FACULTAD DE INGENIERIA
Universidad de Buenos Aires

Polimorfismo

Carlos Fontela
cfontela@fi.uba.ar

A3F

Temario

Métodos virtuales

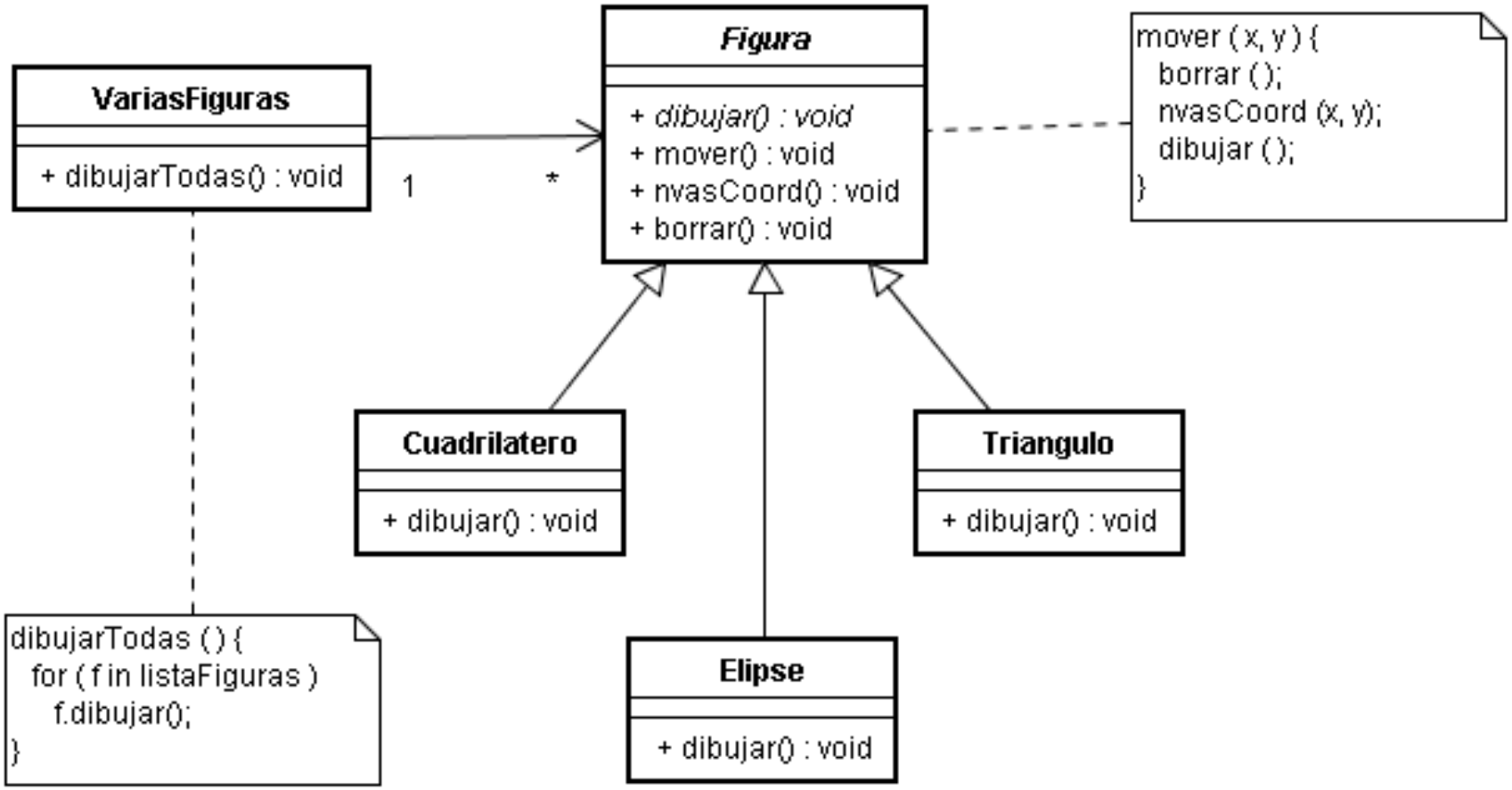
Métodos abstractos

Polimorfismo como concepto

Interfaces

Clases internas

Métodos virtuales (1)



Métodos virtuales (2)

```
// llama al dibujar de Elipse:
unaElipse.mover();
// llama al dibujar de Triangulo:
unTriangulo.mover();
// llama al dibujar de la clase de cada figura:
v.dibujarTodas();
// porque dibujar es virtual
```

En Java y en Smalltalk la “virtualidad” se da por defecto

En C# y C++ debemos declarar métodos como “virtual”

Métodos de clase, privados y finales no son virtuales

Métodos virtuales (3)

- Los métodos virtuales agregan ineficiencias
  - Pero garantizan reutilización
  - Eliminar la “virtualidad” sólo si se demuestra que no se van a redefinir y la presunta ineficiencia
- Un método debe ser virtual sí o sí cuando se lo redefinirá y es llamado desde:
  - Un método en una clase ancestro
  - Un método que delegue en el método en cuestión de la clase ancestro

Ejemplo estándar en C#

En Object

```
public virtual String ToString ( ) { … }
```

Console.WriteLine usa ToString()

Si quiero que un objeto sea imprimible, debo redefinir ToString:

En CuentaBancaria

```
public String ToString() {
   return ( numero + titular + saldo.ToString() ); }
```

Luego...

```
CuentaBancaria cuenta = new CtaCte(12,”Juan”);
Console.WriteLine(cuenta);
```

Métodos abstractos

Nunca van a ser invocados

- Caso del dibujar de Figura
- Pero se necesita para el compilador

No tiene implementación

Corolarios

- Debe redefinirse
- Debe ser virtual

En Java y C#:

- public void abstract dibujar();
- Si una clase tiene métodos abstractos debe ser abstracta

Polimorfismo

- Objetos de distintas clases de una misma familia entienden los mismos mensajes
  - Igual semántica (significado)
  - Implementaciones diferentes
  - Un mismo mensaje puede provocar la invocación de métodos distintos
- Vinculación tardía
  - Se retarda la decisión sobre el tipo del objeto hasta el momento en que vaya a ser utilizado el método

Interfaces: clases “muy abstractas”

Son como clases

- Abstractas
- Todos los métodos abstractos
- Sin atributos (sin estado)

Ejemplo

```
public interface Imprimible {
    void imprimir();     // público y abstracto por defecto
}
```

Podrían heredar de otras interfaces

Uso

```
public class CajaAhorro extends CuentaBancaria implements Imprimible {...}
```

Corolario

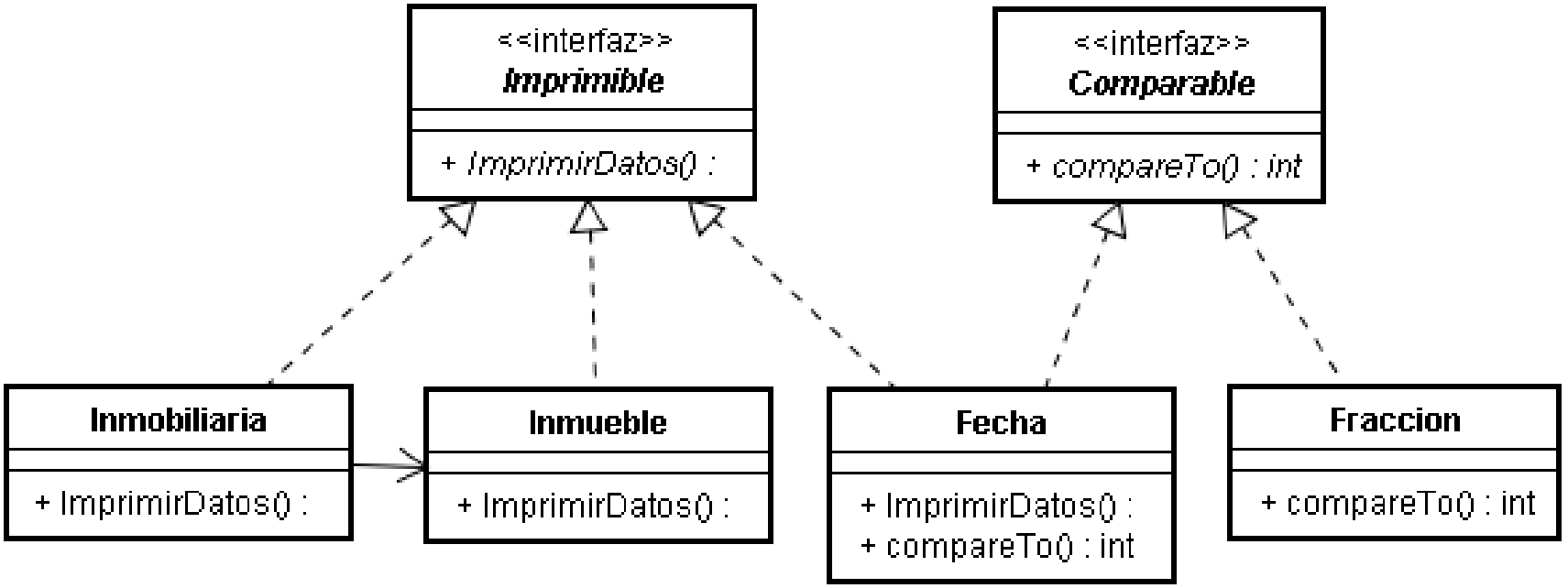
- Si una clase declara implementar una interfaz y no define uno de sus métodos es abstracta

Interfaces: protocolos

Son grupos de métodos sin implementar

Una clase puede implementar varias

- Ojo con los conflictos de nombres



Interfaces y polimorfismo (1)

Variables cuyo tipo es una interfaz

```
Imprimible i = new Fecha(20,6,1964);
Imprimible[ ] lista = new Imprimible[3];
lista[0] = new Fecha (20, 1, 2000);
lista[1] = new Inmueble ( );
lista[2] = new Fecha (8, 6, 2002);
…
for (int i=0; i < 3; i++) v[i].imprimir();
```

Ojo que sólo puedo instanciar clases

Interfaces y polimorfismo (2)

- Cada objeto puede tener muchas "caras"

```
Fecha f = new Fecha(20,6,1964);
Imprimible i = f;
Comparable c = f;
Serializable s = f;
```

- Todos se refieren al mismo objeto
  - Pero "lo ven" distinto
  - Cada variable sólo puede usar los métodos de su interfaz

¿Qué es una interfaz?

Visión de lenguaje

Una clase “muy abstracta” que se puede usar para herencia múltiple

Visión desde el uso

Un tipo de datos que permite que ver a un mismo objeto con distintos tipos

=> Cada tipo implica un comportamiento

Interfaces predefinidas

Caso de Comparable

En java.lang.Comparable

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

Devuelve valores <0, 0 o >0

¡Muy útil en colecciones!

Hay otras predefinidas

Comparator, Serializable, Cloneable, etc.

Uso de Comparable

```
public class Fraccion implements Comparable {
    private int numerador;
    private int denominador;

    // otros métodos
    public int compareTo(Object otro) {
      Fraccion otra = (Fraccion) otro;
      if (numerador * otra.denominador > denominador * otra.numerador)
            return 1;
      else if (numerador * otra.denominador < denominador * otra.numerador)
            return -1;
      else return 0;
    }
}
```

Comparable y arreglos

Clase Arrays: uso

```
Fraccion [ ] v = new Fraccion[4];

Arrays.sort (v);

int posicion = Arrays.binarySearch (v, x);
```

Método sort(): definición

```
public void sort (Comparable [ ] w) { … }
```

Si Fraccion implementa Comparable, puedo usar

Arrays.sort(v)

¿Y si no?

Rarezas: Comparator

Si la clase de v no implementa Comparable, existe otro sort():

public void sort (Object [ ] v, Comparator c) { … }

Que puedo usar así:

Comparator comp = new ComparadorFracciones();

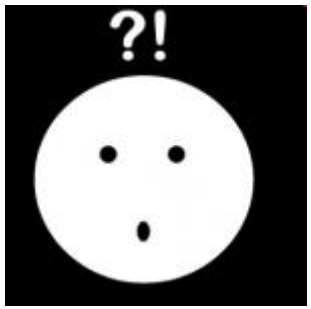
Arrays.sort (x, comp);

¿Y qué es ComparadorFracciones?

Una clase que implementa java.util.Comparator…

Y su método:

public int compare (Object o1, Object o2);



Implementación del comparador

```
public class ComparadorFracciones implements java.util.Comparator {
    public int compare (Object o1, Object o2) {
      Fraccion f1 = (Fraccion)o1; Fraccion f2 = (Fraccion)o2;
      if ( f1.getNumerador() * f2.getDenominador() >
                  f1.getDdenominador() * f2.getNumerador() )
            return 1;
      else if ( f1.getNumerador() * f2.getDenominador() <
                  f1.getDenominador() * f2.getNumerador() )
            return -1;
      else return 0;
       }
}
```

¿Qué hicimos?

Creamos una clase ¡que no tiene estado!

- ¡Y la instanciamos!

Tampoco se refiere a una entidad de dominio

- Aparece por necesidades de diseño (solución)

Sólo sirve por el método que lleva dentro

- Esto es un patrón de diseño: Command

Otro uso de Comparator:

- Para definir otra forma de ordenamiento

Clases internas

Pueden utilizar métodos y atributos privados de su clase externa (sin relación de herencia)

Pueden ser descendientes de cualquier clase visible de la clase externa

En una clase descendiente de la externa se puede declarar una clase interna que descienda de la interna del ancestro

Se usan poco

Clases dentro de métodos y anónimas (Java)

```
public class Externa {
    //...
    public void m ( ) {
     class Interna extends Base {
           // ...
     }
    Interna i = new Interna ( );
    // ...
    }
}
```

```
public class Externa {
    //...
    public Base m( ) {
      // ...
      return new Base( ) {
           // declaración de la
           // clase anónima ...
      }
    }
}
```

Claves

Métodos virtuales garantizan reutilización genuina

Interfaces definen qué puede hacer un objeto

Polimorfismo = distintos comportamientos para un mismo mensaje

Lectura obligatoria

Domain Driven Design, de Eric Evans, capítulo 1

Lo tienen en:
http://domaindrivendesign.org/sites/default/files/books/chapter01.pdf

Lecturas opcionales

Orientación a objetos, diseño y programación,
Carlos Fontela 2008, capítulos 7 y 8
"Polimorfismo basado en herencia" y
"Polimorfismo basado en interfaces"

Qué sigue

Smalltalk

Excepciones

Colecciones e iteradores

Primer parcial