Técnico en

DESARROLLO DE SOFTWARE

Programación Orientada a Objetos I

Programación Orientada a Objetos I

Unidad III

1. Clases y métodos abstractos

Cuando tenemos objetos que pueden clasificarse en muchos tipos, pero estos no tienen acciones en común, entonces utilizamos clases y métodos abstractos. Otra funcionalidad para la cual podemos utilizar clases y métodos abstractos es para poder agrupar un conjunto de objetos que tienen acciones en común o que tienen algo en común. Un motivo para el cual podemos necesitar agrupar objetos es para crear arreglos de un objeto y poder manipularlos como un solo tipo de objeto padre.

Una clase puede ser declarada como abstracta a pesar de no tener métodos abstractos. Pero al declararla como abstracta se está indicando que la implementación está incompleta, por lo que se está utilizando como clase padre para una o más subclases para que éstas completen su implementación. Por lo mismo, estas clases no pueden ser instanciadas.

Sintaxis

A continuación vamos a mostrar la sintaxis de una clase abstracta:

```
public abstract class NombreClase {

        public abstract TipoQueDevuelve  NombreMetodo1(parametros);

        public abstract TipoQueDevuelve  NombreMetodo2(parametros);

}
```

Notemos que en lugar de definir la funcionalidad de la declaración de nuestros métodos, vamos a colocar punto y coma.

La sintaxis de las clases que van a heredar de nuestras clases abstractas se deberían de mandar a llamar de la siguiente forma:

```
public class NombreClaseHija extends NombreClase {
   tipoAtributo atributo1;
   tipoAtributo atributo2;
   tipoAtributo atributoN;

   public NombreClaseHija (Parametros) {
        //Inicializacion de parametros
   }

   public TipoQueDevuelve  NombreMetodo1(parametros) {
//Ejecutar acciones de NombreMetodo1
   }
```

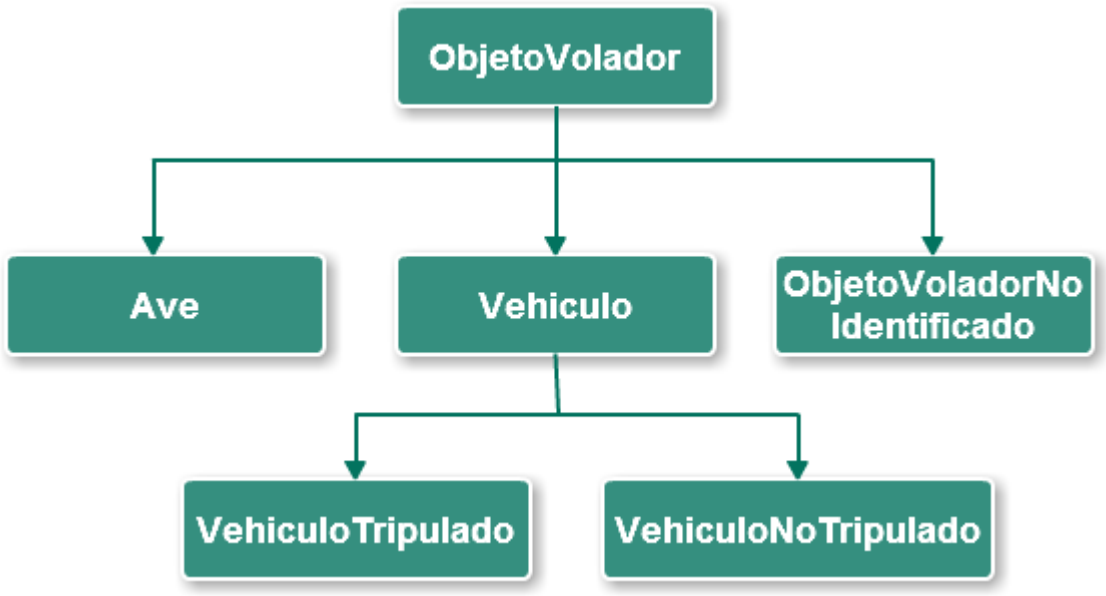
```
  public TipoQueDevuelve NombreMetodo2(parametros) {
//Ejecutar acciones de NombreMetodo2
  }
}
```

Ejemplo

Si tenemos nuestra clase ***ObjetoVolador***, al querer abstraer sus atributos y métodos podemos darnos cuenta que nuestro objeto está tan generalizado que no tenemos atributos tan específicos porque no todos tienen alas, no todos tienen turbinas, etc. Las acciones básicas de nuestro objeto ***ObjetoVolador*** son: *volar()*, *aterrizar()*, *cambiarVelocidad(int nuevaVelocidad)*.

Por lo mismo, nuestro objeto ***ObjetoVolador*** no puede crearse como una clase común y corriente, por lo tanto nuestra clase será definida como una clase abstracta. De igual forma nuestros métodos tampoco pueden definirse al 100% ni detallarse porque los distintos objetos voladores llevan a cabo acciones muy distintas en el proceso aunque el final sea el mismo.

A continuación nuestra clase de ejemplo:



Si definimos esta estructura de nuestra clase ***ObjetoVolador***, definitivamente nuestros objetos Ave, Vehiculo y ***ObjetoVoladorNoIdentificado*** no tienen mayor acción en común, más que todas vuelan y todas pueden parar o cambiar la velocidad.

Al momento de trasladar nuestro ejemplo a Java, nuestra clase ***ObjetoVolador*** quedaría definido de la siguiente forma:

```
public abstract class ObjetoVolador {
        public abstract void Volar();
        public abstract void Aterrizar();
        public abstract void cambiarVelocidad(int nuevaVelocidad);
}
```

Por cuestiones de ejemplo mostraremos cómo quedaría la clase hija ***Ave***:

Clase Ave

```
public class Ave extends ObjetoVolador {
  int velocidad;
  String nombre;
  String especie;
  String tamanio;
        public Ave (String nombre, String especie, String tamanio) {
                this.nombre = nombre;  //utilizamos la palabra reservada
                this.especie = especie;  //this para referirnos a un parámetro
                this.tamanio = tamanio;  //o método de la clase actual
                this.velocidad = 0;
        }
        public void Volar() {
                //La velocidad aumenta poco a poco hasta alcanzar una
                //velocidad constante. Este método debería variar dependiendo
                //de la especie y dependiendo del ave.
                //Para cuestiones de ejemplo diremos que el ave inicia en 0km/hora
                //y aumenta la velocidad gradualmente de 10 en 10Km/hora hasta
llegar
                //a los 50km/hora.
                System.out.println("El  ave de especie " + this.especie + " con nombre
```

```
" + this.nombre + " y de tamanio " + this.tamanio + " ha iniciado el vuelo.");
                this.cambiarVelocidad(10);
                this.cambiarVelocidad(20);
                this.cambiarVelocidad(30);
                this.cambiarVelocidad(40);
                this.cambiarVelocidad(50);
        }

        public void Aterrizar() {
                this.cambiarVelocidad(40);
                this.cambiarVelocidad(30);
                this.cambiarVelocidad(20);
                this.cambiarVelocidad(10);
                this.cambiarVelocidad(0);
                System.out.println("El  ave ha aterrizado");
        }

        public void cambiarVelocidad(int nuevaVelocidad) {
                System.out.println("Cambiando  velocidad de " + this.velocidad + " a "
+ nuevaVelocidad + ".");
                this.velocidad = nuevaVelocidad;
        }
}
```

Clase Main de Ave

```
/*
*  Clase Main
*/


public class Main {


	public static void main(String[] args) {
		ObjetoVolador canario = new Ave("Pauline","Canario","pequenio");
		canario.Volar();
		canario.Aterrizar();
	}
}
```

Resultado

```
>java Main


El ave de especie Canario con nombre Pauline y de tamanio pequenio ha iniciado el vuelo.
Cambiando velocidad de 0 a 10.
Cambiando velocidad de 10 a 20.
Cambiando velocidad de 20 a 30.
```

```
Cambiando velocidad de 30 a 40.
Cambiando velocidad de 40 a 50.
Cambiando velocidad de 50 a 40.
Cambiando velocidad de 40 a 30.
Cambiando velocidad de 30 a 20.
Cambiando velocidad de 20 a 10.
Cambiando velocidad de 10 a 0.
El ave ha aterrizado
```

Existen ciertas reglas que debemos seguir al definir clases y métodos abstractos:

- Debemos colocar la palabra “abstract” a toda clase o método que queramos definir como abstracta.
- En una clase abstracta sólo se definen los métodos, pero no se implementan.
- Si una clase subclase de una clase abstracta no implementa todos los métodos de dicha clase, entonces esa clase es una clase abstracta y debe ser definida como una clase abstracta.
- Todas las clases abstractas deben ser públicas.
- Una clase abstracta no puede ser instanciada.

Recordatorio

Una instancia es una variable de tipo objeto. Una clase puede tener muchas instancias y cada una es un objeto diferente.

2. Interfaces

Las interfaces nos sirven para definir los atributos y métodos de un objeto. Se diferencian de las clases abstractas con que las clases abstractas no se implementan, sólo se heredan y las interfaces si se pueden implementar. Las interfaces sirven como diccionarios para poder saber qué recibe cada método y qué atributos tiene dicha clase. Todas las interfaces son públicas. Al definir una interfaz, sustituimos la palabra *class* por *interface*.

Muchas veces se utilizan las interfaces para definir la base de un objeto y al implementarlas se aprovecha para heredar de otras clases y poder definir una nueva clase hija mejorada la cual no solo tendrá métodos y atributos sino que la definición de nuevos métodos que lo complementan. Más adelante veremos un ejemplo de esto.

Sintaxis

```
public interface NombreObjeto {
        public TipoQueDevuelve NombreMetodo1(parametros);
        public TipoQueDevuelve NombreMetodo2(parametros);
        public TipoQueDevuelve NombreMetodoN(parametros);
}
```

La sintaxis de las clases que van a heredar de nuestras clases abstractas se deberían de mandar a llamar de la siguiente forma:

```
public class NombreClaseHija implements NombreClase {
  tipoAtributo atributo1;
  tipoAtributo atributo2;
  tipoAtributo atributoN;

  public NombreClaseHija (Parametros) {
      //Inicializacion de parametros
  }

  public TipoQueDevuelve NombreMetodo1(parametros) {
//Ejecutar acciones de NombreMetodo1
  }

  public TipoQueDevuelve NombreMetodo2(parametros) {
//Ejecutar acciones de NombreMetodo2
  }
}
```

Existen ciertas reglas que debemos seguir al definir interfaces:

- Todos los métodos de una interfaz están implícitamente creados como métodos abstractos, por lo tanto al finalizar su inicialización es necesario colocar punto y coma en lugar de la implementación del mismo.
- Por convención no se coloca la palabra “abstract” en los métodos de las interfaces.
- Todos los métodos de las interfaces y las interfaces son públicas y se puede omitir la palabra *public* al definirlos.
- Una interfaz puede heredar de una o más interfaces y puede heredar de otra clase.

Ejemplo

Si tenemos nuestro objeto Teléfono, y queremos crear una definición de todas las acciones que tiene nuestro objeto para tenerlo como referencia o simplemente consideramos que no estamos listos para definir todos los métodos, entonces vamos a crear nuestra interfaz Teléfono de la siguiente forma.

```
public interface Telefono {
  public void llamar(int Telefono);
  public double colgar();
  public double obtenerCobro();
  double obtenerTiempoLlamadas();
}
```

Para implementar nuestro objeto Teléfono, lo haremos de la siguiente forma:

```
import java.util.Random;


public class Telefono_impl implements Telefono {


  double tiempoLlamadas;

  double precioLlamada;

  double totalLlamadas;


  public Telefono_impl () {

        this.precioLlamada = 0.25;

  }


  public void llamar(int Telefono) {

        System.out.println("Iniciando llamada al telefono " + Telefono);

        double total_llamada = colgar();

        this.tiempoLlamadas = this.tiempoLlamadas + total_llamada;

        double precio_llamada = total_llamada*this.precioLlamada;

        System.out.println("La llamada ha terminado y tuvo un costo de Q." +
String.format("%.2f", precio_llamada));

  }

  public double colgar() {

        Random rand_num = new Random();

        return rand_num.nextInt(100)+1;
```

```
  }
  public double obtenerCobro() {
        return this.precioLlamada * this.tiempoLlamadas;
  }
  public double obtenerTiempoLlamadas() {
        return this.tiempoLlamadas;
  }
}
```

Y la clase Main que tiene su método main que manda a llamar a su clase es la siguiente:

```
/*
*  Clase MainTelefono
*/

public class MainTelefono  {

        public static void main(String[] args) {
                Telefono_impl llamadaCasa = new Telefono_impl();
                llamadaCasa.llamar(37465091);
                System.out.println("Total  hasta el momento: " + String.format("%.2f",
llamadaCasa.obtenerCobro()) + " y tuvo una duración de " + String.format("%.2f",
```

```
llamadaCasa.obtenerTiempoLlamadas()));
        }
}
```

Al ejecutarlo se mostrará algo similar a lo siguiente:

```
>java MainTelefono

Iniciando llamada al telefono 37465091
La llamada ha terminado y tuvo un costo de Q.3.25
Total hasta el momento: 3.25 y tuvo una duracion de 13.00
```

3. Clases envolventes

En general, una clase envolvente es una clase cualquiera que se encarga de envolver o encapsular funcionalidad de otra clase o componente.

También existe la funcionalidad en la que las clases envolventes proveen una forma de utilizar los tipos de datos primitivos como objetos. Como vimos en la primera unidad, por cada dato primitivo tenemos una clase envolvente para él. Por ejemplo, para int Integer, byte Byte, long Long, char Character, etc.

	Nombre	Tipo	Tamaño	Rango
Tipos Primitivos (No poseen métodos, no son objetos y no necesitan de una invocación para ser creados)	byte	Entero	1 byte	-128 a 127
	short	Entero	2 bytes	-32,768 a 32,767
	int	Entero	4 bytes	-2^{31} a $2^{31}-1$
	long	Entero	8 bytes	-2^{63} a $2^{63}-1$
	float	Decimal simple	4 bytes	floating point de 32 bits
	double	Decimal doble	8 bytes	floating point de 64 bits
	char	Carácter simple	2 bytes	'\u0000' o 0 al '\uffff' o 65,535
	boolean	Booleano	1 byte	true o false
Tipos Objeto (Tienen métodos y necesitan de una invocación para ser creados)	Tipos de la biblioteca estándar de Java	String, ArrayList, TreeSet, entre otros.		
	Tipos definidos por el programador / usuario	Perro, Gato, Ballena, Alumno, Profesor, o el que ustedes vayan a definir en sus tareas.		
	arrays	Serie de elementos o formación tipo vector o matriz.		
	Tipos wrapper: Equivalentes a los tipos primitivos pero como objetos.	Byte		
		Short		
		Integer		
		Long		
		Float		
		Double		
		Character		
		Boolean		

Ejemplo para clases primitivas

```
int numero = 22;

Integer numeroEntero = new Integer(number); //Ahora tenemos nuestro entero 22

//convertido en un objeto entero


int valorEntero = numeroEntero.intValue();  //y para obtener el valor de nuestro
entero

// convertido a un objeto entero debemos de utilizar un método del objeto Integer.
```

También se pueden transformar datos de un tipo a otro de la siguiente forma:

```
double numero1 = 21.5;

int numeroEntero = (int) numero1;

System.out.println(numero1);

System.out.println(numeroEntero);
```

Al ejecutarlo va a devolver lo siguiente:

```
21.5


21
```

A esto lo conocemos como *casting*.

Referencias

- API de Java: https://docs.oracle.com/javase/7/docs/api/
- Java in a Nutshell - Benjamin J. Evans & David Flanagan: http://www.ebooksbucket.com/uploads/itprogramming/java/Java_in_a_Nutshell_6th_Edition.pdf