

Programación 2

Polimorfismo e Interfaces

Profesores:

Ismael Figueroa - ifigueroap@gmail.com

Eduardo Godoy - eduardo.gl@gmail.com

¿Qué es el polimorfismo?

Definición

La palabra *polimorfismo* significa literalmente *muchas formas*. En programación, es una técnica para *programar de manera general*, procesando de manera uniforme objetos que comparten una misma superclase.

Ejemplo

Por ejemplo, un editor de dibujos trabaja con figuras en 2D: círculos y cuadrados. Estas figuras se pueden mover, mediante el método `mover`. ¿Cómo podemos implementar un método general para desplazar todas las figuras 100 pixeles hacia la derecha?

Código tentativo

```
public class Editor2D {  
  
    List<Cuadrado> cuadrados;  
    List<Circulo> circulos;  
  
    public void desplazar(int dx, int dy) {  
  
        for(Cuadrado c : cuadrados) {  
            c.mover(dx, dy);  
        }  
  
        for(Circulo c: circulos) {  
            c.mover(dx, dy);  
        }  
    }  
}
```

¿Qué pasaría ahora si agregamos nuevas figuras 2D: rectangulo, elipse, triangulo, trapezoide?

Código tentativo extendido

```
public class Editor2D {  
  
    List<Cuadrado> cuadrados;  
    List<Circulo> circulos;  
    List<Elipse> elipses;  
    List<Triangulo> triangulos;  
    List<Trapezoide> trapezoides;  
  
    public void  
    desplazar(int dx, int dy) {  
  
        for(Cuadrado c : cuadrados)  
        {  
            c.mover(dx, dy);  
        }  
  
        for(Circulo c: circulos) {  
            c.mover(dx, dy);  
        }  
    }  
}
```

```
        for(Elipse e: elipses) {  
            e.mover(dx, dy);  
        }  
  
        for(Triangulo t: triangulos) {  
            t.mover(dx, dy);  
        }  
  
        for(Trapezoide t: trapezoide) {  
            t.mover(dx, dy);  
        }  
    }  
}
```

Extensibilidad

El código no es extensible

Cada vez que agregamos una subclase en la jerarquía, tenemos que modificar el código que utiliza figuras 2D. No estamos ganando ningún beneficio por usar jerarquía de clases

Definición

Un software se considera extensible si para agregarle nueva funcionalidad se debe modificar una cantidad acotada de elementos, archivos, o clases. Es decir, se minimiza la cantidad de código a modificar ante la incorporación de nuevas clases o métodos.

Polimorfismo basado en Herencia

- En la programación orientada a objetos, y en Java en particular, el polimorfismo funciona explotando los elementos comunes de una jerarquía de herencia.
- La idea es aprovechar la relación *es un*. Entonces, el código se implementa de manera general haciendo referencia solamente al tipo de una superclase en específico.
- La gracia es que *las instancias de las subclases **también son instancias de la super clase***, y que se invoca el método “correcto”...

Tipo Aparente y Tipo Actual

Tipo Aparente

El *tipo aparente* de una variable es aquel que se puede determinar estáticamente, es decir, en tiempo de compilación.

Tipo Actual

El *tipo actual* de una variable es aquel que realmente toma cuando se construye la instancia, y que siempre se puede conocer durante la ejecución del programa.

Tipo Aparente y Tipo Actual

```
public class Main {  
    public static void main(String[] args) {  
        // tipo aparente: Base / tipo actual: A  
        Base b1 = new A();  
  
        // tipo aparente: Base / tipo actual: B  
        Base b2 = new B();  
  
        // tipo aparente: A / tipo actual: B  
        A a1 = ((A) b2);  
    }  
}
```


Dispatch dinámico de mensajes en Java

Definición

Java posee *dispatch dinámico de mensajes*. Esto significa que cuando se invoca un método, el código que se ejecuta corresponde ***al método del TIPO ACTUAL al que pertenece la instancia que ejecutará el método.***

Dispatch dinámico de mensajes en Java

```
abstract class Base {  
    abstract void saludar();  
}  
  
class A extends Base {  
    void saludar() { System.out.println("Hola A"); }  
}  
  
class B extends A {  
    void saludar() { System.out.println("Hola B"); }  
}  
  
public class Main {  
    public static void main(String[] args) {  
        Base b1 = new A(); Base b2 = new B();  
        b1.saludar(); b2.saludar();  
        ((A) b2).saludar();  
    }  
}
```

Polimorfismo = herencia + dispatch dinámico

La combinación de herencia y dispatch dinámico es la que permite el polimorfismo en Java.

Ejemplo Revisado

```
public class Editor2D {  
    List<Cuadrado> cuadrados;  
    List<Circulo> circulos;  
    List<Elipse> elipses;  
    List<Triangulo> triangulos;  
    List<Trapezoide> trapezoides;  
    /* .... */  
    public void desplazar(int dx, int dy) {  
        List<Figura2D> figuras = new ArrayList<Figura2D>();  
        figuras.addAll(cuadrados);  
        figuras.addAll(circulos);  
        figuras.addAll(elipses);  
        figuras.addAll(triangulos);  
        figuras.addAll(trapezoides);  
  
        for(Figura f : figuras) {  
            f.mover(dx, dy);  
        }  
    }  
}
```

Ejemplo Re-Revisado

```
public class Editor2D {  
    List<Figura2D> figuras;  
  
    /* .... */  
  
    public void desplazar(int dx, int dy) {  
        for(Figura f : figuras) {  
            f.mover(dx, dy);  
        }  
    }  
}
```

Determinando el tipo actual de un objeto

Problema

¿Cómo podríamos hacer para definir un método `contarCuadrados()`, que dada la lista de figuras podamos contar cuántas de éstas son cuadrados?

Determinando el tipo actual de un objeto

Problema

¿Cómo podríamos hacer para definir un método `contarCuadrados()`, que dada la lista de figuras podamos contar cuántas de éstas son cuadrados?

Solución

Java provee el operador `instanceof` para determinar si el tipo actual de un objeto es o no es igual a alguno que nosotros especifiquemos

Contando cuadrados

```
public class Editor2D {  
    List<Figura2D> figuras;  
  
    public int contarCuadrados() {  
        int total = 0;  
  
        for(Figura f : figuras) {  
            if(f instanceof Cuadrado) {  
                total = total + 1;  
            }  
        }  
  
        return total;  
    }  
}
```


Métodos y clases finales

Mediante la declaración `final` el programador puede indicar que:

- Un método no puede ser sobrescrito
- Una clase no puede ser extendida por una superclase. Implícitamente todos sus métodos son considerados `final`

Esto es fundamental para garantizar ciertos comportamientos, por ejemplo en seguridad. *¿Qué pasaría si puedo sobrescribir el método que verifica los passwords?*

Dispatch de métodos `final`

Los métodos `final` no utilizan dispatch dinámico: se invoca el método de acuerdo a la implementación final, que no puede sobrescribirse. Lo mismo pasa con los métodos privados que, implícitamente, también son finales.

¿Qué son las Interfaces?

En Java una interfaz es:

- Una especificación formal y verificada en compilación sobre un conjunto de valores constantes y métodos. Es similar a una clase abstracta, pero *no necesita que exista una jerarquía de clases*.
- Un *tipo de dato* que puede ser usado para programar con polimorfismo, en base a la especificación dada por la interfaz.
- Un mecanismo para implementar *ocultamiento de la información*. Esto porque solo se conoce la interfaz pública, pero no la implementación interna, la cual puede ser cambiada de manera transparente.
- Un mecanismo para simular *herencia múltiple*

Cómo se declara una interfaz

Consideremos la funcionalidad de que un objeto puede ser *archivable*, es decir, podemos guardarlo en un archivo de texto, y luego recuperar su valor.

```
public interface Archivable {  
    void archivar(String filePath);  
    void restaurar(String filePath);  
}
```

- El código define la interfaz `Archivable`
- La interfaz tiene el método `archivar` para guardar el estado del objeto en un archivo dado
- La interfaz tiene el método `restaurar` para recuperar su estado interno desde un archivo dado

¿Cuáles clases pueden ser Archivables?

¿Quién puede implementar esta interfaz?

Cualquier clase puede, eventualmente, implementar la interfaz `Archivable` o cualquier otra.

Ejemplo

Por ejemplo, la clase `Persona` y la clase `Rectangulo` pueden ser archivables! Esto sin importar que no estén relacionados por herencia

Una clase declara que va a implementar una interface usando la palabra clave `implements`

Implementando Persona Archivable

```
public class Persona implements Archivable {
    String nombre;
    String apellido;
    Integer edad;

    public Persona(String nombre, String apellido, Integer edad) {
        this.nombre = nombre;
        this.apellido = apellido;
        this.edad = edad;
    }

    public void archivar(String filePath) { // desde Archivable
        /* abrir archivo y escribir datos de nombre, apellido y edad */
    }

    public void restaurar(String filePath) { // desde Archivable
        /* abrir archivo y leer datos para nombre, apellido y edad */
    }
}
```

Implementando Rectángulo Archivable

```
public class Rectangulo extends Figura implements Archivable {
    Integer largo, ancho;

    public Rectangulo(Punto2D centro, Integer largo, Integer ancho) {
        /*constructor */
    }

    public Double area() { /* impl. area, desde Figura */ }

    public Double perimetro() { /* impl. perimetro, desde Figura */ }

    public void archivar(String filePath) { // desde Archivable
        /* abrir archivo y escribir datos de nombre, apellido y edad */
    }

    public void restaurar(String filePath) { // desde Archivable
        /* abrir archivo y leer datos para nombre, apellido y edad */
    }
}
```

Usando polimorfismo con interfaces

Podemos crear código general para objetos que sean Archivables:

```
void archivarTodo(List<Archivable> objs) {  
    int index = 0;  
    for(Archivable o : objs) {  
        o.archivar("archivo_" + index + ".txt");  
    }  
}
```

Usando polimorfismo con interfaces

También podemos usar `instanceof` para saber si un objeto implementa o no implementa una determinada interfaz:

```
{  
    Cuadrado c = new Cuadrado(40);  
    Rectangulo r = new Rectangulo(20, 30);  
  
    if(c instanceof Archivable) {  
        System.out.println("El objeto " + c + " es archivable");  
    }  
  
    if(r instanceof Archivable) {  
        System.out.println("El objeto " + r + " es archivable");  
    }  
  
    // instanceof puede ser verdadero en distintos casos  
    System.out.println(r instanceof Archivable);  
    System.out.println(r instanceof Rectangulo);  
    System.out.println(r instanceof Figura);  
}
```


Especificación vs Implementación

Especificación

Es una descripción formal sobre el comportamiento esperado por parte de una clase. Indica los métodos que se deben implementar, el tipo de retorno, el tipo de los parámetros

Herramientas para Especificación

Las **clases abstractas** y las **interfaces** son mecanismos para especificar los métodos que una clase o sub-clase **deben implementar**. En caso que falte alguna implementación, el código no va a compilar

Implementación

Una clase **concreta** que desea satisfacer una especificación debe implementar todos los métodos indicados por la especificación

Especificación Clases Abstractas con Interfaces

```
public abstract class
Figura2DArchivable extends Figura implements Archivable {
    /* las subclases deben implementar Figura y ademas Archivable */
}

public class RectanguloArchivable extends Figura2DArchivable {
    /* debe implementar los métodos de Figura: area, perimetro */
    /* debe implementar los métodos de Archivable: archivar, restaurar */
}
```

Preguntas

Preguntas?