PROGRAMMAZIONE 2

4. Ereditarietà

Ereditarietà tra classi

- Esamineremo solo la nozione di ereditarietà tra classi
- La nozione di ereditarietà vale anche per le interfacce: dettagli in qualsiasi manuale
- Strumento tipico dell'OOP per riusare il codice e creare una gerarchia di astrazioni
- Generalizzazione: una super-classe generalizza una sotto-classe fornendo un comportamento che è condiviso dalle sotto-classi
- Specializzazione: una sotto-classe specializza (concretizza) il comportamento di una super-classe

Perché è importante?

- Permette di specializzare il comportamento di una classe, prevedendo nuove funzionalità, ma al tempo stesso mantenendo le vecchie e quindi senza influenzare codice cliente già scritto
- Riuso del codice!!
- Subtype polymorphism: tramite l'ereditarietà un variabile può assumere tipi (di classi) differenti
- Una funzione con parametro formale di tipo T può operare con un parametro attuale di tipo S a patto che S sia un sotto-tipo di T

Subtyping

- B è un sotto-tipo di A: “every object that satisfies interface B also satisfies interface A”

- Obiettivo metodologico: il codice scritto guardando la specifica di A opera correttamente anche se viene usata la specifica di B

Sotto-tipi e Principio di Sostituzione

- B è sotto-tipo di A: B può essere sostituito ad A
  - una istanza del sotto-tipo soddisfa le proprietà del super-tipo
  - una istanza del sotto-tipo può avere maggiori vincoli di quella del super-tipo

Sotto-tipo: nozione semantica

- Sotto-tipo è una nozione semantica
  - B è un sotto-tipo di A se (e solo se) un oggetto di B si può mascherare come uno di A in tutti i possibili contesti
- Ereditarietà è una nozione di implementazione
  - creare una nuova classe evidenziando solo le differenze (il codice nuovo)

Ereditarietà in Java

- Una sotto-classe si definisce usando la parola chiave **`extends`**
  - **`class B extends A { … }`**
- La sotto-classe eredita le variabili d'istanza ed i metodi (a meno di quelli che vengono sovrascritti)

- Osservazione: l'ereditarietà in Java è semplice!
  - una classe può implementare più interfacce, ma estendere solo una super-classe
  - questo non vale in altri linguaggi a oggetti: l'esempio classico è C++

Esempio: senza costruttore

```
class D {
      private int x, y;
      public int addBoth( ) { return x + y; }
}

class C extends D { // C è un sottotipo di D
     private int z;
public int addThree( ) { return addBoth() + z; }
}
```

La classe C eredita implicitamente i campi definiti dalla classe D

Nella classe C non sono visibili le variabili e i metodi dichiarati **private**
Quindi fanno parte dello stato degli oggetti instanziati ma non sono riferibili (direttamente) dal nuovo metodo

Esempio

```
class D {
      protected int x, y;
      public int addBoth( ) { return x + y; }
     }

class C extends D { // C è un sottotipo di D
      private int z;
public int addThree( ) { return x+y+z; }
}
```

Nella classe C sono visibili le variabili e i metodi dichiarati **protected.** Quindi sono riferibili direttamente... ma in questo caso la soluzione precedente era metodologicamente migliore

Metodo costruttore e `Super`

- Un aspetto critico della nozione di ereditarietà è che il metodo costruttore non viene ereditato
- Tipicamente il metodo costruttore della sotto-classe deve accedere anche alle variabili di istanza della super-classe
- Java fornisce un meccanismo specifico per affrontare questo aspetto

```
class D {
 private int x, y;
    public D (int initX, int initY) {
              x = initX; y = initY;}
    public int addBoth( ) { return x + y; }
     }


class C extends D { // C è un sottotipo di D
     private int z;
public C (int initX, int initY, int initZ) {
              super(initX, initY);
                    //invocazione del costruttore di D
              z = initZ;
       }
 public int addThree( ) { return addBoth( ) + z; }
}
```

Oggetti e this

- All'interno di un metodo o di un costruttore, la parola chiave **`this`** permette di riferire l'oggetto corrente
- **`this`** è un riferimento all'oggetto corrente: l'oggetto il cui metodo o costruttore viene chiamato

Esempio: per disambiguare

```
public class Point {
    private int x;
    private int y;

      // constructor
public Point (int a, int b)
{x = a; y = b;}
}
```

```
public class Point {
    private int x;
    private int y;

      // constructor
 public Point (int x, int y) {
            this.x = x;
            this.y = y;}
}
```

Costruttore implicito: sintassi alternativa

```
public class Point {
    private int x;
    private int y;

      // constructor
 public Point (int x, int y) {
            this(x,y);}
}
```

Chiamata al costruttore
con due parametri

Esempio

```
public class Point {
    private final int x, y;
    private final String name;

public Point(int x, int y) {
        this.x = x;
        this.y = y;
        name = makeName( );
    }

protected String makeName( ) {
        return "["+x+", "+y+"]";
    }

public final String toString( )
{return name;
    }
}
```

Si inizializzano solo con i costruttori!

Non può esser sovrascritto!

```
public class ColorPoint extends Point {
    private final String color;

public ColorPoint(int x, int y, String color){
        super(x,y);
        this.color = color;
    }

    protected String makeName( ) {
        return super.makeName( ) + ":" + color;
    }

    }
}
```

Esempio

```
public static void main(String[ ]( args))  {
ColorPoint p=new ColorPoint(4, 2, “viola”);
System.out.println(p.toString());
System.out.println(p.makename());

}
```

```
TestPoint.main({ });
[4, 2]:null
[4, 2]:viola
```

Metodi addizionali

- Esistono vari metodi definiti nella classe **`Object`** che possono essere ereditati quando serve e ridefiniti da qualunque classe
- Alcuni esempi
  - **`equals`**
  - **`toString`**
  - **`clone`**

Metodi addizionali: `equals`

- In `Object` il metodo `equals` verifica se due oggetti sono lo stesso oggetto (stesso riferimento)
  - non se i due oggetti hanno lo stesso stato
  - deve essere ridefinita per i tipi non modificabili
    - ✓ in termini di uguaglianza fra gli stati
- In `Object` è presente un metodo hashCode che produce, dato un oggetto, un valore da usare come chiave in una tabella hash
  - stesso valore per oggetti equivalenti (secondo equals)
  - se un tipo non modificabile è usato come chiave, deve ridefinire anche hashCode

Metodi addizionali: `toString`

- In `Object` genera una stringa contenente il tipo dell'oggetto e il suo hash code
- Normalmente si vorrebbe ottenere una stringa composta da
  - tipo
  - valori dello stato
- Bisogna ridefinirlo (utile per testing)

Metodi addizionali: `clone`

- In **`Object`** genera una copia dell'oggetto
  - nuovo oggetto con lo stesso stato
- Questa implementazione non è sempre corretta
  - creando una situazione di condivisione (con trasmissione di modifiche) non desiderata
- Il metodo viene ereditato solo se l'header della classe contiene la clausola **`implements Cloneable`**
- Se non va bene quella di default si deve implementare di nuovo

Un Esempio finale

```
public class Person
{private String name;
 private int yearOfBirth;
public Person (String name, int yearOfBirth)
    {this.name = name;
     this.yearOfBirth = yearOfBirth;}

public String toString()    //redefined from Object
    {return "Name: " + name + "\n" +
                "Year of birth: " + yearOfBirth + "\n";
    }
**Restituisce true se this object e p sono la stessa persona.
     */
public boolean equals(Object p)    //redefined from Object
    {Person x= (Person) p;
        return  this.name.equals(x.name) &&
        this.yearOfBirth==x.yearOfBirth;
}
}
```

Una sottoclasse

```
public class Student extends Person
{private String IDnumber;
public Student(String m)
    {super("unnkown name",000);
     this.IDnumber = m;}
public Student(String name, int yearOfbirth y, String m)
    {super(name,y);this.IDnumber = m;}

public String toString()    //redefined from Person
   {return super.tostring() + IDnumber + "\n";
    }
public boolean equals(Object p)    //redefined from Person
{   Student x= (Student) p;
return super.equals(p) && x.IDnumber.equals(this.IDnumber);}
}
```

Esempio

```
public static void main(String[] args){
    Student s1=new Student("francesca",1968, "elodie");
    Student s2=new Student("francesca",2000,"elodie");
    System.out.println(s1.equals(s2));
    s1=s2;
    System.out.println(s1.equals(s2));
```

Upcasting & Downcasting

- Supponiamo che T sia una sotto-classe di S (in una gerarchia)
- Upcasting: un oggetto di tipo T può essere legato a una variabile di tipo S
- Downcasting: un oggetto di tipo S può essere legato a una variabile di tipo T

```
class Person { … };
class Student extends Person;
        //Student  sotto-tipo di Person
Person  v = (Person) new Student ( ); // upcasting
Student  c = (Student) new Person( );  // downcasting
```

Upcasting & Downcasting

- Upcasting è implicito
- Downcasting deve essere esplicito
  - non sono possibili operazioni di cast al di fuori della struttura descritta dalla gerarchia!!