Lezione 4

JAVA

Basi e Fondamenti

R-DeV
Revolution Development
learning@r-dev.it

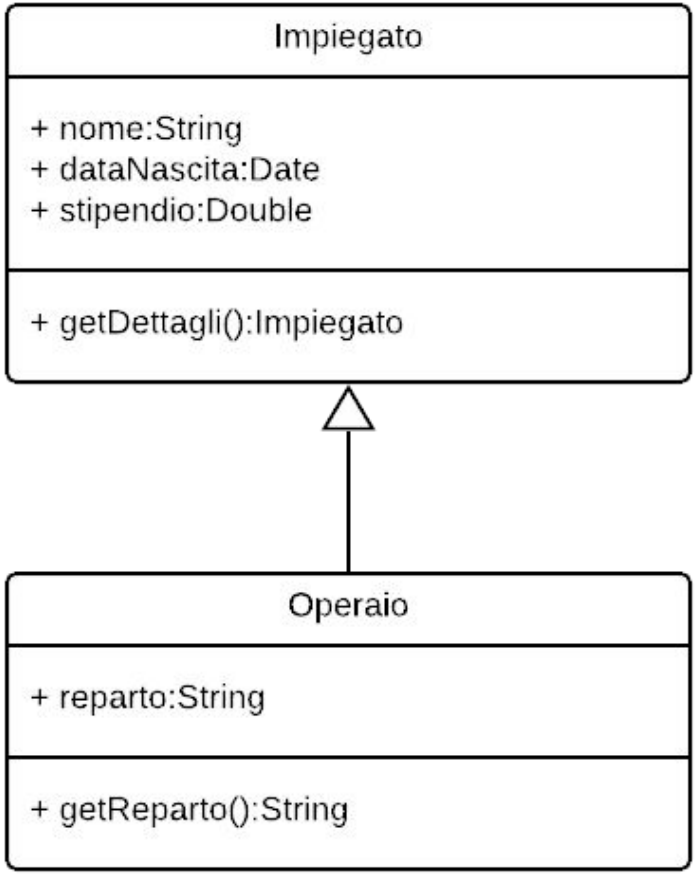
Argomenti della lezione 4

- Ereditarietà,
- Overloading e overriding
- Classi astratte e interfacce
- Optional
- Generics

Ereditarietà

Permette ad una classe di avere, come se fossero suoi, i membri dati e i membri funzione ereditabili da un’altra classe. Da questo segue che possiamo definire tipi e sottotipi.

L'ereditarietà ci permette di vedere le relazioni di parentela tra classi che ereditano dalla stessa superclasse, come un albero radicato.



- Una sottoclasse eredita tutti i metodi e le variabili della superclasse.
- Una sottoclasse NON eredita i costruttori della superclasse.

```
public class Impiegato {
    public String nome;
    public Date dataNascita;
    public Double stipendio;

    public String getDettagli(){
        ...istruzioni;
    }
}


public class Operaio extends Impiegato{
    public String reparto;

    public String getReparto(){
        return this.reparto();
    }
}
```

Polimorfismo

Proprietà per cui un oggetto può assumere la forma di un oggetto più generico che lo descrive (per questo poliforme). Questo concetto è legato all'ereditarietà, e permette la creazione di un oggetto A, istanziandolo da una sua sottoclasse B (cioè che eredita).

```
Impiegato topolino = new Impiegato();

Impiegato pippo = new Operaio();
```

NB: se dichiaro la variabile `pippo` come `Impiegato`, anche se lo instanzio come `Operaio`, non potrò mai avere accesso al metodo `getReparto()`, definito solo nella classe `Operaio`.

Argomenti polimorfici

Si possono costruire metodi che accettano come parametro un riferimento “generico” ed operare in modo automatico su un più vasto insieme di oggetti.
In questo caso è il metodo ad ignorare la reale natura (forma) dell’oggetto che gli viene specificato come parametro attuale.
Per esempio, poiché un `Operaio` è un `Impiegato`:

```
public double getIrpef( Impiegato i ) {
...
}
```

è legale invocare il metodo passando in input:

```
Impiegato o = new Operaio();
getIrpef(o);
```

L'operatore *instanceof*

Poiché è lecito scambiarsi oggetti usando riferimenti ai loro antenati (nella gerarchia ad albero), potrebbe essere a volte necessario sapere esattamente la forma dell'oggetto con cui si ha a che fare.

Se si riceve un oggetto usando un riferimento ad Impiegato, esso potrebbe essere anche realmente un Quadro o un Segretario. Per saperlo basta utilizzare l'operatore `instanceof`

```
public void faQualcosa ( Impiegato i ) {
     if ( i instanceof Operaio ) {
          // elabora un Operaio
     } else {
          // elabora un Impiegato qualunque
     }
}
```

Casting tra oggetti

Nell'eventualità che si riceva un riferimento ad un oggetto e che (usando **instanceof**) si sappia che l'oggetto è realmente di una sottoclasse, si dovrebbe comunque usare quell'oggetto come se fosse della superclasse.
La formalizzazione di **polimorfismo** rende invisibili tutte le specializzazioni proprie di quell'oggetto. Per rendere visibili tutte le caratteristiche dell'oggetto occorre fare un "**cast**" esplicito:

```
public void faQualcosa ( Impiegato i ) {
     if ( i instanceof Operaio ) {
          Operaio o = ( Operaio ) i ;
          System.out.println("Questo è l'operaio del reparto " +
          o.getReparto ());
     }
     …
}
```

Se non ci fosse il cast, il metodo *getReparto* sarebbe stato inaccessibile al compilatore.

Overloading

Overloading di metodi

Metodi con lo stesso nome, nella stessa classe, che svolgono lo stesso compito con diversi argomenti.

Per esempio:

```
public void println( int i )
public void println( float f )
public void println( String s )
```

La lista degli argomenti **DEVE** essere diversa, perché essa sola è usata per distinguere i metodi.

Il tipo di ritorno **PUÒ** essere diverso ma non è usato per distinguere i metodi (cioè, metodi con uguale nome e lista di parametri, ma diverso tipo di ritorno sono indistinguibili ed il compilatore segnale l'errore).

Overloading

Overloading di costruttori

Stesse regole dei metodi.
Il riferimento **this** può essere usato, NELLA PRIMA LINEA di un costruttore, per invocare un altro costruttore.

```
public class Impiegato {
     private String nome;
     private double stipendio;
     private Date dataNascita;

     public Impiegato ( String nome , double stip , Date nasc ) {
          this.nome = nome; this.stipendio = stip; this.dataNascita = nasc ;
     }
     public Impiegato ( String nome , double stip ) {
          this( nome , stip , null );
     }

}
```

Overriding

Overriding di metodi

Una sottoclasse può sovrascrivere un metodo ereditato e visibile, a patto che, rispetto al metodo della superclasse, il nuovo metodo abbia:

- lo stesso nome;
- lo stesso tipo di ritorno oppure una sua sottoclasse;
- la stessa lista di argomenti;
- visibilità non inferiore.

Overriding

ATTENZIONE! L'overriding viene applicato solo sui metodi non statici delle classi!
Attributi e metodi statici e non statici non vengono sovrascritti ma solo "nascosti" nello scope della classe derivata!

```
class A {
        public int idx = 10;
        public int getIdx() { return idx; }
}
class B extends A {
        public int idx = 100;
        public int getIdx() { return idx; }
}
…
public static void main(String[] args) {
        A a = new B();
        System.out.println(a.idx + " " + a.getIdx());  // Cosa stamperà questa istruzione?
        B b = (B) a;
        System.out.println(b.idx + " " + b.getIdx());  // Ed in questo caso?
}
```

Overriding

Le stesse considerazioni fatte per gli attributi possono essere estese anche ad attributi e metodi statici delle classi, anche in questo caso la classe derivata non sovrascriverà attributi e metodi ma si limiterà a nasconderli.

keyword final

- Significa letteralmente *finale*, *non mutabile*
- Può essere utilizzato per definire delle **costanti**
- **Final** può essere applicato anche ai **metodi** di una determinata classe e un metodo definito come `final` implica che se eredito la classe che contiene il metodo su questo non potrà essere eseguito l'override
- Anche una classe può essere definita `final`, ma di conseguenza non potrà essere estesa

keyword super

- `super` è usata in una classe per riferirsi alla superclasse.
- `super` è usata per riferirsi ai membri della superclasse (attributi e metodi) con la dot notation.
- I membri della superclasse accessibili con `super` sono anche quelli che la superclasse ha ereditato (quindi ad esempio i metodi di Object) e non solo quelli esplicitamente definiti in essa.
- NON si può usare `super.super.membro` per accedere a membri di classi di livello superiore alla prima superclasse.

equals e ==

- L'operatore == determina se due riferimenti sono identici (cioè se riferiscono allo stesso, unico, oggetto).
- Il metodo equals determina se le due variabili si riferiscono ad oggetti (anche distinti) appartenenti alla stessa classe di equivalenza, rispetto ad una particolare relazione di equivalenza definita dall'utente.
- La realizzazione di equals nella classe Object fa uso di ==. Quindi le classi di equivalenza di == e di equals coincidono in Object (relazione di identità).
- Lo scopo del metodo è quello di essere sovrapposto nelle classi sviluppate dagli utenti, in modo da realizzare nuove relazioni di equivalenza.
- Alcune classi di sistema lo fanno già. Per esempio la classe String realizza il proprio equals, confrontando le stringhe carattere per carattere.
- Viene raccomandato di sovrapporre, insieme con equals, anche il metodo hashCode.

keyword abstract

- Abstract può essere applicato sia a **metodi** che alle **classi**, ma non a variabili
- Inserire il modificatore `abstract` ad un metodo significa definirne una firma dello stesso
- Un metodo abstract non ha corpo
- Se all'interno di una classe dichiariamo un metodo astratto, anche la classe **dovrà** essere definita come tale.
- Una classe, se definita astratta, **non può essere in alcun modo istanziata**
- I metodi abstract che **non possono essere private**, altrimenti si genera un errore a *compile time*.

Classi astratte

Le classi astratte sono classi che non possono essere istanziate direttamente. Esse hanno generalmente, ma non esclusivamente, dei metodi che sono anch'essi definiti astratti e che sono privi di corpo di definizione.

Una classe astratta può comunque essere derivata, e le classi che la estendono devo implementare gli eventuali metodi astratti, sovrascrivendoli con la propria logica.

```
… abstract class identifier … {
        abstract return_type identifier( … ) [throw ..];
}
```

Classi astratte

Le classi astratte sono classi che non possono essere istanziate direttamente. Esse hanno generalmente, ma non esclusivamente, dei metodi che sono anch'essi definiti astratti e che sono privi di corpo di definizione.

Una classe astratta può comunque essere derivata, e le classi che la estendono devo implementare gli eventuali metodi astratti, sovrascrivendoli con la propria logica.

```
… abstract class identifier … {
        abstract return_type identifier( … ) [throw ..];
}
```

Interfacce

Una “interfaccia” è un **contratto** tra il codice cliente e la classe che “implementa” quell’interfaccia.

In Java viene considerata come la formalizzazione di un contratto, dove sono scritte le regole da rispettare per le classe che la implementano, ovvero variabili e firme dei metodi (**senza corpo**)

Un’interfaccia, generalmente, modella un determinato “comportamento” che una classe può scegliere di supportare. Nel caso decida di farlo si impegna ad onorarne i vincoli implementando i membri funzione dichiarati nell’interfaccia.

Molte classi possono implementare la stessa interfaccia

Una classe può implementare molte interfacce

Una interfaccia può estendere altre interfacce

Interfacce

un’interfaccia è una sorta di “classe astratta” che dichiara, principalmente dei metodi che le classi che la implementano devono poi definire.

Contiene principalmente una seria di **metodi** astratti che sono implicitamente **public** ed **abstract**.

Può contenere dei **campi** che implicitamente sono **public, static** e **final**

Può contenere dichiarazioni di **classi** ed **interfacce** annidate che sono implicitamente **public** e **static**

Dalla versione 8 di Java possono contenere anche dei **metodi con un’implementazione** (vanno dichiarati con la keyword default) che sono implicitamente **public.** Questi metodi possono essere anche statici

Dalla versione 9 di Java è possibile dichiarare dei metodi private all’interno di un’interfaccia a patto che non si tratti di un metodo di default

Interfacce

implementazione di un’interfaccia in una classe

```
class MiaClasse implements MiaInterfaccia {
        … definizione metodi ...
}
```

è possibile creare un’interfaccia che ne estenda un’altra

```
interface MiaInterfaccia extends InterfacciaPadre {
        ...
}
```

Classi anonime

Una classe anonima è una classe interna che non è un membro di un'altra classe; essa è dunque una classe locale ad un blocco di codice che però non ha un nome.

Una classe anonima può essere creata con la seguente sintassi:

```
new MiaClasse() { … }          // Creazione istanza a partire da una classe
new MiaInterfaccia() { … }     // Creazione istanza a partire da un'interfaccia
```

La creazione, quindi, prevede l'utilizzo dell'operatore new seguito dal nome della classe da estendere o interfaccia da implementare. Nella definizione del corpo è possibile inserire nuovi campi o metodi ma anche effettuare l'override di metodi esistenti.

Classi anonime

Quando si creano classi anonime bisogna tenere presente che:

- il nome della superclasse utilizzato dopo la keyword new è a tutti gli effetti l'invocazione del costruttore di quella classe quindi nulla vieta l'utilizzo di eventuali costruttori parametrici presenti;
- Nel corpo della classe non possono essere inseriti dei costruttori (del resto si tratta di una classe anonima!);
- L'operatore new crea un'istanza di una classe il cui riferimento viene utilizzato nel contesto di valutazione dell'espressione nella quale si trova tale creazione;
- Le classi anonime non possono mai essere static, abstract e final (fino alla versione 8 del linguaggio erano invece implicitamente final).

L'identificatore var può essere utilizzato per dichiarare una variabile inizializzata con un'espressione di creazione di una classe anonima. In questo caso il compilatore sarà in grado di inferire il tipo dell'effettiva classe anonima creata.

La classe Optional<T>

La classe java.util.Optional è stata introdotta dalla versione 8 di Java con lo scopo di gestire con maggiore facilità e robustezza il valore di un oggetto.

E' un oggetto contenitore che PUO' contenere un valore nullo:

- `isPresent()` restituisce true se l'oggetto è valorizzato
- `get()` restituisce l'oggetto se presente altrimenti lancia `NoSuchElementExcetption`

L'inserimento della classe Optional nel linguaggio Java è nata dall'esigenza di rendere più robusto il codice, spesso soggetto ad eccezioni di tipo NullPointer. Ad esempio, un metodo potrebbe ritornare un reference nullo e causare problemi, invece utilizzando Optional ci costringiamo a gestire a livello di compilazione la situazione potenzialmente pericolosa.

Generics

Ci è già capitato di vedere la notazione <T> : in Java simboleggia un tipo reference generico, o meglio non specificato in fase di scrittura del codice.

Con questa notazione, introdotta dalla versione 1.5 di Java, potremmo scrivere un metodo generico per ordinare un array di oggetti, quindi invocare il metodo generico con array di Integer, di Double, di String e così via, per ordinare gli elementi dell'array.

Si tratta della possibilità di dotare classi, interfacce e metodi di parametri di tipo

Simili ai normali parametri dei metodi, questi parametri hanno come possibili valori i tipi (non primitivi) del linguaggio

Questo meccanismo consente di scrivere codice più robusto dal punto di vista dei tipi di dati, evitando in molti casi il ricorso alle conversioni forzate (cast)

Questa funzionalità è una forma di polimorfismo parametrico

Generics - classi

Esempio di classe parametrica:

```
public class Contenitore <T>{
    private T contenuto;
    public T getContenuto(){ return this.contenuto; }
    public void setContenuto(T contenuto) {
        this.contenuto = contenuto;
    }
}
```

La classe Contenitore ha un parametro di tipo T (per convenzione si usa sempre la T ma è possibile usare qualsiasi lettera o stringa)
Se una classe è parametrica va indicato dopo il nome della classe inserendo il parametro tra parentesi angolate, se ci sono più parametri vanno separati da virgole

Generics - invocazione

Nella dichiarazione di una variabile parametrica è necessario specificare il tipo:

```
Contenitore<String> c = new Contenitore<String>();
```

In questo caso il tipo di `T` sarà `String`

Nell'instanziazione dell'oggetto non è necessario specificare il tipo (*diamond operator*)

```
Contenitore<String> c = new Contenitore<>();
```

Anche i singoli metodi e costruttori possono avere parametri di tipo, indipendentemente dal fatto che la classe cui appartengono sia parametrica o meno

I metodi statici non possono utilizzare i parametri di tipo della classe in cui sono contenuti

Generics - metodi

Il seguente metodo generico restituisce il primo elemento di un array, di qualunque tipo esso sia:

```
public static <T> T getFirst(T[] a) {
    return a[0];
}
```

Se un metodo è parametrico va indicato nella firma prima del tipo di ritorno, inserendo il parametro (o la lista di parametri separata da virgole) tra parentesi angolate

Questo parametro è visibile solo all'interno del metodo

Generics - metodi

Un metodo è detto generico quando accetta diversi tipi di dato su cui esegue uno stesso algoritmo. Se non vi fosse la possibilità di scrivere il metodo in forma generica si dovrebbe ricorrere a
Il seguente metodo generico restituisce il primo elemento di un array, di qualunque tipo esso sia:

```
public static <T> T getFirst(T[] a) {
    return a[0];
}
```

Se un metodo è parametrico va indicato nella firma prima del tipo di ritorno, inserendo il parametro (o la lista di parametri separata da virgole) tra parentesi angolate

Questo parametro è visibile solo all'interno del metodo