



Alma Mater Studiorum-Università di Bologna Scuola di Ingegneria

Dalle classi alle interfacce

Corso di Laurea in Ingegneria Informatica
Anno accademico 2021/2022

Prof. ENRICO DENTI

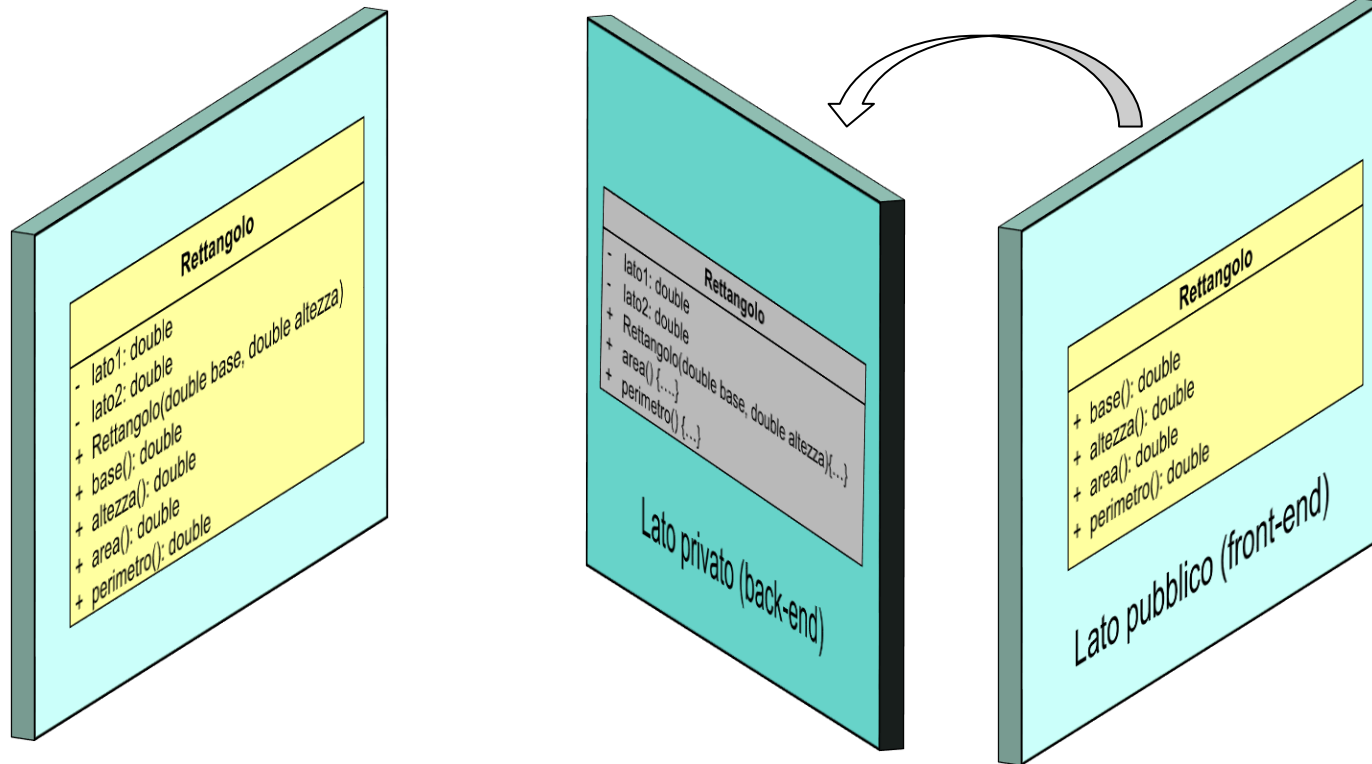
Dipartimento di Informatica – Scienza e Ingegneria (DISI)



CLASSI: FRONT-END e BACK-END

- Una classe fornisce la **definizione** di un ADT
 - parte visibile esternamente (signature dei metodi pubblici)
→ **front-end (pubblico)**
 - implementazione (corpo dei metodi e dati privati)
→ **back-end (privato)**
- Quindi, definire una classe (non astratta) implica specificare *contemporaneamente* tutti i vari aspetti
 - non solo QUALI operazioni ci debbano essere (front-end)
 - ma anche COME esse debbano essere FATTE (back-end)
- Come ora vedremo, ***questa contemporaneità è un limite*** che occorre superare per acquisire gradi di libertà di progetto.

CLASSI: FRONT-END e BACK-END



La classe

- espone le signature dei metodi pubblici
- ma specifica anche, **nello stesso costruito**, la struttura interna (corpo dei metodi e dati privati)

Quindi

- La classe ha sì un *front-end pubblico* e un *back-end privato*
- MA il punto è che li deve specificare **contemporaneamente**



CLASSI: LIMITI

- Perché questa contemporaneità nella specifica di front-end e back-end è un limite?
- Tre questioni:
 - talora *non si è in grado di dare tali dettagli* in fasi preliminari
 - più spesso *non si vuole deciderli / vincolarli a priori*
 - inoltre, *manca* il supporto *all'ereditarietà multipla*



LA PRIMA QUESTIONE

- Per evitare di dover specificare tutti i dettagli di funzionamento, si potrebbe definire una classe astratta
 - permette proprio di dire QUALI operazioni ci sono
 - SENZA doverle implementare subito
- Però, *ciò non è del tutto soddisfacente*
 - *introduce un vincolo*: le operazioni lasciate in bianco possono essere implementate *solo in una sottoclasse*
 - MA ciò potrebbe essere *inopportuno o impossibile*, se la classe deve logicamente stare altrove nella gerarchia!
 - È una soluzione *troppo legata all'ereditarietà* (che peraltro in Java e derivati, fra classi, è intenzionalmente solo singola)

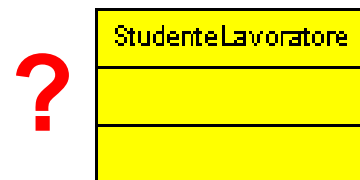
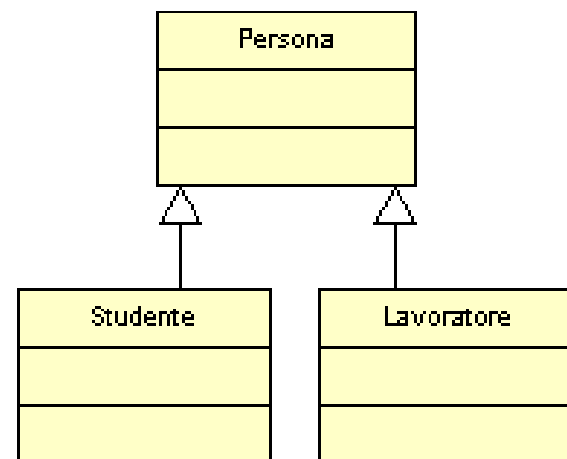


LA SECONDA QUESTIONE

- Più spesso, **non si vuole dover stabilire a priori chi debba fornire certi servizi**
 - perché può essere troppo presto per fare una scelta oculata
 - perché magari la scelta migliore si può fare solo a runtime
 - *magari in base alla situazione* (FACTORY.. 😊)
 - per non perdere di generalità quando non è necessario
 - per non introdurre vincoli inutili
- Le classi astratte non costituiscono una soluzione soddisfacente a questo riguardo, perché vincolano a implementare *in una sottoclasse* i metodi lasciati astratti
 - di nuovo, le classi astratte appaiono una soluzione molto legata all'ereditarietà, che oltre tutto in Java e derivati è solo singola

LA TERZA QUESTIONE

- L'ereditarietà singola **non** permette di esprimere tutte le situazioni che si possono avere della realtà
 - se **Studiante** e **Lavoratore** estendono **Persona**, come si modella **StudianteLavoratore**?
 - idem per le forme geometriche (**Rettangolo**, **Quadrato**,...)
- ma al contempo **l'ereditarietà multipla fra classi genera solo caos**
 - l'esperienza ha dimostrato che unire *implementazioni* è dannoso e controproducente.





TRE QUESTIONI, UNA RISPOSTA

1. SEPARARE

- il momento in cui si specifica la *vista esterna* (front-end)
- dal momento in cui si dettaglia la *realizzazione interna* (back-end)

MA senza dipendere troppo dall'ereditarietà (mentre le classi astratte..)

2. SUPERARE l'ereditarietà singola

- sì all'ereditarietà multipla *come strumento concettuale*
- ma *non* per mixare implementazioni (con relativi guai)

Come farlo?

- superando i *limiti intrinseci* del costrutto `class`
- *separando fisicamente* interfaccia e implementazione

Java

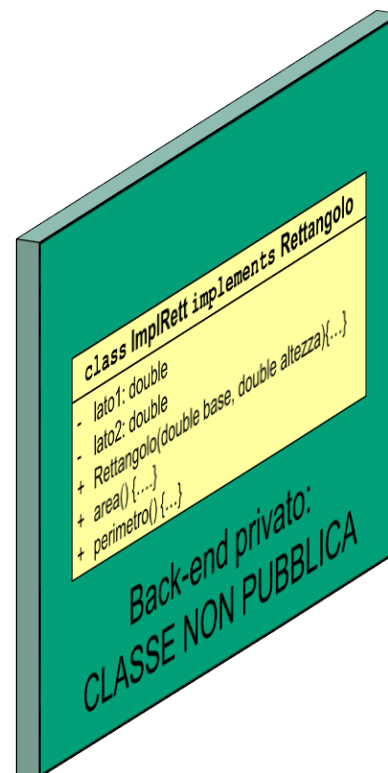
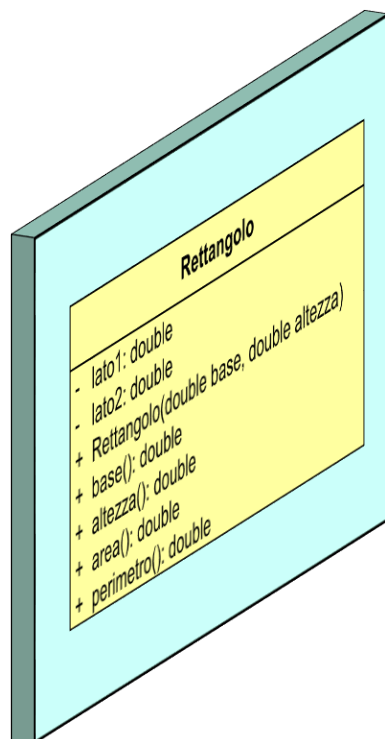
C#

Kotlin

Nuovo costrutto: **interface**

Scala: *trait*

SEPARARE FRONT-END e BACK-END



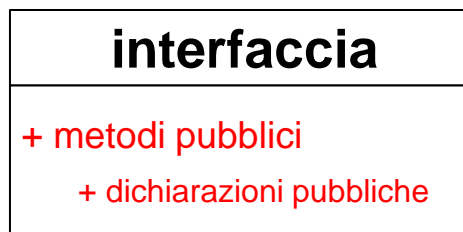
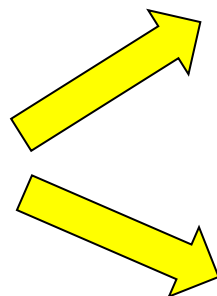
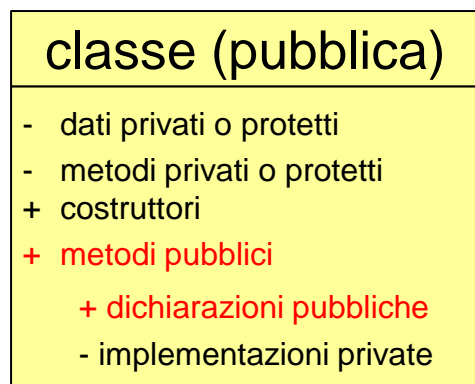
FINORA: UN SOLO COSTRUTTO

- Tutto è specificato nella *classe pubblica*
- Si specificano nello stesso costrutto e nello stesso momento sia il front-end (pubblico) sia il back-end (privato)

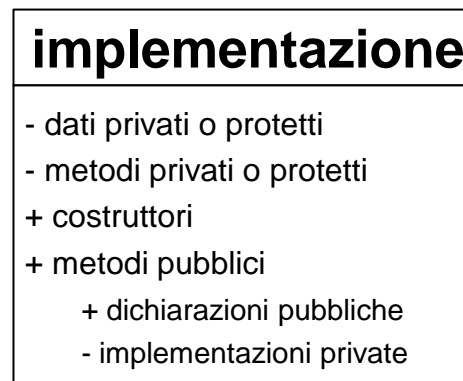
IDEA: USARE DUE COSTRUTTI DISTINTI

- Front-end pubblico: *interfaccia*
- Back-end privato: *classe* (non più pubblica)
- Le due parti sono specificate in costrutti diversi e non più contemporaneamente

LA SEPARAZIONE



public
interface
(Scala: *trait*)



class that
implements
the interface



LA SEPARAZIONE

classe pubblica
(ma con parti non pubbliche)

```
public class Rettangolo{ Java C# ~Kotlin ~Scala  
    private double lato1, lato2;  
    public Rettangolo(double altezza, double base) {  
        lato1 = base; lato2 = altezza; }  
    public double altezza() { return lato2; }  
    public double base() { return lato1; }  
}
```

NB: può contenere anche altri metodi (pubblici e non), oltre a quelli definiti dall'interfaccia

```
public interface Rettangolo {  
    public double altezza();  
    public double base();  
} Java C# ~Kotlin Scala: trait
```

100% pubblica

non necessariamente pubblica

```
class ImplRettangolo implements Rettangolo {  
    private double lato1, lato2;  
    public ImplRettangolo(double altezza,  
        double base) { lato1 = base; lato2 = altezza; }  
    public double altezza() { return lato2; }  
    public double base() { return lato1; }  
} Java ~C# ~Kotlin ~Scala
```



LA NOZIONE DI INTERFACCIA

- Un'interfaccia è una *classe astratta portata all'estremo*
 - così astratta da non contenere più implementazioni
 - ma *solo dichiarazioni di metodi* (ed eventualmente costanti)
 - e nessun costruttore (perché non c'è nessun dato da inizializzare)
- Come una classe, *introduce un tipo*
 - usabile per definire riferimenti e argomenti di funzioni
- Ma, a differenza di una classe, *separa nettamente*
 - il momento (*attuale*) in cui si specifica la *vista esterna*
ossia in cui si specifica l'interfaccia
 - il momento (*futuro*) in cui si specifica la *parte interna*
ossia in cui si dettaglia la classe che la implementa
- Una *soluzione pulita* alla questione "ereditarietà multipla".



CLASSI ASTRATTE vs. INTERFACCE

CLASSI ASTRATTE

- Il nome della classe deve *coincidere* col nome del file `.java`
- Le classi (astratte e non) possono definire dati, costruttori e implementazioni di metodi.
- Le classi (astratte e non) *possono* essere private o pubbliche
- Le classi (astratte e non) possono avere membri privati o pubblici
- L'implementazione dei metodi "lasciati in bianco" dev'essere fornita *da una sottoclasse*
→ relazione di *Ereditarietà*

INTERFACCE

- Il nome *dell'interfaccia* deve *coincidere* col nome del file `.java`
- Le interfacce non definiscono dati, non hanno costruttori né implementazioni di metodi (*tranne eventualmente dei default..*)
- Le interfacce sono sempre e solo pubbliche
- Le interfacce possono dichiarare solo costanti e metodi pubblici.
- L'implementazione dei metodi "lasciati in bianco" può essere fornita *da una classe qualsiasi purché si impegni a realizzare/implementare l'interfaccia*
→ relazione di *Realizzazione*



CONCETTO & COSTRUTTO

- Un' INTERFACCIA **dichiara** metodi (e costanti)
ma, a differenza di una classe,
- ***non li implementa affatto***
 - esprime una pura specifica di comportamento
 - permette di dire *cosa si vuole* senza dover dire adesso *come* farlo

Talora sono tuttavia consentite implementazioni «di default», per comodità

ESEMPIO

```
public interface Rettangolo {  
    public double base();  
    public double altezza();  
}
```

Java

C#

~Kotlin

Scala: *trait*

*Specifica un'astrazione: definisce **Rettangolo** come un'entità dotata delle due proprietà **base** e **altezza**, indipendentemente da *chi*, *quando* e *come* la realizzerà.*



INTERFACCE COME SPECIFICA

Un'interfaccia è un potente strumento di modellazione

- permette di definire entità in termini di *vista esterna*, ossia di **comportamento osservabile** – *cosa ci si aspetta che sappia fare*
- **non anticipa scelte realizzative** – come si faranno quelle cose – che verranno decise *da altri* in *tempi successivi*.

Naturalmente, prima o poi si dovrà riempire il "guscio vuoto":
apposite classi implementeranno le interfacce

- nasce la **relazione di REALIZZAZIONE** (concretizzazione):
una classe conterrà il codice che *implementa* (realizza)
ciò che l'interfaccia ha solo *dichiarato* (promesso)

C# :

Kotlin :

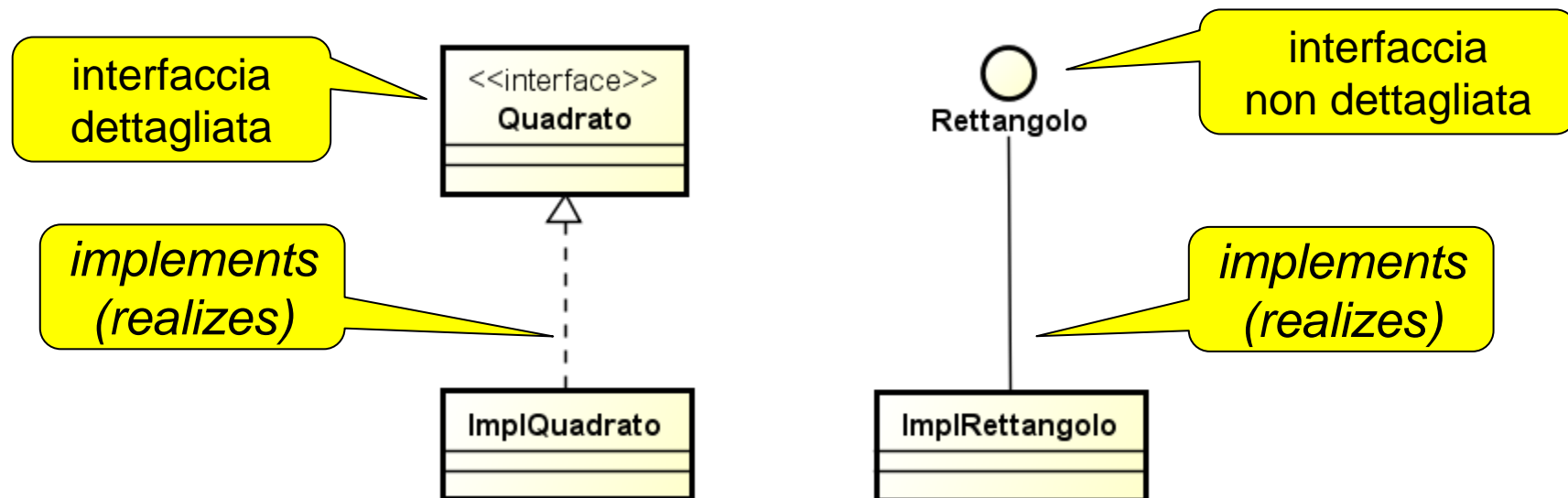
Scala: **extends**

keyword **implements**

Java

UML: RELAZIONE DI REALIZZAZIONE

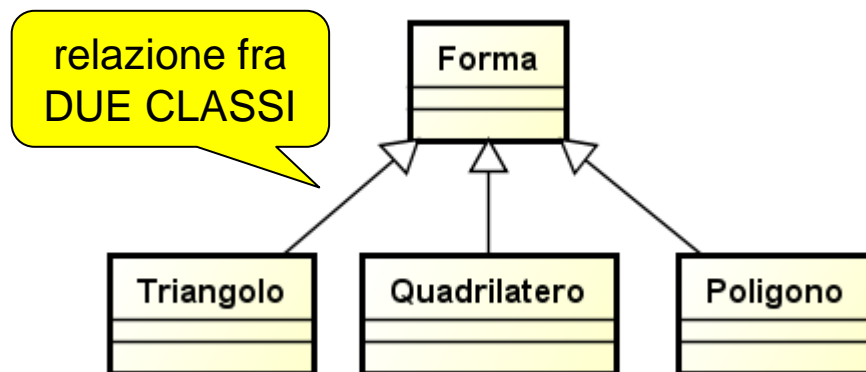
La **relazione di realizzazione** (o **implementazione**) fra classi e interfacce espressa in UML con due possibili notazioni:



EREDITARIETÀ vs. REALIZZAZIONE

Notare la differenza:

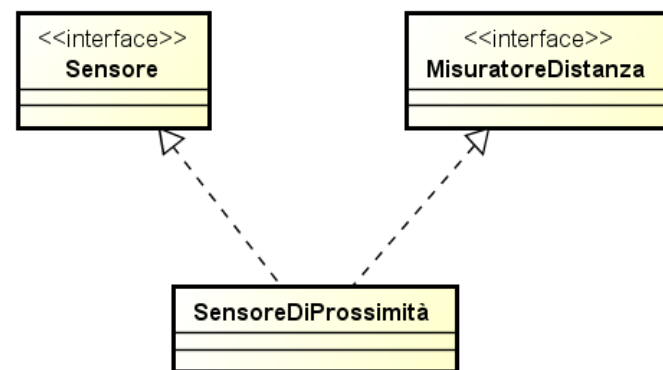
- l'ereditarietà via finora è una relazione *fra classi*
 - una classe estende un'altra classe
- la **realizzazione** è una relazione *fra classe e interfaccia*
 - associa una classe con l'interfaccia che dichiara le funzionalità fornite (implementate) dalla classe stessa



IMPLEMENTAZIONE DI INTERFACCE

Una classe può implementare anche *più interfacce*

- si intende allora che essa implementa *tutte* le funzionalità previste da *tutte* le interfacce implementate
 - *non può mancare neppure un metodo*, altrimenti resterebbe un vulnus
- ESEMPIO
 - nel caso in figura, la classe **SensoreDiProssimità** implementa
 - tutti i metodi dichiarati in **MisuratoreDistanza**
 - più tutti quelli dichiarati in **Sensore**
 - pertanto, ogni **SensoreDiProssimità** è in grado di fungere da (è intercambiabile con)
 - sia **MisuratoreDistanza**
 - sia **Sensore**





EREDITARIETÀ FRA INTERFACCE

- Le interfacce possono anche *ereditare* le une dalle altre
 - come fra classi, una interfaccia può *estenderne* un'altra
 - solita grafica UML («freccia con la punta bianca»)
 - in Java e Scala si usa la solita keyword **extends**, in C# e Kotlin il :
 - ciò determina anche qui l'instaurarsi della relazione tipo/sottotipo, con semantica di inclusione insiemistica
- Ma.. *cosa significa "ereditare" fra interfacce?*
 - essendo gusci vuoti, non significa ereditare *codice*
 - *significa stabilire relazioni tipo/sottotipo fra astrazioni*
 - diventa possibile definire astrazioni e relazioni fra astrazioni senza dover anticipare allo stadio di progetto scelte realizzative
 - notevole capacità espressiva: vero "uovo di Colombo" del design!



INTERFACCE & ERED. MULTIPLA

Non implementando nulla, *l'interfaccia elimina alla radice il rischio di collisione fra metodi / dati omonimi*

- per questo, supporta l'ereditarietà multipla senza problemi

```
public interface Rettangolo {  
    ...  
    public double base();  
    public double altezza();  
}
```

Java

C#

~Kotlin

Scala: *trait*

```
public interface Rombo {  
    ...  
    ...  
    public double lato();  
}
```

Java

C#

~Kotlin

Scala: *trait*

Scala: *trait*

```
public interface Quadrato extends Rettangolo, Rombo {
```

```
    ...  
}
```

Esprime l'idea che i **Quadrato** abbiano *tutte le proprietà e i servizi dei **Rettangolo** più tutte quelle dei **Rombo**.*

C# :

Kotlin :

Scala: *extends*



INTERFACCE & PROGETTO

- L'introduzione della nozione di interfaccia nello spazio concettuale aggiunge un nuovo, potente "attrezzo mentale"
- Le interfacce inducono un *diverso modo di progettare*
 - **PRIMA LE INTERFACCE**, la cui tassonomia riflette le opportune *scelte di progetto (pulizia concettuale)*
 - **SOLO DOPO LE CLASSI**, la cui tassonomia riflette di norma *scelte implementative (efficienza ed efficacia)*
- La relazione classe/interfaccia implica *compatibilità di tipo* fra
 - il **TIPO-INTERFACCIA**
 - i **TIPI-CLASSE** che la implementano.



TIPI (riferimenti a) INTERFACCIA

- L'interfaccia costituisce un *tipo*, di cui però *non si possono creare istanze* perché è solo un "guscio vuoto"
- Le *classi che implementano quell'interfaccia*, tuttavia, ne costituiscono per definizione una *valida realizzazione*
 - è tramite queste che si può "dare sostanza" all'interfaccia
 - a ciò serve la compatibilità di tipo fra il TIPO-INTERFACCIA e i TIPI-CLASSE che la implementano
- Riferimenti di *tipo-interfaccia* possono *referenziare istanze* delle corrispondenti classi:

```
Rettangolo r = new ImplRettangolo();  
Sensore s = new SensoreDiProssimità();
```

Java

C#

~Kotlin

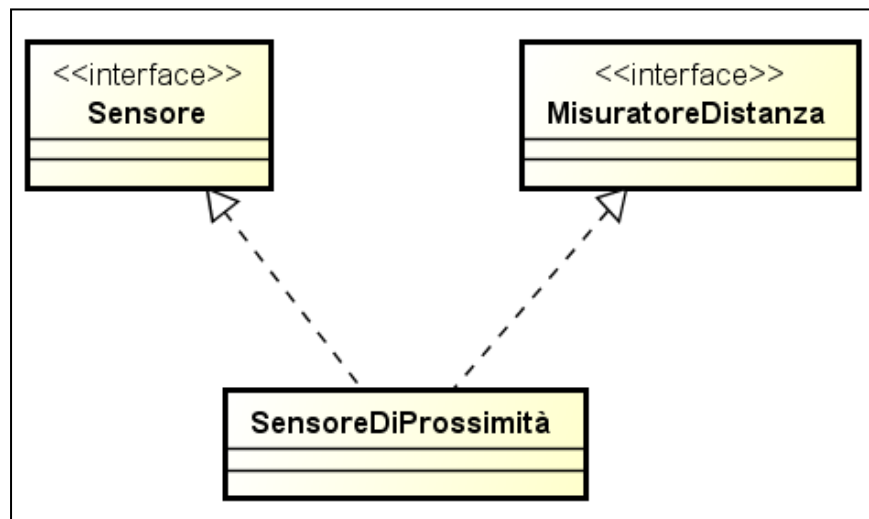
~Scala

Interfaccia

Classe (che la realizza)

COMPATIBILITÀ DI TIPO

- Il tipo-interfaccia è *compatibile in assegnamento con tutti i tipi-classe che la implementano*
 - nel caso dell'esempio, un riferimento a **MisuratoreDistanza** o **Sensore** potrà maneggiare istanze di **SensoreDiProssimità**
 - MA al contempo **MisuratoreDistanza** e **Sensore** rimangono concetti e tipi disgiunti, *incompatibili fra loro* – come deve essere!





RIFERIMENTI A INTERFACCE: PERCHÉ

- Diviene possibile *manipolare entità con certe caratteristiche*
 - espresse dal tipo-interfaccia
- *senza dover stabilire a priori cosa saranno esattamente*
 - ossia, di che classe saranno
 - in particolare, per definire *metodi* che manipolino entità di un certo tipo (interfaccia) *pur senza vincolarne precisamente la classe*
- Risultato: grande disaccoppiamento architetturale
 - notevoli vantaggi ingegneristici
 - pulizia, facilità di manutenzione ed estensione, testing

Vedremo tra poco un piccolo esempio,
seguito da un più completo caso di studio



INTERFACCE vs. CLASSI: CONVENZIONI DI NAMING

- Diversi linguaggi adottano diverse «scuole di pensiero»
- In **Java** **Scala** **Kotlin**
 - l'interfaccia/tratto tipicamente ha un nome «chiaro e corto»
 - la classe che la implementa ha un nome «più implementativo», solitamente analogo ma con un *prefisso identificativo*
 - esempi: interfaccia **Rettangolo**, classe **ImplRettangolo**; interfaccia **Controller**, classe **MyController**
- In **C#**
 - l'interfaccia tipicamente ha un nome della forma **IQualcosa**
 - la classe che la implementa ha un nome **Qualcosa**
 - Esempio: interfaccia **IRettangolo**, classe **Rettangolo**

UN PICCOLO ESEMPIO (1/7)

- Questa interfaccia Rettangolare cattura le proprietà di qualunque cosa «piana e rettangolare»

```
public interface Rettangolare {  
    public double larghezza();  
    public double lunghezza();  
}
```

Java

C#

~Kotlin

Scala: *trait*

- Concretamente, moltissime cose sono rettangolari:

- un tavolo
- un libro
- un foglio A4
- un appezzamento di terreno
- ...





UN PICCOLO ESEMPIO (2/7)

- Sebbene siano diversissime, essendo «rettangolari» *hanno tutte una «larghezza» e una «lunghezza»*
 - quindi se ne può ed esempio calcolare l'area *allo stesso modo*:

```
public class MyMath {  
    public static double area(Rettangolare r) {  
        return r.larghezza() * r.lunghezza();  
    }  
}
```

Java

C#

~Kotlin

~Scala

- ai fini del calcolo dell'area, non importa se si stia lavorando su un libro, un tavolo, o un terreno: conta solo che sia un «rettangolo»!
- questa libreria opera su Rettangoli di ogni specie, *presenti e futuri*

UN PICCOLO ESEMPIO (3/7)

- Ad esempio, il Tavolo potrebbe essere modellato così:

```
public class Tavolo implements Rettangolare {  
    private double largh, lungh, h;  
  
    public Tavolo(double largh, double lungh, double h){  
        this.largh=largh; this.lungh=lungh; this.h=h;}  
  
    @Override  
    public double larghezza() { return largh;}  
  
    @Override  
    public double lunghezza() { return lungh; }  
  
    public double altezza() { return h; }  
    public double peso() {...} // in base al peso del legno...  
    ... // altre cose specifiche del tavolo: colore...  
}
```

Java

C#

~Kotlin

~Scala



UN PICCOLO ESEMPIO (4/7)

- Il Libro, invece, così:

```
public class Libro implements Rettangolare {  
    private double largh, lungh, spessore;  
  
    public Libro(double largh, double lungh, double sp){  
        this.largh=largh; this.lungh=lungh; this.spessore=sp;}  
  
    @Override  
    public double larghezza() { return largh;}  
  
    @Override  
    public double lunghezza() { return lungh; }  
  
    public double spessore() { return spessore; }  
  
    public double peso() {...} // in base al peso della carta..  
    ... // altre cose specifiche del libro: ISBN, editore..  
}
```

Java

C#

~Kotlin

~Scala



UN PICCOLO ESEMPIO (5/7)

- L'Appezamento di terreno, infine, così:

```
public class Appezamento implements Rettangolare {  
    private double largh, lungh;  
    private String gps; // GPS coordinates  
    public Appezamento(double largh, double lungh, String gps) {  
        this.largh=largh; this.lungh=lungh; this.gps=gps;  
    }  
    @Override  
    public double larghezza() { return largh; }  
    @Override  
    public double lunghezza() { return lungh; }  
    public String gpsCoordinates() { return gps; }  
    ... // altre cose specifiche dell'appezamento di terreno  
}
```

Java

C#

~Kotlin

~Scala





UN PICCOLO ESEMPIO (6/7)

- ..e tuttavia, di tutte possiamo calcolare l'area!

```
public static void main (String[] args){  
    Libro libroJava = new Libro(22,16,3);  
    Tavolo tavoloCucina = new Tavolo(120,70,74);  
    Appezzamento terreno = new Appezzamento(  
        21, 50, "44.49179603211135, 11.330909770799916");  
    System.out.println(MyMath.area(libroJava));  
    System.out.println(MyMath.area(tavoloCucina));  
    System.out.println(MyMath.area(terreno));  
}
```

Java

C#

~Kotlin

~Scala

Sono tutti Rettangolari!

– output:

```
352.0  
8400.0  
1050.0
```



UN PICCOLO ESEMPIO (7/7)

- RIASSUMENDO

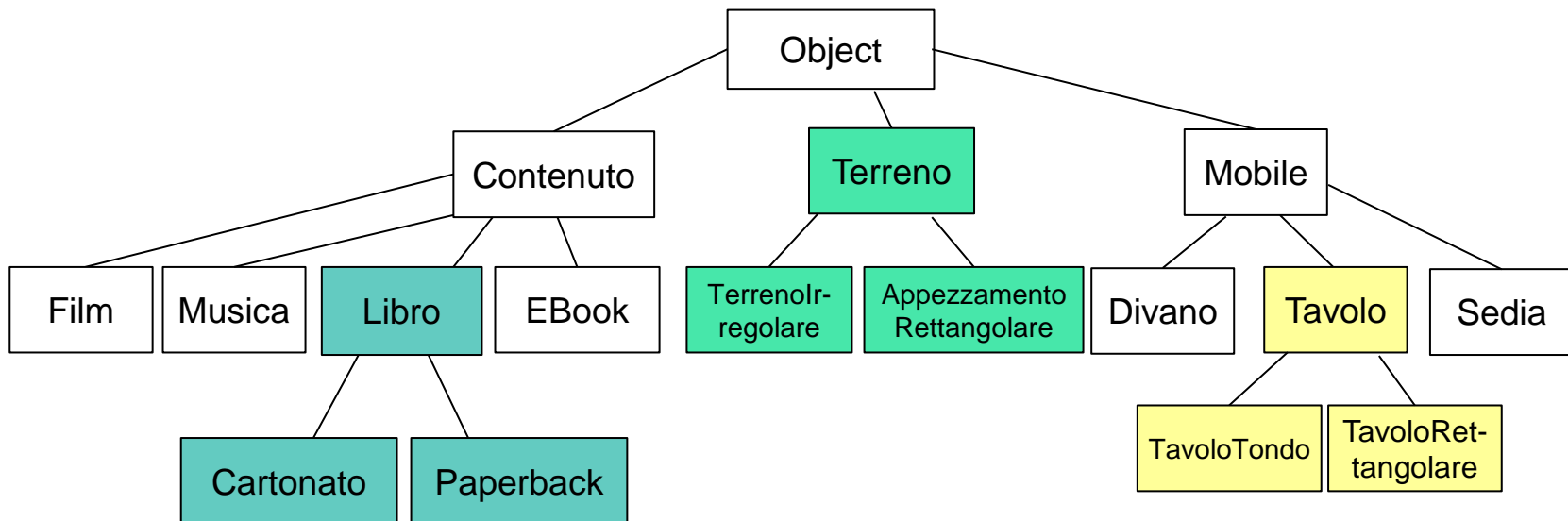
- **Libro**, **Tavolo** e **Appezamento** sono *classi totalmente diverse*, che si collocano in *posizioni totalmente distinte della tassonomia*
- tuttavia, ciò non impedisce all'interfaccia **Rettangolare**, che tutte implementano, di catturare una loro *caratteristica comune*, *trasversale alla tassonomia di ereditarietà*

- RIFERIMENTI A CLASSI vs. A INTERFACCE

- un *riferimento a una classe* spazia solo fra le sue sottoclassi, ossia all'interno di un singolo sotto-albero della tassonomia
 - ad esempio, un riferimento a Tavolo può puntare solo ad altri tavoli più specifici
- un *riferimento a un'interfaccia* spazia invece su tutte le classi che implementano tale interfaccia, ovunque siano – quindi anche su più sottoalberi disgiunti della tassonomia

RIFERIMENTI A INTERFACCE vs. RIFERIMENTI A CLASSI

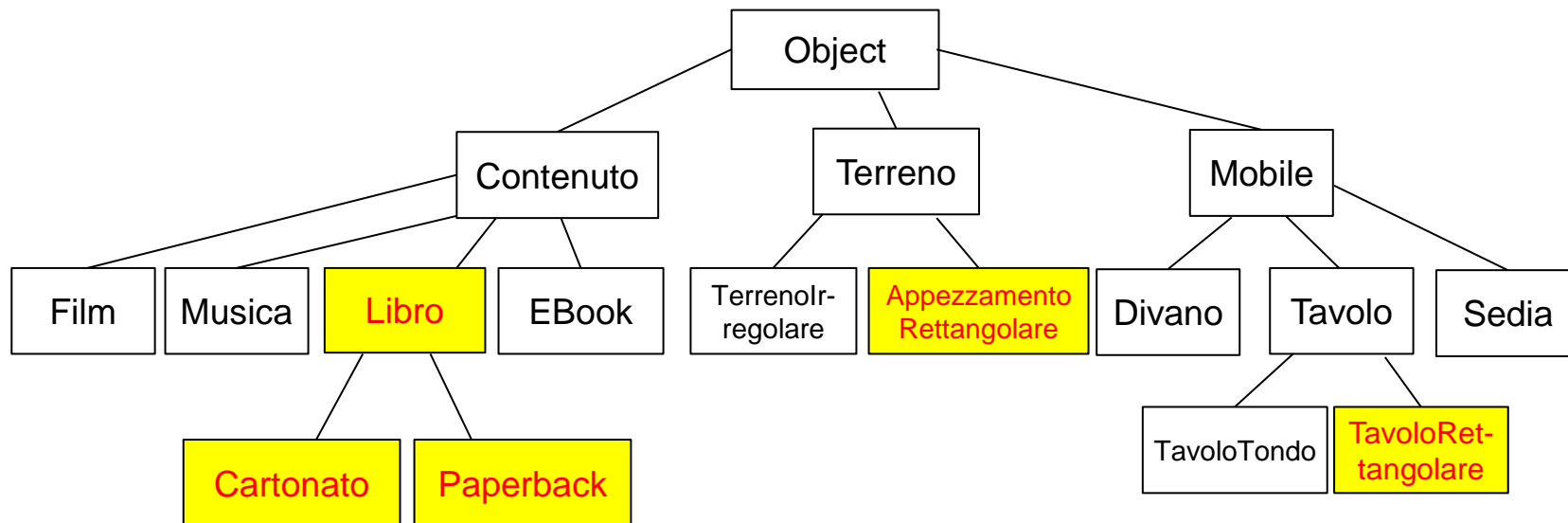
- Generalizzando la tassonomia *di classi* dell'esempio:



- un riferimento a **Libro** può spaziare solo su **Libro**, **Cartonato**, **Paperback**
- un riferimento a **Terreno** può spaziare solo sui vari tipi di terreni
- un riferimento a **Tavolo** può spaziare solo sui vari tipi di tavoli

RIFERIMENTI A INTERFACCE vs. RIFERIMENTI A CLASSI

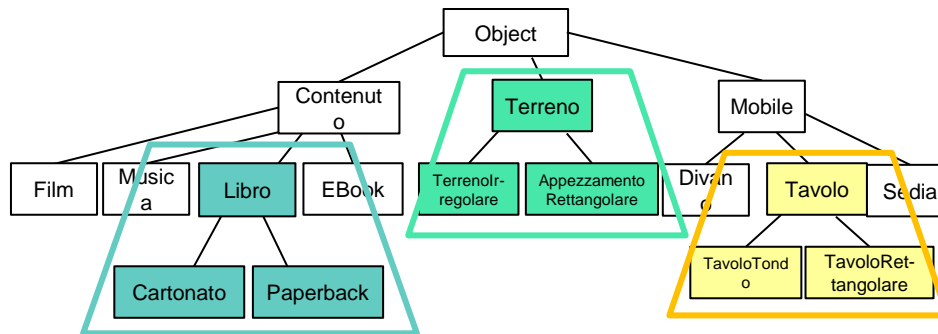
- Generalizzando la tassonomia *di classi* dell'esempio:



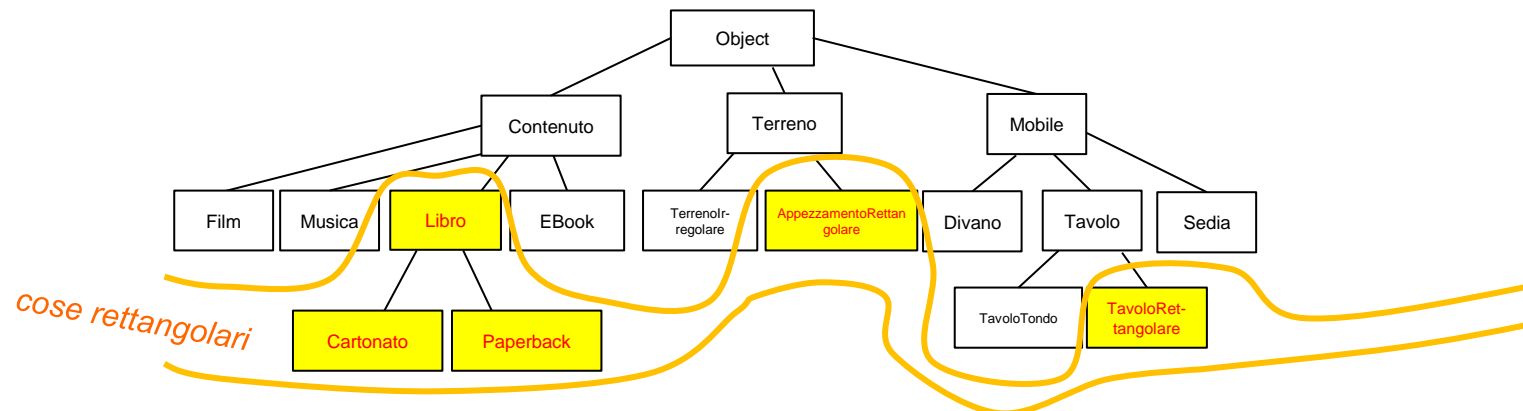
- ma un riferimento a **Rettangolare** può spaziare su *tutto ciò che è rettangolare*, ovunque (=di qualunque classe) sia

RIFERIMENTI A INTERFACCE vs. RIFERIMENTI A CLASSI

- Dunque:
 - i riferimenti a **classi** catturano **viste «verticali»** della tassonomia



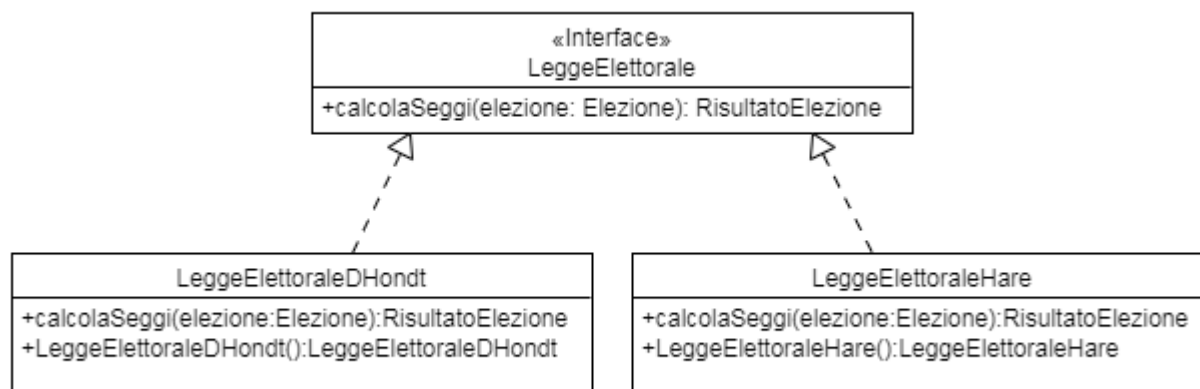
- i riferimenti a **interfacce** catturano **viste «orizzontali»** della tassonomia, che intercettano entità caratterizzate da certe proprietà



Un primo caso di studio: elezioni e leggi elettorali

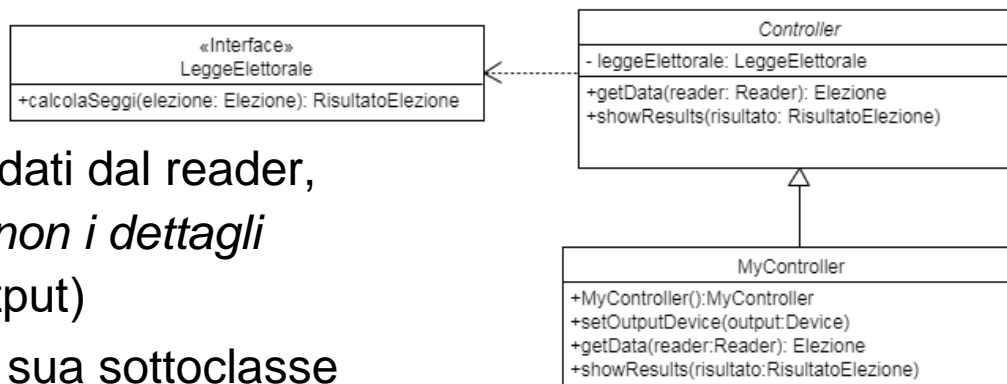
UN ESEMPIO: ELEZIONI

- Ogni elezione abbisogna di una *legge elettorale* che trasformi i *voti* in *seggi*, secondo un certo algoritmo
 - l'obiettivo è chiaro, ma il *modo di farlo* può variare anche molto
 - è il classico caso d'uso di una *interfaccia*
 - apposite *classi* implementeranno poi *specifiche leggi elettorali*
- Esempio:



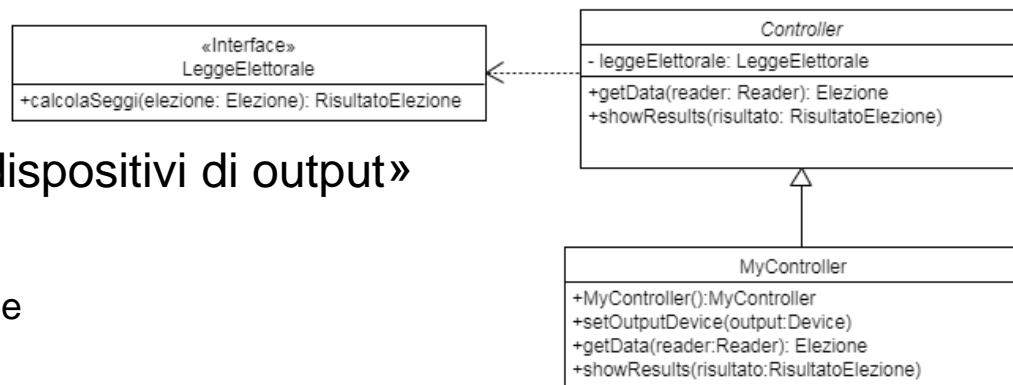
UN ESEMPIO: ELEZIONI

- Grazie all'interfaccia, **il resto dell'applicazione è *totalmente indipendente dalle specifiche leggi elettorali***
 - non sa neppure quali e quante siano: è quindi facile estenderne il set in futuro, aggiungerne di nuove, etc.
 - ad esempio, il Controller (qui, una classe astratta) *usa la legge elettorale* per ottenere i seggi, dati i voti
 - a tal fine incapsula il riferimento alla «Legge elettorale»
 - in quanto astratto, il Controller stabilisce cosa fare (recuperare i dati dal reader, mostrare i risultati) ma *non i dettagli* (es. *dove* emettere l'output)
 - ciò sarà compito di una sua sottoclasse concreta, ad esempio MyController



UN ESEMPIO: ELEZIONI

- A ben guardare, questa architettura usa in effetti anche *altre potenziali interfacce*
 - il metodo getData del Controller prende i dati da un «Reader»
 - il metodo setOutputDevice di MyController riceve un «Device»
- sono tutti concetti astratti!
- Reader potrebbe ben essere un'interfaccia, concretizzata poi da tanti possibili tipi di Reader concreti *che non ci interessa conoscere a questo stadio*
- idem Device: chissà quanti tipi concreti di «dispositivi di output» potrebbero esserci..
 - console di output testuale
 - finestra grafica
 - pannello per grafico a torta, istogramma...





UN ESEMPIO: ELEZIONI

- MORALE: le interfacce sono un potente strumento di *modellazione* e di *separazione di concetti e livelli*
- Consentono di *ragionare per concetti*, evidenziando le *relazioni fra le entità* e stabilendo *l'architettura logica*
- Prescindendo da ogni realizzazione concreta, sono la chiave del «*design for change*»
 - dietro ogni interfaccia si celano potenzialmente N diverse *realizzazioni intercambiabili*, che potremmo fare domani
 - ogni realizzazione concreta può essere sostituita / aggiornata *senza che il resto dell'applicazione se ne accorga*
- Slogan: «*interfaces first*»

Interfacce & Factory



INTERFACCE & FACTORY (1)

- La possibilità di *manipolare entità con certe caratteristiche*
 - espresse dal tipo-interfaccia
- senza dover stabilire a priori *cosa saranno esattamente*
 - ossia, *di che classe specifica* saranno
- è di **particolare interesse** nel caso delle **factory**
 - i metodi di una factory hanno solitamente come **tipo di ritorno** proprio **un'interfaccia**
 - così, **possono costruire e restituire un'istanza di una qualunque classe che la implementi**, in base alle loro scelte e logiche interne
 - al cliente non deve interessare *cosa sia veramente* l'oggetto che gli viene restituito: *basta che rispetti l'interfaccia!*



INTERFACCE & FACTORY (2)

APPROCCIO

1. il cliente chiede un'entità con le caratteristiche espresse dall'*interfaccia che costituisce il tipo di ritorno*
 - il cliente vede e usa sempre e solo il tipo-interfaccia
2. la factory restituisce un'istanza di una "opportuna classe" che implementi tale interfaccia, scegliendo quale usare in base a sue logiche interne
 - possibilità di *cambiare implementazione* quando si vuole
 - possibilità di *scegliere dinamicamente* quali oggetti creare
 - .. e molte altre simpatiche forme di *disaccoppiamento*

```
Rettangolo r = FormFactory.getRettangolo (...);  
Sensore s = SensorFactory.getSensor (...);
```

Java

C#

~Kotlin

~Scala



INTERFACCE & FACTORY (3)

```
public class FormFactory {  
    public static Rettangolo getRettangolo(...) {  
        return new ... // decide che oggetto creare in base ai parametri, alle risorse...  
    }  
}
```

Ad esempio, di norma `ImplRettangolo` ma magari un `ImplQuadrato` se vede che i lati sono uguali... ☺

Java

C#

```
public object FormFactory {  
    public fun getRettangolo(...) : Rettangolo {  
        return ...;  
    }  
}
```

Kotlin

```
object FormFactory {  
    def getRettangolo(...) : Rettangolo = {  
        return ...  
    }  
}
```

Scala



INTERFACCE & FACTORY (4)

- Da alcuni anni è anche possibile *internalizzare la factory* nelle interfacce, che possono *contenere metodi statici*
 - tali metodi *non* fanno parte dell'interfaccia intesa come tipo: sono semplicemente «ospitati» lì per comodità
 - in Java: da Java 8 (2014)
 - in C#: da C# 8.0 (2019)
- In Scala e Kotlin, analogo risultato può essere ottenuto definendo appositi *companion objects*
 - Scala: un *object* con lo stesso nome dell'interfaccia
 - Kotlin: un *companion object* interno all'interfaccia stessa



INTERFACCE & FACTORY (5)

```
public interface Rettangolo {
```

Java 8

C# 8

```
// tutti i metodi dell'interfaccia classica
```

```
public static Rettangolo getRettangolo(...) {
```

```
    return new ImplRettangolo(...);
```

```
}
```

```
}
```

```
public interface Rettangolo {
```

Kotlin

```
// tutti i metodi dell'interfaccia classica
```

```
companion object {
```

Companion object: in Kotlin, un object speciale definito *internamente* all'interfaccia stessa

```
    public fun getRettangolo(...) : Rettangolo {
```

```
        return ImplRettangolo(...); }
```

```
}
```

```
}
```

(di fatto, un posto dove scrivere i metodi «ex statici»...)



INTERFACCE & FACTORY (6)

```
public interface Rettangolo {
```

Java 8

C# 8

```
    // tutti i metodi dell'interfaccia classica
```

```
    public static Rettangolo getRettangolo(...) {
```

```
        return new ImplRettangolo(...);
```

```
    }
```

```
}
```

Si può migliorare ancora: perché ripetere "Rettangolo" tante volte?

```
trait Rettangolo {
```

Scala

```
    // tutti i metodi dell'interfaccia classica
```

```
}
```

```
object Rettangolo {
```

```
    def getRettangolo(...) : Rettangolo = {
```

```
        return new ImplRettangolo(...);
```

```
    }
```

```
}
```

Companion object: in Scala, un object omonimo del tratto, ma definito esternamente ad esso

(di fatto, un posto dove scrivere i metodi «ex statici»...)

INTERFACCE & FACTORY (7)

- **Internalizzare la factory** presenta vantaggi:
 - si evita il **proliferare di factory** dal nome simile (ma non uguale) all'interfaccia che supportano
 - si mantiene l'**unitarietà** fra **astrazione** e **fabbrica dell'astrazione**
- A fronte di ciò, vi è anche qualche svantaggio:
 - si **perde la fabbrica "generale"** che costruisce "ogni astrazione"

CON LA FABBRICA UNICA GENERALE:

- la fabbrica contiene *vari* metodi factory, *uno per tipo di cosa da costruire*
→ i metodi devono avere un *nome che indichi cosa costruiscono*

CON LA FABBRICA INTERNALIZZATA NELL'INTERFACCIA:

- ogni interfaccia contiene *il solo metodo factory* di *quel particolare* tipo di oggetto
- l'indicazione su cosa viene costruito è già nel nome dell'interfaccia
→ **si può accorciare il nome del metodo** → spesso si usa **of**



INTERFACCE & FACTORY (8)

```
public interface Rettangolo {  
    // tutti i metodi dell'interfaccia classica  
    public static Rettangolo of(...){  
        return new ImplRettangolo(...);  
    }  
}
```

Java 8

C# 8

```
trait Rettangolo {  
    // metodi dell'interfaccia  
}  
object Rettangolo {  
    def of(...) : Rettangolo = {  
        return new ImplRettangolo(...);  
    }  
}
```

Scala

```
public interface Rettangolo {  
    // metodi dell'interfaccia  
    companion object {  
        public fun of(...) : Rettangolo{  
            return ImplRettangolo(...);  
        }  
    }  
}
```

Kotlin

```
Rettangolo r = Rettangolo.of(...);  
Sensore s = Sensor.of(...);
```

Netto miglioramento
dell'usabilità lato cliente!

Il caso di studio "forme geometriche": reingegnerizzazione con interfacce



FORME GEOMETRICHE: IL RITORNO ☺

- La tassonomia di forme geometriche *modellata tramite classi* ha mostrato *limiti espressivi*
 - l'ereditarietà singola consentiva di classificare solo *per sottoinsiemi*
→ *un solo criterio per volta*
 - la realtà però richiedeva *due criteri ortogonali fra loro*
lati paralleli vs. *angoli retti*
 - i limiti dell'ereditarietà singola non permettevano di collocare bene rettangoli, quadrati, etc.
- Possiamo ora *riprogettarla* con le *interfacce*, sfruttando anche l' *ereditarietà multipla*.

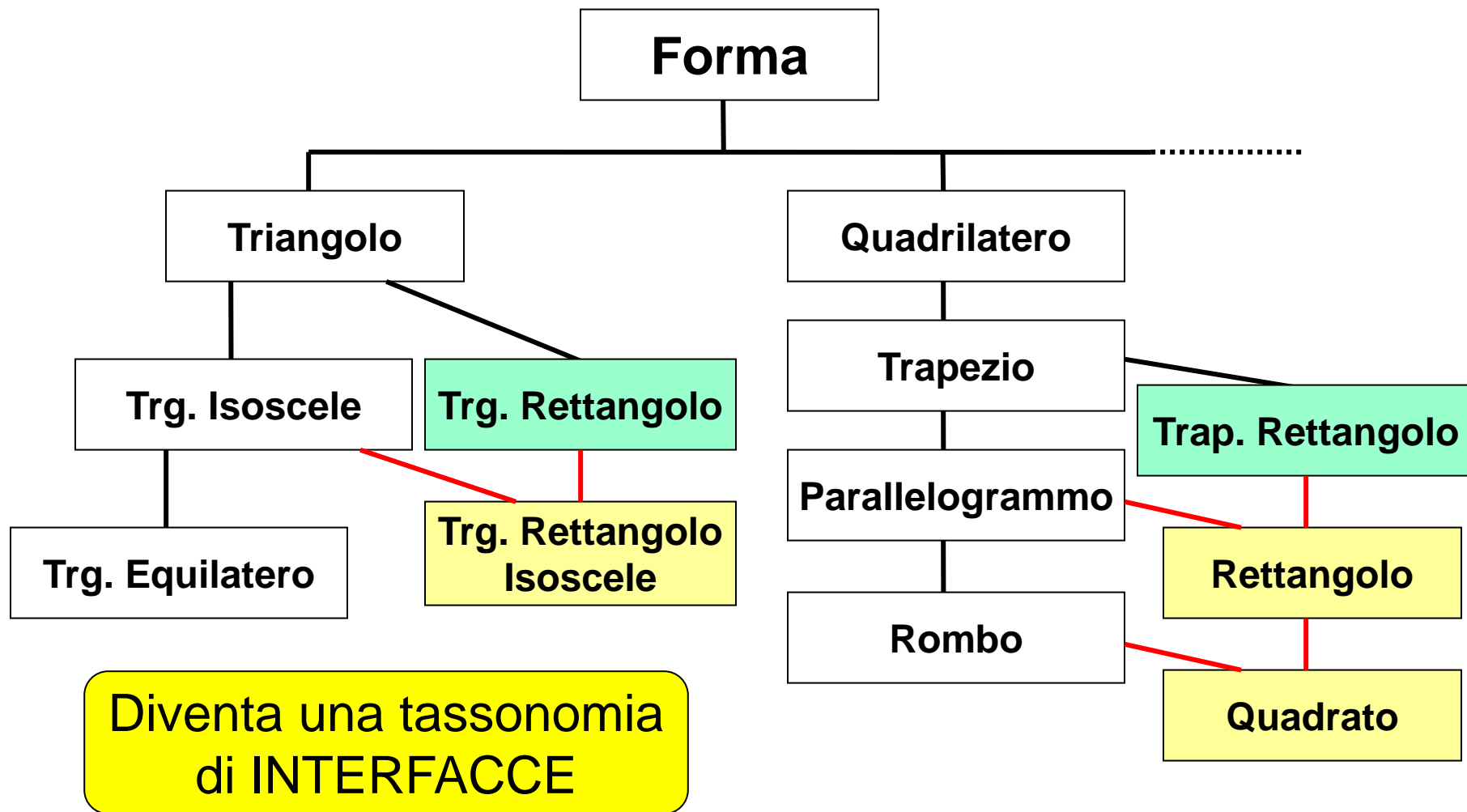


FORME GEOMETRICHE MODELLATE CON INTERFACCE

- Usando interfacce, possiamo
 - a) sfruttare l'*ereditarietà multipla* per:
 - esprimere *intersezioni* fra insiemi
 - *applicare più criteri di classificazione* componendoli in modo naturale
 - criterio dei *lati paralleli*
 - criterio degli *angoli retti*
 - b) prevedere *factory* (eventualmente internalizzate)
- Riconsideriamo perciò la tassonomia di classi definita in passato, **reinterpretando tutte le entità come *interfacce***



TASSONOMIA DI INTERFACCE

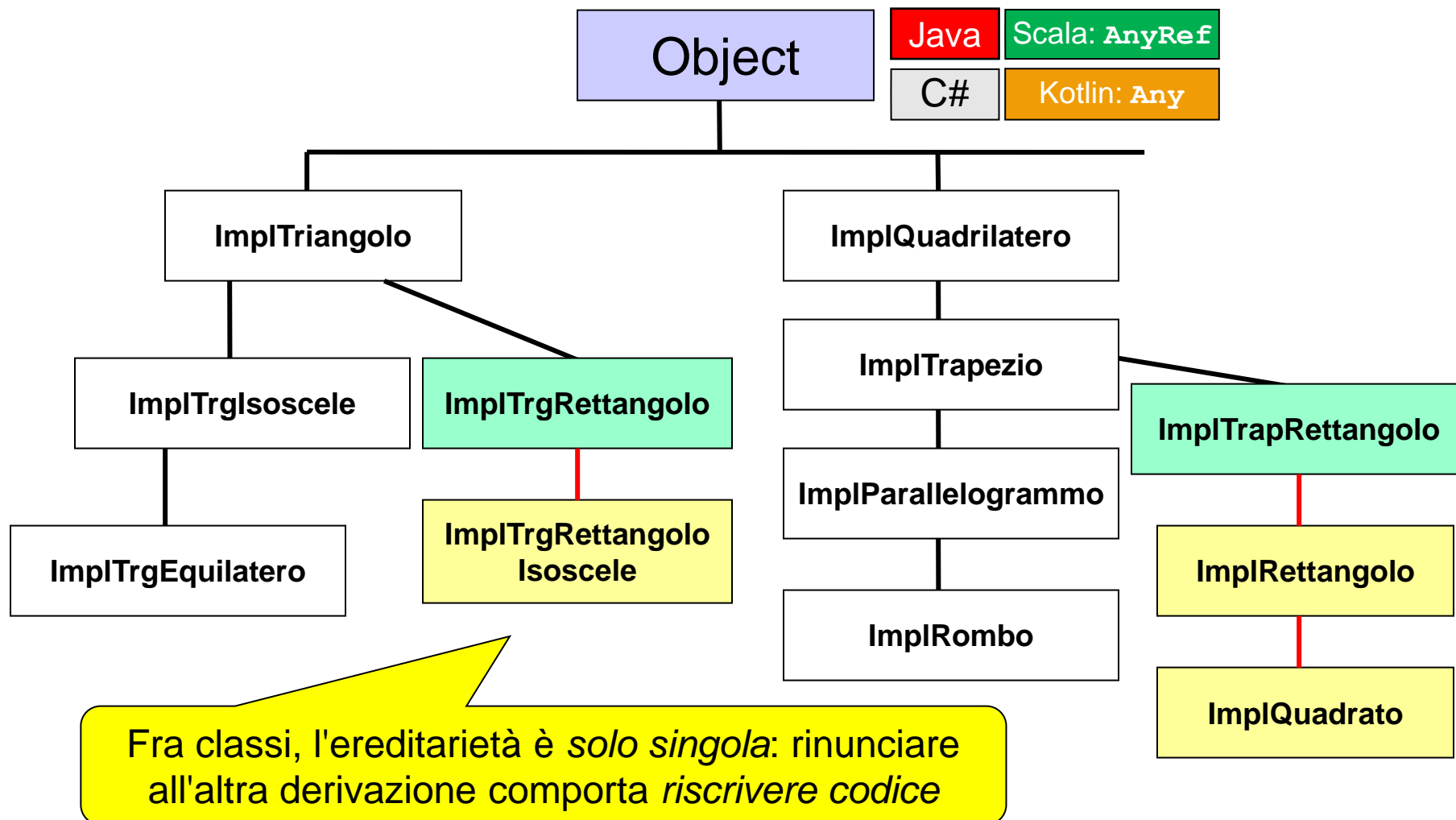




.. E L'IMPLEMENTAZIONE ?

- La tassonomia di interfacce ora è soddisfacente:
come implementarla?
 - Fra classi non c'è ereditarietà multipla, quindi non si potrà replicare "pari pari" la tassonomia di interfacce sulle classi sottostanti
 - Bisognerà fare delle scelte: *criteri di riuso, efficienza, etc.*
- Comunque, *i clienti vedono solo le interfacce*
 - Le classi *retrocedono sullo sfondo*: diventano solo un mezzo per implementare l'astrazione espressa dall'interfaccia
 - La scelta di quali classi definire, e in che relazione porle fra loro, tenderà a privilegiare la "linea di derivazione" che permette di riusare (in modo pulito) più codice
 - un'opportuna *factory (eventualmente internalizzata)* chiuderà il cerchio, nascondendo del tutto la creazione delle istanze.

TASSONOMIA DI CLASSI



LA NUOVA ARCHITETTURA: INTERFACCE



All Classes

Forma

ImplParallelogrammo

ImplQuadrato

ImplQuadrilatero

ImplRettangolo

ImplRombo

ImplTrapezio

ImplTrapezioRettangolo

ImplTriangolo

ImplTriangoloEquilatero

ImplTriangoloIsoscele

ImplTriangoloRettangolo

ImplTriangoloRettangoloIsoscele

Parallelogrammo

Quadrato

Quadrilatero

Rettangolo

Rombo

Trapezio

TrapezioRettangolo

Triangolo

TriangoloEquilatero

TriangoloIsoscele

TriangoloRettangolo

TriangoloRettangoloIsoscele

PACKAGE CLASS TREE DEPRECATED INDEX HELP

PREV CLASS NEXT CLASS FRAMES NO FRAMES

SUMMARY: NESTED | FIELD | CONSTR | METHOD DETAIL: FIELD | CONSTR | METHOD

Interface Forma

Java

All Known Subinterfaces:

Parallelogrammo, Quadrato, Quadrilatero, Rettangolo, Rombo, Trapezio, TrapezioRettangolo, Triangolo, TriangoloEquilatero, TriangoloIsoscele, TriangoloRettangolo, TriangoloRettangoloIsoscele

All Known Implementing Classes:

ImplParallelogrammo, ImplQuadrato, ImplQuadrilatero, ImplRettangolo, ImplRombo, ImplTrapezio, ImplTrapezioRettangolo, ImplTriangolo, ImplTriangoloEquilatero, ImplTriangoloIsoscele, ImplTriangoloRettangolo, ImplTriangoloRettangoloIsoscele

public interface **Forma**

Questa *interfaccia* modella il concetto geometrico di *forma*. Si ipotizza che ogni forma sia caratterizzata da *area*, *perimetro* e naturalmente il proprio *nome*. In più, si stabilisce che ogni forma debba prevedere un'opportuna versione del metodo `toString`.

ALMA MATER STUDIORUM - UNIVERSITÀ DI BOLOGNA



L'INTERFACCIA Forma

```
public interface Forma
```

Questa *interfaccia* modella il concetto geometrico di *forma*. Si ipotizza che ogni forma sia caratterizzata da *area*, *perimetro* e naturalmente il proprio *nome*. In più, si stabilisce che ogni forma debba prevedere un'opportuna versione del metodo `toString`.

Method Summary

Java

All Methods	Instance Methods	Abstract Methods
Modifier and Type	Method and Description	
double	area()	Restituisce l'area della figura.
java.lang.String	nome()	Restituisce il nome della figura.
double	perimetro()	Restituisce il perimetro della figura.
java.lang.String	toString()	Restituisce la descrizione della figura sotto forma di stringa.



L'INTERFACCIA Triangolo

```
public interface Triangolo
extends Forma
```

Questa *interfaccia* modella il concetto geometrico di *triangolo*. Un triangolo è univocamente caratterizzato dai tre lati: ogni altra proprietà (perimetro, area, angoli, altezze) è calcolata a partire da essi.

Method Summary

Java

All Methods	Instance Methods	Abstract Methods
Modifier and Type	Method and Description	
double	altezzaa1()	Restituisce l'altezza relativa al "primo lato" (il primo fornito al costruttore)
double	altezzaa2()	Restituisce l'altezza relativa al "secondo lato" (il secondo fornito al costruttore)
double	altezzaa3()	Restituisce l'altezza relativa al "terzo lato" (il terzo fornito al costruttore)
double	angoloOppostoLato1()	Restituisce l'angolo (in gradi) opposto al "primo lato" (il primo fornito al costruttore)
double	angoloOppostoLato2()	Restituisce l'angolo (in gradi) opposto al "secondo lato" (il secondo fornito al costruttore)
double	angoloOppostoLato3()	Restituisce l'angolo (in gradi) opposto al "terzo lato" (il terzo fornito al costruttore)
double	lato1()	Restituisce il "primo lato" (il primo fornito al costruttore)
double	lato2()	Restituisce il "secondo lato" (il secondo fornito al costruttore)
double	lato3()	



L'INTERFACCIA TriangoloRettangolo

Interface TriangoloRettangolo

All Superinterfaces:

Forma, Triangolo

All Known Subinterfaces:

TriangoloRettangoloIsoscele

All Known Implementing Classes:

ImplTriangoloRettangolo, ImplTriangoloRettangoloIsoscele

```
public interface TriangoloRettangolo
    extends Triangolo
```

Questa interfaccia modella il concetto geometrico di *triangolo rettangolo*, univocamente caratterizzato da DUE valori: i CATETI. Ogni altra proprietà (ipotenusa, perimetro, area, angoli, altezze) è calcolata a partire da essi.

Java

Method Summary

Java

All Methods Instance Methods Abstract Methods

Modifier and Type	Method and Description
double	catetoMaggiore() Restituisce il cateto maggiore
double	catetoMinore() Restituisce il cateto minore
double	ipotenusa() Restituisce l'ipotenusa

Methods inherited from interface Triangolo

altezzaa1, altezzaa2, altezzaa3, angoloOppostoLato1, angoloOppostoLato2, angoloOppostoLato3, lato1, lato2, lato3

Methods inherited from interface Forma

area, nome, perimetro, toString



L'INTERFACCIA Quadrilatero

Interface Quadrilatero

Java

All Superinterfaces:

Forma

All Known Subinterfaces:

Parallelogrammo, Quadrato, Rettangolo, Rombo, Trapezio, TrapezioRettangolo

All Known Implementing Classes:

ImplParallelogrammo, ImplQuadrato, ImplQuadrilatero, ImplRettangolo, ImplRombo, ImplTrapezio, ImplTrapezioRettangolo

```
public interface Quadrilatero  
extends Forma
```

Questa *interfaccia* modella il concetto geometrico di *quadrilatero*. A differenza del triangolo, questo **non** è univocamente caratterizzato dai quattro lati: *occorre anche un angolo*, oltre all'informazione se la figura sia convessa o concava (infatti, quattro lati e un angolo rendono ancora possibili due figure distinte, una convessa e una concava, idealmente costituite da due triangoli appoggiati o dalla stessa parte, o da parti diverse di una diagonale comune). Ogni altra proprietà può essere calcolata di conseguenza.

L'INTERFACCIA Quadrilatero

Interface Quadrilatero

All Superinterfaces:

Forma

All Known Subinterfaces:

Parallelogrammo, Quad

All Known Implementing Classes:

ImplParallelogrammo, ImplTrapezio, ImplTra

```
public interface Quadrilatero
    extends Forma
```

Questa *interfaccia* model...
questo **non** è univocamente
all'informazione se la figura
ancora possibili due figure
triangoli appoggiati o dal
proprietà può essere calcolata

All Methods	Instance Methods	Abstract Methods
Modifier and Type	Method and Description	
double	angoloAB()	Restituisce l'angolo compreso fra primo e secondo lato (fornito al costruttore)
double	angoloAD()	Restituisce l'angolo compreso fra quarto e primo lato
double	angoloBC()	Restituisce l'angolo compreso fra secondo e terzo lato
double	angoloCD()	Restituisce l'angolo compreso fra terzo e quarto lato
double	diagonaleX()	Restituisce l'altra delle due diagonali
double	diagonaleZ()	Restituisce una delle due diagonali (quella su cui si scompone la figura in triangoli)
boolean	èConcavo()	Predicato che controlla se la figura è convessa o concava
double	lato1()	Restituisce il "primo lato" (il primo fornito al costruttore)
double	lato2()	Restituisce il "secondo lato" (il secondo fornito al costruttore)
double	lato3()	Restituisce il "terzo lato" (il terzo fornito al costruttore)
double	lato4()	Restituisce il "quarto lato" (il quarto fornito al costruttore)
double	sommaAngoli()	Restituisce l'angolo compreso fra quarto e primo lato

Java



L'INTERFACCIA Rettangolo

```
public interface Rettangolo
extends TrapezioRettangolo, Parallelogrammo
```

Java

Questa interfaccia modella il concetto geometrico di *rettangolo*, ossia quel trapezio rettangolo *che è anche un parallelogrammo* e ha quindi i due lati obliqui paralleli fra loro e perpendicolari agli altri due. Gli angoli sono tutti retti.

Method Summary

All Methods

Instance Methods

Abstract Methods

Modifier and Type

Method and Description

double

altezza()

Restituisce l'altezza del rettangolo (primo valore passato al costruttore)

double

base()

Restituisce la base del rettangolo (secondo valore passato al costruttore)

Methods inherited from interface Trapezio

baseMaggiore, baseMinore

Methods inherited from interface Quadrilatero

angoloAB, angoloAD, angoloBC, angoloCD, diagonaleX, diagonaleZ, èConcavo, lato1, lato2, lato3, lato4, sommaAngoli

Methods inherited from interface Forma

area, nome, perimetro, toString



LA NUOVA ARCHITETTURA: CLASSI

All Classes

Forma
ImplParallelogrammo
ImplQuadrato
ImplQuadrilatero
ImplRettangolo
ImplRombo
ImplTrapezio
ImplTrapezioRettangolo
ImplTriangolo
ImplTriangoloEquilatero
ImplTriangoloIsoscele
ImplTriangoloRettangolo
ImplTriangoloRettangoloIsoscele
Parallelogrammo
Quadrato
Quadrilatero
Rettangolo
Rombo
Trapezio
TrapezioRettangolo
Triangolo
TriangoloEquilatero
TriangoloIsoscele
TriangoloRettangolo
TriangoloRettangoloIsoscele

PACKAGE CLASS TREE DEPRECATED INDEX HELP

PREV CLASS NEXT CLASS FRAMES NO FRAMES

SUMMARY: NESTED | FIELD | CONSTR | METHOD DETAIL: FIELD | CONSTR | METHOD

Interface Forma

All Known Subinterfaces:

Parallelogrammo, Quadrato, Quadrilatero, Rettangolo, Rombo, Trapezio, TrapezioRettangolo, Triangolo, TriangoloEquilatero, TriangoloIsoscele, TriangoloRettangolo, TriangoloRettangoloIsoscele

All Known Implementing Classes:

ImplParallelogrammo, ImplQuadrato, ImplQuadrilatero, ImplRettangolo, ImplRombo, ImplTrapezio, ImplTrapezioRettangolo, ImplTriangolo, ImplTriangoloEquilatero, ImplTriangoloIsoscele, ImplTriangoloRettangolo, ImplTriangoloRettangoloIsoscele

```
public interface Forma
```

Questa *interfaccia* modella il concetto geometrico di *forma*. Si ipotizza che ogni forma sia caratterizzata da *area*, *perimetro* e naturalmente il proprio *nome*. In più, si stabilisce che ogni forma debba prevedere un'opportuna versione del metodo `toString`.

CLASSI: ImplTriangoloRettangolo

Java

Class ImplTriangoloRettangolo

```
java.lang.Object  
  ImplTriangolo  
    ImplTriangoloRettangolo
```

Qui si è scelto di implementarla derivando direttamente da **ImplTriangolo**

All Implemented Interfaces:

Forma, Triangolo, TriangoloRettangolo

Ma quello che conta è che implementi le interfacce giuste!

Direct Known Subclasses:

ImplTriangoloRettangoloIsoscele

```
public class ImplTriangoloRettangolo  
  extends ImplTriangolo  
  implements TriangoloRettangolo
```

Questa classe realizza il concetto geometrico di *triangolo rettangolo* definito dall'omonima interfaccia. Un triangolo rettangolo è univocamente caratterizzato da DUE valori: i CATETI. Ogni altra proprietà può essere calcolata a partire da essi.

Constructor Summary

Constructors

Constructor and Description

ImplTriangoloRettangolo(double cateto1, double cateto2)

Costruisce un triangolo rettangolo dati i *cateti*, nessuno dei quali dev'essere negativo o nullo.



UNA FABBRICA DI FORME

Nascondendo i nomi delle classi *dall'unico posto dove si sarebbero ancora viste*, la factory completa il quadro

Ci sono varie possibilità:

- unica factory + **lasciar scegliere all'utente** cosa creare
 - in tal caso, la fabbrica dovrà offrire metodi specializzati per ogni tipo di forma da creare: `getTriangolo`, `getTriangoloIsoscele`, `getTriangoloEquilatero`, `getTriangoloRettangolo`...
- unica factory + **delegare completamente alla fabbrica** la scelta di quale astrazione creare
 - in tal caso, la fabbrica offrirà un unico metodo per ogni tipo fondamentale di forma da creare: `getTriangolo` (o magari `of`)
- molte factory **internalizzate in ogni singola astrazione**
 - ogni interfaccia offrirà un metodo (`of`) per crearne istanze



UNA FABBRICA DI FORME

Nascondendo i nomi delle classi *dall'unico posto dove si sarebbero ancora viste*, la factory completa il quadro

Lato cliente:

Java

```
Triangolo t = FormFactory.getTriangolo(a,b,c);
```

Factory:

Java

```
public static Triangolo getTriangolo(int x,int y,int z){  
    if (isoscele) return new ImplTrgIsoscele(x,y,z); else  
    if (equilatero) return new ImplTrgEquilatero(x,y,z);  
    else ... // gli altri casi  
}
```

La factory può *decidere dinamicamente* quale oggetto costruire (e restituire) *in base alla situazione e agli argomenti ricevuti*



LA FABBRICA INTERNALIZZATA

In Java ≥ 8 , C# ≥ 8 , Scala, Kotlin (via companion objects) **ogni interfaccia può *internalizzare* la sua factory:**

Lato cliente:

Java

```
Triangolo t = Triangolo.of(a,b,c);
```

Eventuale fabbrica «dei triangoli» (**Triangoli**):

Java

```
public static Triangolo of(int x,int y,int z){  
    if (isoscele) return new ImplTrgIsoscele(x,y,z); else  
    if (equilatero) return new ImplTrgEquilatero(x,y,z);  
    else return new ImplTriangolo (x,y,z);  
}  
  
// + metodo of a 2 soli argomenti (per trg isosceli)  
// + metodo of a 1 solo argomento (per trg equilateri)
```



UN PICCOLO MAIN DI PROVA

```
object TanteForme {  
  def main(args: Array[String]) : Unit = {  
    val forme : Array[Forma] = Array(  
      Triangolo.of(2,3,4),  
      TriangoloIsoscele.of(2,3),  
      TriangoloEquilatero.of(3) );  
    for(forma <- forme) println(forma);  
  
    val forme2 : Array[Forma] = Array(  
      Triangoli.of(2,3,4), Triangoli.of(2,3), Triangoli.of(3) );  
    for(forma <- forme2) println(forma);  
  }  
}
```

~Java

Scala

Factory internalizzate nelle
singole interfacce/tratti:
il cliente indica cosa costruire

Unica factory generale dei
triangoli: *la factory stabilisce
cosa costruire*

```
Triangolo di area 2.9047375096555625 e perimetro 9.0  
Triangolo isoscele di area 2.8284271247461903 e perimetro 8.0  
Triangolo equilatero di area 3.8971143170299736 e perimetro 9.0
```



ORGANIZZAZIONE GLOBALE

- Le classi retrocedono sullo sfondo: non sono più pubbliche
- Sono pubbliche solo le interfacce (e le factory)
 - factory internalizzata per ogni interfaccia (+ factory globale..?)

Ciò richiede di *confezionare un package* in cui siano:

- pubbliche le interfacce (ed eventualmente la factory)
- private le classi usate per implementare tali servizi.

RISULTATO:

- pieno supporto all'ereditarietà multipla *lato utente*
- apertura ai cambiamenti (modifiche all'implementazione)
- scelte di costruzione effettuate sul momento, in base alle politiche "aziendali" e alla situazione contingente.



SKETCH IMPLEMENTAZIONE: Java

```
interface Forma {  
    public double area();  
    public double perimetro();  
    public String nome();  
}
```

Java

```
interface Parallelogrammo  
    extends Trapezio {  
}
```

```
interface Quadrilatero  
    extends Forma {  
    public double lato1();  
    ...  
    public double diagonaleZ();  
    public double diagonaleX();  
    public double angoloABC();  
    ...  
}
```

...

```
interface Rettangolo extends TrapezioRettangolo, Parallelogrammo {  
    public double altezza();  
    public double base();  
}
```

L'altezza in realtà sarebbe già definita nel trapezio rettangolo...

```
interface Quadrato extends Rettangolo, Rombo {  
    // VUOTA: serve per unire le proprietà di Rettangolo e Rombo  
}
```



SKETCH IMPLEMENTAZIONE: Java

Una classe Java può *estenderne* solo un'altra (ereditarietà singola) ma può *implementare* molte altre interfacce

```
class ImplQuadrato extends Impl Rettangolo implements Quadrato{  
    public ImplQuadrato(double lato){    super(lato, lato); }  
    // NOTA: lato() era già implementata in ImplRombo ma occorre  
    // riscriverla perché va persa, dato che ImplQuadrato deriva  
    // solo da Impl Rettangolo  
    public double lato() { return lato1(); }  
    public String nome() { return "Quadrato"; }  
}
```

Java

```
class Impl Rettangolo extends ImplTrapezio Rettangolo  
    implements Rettangolo {  
    public Impl Rettangolo(double altezza, double base){  
        super(altezza,base,base); }  
    ...  
}
```

SKETCH IMPLEMENTAZIONE: Scala

```
trait Forma {  
  def area(): Double;  
  def perimetro(): Double;  
  def nome() : String;  
}
```

Scala

```
trait Parallelogrammo  
  extends Trapezio {  
}
```

```
trait Quadrilatero extends Forma {  
  def lato1() : Double ;  
  def lato2() : Double ;  
  ...  
  def diagonaleZ() : Double ;  
  def diagonaleX() : Double ;  
  def angoloABC() : Double ;  
  ...  
}
```

...

```
trait Rettangolo extends TrapezioRettangolo with Parallelogrammo {  
  def base() : Double;  
}
```

L'altezza è già definita
nel trapezio rettangolo!

Composizione MIX-IN
Un *tratto* Scala può estenderne
uno e comporsi (*with*) con gli altri

```
trait Quadrato extends Rettangolo with Rombo {  
  // VUOTO: serve per unire le proprietà di Rettangolo e Rombo  
}
```




SKETCH IMPLEMENTAZIONE: Scala

Composizione MIX-IN

Una classe Scala può *estenderne* solo un'altra (ereditarietà singola) ma può *comporsi* con (molti) altri *tratti*

Scala

```
class ImplQuadrato(lato:Double)
    extends Impl Rettangolo(lato,lato) with Quadrato {
// NOTA: lato() era già implementata in ImplRombo ma occorre
// riscriverla perché va persa, dato che ImplQuadrato deriva
// solo da Impl Rettangolo
    override def lato() : Double = { return lato1(); }
    override def nome() : String = { return "Quadrato"; }
}
```

```
class Impl Rettangolo(altezza:Double, base:Double)
    extends Impl Trapezio Rettangolo(altezza,base,base)
    with Rettangolo {
    ...
}
```

I parametri del costruttore si passano *solo alla classe base*, non ai tratti



SKETCH IMPLEMENTAZIONE: Kotlin

In Kotlin, l'operatore `:` funge sia da «extends» che da «implements»

```
public class ImplQuadrato(lato:Double) :  
    Impl Rettangolo(lato,lato), Quadrato {  
  
    public constructor(lato:Number) : this(lato.toDouble());  
  
    public override fun lato() : Double { return lato1(); }  
    public override fun nome() : String { return "Quadrato"; }  
}
```

Kotlin

Kotlin è rigido sui tipi numerici, quindi è utile un costruttore ausiliario per accettare anche interi (e non solo double)

```
public open class Impl Rettangolo(altezza:Double, base:Double)  
    : Impl Trapezio Rettangolo(altezza,base,base), Rettangolo {  
    ...  
}
```

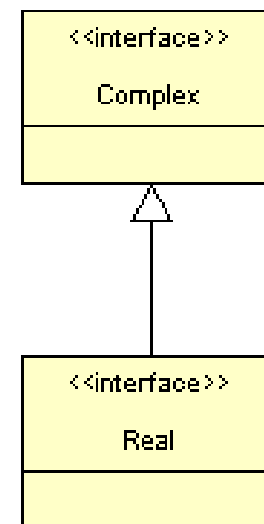
Cruciale dichiarare le classi *open* per permettere ereditarietà!

I parametri del costruttore si passano *solo alla classe base*, non ai tratti

Il caso di studio "complessi & reali": reingegnerizzazione con interfacce

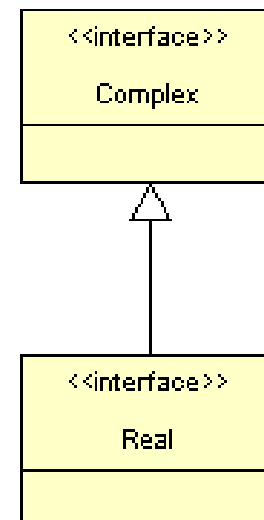
IL CASO DI STUDIO – RIASSUNTO

- In passato avevamo concluso che i reali dovessero essere un sottoinsieme dei complessi (non viceversa)
- All'epoca lo avevamo realizzato con classi
 - corretto, ma inefficiente (tutti i reali hanno parte $im = 0$)
- REFACTORING: usiamo *interfacce* per esprimere *proprietà osservabili e operazioni*
 - **Complex** è l'interfaccia-base
 - **Real** la specializza



L' INTERFACCIA Complex

```
public interface Complex {
    public double getReal();
    public double getIm();
    public double module();
    public Complex cgt();
    public Complex divByFactor(double x);
    public Complex sum(Complex z);
    public Complex sub(Complex z);
    public Complex mul(Complex z);
    public Complex div(Complex z);
    public String toString();
}
```



Non ci sono dati perché non si stabilisce qui come i Complex siano rappresentati dentro.

Qui c'è solo la *vista esterna*: si definiscono solo i metodi di accesso alle proprietà previste.

L' INTERFACCIA `Real`

```
public interface Real extends Complex {  
    public Real sum(Real x);  
    public Real sub(Real x);  
    public Real mul(Real x);  
    public Real div(Real x);  
}
```

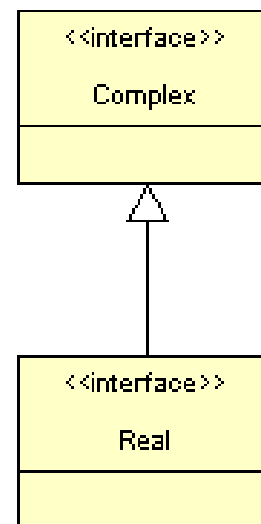
NB: non occorre ripetere `toString`, perché la sua signature è identica a quella di `Complex`

Java

~C#

~Kotlin

~Scala



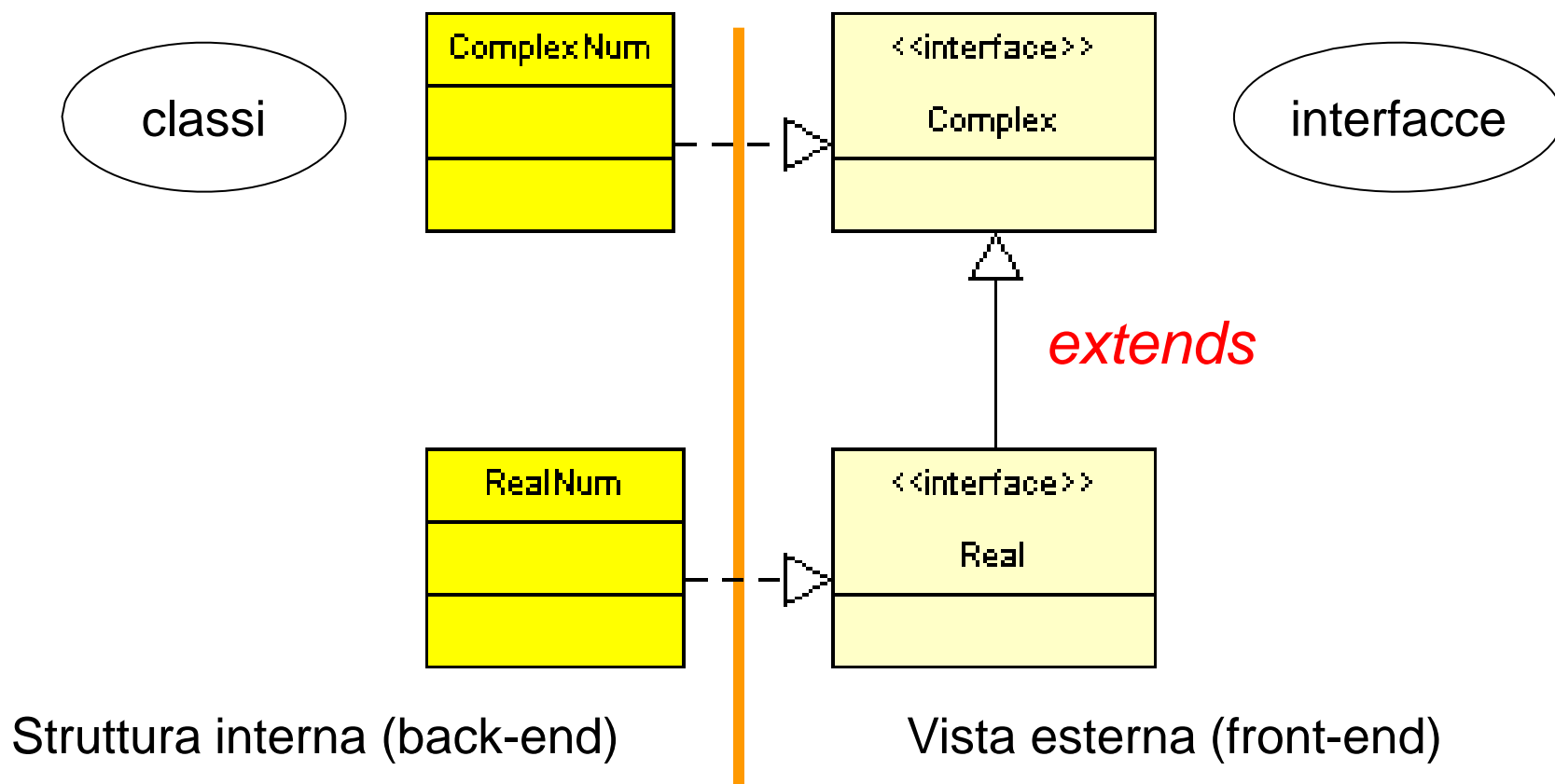


IMPLEMENTAZIONE

- Per implementare le interfacce, dobbiamo usare classi
- Stabiliamo che
 - **RealNum** implementi **Real**
 - **ComplexNum** implementi **Complex**
- Siamo liberi di decidere se le classi siano in qualche relazione fra loro (e se sì, quale)
 - non è detto che siano in una qualche relazione fra loro
 - anzi, possono benissimo essere due classi indipendenti
 - fare una scelta o l'altra *cambia la struttura interna del sistema software, ma non la vista esterna percepita dall'utente*

PROGETTO

L'architettura complessiva:





LA CLASSE RealNum

```
public class RealNum implements Real {
```

```
    protected double re;
```

```
    public RealNum() { re=0; }
```

```
    public RealNum(double x) { re=x; }
```

```
    public double getReal() { return re; }
```

```
    public double getIm() { return 0; }
```

```
    public double module() { return re<0 ? -re : re; }
```

```
    // operazioni con entrambi gli operandi reali
```

```
    public Real sum(Real x){return new RealNum(re + x.getReal());}
```

```
    public Real sub(Real x){return new RealNum(re - x.getReal());}
```

```
    public Real mul(Real x){return new RealNum(re * x.getReal());}
```

```
    public Real div(Real x){return new RealNum(re / x.getReal());}
```

```
    ... // segue
```

Java

~C#

~Kotlin

~Scala



LA CLASSE RealNum

```
...  
// operazioni con this reale ma operando Complex  
public Complex sum(Complex z) {  
    return new ComplexNum( re+z.getReal(), z.getIm() ); }  
public Complex sub(Complex z) {  
    return new ComplexNum( re-z.getReal(), z.getIm() ); }  
public Complex mul(Complex z) {  
    return new ComplexNum( re*z.getReal(), re*z.getIm() ); }  
public Complex div(Complex z) {  
    return new ComplexNum( re/z.getReal(), re/z.getIm() ); }  
public Complex cgt() { return this; }  
public Complex divByFactor(double x) {  
    return new RealNum(re/x); }  
public String toString() {  
    return Double.toString(re); }  
}
```

Java

~C#

~Kotlin

~Scala

Il coniugato di un Real è il Real stesso

Il risultato è in effetti un Real, ma l'interfaccia prevede un Complex



LA CLASSE ComplexNum

```
public class ComplexNum implements Complex {  
    protected double re, im;  
    public ComplexNum() { re = im = 0; }  
    public ComplexNum(double x) { re = x; im = 0; }  
    public ComplexNum(double x, double y) { re = x; im = y; }  
  
    public double getReal() { return re; }  
    public double getIm() { return im; }  
    public double module() { return Math.sqrt(re*re+im*im); }  
    public Complex cgt() { return new ComplexNum(re, -im); }  
    public Complex divByFactor(double x) {  
        return new ComplexNum(re/x, im/x); }  
    public Complex sum(Complex z) {  
        return new ComplexNum( re+z.getReal(), im+z.getIm()); }  
    public Complex sub(Complex z) {  
        return new ComplexNum( re-z.getReal(), im-z.getIm()); }  
    ...  
}
```

Java

~C#

~Kotlin

~Scala



LA CLASSE ComplexNum

```
...
public Complex mul(Complex z) {
    return new ComplexNum(
        re*z.getReal()-im*z.getIm(),
        re*z.getIm()+im*z.getReal()); }

public Complex div(Complex z) {
    double mod = z.module();
    return mul(z.cgt()).divByFactor(mod*mod);
}

public String toString() { // stampa di un ComplexNum
    String res;
    if (re==0.0 && im==0.0) return "0";
    if (re==0.0) res = ""; else
    { res = Double.toString(re); if (im>=0.0) res += "+"; }
    res += (im==1 || im==-1 ? "" : Double.toString(im)) + "i";
    return res;
}
}
```

Java

~C#

~Kotlin

~Scala



UN PRIMO COLLAUDO

```
public class Prova {  
    public static void main(String args)  
  
        Real    r1 = new RealNum(18.5),      r2 = new RealNum(3.14);  
        Complex c1 = new ComplexNum(-16, 0), c2 = new ComplexNum(3, 2),  
            c3 = new ComplexNum(0, -2);  
  
        Real    r = r1.sum(r2);  
        Complex c = c1.sum(c2);  
  
        System.out.println("r1 + r2 = " + r);           // il reale 21.64  
        System.out.println("c1 + c2 = " + c);           // il complesso -13+2i  
        c = c.sum(c3);  
        System.out.println("c + c3 = " + c);           // il complesso -13+0i  
        c = r;  
        System.out.println("c = r; c = " + c);         // qui c è reale  
        // POLIMORFISMO: scatta la toString dei reali --> 21.64  
    }  
}
```

Peccato: si vedono i nomi
delle classi del back-end

Java

~C#

~Kotlin

~Scala

```
r1 + r2 = 21.64  
c1 + c2 = -13.0+2.0i  
c + c3 = -13.0+0.0i  
c = r; c = 21.64
```

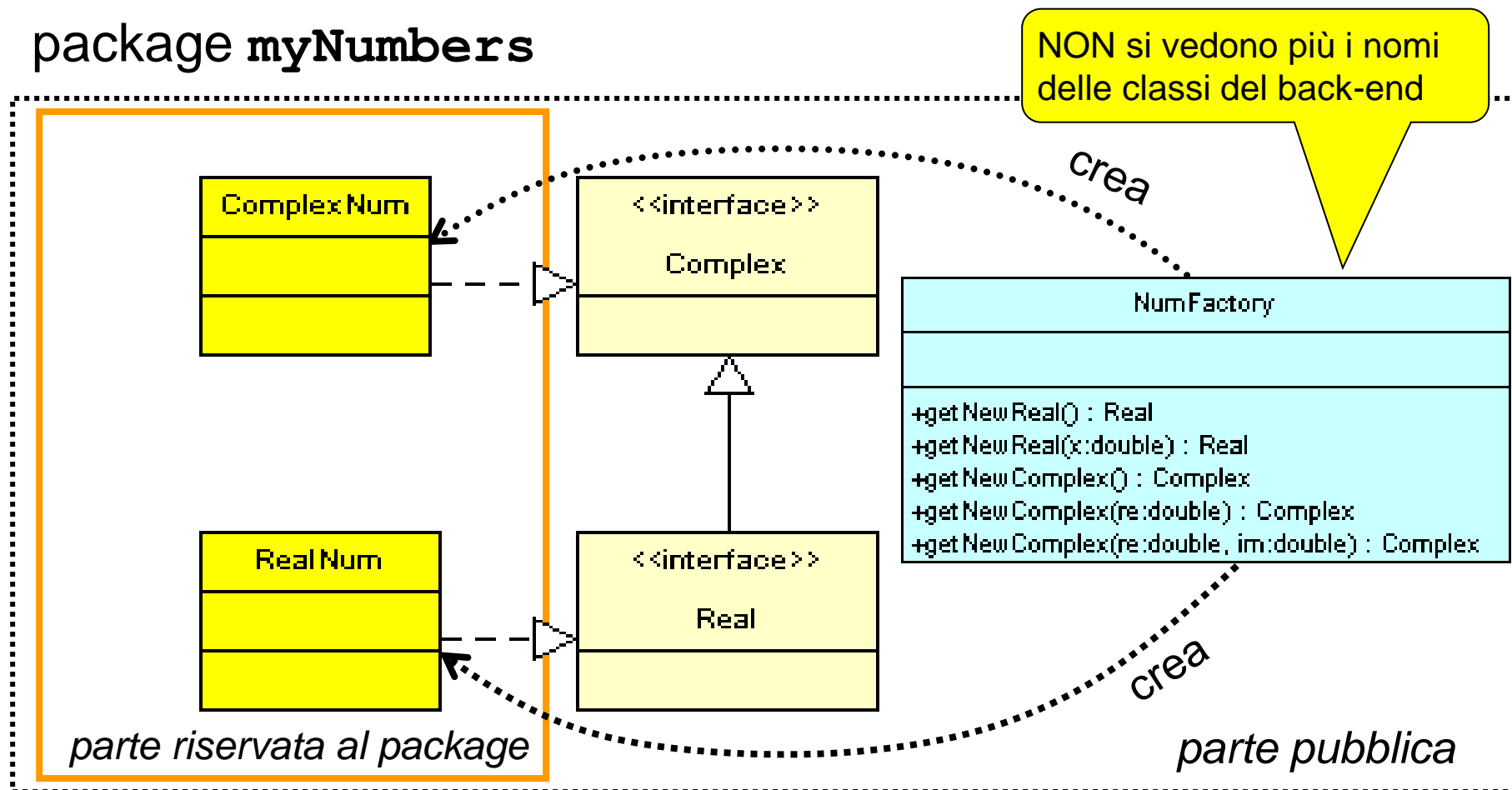


VERSIONE CON FACTORY

- È un peccato che si vedano le classi del back-end!
 - è una dipendenza che vorremmo eliminare
- Per far retrocedere le classi sullo sfondo si può *delegare a una factory (esterna o internalizzata) la fase di creazione*
- Due approcci possibili:
 1. Factory generale **NumFactory** (in due sotto-varianti)
 - crea ogni tipo di numeri (eventualmente, decidendo lei *quali*)
 - Package **myNumbers** contenente factory, interfacce e classi
 2. Factory internalizzata nelle interfacce **Real** e **Complex**
 - ogni mini-factory crea *solo entità del suo tipo*
 - Package **myNumbers** contenente solo interfacce e classi

VERSIONE 1: ARCHITETTURA

package myNumbers





NUMFACTORY

NumFactory è l'unica classe *pubblica* del package

```
package myNumbers;
```

```
public class NumFactory {
```

```
    public static Real getNewReal() {
```

```
        return new RealNum(); }
```

```
    public static Real getNewReal(double x) {
```

```
        return new RealNum(x); }
```

```
    public static Complex getNewComplex() {
```

```
        return new ComplexNum(); }
```

```
    public static Complex getNewComplex(double x) {
```

```
        return new ComplexNum(x); }
```

```
    public static Complex getNewComplex(double x, double y) {
```

```
        return new ComplexNum(x,y); }
```

```
}
```

Bene: non si vedono più i nomi delle classi del back-end

Java

~C#

~Kotlin

~Scala

Ma non benissimo: è sempre l'utente a dover dire «cosa costruire»



VERSIONE 1: MAIN DI TEST

```
import myNumbers.*;
public class MyMain{
    public static void main(String args[]){
        Real    r1 = NumFactory.getNewReal(18.5),
               r2 = NumFactory.getNewReal(3.14);
        Complex c1 = NumFactory.getNewComplex(-16, 0),
               c2 = NumFactory.getNewComplex(3, 2),
               c3 = NumFactory.getNewComplex(0, -2);

        Real    r = r1.sum(r2);    Complex c = c1.sum(c2);
        System.out.println("r1 + r2 = " + r);    // il reale 21.64
        System.out.println("c1 + c2 = " + c);    // il complesso -13+2i
        System.out.println("c1 + c2 -i = " + c.sub(new Complex(0,1)));
        c = c.sum(c3);
        System.out.println("c + c3 = " + c);    // il complesso -13+0i
        c = r;
        System.out.println("c = r; c = " + c);    //
    }
}
```

Java

~C#

~Kotlin

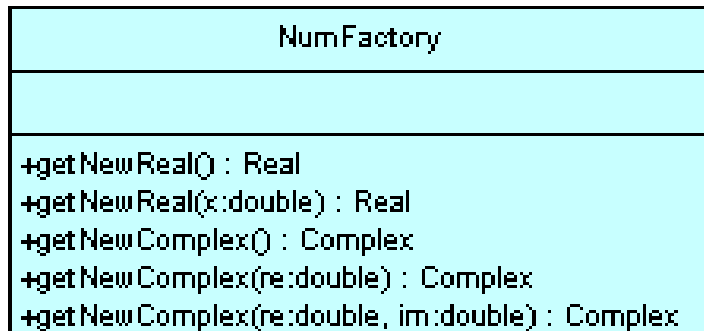
~Scala

Creazione indiretta
tramite factory

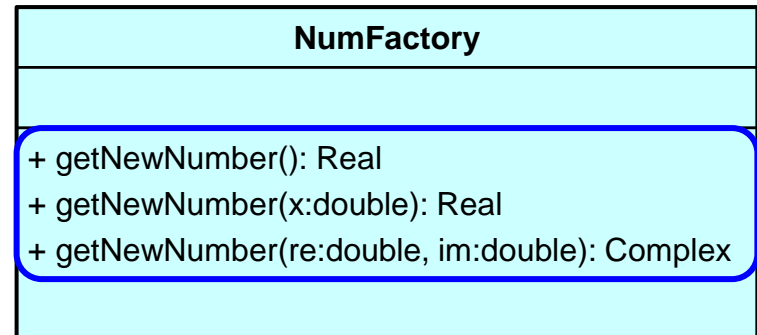
```
r1 + r2 = 21.64
c1 + c2 = -13.0+2.0i
c + c3 = -13.0+0.0i
c = r; c = 21.64
```

VERSIONE 2: REFACTORING

- E se delegassimo a **NumFactory** *la scelta dello specifico tipo di numero da creare?*
 - perché costringere l'utente a farsi carico di questa scelta?
 - perché non può essere la factory a *capire da sola* se serve un **Complex** o un **Real**..? 😊
- Non più:



ma:





NUMFACTORY: v2

NumFactory è l'unica classe *pubblica* del package

```
package myNumbers;
```

```
public class NumFactory {
```

Java

~C#

~Kotlin

~Scala

```
    public static Real getNewNumber() {  
        return new RealNum(); }  
    
```

```
    public static Real getNewNumber(double x) {  
        return new RealNum(x); }  
    
```

```
    public static Complex getNewNumber(double x, double y) {  
        return new ComplexNum(x,y); }  
    
```

```
}
```

Ora non è più l'utente a decidere
cosa creare: è la factory!



MAIN DI TEST: v2

```
import myNumbers.*;
public class MyMain{
    public static void main(String args[]){
        Real    r1 = NumFactory.getNewNumber(18.5),
               r2 = NumFactory.getNewNumber(3.14);
        Complex c1 = NumFactory.getNewNumber(-16),
               c2 = NumFactory.getNewNumber(3, 2),
               c3 = NumFactory.getNewNumber(0, -2);

        Real    r = r1.sum(r2);    Complex c = c1.sum(c2);
        System.out.println("r1 + r2 = " + r);    // il reale 21.64
        System.out.println("c1 + c2 = " + c);    // il complesso -13+2i
        System.out.println("c1 + c2 -i = " + c.sub(new Complex(0,1)));
        c = c.sum(c3);
        System.out.println("c + c3 = " + c);    // il complesso -13+0i
        c = r;
        System.out.println("c = r; c = " + c);    //
    }
}
```

Java

~C#

~Kotlin

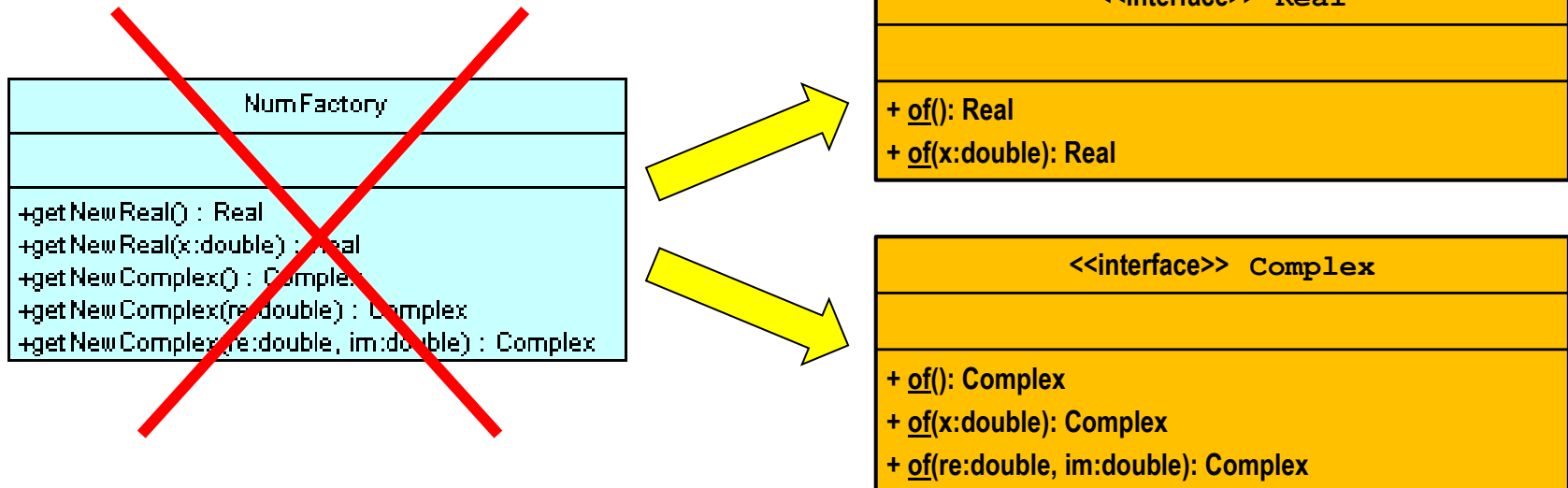
~Scala

Ora è la factory a decidere quali oggetti sia «giusto» creare

```
r1 + r2 = 21.64
c1 + c2 = -13.0+2.0i
c + c3 = -13.0+0.0i
c = r; c = 21.64
```

L'APPETITO VIEN MANGIANDO...

- In alternativa a un'unica **NumFactory** che sceglie e fa ciò che vuole, si può pensare di *internalizzare mini-factory specializzate nelle interfacce **Real** e **Complex***
 - ogni mini-factory crea solo entità del suo tipo





VERSIONE 3: REFACTORING

```
public interface Complex {  
    public double getReal();  
    public double getIm();  
    public double module();  
    public Complex cgt();  
    public Complex divByFactor(double x);  
    public Complex sum(Complex z);  
    public Complex sub(Complex z);  
    public Complex mul(Complex z);  
    public Complex div(Complex z);  
  
    public static Complex getNewComplex() {  
        return new ComplexNum(); }  
    public static Complex getNewComplex(  
        double x) {  
        return new ComplexNum(x); }  
    public static Complex getNewComplex(  
        double x, double y) {  
        return new ComplexNum(x,y); }  
}
```

Java

~C#

~Kotlin

~Scala

```
public interface Real extends Complex {  
    public Real sum(Real x);  
    public Real sub(Real x);  
    public Real mul(Real x);  
    public Real div(Real x);  
    public static Real getNewReal() {  
        return new RealNum(); }  
    public static Real getNewReal(double x) {  
        return new RealNum(x); }  
}
```

REFACTORING – FASE A
Distribuiamo la factory nelle
varie interfacce: *ognuna ospita
la "sua" factory*

OSSERVA: i nomi dei metodi factory sono ora *inutilmente lunghi*, dato che è già chiaro dal nome dell'interfaccia cosa facciano → sono *accorciabili*



VERSIONE 3: REFACTORING

```
public interface Complex {  
    public double getReal();  
    public double getIm();  
    public double module();  
    public Complex cgt();  
    public Complex divByFactor(double x);  
    public Complex sum(Complex z);  
    public Complex sub(Complex z);  
    public Complex mul(Complex z);  
    public Complex div(Complex z);  
}
```

Java

~C#

~Kotlin

~Scala

```
public static Complex of(){  
    return new ComplexNum(); }  
public static Complex of(double x) {  
    return new ComplexNum(x); }  
public static Complex of(double x,  
    double y) {  
    return new ComplexNum(x,y); }  
}
```

```
public interface Real extends Complex {  
    public Real sum(Real x);  
    public Real sub(Real x);  
    public Real mul(Real x);  
    public Real div(Real x);  
    public static Real of() {  
        return new RealNum(); }  
    public static Real of(double x){  
        return new RealNum(x); }  
}
```

REFACTORING – FASE B

Nome accorciato: **of**

VERSIONE 3: MAIN DI TEST

```
import myNumbers.*;
public class MyMain{
    public static void main(String args[]){
```

```
        Real    r1 = Real.of(18.5),
               r2 = Real.of(3.14);
        Complex c1 = Complex.of(-16, 0),
               c2 = Complex.of(3, 2),
               c3 = Complex.of(0, -2);
```

```
        Real    r = r1.sum(r2);      Complex c = c1.sum(c2);
```

```
        System.out.println("r1 + r2 = " + r);           // il reale 21.64
        System.out.println("c1 + c2 = " + c);           // il complesso -13+2i
        System.out.println("c1 + c2 -i = " + c.sub(new Complex(0,1)));
        c = c.sum(c3);
        System.out.println("c + c3 = " + c);           // il complesso -13+0i
        c = r;
        System.out.println("c = r; c = " + c);         //
```

Java	~C#
~Kotlin	~Scala

Factory distribuita ed
embedded nelle interfacce
Nomi metodi corti e uniformi: **of**

```
r1 + r2 = 21.64
c1 + c2 = -13.0+2.0i
c + c3 = -13.0+0.0i
c = r; c = 21.64
```


BILANCIO

Con un'unica factory esterna:

- la fabbrica costruisce ogni possibile tipo di oggetto
- si può delegarle la *scelta generale* di cosa costruire (un complesso? Un reale?) in base ai dati forniti

NumFactory
+getNewReal() : Real +getNewReal(x:double) : Real +getNewComplex() : Complex +getNewComplex(re:double) : Complex +getNewComplex(re:double, im:double) : Complex

Con le factory internalizzate:

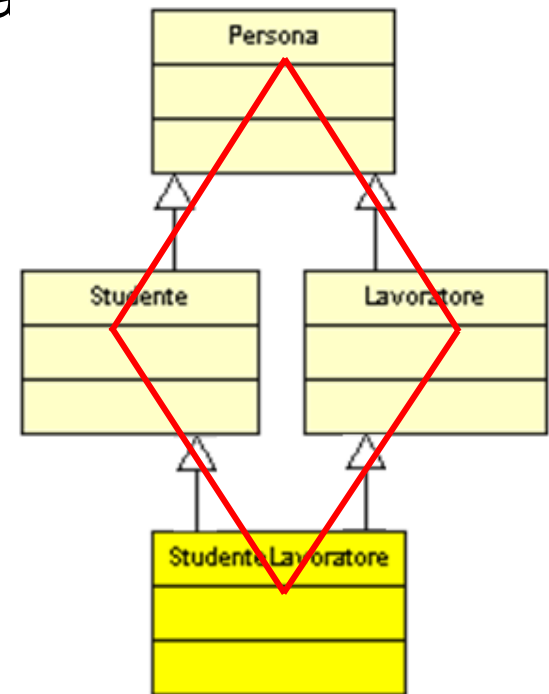
- ogni mini-fabbrica costruisce solo gli oggetti «della sua interfaccia»
- la delega è limitata agli oggetti di quella specifica interfaccia (se l'utente si rivolge a Real, avrà un Real)

<<interface>> Real
+ <u>of</u> () : Real + <u>of</u> (x:double) : Real

<<interface>> Complex
+ <u>of</u> () : Complex + <u>of</u> (x:double) : Complex + <u>of</u> (re:double, im:double) : Complex

DOVE TUTTO EBBE INIZIO: DIAMOND INHERITANCE

- Ritorniamo all'esempio iniziale: esso richiede *ereditarietà multipla* per catturare *intersezione insiemistica*
- Questa configurazione, in cui una classe deriva da 2+ classi con un antenato comune, è tipica e viene chiamata **diamond inheritance**
 - sappiamo che ***fra classi*** genererebbe solo caos, quindi il front-end va strutturato come ***tassonomia di interfacce***
 - il back-end potrà avere:
 - o classi tutte indipendenti fra loro
 - o classi che ereditano parzialmente fra loro, per riusare codice



IMPLEMENTAZIONE: Java (1/6)

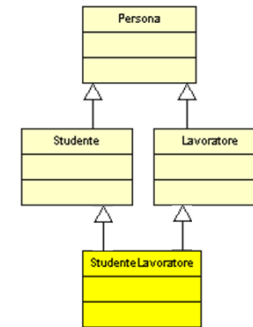
```
interface Persona {  
    public String cognome();  
    public String nome();  
    public LocalDateTime dataDiNascita();  
    public String luogoDiNascita();  
}
```

Java

```
interface Studente extends Persona {  
    public Esame[] esami();  
}
```

```
interface Lavoratore extends Persona {  
    public String impiego();  
    public double stipendio();  
}
```

```
interface StudenteLavoratore extends Studente, Lavoratore {  
    // interfaccia vuota! Definisce il tipo-intersezione  
}
```



```
interface Esame {  
    public String denominazione();  
    public int voto();  
    public String votoAsString();  
    public boolean lode();  
}
```



IMPLEMENTAZIONE: Java (2/6)

Java

```
class LaPersona implements Persona {  
    private String cognome, nome, luogoDiNascita;  
    private LocalDateTime dataDiNascita;  
  
    public LaPersona(String cognome, String nome, LocalDateTime dataDiNascita,  
                     String luogoDiNascita) {  
        this.cognome=cognome; this.nome=nome;  
        this.dataDiNascita=dataDiNascita; this.luogoDiNascita=luogoDiNascita;  
    }  
  
    @Override public String cognome() { return  
    @Override public String nome() { return nome;  
    @Override public String luogoDiNascita() {  
    @Override public LocalDateTime dataDiNascita() { return dataDiNascita ; }  
  
    private DateTimeFormatter formatter =  
        DateTimeFormatter.ofLocalizedDateTime(FormatStyle.SHORT);  
  
    @Override public String toString() { return cognome()+ " " + nome() +  
        " nato a " + luogoDiNascita() + " il " +  
        dataDiNascita().format(formatter); }  
}
```

In Java, l'annotazione @Override si usa anche per classi che implementano metodi dichiarati da interfacce

IMPLEMENTAZIONE: Java (3/6)

```
class LoStudente extends LaPersona implements Studente {
```

```
    private Esame[] esami;
```

```
    public LoStudente(String cognome, String nome, LocalDateTime dataDiNascita,
                      String luogoDiNascita, Esame[] esami) {
        super(cognome, nome, dataDiNascita, luogoDiNascita);
        this.esami = esami;
    }
```

```
    @Override public Esame[] esami() { return esami; }
```

```
    @Override public String toString() { return super.toString() +
        " e ha in carriera in seguenti esami " + Arrays.toString(esami); }
```

```
}
```

Java

Nel back-end, fra classi, LoStudente eredita da LaPersona perché così riutilizza codice. Ma ciò che conta è che implementi Studente!

```
class ILlavoratore extends LaPersona implements Lavoratore {
```

```
    private String impiego; private double stipendio;
```

```
    public ILlavoratore(String cognome, String nome, LocalDateTime dataDiNascita,
                       String luogoDiNascita, String impiego, double stipendio) {
        super(cognome, nome, dataDiNascita, luogoDiNascita);
        this.impiego = impiego; this.stipendio = stipendio; }
    }
```

```
    @Override public String impiego() { return impiego; }
```

```
    @Override public double stipendio() { return stipendio; }
```

```
    @Override public String toString() { return super.toString() + ", di mestiere fa "
        + impiego() + " e guadagna € " + stipendio(); }
```

```
}
```

Java

Analogamente, nel back-end anche ILlavoratore eredita da LaPersona



IMPLEMENTAZIONE: Java (4/6)

```
class LoStudenteLavoratore extends LoStudente implements StudenteLavoratore {  
    private String impiego;  
    private double stipendio;  
    public LoStudenteLavoratore (String cognome, String nome, LocalDateTime dataNasc,  
        String luogoDiNascita, Esame[] esami, String impiego, double stipendio){  
        super(cognome, nome, dataNasc, luogoDiNascita, esami);  
        this.impiego = impiego; this.stipendio = stipendio;  
    }  
    @Override public String toString() { return super.toString()  
        + ", di mestiere fa il " + impiego() + " e guadagna € " + stipendio(); }  
    @Override public String impiego() { return impiego; }  
    @Override public double stipendio() { return stipendio; }  
}
```

Java

Si re-implementa il codice «lato lavoratore» perché la classe LoStudenteLavoratore ha ereditato solo da LoStudente, non dall'altra.

Ma implementa anche l'interfaccia StudenteLavoratore, quindi l'architettura di front-end è salvaguardata!

LoStudenteLavoratore invece non può ereditare da entrambe le classi, perché fra classi l'ereditarietà è solo singola.

Si sceglie la via che fa risparmiare più codice: la parte non ereditata va riscritta



IMPLEMENTAZIONE: Java (5/6)

Java

```
class LEsame implements Esame {
    private String denominazione;
    private int voto;
    private boolean lode;
    public LEsame(String denominazione, int voto, boolean lode){
        this.denominazione=denominazione;
        this.voto=voto;
        this.lode=lode; // solo se voto==30
    }
    public LEsame(String denominazione){ this(denominazione, 0 ,false); }

    @Override public String denominazione() {return denominazione; }
    public int voto() { return voto; }
    public String votoAsString() { return String.valueOf(voto); }
    public boolean lode() { return lode; }

    @Override public String toString() {
        return denominazione() + "\t" + voto() + (lode()?"L":"" ); }
}
```



IMPLEMENTAZIONE: Java (6/6)

Java

```
public static void main(String[] args){
    Persona[] persone = {
        new LaPersona("Rossi", "Mario", LocalDateTime.of(1998,12,25,13,20), "Bologna"),
        new IlLavoratore("Neri", "Giacomo", LocalDateTime.of(1985,2,15,11,50),
            "Bologna", "lo chef", 50000),
        new LoStudiante("Verdi", "Paolo", LocalDateTime.of(2001,3,27,14,14), "Bologna",
            new Esame[]{
                new LEsame("Analisi 1", 30, true), new LEsame("Fondamenti 1", 28, false)}),
        new LoStudianteLavoratore("Bruni", "Elvio", LocalDateTime.of(1999,4,25,4,51),
            "Bologna", new Esame[]{
                new LEsame("Analisi 1", 25, false), new LEsame("Fondamenti 1", 30, true) },
            "il rider part time", 5000)
    };
    for (Persona p : persone) System.out.println(p);
}
```

```
Rossi Mario nato a Bologna il 25/12/98, 13:20
Neri Giacomo nato a Bologna il 15/02/85, 11:50, di mestiere fa lo chef e guadagna € 50000.0
Verdi Paolo nato a Bologna il 27/03/01, 14:14 e ha in carriera in seguenti esami [Analisi 1
    30L, Fondamenti 1 28]
Bruni Elvio nato a Bologna il 25/04/99, 04:51 e ha in carriera in seguenti esami [Analisi 1
    25, Fondamenti 1 30L], di mestiere fa il il rider part time e guadagna € 5000.0
```




SKETCH IMPLEMENTAZIONE: Scala

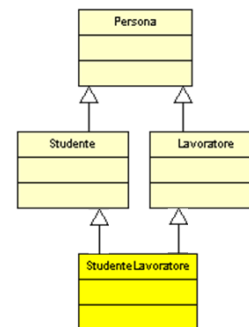
```
trait Persona {  
  def cognome() : String;  
  def nome() : String;  
  def dataDiNascita() : LocalDateTime;  
  def luogoDiNascita() : String;  
}
```

Scala

```
trait Studente extends Persona {  
  def esami() : Array[Esame];  
}
```

```
trait Lavoratore extends Persona {  
  def impiego() : String;  
  def stipendio() : Double;  
}
```

```
trait StudenteLavoratore extends Studente with Lavoratore {  
  // tratto vuoto! Definisce il tipo-intersezione  
}
```



```
trait Esame {  
  def denominazione() : String ;  
  def voto() : Int;  
  def votoAsString() : String;  
  def lode() : Boolean;  
}
```



SKETCH IMPLEMENTAZIONE: Scala

```
class LaPersona( val cognome:String, val nome:String,
                 val dataDiNascita:LocalDateTime,
                 val luogoDiNascita:String) extends Persona {

  private val formatter = DateTimeFormatter.ofLocalizedDateTime(FormatStyle.SHORT);

  override def toString() :String = { return cognome+ " " + nome + " nato a " +
    luogoDiNascita + " il " + dataDiNascita.format(formatter); }

}
```

Scala

```
class LoStudente( cognome:String, nome:String, dataDiNascita:LocalDateTime,
                  luogoDiNascita:String, val esami: Array[Esame])
  extends LaPersona(cognome, nome, dataDiNascita, luogoDiNascita)
  with Studente {

  override def toString() : String = { return super.toString() +
    " e ha in carriera in seguenti esami " +
    Arrays.toString(esami.asInstanceOf[Array[Object]])

}
```

Scala

In Scala, la keyword override si usa solo per i metodi che effettivamente sovrascrivono una precedente implementazione

```
class LoStudenteLavoratore(cognome:String, nome:String, dataDiNascita:LocalDateTime,
                           luogoDiNascita:String, esami: Array[Esame],
                           val impiego:String, val stipendio:Double)

  extends LoStudente(cognome, nome, dataDiNascita, luogoDiNascita, esami)
  with StudenteLavoratore {

  override def toString() : String = { return super.toString() + ", di mestiere fa " +
    impiego + " e guadagna € " + stipendio; }

}
```

Scala

SKETCH IMPLEMENTAZIONE: Kotlin

```
interface Persona {  
    public val cognome() : String;  
    public val nome() : String;  
    public val dataDiNascita() : LocalDateTime;  
    public val luogoDiNascita() : String;  
}
```

Kotlin

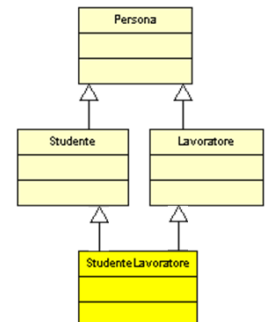
Per variare, sviluppiamo una Implementazione basata su **dati astratti** (**val**) anziché metodi (**def**)
Comodo perché sfrutta il **val** nei costruttori delle classi del back-end!
NB: si sarebbe potuta fare anche in Scala

```
interface Studente : Persona {  
    public val esami : Array<Esame>;  
}
```

```
interface Lavoratore : Persona {  
    public val impiego : String;  
    public val stipendio : Double;  
}
```

```
interface Esame {  
    public val denominazione : String ;  
    public val voto : Int;  
    public val votoAsString : String;  
    public val lode : Boolean;  
}
```

```
interface StudenteLavoratore : Studente , Lavoratore {  
    // interfaccia vuota! Definisce il tipo-intersezione  
}
```





SKETCH IMPLEMENTAZIONE: Kotlin

```
open class LaPersona( override val cognome:String, override val nome:String,
                      override val dataDiNascita:LocalDateTime,
                      override val luogoDiNascita:String) : Persona {

    private val formatter = DateTimeFormatter.ofPattern("dd/MM/yyyy", Locale.ITALY);
    public override fun toString() : String { return super.toString() + nome + " nato a " +
        luogoDiNascita + " il " + dataDiNascita.format(formatter); }

}
```

I *dati concreti* sovrascrivono
(**override val**) le dichiarazioni
astratte dell'interfaccia

Kotlin

```
open class LoStudente( cognome:String, nome:String, dataDiNascita:LocalDateTime,
                       luogoDiNascita:String, override val esami: Array<Esame>)
    : LaPersona(cognome, nome, dataDiNascita, luogoDiNascita), Studente {

    public override fun toString() : String { return super.toString() +
        " e ha in carriera in seguenti esami " + java.util.Arrays.toString(esami); }

}
```

Kotlin

```
open class LoStudenteLavoratore(cognome:String, nome:String, dataDiNascita:LocalDateTime,
                                luogoDiNascita:String, esami: Array<Esame>,
                                override val impiego:String,
                                override val stipendio:Double)
    : LoStudente(cognome, nome, dataDiNascita, luogoDiNascita, esami), StudenteLavoratore {

    public override fun toString() : String { return super.toString() + ", di mestiere fa "
        + impiego + " e guadagna € " + stipendio; }

}
```

Kotlin

Interfacce standard Comparable & Comparator



INTERFACCE STANDARD

Esistono molte interfacce standard, destinate a esprimere:

- *abilità, capacità di fare qualcosa*:
per questo hanno spesso nomi che terminano in "-abile"
 - *confrontabile, serializzabile, clonabile, stampabile...*
 - tipicamente *dichiarano un solo metodo*, il cui nome esprime l'abilità
 - `compareTo` per `Comparable`, `print` per `Printable`, etc.
 - a volte, però, non ne dichiarano affatto: *sono vuote!*
Sembra un assurdo, ma non lo è: sono **INTERFACCE MARKER**
 - «tag» usati per *esplicitare* che una proprietà è *già presente*...
 - ... sfruttando il compilatore per i relativi controlli incrociati ☺
- *concetti astratti*
 - *azione, ascoltatore di eventi, servizio di stampa, ...*



IL TEMA DEL CONFRONTO

- Tutti i linguaggi offrono fornite funzioni di utilità o librerie per effettuare *ordinamenti* e *ricerche* in strutture dati
- Obiettivo: implementarle in modo generico e configurabile
- A tal fine, il punto chiave è la capacità di *confrontare* due elementi, di qualunque tipo essi siano
- La «**confrontabilità**» è una *tipica proprietà* che un oggetto potrebbe possedere → situazione ideale da rappresentare con un'opportuna *interfaccia*
 - in Java, Kotlin: **Comparable**
 - in C#: **IComparable**
 - in Scala: **Ordered**



CONFRONTABILITÀ

- Dicendo «**confrontabilità**» si sottintende solitamente la presenza di un «**criterio naturale**» (*natural ordering*) di confronto, *intrinseco* rispetto alla natura dell'oggetto
 - per stringhe appare «naturale» confrontarle *alfabeticamente*
 - per numeri appare «naturale» confrontarli *secondo il loro valore*
 - per **Counter** e **Frazione** sarebbe «naturale» confrontarli *secondo il loro valore, coerentemente con quanto fatto nella equals*
- Non sempre tale criterio esiste: le cose non vanno forzate!
 - per Persona quale sarebbe il criterio «naturale» ..?
(Per cognome? Per nome? Per età? Un mix di questi..?)



CONFRONTABILITÀ & ALGORITMI

- Strutture dati di oggetti *confrontabili* possono essere **ordinate** in modo altrettanto «naturale» tramite algoritmi generali e polimorfi
 - array: metodo statico `Arrays.sort`
 - liste: metodo `Collections.sort`
- In tali strutture dati ordinate si possono altresì effettuare **ricerche** efficienti tramite algoritmi di ricerca binaria
 - array: metodo statico `Arrays.binarySearch`
 - liste: metodo `Collections.binarySearch`
- ... e molto altro



UN'INTERFACCIA PER LA CONFRONTABILITÀ

- Ipotesi: una classe **T** i cui oggetti sono dotati della capacità di «confrontarsi» *implementa un'apposita interfaccia tipizzata*
 - tipizzata perché deve essere specializzata sul tipo **T** della classe
 - Java: **Comparable<T>** Java
 - C#: **IComparable<T>** C#
 - Scala: **Ordered[T]** Scala
 - Kotlin: **Comparable<T>** Kotlin
- Essa dichiara un solo metodo, che esprime il confronto:
 - Java: **public int compareTo(T that) ;** Java
 - C#: **public int CompareTo(T that) ;** C#
 - Scala: **def compare(other : T) : Int;** Scala
 - Kotlin: **public fun compareTo(other : T) : Int;** Kotlin



UN'INTERFACCIA PER LA CONFRONTABILITÀ

- Il metodo che esprime il confronto:

- Java: `public int compareTo(T that);`

Java

- C#: `public int CompareTo(T that);`

C#

- Scala: `def compare(other : T) : Int;`

Scala

- Kotlin: `public fun compareTo(other : T):Int;`

Kotlin

- ha la seguente semantica:

- `a.compareTo(b)` restituisce -1 se $a < b$, +1 se $a > b$, 0 se uguali
- NB: la funzione deve garantire un comportamento coerente con `equals` & `hashCode` nel caso dell'uguaglianza



ESEMPIO: UN Counter CONFRONTABILE

- Ad esempio, per essere «confrontabile», un **Counter** dovrà implementare l'interfaccia:

– in Java o Kotlin `Comparable<Counter>`

Java

Kotlin

– in C# `IComparable<Counter>`

C#

– in Scala `Ordered[Counter]`

Scala

- A tal fine dovrà implementare al suo interno il metodo:

– Java: `public int compareTo(Counter that) {...}`

Java

– C#: `public int CompareTo(Counter that) {...}`

C#

– Scala: `def compareTo(other : Counter) : Int = {...}`

Scala

– Kotlin: `public fun compareTo(other : Counter):Int {...}`

Kotlin



UN COUNTER Comparable

```
public class Counter
    implements Comparable<Counter> {

    ...

    public int compareTo(Counter that) {
        if (val < that.val) return -1;
        if (val > that.val) return +1;
        /* else */ return 0;
    }
}
```

Java

interfaccia tipizzata

Esprimiamo qui il criterio «naturale» di confronto

NB: le versioni in C#, Scala, Kotlin sono analoghe

C#

Scala

Kotlin



UN COUNTER Comparable

```
public class Counter implements Comparable<Counter> {  
    protected int val;  
    public Counter() { val = 1; }  
    public Counter(int v) { val = v; }  
    public void reset() { val = 0; }  
    public void inc() { val++; }  
    public int getValue() { return val; }  
    //  
    public String toString() { return "Counter di valore " + val; }  
    //  
    public boolean equals(Object obj){  
        if(!(obj instanceof Counter)) return false;  
        return this.val == ((Counter)obj).val;  
    }  
    public int hashCode() { return val * 31; }  
    //  
    public int compareTo(Counter that) {  
        if (val < that.val) return -1;  
        if (val > that.val) return +1;  
        /* else */ return 0;  
    }  
}
```

Java

```
using System;  
  
public class Counter : IComparable<Counter> {  
    protected int val;  
    public Counter() { val = 1; }  
    public Counter(int v) { val = v; }  
    public void reset() { val = 0; }  
    public void inc() { val++; }  
    public int getValue() { return val; }  
    //  
    public override string ToString() { return "Counter di valore " + val; }  
    //  
    public override int GetHashCode() { return val * 31; }  
    public override bool Equals(object obj){  
        if(!(obj is Counter)) return false;  
        return this.val == ((Counter)obj).val;  
    }  
    //  
    public int CompareTo(Counter other) {  
        if (val < other.val) return -1;  
        if (val > other.val) return +1;  
        /* else */ return 0;  
    }  
}
```

C#

```
class Counter(private var value : Int = 1 ) extends Ordered[Counter] {  
    def reset() : Unit = { value = 0; }  
    def inc() : Unit = { value +=1; }  
    def getValue() : Int = { return value; }  
    protected def setValue(v:Int) : Unit = { value = v; }  
  
    override def equals(other : Any) : Boolean = {  
        return if(other.isInstanceOf[Counter]) value==other.asInstanceOf[Counter].value  
        else false;  
    }  
  
    override def hashCode() : Int = {  
        return value * 31;  
    }  
  
    override def toString() : String = { return "Counter di valore " + value; }  
  
    // non si chiama più compareTo/1, ma compare/1  
    override def compare(other: Counter) : Int = {  
        if (value < other.value) return -1;  
        if (value > other.value) return +1;  
        /* else */ return 0;  
    }  
}
```

Scala

```
open public class Counter(value : Int = 1 ) : Comparable<Counter> {  
    private var value : Int  
    init{ this.value = value; }  
    public fun reset() : Unit { value = 0; }  
    public fun inc() : Unit { value++; }  
    public fun getValue() : Int { return value; }  
    protected fun setValue(v:Int) : Unit { value = v; }  
  
    public override fun equals(other : Any?) : Boolean {  
        return if(other is Counter) value==other.value; else false;  
    }  
  
    public override fun hashCode() : Int {  
        return value * 31;  
    }  
  
    public override fun toString() : String { return "Counter di valore " + value; }  
  
    public override fun compareTo(other: Counter) : Int {  
        if (value < other.value) return -1;  
        if (value > other.value) return +1;  
        /* else */ return 0;  
    }  
}
```

Kotlin



UN PICCOLO MAIN DI PROVA

```
public class TestComparable {  
  
    public static void main(String args[]) {  
        Counter c1 = new Counter(10);  
        Counter c2 = new Counter(10);  
        //  
        System.out.println("c1 vs c2: " + c1.compareTo(c2)); // dà 0  
        System.out.println("c1 == c2: " + c1.equals(c2));    // dà true  
        c1.inc();  
        System.out.println("c1 vs c2: " + c1.compareTo(c2)); // dà +1  
        System.out.println("c2 vs c1: " + c2.compareTo(c1)); // dà -1  
        System.out.println("c1 == c2: " + c1.equals(c2));    // dà false  
    }  
}
```

Java

```
----- Java Run -----  
c1 vs c2: 0  
c1 == c2: true  
c1 vs c2: 1  
c2 vs c1: -1
```

```
object Counter {  
    def main(args: Array[String]) = {  
        val c1 = new Counter(10);  
        val c2 = new Counter(10);  
        println("c1 vs c2: " + c1.compare(c2)); // dà 0  
        println("c1 == c2: " + c1.equals(c2)); // dà true  
        c1.inc();  
        println("c1 vs c2: " + c1.compare(c2)); // dà +1  
        println("c2 vs c1: " + c2.compare(c1)); // dà -1  
        println("c1 == c2: " + c1.equals(c2)); // dà false  
    }  
}
```

```
----- Scala run -----  
c1 vs c2: 0  
c1 == c2: true  
c1 vs c2: 1  
c2 vs c1: -1  
c1 == c2: false
```

Scala

```
class TestCounterComparable {  
    public static void Main(string[] args) {  
        Counter c1 = new Counter(10);  
        Counter c2 = new Counter(10);
```

```
        WriteLine("c1 vs c2: " + c1.CompareTo(c2)); // dà 0  
        WriteLine("c1 == c2: " + c1.Equals(c2));    // dà true  
        ;  
        WriteLine("c1 vs c2: " + c1.CompareTo(c2)); // dà +1  
        WriteLine("c2 vs c1: " + c2.CompareTo(c1)); // dà -1  
        WriteLine("c1 == c2: " + c1.Equals(c2));    // dà false
```

```
----- C# Run -----  
c1 vs c2: 0  
c1 == c2: True  
c1 vs c2: 1  
c2 vs c1: -1  
c1 == c2: False
```

C#

```
fun main(args: Array<String>){  
    val c1 = Counter(10);  
    val c2 = Counter(10);  
    println("c1 vs c2: " + c1.compareTo(c2)); // dà 0  
    println("c1 == c2: " + c1.equals(c2));    // dà true  
    c1.inc();  
    println("c1 vs c2: " + c1.compareTo(c2)); // dà +1  
    println("c2 vs c1: " + c2.compareTo(c1)); // dà -1  
    println("c1 == c2: " + c1.equals(c2));    // dà false  
}
```

```
----- Kotlin run -----  
c1 vs c2: 0  
c1 == c2: true  
c1 vs c2: 1  
c2 vs c1: -1  
c1 == c2: false
```

Kotlin



ORDINARE UN ARRAY DI COUNTER

```
public class Test {  
    public static void main(String args[]) {  
        Counter[] myCounterArray = {  
            new Counter(110), new Counter(100),  
            new Counter(30), new Counter(50) };  
        for(Counter c: myCounterArray) System.out.println(c);  
        Arrays.sort(myCounterArray);  
        for(Counter c: myCounterArray) System.out.println(c);  
    }  
}
```

Java

Internamente, `sort` invoca
`v[i].compareTo(v[j])`
per fare i necessari confronti.

```
Counter di valore 110  
Counter di valore 100  
Counter di valore 30  
Counter di valore 50  
Counter di valore 30  
Counter di valore 50  
Counter di valore 100  
Counter di valore 110
```




ORDINARE UN ARRAY DI COUNTER

```
object SortArrayOfOrdered {  
  
  def main(args: Array[String]) = {  
    val myCounterArray : Array[Counter] = Array(  
      new Counter(110), new Counter(100), new Counter(30), new Counter(50) );  
  
    for(c <- myCounterArray) println(c);  
  
    scala.util.Sorting.quickSort(myCounterArray); // the old way  
    // alternatively you can use the sorted() method, which produces a new array  
    // without altering the original one:  
    // for(c <- myCounterArray.sorted) println(c);  
  
    for(c <- myCounterArray) println(c);  
  
  }  
}
```

Scala

```
class SortArrayOfComparable {  
  public static void Main(string[] args) {  
    Counter[] myCounterArray = {  
      new Counter(110), new Counter(100),  
      new Counter(30), new Counter(50) };  
  
    foreach(Counter c in myCounterArray) Console.WriteLine(c);  
  
    Array.Sort(myCounterArray);  
  
    foreach(Counter c in myCounterArray) Console.WriteLine(c);  
  
  }  
}
```

C#

```
fun main(args: Array<String>) : Unit {  
  val myCounterArray : Array<Counter> = arrayOf(  
    Counter(110), Counter(100), Counter(30), Counter(50) );  
  
  for(c in myCounterArray) println(c);  
  
  myCounterArray.sort();  
  
  for(c in myCounterArray) println(c);  
}
```

Kotlin



UN ALTRO ESEMPIO: Frazioni confrontabili

- Analogamente, per essere «confrontabile», una **Frazione** dovrà implementare l'interfaccia:
 - in Java o Kotlin `Comparable<Frazione>` Java Kotlin
 - in C# `IComparable<Frazione>` C#
 - in Scala `Ordered[Frazione]` Scala
- il cui metodo di confronto dovrà incapsulare un criterio molto simile a quello di «equivalenza» già incapsulato in `equals`

Please do it by yourself 😊

CONFRONTABILITÀ & ALGORITMI

- La confrontabilità secondo un «*criterio naturale*» non cattura però tutte le esigenze, perché:
 - può non esistere un criterio «naturale»
 - anche se esiste, può accadere che non se ne sia tenuto conto in fase di progetto di quella classe
 - può comunque sorgere l'esigenza di confrontare elementi, in date situazioni, *con un criterio diverso*
- A questo fine si introduce il concetto di **comparatore**
 - un componente «terzo» che *incapsula un arbitrario criterio di confronto* fra elementi di un certo tipo
 - analogia hardware: un chip «pluggabile», che porta in dote una sua capacità di «confrontare cose»



CONFRONTABILITÀ TRAMITE COMPARATORI

- Un **comparatore** è l'analogo software di un chip hardware: incorpora una relazione d'ordine
 - ordinamento di **Counter** modulo quel-che-vogliamo...
 - ... ordinamento di stringhe per lunghezza, o ignorando gli spazi...
 - ... e ogni altra necessità *diversa dall' «ordinamento naturale»*
- È un componente «pluggabile»: dovunque venga inserito, «porta in dote» il criterio di confronto che incapsula
 - in particolare, quindi, basta passare a **sort** un certo comparatore per ordinare la struttura dati secondo il criterio del comparatore
 - *indipendentemente* dal fatto che la struttura avesse, o meno, un proprio criterio «naturale» espresso da **Comparable**





CONFRONTABILITÀ TRAMITE COMPARATORI: INTERFACCIA

- Un **comparatore per T** è una classe che incorpora un metodo che confronta *due argomenti* di tipo T
 - a differenza di **Comparable**, non esprime un metodo dell'oggetto da confrontare: il comparatore è un *ente terzo*, che confronta due oggetti
 - Java, Kotlin: **Comparator<T>**

Java

Kotlin
 - C#: **IComparer<T>**

C#
 - Scala: **Ordering[T]**

Scala
- Dichiarare un solo metodo, che cattura il criterio di confronto:
 - Java: **public int compare(T a, T b);**

Java
 - C#: **public int Compare(T a, T b);**





C#
 - Scala: **def compare(a:T, b:T) : Int;**

Scala
 - Kotlin: **public fun compare(a:T, b:T) : Int**

Kotlin



CONFRONTABILITÀ TRAMITE COMPARATORI: INTERFACCIA

- Il metodo che esprime il confronto:
 - Java: `public int compare(T a, T b);` 
 - C#: `public int Compare(T a, T b);` 
 - Scala: `def compare(a:T, b:T) : Int;` 
 - Kotlin: `public fun compare(a:T, b:T) : Int;` 
- ha una seguente semantica analoga alla precedente:
 - `cmp.compare(a,b)` restituisce -1 se $a < b$, +1 se $a > b$, 0 se uguali
 - `cmp` è una istanza di comparatore che incapsula il criterio di confronto desiderato
- Si possono definire *molti comparatori diversi* – idealmente, uno per ogni criterio di confronto che serve 😊



ESEMPIO: UN COMPARATORE PER Counter

- Ad esempio, un comparatore per **Counter** dovrà implementare l'interfaccia:

– in Java o Kotlin `Comparator<Counter>`

Java

Kotlin

– in C# `IComparer<Counter>`

C#

– in Scala `Ordering[Counter]`

Scala

- A tal fine dovrà implementare al suo interno il metodo:

– Java: `public int compare(Counter a, Counter b) {...}`

Java

– C#: `public int Compare(Counter a, Counter b) {...}`

C#

– Scala: `def compare(a: Counter, b: Counter): Int = {...}`

Scala

– Kotlin: `public fun compare(a: Counter, b: Counter): Int {...}`

Kotlin



ESEMPIO: UN COMPARATORE PER Counter

- Supponiamo di voler ordinare un array di **Counter** in base al loro valore *modulo 24*
- Non possiamo né vogliamo modificare l'ordinamento naturale dei **Counter**, che è crescente per valore (e va bene per tanti altri usi): *vogliamo risolvere una specifica esigenza*
- Implementiamo un comparatore che inglobi questo criterio:

```
public class MyComp implements Comparator<Counter> {  
    public int compare(Counter x, Counter y) {  
        if (x.val%24 < y.val%24) return -1;  
        if (x.val%24 > y.val%24) return +1;  
        /* else */ return 0;  
    }  
}
```

Java



ESEMPIO: UN COMPARATORE PER Counter

```
class MyComp : IComparer<Counter> {  
    public int Compare(Counter x, Counter y) {  
        if (x.GetValue()%24 < y.GetValue()%24) return -1;  
        if (x.GetValue()%24 > y.GetValue()%24) return +1;  
        /* else */ return 0;  
    }  
}
```

C#

```
class MyComp extends Ordering[Counter] {  
    override def compare(a: Counter, b: Counter) : Int = {  
        if (a.GetValue()%24 < b.GetValue()%24) return -1;  
        if (a.GetValue()%24 > b.GetValue()%24) return +1;  
        /* else */ return 0;  
    }  
}
```

Scala

```
class MyComp : Comparator<Counter> {  
    public override fun compare(a: Counter, b: Counter) : Int {  
        if (a.GetValue()%24 < b.GetValue()%24) return -1;  
        if (a.GetValue()%24 > b.GetValue()%24) return +1;  
        /* else */ return 0;  
    }  
}
```

Kotlin



ORDINARE UN ARRAY DI COUNTER CON UN COMPARATORE AD HOC

```
public class Test {  
    public static void main(String args[]) {  
        Counter[] myCounterArray = {  
            new Counter(110), new Counter(100),  
            new Counter(30), new Counter(50) };  
        for(Counter c: myCounterArray) System.out.println(c);  
        Arrays.sort(myCounterArray, new MyComp() );  
        for(Counter c: myCounterArray) System.out.println(c);  
    }  
}
```

Java

~C#

~Scala

~Kotlin

Usa questo comparatore invece
dell'ordinamento *naturale* dei Counter

Perché

$50 \% 24 = 2$	$30 \% 24 = 6$
$100 \% 24 = 4$	$110 \% 24 = 14$

```
Counter di valore 110  
Counter di valore 100  
Counter di valore 30  
Counter di valore 50  
Counter di valore 50  
Counter di valore 100  
Counter di valore 30  
Counter di valore 110
```

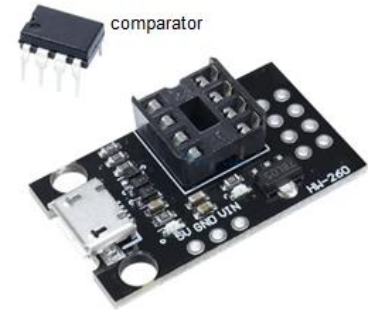


UN ALTRO ESEMPIO: COMPARATORI DI PERSONE

- La classe **Persona** sviluppata tempo addietro
 - proprietà: cognome, nome, etànon incorporava alcun criterio di confronto
- Si potrebbe renderla confrontabile implementando **Comparable**, ma.. **esiste un criterio davvero «naturale»?**
- In realtà, no:
 - ci sono pari ragioni per ordinarle per cognome come per nome o età, come pure con una qualunque combinazione di questi, o altri
 - *non c'è un vero criterio «naturale»* come per le frazioni o le stringhe!
- Imporre un criterio «built-in» tramite **Comparable** sarebbe quindi una forzatura: molto meglio dei **comparatori esterni**

UN ALTRO ESEMPIO: COMPARATORI DI PERSONE

- Potrebbero servirci comparatori
 - per cognome
 - per nome
 - per età
- Facile: basta fare tre classi-**Comparator**, ognuna con una diversa implementazione del metodo **compare**
 - poi basterà passare al metodo **sort** il «giusto» comparatore 😊



```
Arrays.sort(persone, new CognomeComparator() );  
System.out.println(String.join(", ", persone));  
  
Arrays.sort(persone, new NomeComparator() );  
System.out.println(String.join(", ", persone));  
  
Arrays.sort(persone, new EtaComparator() );  
System.out.println(String.join(", ", persone));
```

Java



UN ALTRO ESEMPIO: COMPARATORI DI PERSONE

```
class CognomeComparator implements Comparator<Persona> {  
    public int compare(Persona p1, Persona p2) {  
        return p1.getCognome().compareTo(p2.getCognome());  
    }  
}
```

Java

Sfrutta il fatto che le stringhe siano Comparable
e abbiano quindi una loro compareTo

```
class NomeComparator implements Comparator<Persona> {  
    public int compare(Persona p1, Persona p2) {  
        return p1.getNome().compareTo(p2.getNome());  
    }  
}
```

Sfrutta il fatto che le stringhe siano Comparable
e abbiano quindi una loro compareTo

```
class EtaComparator implements Comparator<Persona> {  
    public int compare(Persona p1, Persona p2) {  
        return Integer.compare(p1.getEtà(), p2.getEtà());  
    }  
}
```

Sfrutta la funzione statica di confronto
per int della classe Integer



UN ALTRO ESEMPIO: ORDINARE UN ARRAY DI PERSONE

- Ordiniamo l'array nei tre modi e stampiamo:

```
System.out.println(String.join(", ", persone));  
  
Arrays.sort(persone, new CognomeComparator() );  
System.out.println(String.join(", ", persone));  
  
Arrays.sort(persone, new NomeComparator() );  
System.out.println(String.join(", ", persone));  
  
Arrays.sort(persone, new EtaComparator() );  
System.out.println(String.join(", ", persone));
```

Metodo furbo per
stampare un array

Java

```
----- unsorted -----  
John Doe ha 25 anni  
Jane Doe ha 45 anni  
Anne Bee ha 31 anni  
Jane Doe ha 22 anni  
----- per cognome -----  
Anne Bee ha 31 anni  
John Doe ha 25 anni  
Jane Doe ha 45 anni  
Jane Doe ha 22 anni
```

```
----- per nome -----  
Anne Bee ha 31 anni  
Jane Doe ha 45 anni  
Jane Doe ha 22 anni  
John Doe ha 25 anni  
----- per età -----  
Jane Doe ha 22 anni  
John Doe ha 25 anni  
Anne Bee ha 31 anni  
Jane Doe ha 45 anni
```



UN ALTRO ESEMPIO: ORDINARE UN ARRAY DI PERSONE

- Ovviamente, i comparatori si possono definire anche come oggetti a se stanti, eventualmente riutilizzabili più volte:

```
Comparator<Persona> cmp1 = new CognomeComparator();  
Comparator<Persona> cmp2 = new NomeComparator();  
Comparator<Persona> cmp3 = new EtaComparator();
```

Java

```
print("unsorted", persone);  
Arrays.sort(persone, cmp1);  
System.out.println(String.join(", ", persone));  
Arrays.sort(persone, cmp2);  
System.out.println(String.join(", ", persone));  
Arrays.sort(persone, cmp3);  
System.out.println(String.join(", ", persone));
```

Riusabili più e più volte
nel programma: tanto,
non hanno stato!



UN ALTRO ESEMPIO: ORDINARE UN ARRAY DI PERSONE

- Questo approccio funziona, ma:
 - porta a definire *una classe per ogni criterio*, per poi *istanziarla una e una sola volta*
 - se la singola istanza di ogni comparatore è usata una sola volta, appoggiarla a un riferimento esplicito (`cmp1`, `cmp2`, `com3`) è inutile: si può fare la `new` direttamente *inline*

Molto chiaro

MA anche, secondo alcuni, *prolisso e verboso*

```
Comparator<Persona> cmp1 = new CognomeComparator();  
Comparator<Persona> cmp2 = new NomeComparator();  
Comparator<Persona> cmp3 = new EtaComparator();  
  
Arrays.sort(persone, cmp1);  
print("per solo cognome", persone);  
  
Arrays.sort(persone, cmp2);  
print("per solo nome", persone);  
  
Arrays.sort(persone, cmp3);  
print("per sola età", persone);
```

Java



IMPLEMENTAZIONE «INLINE» CON CLASSI ANONIME

- Se un comparatore serve *una sola volta* e ne viene costruita *una sola istanza*, un'alternativa è *evitare di scrivere la classe esplicitamente*, creando un'istanza di *CLASSE ANONIMA*
 - si evita di scrivere direttamente una nuova classe perché *se ne delega la creazione al compilatore, «sotto banco»*
 - *si collassano tre cose insieme:*
 - la definizione della classe-comparatore
 - la definizione al suo interno della `compare` appropriata
 - la costruzione dell' istanza-singleton del comparatore

Vediamo come



IMPLEMENTAZIONE «INLINE» CON CLASSI ANONIME

- Finora, per usare un nostro comparatore, abbiamo dovuto:
 - definire una classe-comparatore specifica

```
class CognomeComparator implements Comparator<Persona> { Java  
    ...  
}
```

- scriverci dentro la **compare** appropriata

```
class CognomeComparator implements Comparator<Persona> { Java  
    public int compare(Persona p1, Persona p2) {  
        return p1.getCognome().compareTo(p2.getCognome());  
    }  
}
```

- creare (e passare a **sort**) l'istanza-singleton di *quel* comparatore:

```
Arrays.sort(persone, new CognomeComparator()); Java
```



IMPLEMENTAZIONE «INLINE» CON CLASSI ANONIME

- Con una **classe anonima**, i tre passaggi collassano:
 - NON si inventa più il nome della classe che implementa `Comparator`, perché *lo farà il compilatore al posto nostro*

```
class CognomeComparator implements Comparator<Persona> {  
    public int compare(Persona p1, Persona p2) {  
        return p1.getCognome().compareTo(p2.getCognome());  
    }  
}
```

Java

- di conseguenza, anziché costruire il comparatore «per nome»:

```
Arrays.sort(persone, new CognomeComparator() );
```

Java

- si scrive direttamente **la new di «tutto quel che resta», inline**:

```
Arrays.sort(persone, new Comparator() { ... } );
```

Java

Codice di **compare**



IMPLEMENTAZIONE «INLINE» CON CLASSI ANONIME

- Risultato: un'implementazione *anonima*!

```
Arrays.sort(persone, new Comparator<Persona>() {  
    public int compare(Persona p1, Persona p2) {  
        return p1.getCognome().compareTo(p2.getCognome());  
    }  
});
```

Java

- ATTENZIONE: **non stiamo tentando di "istanziare un'interfaccia"** (cosa vietata e comunque assurda, è un guscio vuoto), stiamo semplicemente ***lasciando scegliere al compilatore il nome della classe*** che implementa tale interfaccia (classe che comunque *deve* esistere!)



ORDINARE UN ARRAY DI PERSONE CON CLASSE ANONIMA

```
public class TestPersone {  
    public static void main(String args[]) {  
        ...  
        Arrays.sort(persone, new CognomeComparator() );  
    }  
}
```

Java

PRIMA

```
public class TestPersone {  
    public static void main(String args[]) {  
        ...  
        Arrays.sort(persone,  
            new Comparator<Persona>() {  
                public int compare(Persona p1, Persona p2) { ... }  
            }  
        );  
    }  
}
```

DOPO

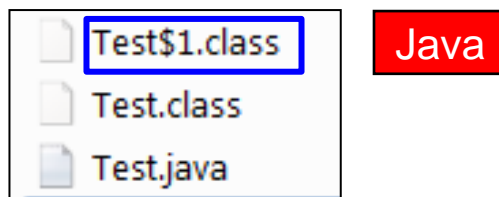
Questo box era
CognomeComparator

Non istanzia l'interfaccia: istanzia la classe
anonima creata per noi dal compilatore
«dietro le quinte»

Implementazione inline tramite
CLASSE ANONIMA

E INFATTI, DIETRO LE QUINTE...

- Dietro le quinte, il compilatore genera effettivamente una nuova classe, con nome uguale alla **classe ospite+\$1**
 - basta guardare i file prodotti!
 - se la classe esterna si chiama **Test**, i due file **.class** prodotti si chiamano **Test** e **Test\$1**



- Conclusione: fu vera gloria?
 - di per sé, le classi anonime non sembrano un gran guadagno...
 - in effetti, il loro vero valore sta nel passo logico successivo: le *lambda expression* (che vedremo presto)

COMPARATORI ANONIMI NEGLI ALTRI LINGUAGGI

- Le classi anonime esistono anche in C#, Scala, Kotlin
- In Scala e Kotlin, la nozione di *oggetto singleton* incorporata nel linguaggio (keyword `object`) snellisce anche la sintassi
 - in Scala, anziché scrivere `new Ordering[Tipo]{...}`:

```
scala.util.Sorting.quickSort(myCounterArray) {  
  new Ordering[Counter] {  
    def compare(a: Counter, b: Counter) :  
  }}
```

- si può passare o definire al volo un *object singleton*: →

```
scala.util.Sorting.quickSort(myCounterArray) (MyComp)  
// using an object, no need to instantiate a new Comparator  
// otherwise a class could be used, with the new keyword here  
  
scala.util.Sorting.quickSort(myCounterArray) (MyBetterComp)  
// using an object, no need to instantiate a new Comparator  
// otherwise a class could be used, with the new keyword here  
  
}  
}  
  
object MyComp extends Ordering[Counter] {  
  override def compare(a: Counter, b: Counter) : Int = {  
    if (a.getValue() % 24 < b.getValue() % 24) return -1;  
    if (a.getValue() % 24 > b.getValue() % 24) return +1;  
    /* else */ return 0;  
  }  
}  
  
object MyBetterComp extends Ordering[Counter] {  
  override def compare(a: Counter, b: Counter) : Int =  
    a.getValue() % 24 compare b.getValue() % 24;  
}
```

Scala

COMPARATORI ANONIMI NEGLI ALTRI LINGUAGGI

- Le classi anonime esistono anche in C#, Scala, Kotlin
- In Scala e Kotlin, la nozione di *oggetto singleton* incorporata nel linguaggio (keyword `object`) snellisce anche la sintassi
 - analogamente, in Kotlin conviene *creare inline il singleton stesso* con la notazione `object: Comparator<Tipo> {...}`

```
myCounterArray.sortWith( object: Comparator<Counter> {  
    override fun compare(a: Counter, b: Counter) : Int {  
        if (a.getValue()%24 < b.getValue()%24) return -1;  
        if (a.getValue()%24 > b.getValue()%24) return +1;  
        /* else */ return 0;  
    }  
});
```

Kotlin

Interfacce marker



IL LIMITE ESTREMO: INTERFACCE VUOTE ("MARKER")

- Per sfruttare appieno le interfacce come strumento concettuale di modellazione, si introducono talora delle **interfacce vuote**, dette *marcatori (marker)*
- Sono **utili proprio perché introducono un tipo**:
 - implementarle non costa nulla (*non c'è codice da scrivere*)
 - ma *definiscono un tipo* con cui spaziare nella tassonomia di classi
- Ciò permette di **sfruttare il compilatore per scovare incongruenze** nelle scelte e nell'uso dei tipi
 - se una funzione si aspetta un argomento di un certo tipo (interfaccia) e viene passato un oggetto che non lo è, *ci viene segnalato* 😊
 - ESEMPI: *Cloneable*, *Serializable*, ...



INTERFACCE "MARKER": ESEMPIO

- Si supponga che in un sistema *solo certe classi* debbano essere *mostrabili a video* tramite un certo metodo **show**
 - naturalmente, ogni classe ha una `toString` appropriata, ma noi vogliamo che **solo alcune**, quelle «taggate», siano *visualizzabili*
- IDEA: introduciamo il tipo-marker **Dentable**
 - ASSUNTO: solo le classi (ad-) "dentabili" ☺ devono poter essere visualizzabili

```
public interface Dentable {}
```

Java

C#

~Scala

Kotlin

- è un'interfaccia vuota, perché *in realtà non occorre scrivere codice*: tutti gli oggetti hanno già una `toString` !
- **lo scopo è distinguere queste classi dalle altre**: così, possiamo sfruttare il compilatore per garantire che *solo loro si stampino*.

INTERFACCE "MARKER": ESEMPIO

- Definiamo allora il metodo **show** in modo che *accetti solo oggetti di tipo **Dentable***

```
public static void show(Dentable d) {  
    System.out.println(d);  
}
```

Java

~C#

~Kotlin

~Scala

Anche se concretamente qualunque oggetto avrebbe una `toString`...

..grazie al tipo-interfaccia usato come marcatore, solo i *dentabili* si stampano!

- Risultato: come si voleva, è impossibile chiamare **show** su oggetti «non **Dentable**» 😊
 - il bello è che **è il compilatore a fare questo controllo per noi !**
 - GARANZIA DI CONSISTENZA: *esprimendo il vincolo attraverso il type system*, abbiamo potuto sfruttare il sistema di type check del compilatore per farci fare questo controllo *gratis* 😊**RICORDA: il type system è un alleato, non un nemico!**



INTERFACCE "MARKER": ESEMPIO

- E infatti...

```
public class Good implements Dentable {  
    ...  
}
```

Java

~C#

~Kotlin

~Scala

```
public class Bad { // not Dentable  
    ...  
}
```

Java

~C#

~Kotlin

~Scala

```
Good g = new Good(..);  
show(g); // OK  
  
Bad b = new Bad(..);  
show(b); // NO!
```

ERROR: method show cannot be applied to given types
required: Dentable; found: Bad
Reason: **actual argument Bad cannot be converted to Dentable** by method invocation conversion