Le interfacce

Un **metodo astratto** è un metodo senza corpo, con un "**;**" dopo l'intestazione.

Una **interfaccia** (**interface**) in Java ha una struttura simile a una classe, ma può contenere SOLO **costanti** e **metodi d'istanza astratti** (quindi non può contenere né *costruttori*, né *variabili statiche*, né *variabili di istanza*, né *metodi statici*).

Ad esempio, questa è la dichiarazione dell'interfaccia `java.lang.Comparable`:

```
public interface Comparable {

    public int compareTo(Object o);

}
```

Spesso l'interfaccia comprende anche una descrizione informale del significato dei metodi (che chiameremo **specifica** o **contratto**).

Ad esempio, nella API di `Comparable` [locale, Medialab, Sun] si vede che al metodo `compareTo` è associata una precisa interpretazione (descritta a parole): esso definisce un ordinamento totale sugli oggetti della classe. Sotto è riportato un estratto della specifica di `compareTo`:

> **compareTo:** Compares `this` object with the specified object for order. Returns a negative integer, zero, or a positive integer as this object is less than, equal to, or greater than the specified object.
> [...] The implementor must also ensure that the relation is transitive: `(x.compareTo(y)>0 && y.compareTo(z)>0)` implies `x.compareTo(z)>0`.
> [...] It is strongly recommended, but not strictly required that `(x.compareTo(y)==0) == (x.equals(y))`. [...]

Implementare un'interfaccia

Si può dichiarare che una classe **implementa** (**implements**) una data interfaccia: in questo caso la classe deve fornire una **realizzazione** per tutti i metodi astratti dell'interfaccia, cioè la classe deve fornire metodi con la stessa firma descritta nell'interfaccia (e con il corpo, naturalmente). La realizzazione di un metodo deve anche rispettare la "specifica" del corrispondente metodo astratto.

Ad esempio, la seguente classe `Intero` (una versione semplificata di `Integer`) implementa `Comparable` rispettando la specifica di `compareTo`.

```
public class Intero implements Comparable {

    private int n;

    public Intero(int n) {
        this.n = n;
    }

    public int compareTo(Object o){
        Intero int1 = (Intero) o;
        if (n < int1.n) return -1;
        else if (n > int1.n) return 1;
        else return 0;
    }

    public String toString() {
        return ""+n;
    }
}
```

A che servono le interfacce?

Le interfacce possono essere utilizzate:

- per definire **Tipi di Dati Astratti** (che vedrete nel corso di MP); si pensi al TDA degli *insiemi*, definiti come entità matematiche caratterizzate dalle usuali operazioni (*unione*, *appartenenza*, ...), e a diverse possibili implementazioni del TDA (*liste con/senza ripetizioni*, *array*, *tabelle hash*, *alberi binari di ricerca*, ...);

- come **contratto** tra chi implementa una classe e chi la usa: le due parti possono essere sviluppate e compilate separatamente;

- per **evidenziare funzionalità comuni** a più classi, sopperendo alle limitazioni dell'ereditarietà singola (come nell'esempio di `Comparable` che stiamo vedendo);
- per scrivere **programmi generici** (applicabili a più classi), evitando di duplicare il codice, come nell'esempio che segue;
- per determinare se una classe soddisfa oppure no una certa proprietà (ad esempio, l'intefaccia `Cloneable`).

Un esempio di uso di `Comparable`

La classe `Max` fornisce il metodo `max` che restituisce il massimo elemento di un array di `Comparable`, usando il metodo `compareTo` per confrontare gli oggetti:

```
public class Max {
    public static Object max (Comparable [] array){
        if (array == null) return null;
        Comparable max = null;
        for (int i = 0; i < array.length; i++){
            if (max == null || array[i].compareTo(max) > 0)
                max = array[i];
        }
        return max;
    }
}
```

La classe `MaxInteroTest` usa `Max.max` per stampare il massimo di un array di `Intero`: questo è possibile perché `Intero` implementa `Comparable`.

```
public class MaxInteroTest{
    public static void main(String [] args){
        Intero [] array =
        {new Intero(4), new Intero(7), new Intero(2)};
        System.out.println(Max.max(array));
    }
}
```

La classe `MaxStringTest` usa invece `Max.max` per stampare il massimo di un array di `String`: infatti anche `String` implementa `Comparable`.

```
public class MaxStringTest{
    public static void main(String [] args){
        String [] array =
        {"pluto", "pippo", "zorro", "paperino"};
        System.out.println(Max.max(array));
    }
}
```

Analogamente, sfruttando l'interfaccia `Comparable`, si possono scrivere algoritmi di ordinamento generici, applicabili a istanze di qualunque classe che la implementi.

Regole per l'uso di interfacce

Nell'uso delle interfacce in un programma, ricordarsi delle seguenti regole:

- Possiamo dichiarare una variabile indicando come tipo un'interfaccia:

  ```
  Comparable cmp;
  ```

- Non possiamo istanziare un'interfaccia:

  ```
  Comparable com = new Comparable(); // VIETATO
  ```

- Ad una variabile di tipo interfaccia possiamo assegnare solo istanze di classi che implementano l'interfaccia:

  ```
  Comparable com = new Intero(5);
  ```

- Su di una variabile di tipo interfaccia possiamo invocare solo metodi dichiarati nell'interfaccia (o nelle sue "super-interfacce").

```
...

Comparable stuff, stuff1;       // OK

stuff = new Comparable();       // NO!!

stuff = new Intero(5);          // OK

stuff1 = "pippo";               // OK

System.out.print(stuff1.length());  // NO!!
```

```
System.out.print(stuff.compareTo(stuff1));
     //OK, compila correttamente.
...
```

Dichiarazione di interfacce

Si dichiara come una classe, ma con la parola chiave **`interface`** (al posto di **`class`**):

```
public interface <Int> [extends <Int1>, <Int2>, ...] {
   <tipo1> <var1> = <val1>;
   ...

   <tipoN> <varN> = <valN>;

   <tipo-res1> <metodo1> ( <lista-parametri1> );

   ...

   <tipo-resM> <metodoM> ( <lista-parametriM> );

}
```

Si noti che:

- Le variabili devono essere inizializzate e non possono essere modificate successivamente: anche se non sono dichiarate **`final`** di fatto sono delle **costanti**;
- I metodi sono tutti astratti: infatti al posto del corpo c'è solo un punto e virgola;
- I metodi dichiarati in una interfaccia sono sempre **`public`**. Di conseguenza, i corrispondenti metodi di una classe che implementa l'interfaccia devono essere **`public`**.
- Una interfaccia può estendere una o più **interfacce** (non **classi**), indicate dopo la parola chiave **`extends`**. Per le interfacce non vale la restrizione di "ereditarietà singola" che vale per le classi.

Le relazioni `extends` e `implements`

- Ogni classe estende (`extends`) una sola altra classe (`Object` se non specificata);
- Una interfaccia può estendere (`extends`) una o più interfacce:

```
public interface <Int> [extends <Int1>, <Int2>, ...] {
   ...
}
```

- La gerarchia di ereditarietà singola delle classi e la gerarchia di ereditarietà multipla delle interfacce sono completamente disgiunte.
- Una classe può implemetare (`implements`) una o più interfacce:

```
public class <nomeClasse> extends <nomeSuperclasse>
       implements <Int1>, <Int2>, ..., <Intn> {
   ...
}
```

- In questo caso `<nomeClasse>` deve fornire una realizzazione per tutti i metodi delle interfacce `<Int1>, <Int2>, ...,<Intn>` che implementa, nonché per i metodi di eventuali super-interfacce da cui queste ereditano.

Interfacce come tipi di dati astratti

Come sappiamo, in Java ogni classe definisce un **tipo di dati**, i cui **elementi** sono le istanze della classe (la cui struttura è determinata dalle variabili d'istanza), e le cui **operazioni** sono i metodi.

Una interfaccia definisce invece un **tipo di dati astratto**, di cui fornisce la **specifica delle operazioni**: la struttura dei suoi elementi e il modo in cui le operazioni sono effettivamente definite verrà determinato dal tipo di dati (la

classe) che realizza (**`implements`**) l'interfaccia.

Esempio: L'interfaccia **`List`** [locale, Medialab, Sun] e le classi **`ArrayList`** e **`Vector`**.

Interfacce come 'contratto'

Le interfacce forniscono un supporto linguistico alla nozione di **contratto** tra chi usa gli oggetti di una classe, e chi realizza la classe stessa.

- Chi scrive la porzione di programma che usa gli oggetti della classe ne ha una visione astratta: conosce solo le firme dei metodi da usare e una descrizione (informale) delle operazioni, indipendentemente dalla realizzazione.
- Chi scrive la classe deve fornire una realizzazione dei metodi dell'interfaccia, fornendo una rappresentazione concreta degli oggetti.

Questo permette di procedere in parallelo alla scrittura delle due parti, e di integrarle alla fine.

Inoltre la classe che implementa l'interfaccia può essere sostituita con un'altra più efficiente (o più conveniente...), senza modificare il programma che la utilizza.

Interfaccia come proprietà

Si possono usare le interfacce per associare alle classi una certa proprietà, nel modo seguente:

- Si definisce una interfaccia, tipicamente senza alcun metodo, che rappresenta la proprietà. Ad esempio: `Cloneable`, `java.io.Serializable`.
- Si dichiara che una classe implementa l'interfaccia per esprimere il fatto che la classe soddisfa quella proprietà. Ad esempio, `ArrayList` implementa sia `Cloneable` che `java.io.Serializable`; invece `Integer` implementa `java.io.Serializable` ma non `Cloneable`.
- Quando necessario, si può controllare se un oggetto soddisfa la proprietà usando `instanceof`.

```
import java.io.Serializable;
import java.util.ArrayList;
public class prova {
    public static void main(String[] args) {

        Object  obj  = new ArrayList();
        System.out.println("ArrayList implementa Cloneable: " +
                           (obj instanceof Cloneable));
        System.out.println("ArrayList implementa Serializable: " +
                           (obj instanceof Serializable));
        obj  = new Integer(5);
        System.out.println("Integer implementa Cloneable: " +
                           (obj instanceof Cloneable));
        System.out.println("Integer implementa Serializable: " +
                           (obj instanceof Serializable));
    }
}
```

Si osservi che con questo meccanismo, una proprietà viene ereditata dalle sottoclassi (se una classe implementa un'interfaccia, tutte le sue sottoclassi la implementano).

L'interfaccia `Cloneable` [locale, Medialab, Sun] viene utilizzata per determinare se un oggetto può essere **clonato** oppure no. Si guardi la documentazione del metodo `clone()` di `Object` [locale, Medialab, Sun].

L'interfaccia `java.io.Serializable` viene utilizzata per determinare se un oggetto può essere **serializzato** oppure no. Vederemo la serializzazione di oggetti in seguito.