  static void u(Monster b) { b.menace(); }
  static void v(DangerousMonster d) {
    d.menace();
    d.destroy();
  }
  static void w(Lethal l) { l.kill(); }
  public static void main(String[] args) {
    DangerousMonster barney = new DragonZilla();
    u(barney);
    v(barney);
    Vampire vlad = new VeryBadVampire();
    u(vlad);
    v(vlad);
    w(vlad);
  }
}
```

Interfacce (22)

- E' possibile dichiarare all'interno delle interfacce attributi che sono soltanto `public static final` (Costanti)
- Esempio

```
public interface Months {
  int
    JANUARY = 1, FEBRUARY = 2, MARCH = 3,
    APRIL = 4, MAY = 5, JUNE = 6, JULY = 7,
    AUGUST = 8, SEPTEMBER = 9, OCTOBER = 10,
    NOVEMBER = 11, DECEMBER = 12;
}
```

Interfacce (23)

```
public interface Months {
  Month JANUARY = new Month(1, “JANUARY”);
  Month FEBRUARY = new Month(2, “FEBRUARY”);
  Month MARCH = new Month(3, “MARCH”);
  Month APRIL = new Month(4, “APRIL”);
  Month MAY = new Month(5, “MAY”);
  Month JUNE = new Month(6, “JUNE”);
  Month JULY = new Month(7, “JULY”);
  Month AUGUST = new Month(8, “AUGUST”);
  Month SEPTEMBER = new Month(9, “SEPTEMBER”);
  Month OCTOBER = new Month(10, “OCTOBER”);
  Month NOVEMBER = new Month(11, “NOVEMBER”);
  Month DECEMBER = new Month(12, “DECEMBER”);
}
```

Interfacce (24)

```
public class Month {
    private int month;
    private String name;

    public Month(int month, String name ) {
       this.month = month;
       this.name = name;
    }

    public int getMonth() {
       return month;
    }
    public String getName() {
       return name;
    }
}
```

Nota: non sono definiti i metodi setter → oggetti sono immutabili

Metodi default (1)

- Java 8 ha introdotto la possibilità di dare una implementazione predefinita ai metodi di una interfaccia (keyword `default`)
- Metodi di default permettono aggiungere nuove funzionalità ad interfacce di librerie assicurando compatibilità di codice scritto con versioni precedenti

Metodi default (2)

› Esempio

```
public interface Solista {
    default void eseguiAssolo() {
      //Scala maggiore in DO
      System.out.println("DO RE MI FA SOL LA SI");
    }
}
public class Musicista implements Solista {
    //Non serve implementare il metodo eseguiAssolo()

}
```

Metodi default (3)

- Quando viene estesa un'interfaccia che contiene un metodo di default è possibile
  1. Non menzionare il metodo di default, in tal caso viene ereditato
  2. Ridichiarare il metodo di default che diventa astratto
  3. Ridefinire il metodo di default che viene sovrascritto
- Esempio (1)

```
public interface SolistaRock extends Solista {
}
```

- Qualsiasi classe che imlementa l'interfaccia `SolistaRock` eredita il metodo di default `Solista.eseguiAssolo`

```
public class RogerWaters implements SolistaRock {
}
```

Metodi default (4)

› Esempio (2)

```
public interface SolistaBlues extends Solista {
   public void eseguiAssolo();
}
```

› Qualsiasi classe che implementa l'interfaccia `SolistaBlues` dovrà implementare il metodo `eseguiAssolo`

```
public class BBKing implements SolistaBlues {
   public void eseguiAssolo() {
   }
}
```

Metodi default (5)

› Esempio (3)

```
public interface SolistaCountry extends Solista {
    default public void eseguiAssolo() {
       //Un'altra implementazione
    }
}
```

› Qualsiasi classe che implementa interfaccia `SolistaCountry` userà implementazione specificata nell'interfaccia

```
public class BobDylan implements SolistaCountry {
}
```

Ereditarietà Metodi default (1)

- Metodi di default e metodi astratti nelle interfacce sono ereditati come i metodi di istanza
- Quando il supertipo di una classe o interfaccia fornisce diversi metodi di default con la stessa segnatura, il compilatore segue le seguenti regole per risolvere i conflitti

Ereditarietà Metodi default (2)

- Metodi istanza definiti nelle classi vengono prima dei metodi di default

```
public class Horse {
    public String identifyMyself() {
        return "I am a horse.";
    }
}
public interface Flyer {
    default public String identifyMyself() {
        return "I am able to fly.";
    }
}
public interface Mythical {
    default public String identifyMyself() {
        return "I am a mythical creature.";
    }
}
```

Ereditarietà Metodi default (3)

```
public class Pegasus extends Horse implements Flyer, Mythical {
    public static void main(String[] args) {
        Pegasus myApp = new Pegasus();
        System.out.println(myApp.identifyMyself());
    }
}
```

OUTPUT

```
I am a horse
```

Ereditarietà Metodi default (4)

› Metodi che sono già sovrascritti da altri candidati sono ignorati (Si verifica quando supertipi condividono un antenato comune)

```
public interface Animal {
    default public String identifyMyself() {
        return "I am an animal.";
    }
}

public interface EggLayer extends Animal {
    default public String identifyMyself() {
        return "I am able to lay eggs.";
    }
}
public interface FireBreather extends Animal { }
```

Ereditarietà Metodi default (5)

```
public class Dragon implements EggLayer, FireBreather {
    public static void main (String[] args) {
        Dragon myApp = new Dragon();
        System.out.println(myApp.identifyMyself());
    }
}
```

OUTPUT

```
I am able to lay eggs
```

Ereditarietà Metodi default (6)

› Se due o più metodi di default definiti indipendentemente sono in conflitto con un metodo astratto, il compilatore produce errore. E' necessario effettuare override esplicito del metodo

```
public interface OperateCar {
    // ...
    default public int startEngine(EncryptedKey key) {
        // Implementation
    }
}
public interface FlyCar {
    // ...
    default public int startEngine(EncryptedKey key) {
        // Implementation
    }
}
```

Ereditarietà Metodi default (7)

- Classe che implementa sia `OperateCar` che `FlyCar` deve sovrascrivere il metodo `startEngine` altrimenti da errore in compilazione
- Si può invocare l'implementazione di default usando `super`

```
public class FlyingCar implements OperateCar, FlyCar {
    // ...
    public int startEngine(EncryptedKey key) {
        FlyCar.super.startEngine(key);
        OperateCar.super.startEngine(key);
    }
}
```

Metodi Statici

› E' possibile definire metodi statici all'interno di una interfaccia

```
public interface TimeClient {
    // ...
    static public ZoneId getZoneId (String zoneString) {
        …
    }

    default public ZonedDateTime getZonedDateTime(String zoneString) {
        …
    }
}
```

Metodi privati (1)

- A partire da **Java 9** è possibile dichiarare metodi privati sia statici che non
- Metodo privato può essere invocato soltanto all'interno di un metodo privato dell'interfaccia
- Metodi privati statici possono essere invocati all'interno di metodi privati statici e non dell'interfaccia
- Ovviamente i metodi privati non vengono ereditati

Metodi privati (2)

```
public interface CustomInterface {
   public abstract void method1();
   public default void method2() {
        method4();  //private method inside default method
        method5();  //static method inside other non-static method
        System.out.println("default method");
    }
    public static void method3() {
        method5(); //static method inside other static method
        System.out.println("static method");
    }
    private void method4(){
        System.out.println("private method");
    }
    private static void method5(){
        System.out.println("private static method");
    }
}
```

Metodi privati (3)

```
public class CustomClass implements CustomInterface {

    @Override
    public void method1() {
        System.out.println("abstract method");
    }

    public static void main(String[] args){
        CustomInterface instance = new CustomClass();
        instance.method1();
        instance.method2();
        CustomInterface.method3();
    }
}
```