Functors

Versión orientada a objetos de los punteros a función

Introducción

En la elaboración de un programa a veces es necesario utilizar funciones *callback* (retrollamada). Este tipo de funciones presentan la peculiaridad de ser invocadas por el programa en un futuro, después de que se haya producido algún evento determinado. Para que esto funcione, el sistema debe reconocer de alguna manera que una función es del tipo callback y no una función ordinaria, por lo que se suele utilizar algún mecanismo de registro que permita a un sistema identificar a este tipo de funciones.

Existen multitud de casos en los que se emplean funciones de *callback*, siendo el ejemplo paradigmático el uso de aplicaciones GUI en sistemas conducidos por eventos. En este tipo de aplicaciones, suele ser habitual que al pulsar un botón o seleccionar una opción en un menú se lleve a cabo algún tipo de acción. Tal acción es la función de *callback*. Otro ejemplo sería el caso de la programación con AJAX (**A**synchronous **J**avascript **A**nd **X**ML) en aplicaciones web: cuando se realiza una petición AJAX desde el navegador hacia el servidor, indicamos que cuando llegue la respuesta del servidor ésta se pase a una función determinada que llevará a cabo su procesamiento. Esta función es del tipo *callback*.

En lenguajes como C y C++ se utiliza como mecanismo de *callback* punteros a función. Sin embargo, esto no es posible en Java, ya que ni existe el concepto de puntero ni está permitido crear funciones fuera de una clase. En Java es necesario crear una clase que desempeñe el rol de puntero a función, esto es, un *functor*.

Un *functor* es una clase que normalmente implementa un único método y cuyo único propósito es que su instancia haga las veces de puntero a función. Un objeto *functor* puede ser creado, pasado como parámetro y manipularse del mismo modo que lo haría un puntero a función en un lenguaje con soporte para éstos.

Ejemplo 1

Consideremos la siguiente clase, Utilidades.java, que dispone de un método para comparar dos objetos. Queremos que sea posible especificar el método de comparación en tiempo de ejecución,

por lo que añadimos un argumento del tipo Comparador al método:

Utilidades.java

```
package comparador;

public class Utilidades {
	public static int comparar(Object a, Object b, Comparador c) {
		return c.comparar(a, b);
	}
}
```

El objeto Comparador es un *functor*, ya que actúa como un puntero a la función comparar(). Para que sea posible disponer de diferentes tipos de comparadores necesitamos crear algún de jerarquía, y así beneficiarnos del polimorfismo. En este caso utilizaremos jerarquía de interfaces, por lo que creamos la siguiente interfaz:

Comparador.java

```
package comparador;

public interface Comparador {

	int comparar(Object a, Object b);

}
```

Muchas de las implementaciones de *functors* en Java implican el uso de interfaces de la manera en que se muestra en este ejemplo.

A continuación vemos el código para un comparador de números:

NumComparador.java

```
package comparador;

public class NumComparador implements Comparador {

	@Override
	public int comparar(Object a, Object b) {

		if (!(a instanceof Number) || !(b instanceof Number)) {
			throw new IllegalArgumentException("El tipo de los parametros
no es apropiado para este comparador");
		}

		int x = ((Number)a).intValue();
		int y = ((Number)b).intValue();
		if (x < y)
			return -1;
		else if (x > y)
```

```
                return 1;
            else
                return 0;
        }
}
```

Vamos ahora a crear una clase Alumno. Un objeto Alumno tendrá tres atributos: el código, el nombre y la fecha de nacimiento. Posteriormente crearemos un comparador capaz de comparar alumnos tomando su fecha de nacimiento como criterio:

Alumno.java

```
package comparador;

import java.text.Format;
import java.text.SimpleDateFormat;
import java.util.Date;

public class Alumno {

	private String codigo;
	private String nombre;
	private Date fechaNacimiento;

	public Alumno(String codigo, String nombre, Date fechaNacimiento) {
		this.codigo = codigo;
		this.nombre = nombre;
		this.fechaNacimiento = fechaNacimiento;
	}

	public String getCodigo() { return codigo; }
	public String getNombre() {  return nombre; }
	public Date getFechaNacimiento() { return fechaNacimiento; }

	@Override
	public String toString() {
		return "Alumno [codigo=" + codigo + ", nombre=" + nombre
				+ ", fechaNacimiento=" +
getFormatoBonito(fechaNacimiento) + "]";
	}

	private String getFormatoBonito(Date fechaNacimiento2) {
		Format fmt = new SimpleDateFormat("dd/MM/yyyy");
		return fmt.format(fechaNacimiento);
	}

}
```

El comparador para la clase Alumno queda como sigue:

AlumnoComparador.java

```
package comparador;

import java.util.Date;

public class AlumnoComparador implements Comparador {

	@Override
	public int comparar(Object a, Object b) {

		if (!(a instanceof Alumno) || !(b instanceof Alumno)) {
			throw new IllegalArgumentException("El tipo de los parametros
no es apropiado para este comparador");
		}

		Date fecha1 = ((Alumno) a).getFechaNacimiento();
		Date fecha2 = ((Alumno) b).getFechaNacimiento();

		if (fecha1.before(fecha2)) // fecha1 es mayor
			return 1;
		else if (fecha1.after(fecha2)) // fecha1 es menor
			return -1;
		else
			return 0;
	}

}
```

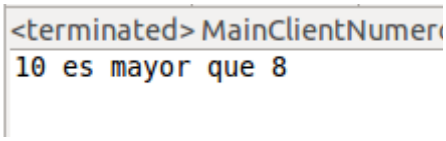
Ahora, para no mezclar código, podemos crearnos dos clases cliente. Una para utilizar el comparador de números y otra para crear alumnos y su comparador.

MainClientNumeros.java

```
package comparador;

public class MainClientNumeros {

	public static void main(String[] args) {

		int a = 10, b = 8;

		Comparador c = new NumComparador();

		int resultado = Utilidades.comparar(a, b, c);

		if (resultado == -1 ) {
			System.out.println(a + " es menor que " + b);
		} else if (resultado == 1 ) {
			System.out.println(a + " es mayor que " + b);
		} else {
			System.out.println(a + " es igual a " + b);
```

```
		}
	}

}
```

Salida:



MainClientAlumnos.java

```
package comparador;

import java.util.Calendar;
import java.util.Date;

public class MainClientAlumnos {

	public static void main(String[] args) {

		Alumno alumno1 =
			new Alumno("1000-X", "Javier Garcia", getFecha(1, 9, 1980));

		Alumno alumno2 =
			new Alumno("1001-S", "Maria Navarro", getFecha(24, 4, 1979));

		Comparador comparador = new AlumnoComparador();

		int resultado = Utilidades.comparar(alumno1, alumno2, comparador);

		if (resultado == -1 ) {
			System.out.println(alumno1 + "\nes menor que\n" + alumno2);
		} else if (resultado == 1 ) {
			System.out.println(alumno1 + "\nes mayor que\n" + alumno2);
		} else {
			System.out.println(alumno1 + "\nes igual a\n" + alumno2);
		}
	}

	private static Date getFecha(int dia, int mes, int anyo) {
		Calendar calendar = Calendar.getInstance();
		calendar.set(Calendar.DAY_OF_MONTH, dia);
		calendar.set(Calendar.MONTH, mes - 1); // cosas que tiene Calendar!
		calendar.set(Calendar.YEAR, anyo);
		return calendar.getTime();
	}

}
```

Salida:

```
<terminated> MainClientAlumnos [Java Application] /usr/lib/jvm/java-6-openjdk/bin/jav
Alumno [codigo=1000-X, nombre=Javier Garcia, fechaNacimiento=01/09/1980]
es menor que
Alumno [codigo=1001-S, nombre=Maria Navarro, fechaNacimiento=24/04/1979]
```

Notad que el criterio para comparar alumnos ha sido sus respectivas fechas de nacimiento.

Ejemplo 2

Como se mencionó anteriormente en la introducción, las funciones de *callback* son utilizadas abundantemente en las GUI. Por ejemplo, cuando se selecciona una opción en un menú o se pulsa un botón se produce un evento de tipo *action* que es enviado a todos los *listeners* que se hayan registrado para atender a ese tipo de eventos. Cualquier clase puede ser un *listerner*, tan sólo debe implementar la interfaz ActionListener. El hecho de implementar esa interfaz obliga a implementar el único método declarado en ella: actionPerformed(). El método actionPerformed() es una función de *callback* implementada por un objeto *listener*, que será invocado por una opción de menú o por un botón (esto realmente es el comportamiento del patrón Observer).

Veamos a continuación una sencilla aplicación GUI basada en la librería AWT. Se trata de tanto sólo una clase que crea una ventana con dos botones: uno para realizar virtualmente la tirada de un dado y otro para cerrar la ventana y finalizar el programa.

EjemploAWT.java

```
package gui1;

import java.awt.Button;
import java.awt.Frame;
import java.awt.Label;
import java.awt.Panel;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import java.util.Random;

public class EjemploAWT extends Frame implements ActionListener {

	private static final long serialVersionUID = 1L;

	private Button btnTirarDado = new Button("Tirar el dado");
	private Button btnSalir = new Button("Salir");
	private Label lblTirada = new Label("                  ");

	public EjemploAWT() {
```

```
		super("Ejemplo AWT");
		crearGUI();
	}

	public static void main(String[] argv) {
		EjemploAWT ejemplo = new EjemploAWT();
		ejemplo.setSize(400, 75);
		ejemplo.setVisible(true);
	}

	private void crearGUI() {
		Panel panelBase = new Panel();
		panelBase.add(btnTirarDado);
		panelBase.add(btnSalir);
		panelBase.add(lblTirada);
		add(panelBase);

		/*
		 * Informamos a los botones que el listener que tratará el evento
		 * de accion (action) será la instancia de esta propia clase (this)
		 */
		btnTirarDado.addActionListener(this);
		btnSalir.addActionListener(this);
		pack();
	}

	/*
	 * Manejar las acciones GUI
	 * Esta es la función de callback
	 */
	@Override
	public void actionPerformed(ActionEvent event) {
		Object src = event.getSource();
		if (src == btnTirarDado)
			tirarDado();
		else if (src == btnSalir)
			System.exit(0);
	}

	private void tirarDado() {
		int tirada = new Random().nextInt(6) + 1;
		lblTirada.setText("Resultado: " + tirada);
	}
}
```

Salida:

El programa funciona. Es muy simple pero cumple su cometido. No obstante, hay un par de cosas que se consideran malas prácticas:

- Lo primero: nuestra clase, EjemploAWT, hace muchas otras cosas más a parte de proporcionar el método actionPerformed(), que es el método de *callback* para el

tratamiento de los botones. Esto no es adecuado, ya que debemos diseñar clases que tengan una cohesión lo más alta posible, es decir, que los métodos que implemente sean lo más coherente posibles con la abstracción que represente la clase y con la funcionalidad que esta debe ofrecer. Lógicamente, a los botones les es indiferente dónde estén definidos sus *listeners*; ellos sólo ven el método actionPerformed() que corresponde al objeto de retrollamada. Sin embargo, un programador debe tener la sensación de que la funcionalidad no se ha encapsulado correctamente en esta clase. Quizás, una mejor opción sea tener una clase diferente sólo para la función de *callback*.

- Lo segundo: La implementación realizada del método actionPerformed() es un tanto molesta. EjemploAWT se ha registrado como *listener* de sólo dos botones, pero supongamos que el ejemplo presentado tuviera unos cuantos botones más y un menú con varios elementos. Esto implicaría un método actionPerformed() encargado de manejar eventos *action* para todos estos botones y para el menú. Probablemente, esto se implementaría con una larga estructura condicional i*f-else if*, por lo que sería incómodo de entender y propenso a errores de mantenimiento. Visto esto, parece una mejor opción tener una clase de *callback* por cada botón de la aplicación. Pero en este último caso tenemos otro problema: al llevar los métodos de *callback* a sus propias clases ¿cómo acceden a los métodos y atributos de EjemploAWT? La solución a esto es utilizar clases anónimas.

Los siguiente fragmentos de código muestran los cambios necesarios en EjemploAWT para obtener un mejor diseño que el visto anteriormente.

Primero: ahora ya no es necesario que EjemploAWT implemente la interfaz ActionListener, por lo que nos aseguramos que la declaración de la clase quede como sigue:

```
public class EjemploAWT extends Frame {
```

Segundo, eliminamos por completo el método actionPerformed(), puesto que ya no es necesario, y añadimos el siguiente código en el método crearGUI():

```
private void crearGUI() {
	Panel panelBase = new Panel();
	panelBase.add(btnTirarDado);
	panelBase.add(btnSalir);
	panelBase.add(lblTirada);
	add(panelBase);

	/*
	 * Se utilizan instancias de clases anidadas anónimas como
	 * objetos callback para el manejo de los eventos de los botones
	 */
	btnTirarDado.addActionListener(new ActionListener() {
		@Override
```

```
            public void actionPerformed(ActionEvent e) {
                  tirarDado();
            }

      });
      btnSalir.addActionListener(new ActionListener() {
            @Override
            public void actionPerformed(ActionEvent e) {
                  System.exit(0);
            }

      });
      pack();
}
```

Estas instancias de clases anónimas son *functors*. Son objetos actuando como punteros a funciones.