Interfacce

Interfacce

- **Dichiarazioni di tipi riferimento che descrivono oggetti in modo astratto**
- **Specificano solo le firme dei metodi tralasciando tutti gli aspetti di implementazione**

Interfacce

```
interface Polynomial
{
   public void somma (Polynomial p);
   public void derivata();
   public int grado();
   public double coeff(int grado);
}
```

Interfacce

```
interface Collection<E>
{
   public void add(E element);
   public void remove(E element);
   public boolean find(E element);
   . . .
}
```

Classi implementano Interfacce

- La keyword `implements` indica che una classe implementa una interfaccia

```
class ArrayPoly implements Polynomial
{
   ...
}
```

```
class TreeSet<E> implements Collection<E>
{
   ...
}
```

Classi implementano Interfacce

- **Una classe che implementa una interfaccia deve implementare tutti i metodi dell'interfaccia**
- **Tutti I metodi dell'interfaccia devono essere dichiarati public nella classe**
- **Nulla vieta che la classe dichiari altri metodi oltre quelli dell'interfaccia che implementa**

Classi implementano Interfacce

- **Una interfaccia può essere implementata da più classi**

```
class ArrayPoly implements Polynomial
{
   ...
}
```

```
class ListPoly implements Polynomial
{
   ...
}
```

Classi implementano Interfacce

- **Una interfaccia può essere implementata da più classi**

```
class HashMap<E> implements Collection<E>
{
   ...
}
```

```
class TreeSet<E> implements Collection<E>
{
   ...
}
```

Conversioni di tipo

- **Una interfaccia “rappresenta” tutte le classi che la implementano**
- **Il tipo interfaccia si può associare a tutti gli oggetti delle classi che la implementano**

Continua…

Conversioni di tipo

- **Possiamo assegnare un riferimento di tipo classe ad una variabile di tipo interfaccia purchè la classe implementi l'interfaccia**

```
Collection<Double> c = new TreeSet<Double>();

Collection<Double> d = new HashMap<Double>();
```

```
Polynomial p = new ArrayPoly();
Polynomial q = new ListPoly();
```

Continua…

Conversioni di tipo

- La conversione è lecita solo in determinate situazioni

```
Polynomial x = new TreeSet<Double>(); // ERRORE
```

- Problema: `TreeSet<Double>()` non implementa `Polynomial`

Conversioni di Tipo

- **Sottotipo (<:)**
  - La relazione che permette di decidere quando è lecito convertire un tipo riferimento in un altro
  - Per il momento diciamo

> S <: T sse
> - S è una classe, T è una interfaccia
> - S implementa T.

- **Principio di sostituibilità**
  - Ovunque ci si aspetti un riferimento di un tipo è lecito utilizzare un rifetimento di un sottotipo

Continua…

Conversioni di Tipo

- **Assumiamo S <: T**
- **Regola di assegnamento**
  - Un riferimento di tipo S si puo sempre assegnare ad una variabile di tipo T
  - Un riferimento di tipo classe può sempre essere assegnato ad una variabile di tipo interfaccia che la classe implementa
- **Regola di passaggio di parametri**
  - Un riferimento di tipo S si puo sempre passare per un parametro di tipo T
  - Un riferimento di tipo classe può sempre essere passato per un parametro di tipo interfaccia che la classe implementa

Conversioni di Tipo

- **Le conversioni abilitate dalla regola**

  `C` <: `I` sse `C` implementa `I`

  sono corrette

- **Dimostrazione intuitiva:**
  - Se **`C`** implementa **`I`** , **`C`** deve definire tutti i metodi dichiarati da **`I`** (e dichiararli public)
  - Quindi tutti le invocazioni di metodo ammesse da **`I`** trovano effettivamente un metodo definito in **`C`**

Continua…

Polimorfismo – *dynamic dispatch*

- **Le regole del linguaggio garantiscono che una variabile di tipo interfaccia ha sempre come valore un riferimento di una classe che implementa l'interfaccia**
- **Se una interfaccia è implementata da più classi, il tipo dei valori legati alla variabile può cambiare**

```
Polynomial p;
p = new ListPoly(...);
p = new ArrayPoly(...);
```

Continua…

Polimorfismo – *dynamic dispatch*

- **Possiamo invocare ognuno dei metodi dell'interfaccia:**

```
int g = p.grado();
```

- **Quale metodo invoca?**

Polimorfismo – *dynamic dispatch*

```
int g = p.grado();
```

- **Se `p` riferisce un `ListPoly`, invoca il metodo `ListPoly.grado()`**
- **Se `p` riferisce un `ArrayPoly`, invoca il metodo `ArrayPoly.grado();`**
- **Polimorfismo (molte forme):**
  - il metodo invocato dipende dipende dal tipo del riferimento legato alla variabile

Continua…

Esempio

- **Costruiamo una applicazione per disegnare un insieme di forme geometriche contenute in una componente grafico:**
  - definiamo **`GWin,`** una classe che descrive un contenitore di forme geometriche disegnate mediante una invocazione del metodo **`paint()`**
  - per esemplificare, consideriamo due tipi di forme**:** **`Quadrato`** e **`Cerchio`**

Forme grafiche

```
class Quadrato
   {
      . . .
      public void draw(Graphics2D g)
      {
         // Istruzioni per il disegno
         . . .
      }
   }
```

```
class Cerchio
   {
      . . .
      public void draw(Graphics2D g)
      {
         // Istruzioni per il disegno
         . . .
      }
   }
```

GWin

- Un contenitore di `Quadrati` e `Cerchi`

```
/**
 Una finestra che contiene Quadrati e Cerchi
*/
class GWin
{
  /**
    Disegna tutte le forme di questo component
   */
   public void paint(){ /* disegna su g  */ }
   /**
    Componente grafica su cui disegnare
   */
   private Graphics2D g;
}
```

Domanda

- Che struttura utilizziamo per memorizzare le forme contenute nella `GWin`?
- Come definiamo il metodo `paint()` in modo che disegni tutte le forme della componente?

Risposte

- definiamo una nuova interfaccia: **`Shape`**

```
interface Shape { void draw(Graphics2D g); }
```

- Ridefiniamo le classi **`Quadrato`** e **`Cerchio`** in modo che implementino **`Shape`**
- Memorizziamo gli oggetti della componente in una **`ArrayList<Shape>`**

Shapes

```
class Quadrato implements Shape
   {
      . . .
      public void draw(Graphics2D g)
      {
         // Istruzioni per il disegno
         . . .
      }
   }
```

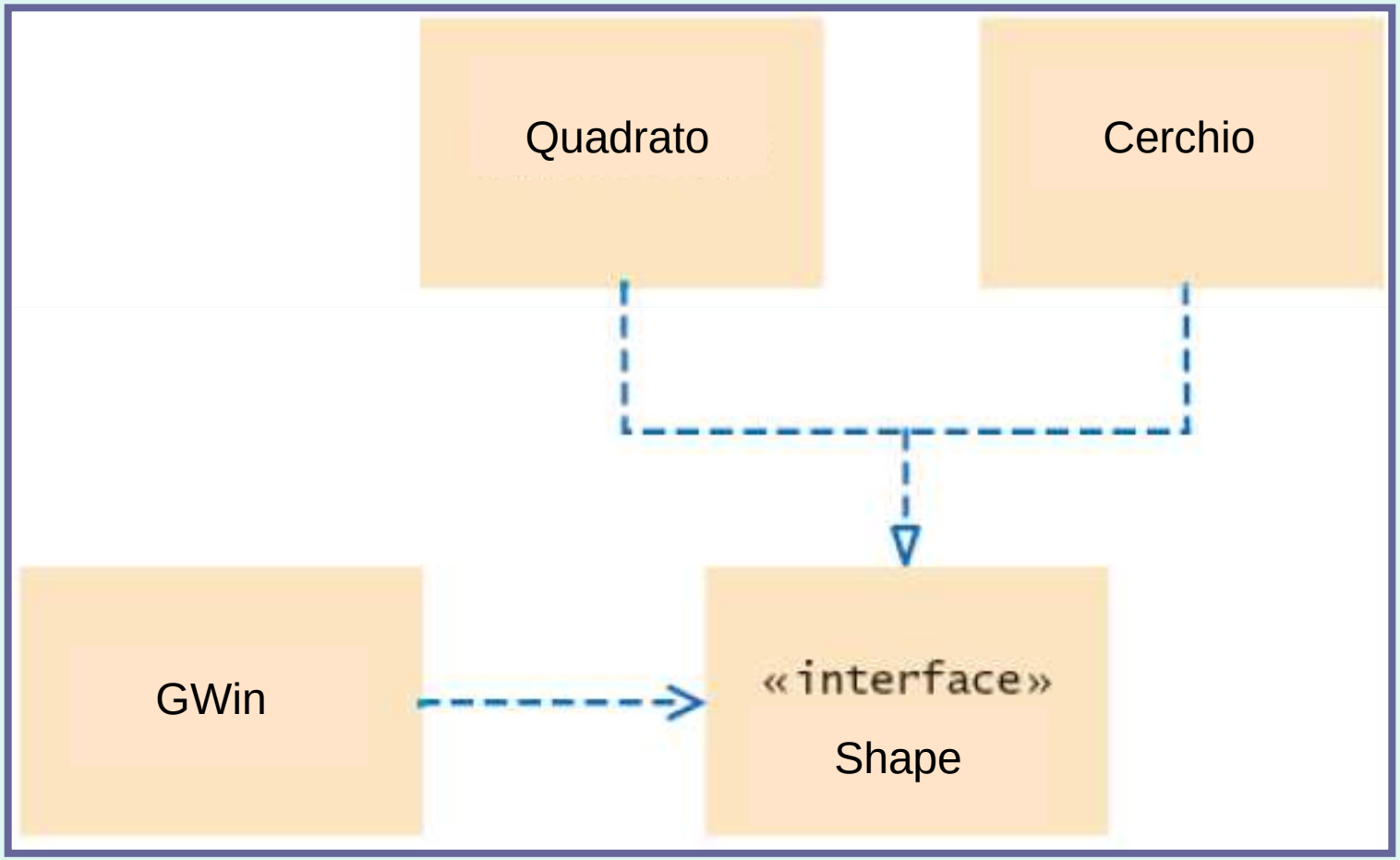
```
class Cerchio implements Shape
 {
      . . .
      public void draw(Graphics2D g)
      {
         // Istruzioni per il disegno
         . . .
      }
  }
```

GWin

Mantiene una `ArrayList<Shape>`

```
class GWin
{
    private Graphics2D g;
    private ArrayList<Shape> shapes;
    // crea una GWin con un insieme di forme
    public GWin(Shape... shapes)
    {
        Graphics2D g = new Graphics2D();
        this.shapes = new ArrayList<Shape>();
        for (Shape s:shapes) this.shapes.add(s);
    }
    // disegna tutte le componenti della GWin
    public void paint()
    {
        for (Shape s:shapes) s.draw(g);
    }
}
```

Diagramma delle Classi



Polimorfismo – *dynamic dispatch*

- ***Dynamic dispatch*:**
  - Il metodo da invocare per rispondere ad un messaggio è deciso a tempo di esecuzione