Rev 1.1 20150216

Programación II

Tema 1. Clases y objetos

Contenido

Programación II....1
Tema 1. Clases y objetos....1
1Introducción....2
2Clases y objetos....2
2.1Diagrama representativo de una clase....2
2.2Implementación en Java de clases....3
2.3Los distintos tipos de datos....4
2.4Objetos....5
3Visibilidad básica....5
4Constructores....6
5Getters....7
6El método *toString()*....8
7Constantes....9
7.1Constantes complejas....12
8La referencia *this*....13
9La clase Punto....15

1 Introducción

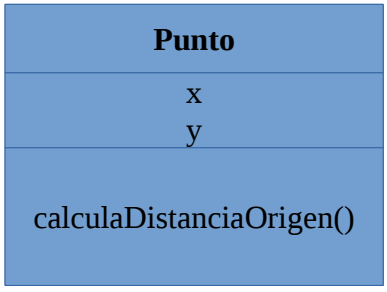
El desarrollo de la Programación Orientada a Objetos (POO), empieza a destacar durante la década de los 80, partiendo de la programación estructurada, a la que engloba, y dotando al programador de un enfoque nuevo: en lugar de tratar de dividir y estructurar el programa de manera *top-down*, es decir, de arriba a abajo, el enfoque consiste en identificar las entidades en el problema a resolver, y simular la forma de interaccionar entre ellas.

2 Clases y objetos

Una clase permite describir objetos similares mediante la definición de sus estructuras de datos y métodos comunes. Las clases son plantillas para objetos, permitiendo la agrupación de objetos que comparten las mismas propiedades y comportamiento. Así, existen puntos cartesianos en distintas posiciones, pero todos ellos pertenecen a la clase **Punto**, que define los atributos comunes para todos *x* e *y*, a la vez que el método *calculaDistanciaOrigen()* (que consiste en calcular la raíz cuadrada de la suma de *x* e *y* al cuadrado), también común para todos.

2.1 Diagrama representativo de una clase

Una forma muy estandarizada de representar clases es utilizar la notación del *Unified Modelling Language* (UML), de la que en esta asignatura se verá una parte muy pequeña. En concreto, para representar una clase, se utiliza un rectángulo dividido en tres partes. En la superior se coloca el nombre de la clase, en la intermedia una lista de los atributos, y finalmente, en la inferior, una lista de métodos.



2.2 Implementación en Java de clases

Desde un punto de vista totalmente pragmático y funcional, y conociendo lenguajes de programación como C, se puede entender una clase como una estructura a la que se añaden funciones que actúan directamente sobre los atributos.

```
class Punto {
	int x;
	int y;

	double calculaDistanciaOrigen() {
		return Math.sqrt( ( x * x ) + ( y * y ) );
	}
}
```

Así, en el ejemplo anterior el método (la función de la clase) *calculaDistanciaOrigen()* no necesita ningún parámetro, pues toda la información que necesita ya está disponible en la propia clase. Para calcular la raiz cuadrada se utiliza el método *static sqrt()* de la clase **Math**. Los métodos *static* se verán más adelante.

Cuando se codifica una clase, el nombre de esta se escribe con la inicial en mayúsculas. Dentro del cuerpo de la clase, delimitado por llaves, se indican primero los atributos, y a continuación los métodos. Tanto los nombres de los atributos como los de los métodos comienzan por minúscula, y en caso de emplear varias palabras, estas se concatenan con su inicial en mayúscula.

2.3 Los distintos tipos de datos

Al igual que en otros lenguajes cuya sintaxis está basada en C, los atributos en Java se crean indicando primeramente el tipo, y a continuación, el nombre del mismo. Los tipos más importantes disponibles en Java aparecen en la tabla.

Tipo	Explicación	Tamaño
int	Números enteros.	32bits -2147483648 a 2147483647
long	Números enteros de precisión doble.	64bits -9223372036854775808 a 9223372036854775807
double	Números reales de precisión doble. Norma IEEE 754.	64bits $\pm 1.8x10^{-308}$ a $\pm 1.8x10^{308}$
char	Carácter	16bits. Unicode, UTF-16.
String	Texto o cadena de caracteres.	N/A
boolean	Booleano: acepta valores **true**, **false**.	N/A

2.4 Objetos

Una vez que ya está disponible la clase[1], es posible crear objetos a partir de ella. Para ello, se utiliza el operador **new**, seguido del nombre de la clase y dos paréntesis. Por ejemplo:

```
> Punto p = new Punto();
> p.x
(int) 0
```

Así, se crea una referencia, y se la hace apuntar a un nuevo objeto de la clase **Punto**. Sin la referencia, es imposible poder trabajar con el objeto, como se ve a continuación.

```
> Punto p = new Punto();
> p.x = 5;
> p.y = 5;
> p.calculaDistanciaOrigen()
(double) 7.071
```

Se pueden crear todos los objetos que se deseen, y todos serán iguales entre ellos, a excepción de los valores de los atributos.

3 Visibilidad básica

En el ejemplo anterior del punto, es posible manejar desde fuera del objeto los valores de los atributos. Esto no es una buena idea, puesto que las clases deben ofrecer una interfaz (el conjunto de sus miembros públicos) tan pequeña al exterior como sea posible. Téngase en cuenta que el tamaño de la interfaz influye directamente con las posibles relaciones de una clase con otras. Cuantas más relaciones, más posibilidades existen de que futuras necesidades hagan difícil acometer cambios en la misma.

Así, la norma básica es mantener siempre los atributos como no visibles o privados, y que estos sean manejados a través de los métodos de la clase. Los métodos, a su vez suelen ser públicos, si bien es posible tener métodos no visibles que se utilicen como ayuda para el resto de métodos. De hecho, en el diagrama que se mostraba más arriba para las clases, ya se asume que los atributos son privados y los métodos públicos, por defecto.

Para indicar que un miembro de la clase es privado, se le añade el modificador *private* delante, mientras que para indicar que es público, se incluye el modificador *public*.

1 Se asume que se usa **BlueJ**, y que el código a continuación se escribe en el *CodePad*.

```
class Punto {
	private int x;
	private int y;

	public double calculaDistanciaOrigen()
	{
		return Math.sqrt( ( x * x ) + ( y * y ) );
	}
}
```

Si intentamos utilizar en este momento la clase **Punto**:

```
> Punto p = new Punto();
> p.x
ERROR: x has private access in Punto
> p.calculaDistanciaOrigen()
(double) 0.0
```

Ahora los atributos no son accesibles desde el exterior de la clase, por lo que todos los puntos que se creen tendrán exactamente los mismos valores: los que Java asigna por defecto a los valores enteros, 0. Para solucionar este problema, es necesario poder asignar valores a los atributos en el momento de la creación del objeto, como se verá a continuación.

4 Constructores

Los constructores son métodos especiales que se ejecutan justo después de crear el objeto. Típicamente se utilizan para inicializar los atributos con valores que se pasan en el momento de crear el objeto con **new**. Los constructores tienen el mismo nombre que la clase, y no devuelven nada, es decir, no se indica tipo de devolución como en el resto de métodos.

```
class Punto {
	private int x;
	private int y;

	public Punto(int a, int b)
	{
		x = a;
		y = b;
	}

	public double calculaDistanciaOrigen()
	{
		return Math.sqrt( ( x * x ) + ( y * y ) );
	}
}
```

En este momento, sí es posible crear un objeto **Punto** pasándole los valores que requiere el constructor, dándole valores a los atributos. Estos valores se pasan entre paréntesis, después del nombre de la clase.

```
> Punto p = new Punto( 5, 5 );
> p.calculaDistanciaOrigen();
(double) 7.071
```

Al no tener acceso a los atributos de la clase, no se puede conocer su valor, a no ser que lo recordemos del momento en el que el objeto fue creado. Para esto, es útil crear los llamados *getters*, que devuelven los valores de los atributos.

5 Getters

Los *getters* son métodos normales que se crean exclusivamente para devolver la información guardada por los atributos (siempre y cuando esa información pueda ser compartida, claro está). Su nombre siempre empieza por *get*, continúan con el nombre del atributo, y se caracterizan por devolver directamente el valor de ese atributo.

```
class Punto {
	private int x;
	private int y;

	public Punto(int a, int b)
	{
		x = a;
		y = b;
	}

	public int getX()
	{
		return x;
	}

	public int getY()
	{
		return y;
	}

	public double calculaDistanciaOrigen()
	{
		return Math.sqrt( ( x * x ) + ( y * y ) );
	}
}
```

Ahora ya es posible obtener la información guardada en los atributos, llamando a *getX()* o *getY()*.

```
> Punto p = new Punto( 5, 5 );
> p.calculaDistanciaOrigen();
(double) 7.071
> p.getX();
(int) 5
> p.getY();
(int) 5
```

6 El método *toString()*

Tiene mucha importancia un método llamado *toString()*. Este método es el que se utiliza siempre que se desea mostrar la información total del objeto por pantalla, por ejemplo. Así, es altamente recomendable aportar siempre un método *toString()* en las clases que creemos.

```
class Punto {
	private int x;
	private int y;

	public Punto(int a, int b)
	{
		x = a;
		y = b;
	}

	public int getX()
	{
		return x;
	}

	public int getY()
	{
		return y;
	}

	public double calculaDistanciaOrigen()
	{
		return Math.sqrt( ( x * x ) + ( y * y ) );
	}

	public String toString()
	{
		return "(" + getX() + ", " + getY() + ")";
	}
}
```

El objetivo es que se pueda obtener de manera textual la información que aloja el objeto, como se puede ver a continuación.

```
> Punto p = new Punto( 5, 5 );
> p.calculaDistanciaOrigen();
(double) 7.071
> p.getX();
(int) 5
> p.getY();
(int) 5
> p.toString();
“(5, 5)”
```

7 Constantes

Las constantes se indican en Java como cualquier otro atributo, pero anteponiendo el modificador *final*. Los nombres de las constantes se indican en mayúsculas, con cada palabra separada por guiones bajos.

```
class Punto {
	public final int ORIGEN_X = 0;
	public final int ORIGEN_Y = 0;

	private int x;
	private int y;

	public Punto(int a, int b)
	{
		x = a;
		y = b;
	}

	public int getX()
	{
		return x;
	}

	public int getY()
	{
		return y;
	}

	public double calculaDistanciaOrigen()
	{
		return Math.sqrt( ( x * x ) + ( y * y ) );
	}
```

```
      public String toString()
      {
            return "(" + getX() + ", " + getY() + ")";
      }
}
```

Así, estas constantes no pueden cambiar durante la vida del objeto. Esto tiene sentido, pues las coordenadas *x* e *y* del punto de origen siempre serán cero.

```
> Punto p = new Punto( 5, 5 );
> p.calculaDistanciaOrigen();
(double) 7.071
> p.getX();
(int) 5
> p.getY();
(int) 5
> p.toString();
"(5, 5)"
> p.ORIGEN_X
(int) 0
> p.ORIGEN_Y
(int) 0
> Punto org = new Punto( p.ORIGEN_X, p.ORIGEN_Y );
> org.toString();
"(0, 0)"
```

Esto es bastante interesante, aunque sin embargo presenta dos inconvenientes. El primero es que al formar parte de los atributos del objeto, las constantes ORIGEN_X y ORIGEN_Y estarán duplicadas en todos los objetos que se creen de la clase **Punto**. La segunda es que es necesario crear previamente un objeto de la clase **Punto** para poder acceder a esas constantes.

El efecto de la palabra clave *static* como modificador soluciona ambos problemas. Este modificador, cuando se aplica a un miembro de la clase, indica que ese miembro no pertenece a los objetos que se creen de esa clase, sino a la clase propiamente dicha. Y por tanto, solamente existirá una copia de ese miembro. Esto puede aplicarse a las constantes creadas más arriba.

```
class Punto {
	public static final int ORIGEN_X = 0;
	public static final int ORIGEN_Y = 0;

	private int x;
	private int y;

	public Punto(int a, int b)
	{
		x = a;
		y = b;
	}

	public int getX()
	{
		return x;
	}

	public int getY()
	{
		return y;
	}

	public double calculaDistanciaOrigen()
	{
		return Math.sqrt( ( x * x ) + ( y * y ) );
	}

	public String toString()
	{
		return "(" + getX() + ", " + getY() + ")";
	}
}
```

Con el modificador *static* aplicado a las constantes, estas se pueden acceder desde la propia clase.

```
> Punto.ORIGEN_X
(int) 0
> Punto.ORIGEN_Y
(int) 0
> Punto org = new Punto( Punto.ORIGEN_X, Punto.ORIGEN_Y );
> org.toString();
"(0, 0)"
```

7.1 Constantes complejas

Los atributos se crean cuando se crea el objeto. Los miembros estáticos ya existen cuando el primer objeto de la clase puede ser creado. Pero no tiene por qué tratarse de valores simples, como en el caso anterior, sino que pueden ser objetos completos. Así, el punto de origen debería ser un objeto **Punto**.

```
class Punto {
      public static final Punto ORIGEN = new Punto( 0, 0 );

      private int x;
      private int y;

      public Punto(int a, int b)
      {
            x = a;
            y = b;
      }

      public int getX()
      {
            return x;
      }

      public int getY()
      {
            return y;
      }

      public double calculaDistanciaOrigen()
      {
            return Math.sqrt( ( x * x ) + ( y * y ) );
      }

      public String toString()
      {
            return "(" + getX() + ", " + getY() + ")";
      }
}
```

Con la modificación anterior, ahora es mucho más natural referirse al origen de coordenadas.

```
> Punto.ORIGEN.toString();
"(0, 0)"
> Punto.ORIGEN.getX();
(int) 0
```

8 La referencia *this*

En todos los métodos de cualquier clase está disponible la referencia *this*, que apunta al objeto que está ejecutando el método. Por ejemplo, el constructor de la clase **Punto** podría reescribirse como sigue:

```
class Punto {
	//...
	public Punto(int x, int y)
	{
		this.x = x;
		this.y = y;
	}
	//...
}
```

El objetivo de la referencia *this* es la posibilidad de eliminar ambigüedades a la hora de manejar los propios atributos, los parámetros de un método, o incluso otro objeto de la misma clase pasado por parámetro. A continuación se muestra la clase **Punto** con el método *calculaDistancia()*, que acepta otro objeto **Punto** pasado por parámetro.

```
class Punto {
	public static final Punto ORIGEN = new Punto( 0, 0 );

	private int x;
	private int y;

	public Punto(int x, int y)
	{
		this.x = x;
		this.y = y;
	}

	public int getX()
	{
		return x;
	}

	public int getY()
	{
		return y;
	}

	public double calculaDistanciaOrigen()
	{
		return Math.sqrt( ( x * x ) + ( y * y ) );
	}
```

```
	public double calculaDistancia(Punto p2)
	{
		int difX = p2.getX() - this.getX();
		int difY = p2.getY() - this.getY();

		return Math.sqrt( ( difX * difX ) + ( difY * difY ) );
	}

	public String toString()
	{
		return "(" + getX() + ", " + getY() + ")";
	}
}
```

La distancia de un punto a otro se calcula de la misma forma que se calcula la distancia al origen. Es necesario restar las diferencias entre las componentes *x* y las componentes *y* para poder calcular la distancia utilizando la misma fórmula.

```
> Punto.ORIGEN.toString();
"(0, 0)"
> Punto.ORIGEN.getX();
(int) 0
> Punto p = new Punto( 5, 5 );
> p.calculaDistanciaOrigen();
(double) 7.071
> p.calculaDistancia( Punto.ORIGEN );
(double) 7.071
> Punto p2 = new Punto( 6, 6 );
> p.calculaDistancia( p2 );
(double) 1.4142
```

Nunca es obligatorio utilizar *this*, solamente es necesario cuando coinciden los nombres de los parámetros con los de los atributos (aunque siempre es posible renombrar dichos parámetros), o cuando se desee aclarar más allá de toda duda que el miembro es accedido por el mismo objeto que en ese momento ejecuta el método.

9 La clase Punto

La clase Punto ha crecido bastante desde el comienzo del texto, y su diagrama es, por tanto, actualizado como aparece más abajo.