

## Clase adicional 3

Temas

- Métodos
  - Definir un método
  - Llamar a un método
  - Llamada por valor
  - Constructores
  - Método *static*
- *Arrays*
- Vectores
- Problemas de la clase adicional
- Problemas de diseño

## Métodos

En una clase, cada método realiza una tarea relativamente autónoma. Este permite que los programadores puedan dividir un problema complejo en varias piezas más pequeñas y manejables. Una vez que el método se ha escrito y se ha probado correctamente, el programador ya no necesita preocuparse por su funcionamiento interno. Los usuarios simplemente utilizan el método y los argumentos adecuados para invocarlo siempre que se necesite su funcionalidad.

Cada método cuenta con:

- Identificador de acceso. Al igual que las variables, los métodos pueden ser *public* o *private* (existen otros identificadores de acceso que trataremos más adelante en el curso). A los métodos *private* solamente se puede acceder desde dentro de la clase. Por su parte, a los métodos *public* puede acceder cualquiera, tanto desde dentro como desde fuera de la clase.
- Tipo devuelto. Un método puede devolver un valor al usuario. Puede tratarse de un tipo de datos simple, como *int* o de otra clase. Un tipo devuelto *void* indica que no se devolverá ningún valor.
- Nombre. El nombre de un método debe comenzar con una letra, distingue entre mayúsculas y minúsculas y no puede ser una palabra reservada de Java. Una buena práctica es utilizar nombres descriptivos, por ejemplo, *setHeight*.
- Lista de argumentos. Los argumentos son campos de entrada para un método. Un parámetro puede ser un tipo de datos simple y otra clase.

En Java, los métodos DISTINTOS de la MISMA clase pueden tener el mismo nombre y devolver el mismo tipo de datos, pero tener un conjunto de parámetros DISTINTO (este conjunto de parámetros recibe el nombre de firma del método). Este hecho se conoce como la "sobrecarga de métodos". Se llamará al método correcto dependiendo del tipo y del número de argumento(s).

Aunque este párrafo es bastante corto, el concepto de sobrecarga de métodos es muy importante y realmente útil en muchos programas de Java. Verá distintos ejemplos de sobrecarga de métodos en las siguientes clases y en los boletines de problemas.

## Definir un método

Para poder invocar un método, primero es necesario definirlo. La definición del método proporciona los detalles sobre el modo de realizar las tareas. Dado que cada método es un fragmento de código autónomo, puede definir sus propias variables y métodos.

Utilizando nuestra clase *Box*, vamos a definir 7 métodos:

```
public class Box
{
    private double width;
    private double height;
    private double length;

    // Método para calcular el volumen de una caja
    public double volume()
    {
        return width*height*length;
    }

    // Método para establecer el valor del ancho de la caja
    public void setWidth(double w) {
        width = w;
    }

    // Método para establecer el valor del alto de altura de la caja
    public void setHeight(double h){
        height = h;
    }

    // Método para establecer el valor de la longitud de la caja
    public void setLength(double l){
        length = l;
    }

    // Método para obtener el valor del ancho de la caja
    public double getWidth(){
        return width;
    }

    // Método para obtener el valor de la altura de la caja
    public double getHeight(){
        return height;
    }

    // Método para obtener el valor de la longitud de la caja
    public double getLength(){
        return length;
    }
}
```

```
}
```

Utilizando los métodos de la clase *Box*:

Para llamar a los métodos que hemos definido en la clase *Box*, utilizamos un `.` (operador de punto). Por ejemplo:

```
public class Example{
    public static void main (String args[]) {
        Box myFirstBox = new Box();    // creación de la caja
        myFirstBox.setWidth(7.5);
        myFirstBox.setHeight(6.97);
        myFirstBox.setLength(2);
        System.out.println("El volumen es " + myFirstBox.volume());
    }
}
```

Datos de salida: El volumen es 104.55

### Llamada por valor

Cuando se invoca un método, Java crea una variable temporal para cada uno de sus parámetros y copia el valor de dichos parámetros en la variable temporal. Dicho de otro modo, el método obtiene solamente el valor de dicho parámetro, no el parámetro en sí. Cualquier cambio realizado dentro del método únicamente afectará a la variable temporal, no al parámetro pasado al método. En informática, esto se conoce como "llamada\_por\_valor".

Tenga en cuenta que Java siempre utiliza la llamada\_por\_valor para pasar parámetros. Este proceso resulta mucho más sutil de lo que puede parecer a primera vista, sobre todo cuando los objetos se están pasando, tal como se explica a continuación.

En el siguiente ejemplo, hemos definido un método llamado *incremento* que toma dos argumentos: un entero y un *Box*. Dentro del método, hemos cambiado ambos argumentos:

```
void doubleWidth (double w, Box b) {
    w = w * 2;
    b.setWidth (w);
    System.out.println("Dentro del método: w = " + w + " width = " + b.getWidth());
}
```

Aquí está el programa *main*:

```
public static void main (String args[]) {
    Box myFirstBox = new Box();
    myFirstBox.setWidth(7.5);
    double w = myFirstBox.getWidth();
}
```

```
System.out.println("Antes de llamar al método: w = " + w + " ancho = " +  
myFirstBox.getWidth());  
Example ex = new Example();  
ex.doubleWidth(w, myFirstBox);  
System.out.println("Después de llamar al método: w = " + w + " ancho = " +  
myFirstBox.getWidth());  
}
```

Aquí están los datos de salida:

Antes de llamar al método: w = 7.5 ancho = 7.5  
Dentro del método: w = 15.0 ancho = 15.0

Después de llamar al método: w = 7.5 ancho = 15.0

Como era de esperar, el valor de *w* no ha cambiado antes y después de llamar al método. Pero, espere: ¿por qué el ancho de *myFirstBox* cambia a 15 tras llamar al método? ¿No habíamos dicho que el método no cambiaría el contenido de los parámetros?

Bueno, examinemos la forma en que se llama al método.

```
ex.doubleWidth(w, myFirstBox);
```

Entonces, ¿qué es *myFirstBox*? ¿Es un objeto o una referencia al objeto? (si no sabe la respuesta, consulte la clase adicional anterior). Tiene razón: es una referencia al objeto creado por el nuevo operador.

Dentro del método, Java ha creado una variable temporal *b* (que también es una referencia) y ha copiado el contenido de *myFirstBox* en *b*. Ahora, tanto *b* como *myFirstBox* son referencias al mismo objeto. Por lo tanto, si cambiamos el ancho del objeto al que apunta *b*, también estaremos cambiando el ancho del objeto al que apunta *myFirstBox*, ya que son el mismo objeto.

Para los que no terminen de verlo claro, incluimos un breve resumen final.

- Si el parámetro es un tipo de datos primitivo (int, double, etc.), los cambios realizados dentro del método NO afectarán al llamador
- Si el parámetro es una referencia al objeto, los cambios realizados dentro del método SÍ afectarán al llamador

Consulte el libro de texto (págs. 148-151) para obtener más información y un ejemplo detallado.

## Un método especial: constructor

Los constructores son métodos que se utilizan para CREAR un objeto de la clase. Se llaman automáticamente cuando se utiliza el nuevo operador. Un constructor no tiene un tipo devuelto; tiene el mismo nombre que su clase y puede tener cualquier número de parámetros.

Una clase puede contener varios constructores. Se utilizan, sobre todo, para inicializar variables de miembros de un objeto. Por ejemplo, a continuación se muestra un ejemplo de los constructores de la clase *Box*:

```
//Constructor predeterminado
```

```
public Box() {  
    width = 0;  
    height = 0;  
    length = 0;  
}
```

```
//Constructor con argumentos
```

```
public Box (double w, double h, double l) {  
    width = w;  
    height = h;  
    length = l;  
}
```

Los constructores se llaman del siguiente modo:

```
Box b1 = new Box();  
Box b2 = new Box(2.0,2.5,3.0);
```

Observe que éste es un caso típico de sobrecarga de métodos. Ambos constructores tienen el mismo nombre (*Box*) pero distintos conjuntos de argumentos. Como resultado, son métodos distintos: el objeto *b1* tiene un ancho, altura y longitud de cero, mientras que *b2* tiene un ancho de 2.0, una altura de 2.5 y una longitud de 3.0

Si no hay ningún constructor definido para una clase, el compilador crea uno predeterminado que no tome ningún argumento. Este constructor predeterminado inicializará de forma implícita las variables de las instancias a sus valores predeterminados (0 para tipos numéricos primitivos, *false* para booleanos y *null* para referencias).

## Método *static*

Cuando la definición de un método viene precedida por la palabra clave *static*, recibe el nombre de método *static*. La diferencia fundamental entre los métodos de clases normales y los métodos *static* es que éstos operan en la clase, mientras que los normales lo hacen en las instancias de la clase.

Por tanto, los métodos *static* se llaman utilizando el nombre de la clase, no el de la instancia.

Por ejemplo:

```
public static void printBox (Box b) {  
    System.out.println(b.getWidth());  
    System.out.println(b.getHeight());  
    System.out.println(b.getLength());  
}
```

Ahora es posible llamar al método *printBox* desde un programa del siguiente modo

```
Box myBox = new Box (1.0, 2.0, 3.0);  
Box.printBox(myBox);
```

Compare esto con una llamada al método normal *volume()*:

```
double result = myBox.volume();
```

## Arrays

Un *array* es un conjunto de valores del mismo tipo. Para hacer referencia a un elemento concreto del *array*, especificamos el nombre del *array* seguido del número de índice del elemento en particular entre corchetes []. Observe que el primer elemento del *array* está indexado como 0, no como 1.

En el siguiente ejemplo, queremos crear un *array* *c* que contenga 6 enteros (de *c*[0] a *c*[5]).

```
c[0]  2  
c[1]  5  
c[2]  1  
c[3] 43  
c[4]  8  
c[5] 66
```

Ésta es la declaración:

```
int[] c = new int[6]; //Reserva memoria para 6 enteros en el array c  
c[0] = 2;  
c[1] = 5;  
.....
```

O bien puede inicializar y declarar el *array* al mismo tiempo.

```
int [] c = {2, 5, 1, 43, 8, 66};
```

El *array* puede almacenar, no sólo tipos de datos primitivos, sino también objetos. Por ejemplo:

```
Box[] boxArray = new Box[3];
```

Cada objeto del *array* tiene un miembro de datos entero llamado *length* que devuelve el número de elementos de dicho *array*. Por ejemplo:

```
System.out.println(c.length); //se obtiene 6 como resultado
```

## Vectores

Los *arrays* presentan dos inconvenientes fundamentales:

- La longitud del *array* es fija. No es posible aumentarla o reducirla según se necesite. La memoria se asigna según la longitud, incluso si el *array* no utiliza todo el espacio.
- Sólo pueden contener elementos de un mismo tipo

Para salvar estas limitaciones, Java creó una clase llamada *Vector* en el paquete `java.util`

Al igual que un *array*, un vector es una colección de objetos, pero:

- Un vector puede aumentar su longitud automáticamente cuando lo necesite.
- Un vector puede contener elementos de distintos tipos siempre y cuando cada uno sea un objeto (pero NO un tipo de datos primitivo; no es posible incluir *integers* o *doubles* en un vector).
- Es posible acceder a un vector, agregarlo, modificarlo y consultarlo utilizando métodos de la clase *Vector*

A continuación, mostramos algunos de los métodos más utilizados:

- `Vector(int initialCapacity)`
- `add(Object element)`
- `elementAt(int index)`
- `get(int index)`
- `isEmpty()`
- `remove(int index)`
- `size()`

Tenga en cuenta que la longitud de un vector *v* es `v.size()`, no `v.length`

Aquí tenemos un ejemplo de uso de la clase *Vector*

```
import java.util.*;
```

```
class TestVector{  
    public static void main (String[] args) {  
        Vector bv = new Vector(2); //crea un vector con capacidad 2  
        System.println("Capacidad = " + bv.capacity() + " tamaño = " + bv.size());  
    }  
}
```

```

Box b1 = new Box(1.0, 2.0, 3.0);
Box b2 = new Box();
Box b3 = new Box(0.5, 1.5, 2.5);
bv.add(b1); //agrega b1 al final del vector
bv.add(b2); //agrega b2 al final del vector
bv.add(b3); //agrega b3 al final del vector

System.println ("Capacidad = " + bv.capacity() + " tamaño = " + bv.size());
System.println ("Ancho del primer elemento = " +
((Box)bv.firstElement()).getWidth());
System.println ("Ancho del segundo elemento = " +
((Box)bv.elementAt(1)).getWidth());
System.println ("Ancho del último elemento = " +
((Box)bv.lastElement()).getWidth());
bv.remove(2);
System.println("Capacidad = " + bv.capacity() + " tamaño = " + bv.size());
}

```

Los datos de salida son:

```

Capacidad = 2 tamaño = 0
Capacidad = 4 tamaño = 3
Ancho del primer elemento = 1.0
Ancho del segundo elemento = 0.0
Ancho del último elemento= 0.5
Capacidad = 4 tamaño 2

```

## Problemas de la clase adicional

Constructores y sobrecarga de métodos

1. Escriba los datos de salida que se obtendrían con el siguiente programa.

```

public class Item {
    public int value;
    public final int defaultValue = 5;

    public Item (int a, int b) {
        value = a * b;
    }

    public Item () {
        value = defaultValue;
    }

    public static void main (String args[]){
        Item a1 = new Item ();
        Item a2 = new Item (2,2);
        Item a3 = a1;
    }
}

```

```
    System.out.println ("El valor del elemento 1 es " + a1.value);
    System.out.println ("El valor del elemento 2 es " + a2.value);
    System.out.println ("El valor del elemento 3 es " + a3.value);
}

}
```

Escriba los datos que se obtendrían a la salida del programa anterior en el siguiente espacio:

Línea 1:

Línea 2:

Línea 3:

Campo *Static*

2. Modifique la definición de la clase anterior y agregue un campo de datos llamado `numOfItems`, que mantiene el registro del número de elementos de los que se ha creado una instancia. A continuación, imprima su valor al comienzo y al final del programa *main*. ¿Cuál es el valor de `numOfItems` en el ejemplo anterior?

*Array*

3. Conteste a las siguientes preguntas relacionadas con un *array* llamado *fracciones*.

- a. Defina una variable constante `ARRAY_SIZE` inicializada en 5.
- b. Declare un *array* con elementos `ARRAY_SIZE` de tipo *float* e inicialice todos los elementos en 0.
- c. Nombre el cuarto elemento del *array*.
- d. Asigne 1.5 al cuarto elemento del *array*.
- e. Sume todos los elementos del *array* utilizando una estructura de repetición *for*.

## Problema de diseño

Escriba un programa de Java que ayude al MIT a organizar las aulas. Cada aula cuenta con un número de aula (por ejemplo, 5-008), una capacidad y está disponible para su uso entre las 9 de la mañana y las 5 de la tarde, en bloques de una hora. Cada curso tiene un nombre, un ID de curso y un número de estudiantes. Defina un conjunto de clases de Java, incluidos sus datos de miembros y sus métodos, que represente esta situación y que ofrezca funcionalidad para que alguien pueda organizar las aulas.

Para crear una planificación para un aula en concreto, se necesita el ID del curso, la capacidad y el horario. Si una franja horaria ya se ha reservado para otra aula, el programa debería imprimir un mensaje de error. Si el aula no tiene la capacidad necesaria, el programa también deberá imprimir un mensaje de error.

Sugerencia: piense en las franjas horarias como si fueran un *array*.