Lezione 11
Interfacce

Le interfacce sono una completa estremizzazione del concetto di classe astratta. Servono per specificare cosa una classe debba fare, ma non come debba farlo. Sommariamente, è possibile pensare alle interfacce come a delle classi astratte costituite da soli metodi astratti. Ciononostante, il paragone rischia di essere forviante. Le interfacce, infatti, rappresentano anche la via di Java all'ereditarietà multipla.

11.1 - Definizione di un'interfaccia

La definizione di un'interfaccia, a grosse linee, ricalca quella di una classe:

```
interface NomeInterfaccia {

  accesso tipo-ritorno nomeMetodo1(lista-argomenti);

  accesso tipo-ritorno nomeMetodo2(lista-argomenti);

  accesso tipo-ritorno nomeMetodo3(lista-argomenti);

  ...

}
```

Rispetto alle classi, è importante osservare le seguenti differenze:

1. La parola chiave utilizzata per la definizione di un'interfaccia è *interface*, anziché *class*.
2. Tutti i metodi dichiarati in un'interfaccia non hanno corpo. Pertanto, sono automaticamente metodi astratti, senza il bisogno di dichiararlo esplicitamente con la parola *abstract*.

I metodi contenuti in un'interfaccia possono essere solamente *public* o ad accesso non specificato. Non è ammessa la dichiarazione di metodi *private* o *protected*, come invece è lecito fare nelle classi. La motivazione è presto spiegata. Le interfacce servono per fornire dei modelli di comportamento. Lo dice il loro stesso nome. Una classe che implementa un'interfaccia, come si vedrà nel paragrafo successivo, deve fornire un corpo ad ogni metodo da questa richiesto. La situazione è analoga a quella esaminata con le classi astratte. Non importa come ogni singola classe vada ad implementare i metodi richiesti da un'interfaccia. L'unica cosa che conta, è che la classe fornirà quei metodi al programmatore che intenderà utilizzarla. I metodi *private* e *protected*, al contrario, servono per l'implementazione dei meccanismi interni, non rilevanti per chi la classe la dovrà solamente utilizzare. Non ha alcun senso richiedere metodi privati o protetti attraverso un'interfaccia, giacché non hanno alcuna influenza su come la classe si presenterà ai suoi fruitori.

Prendiamo in esame una prima semplice interfaccia. Nel corso della lezione precedente è stata sviluppata la classe astratta *Animale*. Applicando alcune piccole variazioni, è possibile mutare la classe in un'interfaccia:

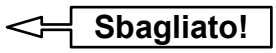
```
interface Animale {
```

```
  public void faiVerso();

  public void dimmiCiboPreferito();

}
```

Il sorgente di un'interfaccia va trattato come quello di una classe. Nel caso appena visto, ad esempio, è sufficiente raccogliere il codice dell'interfaccia *Animale* in un file chiamato *Animale.java*. Già così è possibile la compilazione, che produrrà il bytecode *Animale.class*. Come avviene con le classi astratte, non è possibile istanziare direttamente un'interfaccia:

```
Animale a = new Animale();
```


Un'interfaccia esiste solamente per essere implementata da una o più classi.

11.2 - Implementazione di un'interfaccia

Implementare un'interfaccia significa realizzare una classe che dia definizione ai metodi da essa richiesti. La parola chiave al centro dell'implementazione è *implements*:

```
class NomeClasse implements NomeInterfaccia {
  // ...
}
```

Una classe può contemporaneamente estendere un'altra classe ed implementare una o più interfacce. Ecco un esempio pratico:

```
// Gatto.java
class Gatto implements Animale {

  public void faiVerso() {
    System.out.println("Miaaaaaaaaooo!");
  }

  public void dimmiCiboPreferito() {
    System.out.println("Pesce!");
  }

}

// Cane.java
class Cane implements Animale {

  public void faiVerso() {
    System.out.println("Bau!");
  }

  public void dimmiCiboPreferito() {
    System.out.println("Carne!");
  }

}

// Topo.java
class Topo implements Animale {

  public void faiVerso() {
    System.out.println("Squit!");
  }
```

```
  public void dimmiCiboPreferito() {
    System.out.println("Formaggio!");
  }

}
```

Sono state realizzate tre classi che implementano l'interfaccia *Animale*. All'interno di ognuna di esse, è stato necessario dare corpo ai due metodi *faiVerso()* e *dimmiCiboPreferito()*. Non si può prescindere da questo punto. Se non si dà implementazione a tutti i metodi richiesti da un'interfaccia, si riceve un errore di compilazione come il seguente:

```
NomeClasse should be declared abstract; it does not define nomeMetodo()
```

Solo le classi astratte possono permettersi il lusso di dimenticarsi dei metodi richiesti da un'interfaccia. L'obbligo di dare implementazione ai metodi esclusi ricadrà sulle spalle di chi estenderà la classe astratta.

Ora che sono state realizzate le tre classi *Gatto*, *Cane* e *Topo*, è possibile eseguire un test:

```
class Test {

  public static void main(String[] args) {
    Animale a1 = new Gatto();
    Animale a2 = new Cane();
    Animale a3 = new Topo();
    System.out.println("- Gatto ----------------");
    a1.faiVerso();
    a1.dimmiCiboPreferito();
    System.out.println("- Cane -----------------");
    a2.faiVerso();
    a2.dimmiCiboPreferito();
    System.out.println("- Topo -----------------");
    a3.faiVerso();
    a3.dimmiCiboPreferito();
  }

}
```

L'output è abbastanza ovvio:

```
- Gatto ----------------
Miaaaaaaaaooo!
Pesce!
- Cane -----------------
Bau!
Carne!
- Topo -----------------
Squit!
Formaggio!
```

Benché non sia possibile istanziare direttamente degli oggetti di tipo *Animale*, è possibile creare dei riferimenti di questo tipo. La situazione ricorda da vicino i temi già discussi dell'ereditarietà. Su un riferimento di tipo *Animale* è possibile richiamare tutti i metodi

richiesti dall'interfaccia *Animale*. Lo smistamento verso la reale definizione del metodo è automatico e affidato direttamente al runtime di Java.

11.3 - Implementare più interfacce

Come si è detto più volte, le interfacce permettono una sorta di ereditarietà multipla. Anzitutto, una classe può contemporaneamente estendere un'altra classe ed implementare un'interfaccia. Inoltre, ogni classe può implementare due o più interfacce simultaneamente:

```
class NomeClasse implements Interfaccia1, Interfaccia2, Interfaccia3, ... {
  // ...
}
```

Si prendano in esame le interfacce esemplificative *StrumentoMusicale* e *OggettoDiLegno*:

```
// StrumentoMusicale.java
interface StrumentoMusicale {

  public void suona();

}

// OggettoDiLegno.java
interface OggettoDiLegno {

  public void dimmiTipoLegno();

}
```

Una chitarra è sia uno strumento musicale sia un oggetto di legno. Quindi, si può definire una classe *Chitarra* al seguente modo:

```
class Chitarra implements StrumentoMusicale, OggettoDiLegno {

  private String tipoLegno;

  public Chitarra(String tipoLegno) {
    this.tipoLegno = tipoLegno;
  }

  // Richiesto da StrumentoMusicale:
  public void suona() {
    System.out.println("Do Re Mi Fa Sol La Si");
  }

  // Richiesto da OggettoDiLegno:
  public void dimmiTipoLegno() {
    System.out.println(tipoLegno);
  }

}
```

In questo caso, ci si può riferire ad un oggetto *Chitarra* in tre maniere differenti:

1. Usando una variabile di tipo *Chitarra*, alla maniera classica. In questo caso, tutti i metodi esposti da *Chitarra* possono sempre essere richiamati.
2. Usando una variabile di tipo *StrumentoMusicale*. In questo caso, sono a

disposizione esclusivamente i metodi esposti dall'interfaccia *StrumentoMusicale*, cioè il solo metodo *suona()*.

3. Usando una variabile di tipo *OggettoDiLegno*. In questo caso, sono a disposizione esclusivamente i metodi esposti dall'interfaccia *OggettoDiLegno*, cioè il solo metodo *dimmiTipoLegno()*.

Si verifichi il funzionamento delle strutture realizzate, mediante il seguente test:

```
class Test {

  public static void main(String[] args) {
    // Definisco un oggetto Chitarra usando un riferimento
    // del medesimo tipo.
    Chitarra c = new Chitarra("Palissandro");
    // Uso normalmente i metodi di Chitarra.
    c.suona();
    c.dimmiTipoLegno();
    // Passo ad un riferimento di tipo StrumentoMusicale.
    StrumentoMusicale s = c;
    // Ora ho a disposizione solo suona();
    s.suona();
    // Passo ad un riferimento di tipo OggettoDiLegno.
    OggettoDiLegno o = c;
    // Ora ho a disposizione solo dimmiTipoLegno();
    o.dimmiTipoLegno();
  }

}
```

11.4 - Ereditarietà tra interfacce

Due interfacce possono derivare l'una dall'altra. Il meccanismo è semplice:

```
// Interfaccia1.java
interface Interfaccia1 {

  public void metodo1();

}

// Interfaccia2.java
interface Interfaccia2 extends Interfaccia1 {

  public void metodo2();

}
```

Interfaccia2 eredita da *Interfaccia1*. Questo significa che le classi che implementeranno *Interfaccia2* dovranno definire sia i metodi richiesti da *Interfaccia2* sia quelli voluti da *Interfaccia1*. Ecco un esempio:

```
class Esempio implements Interfaccia2 {

  // Richiesto da Interfaccia1.
  public void metodo1() {
    System.out.println("metodo1()");
  }

  // Richiesto da Interfaccia2.
```

```
  public void metodo2() {
    System.out.println("metodo2()");
  }

}
```

Agli oggetti di tipo *Esempio* ci si potrà riferire usando variabili sia di tipo *Esempio* sia di tipo *Interfaccia1* sia di tipo *Interfaccia2*, secondo le norme già note:

```
class Test {

  public static void main(String[] args) {
    Esempio e = new Esempio();
    e.metodo1();
    e.metodo2();
    Interfaccia1 i1 = e;
    i1.metodo1();
    Interfaccia2 i2 = e;
    i2.metodo2();
  }

}
```

11.5 - Interfacce come collezioni di costanti

Un altro utilizzo delle interfacce permette la raccolta di costanti. In tal caso, invece che definire dei metodi, l'interfaccia conterrà delle proprietà. Si consideri il seguente esempio:

```
interface CostantiMatematiche {

  public double PI = 3.14;
  public double E = 2.71;

}
```

L'interfaccia *CostantiMatematiche* contiene due proprietà: *PI* (*pi greco*) ed *E* (base dei logaritmi naturali). Questi valori possono essere impiegati in diversi modi:

```
// TestCostanti1.java
class TestCostanti1 {

  public static void main(String[] args) {
    System.out.println(CostantiMatematiche.PI);
    System.out.println(CostantiMatematiche.E);
  }

}

// TestCostanti2.java
class TestCostanti2 implements CostantiMatematiche {

  public static void main(String[] args) {
    System.out.println(PI);
    System.out.println(E);
  }

}
```

La classe *TestCostanti1*, mostrata per prima, non stringe alcun rapporto con l'interfaccia

CostantiMatematiche. Semplicemente, accede ai valori in essa conservati, usando le diciture:

```
CostantiMatematiche.PI
CostantiMatematiche.E
```

TestCostanti2, al contrario, implementa l'interfaccia *CostantiMatematiche*. Così facendo, importa al proprio interno le variabili *PI* ed *E*. Per richiamarle, non dovrà anteporre nulla al loro nome. Infatti, le richiama scrivendo direttamente:

```
PI
E
```

In ambo i casi, ad ogni modo, vale la medesima regola: le proprietà di un'interfaccia sono automaticamente considerate *static* e *final*. Quindi:

1. Possono essere richiamate senza che esista un'istanza di un oggetto che implementi l'interfaccia.
2. Il loro valore non può essere modificato.

Inoltre, costruendo un'interfaccia, non è possibile dichiarare una costante senza inizializzarla. Ad esempio, la seguente interfaccia non può essere compilata:

```
interface TestErrore {

  public int A;

}
```

L'errore che si riceve è il seguente:

```
TestErrore.java:3: = expected
  public int A;
             ^
```

Comunque sia, un'interfaccia può contemporaneamente sia definire dei metodi sia dichiarare delle costanti.

11.6 - Interfacce e casting

Si torni a considerare l'esempio di *Animale*, *Gatto*, *Cane* e *Topo*. E' stato detto che una variabile di tipo *Animale* (che è un'interfaccia) può tenere riferimento di un oggetto di tipo *Gatto*, *Cane* o *Topo* (che sono classi che implementano l'interfaccia *Animale*). La faccenda è semplice:

```
Animale a = new Gatto();
```

In certi casi, si desidera poter compiere l'operazione inversa. Ad esempio, avendo a disposizione un riferimento di tipo *Animale*, si immagini di voler risalire al riferimento verso la reale classe dell'oggetto. L'operazione è possibile attraverso un casting esplicito, a patto di conoscere quale classe sia quella di appartenenza dell'oggetto:

```
Animale a = new Gatto();
// ...
```

```
Gatto g = (Gatto) a;
```

Con questa tecnica, è possibile tornare al riferimento più dettagliato. Il trucco, oltre che con le interfacce, funziona anche nei casi di semplice ereditarietà.
Può capitare che non si conosca con certezza quale sia la classe di appartenenza dell'oggetto trattato. Avendo a disposizione un riferimento di tipo *Animale*, ad esempio, si potrebbe non sapere a priori se questo è un *Gatto*, un *Cane* oppure un *Topo*. L'operatore *instanceof*, in situazioni del genere, permette di vagliare i casi possibili, in modo da poter eseguire dei casting a prova di bomba:

```
public void gestisciAnimale(Animale a) {
  if (a instanceof Gatto) {
    Gatto g = (Gatto)a;
    // ...
  } else if (a instanceof Cane) {
    Cane c = (Cane)a;
    // ...
  } else if (a instanceof Topo) {
    Topo t = (Topo)a;
    // ...
  }
}
```

Seguire questa direttiva è molto importante! Il casting esplicito, infatti, è una dichiarazione attraverso la quale il programmatore si assume la responsabilità di quello che sarà fatto. Se l'operazione non viene ben gestita, è molto probabile che il programma andrà in crash oppure che presenterà dei malfunzionamenti. L'errore che si riceve quando si tenta una conversione scorretta (ad esempio, un *Cane* in un *Gatto*) è il seguente:

```
java.lang.ClassCastException
```

Si tratta di un errore di runtime. Il compilatore non può preventivamente accorgersi di un malfunzionamento di questo tipo, poiché dipende da condizioni verificabili solo durante l'esecuzione. In compenso, l'errore si propaga sotto forma di eccezione, e le eccezioni possono essere gestite (come si vedrà tra qualche lezione).