

5.C. Métodos.

Sitio: [Formación Profesional a Distancia](#)

Curso: Programación

Libro: 5.C. Métodos.

Imprimido por: Iván Jiménez Utiel

Día: martes, 7 de enero de 2020, 22:18

Tabla de contenidos

1. Métodos.

1.1. Declaración de un método.

1.2. Cabecera de método.

1.3. Modificadores en la declaración de un método.

1.4. Parámetros en un método.

1.5. Cuerpo de un método.

1.6. Sobrecarga de métodos.

1.7. La referencia this.

1.8. Sobrecarga de operadores.

1.9. Métodos estáticos.

1. Métodos.

Como ya has visto anteriormente, los **métodos** son las herramientas que nos sirven para definir el comportamiento de un [objeto](#) en sus interacciones con otros objetos. Forman parte de la estructura interna del [objeto](#) junto con los atributos.

En el proceso de declaración de una [clase](#) que estás estudiando ya has visto cómo escribir la cabecera de la [clase](#) y cómo especificar sus atributos dentro del cuerpo de la [clase](#). Tan solo falta ya declarar los métodos, que estarán también en el interior del cuerpo de la [clase](#) junto con los atributos.

Los métodos suelen declararse después de los atributos. Aunque atributos y métodos pueden aparecer mezclados por todo el interior del cuerpo de la [clase](#) es aconsejable no hacerlo para mejorar la **claridad** y la **legibilidad** del código. De ese modo, cuando echemos un vistazo rápido al contenido de una [clase](#), podremos ver rápidamente los atributos al principio (normalmente ocuparán menos líneas de código y serán fáciles de reconocer) y cada uno de los métodos inmediatamente después. Cada [método](#) puede ocupar un número de líneas de código más o menos grande en función de la complejidad del proceso que pretenda implementar.

Los métodos representan la [interfaz](#) de una [clase](#). Son la forma que tienen otros objetos de comunicarse con un [objeto](#) determinado solicitándole cierta información o pidiéndole que lleve a cabo una determinada acción. Este modo de programar, como ya has visto en unidades anteriores, facilita mucho la tarea al desarrollador de aplicaciones, pues le permite abstraerse del contenido de las clases haciendo uso únicamente del [interfaz](#) (métodos).

Autoevaluación

¿Qué elementos forman la [interfaz](#) de un [objeto](#)?

- ☐ Los atributos del [objeto](#).
- ☐ Las variables locales de los métodos del [objeto](#).
- ☐ Los métodos.
- ☐ Los atributos estáticos de la [clase](#).

1.1. Declaración de un método.

La definición de un [método](#) se compone de dos partes:

- **Cabecera del [método](#)**, que contiene el nombre del [método](#) junto con el tipo devuelto, un conjunto de posibles modificadores y una lista de parámetros.
- **Cuerpo del [método](#)**, que contiene las sentencias que implementan el comportamiento del [método](#) (incluidas posibles sentencias de declaración de variables locales).

Los **elementos mínimos** que deben aparecer en la declaración de un [método](#) son:

- El tipo devuelto por el [método](#).
- El nombre del [método](#).
- Los paréntesis.
- El cuerpo del [método](#) entre llaves: { }.

Por ejemplo, en la [clase Punto](#) que se ha estado utilizando en los apartados anteriores podrías encontrar el siguiente [método](#):

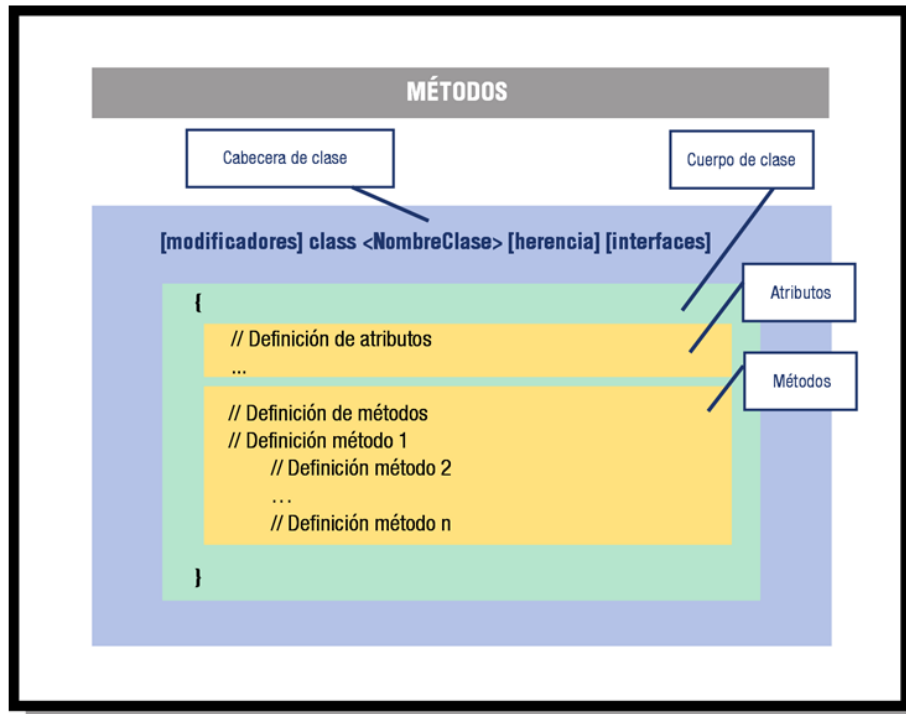
```
int obtenerX ()  
{  
    // Cuerpo del método  
    ...  
}
```

Donde:

- El tipo devuelto por el [método](#) es [int](#).
- El nombre del [método](#) es [obtenerX](#).
- No recibe ningún parámetro: aparece una lista vacía entre paréntesis: ().
- El cuerpo del [método](#) es todo el código que habría encerrado entre llaves: { }.

Dentro del cuerpo del [método](#) podrás encontrar declaraciones de variables, sentencias y todo tipo de estructuras de [control](#) (bucles, condiciones, etc.) que has estudiado en los apartados anteriores.

Ahora bien, la declaración de un [método](#) puede incluir algunos elementos más. Vamos a estudiar con detalle cada uno de ellos.



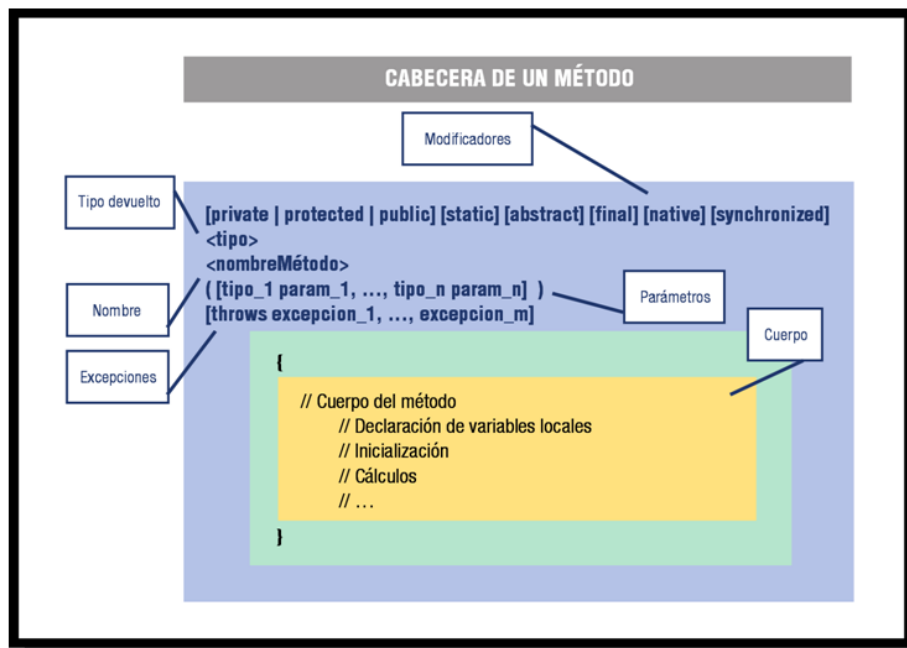
1.2. Cabecera de método.

La declaración de un [método](#) puede incluir los siguientes elementos:

1. **Modificadores** (como por ejemplo los ya vistos **public** o **private**, más algunos otros que irás conociendo poco a poco). No es obligatorio incluir modificadores en la declaración.
2. El **tipo devuelto** (o tipo de retorno), que consiste en el [tipo de dato](#) (primitivo o referencia) que el [método](#) devuelve tras ser ejecutado. Si eliges **void** como tipo devuelto, el [método](#) no devolverá ningún valor.
3. El **nombre del [método](#)**, aplicándose para los nombres el mismo convenio que para los atributos.
4. Una **lista de parámetros** separados por comas y entre paréntesis donde cada parámetro debe ir precedido por su tipo. Si el [método](#) no tiene parámetros la lista estará vacía y únicamente aparecerán los paréntesis.
5. Una **lista de excepciones** que el [método](#) puede lanzar. Se utiliza la palabra reservada **throws** seguida de una lista de nombres de excepciones separadas por comas. No es obligatorio que un [método](#) incluya una lista de excepciones, aunque muchas veces será conveniente. En unidades anteriores ya has trabajado con el concepto de [excepción](#) y más adelante volverás a hacer uso de ellas.
6. El **cuerpo del [método](#)**, encerrado entre llaves. El cuerpo contendrá el código del [método](#) (una lista sentencias y estructuras de [control](#) en lenguaje Java) así como la posible declaración de variables locales.

La [sintaxis](#) general de la cabecera de un [método](#) podría entonces quedar así:

```
[private | protected | public] [static] [abstract] [final] [native] [synchronized]  
<tipo> <nombreMétodo> ( [<lista_parametros>] )  
[throws <lista_excepciones>]
```



Como sucede con todos los identificadores en Java (variables, clases, objetos, métodos, etc.), puede usarse cualquier identificador que cumpla las normas. Ahora bien, para mejorar la legibilidad del código, se ha establecido el siguiente convenio para nombrar los métodos: **utilizar un verbo en minúscula o bien un nombre formado por varias palabras que comience por un verbo en minúscula, seguido por adjetivos, nombres, etc. los cuales sí aparecerán en mayúsculas.**

Algunos ejemplos de métodos que siguen este convenio podrían ser: ejecutar, romper, mover, subir, responder, obtenerX, establecerValor, estaVacio, estaLleno, moverFicha, subirPalanca, responderRapido, girarRuedaIzquierda, abrirPuertaDelantera, CambiarMarcha, etc.

En el ejemplo de la [clase Punto](#), puedes observar cómo los métodos `obtenerX` y `obtenerY` siguen el convenio de nombres para los métodos, devuelven en ambos casos un tipo `int`, su lista de parámetros es vacía (no tienen parámetros) y no lanzan ningún tipo de [excepción](#):

```
* int obtenerX ()
* int obtenerY ()
```

Autoevaluación

¿Con cuál de los siguientes modificadores no puede ser declarado un [método](#) en Java?

- ☐ private.
- ☐ extern.
- ☐ static.
- ☐ public.

1.3. Modificadores en la declaración de un método.

En la declaración de un [método](#) también pueden aparecer modificadores (como en la declaración de la [clase](#) o de los atributos). Un [método](#) puede tener los siguientes tipos de modificadores:

- **Modificadores de acceso.** Son los mismos que en el caso de los atributos (por omisión o de paquete, [public](#), [private](#) y [protected](#)) y tienen el mismo cometido (acceso al [método](#) sólo por parte de clases del mismo paquete, o por cualquier parte del programa, o sólo para la propia [clase](#), o también para las subclases).
- **Modificadores de contenido.** Son también los mismos que en el caso de los atributos ([static](#) y [final](#)), aunque su significado no es el mismo.
- **Otros modificadores** (no son aplicables a los atributos, sólo a los métodos): [abstract](#), [native](#), [synchronized](#).

Un [método static](#) es un [método](#) cuya implementación es igual para todos los objetos de la [clase](#) y sólo tendrá acceso a los atributos estáticos de la [clase](#) (dado que se trata de un [método](#) de [clase](#) y no de [objeto](#), sólo podrá acceder a la información de [clase](#) y no la de un [objeto](#) en particular). Este tipo de métodos pueden ser llamados sin necesidad de tener un [objeto](#) de la [clase](#) instanciado.

En Java un ejemplo típico de métodos estáticos se encuentra en la [clase Math](#), cuyos métodos son todos estáticos ([Math.abs](#), [Math.sin](#), [Math.cos](#), etc.). Como habrás podido comprobar en este ejemplo, la llamada a métodos estáticos se hace normalmente usando el nombre de la propia [clase](#) y no el de una [instancia](#) ([objeto](#)), pues se trata realmente de un [método](#) de [clase](#). En cualquier caso, los objetos también admiten la invocación de los métodos estáticos de su [clase](#) y funcionaría correctamente.

Un [método final](#) es un [método](#) que no permite ser sobrescrito por las clases descendientes de la [clase](#) a la que pertenece el [método](#). Volverás a ver este modificador cuando estudies en detalle la [herencia](#).

El modificador [native](#) es utilizado para señalar que un [método](#) ha sido implementado en código nativo (en un lenguaje que ha sido compilado a lenguaje máquina, como por ejemplo [C](#) o [C++](#)). En estos casos simplemente se indica la cabecera del [método](#), pues no tiene cuerpo escrito en Java.

Un [método abstract](#) ([método abstracto](#)) es un [método](#) que no tiene implementación (el cuerpo está vacío). La implementación será realizada en las clases descendientes. Un [método](#) sólo puede ser declarado como [abstract](#) si se encuentra dentro de una [clase abstract](#). También volverás a este modificador en unidades posteriores cuando trabajes con la [herencia](#).

Por último, si un [método](#) ha sido declarado como [synchronized](#), el entorno de ejecución obligará a que cuando un proceso esté ejecutando ese [método](#), el resto de procesos que tengan que llamar a ese mismo [método](#) deberán esperar a que el otro proceso termine. Puede resultar útil si sabes que un determinado [método](#) va a poder ser llamado concurrentemente por varios procesos a la vez. Por ahora no lo vas a necesitar.

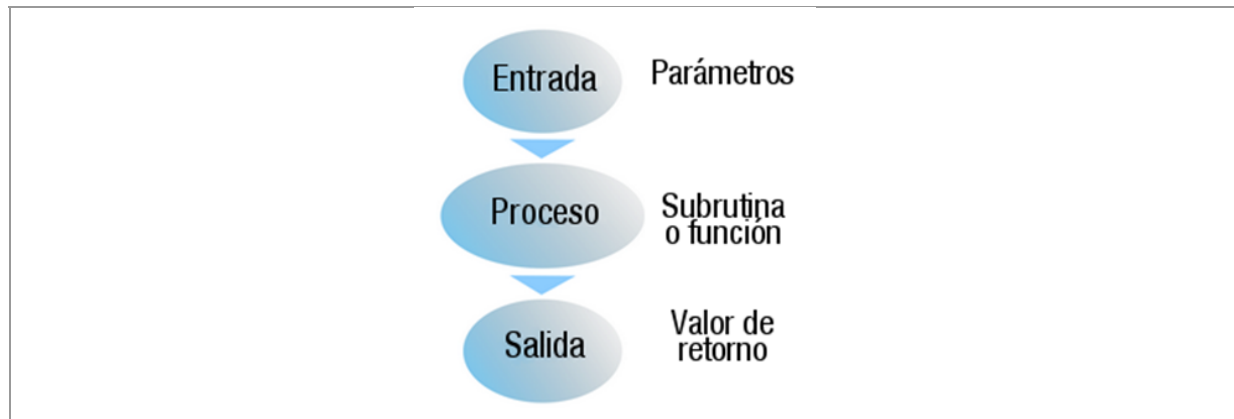
Dada la cantidad de modificadores que has visto hasta el momento y su posible aplicación en la declaración de clases, atributos o métodos, veamos un resumen de todos los que has visto y en qué casos pueden aplicarse:

Cuadro de aplicabilidad de los modificadores.			
	Clase	Atributo	Método
Sin modificador (paquete)	Sí	Sí	Sí
public	Sí	Sí	Sí
private		Sí	Sí
protected	Sí	Sí	Sí
static		Sí	Sí
final	Sí	Sí	Sí
synchronized			Sí
native			Sí
abstract	Sí		Sí

1.4. Parámetros en un método.

La lista de parámetros de un [método](#) se coloca tras el nombre del [método](#). Esta lista estará constituida por pares de la forma "<tipoParametro> <nombreParametro>". Cada uno de esos pares estará separado por comas y la lista completa estará encerrada entre paréntesis:

```
<tipo> nombreMetodo ( <tipo_1> <nombreParametro_1>, <tipo_2> <nombreParametro_2>,  
..., <tipo_n> <nombreParametro_n> )
```



Si la lista de parámetros es vacía, tan solo aparecerán los paréntesis:

```
<tipo> <nombreMetodo> ( )
```

A la hora de declarar un [método](#), debes tener en cuenta:

- Puedes incluir cualquier cantidad de parámetros. Se trata de una decisión del programador, pudiendo ser incluso una lista vacía.
- Los parámetros podrán ser de cualquier tipo (tipos primitivos, referencias, objetos, arrays, etc.).
- No está permitido que el nombre de una [variable](#) local del [método](#) coincida con el nombre de un parámetro.
- No puede haber dos parámetros con el mismo nombre. Se produciría ambigüedad.
- Si el nombre de algún parámetro coincide con el nombre de un [atributo](#) de la [clase](#), éste será ocultado por el parámetro. Es decir, al indicar ese nombre en el código del [método](#) estarás haciendo referencia al parámetro y no al [atributo](#). Para poder acceder al [atributo](#) tendrás que hacer uso del operador de autorreferencia [this](#), que verás un poco más adelante.
- En Java el paso de parámetros es siempre por valor, excepto en el caso de los tipos referenciados (por ejemplo los objetos) en cuyo caso se está pasando efectivamente una referencia. La referencia (el [objeto](#) en sí mismo) no podrá ser cambiada pero sí elementos de su interior (atributos) a través de sus métodos o por acceso directo si se trata de un miembro público.

Es posible utilizar una construcción especial llamada **varargs** (argumentos variables) que permite que un método pueda tener un número variable de parámetros. Para utilizar este mecanismo se colocan unos puntos suspensivos (tres puntos: "...") después del tipo del cual se puede tener una lista variable de argumentos, un espacio en blanco y a continuación el nombre del parámetro que aglutinará la lista de argumentos variables.

`<tipo> <nombreMetodo> (<tipo>... <nombre>)`

Es posible además mezclar el uso de **varargs** con parámetros fijos. En tal caso, la lista de parámetros variables debe aparecer al final (y sólo puede aparecer una).

En realidad se trata una manera transparente de pasar un array con un número variable de elementos para no tener que hacerlo manualmente. Dentro del método habrá que ir recorriendo el array para ir obteniendo cada uno de los elementos de la lista de argumentos variables.

Para saber más

Si quieres ver algunos ejemplos de cómo utilizar el mecanismo de argumentos variables, puedes echar un vistazo a los siguientes enlaces (en inglés):

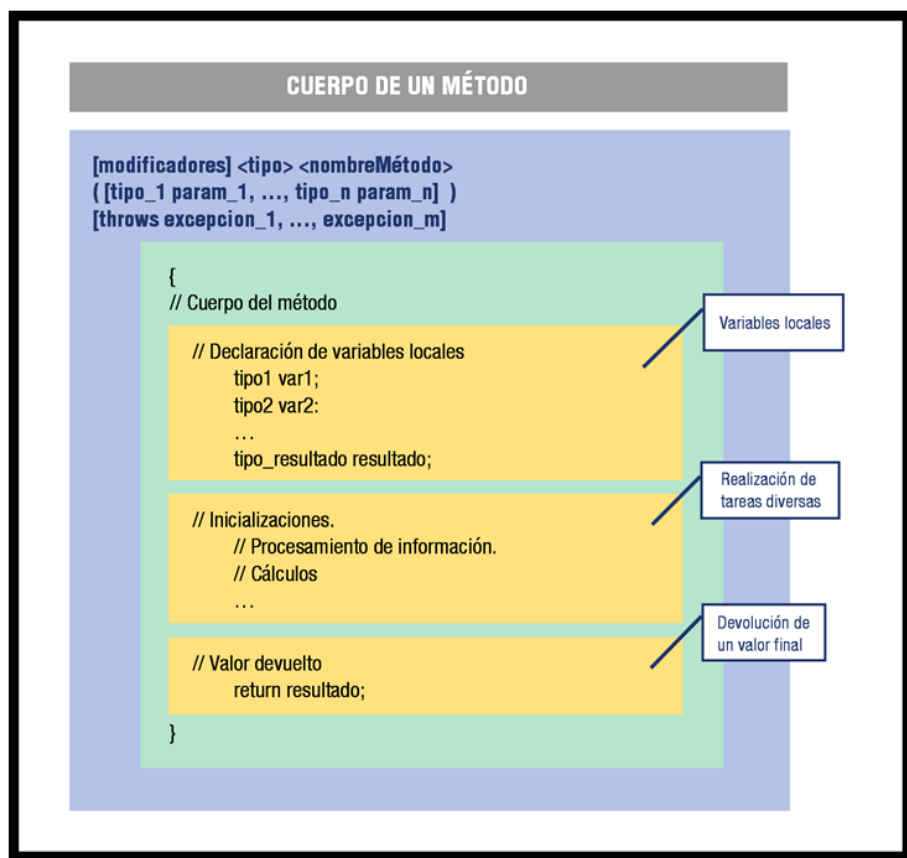
[Vargars.](#)

[Passing Information to a Method or a Constructor.](#)

1.5. Cuerpo de un método.

El interior de un [método](#) (cuerpo) está compuesto por una serie de sentencias en lenguaje Java:

- Sentencias de **declaración de variables locales** al [método](#).
- Sentencias que implementan la **lógica del método** (estructuras de [control](#) como bucles o condiciones; utilización de métodos de otros objetos; cálculo de expresiones matemáticas, lógicas o de cadenas; creación de nuevos objetos, etc.). Es decir, todo lo que has visto en las unidades anteriores.
- Sentencia de **devolución del valor de retorno** ([return](#)). Aparecerá al final del [método](#) y es la que permite devolver la información que se le ha pedido al [método](#). Es la última parte del proceso y la forma de comunicarse con la parte de código que llamó al [método](#) (paso de mensaje de vuelta). Esta sentencia de devolución siempre tiene que aparecer al final del [método](#). Tan solo si el tipo devuelto por el [método](#) es [void](#) (vacío) no debe aparecer (pues no hay que devolver nada al código llamante).



En el ejemplo de la [clase Punto](#), tenías los métodos [obtenerX](#) y [obtenerY](#).

```
int obtenerX ()
```

```
{
```

```
    return x;
```

```
}
```

```
int obtenerY ()
```

```
{
```

```
    return y;
```

```
}
```

Veamos uno de ellos:

En ambos casos lo único que hace el [método](#) es precisamente devolver un valor (utilización de la sentencia **return**). No recibe parámetros (mensajes o información de entrada) ni hace cálculos, ni obtiene resultados intermedios o finales. Tan solo devuelve el contenido de un [atributo](#). Se trata de uno de los métodos más sencillos que se pueden implementar: un [método](#) que devuelve el valor de un [atributo](#). En inglés se les suele llamar métodos de tipo **get**, que en inglés significa **obtener**.

Además de esos dos métodos, la [clase](#) también disponía de otros dos que sirven para la función opuesta (**establecerX** y **establecerY**). Veamos uno de ellos:

```
void establecerX (int vx)
```

```
{
```

```
    x= vx;
```

```
}
```

En este caso se trata de pasar un valor al [método](#) (parámetro **vx** de tipo **int**) el cual será utilizado para modificar el contenido del [atributo](#) **x** del [objeto](#). Como habrás podido comprobar, ahora no se devuelve ningún valor (el tipo devuelto es **void** y no hay sentencia **return**). En inglés se suele hablar de métodos de tipo **set**, que en inglés significa poner o fijar (**establecer** un valor). El [método](#) **establecerY** es prácticamente igual pero para establecer el valor del [atributo](#) **y**.

Normalmente el código en el interior de un [método](#) será algo más complejo y estará formado un conjunto de sentencias en las que se realizarán cálculos, se tomarán decisiones, se repetirán acciones, etc. Puedes ver un ejemplo más completo en el siguiente ejercicio.

Ejercicio resuelto

Vamos a seguir ampliando la [clase](#) en la que se representa un rectángulo en el plano ([clase Rectangulo](#)). Para ello has pensado en los siguientes métodos **públicos**:

- Métodos **obtenerNombre** y **establecerNombre**, que permiten el acceso y modificación del [atributo](#) **nombre** del rectángulo.
- [Método](#) **calcularSuperficie**, que calcula el área encerrada por el rectángulo.
- [Método](#) **calcularPerímetro**, que calcula la longitud del perímetro del rectángulo.
- [Método](#) **desplazar**, que mueve la ubicación del rectángulo en el plano en una cantidad X (para el eje X) y otra cantidad Y (para el eje Y). Se trata simplemente de sumar el desplazamiento X a las coordenadas x1 y x2, y el desplazamiento Y a las coordenadas y1 y y2. Los **parámetros** de entrada de este [método](#) serán por

tanto X e Y, de tipo **double**.

- **Método obtenerNumRectangulos**, que devuelve el número de rectángulos creados hasta el momento.

Incluye la implementación de cada uno de esos métodos en la **clase Rectangulo**.

Solución

En el caso del **método obtenerNombre**, se trata simplemente de devolver el valor del **atributo nombre**:

```
public String obtenerNombre () {  
  
    return nombre;  
  
}
```

Para el implementar el **método establecerNombre** también es muy sencillo. Se trata de modificar el contenido del **atributo nombre** por el valor proporcionado a través de un parámetro de entrada:

```
public void establecerNombre (String nom) {  
  
    nombre= nom;  
  
}
```

Los métodos de cálculo de superficie y perímetro no van a recibir ningún parámetro de entrada, tan solo deben realizar cálculos a partir de los atributos contenidos en el **objeto** para obtener los resultados perseguidos. En cada caso habrá que aplicar la expresión matemática apropiada:

- En el caso de la superficie, habrá que calcular la longitud de la **base** y la **altura** del rectángulo a partir de las coordenadas de las esquinas inferior izquierda (x1, y1) y superior derecha (x2, y2) de la figura. La base sería la diferencia entre x2 y x1, y la altura la diferencia entre y2 e y1. A continuación tan solo tendrías que utilizar la consabida fórmula de "base por altura", es decir, una multiplicación.
- En el caso del perímetro habrá también que calcular la longitud de la **base** y de la **altura** del rectángulo y a continuación sumar dos veces la longitud de la base y dos veces la longitud de la altura.

En ambos casos el resultado final tendrá que ser devuelto a través de la sentencia return. También es aconsejable en ambos casos la utilización de variables locales para almacenar los cálculos intermedios (como la base o la altura).

```
public double calcularSuperficie () {  
  
    double area, base, altura; // Variables locales  
  
    // Cálculo de la base  
  
    base= x2-x1;  
  
    // Cálculo de la altura  
  
    altura= y2-y1;  
  
    // Cálculo del área  
  
    area= base * altura;
```

```

    // Devolución del valor de retorno

    return area;
}

public double calcularPerimetro () {

    double perimetro, base, altura; // Variables locales

    // Cálculo de la base

    base= x2-x1;

    // Cálculo de la altura

    altura= y2-y1;

    // Cálculo del perímetro

    perimetro= 2*base + 2*altura;

    // Devolución del valor de retorno

    return perimetro;
}

```

En el caso del [método](#) **desplazar**, se trata de modificar:

- Los contenidos de los atributos x1 y x2 sumándoles el parámetro X,
- Los contenidos de los atributos y1 e y2 sumándoles el parámetro Y.

```

·    public void desplazar (double X, double Y) {
·
·        // Desplazamiento en el eje X
·
·        x1= x1 + X;
·
·        x2= x2 + X;
·
·        // Desplazamiento en el eje Y
·
·        y1= y1 + Y;
·
·        y2= y2 + Y;
·
·    }

```

En este caso no se devuelve ningún valor (tipo devuelto vacío: void).

Por último, el [método](#) **obtenerNumRectangulos** simplemente debe devolver el valor del [atributo](#) **numRectangulos**. En este caso es razonable plantearse que este [método](#) podría ser más bien un [método](#) de [clase](#) (estático) más que un [método](#) de [objeto](#), pues en realidad es una característica de la [clase](#) más que algún [objeto](#) en particular. Para ello tan solo tendrías que utilizar el modificador de acceso static:

```

public static int obtenerNumRectangulos () {

```

```
    return numRectangulos;
}
```

Veamos todo el código:

```
**-----
* Clase Rectangulo
-----*/

public class Rectangulo {

    // Atributos de clase

    private static int numRectangulos;           // Número total de rectángulos creados
    public static final String nombreFigura= "Rectángulo";    // Nombre de la clase
    public static final double PI= 3.1416;         // Constante PI

    // Atributos de objeto

    private String nombre;    // Nombre del rectángulo
    public double x1, y1;     // Vértice inferior izquierdo
    public double x2, y2;     // Vértice superior derecho


    // Método obtenerNombre

    public String obtenerNombre () {

        return nombre;

    }


    // Método establecerNombre

    public void establecerNombre (String nom) {

        nombre= nom;

    }


    // Método CalcularSuperficie
```



```
public double CalcularSuperficie () {  
    double area, base, altura;  
  
    // Cálculo de la base  
    base= x2-x1;  
  
    // Cálculo de la altura  
    altura= y2-y1;  
  
    // Cálculo del área  
    area= base * altura;  
  
    // Devolución del valor de retorno  
    return area;  
}
```

```
// Método CalcularPerimetro  
public double CalcularPerimetro () {  
    double perimetro, base, altura;  
  
    // Cálculo de la base  
    base= x2-x1;  
  
    // Cálculo de la altura  
    altura= y2-y1;  
  
    // Cálculo del perímetro  
    perimetro= 2*base + 2*altura;  
  
    // Devolución del valor de retorno  
    return perimetro;
```

```
}

// Método desplazar
public void desplazar (double X, double Y) {

    // Desplazamiento en el eje X
    x1= x1 + X;
    x2= x2 + X;

    // Desplazamiento en el eje Y
    y1= y1 + Y;
    y2= y2 + Y;
}

// Método obtenerNumRectangulos
public static int obtenerNumRectangulos () {
    return numRectangulos;
}

}
```

1.6. Sobrecarga de métodos.

En principio podrías pensar que un método puede aparecer una sola vez en la declaración de una clase (no se debería repetir el mismo nombre para varios métodos). Pero no tiene porqué siempre suceder así. Es posible tener varias versiones de un mismo método (varios métodos con el mismo nombre) gracias a la **sobrecarga de métodos**.



El lenguaje Java soporta la característica conocida como **sobrecarga de métodos**. Ésta permite declarar en una misma clase varias versiones del mismo método con el mismo nombre. La forma que tendrá el compilador de distinguir entre varios métodos que tengan el mismo nombre será mediante la lista de parámetros del método: si el método tiene una lista de parámetros diferente, será considerado como un método diferente (aunque tenga el mismo nombre) y el analizador léxico no producirá un error de compilación al encontrar dos nombres de método iguales en la misma clase.

Imagínate que estás desarrollando una clase para escribir sobre un lienzo que permite utilizar diferentes tipografías en función del tipo de información que se va a escribir. Es probable que necesitemos un método diferente según se vaya a pintar un número entero (int), un número real (double) o una cadena de caracteres (String). Una primera opción podría ser definir un nombre de método diferente dependiendo de lo que se vaya a escribir en el lienzo. Por ejemplo:

- Método `pintarEntero (int entero)`.
- Método `pintarReal (double real)`.
- Método `pintarCadena (double String)`.
- Método `pintarEnteroCadena (int entero, String cadena)`.

Y así sucesivamente para todos los casos que desees contemplar...

La posibilidad que te ofrece la sobrecarga es utilizar un mismo nombre para todos esos métodos (dado que en el fondo hacen lo mismo: pintar). Pero para poder distinguir unos de otros será necesario que siempre exista alguna diferencia entre ellos en las listas de parámetros (bien en el número de parámetros, bien en el tipo de los parámetros). Volviendo al ejemplo anterior, podríamos utilizar un mismo nombre, por ejemplo `pintar`, para todos los métodos anteriores:

- Método `pintar (int entero)`.
- Método `pintar (double real)`.
- Método `pintar (double String)`.
- Método `pintar (int entero, String cadena)`.

En este caso el [compilador](#) no va a generar ningún error pues se cumplen las normas ya que unos métodos son perfectamente distinguibles de otros (a pesar de tener el mismo nombre) gracias a que tienen listas de parámetros diferentes.

Lo que sí habría producido un error de compilación habría sido por ejemplo incluir otro [método pintar](#) (`int entero`), pues es imposible distinguirlo de otro [método](#) con el mismo nombre y con la misma lista de parámetros (ya existe un [método pintar](#) con un único parámetro de tipo `int`).

También debes tener en cuenta que el **tipo devuelto** por el [método](#) no es considerado a la hora de identificar un [método](#), así que un tipo devuelto diferente no es suficiente para distinguir un [método](#) de otro. Es decir, no podrías definir dos métodos exactamente iguales en nombre y lista de parámetros e intentar distinguirlos indicando un tipo devuelto diferente. El [compilador](#) producirá un error de duplicidad en el nombre del [método](#) y no te lo permitirá.

Es conveniente no abusar de sobrecarga de métodos y utilizarla con cierta moderación (cuando realmente puede beneficiar su uso), dado que podría hacer el código menos legible.

Autoevaluación

En una [clase](#) Java puedes definir tantos métodos con el mismo nombre como desees y sin ningún tipo de restricción pues el lenguaje soporta la sobrecarga de métodos y el [compilador](#) sabrá distinguir unos métodos de otros. ¿Verdadero o falso?

- ☐ Verdadero.
- ☐ Falso.

1.7. La referencia this.

La palabra reservada **this** consiste en una referencia al [objeto](#) actual. El uso de este operador puede resultar muy útil a la hora de evitar la ambigüedad que puede producirse entre el nombre de un parámetro de un [método](#) y el nombre de un [atributo](#) cuando ambos tienen el mismo identificador (mismo nombre). En tales casos el parámetro "oculta" al [atributo](#) y no tendríamos acceso directo a él (al escribir el identificador estaríamos haciendo referencia al parámetro y no al [atributo](#)). En estos casos la referencia **this** nos permite acceder a estos atributos ocultos por los parámetros.

Dado que **this** es una referencia a la propia [clase](#) en la que te encuentras en ese momento, puedes acceder a sus atributos mediante el operador punto (.) como sucede con cualquier otra [clase](#) u [objeto](#). Por tanto, en lugar de poner el nombre del [atributo](#) (que estos casos haría referencia al parámetro), podrías escribir **this.nombreAtributo**, de manera que el [compilador](#) sabrá que te estás refiriendo al [atributo](#) y se eliminará la ambigüedad.

En el ejemplo de la [clase Punto](#), podríamos utilizar la referencia **this** si el nombre del parámetro del [método](#) coincidiera con el del [atributo](#) que se desea modificar. Por ejemplo

```
void establecerX (int x)
{
    this.x= x;
}
```

En este caso ha sido indispensable el uso de **this**, pues si no sería imposible saber en qué casos te estás refiriendo al parámetro **x** y en cuáles al [atributo](#) **x**. Para el [compilador](#) el identificador **x** será siempre el parámetro, pues ha "ocultado" al [atributo](#).

En algunos casos puede resultar útil hacer uso de la referencia **this** aunque no sea necesario, pues puede ayudar a mejorar la legibilidad del código.

Para saber más

Puedes echar un vistazo al artículo general sobre la referencia **this** en los manuales de Oracle (en inglés):

[Using the this Keyword.](#)

Autoevaluación

La referencia **this en Java resulta muy útil cuando se quieren utilizar en un [método](#) nombres de parámetros que coinciden con los nombres de variables locales del [método](#). ¿Verdadero o falso?**

- ☐ Verdadero.
- ☐ Falso.

Ejercicio resuelto

Modificar el [método](#) **obtenerNombre** de la [clase](#) **Rectangulo** de ejercicios anteriores utilizando la referencia **this**.

Solución

Si utilizamos la referencia **this** en este [método](#), entonces podremos utilizar como identificador del parámetro el mismo identificador que tiene el [atributo](#) (aunque no tiene porqué hacerse si no se desea):

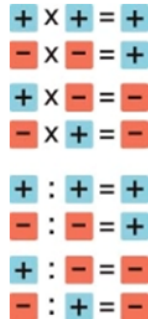
```
public void establecerNombre (String nombre) {
```

```
    this.nombre= nombre;
```

```
}
```

1.8. Sobrecarga de operadores.

Del mismo modo que hemos visto la posibilidad de sobrecargar métodos (disponer de varias versiones de un [método](#) con el mismo nombre cambiando su lista de parámetros), podría plantearse también la opción de sobrecargar operadores del lenguaje tales como +, -, *, (), <, >, etc. para darles otro significado dependiendo del tipo de objetos con los que vaya a operar.



En algunos casos puede resultar útil para ayudar a mejorar la legibilidad del código, pues esos operadores resultan muy intuitivos y pueden dar una idea rápida de cuál es su funcionamiento.

Un típico ejemplo podría ser el de la sobrecarga de operadores aritméticos como la suma (+) o el producto (*) para operar con fracciones. Si se definen objetos de una [clase Fracción](#) (que contendrá los atributos numerador y denominador) podrían sobrecargarse los operadores aritméticos (habría que redefinir el operador suma (+) para la suma, el operador asterisco (*) para el producto, etc.) para esta [clase](#) y así podrían utilizarse para sumar o multiplicar objetos de tipo **Fracción** mediante el algoritmo específico de suma o de producto del [objeto Fracción](#) (pues esos operadores no están preparados en el lenguaje para operar con esos objetos).

En algunos lenguajes de programación como por ejemplo [C++](#) o [C#](#) se permite la sobrecarga, pero no es algo soportado en todos los lenguajes. ¿Qué sucede en el caso concreto de Java?

El lenguaje Java no soporta la sobrecarga de operadores.

En el ejemplo anterior de los objetos de tipo **Fracción**, habrá que declarar métodos en la [clase Fracción](#) que se encarguen de realizar esas operaciones, pero no lo podremos hacer sobrecargando los operadores del lenguaje (los símbolos de la suma, resta, producto, etc.). Por ejemplo:

```
public Fraccion sumar (Fraccion sumando)

public Fraccion multiplicar (Fraccion multiplicando)
```

Y así sucesivamente...

Dado que en este módulo se está utilizando el lenguaje Java para aprender a programar, no podremos hacer uso de esta funcionalidad. Más adelante, cuando aprendas a programar en otros lenguajes, es posible que sí tengas la posibilidad de utilizar este recurso.

Autoevaluación

La sobrecarga de operadores en Java permite “rescribir” el significado de operadores del lenguaje tales como +, -, *, <, >, etc. Esto puede resultar muy útil a la hora de mejorar la legibilidad del código cuando definimos por ejemplo nuevos objetos matemáticos (números racionales, números complejos, conjuntos, etc.). ¿Verdadero o falso?

☐ Verdadero.

☐ Falso.

1.9. Métodos estáticos.

Como ya has visto en ocasiones anteriores, un **método estático** es un **método** que puede ser usado directamente desde la **clase**, sin necesidad de tener que crear una **instancia** para poder utilizar al **método**. También son conocidos como **métodos de clase** (como sucedía con los **atributos de clase**), frente a los **métodos de objeto** (es necesario un **objeto** para poder disponer de ellos).

Los métodos estáticos no pueden manipular atributos de instancias (objetos) sino atributos estáticos (de **clase**) y suelen ser utilizados para realizar operaciones comunes a todos los objetos de la **clase**, más que para una **instancia** concreta.

Algunos ejemplos de operaciones que suelen realizarse desde métodos estáticos:

- Acceso a atributos específicos de **clase**: incremento o decremento de contadores internos de la **clase** (no de instancias), acceso a un posible **atributo** de nombre de la **clase**, etc.
- Operaciones genéricas relacionadas con la **clase** pero que no utilizan atributos de **instancia**. Por ejemplo una **clase** NIF (o DNI) que permite trabajar con el DNI y la letra del NIF y que proporciona funciones adicionales para calcular la letra NIF de un número de DNI que se le pase como parámetro. Ese **método** puede ser interesante para ser usado desde fuera de la **clase** de manera independiente a la existencia de objetos de tipo NIF.

En la biblioteca de Java es muy habitual encontrarse con clases que proporcionan métodos estáticos que pueden resultar muy útiles para cálculos auxiliares, conversiones de tipos, etc. Por ejemplo, la mayoría de las clases del paquete **java.lang** que representan tipos (**Integer**, **String**, **Float**, **Double**, **Boolean**, etc.) ofrecen métodos estáticos para hacer conversiones. Aquí tienes algunos ejemplos:

- **static String valueOf (int i)**. Devuelve la representación en formato **String** (cadena) de un valor **int**. Se trata de un **método** que no tiene que ver nada en absoluto con instancias de concretas de **String**, sino de un **método** auxiliar que puede servir como herramienta para ser usada desde otras clases. Se utilizaría directamente con el nombre de la **clase**. Por ejemplo:

```
String enteroCadena= String.valueOf (23).
```

- **static String valueOf (float f)**. Algo similar para un valor de tipo **float**. Ejemplo de uso:

```
String floatCadena= String.valueOf (24.341).
```

- **static int parseInt (String s)**. En este caso se trata de un **método estático** de la **clase** **Integer**. Analiza la cadena pasada como parámetro y la transforma en un **int**. Ejemplo de uso:

```
int cadenaEntero= Integer.parseInt ("-12").
```

Todos los ejemplos anteriores son casos en los que se utiliza directamente la **clase** como una especie de **caja de herramientas** que contiene métodos que pueden ser utilizados desde cualquier parte, por eso suelen ser

métodos públicos.

Para saber más

Puedes echar un vistazo a algunas clases del paquete `java.lang` (por ejemplo `Integer`, `String`, `Float`, `Double`, `Boolean` y `Math`) y observar la gran cantidad de métodos estáticos que ofrecen para ser utilizados sin necesidad de tener que crear objetos de esas clases:

[Package java.lang.](#)