Métodos

Los métodos definen
el comportamiento de los objetos de una clase dada
(lo que podemos hacer con los objetos de esa clase)

Los métodos exponen la interfaz de una clase.

Un método define la secuencia de sentencias
que se ejecuta para llevar a cabo una operación:

La implementación de la clase se oculta del exterior.

Los métodos…

Nos dicen cómo hemos de usar los objetos de una clase.

Nos permiten cambiar la implementación de una clase sin tener que modificar su interfaz (esto es, sin tener que modificar el código que utiliza objetos de la clase cuya implementación cambiamos)

Ejemplo:
Utilizar un algoritmo más eficiente
para resolver un problema concreto
sin tener que tocar el código del resto del programa.

Definición de métodos

Sintaxis en Java

```
modificadores tipo nombre (parámetros)
{
 cuerpo
}
```

La estructura de un método se divide en:

- **Cabecera** (determina su interfaz)

  modificadores tipo nombre (*parámetros*)

- **Cuerpo** (define su implementación)

```
{
  // Declaraciones de variables
  ...
  // Sentencias ejecutables
  ...
  // Devolución de un valor (opcional)
  ...
}
```

En el cuerpo del método se implementa el algoritmo necesario para realizar la tarea de la que el método es responsable.

El cuerpo de un método se puede interpretar como una caja negra que contiene su implementación:

El método oculta los detalles de implementación.

Cuando utilizamos un método, sólo nos interesa su interfaz.

Ejemplo
El punto de entrada a una aplicación escrita en Java

```
public static void main (String[] args)
{
   ...
}
```

- Como todo en Java, ha de ser un miembro de una clase (esto es, estar definido en el interior de una clase).

- El modificador de acceso `public` indica que se puede acceder a este miembro de la clase desde el exterior de la clase.

- El modificador `static` indica que se trata de un método de clase (un método común para todos los objetos de la clase).

- La palabra reservada `void` indica que, en este caso el método `main` no devuelve ningún valor.

  En general, no obstante, los métodos son capaces de devolver un valor al terminar su ejecución.

- Los paréntesis nos indican que se trata de un método: Lo que aparece entre paréntesis son los parámetros del método (en este caso, un vector de cadenas de caracteres, que se representan en Java con objetos de tipo `String`).

- El cuerpo del método va delimitado por llaves `{ }`.

CONVENCIÓN
El texto correspondiente al código que se ejecuta al invocar un método se sangra con respecto a la posición de las llaves que delimitan el cuerpo del método.

La cabecera de un método

La cabecera de un método determina su interfaz

- **Modificadores de acceso** (p.ej. `public` o `private`)
  Determinan desde dónde se puede utilizar el método.

- **Tipo devuelto** (cualquier tipo primitivo, no primitivo o `void`)
  Indica de que tipo es la salida del método, el resultado que se obtiene tras llamar al método desde el exterior.

  NOTA:
  `void` se emplea cuando el método no devuelve ningún valor.

- **Nombre del método**
  Identificador válido en Java

  CONVENCIÓN:
  En Java, los nombres de métodos comienzan con minúscula.

- **Parámetros formales**
  Entradas que necesita el método para realizar la tarea de la que es responsable.

  MÉTODOS SIN PARÁMETROS:
  Cuando un método no tiene entradas, hay que poner `()`

El cuerpo de un método

El cuerpo de un método define su implementación:

NB: Como cualquier bloque de código en Java,
el cuerpo de un método ha de estar delimitado por llaves `{ }`

La signatura de un método

El nombre de un método, los tipos de sus parámetros y el orden de los mismos definen la signatura de un método.

ß Los modificadores y el tipo del valor devuelto por un método **no** forman parte de la signatura del método.

Sobrecarga

Lenguajes como Java permiten que existan distintos métodos con el mismo nombre siempre y cuando su signatura no sea idéntica (algo que se conoce con el nombre de *sobrecarga*)

Ejemplo

```
System.out.prinln(…);
    - System.out.println()
    - System.out.println(boolean)
    - System.out.println(char)
    - System.out.println(char[])
    - System.out.println(double)
    - System.out.println(float)
    - System.out.println(int)
    - System.out.println(long)
    - System.out.println(Object)
    - System.out.println(String)
```

No es válido definir dos métodos con el mismo nombre
que difieran únicamente por el tipo del valor que devuelven.

De todas formas, no conviene abusar demasiado de esta prestación del lenguaje, porque resulta propensa a errores
(en ocasiones, creeremos estar llamando a una "versión"
de un método cuando la que se ejecuta en realidad es otra).

NOTA: En la creación de *constructores*
sí es importante disponer de esta característica.

Uso de métodos

Para enviarle un mensaje a un objeto,
invocamos (llamamos a) uno de sus métodos:

> La llamada a un método de un objeto
> le indica al objeto que delegamos en él
> para que realice una operación de la que es reponsable.

A partir de una referencia a un objeto,
podermos llamar a uno de sus métodos con el **operador** .

Tras el nombre del método, entre paréntesis,
se han de indicar sus parámetros (si es que los tiene).

El método podemos usarlo cuantas veces queramos.

MUY IMPORTANTE: De esta forma, evitamos la existencia de código duplicado en nuestros programas.

Ejemplo

```
Cuenta cuenta = new Cuenta();

cuenta.mostrarMovimientos();
```

Obviamente, el objeto debe existir antes de que podamos invocar uno de sus métodos. Si no fuese así, en la ejecución del programa obtendríamos el siguiente error:

```
java.lang.NullPointerException
    at ...
```

al no apuntar la referencia a ningún objeto (`null` en Java).

Ejemplo de ejecución paso a paso
Cuando se invoca un método, el ordenador pasa a ejecutar las sentencias definidas en el cuerpo del método:

```
public class Mensajes
{
  public static void main (String[] args)
  {
    mostrarMensaje("Bienvenida");
    // ...
    mostrarMensaje("Despedida");
  }

  private static void mostrarMensaje (String mensaje)
  {
    System.out.println("*** " + mensaje + " ***");
  }
}
```

Al ejecutar el programa (con `java Mensajes`):

1. Comienza la ejecución de la aplicación, con la primera sentencia especificada en el cuerpo del método `main`.
2. Desde `main`, se invoca `mostrarMensaje` con *"Bienvenida"* como parámetro.
3. El método `mostrarMensaje` muestra el mensaje de bienvenida decorado y termina su ejecución.
4. Se vuelve al punto donde estábamos en `main` y se continúa la ejecución de este método.
5. Justo antes de terminar, volvemos a llamar a `mostrarMensaje` para mostrar un mensaje de despedida.
6. `mostrarMensaje` se vuelve a ejecutar, esta vez con *"Despedida"* como parámetro, por lo que esta vez se muestra en pantalla un mensaje decorado de despedida.
7. Se termina la ejecución de `mostrarMensaje` y se vuelve al método desde donde se hizo la llamada (`main`).
8. Se termina la ejecución del método `main` y finaliza la ejecución de nuestra aplicación.

Los parámetros de un subprograma

Un método puede tener parámetros:

A través de los parámetros se especifican los datos de entrada que requiere el método para realizar su tarea.

Los parámetros definidos en la cabecera del método se denominan **parámetros formales**.

Para cada parámetro, hemos de especificar tanto su tipo como un identificador que nos permita acceder a su valor actual en la implementación del método.

Cuando un método tiene varios parámetros, los distintos parámetros se separan por comas en la cabecera del método.

En la definición de un método,
la lista de parámetros formales de un método establece:

- Cuántos parámetros tiene el método
- El tipo de los valores que se usarán como parámetros
- El orden en el que han de especificarse los parámetros

En la invocación de un método,
se han de especificar los valores concretos para los parámetros.

Los valores que se utilizan como parámetros
al invocar un método se denominan
parámetros actuales (o "argumentos").

Cuando se efectúa la llamada a un método, los valores indicados como parámetros actuales se asignan a sus parámetros formales.

En la implementación del método, podemos utilizar entonces los parámetros del método como si fuesen variables normales (y de esta forma acceder a los valores concretos con los que se realiza cada llamada al método).

Obviamente, el número y tipo de los parámetros indicados en la llamada al método ha de coincidir con el número y tipo de los parámetros especificados en la definición del método.

En Java, todos los parámetros se pasan por valor:

Al llamar a un método,
el método utiliza una copia local de los parámetros
(que contiene los valores con los cuales fue invocado).

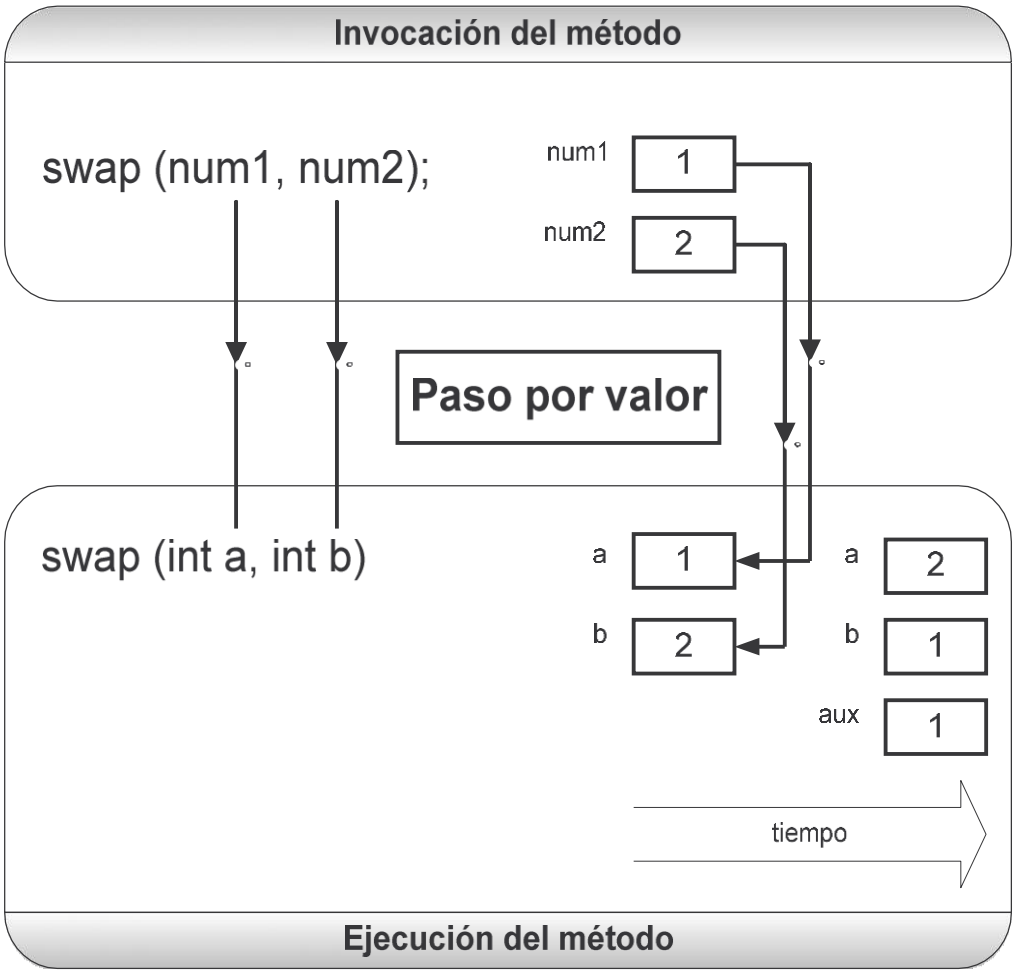
¡OJO! Como las variables de tipos no primitivos son, en realidad, referencias a objetos, lo que se copia en este caso es la referencia al objeto (no el objeto en sí).

Como consecuencia, podemos modificar el estado de un objeto recibido como parámetro si invocamos métodos de ese objeto que modifiquen su estado.

La referencia al objeto no cambia, pero sí su estado.

Ejemplo: Intercambio incorrecto de valores

```
public void swap (int a, int b)   // Definición
{
  int aux;
  aux = b;
  a   =  b;
  b = aux;
}
...
swap(a,b);                        // Invocación
```



- Los valores de `num1` y `num2` se copian en `a` y `b`.
- La ejecución del método `swap` no afecta ni a `num1` ni a `num2`.

`swap` no intercambia los valores de las variables porque el intercambio se hace sobre las **copias locales** de los parámetros de las que dispone el método, no sobre las variables originales.

Convenciones

- Si varios métodos utilizan los mismos parámetros, éstos han de ponerse siempre en el mismo orden.

  De esta forma, resulta más fácil de recordar la forma correcta de usar un método.

- No es aconsejable utilizar los parámetros de una rutina como si fuesen variables locales de la rutina.

  En otras palabras, los parámetros no los utilizaremos para almacenar resultados parciales.

- Se han de documentar claramente las suposiciones que se hagan acerca de los valores permitidos para los distintos parámetros de un método.

  Esta información debería figurar en la documentación del código realizada con la herramienta `javadoc`.

- Sólo se deben incluir los parámetros que realmente necesite el método para efectuar su labor.

  Si un dato no es necesario para realizar un cálculo, no tiene sentido que tengamos que pasárselo al método.

- Las dependencias existentes entre distintos métodos han de hacerse explícitas mediante el uso de parámetros.

  Si evitamos la existencia de variables globales (datos compartidos entre distintos módulos), el código resultante será más fácil de entender.

Devolución de resultados: La sentencia `return`

Cuando un método devuelve un resultado, la implementación del método debe terminar con una sentencia `return`:

```
return expresión;
```

Como es lógico, el tipo de la *expresión* debe coincidir con el tipo del valor devuelto por el método, tal como éste se haya definido en la cabecera del método.

Ejemplo

```
public static float media (float n1, float n2)
{
  return (n1+n2)/2;
}

public static void main (String[] args)
{
  float resultado = media (1,2);

  System.out.println("Media = " + resultado);
}
```

El compilador de Java comprueba que exista una sentencia `return` al final de un método que deba devolver un valor.
Si no es así, nos dará el error

```
Missing return statement
```

El compilador también detecta si hay algo después de la sentencia `return` (un error porque la sentencia return finaliza la ejecución de un método y nunca se ejecuta lo que haya después):

```
Unreachable statement
```

Ejemplo

Figuras geométricas

```
// Title:       Geometry
// Version:     0.0
// Copyright:   2004
// Author:      Fernando Berzal
// E-mail:      berzal@acm.org

public class Point
{
  // Variables de instancia

  private double x;
  private double y;

  // Constructor

  public Point (double x, double y)
  {
    this.x = x;
    this.y = y;
  }

  // Métodos

  public double distance (Point p)
  {
    double dx = this.x - p.x;
    double dy = this.y - p.y;

    return Math.sqrt(dx*dx+dy*dy);
  }

  public String toString ()
  {
    return "(" + x + "," + y + ")";
  }
}
```

NB: La interfaz de la clase habría que documentarlo añadiendo los correspondientes comentarios `javadoc`.

```
public class Circle
{
  private Point  centro;
  private double radio;

  // Constructor

  public Circle (Point centro, double radio)
  {
    this.centro = centro;
    this.radio  = radio;
  }

  // Métodos

  public double area ()
  {
    return Math.PI*radio*radio;
  }

  public boolean isInside (Point p)
  {
     return ( centro.distance(p) < radio );
  }

  public String toString ()
  {
     return "Círculo con radio " + radio
          + " y centro en " + centro;
  }
}
```

Ejemplo de uso

```
public static void main(String[] args)
{
  Circle fig    = new Circle( new Point(0,0), 10);
  Point  dentro = new Point (3,3);
  Point  fuera  = new Point (10,10);

  System.out.println(figura);
  System.out.println(dentro+"? "+fig.isInside(dentro));
  System.out.println(fuera +"? "+fig.isInside(fuera) );
}
```

Constructores. La palabra reservada `this`

Los constructores son métodos especiales que sirven para inicializar el estado de un objeto cuando lo creamos con el operador `new`

- Su nombre ha de coincidir coincide con el nombre de la clase.
- Por definición, no devuelven nada.

```
public class Point
{
  // Variables de instancia
  private double x;
  private double y;

  // Constructor
  public Point (double x, double y)
  {
    this.x = x;
    this.y = y;
  }

  // Acceso a las coordenadas

  public double getX ()
  {
    return x;
  }

  public double getY ()
  {
    return y;
  }
}
```

NOTA:
La palabra reservada `this` permite acceder al objeto sobre el que se ejecuta el método.

La clave de implementar la clase `Point` de esta forma
(y no dar acceso directo a las variables de instancia)
es que podemos cambiar la implementación de `Point`
para usar coordenadas polares
y la implementación de las clases que trabajan con puntos
(p.ej. `Circle`) no hay que tocarla:

```
public class Point
{
  // Variables de instancia

  private double r;
  private double theta;

  // Constructor

  public Point (double x, double y)
  {
    r     = Math.sqrt(x*x+y*y);
    theta = Math.atan2(y,x);
  }

  // Acceso a las coordenadas

  public double getX ()
  {
    return r*Math.cos(theta);
  }

  public double getY ()
  {
    return return r*Math.sin(theta);
  }
}
```

Gracias a la encapsulación,
podemos crear componentes reutilizables
cuya evolución no afectará al resto del sistema.

Podemos definir varios constructores para poder inicializar un objeto de distintas formas (siempre y cuando los constructores tengan signaturas diferentes);

```
public class Contacto
{
  private String nombre;
  private String email;

  public Contacto (String nombre)
  {
    this.nombre = nombre;
  }

  public Contacto (String nombre, String email)
  {
    this.nombre = nombre;
    this.email = email;
  }

}
```

Ejemplo de uso:

```
public class ContactoTest
{
  public static void main(String[] args)
  {
    Contacto nico = new Contacto ("Nicolás");

    Contacto juan = new Contacto ("Juan",
                                  "juan@acm.org");
    ...
  }
}
```

Constructor de copia
Un constructor que recibe como parámetro
un objeto de la misma clase que la del constructor

Otro ejemplo de uso de la palabra reservada `this`
consiste en llamar a un constructor desde otro de los constructores
(algo que, de hacerse, siempre ha de ponerse
al comienzo de la implementación del constructor)

```
public class Contacto
{
  private String nombre;
  private String email;

  public Contacto (String nombre)
  {
    this(nombre, "");
  }

  public Contacto (String nombre, String email)
  {
    this.nombre = nombre;
    this.email = email;
  }

  // Constructor de copia

  public Contacto (Contacto otro)
  {
    this (otro.nombre, otro.email);
  }
}
```

RECORDATORIO: **Encapsulación**
Se consigue con los modificadores de acceso `public` y `private`.

- Las variables de instancia de una clase suelen definirse como privadas (con la palabra reservada `private`).
- Los métodos públicos muestran la interfaz de la clase.
- Pueden existir métodos no públicos que realicen tareas auxiliares manteniendo la separación entre interfaz e implementación.

Métodos estáticos

Los métodos estáticos pertenecen a la clase
(no están asociados a un objeto particular de la clase)

Ya hemos visto algunos ejemplos:

```
Math.pow(x,y)

public static void main (String[] args) …
```

Para invocar un método estático,
usamos directamente el nombre de la clase (p.ej. `Math`)

No tenemos que instanciar antes un objeto de la clase.

`main` es un método estático
de forma que la máquina virtual Java
puede invocarlo sin tener que crear antes un objeto.

Como los métodos estáticos no están asociados a objetos concretos,
no pueden acceder a las variables de instancia de un objeto
(las cuales pertenecen a objetos particulares).

Variables estáticas = Variables de clase

Las clases también pueden tener variables que se suelen emplear para representar constantes y variables globales a las cuales se pueda acceder desde cualquier parte de la aplicación (aunque esto último no es muy recomendable).

Ejemplos `System.out`

Colores predefinidos:
`Color.black`, `Color.red`...

Como es lógico,
los métodos estáticos sólo pueden acceder a variables estáticas.

```
public class Mensajes
{
  private String mensaje = "Hola";   // ¡Error!

  public static void main (String[] args)
  {
    mostrarMensaje(mensaje);
  }

  private static void mostrarMensaje (String mensaje)
  {
    System.out.println("*** " + mensaje + " ***");
  }
}
```

El programa anterior funcionaría correctamente
si hubiésemos declarado `mensaje` como una variable estática:

```
  private static String mensaje = "Hola";
```

> NOTA: No es aconsejable declarar variables estáticas
> salvo para definir constantes, como sucede en la clase `Math`.

Métodos estáticos y variables estáticas en la clase `Math`

- Constantes matemáticas: `Math.PI`, `Math.E`
- Métodos estáticos: Funciones trigonométricas (`sin, cos, tan, acos, asin, atan`), logaritmos y exponenciales (`exp, log, pow, sqrt`), funciones de redondeo (`ceil, floor, round` [`== Math.floor(x+0.5)`], `rint` [al entero más cercano, se coge el par]), máximo y mínimo (`max, min`), valor absoluto (`abs`), generación de número pseudoaleatorios (`random`)...