Contenido

1. CAMBIO DE CONTEXTO AL INVOCAR A UN MÉTODO

En el momento de invocar a un método, el motor de ejecución realiza una operación de **cambio de contexto**: almacena el estado en el que se encuentra el programa (el ***contexto del código llamador***) justo antes de llamar al método. Este contexto está formado por las variables existentes en el código llamador, junto con el estado de ejecución del mismo.

Una vez que el contexto del código llamador ha quedado almacenado, se inicia la ejecución del método invocado. Al comenzar la ejecución de este método, el motor de ejecución emplea un contexto nuevo, local al método, formado por los parámetros recibidos por el método y todas las variables y constantes que se referencian desde ese método.

Cuando finaliza la ejecución del método invocado, el motor de ejecución recupera el estado previo, es decir, el contexto del código llamador en el que se encontraba el programa (que había sido guardado previamente) y puede así continuar la ejecución por la instrucción siguiente a la llamada al método.

No obstante, el contexto del código llamador no es estanco. Por el contrario, dependiendo del tipo de paso de parámetro empleado al invocar al método, es posible que el método pueda modificar las variables del contexto del código llamador, de forma que, al finalizar el método, las variables del código llamador se vean modificadas respecto al valor que tenían antes de llamar al método.

Esta capacidad de modificar el contexto del código llamador desde el cuerpo del método invocado resulta útil, puesto que permite al método devolver resultados al código llamador a través de esas variables del contexto.

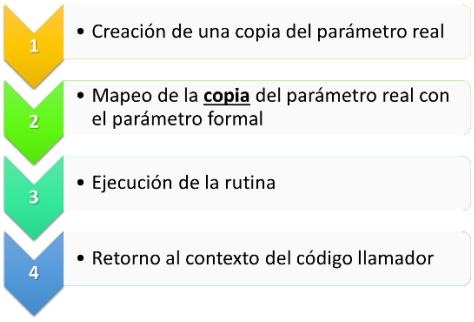
2. PASO DE PARÁMETROS

Java es un lenguaje relativamente pobre en cuanto a la capacidad para definir el comportamiento que tendrán los parámetros al invocar a un método. Dispone de un único modo para pasar parámetros a un método, el paso de parámetro por valor, por lo que no ofrece la flexibilidad existente en otros lenguajes, como C#, para adaptarse a diferentes requerimientos y situaciones en lo que se refiere a la gestión de los parámetros.

Con carácter general, al invocar la ejecución de un procedimiento o función con parámetros, cada uno de los parámetros puede ser manejado de acuerdo a 2 situaciones: paso de parámetro **por valor** o paso de parámetro **por referencia**. Como hemos indicado, Java únicamente soporta el paso por valor, aunque en los apartados siguientes abordaremos ambos modos, ya que constituye un concepto esencial en programación, que puede extrapolarse a muchos otros lenguajes.

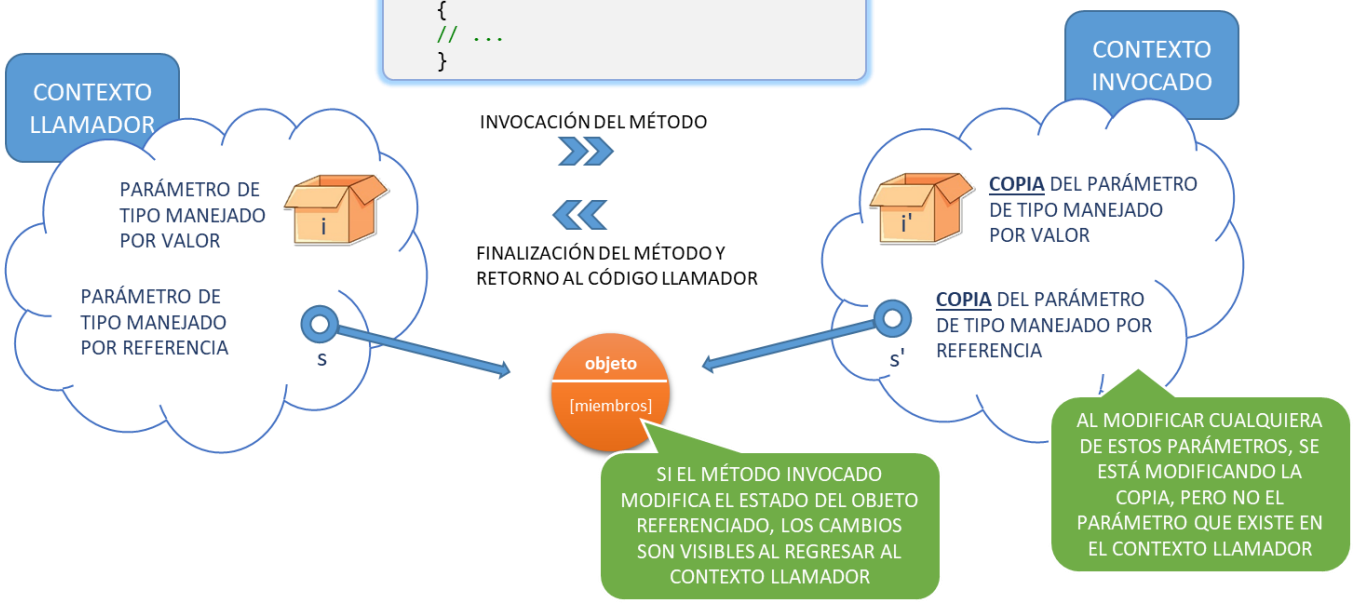
3. PASO DE PARÁMETROS POR VALOR

Este **es el único modo de paso de parámetros** soportado por Java[1]. En este modo de transferencia de parámetros, durante la invocación, **se crea una copia del parámetro** real[2] y se mapea dicha copia con el parámetro formal. De manera esquemática, el proceso sigue estas etapas:




```
public static void main(String[] args)
   {
   int i = 10;
   String s = "hola";
   test(i, s);
   }

static void test (int i, String s)
   {
   // ...
   }
```



Tendremos dos posibilidades, dependiendo de que el parámetro sea un tipo manejado por valor o por referencia, como veremos en los siguientes apartados.

---

[1] Es habitual encontrar fuentes de información que sostienen que Java soporta paso por valor y paso por referencia, pero se trata de un error conceptual. Java soporta únicamente el paso por valor. Sin embargo, el paso por valor de un tipo referenciado es similar al paso por referencia (aunque no es idéntico, como veremos a continuación). De ahí que algunas fuentes introducen erróneamente esta concepción.

[2] Es conveniente tener presente la diferencia entre parámetro real y parámetro formal, que hemos abordado en el documento sobre métodos.

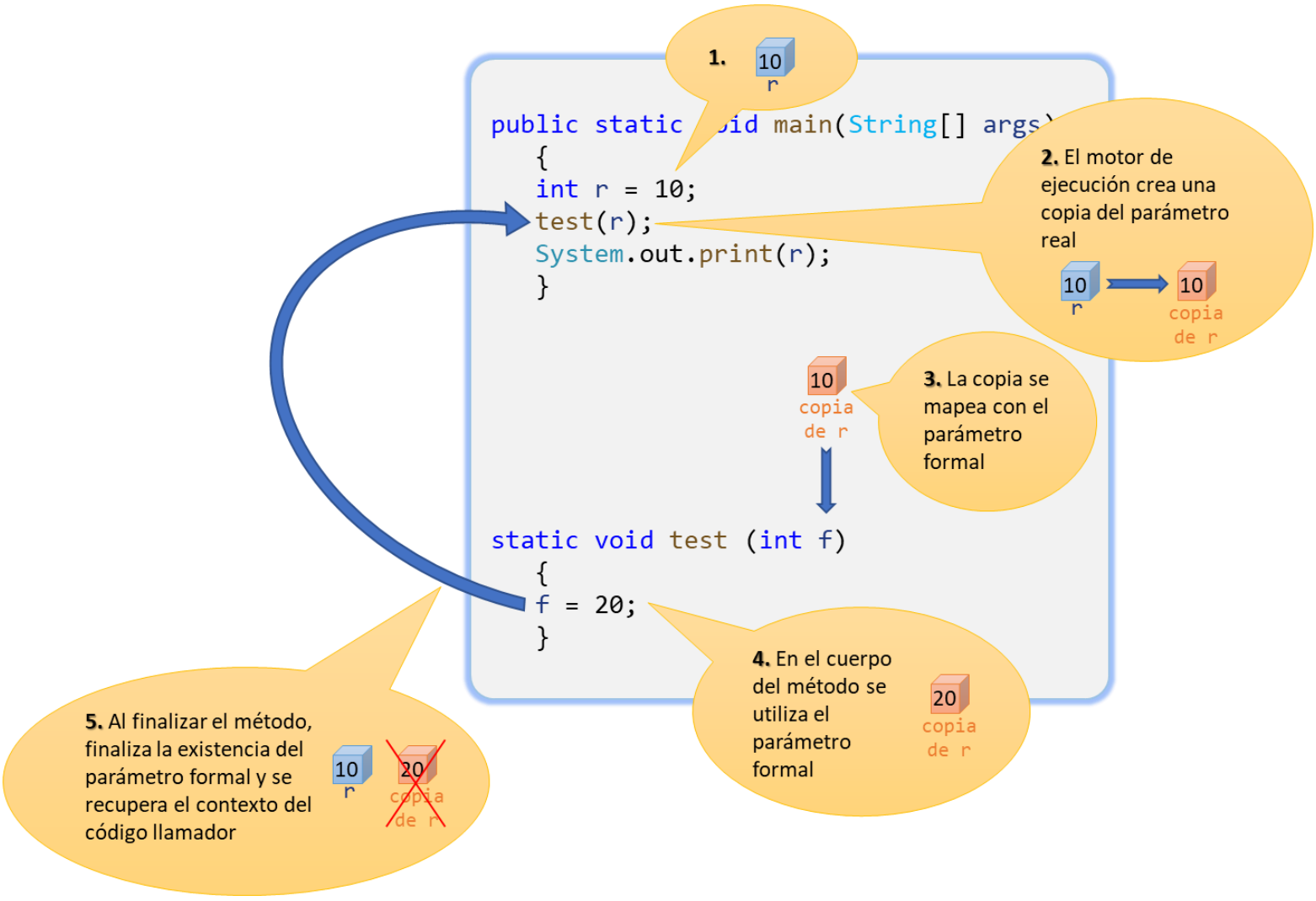
3.1. PASO POR VALOR DE UN PARÁMETRO MANEJADO POR VALOR

Si el parámetro pasado en la invocación del método es una variable o un literal de un tipo **manejado por valor**, el motor de ejecución crea, de forma transparente, **una copia de esa variable o literal** y el método **recibe esa copia,** en vez de recibir el parámetro real (que es la variable o literal que se encuentra en el código llamador).

La copia del parámetro real se mapea (almacena) en el **parámetro formal**, que es el dato que maneja localmente el cuerpo del método.

Cuando finaliza la ejecución del método, las variables locales al método (incluidos los parámetros formales) se destruyen y, al retornar al contexto del código llamador, se restituyen las variables que existían en el código llamador en el momento previo a la invocación del método.

De forma esquemática, el proceso que se sigue en un paso por valor con un parámetro por valor es el siguiente:



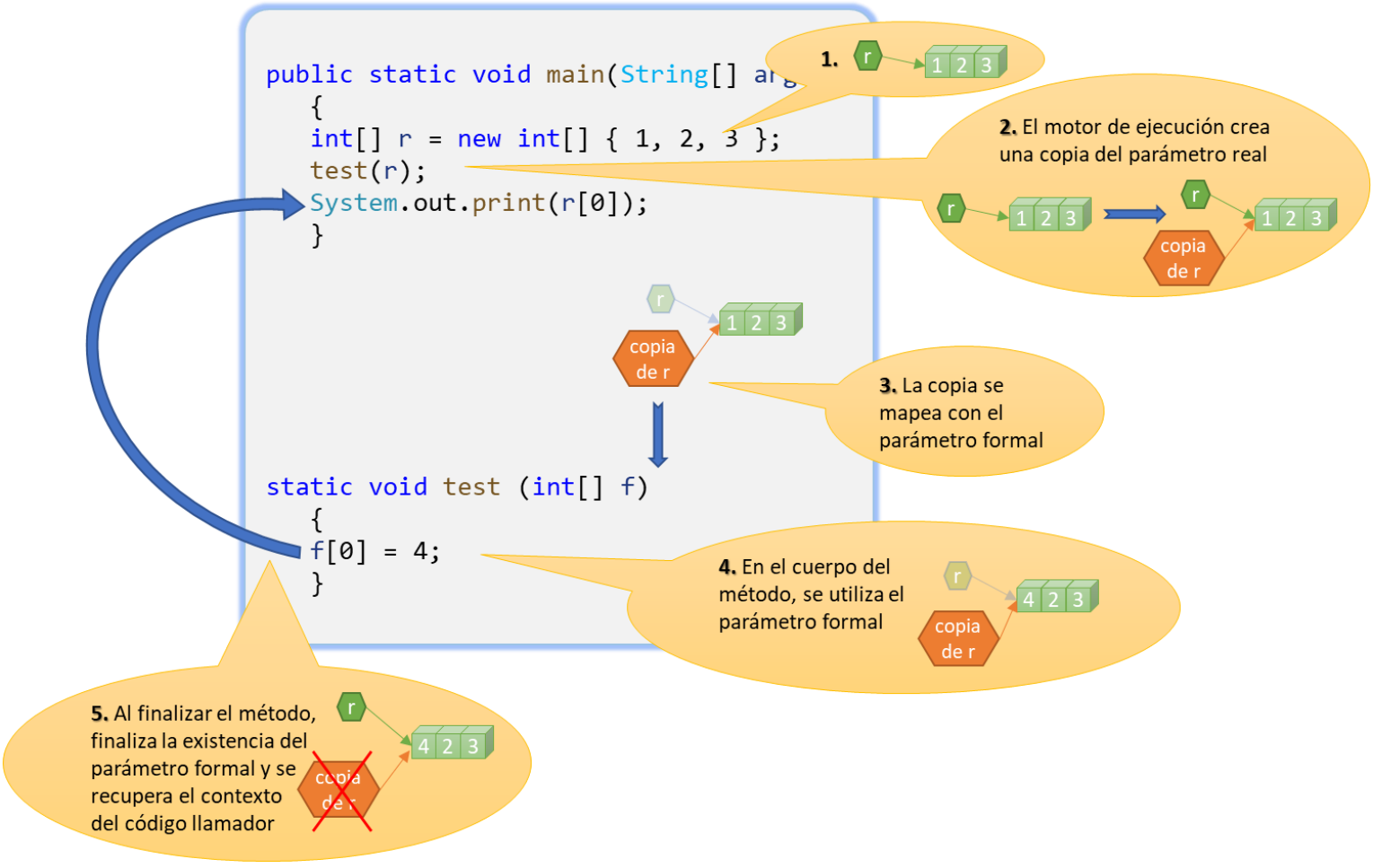
3.2. PASO POR VALOR DE UN PARÁMETRO MANEJADO POR REFERENCIA

Si el parámetro pasado en la invocación del método es una variable o un literal de un tipo **manejado por referencia**, el método recibe **una copia de la referencia** (no la misma referencia que se pasa al invocar al método ni tampoco el objeto en sí).

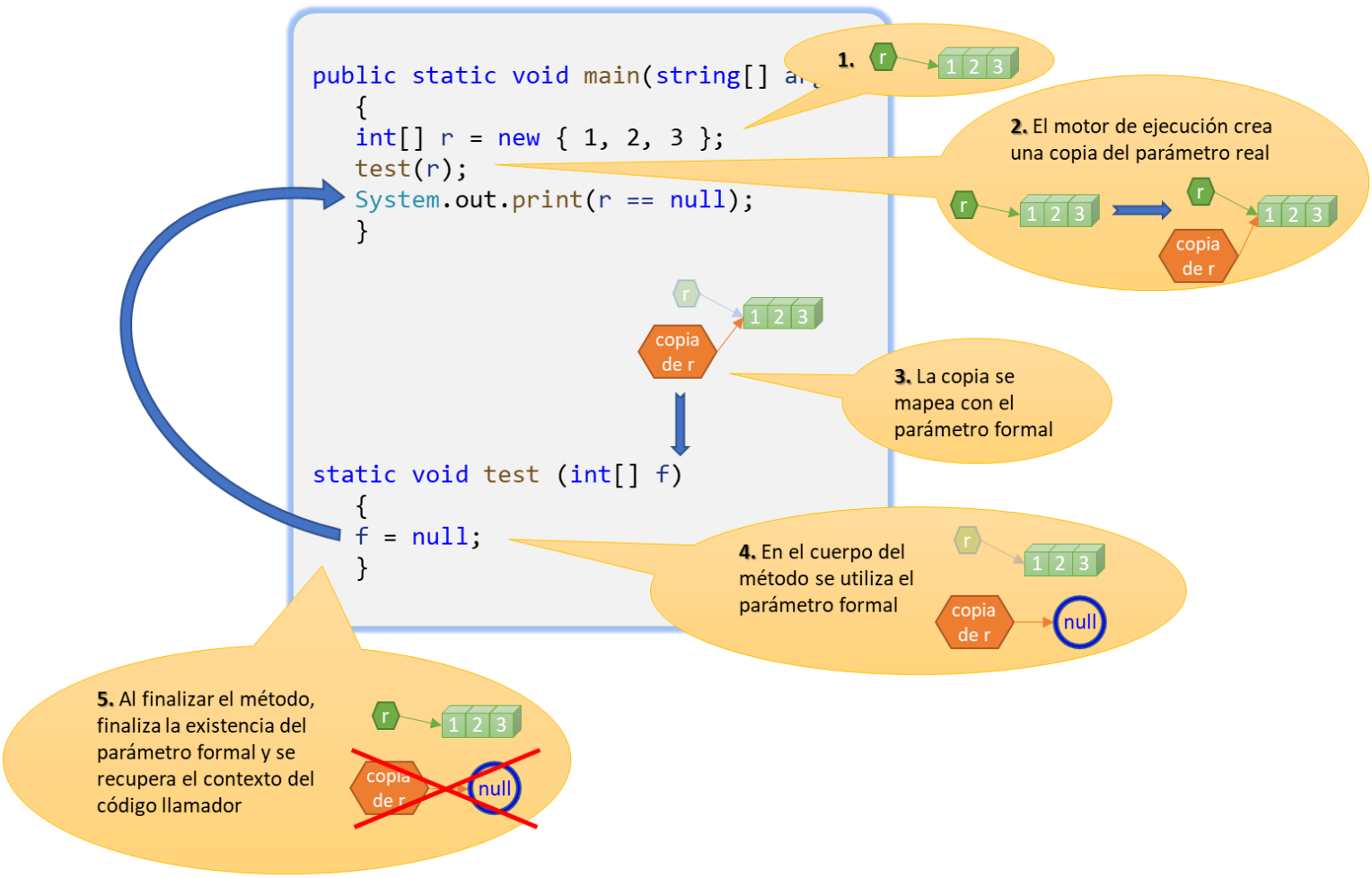
Esta copia de la referencia es manejada localmente en el cuerpo del método por lo que, si se modifican las propiedades del objeto al que apunta la referencia, cuando finaliza la ejecución del método, los cambios efectuados en el objeto son visibles para el código llamador.

No obstante, si se efectúan cambios en la referencia para apuntar a otro objeto o a `null`, dichos cambios no son visibles para el código llamador (que mantiene su referencia original). Ambas situaciones se recogen en los siguientes esquemas:

Escenario 1: paso por valor de un tipo referenciado. En el cuerpo de la rutina se modifica **el dato referenciado**:



Escenario 2: paso por valor de un tipo referenciado. En el cuerpo de la rutina se modifica **la referencia**:



3.2.1. PASO POR VALOR CON EL MODIFICADOR `final`

`final` se utiliza para designar un parámetro como **parámetro de entrada**, destinado únicamente a la introducción de datos al método, sin que éste pueda modificar la referencia representada por el parámetro.

Esto quiere decir el método puede acceder y modificar los miembros del objeto referenciado por el parámetro, pero **no puede modificar** dicha referencia. Por tanto, no se puede hacer que apunte a otro objeto ni a `null`**.**