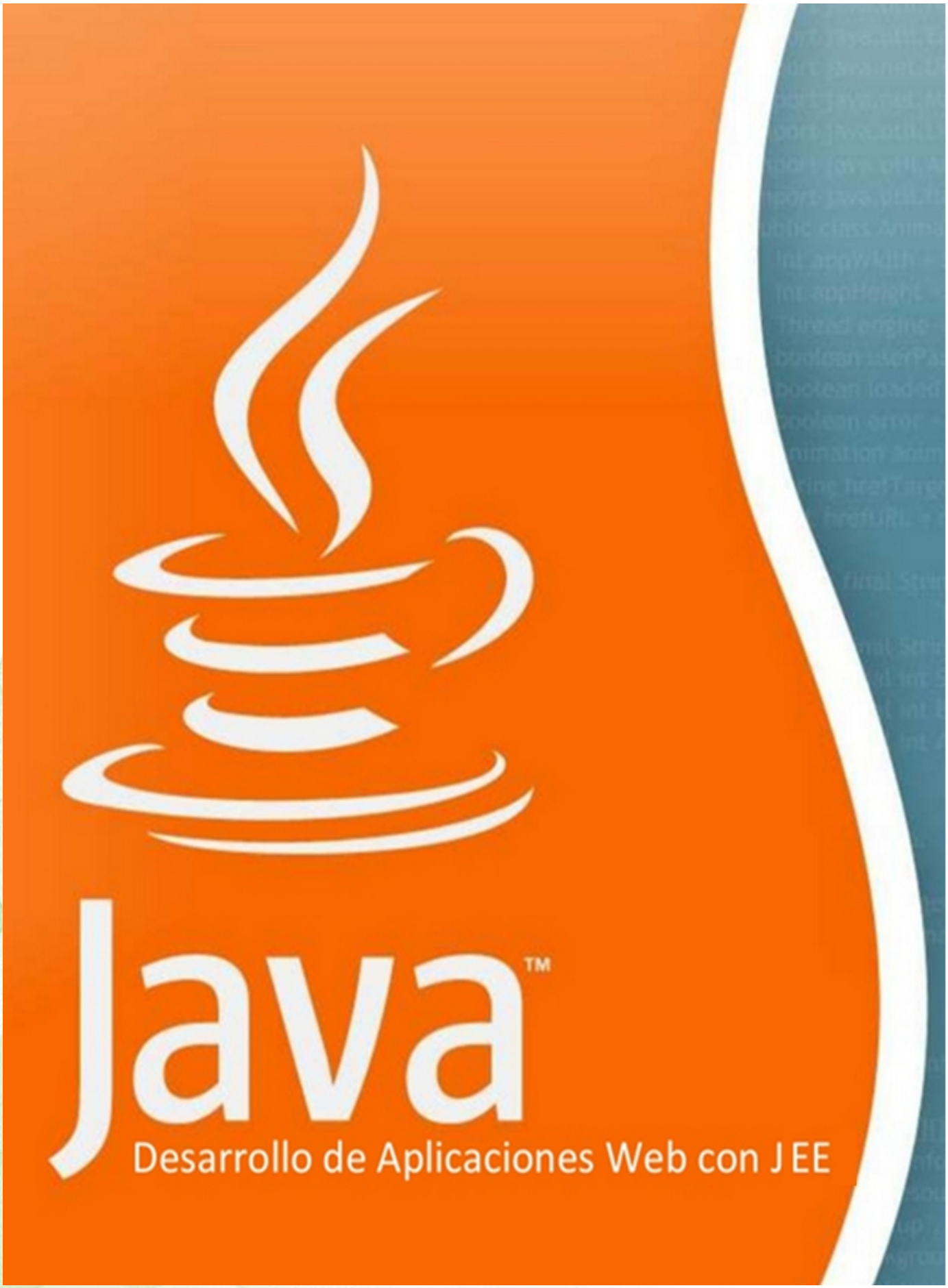
Java™
Desarrollo de Aplicaciones Web con J EE

PARTE VI

CONCURRENCIA DE PROCESOS

Tabla de contenido

No se encontraron entradas de tabla de contenido.

6. Concurrencia de procesos

Se dice que una aplicación soporta **concurrencia cuando permite ejecuciones en paralelo de su código**. Es decir, en el caso de las aplicaciones **Java EE** que estamos estudiando, significaría que **varios usuarios puedan conectarse y ejecutar la aplicación a la vez.**

Las aplicaciones Java EE son concurrentes por naturaleza. El **Servidor de Aplicaciones Java EE se encarga de controlar y permitir la concurrencia de la aplicación** permitiendo al desarrollador prácticamente despreocuparse de esta casuística.

Teniendo en cuenta que **solamente hay una instancia de un Java Servlet por cada JVM**, no debemos olvidarnos nunca del tema y tenerlo siempre presente en nuestros desarrollos para evitar problemas y comportamientos no deseados.

Entonces a la hora de llamar a un servlet, **se crea un objeto HttpServlet**, pero sólo uno. **A ese objeto pueden acceder simultáneamente muchas llamadas**. Por cada llamada se crea un proceso que recorre el código del método de respuesta correspondiente (doGet, doPost…etc) de este objeto único HttpServlet. Pues bien, si esos diferentes procesos quieren acceder a un objeto común a todos, puede producir problemas de concurrencia.

6.1 Ejemplo11

Por ejemplo, imaginemos lo que puede ocurrir si utilizamos variables de instancia en el Java Servlet para manejar una información temporal durante la ejecución de un método, en el que hemos sustituido un posible código que tarda cierto tiempo en ejecutar por un Thread.sleep(5000);

```
public class Ejemplo11 extends HttpServlet {
    private static final long serialVersionUID = 1L;
    private static int val = 0;

    /**
     * @see HttpServlet#HttpServlet()
     */
    public Ejemplo11() {
        super();
        // TODO Auto-generated constructor stub
    }

  protected   void doGet(HttpServletRequest request, HttpServletResponse
  response) throws ServletException, IOException
  {
  val = Integer.parseInt(request.getParameter("param"));
  //Mostramos el proceso que se está ejecutando
  System.out.println("Thread: " + Thread.currentThread().getName() +
  " usa el valor: " + val);
  try
  {
  //Simulamos que el servlet está con alguna operativa antes de
  //volver a utilizar el valor de la variable val.
  Thread.sleep(10000);
  }
  catch(InterruptedException ex)
  {
  System.out.println(ex.toString());
  }
  //Mostramos el proceso que se está ejecutando
  System.out.println("Thread: " + Thread.currentThread().getName() +
  " usa el valor: " + val);
  }
```

Invocamos desde el navegador a este Java Servlet dos veces consecutivas (en menos de 5 segundos) primero pasando el parámetro 5 y luego con el parámetro 6. Observemos en la salida por consola lo que ocurre:

```
Thread: http-nio-8080-exec-3 usa el valor: 5
Thread: http-nio-8080-exec-4 usa el valor: 6
Thread: http-nio-8080-exec-3 usa el valor: 6
Thread: http-nio-8080-exec-4 usa el valor: 6
```

La primera petición la sirve un thread controlado por el Servidor de Aplicaciones Java EE que se llama "http-8080-exec-3" y el inicio de la ejecución del Java Servlet muestra que el parámetro con el que tiene

que trabajar vale 5. Entonces, se duerme durante 10 segundos (ha sido la forma de simular una tarea costosa) y cuando vuelve a por el parámetro, resulta que vale 6.

¿Qué ha pasado? Lo que ha pasado es que antes de que el primer Java Servlet terminara de ejecutarse, el Servidor de Aplicaciones Java EE recibió una segunda petición al mismo Java Servlet con un valor del parámetro distinto. **Como ambos threads, en este caso "http-bio- 8080-4", ejecutan el mismo código, ha machacado el valor al primero**. Se trata de un ejemplo muy sencillo, pero el problema que demuestra puede ser muy complicado de detectar.

Existen distintas soluciones:

- ✓ **Hacer que en el código del servlet no haya código que implique un acceso a objetos comunes, como atributos del propio servlet**, **y trabajar con variables locales definidas en los métodos de servicio** doPost(..), ya que dentro de ese método serán locales a cada proceso. De este modo, no hay problema de acceso concurrente, porque cada hilo obtiene su copia de dichas variables locales sin interferencia con las del resto de hilos asociados al resto de peticiones.
- ✓ **Sincronizar los métodos de servicio.** Esto es una mala solución que ralentizaría el acceso de los clientes a la aplicación. Hasta que no se ejecutara todo el código del método de servicio no se permitiría el acceso al mismo de otra petición.
- ✓ **Implementar el interface javax.servlet.SingleThreadModel (Decrecated)** que **le indica al Servidor de Aplicaciones Java EE, que no puede paralelizar las ejecuciones de este Java Servlet.** Esta solución no es recomendable porque por regla general implica un tiempo de respuesta pésimo a los usuarios. Imaginad cien usuarios pidiendo este servicio a la vez y teniendo que esperar en cola a ser respondidos.
- ✓ **Utilizar técnicas de sincronización del lenguaje Java (synchronized).** Debemos tener presente que los **atributos del servlet, el contexto y las sesiones no son Thread-Safe** (puede ser invocada por múltiples hilos de ejecución sin preocuparnos de que los datos a los que accede dicha función (o método) sean corrompidos por alguno de los hilos), **sí el objeto request y las variables locales de los métodos de servicio (en el doPost/doGet).** Esto significará que, si queremos trabajar con el contexto, por ejemplo, deberemos sincronizar su modificación.

```
synchronized(getServletContext())
{
getServletContext.setAttribute("saludo","hola");
Thread.sleep(10000);
out.println("<b>"+getServletContext.getAttribute("saludo")+"</b>");
}
```

Cuando es empleada la palabra reservada **synchronized** se está indicando una zona restringida para el uso de "Threads", esta zona restringida para efectos prácticos puede ser considerada un candado("lock") sobre la instancia del Objeto en cuestión. Esto implica que si es invocado un método synchronized únicamente el "Thread" que lo invoca tiene acceso a la instancia del Objeto, y cualquier otro "Thread" que intente acceder a esta misma instancia tendrá que esperar hasta que sea terminada la ejecución del método synchronized.

¿Cuándo se libera el lock? Se libera cuando el thread que lo tiene tomado sale del bloque por cualquier razón: termina la ejecución del bloque normalmente, ejecuta un return o lanza una excepción.

En el ejemplo anterior si modificamos el método doGet añadiéndole el modificador synchronized le hemos asegurado de modo que un sólo subproceso a la vez puede acceder a él. Como hemos dicho esta no es buena solución ya que ralentiza el proceso.

```
protected synchronized  void doGet(HttpServletRequest request, HttpServletResponse
response) throws ServletException, IOException
```

Observamos ahora la salida por consola:

```
Thread: http-nio-8080-exec-9 usa el valor: 5
Thread: http-nio-8080-exec-9 usa el valor: 5
Thread: http-nio-8080-exec-10 usa el valor: 6
Thread: http-nio-8080-exec-10 usa el valor: 6
```

Si ahora implementamos un método llamado prueba y lo sincronizamos y lo llamamos dentro del método doGet evitamos que otro proceso no pueda acceder al método de servicio.

```
public synchronized void prueba(HttpServletRequest request) {
    val = Integer.valueOf(request.getParameter("param"));
    System.out.println("Thread: " + Thread.currentThread().getName() + " usa el valor: " + val);

    try {
        Thread.sleep(10000);
    } catch (InterruptedException ex) {

        System.out.println(ex.toString());
    }
    System.out.println("Thread: " + Thread.currentThread().getName() + " usa el valor: " + val);
}
protected void doGet(HttpServletRequest request, HttpServletResponse response)
        throws ServletException, IOException {

    prueba(request);
}
```

Una opción todavía más eficaz sería sincronizar solamente las líneas de código afectadas por ese problema de concurrencia utilizando como candado un objeto Object:

```
private final Object lock = new Object();
public void doGet(HttpServletRequest request, HttpServletResponse response) throws ServletException, IOException {
    try {
        synchronized (lock) {
            val = Integer.valueOf(request.getParameter("param"));
            System.out.println("Thread: " + Thread.currentThread().getName() + " usa el valor: " + val);
            Thread.sleep(10000);
        }
    } catch (InterruptedException ex) {
        ex.printStackTrace();
    }
    System.out.println("Thread: " + Thread.currentThread().getName() + " usa el valor: " + val);
}
```