Programmazione orientata agli oggetti (OOP)

Il meccanismo dell'ereditarietà

Esempio di ereditarietà

Si vuole utilizzare la classe generica `Person` come base di partenza per sviluppare una nuova classe `Student`, che rappresenta una specializzazione di `Person` (nel senso che possiede tutte le caratteristiche di `Person` ma ha anche qualcosa in più che caratterizza uno studente, come ad esempio il numero di matricola).
In questo contesto, `Person` si indica come la classe base o la superclasse, mentre `Student` si indica come la classe derivata, o sottoclasse (subclass). Inoltre `Student` rappresenta una specializzazione della classe `Person`, mentre `Person` è viceversa una generalizzazione della classe `Student`.

```
public class Person {
    protected String name;
    protected String surname;
    protected LocalDate birth;
    public Person(String name, String surname,
                  LocalDate birth) {
        this.name = name;
        this.surname = surname;
        this.date = date;
    }
    public LocalDate getBirth() {
        return this.date;
    }
    public String getInfo() {
        return this.name+" "+this.surname+
         " "+this.birth;
    }
}
```

```
public class Student extends Person {
    private String matricola;
    public Student(String name, String surname,
                   Date birth, String matricola) {
        super(name, surname, birth);
        this.matricola = matricola;
    }
    public String getMatricola() {
        return this.matricola;
    }
    @Override
    public String getInfo() {
        return super.getInfo()+" "+this.matricola;
    }
}
```

Concetti e termini chiave legati all'ereditarietà

- "riutilizzo di codice già esistente, estendendone le funzionalità": creare nuove classi a partire da classi già esistenti
- classe, metodi, attributi.. (elementi generici in OOP)
- superclasse/classe base, sottoclasse/classe derivata
- modificatore d'accesso **protected:** accesso diretto ai membri della classe base dalla classe derivata, ma non dall'esterno
- keyword **super**: indica la classe base (consente di utilizzarne i metodi e i costruttori della classe base all'interno della classe derivata)
- **overriding**: ridefinizione di un metodo della classe base all'interno della classe derivata; il nuovo metodo deve avere la stessa firma del vecchio (**firma** = nome, tipo di ritorno, numero e tipo di parametri);
  è possibile cambiare il modificatore di accesso purché sia meno restrittivo di quello del vecchio metodo (ad esempio di può passare da protected a public ma non viceversa)
- gerarchie di classi e **compatibilità dei tipi**: un oggetto di una classe derivata è anche del tipo della classe base;
  Esempio pratico:

supponiamo che la classe `Student` derivi dalla classe `Person`; normalmente definiremmo un oggetto della classe `Student` nel seguente modo:

```
Student s = new Student();
```

Ma è anche possibile scrivere questo:

```
Person p = new Student();
```

Le due definizioni non sono però equivalenti. Supponiamo che nella classe Student sia stato definito un metodo public `getMatricola()`, non presente nella classe base. In tal caso è possibile scrivere

```
s.getMatricola();
```

ma non

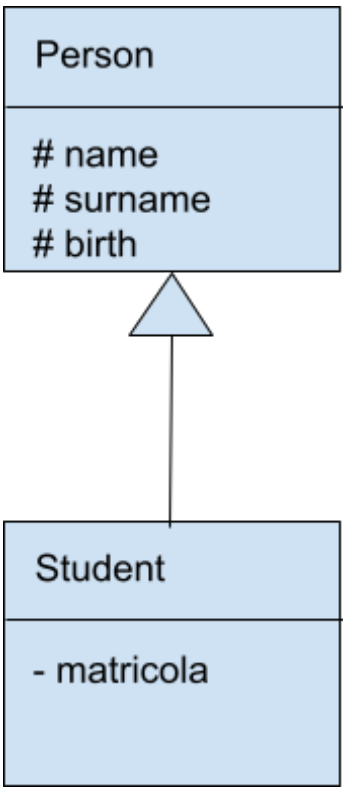
```
p.getMatricola();  // error!
```

perchè l'oggetto puntato da `p`, anche se è effettivamente un'istanza della classe `Student`, può essere utilizzato solo tramite l'interfaccia della classe `Person`;

C'è anche un'altra caratteristica interessante di questo meccanismo: se il metodo `public getInfo()` della classe base fosse ridefinito nella classe derivata `Student` (overriding), potrei scrivere nel mio programma l'istruzione `p.getInfo()` ? Sì, perché `getInfo()` fa parte dell'interfaccia della classe base `Person`. Ma quale versione di `getInfo()` verrà eseguita? Viene eseguito il metodo ridefinito nella classe derivata, poiché è la vera classe di appartenenza dell'oggetto.

- **polimorfismo (dinamico):** consiste nell'ottenere "*comportamenti*" diversi chiamando lo stesso metodo, a seconda della vera classe di appartenenza dell'oggetto; il polimorfismo è possibile grazie alla compatibilità dei tipi tra classe derivata e classe base, e al meccanismo dell'overriding.
  (*) Il polimorfismo statico non è legato al meccanismo dell'ereditarietà e si realizza con l'overloading dei metodi. Si dice statico perché di fatto, a differenza di quello dinamico, è sempre possibile stabilire a priori quali istruzioni saranno eseguite in corrispondenza della chiamata di un metodo (i metodi overloaded si distinguono per numero e/o tipo di parametri)

UML Diagram



In Java, ereditarietà singola: una classe derivata deriva da una sola classe base

In C++, ereditarietà multipla: una classe derivata può avere più di una classe base

Java modifiers

Access level modifiers determine whether other classes can use a particular field or invoke a particular method. There are two levels of access control:

- At the top level—`public`, or *package-private* (no explicit modifier).
- At the member level—`public`, `private`, `protected`, or *package-private* (no explicit modifier).

A class may be declared with the modifier `public`, in which case that class is visible to all classes everywhere. If a class has no modifier (the default, also known as *package-private*), it is visible only within its own package (packages are named groups of related classes)

At the member level, you can also use the `public` modifier or no modifier (*package-private*) just as with top-level classes, and with the same meaning. For members, there are two additional access modifiers: `private` and `protected`. The `private` modifier specifies that the member can only be accessed in its own class. The `protected` modifier specifies that the member can only be accessed within its own package (as with *package-private*) and, in addition, by a subclass of its class in another package.

The following table shows the access to members permitted by each modifier.

Modifier	Same Class	Package	Subclass	World
`public`	**accessible**	**accessible**	**accessible**	**accessible**
`protected`	**accessible**	**accessible**	**accessible**	**no access**
<none>	**accessible**	**accessible**	**no access**	**no access**
`private`	**accessible**	**no access**	**no access**	**no access**

Class members may be declared with one or more modifiers which affect its runtime behavior:

- Access modifiers: `public`, `protected`, and `private`
- Modifier requiring override: `abstract`
- Modifier restricting to one instance: `static`
- Modifier prohibiting value modification: `final`
- Modifier forcing strict floating point behavior: `strictfp`

Final Keyword

- `final` variable = costante
- `final` method = non può essere soggetto a overriding nelle classi derivate
- `final` class = non è possibile creare classi derivate da una classe definita come final

Abstract Classes and Methods

- `abstract` class:
  - può essere utilizzata **solo come classe base** per creare classi derivate
  - **non si possono istanziare oggetti di una classe abstract**
  - tutti gli oggetti delle classi derivate dalla classe abstract possono essere riferiti da una variabile (puntatore) del tipo della classe astratta, come avviene normalmente nell'ereditarietà per il principio di compatibilità dei tipi
  - può contenere uno o più metodi `abstract`
- `abstract` method: è un metodo privo di body, per cui è obbligatorio l'overriding nelle classi derivate

Esempio:

```
public abstract class Pippo {
    public int foo(){
       …
    }
    public void fee(String s) {
       …
    }
    public abstract double boo(int n, double d);
}

public class Pluto extends Pippo {
    public double boo(int n, double d) {
       …
    }
}

Pluto p = new Pluto(); // OK: sistema classico
Pippo p = new Pluto(); // OK: downcast
Pippo p = new Pippo(); // NO: Pippo è abstract
```

Java Interface

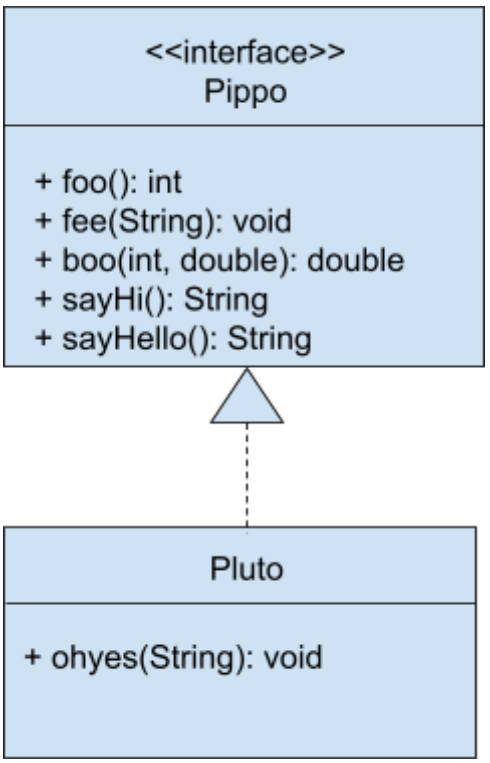
Un'interfaccia Java normalmente contiene le firme dei metodi `public` ed eventualmente delle costanti globali. Gli attributi definiti all'interno di un'interfaccia sono considerati automaticamente e implicitamente come `public static final`.
Invece tutti i metodi definiti all'interno di un'interfaccia sono implicitamente considerati `public` e `abstract`. Di conseguenza, tutti i metodi presenti in un'interfaccia devono essere obbligatoriamente definiti nelle classi che implementano quella interfaccia. Un'eccezione a questa regola è rappresentata dai metodi contrassegnati con la keyword `default`, per i quali è presente un'implementazione iniziale (quindi a differenza di quelli `abstract`, possiedono un body). A partire da Java 9, in un'interfaccia si possono definire anche dei metodi `private`, che potranno essere impiegati solo al suo interno e non saranno quindi accessibili alle classi che la implementano (potranno essere utilizzati ad esempio nelle implementazioni dei metodi `default` dell'interfaccia).

```
public interface Pippo {
    public static final int MaxVal = 0;
    String LabelName = "Mah";

    public abstract int foo(); // abstract è ridond
    public void fee(String s);
    public double boo(int n, double d);
    public default String sayHi() {
       System.out.println("Hi "+util());
    }
    public default String sayHello() {
       System.out.println("Hello "+util());
    }
    private String util() { // ad uso interno
       return "CheSuperSballo";
    }
}
```

Un'interfaccia serve come riferimento per le classi che la implementano: una classe che implementa un'interfaccia (con la keyword `implements`) è obbligata a implementare ogni metodo presente nell'interfaccia (tranne quelli `default`, per i quali può comunque effettuare l'override). Oltre ai metodi dell'interfaccia, la classe può anche implementare altri metodi.

```
public class Pluto implements Pippo {
    public int foo() {
        ...
    }
    public void fee(String s){
        ...
    }
    public double boo(int n, double d){
        ...
    }
    public void ohyes(String s){
        // metodo non presente nell'interfaccia
    }
}
```



A differenza dell'ereditarietà, dove Java impone il vincolo di avere un'unica classe base (ereditarietà singola), non esiste invece un limite sul numero di interfacce che è possibile implementare in una classe:

```
public class Pluto implements Tizio, Caio,
Sempronio, Eusebio, Baldassarre {
    ..
}
```

Lambda expression

Le espressioni *lambda* sono una caratteristica dei linguaggi con paradigma funzionale, come *Scheme*, ma sono disponibili anche in linguaggi come *Python*, *Dart*, *JavaScript*, *Kotlin* e *Java*.
Si tratta sostanzialmente di un modo per definire delle procedure (in Java diremmo metodi..).

Un'espressione *lambda* in Java consente di definire un metodo specificandone parametri e istruzioni, senza però dargli un nome.

Le espressioni *lambda* vengono utilizzate in Java per implementare i metodi delle interfacce funzionali (interfacce che contengono la firma di un unico metodo al loro interno).
L'espressione *lambda* è formata nei parametri del metodo (a sinistra della freccia) e dal codice del metodo (a destra della freccia)
Ecco i tre tipi di sintassi di base:

```
single_parameter -> single_expression
(parameter1, parameter2) -> single_expression
(parameter1, parameter2) -> { code block }
```

L'esempio seguente, ispirato da un'applicazione Swing, mostra come si può utilizzare un'espressione *lambda* in sostituzione dell'implementazione dell'interfaccia `ActionListener`.
Normalmente per assegnare il listener al `JButton` si utilizzerebbe un codice di questo tipo:

```
mybutton.addActionListener(new ActionListener() {
    @Override
    public void actionPerformed(ActionEvent event) {
    }
});
```

In alternativa, utilizzando le espressioni *lambda*, si può passare ad `addActionListener` direttamente il metodo per gestire l'evento, invece di creare un oggetto anonimo con all'interno quel metodo:

```
mybutton.addActionListener((ActionEvent event) -> {
});
```

Quesiti (rispondi verificando con il codice..)

1. E' possibile creare una classe astratta senza metodi abstract al suo interno?
2. E' possibile definire un metodo abstract in una classe standard (non abstract)?
3. E' possibile in casi particolari definire un metodo abstract con un body?

4. E’ possibile in casi particolari non ridefinire un metodo abstract della classe base nella classe derivata?