Rev 1.5 20172601

Programación II

Tema 2. Encapsulación

Contenido

1 Introducción

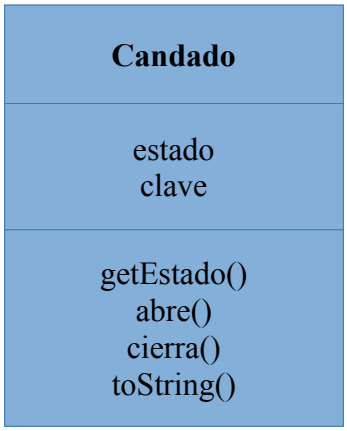
En este tema se abunda en las clases y objetos, la visibilidad de los miembros, y otros temas auxiliares que se presentan acompañando estos conceptos.

2 Identificación de clases

Cuando se plantea un enunciado, es sencillo identificar de manera básica las clases que participan en el problema, así como los atributos y métodos. La cuestión es asociar nombres con clases y atributos, mientras que los verbos indicarán métodos. En el siguiente enunciado, los nombres se identifican con un sombreado, mientras que los verbos se subrayan.

> Un candado electrónico se abre mediante una clave (una secuencia de números), mientras que se puede cerrar sin ella. Se visualiza como "*candado: abierto*" o "*candado: cerrado*". Es posible obtener el estado abierto o cerrado. La clave no puede ser cambiada, se asigna en el momento de crear el candado.

Está claro que tendremos que modelar objetos de la clase **Candado**, cuyos atributos son la *clave* y el *estado* (abierto, o cerrado). Las acciones que soporta el candado son *abre* y *cierra*, además de obtener el estado del candado. Por completitud, debemos aportar un método *toString()*, del cual el enunciado nos indica qué formato debe seguir su respuesta.



Lo siguiente a analizar son los tipos de los atributos. El estado solo puede tener dos valores: abierto o cerrado, que podemos asimilar con un booleano. La clave es una secuencia de números, así que un entero parece ser suficiente. Así, es posible completar la información del diagrama, si se desea:

Los métodos *abre()* y *cierra()* no devuelven ninguna información, ya que son acciones, por lo que su tipo de retorno es *void* (que se puede traducir directamente como: nada). Esta es la forma de indicar que un método no devuelve ningún valor.

```
class Candado {
	private int clave;
	private boolean estado;

	public Candado(int k)
	{
		clave = k;
		estado = false;
	}

	public boolean getEstado()
	{
		return estado;
	}

	public void abre(int k)
	{
		if ( clave == k ) {
			estado = true;
		}
	}

	public void cierra()
	{
		estado = false;
	}
```

```
      public String toString()
      {
            String toret = "abierto";

            if ( !getEstado() ) {
                  toret = "cerrado";
            }

            return "Candado: " + toret;
      }
}
```

Una vez creada la clase, es posible utilizarla para crear objetos **Candado**, como se puede ver a continuación[1].

```
      > Candado c1= new Candado( 123 );
      > c1.toString();
      "Candado: cerrado"
      > c1.abre( 456 );
      > c1.getEstado();
      false
      > c1.abre( 123 );
      > c1.toString();
      "Candado: abierto"
      > c1.cierra();
      > c1.toString();
      "Candado: cerrado"
```

2.1 Discusión sobre visibilidad

Los atributos *estado* y *clave* tienen que ser privados, pues si pudiesen ser manipulados desde el exterior, entonces toda la infraestructura del candado no tendría sentido. Por ejemplo, si la clave fuese pública, se podría hacer algo como lo que sigue:

```
> Candado c1= new Candado( 123 );
> c1.toString();
"Candado: cerrado"
> c1.clave
123
> c1.abre( 123 );
> c1.getEstado()
true
```

Es decir, no sería necesario conocer la clave para abrir el candado, ya que simplemente accediendo a ella podríamos obtener dicha información.

1 Se asume que el lector está trabajando con *BueJ*, y una vez creada la clase, introduce las líneas marcadas con '>' en el *CodePad*.

Así, que *estado* fuese público sería incluso peor. Ni siquiera sería necesario preocuparse por la clave, puesto que no sería necesaria en absoluto.

```
> Candado c1= new Candado( 123 );
> c1.toString();
"Candado: cerrado"
> c1.estado = true;
> c1.toString();
"Candado: abierto"
```

Finalmente, aún manteniendo los atributos *clave* y *estado* privados, no es posible ofrecer un método público *getClave()*, ya que nos encontraríamos en el primer caso enumerado más arriba.

```
> Candado c1= new Candado( 123 );
> c1.toString();
"Candado: cerrado"
> c1.getClave()
123
> c1.abre( 123 );
> c1.getEstado()
true
```

Cuando se diseña una clase puede seguirse ciegamente la regla de que los atributos deben ser privados, lo cual soluciona de por si muchos potenciales errores. Pero además, debe escogerse cuidadosamente a qué información se proporciona acceso mediante los métodos. De nuevo, cuanto más pequeña sea la interfaz de una clase (es decir, cuantos menos métodos públicos tenga), mejor.

2.2 Qué pasa cuando no se indica visibilidad

Por despiste, o por comodidad, es posible escribir código como el que aparece más abajo.

```
class Candado {
      int clave;
      boolean estado;

      public Candado(int k)
      {
            clave = k;
            estado = false;
      }

      // más cosas…
}
```

Los atributos *clave* y *estado* no son privados en la clase. En realidad, son accesibles desde el

exterior. Java tiene el concepto de *package* (paquete). Los paquetes son colecciones de clases. Así, ambos atributos son accesibles desde el mismo paquete, pero inaccesibles desde otros paquetes. Esto, desde luego, no es el comportamiento deseado.

```
> Candado c1= new Candado( 123 );
> c1.toString()
"Candado: cerrado"
> c1.estado = true;
> c1.getEstado()
true
> c1.toString()
"Candado: abierto"
```

Así, es posible cambiar el *estado* del candado, de tal manera que, sin saber la clave, es posible abrirlo.

3 Documentación

Es muy recomendable escribir comentarios que reflejen la funcionalidad del código, especialmente cuando se trata de indicar cuál es la funcionalidad de un método. Java ofrece la posibilidad de, escribiendo los comentarios en un formato especial, poder obtener automáticamente documentación con referencias cruzadas, mediante una herramienta llamada *JavaDoc*.

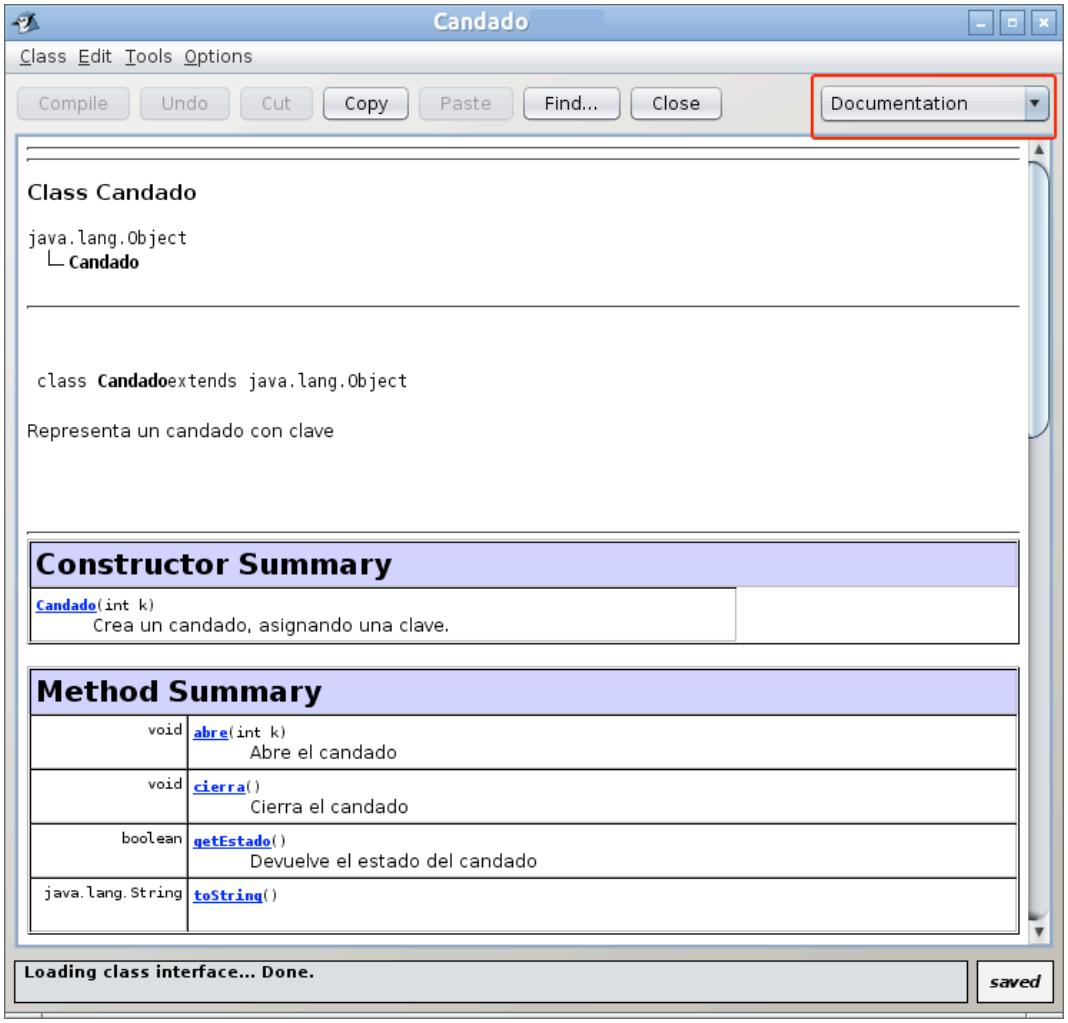
Etiqueta	Explicación
`@param <parámetro> <explicación>`	Proporciona documentación sobre un parámetro en concreto.
`@return <explicación>`	Proporciona documentación sobre el retorno de un método.
`@see <nombre>`	Permite relacionar la documentación de este miembro con otro.

Los comentarios para JavaDoc comienzan con /** en lugar de con /*. Una vez dentro de este comentario, la primera línea consiste en un resumen de lo que hace el método. A continuación, se puede escribir información más específica, mediante unas etiquetas que comienzan siempre con el símbolo @.

A continuación, se muestra la clase **Candado** con los comentarios introducidos de la forma explicada.

```
/** Representa un candado con clave */
class Candado {
	private int clave;
	private boolean estado;

	/** Crea un candado, asignando una clave.
	  * @param k La clave, como entero */
	public Candado(int k)
	{
		clave = k;
		estado = false;
	}

	/** Devuelve el estado del candado
	  * @return true si abierto, false cerrado. */
	public boolean getEstado() {
		return estado;
	}

	/** Abre el candado
	  * @param k La clave a probar, como entero */
	public void abre(int k)
	{
		if ( clave == k ) {
			estado = true;
		}
	}

	/** Cierra el candado */
	public void cierra() {
		estado = false;
	}


	public String toString()
	{
		String toret = "abierto";

		if ( !getEstado() ) {
			toret = "cerrado";
		}

		return "Candado: " + toret;
	}
}
```

Se muestra también una captura sobre cómo el editor de BlueJ permite cambiar entre documentación y código fuente, en el desplegable remarcado en rojo.



Es una ventaja poder generar este tipo de documentación sin apenas esfuerzo. Además, esta información en otros entornos de programación como *Eclipse*, *NetBeans* o *IntelliJ IDEA*, se muestra en el momento de utilizar el constructor de la clase o el método en cuestión.

4 Setters

Al igual que los *getters*, los *setters* son métodos normales, que permiten modificar los atributos de un objeto. Supóngase el siguiente ejemplo a continuación.

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

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

	public int getX()
	{
		return x;
	}

	public int getY()
	{
		return y;
	}

	public void setX(int x)
	{
		this.x = x;
	}

	public void setY(int y)
	{
		this.y = y;
	}
}
```

En el ejemplo anterior, los objetos **Punto** permiten la modificación de las coordenadas que albergan.

```
> Punto p1 = new Punto( 5, 6 );
> String.format( "%d, %d", p1.getX, p1.getY() )
"5, 6"
> p1.setX( 10 );
> p1.setY( 12 );
> String.format( "%d, %d", p1.getX, p1.getY() )
"10, 12"
```

Es importante crear *setters* con moderación. La alternativa a usar setters siempre es la misma: crear un nuevo objeto. En muchas ocasiones, esto es preferible a usar *setters*. La alternativa a una clase **Punto** sin *setters* supondría el comportamiento que aparece a continuación.

```
class Punto {
	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;
	}
}
```

En el ejemplo anterior, los objetos **Punto** no permiten la modificación de las coordenadas que albergan. Por tanto, es necesario crear un nuevo objeto en su lugar.

```
> Punto p1 = new Punto( 5, 6 );
> p1.getX
5
> p1.getY()
6
> p1 = new Punto( 10, 12 );
> p1.getX
10
> p1.getY()
12
```

En el caso concreto de un objeto **Punto**, no existe una clara diferencia conceptual entre tener o no tener métodos *set*. Sí que es necesario tener en cuenta que la creación de nuevos objetos supone siempre una merma en el rendimiento. En un caso más específico, podría ponerse a la clase **Persona**, que aparece a continuación.

```
/** Representa a personas */
class Persona {
	private String nombre;
	private int dni;
	private int anhoNacimiento;

	/** Crea un nuevo objeto Persona
	    @param dni El dni, como int
	    @param nombre El nombre, como String
	    @param anhoNacimiento El año de nacimiento, como int
	*/
	public Persona(int dni, String nombre, int anhoNacimiento)
	{
		this.dni = dni;
		this.nombre = nombre;
		this.anhoNacimiento = anhoNacimiento;
	}

	/** @return El dni, como int */
	public int getDni()
	{
		return this.dni;
	}

	/** @return El año de nacimiento, como int */
	public int getAnoNacimiento()
	{
		return this.anhoNacimiento;
	}

	/** @return La edad, como int */
	public int getEdad()
	{
		return Calendar.getInstance().get( Calendar.YEAR )
				- this.getAnoNacimiento();
	}

	/** @return El nombre, como String */
	public String getNombre()
	{
		return this.nombre;
	}

	/** Cambia el nombre
	    @param n El nuevo nombre, como String
	*/
	public void setNombre(String n)
	{
		this.nombre = n;
	}
}
```

Este es un uso mucho más realista de getters y setters. Si bien el *nombre* de la persona puede ser tanto accedido como modificado, el *dni* solo puede ser accedido. La razón es que una persona puede cambiar de nombre a lo largo de su vida, pero no puede cambiar de *dni*.

Nótese la presencia del método *getEdad()*, que devuelve la edad de la persona calculándola a partir del año de nacimiento. No siempre es necesario que exista un atributo correspondiente al método (*edad*, en este caso), sino que el valor puede calcularse a partir de otro.

4.1 Getter/setter en lugar de atributos públicos

Más arriba se definía el objeto de la clase **Punto** con *getters* y *setters*. De ser así, es razonable plantearse si se puede prescindir de ambos, declarando los atributos como públicos.

```
class Punto {
      public int x;           // ERROR: implementación de la clase expuesta
      public int y;           // ERROR: implementación de la clase expuesta
}
```

Los atributos deben ser siempre privados, el código más arriba está ampliamente desaconsejado. La razón es que, si bien ese código proporciona una funcionalidad inmediata, no escala bien. Cualquier código va a estar sujeto a cambios, debido a ampliación de funcionalidad o corrección de errores. Y el código que use la clase **Punto** inmediatamente superior va a estar fuertemente acoplado a la implementación (es decir, al interior) de la clase.

Como un supuesto específico, supongamos que es necesario cambiar el tipo de las coordenadas de *int* a *double*, debido a que es necesario tratar con coordenadas con decimales. Una clase Punto corregida podría ser la siguiente.

Nótese que sigue siendo posible crear objetos de la clase Punto con valores estrictamente enteros, pero sin pérdida de precisión, puesto que los atributos han cambiado ahora para ser *double*'s. En el caso en el que los atributos son públicos, no queda otra posibilidad que realizar los cambios necesarios no solo en la propia clase, sino también el el código que la usa.

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

	public Punto(double x, double y)
	{
		this.setX( x );
		this.setY( y );
	}

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

	public double getX()
	{
		return x;
	}

	public double getY()
	{
		return y;
	}

	public void setX(double x)
	{
		this.x = x;
	}

	public void setY(double y)
	{
		this.y = y;
	}
}
```

5 Sobrecarga de métodos

En muchos casos, sería interesante tener varias versiones de un método, que según diferentes parámetros que se le pasaran, actuara de una forma u otra. En Java, es de hecho posible tener varios métodos con el mismo nombre, siempre y cuando varíe el número de parámetros, su tipo, o el número y el tipo a la vez. Nótese que los constructores no son sino un tipo de métodos.

En el siguiente ejemplo, se utiliza un segundo constructor para aceptar una secuencia de tres números como clave para el candado.

```
/** Representa un candado con clave */
class Candado {
      private int clave;
      private boolean estado;

      /** Crea un candado, asignando una clave.
        * @param k la clave, como entero */
      public Candado(int k)
      {
            clave = k;
            estado = false;
      }

      /** Crea un candado, asignando una clave.
        * @param a Primer dígito de la clave, como entero.
        * @param b Segundo dígito de la clave, como entero.
        * @param c Tercer dígito de la clave, como entero. */
      public Candado(int a, int b, int c)
      {
            clave = ( ( a % 10 ) * 100 ) + ( ( b % 10 ) * 10 ) + ( c % 10 );
            estado = false;
      }
      //...
}
```

Ahora es posible crear el candado de dos formas: mediante tres dígitos, o bien mediante un solo número (en este último caso puede ser de varios dígitos). En el caso de aportar tres dígitos, se hace el módulo con 10 a cada uno de ellos para asegurar que en cualquier caso el resultado estará entre 0 y 9.

```
> Candado c1 = new Candado( 1, 2, 3 );
> c1.toString();
"Candado: cerrado"
> c1.abre( 123 );
> c1.getEstado()
true
```

Además de sobrecargar el constructor, es posible sobrecargar cualquier otro método. Por ejemplo, se puede sobrecargar el método *abre()* para que también acepte tres dígitos en lugar de un solo número entero.

```
/** Representa un candado con clave */
class Candado {
    private int clave;
    private boolean estado;

    /** Crea un candado, asignando una clave.
      * @param k La clave, como entero */
    public Candado(int k)
    {
        clave = k;
        estado = false;
    }

    /** Crea un candado, asignando una clave.
      * @param a Primer dígito de la clave, como entero.
      * @param b Segundo dígito de la clave, como entero.
      * @param c Tercer dígito de la clave, como entero. */
    public Candado(int a, int b, int c)
    {
        clave = convierteClave( a, b ,c );
        estado = false;
    }

    /** Devuelve el estado del candado
      * @return true si abierto, false cerrado. */
    public boolean getEstado()
    {
        return estado;
    }

    /** Abre el candado
      * @param k La clave a probar, como entero */
    public void abre(int k)
    {
        if ( clave == k ) {
            estado = true;
        }
    }

    /** Abre el candado
      * @param a Primer dígito de la clave, como entero.
      * @param b Segundo dígito de la clave, como entero.
      * @param c Tercer dígito de la clave, como entero. */
    public void abre(int a, int b, int c) {
        if ( clave == convierteClave( a, b, c ) ) {
            estado = true;
        }
    }
```

```
    /** Cierra el candado */
    public void cierra()
    {
        estado = false;
    }


    public String toString()
    {
        String toret = "abierto";

        if ( !getEstado() ) {
            toret = "cerrado";
        }

        return "Candado: " + toret;
    }

   /**
     * Convierte una clave proporcionada en tres dígitos a un solo entero.
     * @param a Primer dígito de la clave, como entero.
     * @param b Segundo dígito de la clave, como entero.
     * @param c Tercer dígito de la clave, como entero. */
    private int convierteClave(int a, int b, int c)
    {
        return ( ( a % 10 ) * 100 ) + ( ( b % 10 ) * 10 ) + ( c % 10 );
    }
}
```

Dado que en este caso, tanto el constructor como el método *abre()* necesitan convertir los dígitos en un solo entero, la lógica se ha movido al método *convierteClave()*. Este es el típico caso de un método auxiliar que no es necesario que sea visible desde el exterior.

```
> Candado c1 = new Candado( 1, 2, 3 );
> c1.toString()
"Candado: cerrado"
> c1.abre( 1, 2, 3 );
> c1.toString()
"Candado: abierto"
```

6 Vectores y matrices primitivos

Java, al igual que muchos otros lenguajes de programación, permite el manejo de vectores y matrices primitivas. Como funcionalidad extra, no es necesario guardar la longitud de un vector, pues siempre está disponible en una constante pública llamada *length*. Las posiciones del vector se indexan desde cero, y por tanto la última siempre es la posición *length – 1*.

El siguiente código crea un vector de diez posiciones, y mediante un bucle *for()* calcula el doble de cada posición, modificándola con el nuevo valor.

Ejemplo

```
int[] v = new int[]{ 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
for(int i = 0; i < v.length; ++i) {
      v[ i ] *= 2;
}
```

Para crear un vector es necesario utilizar el operador **new**, y por lo tanto el uso de una referencia para apuntarlo.

```
int suma = 0;
int[] v = new int[]{ 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };

for(int i = 0; i < v.length; ++i) {
      int x = v[ i ];
      suma += x;
}
```

En el ejemplo anterior, se daba la posibilidad al usuario de la clase de pasar la clave como tres dígitos. La siguiente modificación cambia el método *convierteClave()* para que acepte un vector de dígitos, en lugar de exactamente tres. Una vez cambiado este método, es posible cambiar también el método *abre()* y el constructor.

```
class Candado {
      //...

      /** Crea un candado, asignando una clave.
        * @param digitos Los dígitos de la clave,
        * como un vector de enteros primitivo. */
       public Candado(int[] digitos)
       {
                  clave = convierteClave( digitos );
                  estado = false;
       }

      /** Abre el candado
        * @param digitos Los dígitos de la clave,
        * como un vector de enteros primitivo. */
       public void abre(int[] digitos)
       {
                  abre( convierteClave( digitos ) );
       }

       //...
```

```
      /**
        * Convierte una clave proporcionada en tres dígitos a un solo entero.
        * @param digitos Los dígitos de la clave,
        * como un vector de enteros primitivo. */
       private int convierteClave(int[] digitos)
       {
            int toret = 0;
            final int length = digitos.length;

            for(int i = 0; i < length; ++i) {
                  toret += digitos[ i ] * Math.pow( 10, length - i - 1);
            }

            return toret;
       }
}
```

Ahora es posible llamar tanto al constructor como a *abre()* con un número arbitrario de dígitos.

```
> Candado c1 = new Candado(new int[]{ 1, 2, 3 } );
> c1.toString();
"Candado: cerrado"
> c1.abre( new int[]{ 1, 2, 3 } );
> c1.toString();
"Candado: abierto"
```

7 Miembros *static*

Anteriormente se ha visto el uso de *static* como una forma de crear constantes de manera más eficiente. Este no es, evidentemente, el único uso posible del modificador *static*. Existen ventajas en crear tanto atributos como métodos pertenecientes a la clase. En el caso de la clase **Candado**, se puede llevar la cuenta de los candados creados mediante un atributo y un método *static*, como se ve a continuación.

```
/** Representa un candado con clave */
class Candado {
    private static int numCandados = 0;

    private int clave;
    private boolean estado;

    /** Crea un candado, asignando una clave.
      * @param k La clave, como entero */
    public Candado(int k) {
        clave = k;
        estado = false;
        ++numCandados;
    }

    /** Crea un candado, asignando una clave.
      * @param digitos Los dígitos de la clave,
      * como un vector de enteros primitivo. */
    public Candado(int[] digitos) {
        clave = convierteClave( digitos );
        estado = false;
        ++numCandados;
    }

    /** Devuelve el num. de candados creados hasta el momento.
     *  @return El num. de candados, como entero.
     */
    public static int getNumCandados() {
        return numCandados;
    }

    //...
}
```

Así, en este momento se pueden crear varios candados y obtener su conteo:

```
> Candado c1 = new Candado( 123 );
> Candado c2 = new Candado( 456 );
> Candado.getNumCandados();
(int) 2
```

8 Enumerados

Los tipos enumerados permiten crear un grupo de constantes que toman valores consecutivos. Por ejemplo, en el caso de la clase **Candado** más arriba, se podría utilizar un enumerado que contemplara la posibilidad de ambos estados: abierto o cerrado.

Ejemplo

```
public enum Estado { CERRADO, ABIERTO };
```

Las constantes de un enumerado se indican en mayúsculas, como cualquier otra constante. Además, se les asigna un valor que comienza en cero y se va incrementando de izquierda a derecha. En este caso, la constante **CERRADO** tiene valor 0, y **ABIERTO** tiene valor 1.

A continuación, se muestra el cambio del atributo *estado*, de tipo **boolean** al tipo enumerado **Estado**. Nótese que para hacer referencia a las constantes del tipo enumerado, es necesario cualificarlas con el nombre del tipo.

```
/** Representa un candado con clave */
class Candado {
    public enum Estado { CERRADO, ABIERTO };

    private static int numCandados = 0;

    private int clave;
    private Estado estado;

    /** Crea un candado, asignando una clave.
      * @param k La clave, como entero */
    public Candado(int k)
    {
        clave = k;
        estado = Estado.CERRADO;
        ++numCandados;
    }

    /** Crea un candado, asignando una clave.
      * @param digitos Los dígitos de la clave,
      *        como un vector de enteros primitivo. */
    public Candado(int[] digitos)
    {
        clave = convierteClave( digitos );
        estado = Estado.CERRADO;
        ++numCandados;
    }
```

```
/** Devuelve el num. de candados creados hasta el momento.
  *  @return El num. de candados, como entero.
  */
 public static int getNumCandados()
 {
     return numCandados;
 }

 /** Devuelve el estado del candado
   * @return true si abierto, false cerrado. */
 public Estado getEstado()
 {
     return estado;
 }

 /** Abre el candado
   * @param k La clave a probar, como entero */
 public void abre(int k)
 {
     if ( clave == k ) {
         estado = Estado.ABIERTO;
     }
 }

 /** Abre el candado
   * @param digitos Los dígitos de la clave,
   *        como un vector de enteros primitivo. */
 public void abre(int[] digitos)
 {
     abre( convierteClave( digitos ) );
 }

 /** Cierra el candado */
 public void cierra()
 {
     estado = Estado.CERRADO;
 }


 public String toString()
 {
     String toret = "abierto";

     if ( getEstado() == Estado.CERRADO ) {
         toret = "cerrado";
     }

     return "Candado: " + toret;
 }
```

```
   /**
     * Convierte una clave proporcionada en tres dígitos a un solo entero.
     * @param digitos Los dígitos de la clave, como un vector de enteros primitivo.
     */
    private int convierteClave(int[] digitos)
    {
        int toret = 0;
        final int length = digitos.length;

        for(int i = 0; i < length; ++i) {
            toret += digitos[ i ] * Math.pow( 10, length - i - 1);
        }

        return toret;
    }

}
```

La ventaja de utilizar tipos enumerados frente a cualquier otro tipo, como **boolean**, es que los tipos enumerados son mucho más informativos, como se puede ver a continuación.

```
> Candado c2 = new Candado( 456 );
> c2.getEstado()
(Candado.Estado) CERRADO
> c2.toString()
"Candado: cerrado"
> c2.abre( new int[]{ 4, 5, 6 } );
> c2.getEstado()
(Candado.Estado) ABIERTO
> c2.toString()
"Candado: abierto"
```

Para cualquier tipo enumerado está disponible el método *values()*, que devuelve un vector primitivo con cada una de las constantes que lo componen.

```
> Candado.Estado.ABIERTO
(Candado.Estado) ABIERTO
> Candado.Estado.CERRADO
(Candado.Estado) CERRADO
> Candado.Estado.values().length
(int) 2
> Candado.Estado.values()[ 0 ]
(Candado.Estado) CERRADO
> Candado.Estado.values()[ 1 ]
(Candado.Estado) ABIERTO
```

9 String

Las cadenas de caracteres en Java no son tipos primitivos, sino objetos de la clase **String**, si bien Java se guarda mucho de que nos demos cuenta, al menos al principio.

Ejemplo

```
String s = new String( “hola” );    // es lo mismo que String s = “hola”
```

Así, cada cadena es un objeto en Java, y además, es inmutable. Esto quiere decir que cualquier operación que se realice sobre una cadena devuelve una nueva cadena, no modifica la cadena original.

Por ejemplo, el método *trim()* recorta los espacios al comienzo y al final de la cadena. Sin embargo, al llamar a *trim()* se crea una cadena nueva, no se modifica la cadena original, como se puede ver a continuación.

Ejemplo

```
> String s = "    ESEI    ";
> s.trim();
"ESEI"
> s
"    ESEI    "
```

Si se desea que el cambio sea permanente, no habrá otra opción que asignar la referencia al nuevo objeto, como se puede ver a continuación. El antiguo objeto queda inaccesible, pendiente de recolección.

Ejemplo

```
> String s = "    ESEI    ";
> s = s.trim();
> s
"ESEI"
```

Supóngase para entender la siguiente tabla, que contiene una muestra de los métodos disponibles en **String**, que existe un objeto **String** *s*, con el contenido “ ESEI “.

Método	Explicación	Ejemplo
trim()	Devuelve una nueva cadena eliminando los espacios al comienzo y al final.	`s = s.trim();  // “ESEI”`
charAt(i)	Devuelve el carácter en la posición dada.	`char c = s.charAt( 0 ); // c == 'E'`
toUpperCase()	Devuelve una nueva cadena conviertiendo los caracteres a mayúsculas.	`String s2 = s.toUpperCase(); // “ESEI”`
toLowerCase()	Devuelve una nueva cadena conviertiendo los caracteres a minúsculas.	`String s3 = s.toLowerCase(); // “esei”`
equals()	Devuelve **true** si dos cadenas son iguales.	`s2.equals( s3 ); // false`
equalsIgnoreCase()	Devuelve **true** si dos cadenas son iguales, ignorando mayúsculas y minúsculas.	`s2.equalsIgnoreCase( s3 ); // true`
length()	Devuelve la longitud de la cadena.	`s2.length(); // 4`
indexOf(c)	Devuelve la posición de *c* en la cadena.	`s2.indexOf( 'E' ); // 0` `s2.indexOf( 'j' ); // -1`
concat(s)	Concatena dos cadenas.	`s = s2.concat( s3 ); // s es “ESEIesei”`

El hecho de que las cadenas sean inmutables en Java supone tener en cuenta ciertas cuestiones de eficiencia. En el siguiente ejemplo, el método estático *lista()* crea un texto con el estado de varios candados, pasados como un vector.

```
public class Ppal {
   public static String lista(Candado[] candados)
   {
       String toret = "";

       for(int i = 0; i < candados.length; ++i) {
           toret += candados[ i ].toString() + '\n';
       }

       return toret;
   }
}
```

El problema consiste es que, para cada candado, se crea a) una nueva cadena que es el resultado de concatenar el resultado del *toString()* del candado en cuestión con el salto de línea, y b) una nueva cadena, apuntada por *toret*, que tendrá el antiguo contenido de toret más el resultado de la concatenación anterior, en a). Es decir, dos cadenas se generarán por cada vuelta de bucle. Si el vector pasado como argumento tiene diez objetos, entonces hasta llegar a la cadena final se habrán generado veinte objetos intermedios que ocuparán memoria pero que estarán abandonados, disponibles para la recolección.

Para solucionar este problema, existe la clase **StringBuilder**. Esta clase es parecida a una cadena de texto, pero sí que es modificada cada vez que se realiza una operación sobre ella, por lo que no se crean objetos intermedios. Así, el método anterior quedaría como sigue:

```
public class Ppal {
   public static String lista(Candado[] candados)
   {
       StringBuilder toret = new StringBuilder();

       for(int i = 0; i < candados.length; ++i) {
            toret.append( candados[ i ].toString() );
            toret.append( '\n' );
       }

       return toret.toString();
   }
}
```

Los objetos de la clase **StringBuilder** admiten que se llame al método *append()* todas las veces que sea necesario, haciendo crecer la cadena de texto que guardan en su interior, pero sin crear cadenas auxiliares, resultados intermedios que ocuparían memoria. Una vez que se ha terminado de crear la cadena, llamando al método *toString()* se obtiene ese texto resultante.