Interfacce

Sono simili alle classi, ma contengono solo dichiarazioni di metodo

Dichiarare un metodo=garantire che sarà presente in tutte le sottoclassi

---

Metodi e sottoclassi

Se una classe contiene un metodo `stampa`...

...allora tutte le sottoclassi contengono un metodo `stampa`:

- o è quello ereditato;
- oppure è stato ridefinito.

Comunque sia, un metodo `stampa` è definito in tutte le sottoclassi

---

Interfaccia=dicharazione di metodi

Una interfaccia è simile a una classe, ma contiene solo *dichiarazioni* di metodo

Viene specificata solo la firma del metodo, non la sua implementazione

```
interface Stampabile {
  public void stampa();
}
```

È simile ad una classe astratta con tutti metodi astratti
(c'è una differenza: l'ereditarietà multipla)

A cosa servono le interfacce?

Si possono definire classi che *implementano* i metodi delle interfacce:

```
class Studente implements Stampabile {
  String nome;
  int anno;

  public void stampa() {
    System.out.println(nome+" "+anno);
  }
}
```

Si dice che `Studente` implementa l'interfaccia, e che `Studente` è una implementazione dell'interfaccia

---

Interfacce e sovraclassi

Simili:

- in una variabile `Stampabile` posso mettere un oggetto `Studente`
- ogni implementazione ("sottoclasse") di `Stampabile` contiene il metodo `stampa`

Differenze:

- non si può creare un oggetto `Stampabile`
- le implementazioni *devono* (ri)definire tutti i metodi dell'interfaccia (le sottoclassi possono, ma devono farlo solo se sono metodi astratti)

Variabili interfaccia

Si può creare una variabile di una interfaccia:

```
  public static void main(String args[]) {
    Stampabile s;
```

Si può mettere in essa il riferimento a un qualsiasi oggetto di una sua "sottoclasse":

```
    s=new Studente();
```

Si possono invocare i metodi dell'interfaccia:

```
    s.stampa();
  }
```

Cosa succede quando si invoca un metodo?

La variabile `s` è di tipo `Stampabile`, che è una interfaccia

Non esistono oggetti di tipo `Stampabile`
(la invocazione `s=new Stampabile();` genera un errore)

In `s` si possono mettere *solo* oggetti di una classe che implementa l'interfaccia

Tutte le classi che implementano `Stampabile` devono contenere un metodo `stampa`

L'istruzione `s.stampa()` è corretta: viene invocato il metodo `stampa` dell'oggetto che sta in `s`

Vantaggio delle interfacce

La stessa interfaccia può essere implementata da classi di tipo completamente diverso:

```
class Barattolo implements Stampabile {
  int codice;

  public void stampa() {
    System.out.println(codice);
  }
}
```

Vale lo stesso discorso di `Studente`: un oggetto `Barattolo` si può mettere in una variabile `Stampabile`

```
public static void main(String args[]) {
  Stampabile s;

  if(metodo())
    s=new Studente();
  else
    s=new Barattolo();

  s.stampa();
}
```

Quale metodo viene invocato?

La scelta del metodo

- variabile `Stampabile`
- oggetto `Barattolo`

Si invoca il metodo implementato in `Barattolo`

È sempre la stessa regola

In questo caso non ci sono dubbi (nella interfaccia `Stampabile` il metodo `stampa` è solo dichiarato, non implementato)

Polimorfismo

Quando si fa `s.stampa()`:

oggetto `Studente`
: viene invocato il metodo `stampa` di `Studente`

oggetto `Barattolo`
: viene invocato il metodo `stampa` di `Barattolo`

I due metodi possono anche avere implementazione molto diversa

Il metodo che viene invocato dipende dal tipo dell'oggetto (polimorfismo)

Succede anche con le sottoclassi

A cosa servono le interfacce?

Se una classe implementa un interfaccia, *dichiara* di possedere tutti i metodi dell'interfaccia

In una variabile del tipo dell'interfaccia posso mettere un qualsiasi oggetto di una classe che implementa l'interfaccia

Posso invocare il metodo `stampa` su una variabile dell'interfaccia senza sapere quale è il tipo effettivo dell'oggetto

---

`implements`=garantisco che ho dei metodi

Ogni classe definita con `implements Stampabile` contiene il metodo `stampa`

Una variabile `Stampabile` può solo contenere oggetti per i quali esiste `stampa`

Metodi

Invocare un metodo=copiare i parametri + altre operazioni

Se si può mettere uno `Studente` in una variabile `Stampabile`...

...allora si può passare uno `Studente` a un metodo che ha come parametro una variabile `Stampabile`

```
class DueVolte {
  static void stampaDueVolte(Stampabile s) {
    s.stampa();
    s.stampa();
  }

  public static void main(String args[]) {
    Studente a=new Studente();
    Barattolo b=new Barattolo();

    stampaDueVolte(a);
    stampaDueVolte(b);
  }
}
```

Vantaggio del metodo

Il metodo che stampa due volte funziona su oggetti che possono essere completamente diversi: `Studente` e `Barattolo`

Vincoli:

- gli oggetti che passo devono essere di una classe che implementa `Stampabile` e quindi...
- devono avere un metodo `stampa`

Codice generico

Si possono scrivere frammenti di codice senza specificare esattamente il tipo degli oggetti su cui lavorano
(specifica solo parziale)

Si può usare lo stesso codice per lavorare su oggetti diversi

Non devo definire due o più metodi diversi:

```
  static void stampaDueVolte(Studente s) {
    s.stampa();
    s.stampa();
  }

  static void stampaDueVolte(Barattolo s) {
    s.stampa();
    s.stampa();
  }
```

Ordinare un array

Si può ordinare un array di `Object`?

No, perchè non esiste un ordinamento per tutti gli oggetti

Per ordinare un insieme di oggetti, deve esistere una funzione di confronto di questi oggetti

Interfaccia predefinita `Comparable`: rappresenta tutti gli oggetti sui quali è definito un ordinamento totale

---

Definizione di `Comparable`

```
interface Comparable {
  public int compareTo(Object o);
}
```

A parole: se una classe implementa `Comparable`, allora ha un metodo `compareTo(Object o)`

Quando si definisce una classe, il metodo `compareTo` viene implementato in modo tale che:

```
  a.compareTo(b)
```

Ritorna:

un valore minore di zero
    se `a` è minore di `b`
zero
    se `a` è uguale a `b`
valore maggiore di zero
    se `a` è maggiore di `b`

Cosa ci si aspetta != cosa di può fare

La definizione del metodo `compareTo` è semplicemente che si tratta di un metodo che ha un `Object` come argomento e ritorna un `int`

Il fatto che `a.compareTo(b)` dice se `a` è minore, uguale o maggiore di `b` è una convenzione

Non c'è nessun vincolo sintattico che ci obbliga a implementare `compareTo` come il confronto:

```
class Strana implements Comparable {
  public int compareTo(Object o) {
    return (int) Math.random()*2-1;
  }
}
```

È un errore semantico/concettuale

Effetto di un errore del genere: i metodi che lavorano su oggetti `Comparable` si comportano in modo anomalo

Una cosa simile succede per `equals`

Classe che implementa `Comparable`

La classe `Integer` implementa comparable

Il metodo `compareTo` di `Integer` si comporta esattamente come ci si aspetta

---

Esempio di uso dell'interfaccia

Metodo che ordina un array di due elementi:

```
  static void ordina(Comparable c[]) {
    Comparable t;

    if(c[0].compareTo(c[1])<=0)
      return;

    t=c[0];
    c[0]=c[1];
    c[1]=t;
  }
```

Esempio di programma:

```
  public static void main(String args[]) {
    Integer a[]=new Integer[2];

    a[0]=new Integer(4);
    a[1]=new Integer(2);

    ordina(a);

    System.out.println(a[0]+" "+a[1]);
  }
```

Riuso del metodo

Lo stesso metodo funziona per qualsiasi classe che implementa l'interfaccia `Comparable`

`String` implementa `Comparable`

L'ordinamento fra stringhe è quello lessicografico (detto anche alfabetico...)

```
String b[]=new String[2];

b[0]="Ciccio";
b[1]="Bruno";

ordina(b);

System.out.println(b[0]+" "+b[1]);
```

Uso lo stesso metodo per le stringhe

Nota su `compareTo` di `String`

Un classe che implementa `Comparable` deve contenere un metodo `int compareTo(Object o)`

La classe `String` contiene due metodi `compareTo`:

- int compareTo(Object o)
- int compareTo(String s)

È un metodo sovraccarico

Il primo metodo converte `o` in stringa, se possibile, e poi invoca il secondo

---

Esempio di `compareTo` su `String`

Cosa succede qui sotto?

```
public static void main(String args[]) {
  String s="abcd";
  Comparable c=s;

  System.out.println(s.compareTo("abce"));
  System.out.println(c.compareTo("abce"));
}
```

`s.compareTo("abce")`

stringa come oggetto di invocazione e stringa come argomento:
viene invocato il metodo `int compareTo(String s)`

`c.compareTo("abce")`

- `c` è di tipo `Comparable`
- il metodo `compareTo` di `Comparable` ha come argomento un `Object`
- viene invocato il metodo `int compareTo(Object o)` di `String`

Nonostante l’argomento sia una stringa, viene invocato l’altro metodo.

Perchè tutto questo macello?

Quando `s` è di tipo stringa e si passa una stringa, viene invocato `int compareTo(String s)`

Si risparmia il tempo di controllare il tipo e fare il cast

Altro esempio di interfaccia predefinita

```
interface Runnable {
  void run();
}
```

Ogni implementazione contiene un metodo `run`
(usato per il multithreading)

Non si poteva fare tutto con le sottoclassi?

Se una classe è sottoclasse di un’altra, è comunque garantito che ne ha tutti i metodi

Nel caso di due classi soltanto si poteva anche fare:

```
abstract class Stampabile {
  abstract void stampa();
}

class Studente extends Stampabile {
  ...

  void stampa() {
    ...
  }
}
```

Una classe può essere sottoclasse solo di un’altra, mentre può implementare più interfacce

---

Ereditarietà multipla

Non si può fare:

```
class BorsistaLavoratore extends Borsista, Lavoratore {
  ...
}
```

Motivo: la componente `stipendio` ha un significato in `Borsista` e una altro in `StudenteLavoratore`

Concettualmente, dovrei avere entrambe in `BorsistaLavoratore`

Operativamente, non posso avere due componenti con lo stesso nome

Per questo, in Java non si può fare l'ereditarietà multipla

Si può fare con le interfacce

Implementare più interfacce

Una interfaccia dice solo che una classe deve avere certi metodi

Implementare più interfacce=obbligare a implementare tutti i metodi di tutte le interfacce

```
interface Aggiornabile {
  void cambiaAnno(int anno);
}
```

Rappresenta gli oggetti per i quali è possibile modificare la componente anno

```
class Studente implements Stampabile, Aggiornabile {
  String nome;
  int anno;

  public void stampa() {
    System.out.println(nome+" "+anno);
  }

  public void cambiaAnno(int anno) {
    this.anno=anno;
  }
}
```

Se implemento sia `Stampabile` che `Aggiornabile`, poi devo definire tutti i metodi di queste due interfacce

---

Implementare != definire un metodo

Se non metto `implements Comparable`, non ho implementato l'interfaccia

Questo vale *anche* se poi la classe contiene un metodo `public int compareTo(Object o)`

Anche in questo caso, non si può mettere un oggetto della classe in una variabile `Comparable`

Quindi, non si possono usare i metodi che hanno come parametro un oggetto `Comparable`, ecc.

Implementare una interfaccia: bisogna sia dichiarare che si sta implementando (`implements`) sia definire tutti i metodi dell'interfaccia