Tema 7: Polimorfismo

Antonio J. Sierra

Índice

Introducción (I)

- Capacidad que tienen los objetos de una clase de responder al mismo mensaje o evento en función de los parámetros utilizados durante su invocación.
- Un objeto polimórfico es una entidad que puede contener valores de diferentes tipos durante la ejecución del programa.
- Se puede definir dos o más métodos dentro de la misma clase que tengan el mismo nombre, pero con sus listas de parámetros distintas. (métodos están **sobrecargados**).
- La sobrecarga de método la utiliza Java para implementar el Polimorfismo.

Introducción (I)

- El concepto de polimorfismo se puede aplicar tanto a **funciones** (**métodos**) como a **tipos de datos**. Así nacen los conceptos de funciones polimórficas y tipos polimórficos.
  - Las primeras son aquellas funciones que pueden evaluarse o ser aplicadas a diferentes tipos de datos de forma indistinta;
  - los tipos polimórficos, por su parte, son aquellos tipos de datos que contienen al menos un elemento cuyo tipo no está especificado.

Sobrecarga

- Es la posibilidad de tener dos o más **funciones** con el mismo nombre pero funcionalidad diferente. Es decir, dos o más funciones con el mismo nombre realizan acciones diferentes.
  - El compilador usará una u otra dependiendo de los parámetros usados. A esto se llama también sobrecarga de funciones.
  - Se generara un error si los métodos solo varían en el tipo de retorno.
- La sobrecarga de **operadores** da más de una implementación a un operador.

Ejemplo de sobrecarga de método

```
public class Articulo {
 private float precio;
 public void setPrecio(){
     precio = 3.50f;
 }
 public void setPrecio(
      float nuevoPrecio){
     precio = nuevoPrecio;
 }
 float getPrecio(){
     return precio;
 }
}
```

```
class Principal{
 public static void main(
           String args[]){
  Articulo obj =
          new Articulo ();
  obj.setPrecio();
  obj.setPrecio(4.f);

   System.out.println(
         "precio = " +
         obj.getPrecio());
 }
}
```

Sobrecarga de constructores

```
public class Caja {
   private double altura;
   private double anchura;
   private double profundidad;
   public Caja(double h, double a, double p) {         //constructor
      this.altura = h;
      this.anchura = a;
      this.profundidad = p;
   }
   public Caja() {         //constructor
    this.altura = this.anchura =  this.profundidad = -1;
   }
   public Caja(double dimension) {         //constructor
     this.altura = this.anchura =  this.profundidad = dimension;
   }
   public double volumen () {
      return this.altura * this.anchura * this.profundidad;
   }
}
```

Promoción de tipos con métodos sobrecargados

- Se aplican las reglas de promoción de tipos:
  - `byte` y `short` a `int`
  - Si algún parámetro es `long` -> `long`
  - Si algún parámetro es `float` -> `float`
  - Si algún parámetro es `double` -> `double`
  - El tipo `boolean` es incompatible con el resto y no promociona

Objeto como parámetro

- Por valor
  - Copia el contenido de la variable que queremos pasar en otra dentro del ámbito local de la subrutina.
  - La modificación de uno no afecta al otro.
- Por referencia
  - proporcionar a la subrutina a la que se le quiere pasar el argumento la dirección de memoria del dato.
  - En este caso se tiene un único valor referenciado (o apuntado) desde dos puntos diferentes, el programa principal y la subrutina a la que se le pasa el argumento, por lo que cualquier acción sobre el parámetro se realiza sobre el mismo dato en la memoria.

Por valor

```
class Clase {
   void Metodo (int i, int j){
     i *= 2;
     j /= 2;
   }
}
class LlamadaPorValor {
    public static void main( String args[]) {
     Clase ob = new Clase ();
     int a = 15, b = 20;
     System.out.println("a y b antes    " +a+ " "  +b);
     ob.Metodo (a, b);
     System.out.println("a y b despues  " +a+ " "  +b);
  }
}
```

Por referencia

```
public class ClaseReferencia {
    private int a, b;
    public ClaseReferencia (int i,
                            int j){
     this.a = i ;
     this.b = j ;
    }
    public void
   metodo(ClaseReferencia obj){
        obj.a *= 2;
        obj.b /= 2;
    }
    public int  getA(){  return
   this.a;    }
    public int  getB() {  return
   this.b;    }
}
```

```
class LlamadaPorReferencia {
  public static void main(
           String args[]) {
    ClaseReferencia ob =
     new ClaseReferencia (15, 20);

   System.out.println("a y b antes "
   + ob.getA() + " "  + ob.getB());
   ob.metodo (ob);
   System.out.println("a y b antes “
   + ob.getA() + " "  + ob.getB());
   }
}
```

Devolución de objetos

```
public class Prueba {
    private int a;

    public Prueba (int a){
        this.a = a;
    }

    public Prueba metodo(){
        Prueba local =
      new Prueba (this.a + 10);
        return local;
    }

    public int getA(){
        return this.a;
    }
}
```

```
class Ejemplo {
   public static void main(
         String args[]) {
      Prueba ob1 =
           new Prueba(20);
      Prueba ob2 = null;
   System.out.println("a antes    “
        + ob1.getA());
   ob2 = ob1.metodo ();
   System.out.println("a antes    “
        + ob2.getA());
   }
}
```

Recursividad

- **Recursión** es la forma en la cual se especifica un proceso basado en su propia definición.
  - Las instancias *complejas* de un proceso se definen en términos de instancias más *simples*, estando las **finales** más simples definidas de forma explícita.
- Aquellas funciones cuyo dominio puede ser recursivamente definido pueden ser definidas de forma recurrente.
  - $3! = 3 \cdot (3-1)! = 3 \cdot 2! = 3 \cdot 2 \cdot (2-1)! = 3 \cdot 2 \cdot 1! = 3 \cdot 2 \cdot 1 \cdot (1-1)! = 3 \cdot 2 \cdot 1 \cdot 0! = 3 \cdot 2 \cdot 1 \cdot 1 = 6$

Ejemplo de Recursividad

```
public class Factorial {
    public int fact (int n){
       int result = 1;
       if (n !=1) result = fact(n-1)*n;
       return result;
    }
}
class Recursividad {
   public static void main( String args[]) {
   Factorial f = new Factorial ();
   System.out.println("El factorial de 3 es  "  +
                      f.fact (3));
   System.out.println("El factorial de 4 es  "  +
                      f.fact (4));
 }
}
```

static

- Definición de un miembro de una clase que será utilizado independientemente de cualquier objeto de esa clase.
- `main` () es `static`.
- Las variables de instancia declaradas como `static` son, básicamente, variables globales.
- Restricciones:
- Los métodos `static`, sólo pueden llamar a otros métodos y datos `static`.
  - No se pueden referir a **this** o **super** de ninguna manera.

final

- Su contenido no pueda ser modificado.
- Debe inicializarse cuando se declara.

```
final static int NUEVO_ARCHIVO = 1;
final static int ABRIR_ARCHIVO = 1;
```

- Elegir identificadores en mayúsculas para las variables **`final`**.
- **`final`** se puede aplicar a métodos y clases.

Argumentos en la línea de comandos

- Los argumentos de la línea de órdenes es la información que sigue al nombre del programa en la línea de órdenes al ejecuta el programa.
- El acceso a los argumentos de la línea de órdenes se realiza mediante cadenas almacenadas en la matriz de **`String`** que se pasa a **`main()`**.

Ejemplo

```
public class LineaComandos {
  public static void main(String args[ ] ) {
  for (int i = 0; i<args.length; i++)
     System.out.println("args [ "+i +"] : "
                        + args [ i] );
  }
}
```