Programación 2

Lenguaje Java - *Threads (Hilos)*

Rodrigo Olivares
Mg. en Ingeniería Informática
rodrigo.olivares@uv.cl

2^{do} Semestre de 2016

Contenido

Contenido

Contenido

Threads

Introducción

Introducción

Los procesadores y los Sistemas Operativos modernos permiten la **multitarea**, es decir, la realización simultánea de dos o más actividades (*al menos aparentemente*)

Threads

Introducción

En ordenadores con dos o más procesadores, la **multitarea es real**, ya que cada procesador puede ejecutar un **hilo** o **thread** diferente.

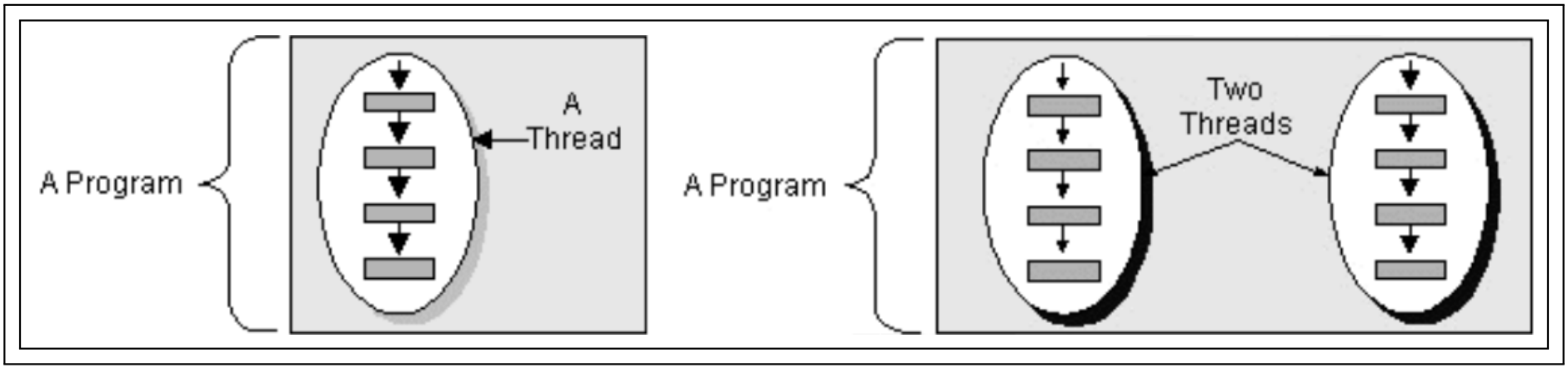

Figura 1: Programa con 1 y con 2 threads o hilos.

Threads

Introducción

Proceso y thread

Un proceso es un **programa** ejecutándose de forma independiente y con un espacio propio de memoria. Un thread o hilo es un **flujo secuencial** simple dentro de un proceso. Un único proceso puede tener varios hilos ejecutándose.

Ejemplo

Un claro ejemplo de un proceso serían algunos navegadores web, como Google Chrome y Mozilla Firefox. Cada pestaña ejecutándose sería un hilo o threads del proceso.

Threads

Introducción

Proceso y thread

Un proceso es un **programa** ejecutándose de forma independiente y con un espacio propio de memoria. Un thread o hilo es un **flujo secuencial** simple dentro de un proceso. Un único proceso puede tener varios hilos ejecutándose.

Ejemplo

Un claro ejemplo de un proceso serían algunos navegadores web, como Google Chrome y Mozilla Firefox. Cada pestaña ejecutándose sería un hilo o threads del proceso.

Threads

Introducción

Proceso y thread

Un sistema multitarea da realmente la impresión de estar haciendo varias cosas a la vez y eso es una gran ventaja para el usuario.

Sin el uso de threads hay tareas que son prácticamente imposibles de ejecutar, particularmente las que tienen tiempos de espera importantes entre etapas.

Threads

Introducción

Proceso y thread

Un sistema multitarea da realmente la impresión de estar haciendo varias cosas a la vez y eso es una gran ventaja para el usuario.

Sin el uso de threads hay tareas que son prácticamente imposibles de ejecutar, particularmente las que tienen tiempos de espera importantes entre etapas.

Threads

Introducción

Proceso y thread

Los threads o hilos de ejecución permiten organizar los recursos del ordenador de forma que pueda haber varios programas actuando en paralelo. Un hilo de ejecución puede realizar cualquier tarea que pueda realizar un programa común y corriente. Bastará con indicar lo que tiene que hacer en el método **run**(), que es el que define la actividad principal de las threads.

Threads

Introducción

Proceso y thread

Los threads pueden ser **daemon** o **no daemon**. Son **daemon** aquellos hilos que realizan en **background** (en un segundo plano) servicios generales. Esto es, tareas que no forman parte de la esencia del programa y que se están ejecutando mientras no finalice la aplicación. Un thread **daemon** podría ser por ejemplo aquél que está comprobando permanentemente si el usuario pulsa un botón. Un programa de Java finaliza cuando sólo quedan corriendo threads de tipo daemon. Por defecto, y si no se indica lo contrario, los threads son del tipo **no daemon**.

Contenido

Threads

Creación de threads

Creación

En Java hay **dos** formas de crear nuevos threads:

- → Crear una nueva clase hija que **herede** de la clase padre **java.lang.Thread** y sobrecargar el método **run**() de dicha clase.
- → Declarar una clase que implemente la *interface* **java.lang.Runnable**, la cual declarará el método **run**(); posteriormente se crea un objeto de tipo **Thread** pasándole como argumento en el constructor, el objeto creado de la nueva clase (la que implementa la interface **Runnable**)

Como ya se ha apuntado, tanto la clase **Thread** como la interface **Runnable** pertenecen al package **java.lang**, por lo que no es necesario importarlas.

Threads

Creación de threads

Creación

En Java hay **dos** formas de crear nuevos threads:

- → Crear una nueva clase hija que **herede** de la clase padre **java.lang.Thread** y sobrecargar el método **run**() de dicha clase.
- → Declarar una clase que implemente la *interface* **java.lang.Runnable**, la cual declarará el método **run**(); posteriormente se crea un objeto de tipo **Thread** pasándole como argumento en el constructor, el objeto creado de la nueva clase (la que implementa la interface **Runnable**)

Como ya se ha apuntado, tanto la clase **Thread** como la interface **Runnable** pertenecen al package **java.lang**, por lo que no es necesario importarlas.

Threads

Creación de threads

Creación

En Java hay **dos** formas de crear nuevos threads:

- → Crear una nueva clase hija que **herede** de la clase padre **java.lang.Thread** y sobrecargar el método **run**() de dicha clase.
- → Declarar una clase que implemente la *interface* **java.lang.Runnable**, la cual declarará el método **run**(); posteriormente se crea un objeto de tipo **Thread** pasándole como argumento en el constructor, el objeto creado de la nueva clase (la que implementa la interface **Runnable**)

Como ya se ha apuntado, tanto la clase **Thread** como la interface **Runnable** pertenecen al package **java.lang**, por lo que no es necesario importarlas.

Threads

Creación de threads

Creación

En Java hay **dos** formas de crear nuevos threads:

- → Crear una nueva clase hija que **herede** de la clase padre **java.lang.Thread** y sobrecargar el método **run**() de dicha clase.
- → Declarar una clase que implemente la *interface* **java.lang.Runnable**, la cual declarará el método **run**(); posteriormente se crea un objeto de tipo **Thread** pasándole como argumento en el constructor, el objeto creado de la nueva clase (la que implementa la interface **Runnable**)

Como ya se ha apuntado, tanto la clase **Thread** como la interface **Runnable** pertenecen al package **java.lang**, por lo que no es necesario importarlas.

Threads

Creación de threads

Ejemplo, Creación de threads derivando de la clase Thread

```
public class SimpleThread extends Thread {
   // constructor
   public SimpleThread (String str) {
      super(str);
   }

   // redefinición del método run()
   public void run() {
      for(int i = 0; i < 10; i++) {
         System.out.println("Este es el thread : " + getName());
      }
   }
}
```

Threads

Creación de threads

Ejemplo, Creación de threads derivando de la clase Thread

En este caso, se ha creado la clase **SimpleThread**, que hereda de **Thread**. En su constructor se utiliza un **String** (opcional) para poner nombre al nuevo **thread** creado, y mediante **super**() se llama al constructor de la super-clase **Thread**. Asimismo, se redefine el método **run**(), que define la principal actividad del thread, para que escriba 10 veces el nombre del thread creado.

Threads

Creación de threads

Ejemplo, Creación de threads derivando de la clase Thread

Para poner en marcha este nuevo **thread** se debe crear un objeto de la clase **SimpleThread**, y llamar al método start(), heredado de la super-clase **Thread**, que se encarga de llamar a **run**(). Por ejemplo:

```
SimpleThread miThread = new SimpleThread("Hilo de prueba");
miThread.start();
```

Threads

Creación de threads

Ejemplo, Creación de threads derivando de la clase Thread

Para poner en marcha este nuevo **thread** se debe crear un objeto de la clase **SimpleThread**, y llamar al método start(), heredado de la super-clase **Thread**, que se encarga de llamar a **run**(). Por ejemplo:

```
SimpleThread miThread = new SimpleThread("Hilo de prueba");
miThread.start();
```

Threads

Creación de threads

Ejemplo, Creación de threads derivando de la clase Thread

Para poner en marcha este nuevo **thread** se debe crear un objeto de la clase **SimpleThread**, y llamar al método start(), heredado de la super-clase **Thread**, que se encarga de llamar a **run**(). Por ejemplo:

```
SimpleThread miThread = new SimpleThread("Hilo de prueba");
miThread.start();
```

Threads

Creación de threads

Ejemplo, Creación de threads implementando la interface Runnable

Esta segunda forma también requiere que se defina el método run(), pero además es necesario crear un objeto de la clase **Thread** para lanzar la ejecución del nuevo hilo. Al constructor de la clase **Thread** hay que pasarle una referencia del objeto de la clase que implementa la interface Runnable. Posteriormente, cuando se ejecute el método **start**() del thread, éste llamará al método run() definido en la nueva clase.

Threads

Creación de threads

Ejemplo, Creación de threads implementando la interface Runnable

```
public class SimpleRunnable implements Runnable {
   // se crea un nombre
   String nameThread;
   public SimpleRunnable (String str) {
      nameThread = str;
   }

   // definición del método run()
   public void run() {
      for(int i = 0; i < 10; i++) {
         System.out.println("Este es el thread : " + nameThread);
      }
   }
}
```

Threads

Creación de threads

Ejemplo, Creación de threads implementando la interface Runnable

El siguiente código crea un nuevo thread y lo ejecuta el segundo procedimiento:

```
SimpleRunnable miRunnable = new SimpleRunnable("Hilo de prueba");
Thread miThread = new Thread(miRunnable);
miThread.start();
```

Threads

Creación de threads

Ejemplo, Creación de threads implementando la interface Runnable

El siguiente código crea un nuevo thread y lo ejecuta el segundo procedimiento:

```
SimpleRunnable miRunnable = new SimpleRunnable("Hilo de prueba");
Thread miThread = new Thread(miRunnable);
miThread.start();
```

Threads

Creación de threads

Ejemplo, Creación de threads implementando la interface Runnable

El siguiente código crea un nuevo thread y lo ejecuta el segundo procedimiento:

```
SimpleRunnable miRunnable = new SimpleRunnable("Hilo de prueba");
Thread miThread = new Thread(miRunnable);
miThread.start();
```

Threads

Creación de threads

Ejemplo, Creación de threads implementando la interface Runnable

El siguiente código crea un nuevo thread y lo ejecuta el segundo procedimiento:

```
SimpleRunnable miRunnable = new SimpleRunnable("Hilo de prueba");
Thread miThread = new Thread(miRunnable);
miThread.start();
```

Threads

Creación de threads

Elección ¿Thread o Runnable?

La elección de una u otra forma depende del tipo de clase que se vaya a crear. Así, si la clase a utilizar ya hereda de otra clase, no quedará más que implementar **Runnable**, aunque normalmente es más "sencillo" heredar de **Thread**.

Contenido

Threads

Ciclo de vida de un thread

En la sección anterior se ha visto cómo crear nuevos objetos que permiten incorporar en un programa la posibilidad de realizar varias tareas simultáneamente. En la Figura 2 se muestran los distintos estados por los que puede pasar un thread a lo largo de su vida.

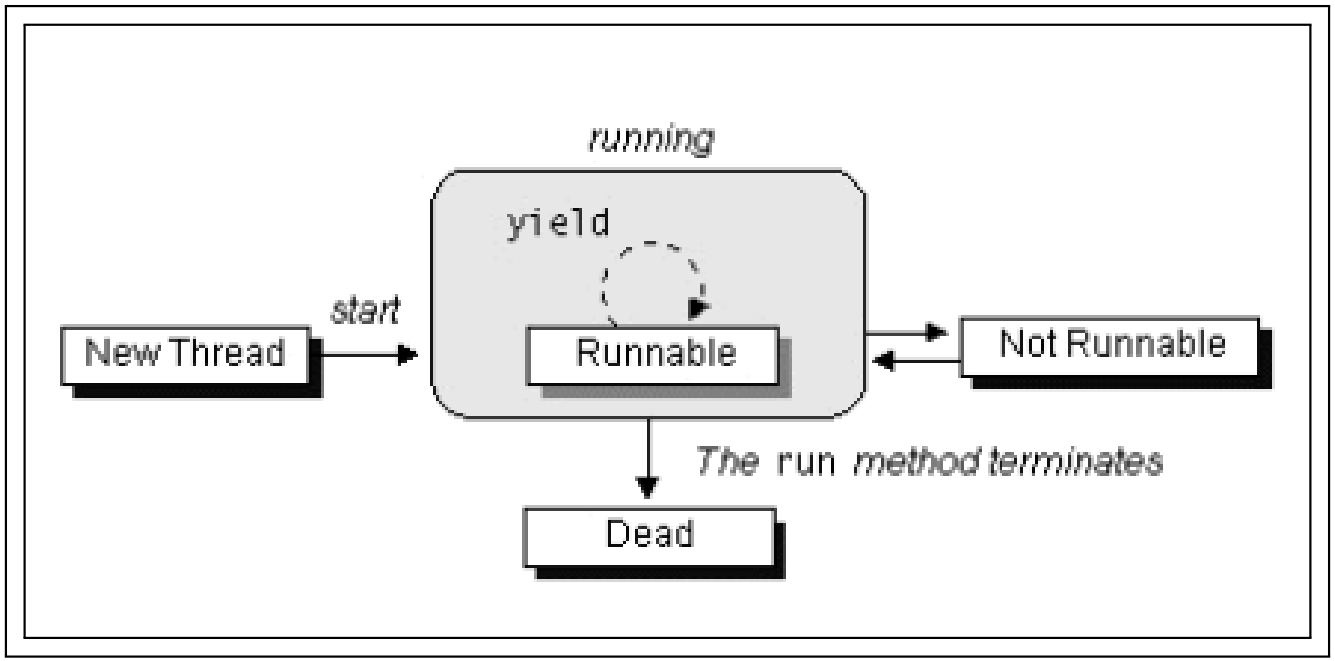

Figura 2: Ciclo de vida de un Thread.

Threads

Ciclo de vida de un thread

Ciclo de vida de un thread

Un **thread** puede presentar **cuatro** estados distintos:

1. Nuevo (**New**): El thread ha sido creado pero no inicializado, es decir, no se ha ejecutado el método **start**() (**IllegalThreadStateException**).
2. Ejecutable (**Runnable**): El thread puede estar ejecutándose, siempre y cuando se le haya asignado un determinado tiempo de CPU. En la práctica puede no estar siendo ejecutado en un instante determinado, en beneficio de otro **thread**.
3. Bloqueado (**Blocked** o **Not Runnable**): El thread podría estar ejecutándose, pero hay alguna actividad interna suya que lo impide, como por ejemplo una espera producida por una operación de escritura o lectura de datos por teclado (E/S). Si un thread está en este estado, no se le asigna tiempo de CPU.
4. Muerto (**Dead**): La forma habitual de que un **thread** muera es finalizando el método **run**(). También puede llamarse al método **stop**() de la clase **Thread**, aunque dicho método es considerado "peligroso" y no se debe utilizar.

Threads

Ciclo de vida de un thread

Ciclo de vida de un thread

Un **thread** puede presentar **cuatro** estados distintos:

1. Nuevo (**New**): El thread ha sido creado pero no inicializado, es decir, no se ha ejecutado el método **start**() (**IllegalThreadStateException**).
2. Ejecutable (**Runnable**): El thread puede estar ejecutándose, siempre y cuando se le haya asignado un determinado tiempo de CPU. En la práctica puede no estar siendo ejecutado en un instante determinado, en beneficio de otro **thread**.
3. Bloqueado (**Blocked** o **Not Runnable**): El thread podría estar ejecutándose, pero hay alguna actividad interna suya que lo impide, como por ejemplo una espera producida por una operación de escritura o lectura de datos por teclado (E/S). Si un thread está en este estado, no se le asigna tiempo de CPU.
4. Muerto (**Dead**): La forma habitual de que un **thread** muera es finalizando el método **run**(). También puede llamarse al método **stop**() de la clase **Thread**, aunque dicho método es considerado "peligroso" y no se debe utilizar.

Threads

Ciclo de vida de un thread

Ciclo de vida de un thread

Un **thread** puede presentar **cuatro** estados distintos:

1. Nuevo (**New**): El thread ha sido creado pero no inicializado, es decir, no se ha ejecutado el método **start**() (**IllegalThreadStateException**).
2. Ejecutable (**Runnable**): El thread puede estar ejecutándose, siempre y cuando se le haya asignado un determinado tiempo de CPU. En la práctica puede no estar siendo ejecutado en un instante determinado, en beneficio de otro **thread**.
3. Bloqueado (**Blocked** o **Not Runnable**): El thread podría estar ejecutándose, pero hay alguna actividad interna suya que lo impide, como por ejemplo una espera producida por una operación de escritura o lectura de datos por teclado (E/S). Si un thread está en este estado, no se le asigna tiempo de CPU.
4. Muerto (**Dead**): La forma habitual de que un **thread** muera es finalizando el método **run**(). También puede llamarse al método **stop**() de la clase **Thread**, aunque dicho método es considerado "peligroso" y no se debe utilizar.

Threads

Ciclo de vida de un thread

Ciclo de vida de un thread

Un **thread** puede presentar **cuatro** estados distintos:

1. Nuevo (**New**): El thread ha sido creado pero no inicializado, es decir, no se ha ejecutado el método **start**() (**IllegalThreadStateException**).
2. Ejecutable (**Runnable**): El thread puede estar ejecutándose, siempre y cuando se le haya asignado un determinado tiempo de CPU. En la práctica puede no estar siendo ejecutado en un instante determinado, en beneficio de otro **thread**.
3. Bloqueado (**Blocked** o **Not Runnable**): El thread podría estar ejecutándose, pero hay alguna actividad interna suya que lo impide, como por ejemplo una espera producida por una operación de escritura o lectura de datos por teclado (E/S). Si un thread está en este estado, no se le asigna tiempo de CPU.
4. Muerto (**Dead**): La forma habitual de que un **thread** muera es finalizando el método **run**(). También puede llamarse al método **stop**() de la clase **Thread**, aunque dicho método es considerado "peligroso" y no se debe utilizar.

Threads

Ciclo de vida de un thread

Ciclo de vida de un thread

Un **thread** puede presentar **cuatro** estados distintos:

1. Nuevo (**New**): El thread ha sido creado pero no inicializado, es decir, no se ha ejecutado el método **start**() (**IllegalThreadStateException**).
2. Ejecutable (**Runnable**): El thread puede estar ejecutándose, siempre y cuando se le haya asignado un determinado tiempo de CPU. En la práctica puede no estar siendo ejecutado en un instante determinado, en beneficio de otro **thread**.
3. Bloqueado (**Blocked** o **Not Runnable**): El thread podría estar ejecutándose, pero hay alguna actividad interna suya que lo impide, como por ejemplo una espera producida por una operación de escritura o lectura de datos por teclado (E/S). Si un thread está en este estado, no se le asigna tiempo de CPU.
4. Muerto (**Dead**): La forma habitual de que un **thread** muera es finalizando el método **run**(). También puede llamarse al método **stop**() de la clase **Thread**, aunque dicho método es considerado "peligroso" y no se debe utilizar.

Contenido

Threads

Sincronización

Sincronización

La **sincronización** nace de la necesidad de evitar que dos o más **threads** traten de acceder a los mismos recursos al mismo tiempo.

Caso

Así, por ejemplo, si un **thread** tratara de escribir en un fichero y otro thread estuviera al mismo tiempo tratando de borrar dicho fichero, se produciría una situación no deseada.

Threads

Sincronización

Sincronización

La **sincronización** nace de la necesidad de evitar que dos o más **threads** traten de acceder a los mismos recursos al mismo tiempo.

Caso

Así, por ejemplo, si un **thread** tratara de escribir en un fichero y otro thread estuviera al mismo tiempo tratando de borrar dicho fichero, se produciría una situación no deseada.

Threads

Sincronización

Secciones críticas

Las secciones de código de un programa que acceden a un mismo recurso (un mismo objeto de una clase, un fichero del disco, etc.) desde dos **threads** distintos se denominan secciones críticas (**critical sections**).

Sincronizar

Para sincronizar dos o más **threads**, hay que utilizar el modificador **synchronized** en aquellos métodos del **objeto-recurso** con los que puedan producirse situaciones conflictivas. De esta forma, Java bloquea (asocia un bloqueo o **lock**) con el recurso sincronizado.

Threads

Sincronización

Secciones críticas

Las secciones de código de un programa que acceden a un mismo recurso (un mismo objeto de una clase, un fichero del disco, etc.) desde dos **threads** distintos se denominan secciones críticas (**critical sections**).

Sincronizar

Para sincronizar dos o más **threads**, hay que utilizar el modificador **synchronized** en aquellos métodos del **objeto-recurso** con los que puedan producirse situaciones conflictivas. De esta forma, Java bloquea (asocia un bloqueo o **lock**) con el recurso sincronizado.

Threads

Sincronización

Sincronizar

La sincronización previene las interferencias solamente sobre un tipo de recurso: la memoria reservada para un objeto. Cuando se prevea que unas determinadas variables de una clase pueden tener problemas de sincronización, se deberán declarar como **private** (o **protected**). De esta forma sólo estarán accesibles a través de métodos de la clase, que deberán estar sincronizados.

Threads

Sincronización

Sincronizar

Es muy importante tener en cuenta que si se sincronizan algunos métodos de un objeto, pero otros no, el programa puede **no funcionar correctamente**. La razón es que los métodos **no sincronizados** pueden acceder libremente a las variables miembro, ignorando el **bloqueo del objeto**. Sólo los métodos sincronizados comprueban si un objeto está bloqueado. Por lo tanto, todos los métodos que accedan a un recurso compartido deben ser declarados **synchronized**.

Sincronizar - Niveles de bloqueo

Existen dos niveles de bloqueo de un recurso. El primero es a **nivel de objetos**, mientras que el segundo es a **nivel de clases**.

Threads

Sincronización

Sincronizar

Es muy importante tener en cuenta que si se sincronizan algunos métodos de un objeto, pero otros no, el programa puede **no funcionar correctamente**. La razón es que los métodos **no sincronizados** pueden acceder libremente a las variables miembro, ignorando el **bloqueo del objeto**. Sólo los métodos sincronizados comprueban si un objeto está bloqueado. Por lo tanto, todos los métodos que accedan a un recurso compartido deben ser declarados **synchronized**.

Sincronizar - Niveles de bloqueo

Existen dos niveles de bloqueo de un recurso. El primero es a **nivel de objetos**, mientras que el segundo es a **nivel de clases**.

Threads

Sincronización

Sincronizar - Niveles de bloqueo - A nivel de objeto

Se consigue declarando todos los métodos de una clase como **synchronized**. Cuando se ejecuta un método **synchronized** sobre un objeto concreto, el sistema bloquea dicho objeto, de forma que si otro **thread** intenta ejecutar algún método sincronizado de ese objeto, este segundo método se mantendrá a la espera hasta que finalice el anterior (y desbloquee por lo tanto el objeto).

Threads

Sincronización

Sincronizar - Niveles de bloqueo - A nivel de clases

Corresponde al bloqueo de los **métodos de clase** o **static**, y por lo tanto con las **variables de clase** o **static**. Si lo que se desea es conseguir que un método bloquee simultáneamente una clase entera, es decir todos los objetos creados de una clase, es necesario declarar este método como **synchronized static**.

Contenido

Threads

Prioridades

Prioridades

Con el fin de conseguir una correcta ejecución de un programa se establecen prioridades en los threads, de forma que se produzca un reparto más eficiente de los recursos disponibles.

Caso

En un determinado momento, interesará que un proceso termine lo antes posible sus cálculos, de forma que habrá que otorgarle más recursos (más tiempo de CPU). Esto no significa que el resto de procesos no requieran tiempo de CPU, sino que necesitarán menos.

Threads

Prioridades

Prioridades

Con el fin de conseguir una correcta ejecución de un programa se establecen prioridades en los threads, de forma que se produzca un reparto más eficiente de los recursos disponibles.

Caso

En un determinado momento, interesará que un proceso termine lo antes posible sus cálculos, de forma que habrá que otorgarle más recursos (más tiempo de CPU). Esto no significa que el resto de procesos no requieran tiempo de CPU, sino que necesitarán menos.

Threads

Prioridades

Prioridades

Cuando se crea un nuevo **thread**, éste hereda la prioridad del **thread** desde el que ha sido inicializado. Las prioridades viene definidas por variables miembro de la clase **Thread**, que toman valores enteros que oscilan entre la máxima prioridad **MAX_PRIORITY** (normalmente tiene el valor 10) y la mínima prioridad **MIN_PRIORITY** (valor 1), siendo la prioridad por defecto **NORM_PRIORITY** (valor 5). Para modificar la prioridad de un thread se utiliza el método **setPriority**(). Se obtiene su valor con **getPriority**().

Threads

Prioridades

Prioridades

Un **thread** puede en un determinado momento renunciar a su tiempo de CPU y otorgárselo a otro thread de la misma prioridad, mediante el método **yield**(), aunque en ningún caso a un **thread** de prioridad inferior.

Contenido

Threads

Grupos de threads

Grupos de threads

Todo hilo de Java **debe** formar parte de un grupo de hilos (**ThreadGroup**). Puede pertenecer al grupo por defecto o a uno explícitamente creado por el usuario. Los grupos de threads proporcionan una forma sencilla de manejar múltiples threads como un sólo objeto. Así, por ejemplo es posible parar varios **threads** con una sola llamada al método correspondiente. Una vez que un **thread** ha sido asociado a un **ThreadGroup**, no puede cambiar de grupo.

Threads

Grupos de threads

Grupos de threads

Cuando se arranca un programa, el sistema crea un **ThreadGroup** llamado **main**. Si en la creación de un nuevo thread no se especifica a qué grupo pertenece, automáticamente pasa a pertenecer al **ThreadGroup** del **thread** desde el que ha sido creado (conocido como **current thread group** y **current thread**, respectivamente). Si en dicho programa no se crea ningún **ThreadGroup** adicional, todos los **threads** creados pertenecerán al grupo **main** (en este grupo se encuentra el método main()).

Threads

Grupos de threads

Grupos de threads

Cuando se arranca un programa, el sistema crea un **ThreadGroup** llamado **main**. Si en la creación de un nuevo thread no se especifica a qué grupo pertenece, automáticamente pasa a pertenecer al **threadgroup** del **thread** desde el que ha sido creado (conocido como **current thread group** y **current thread**, respectivamente). Si en dicho programa no se crea ningún **ThreadGroup** adicional, todos los **threads** creados pertenecerán al grupo **main** (en este grupo se encuentra el método main()).

Threads

Grupos de threads

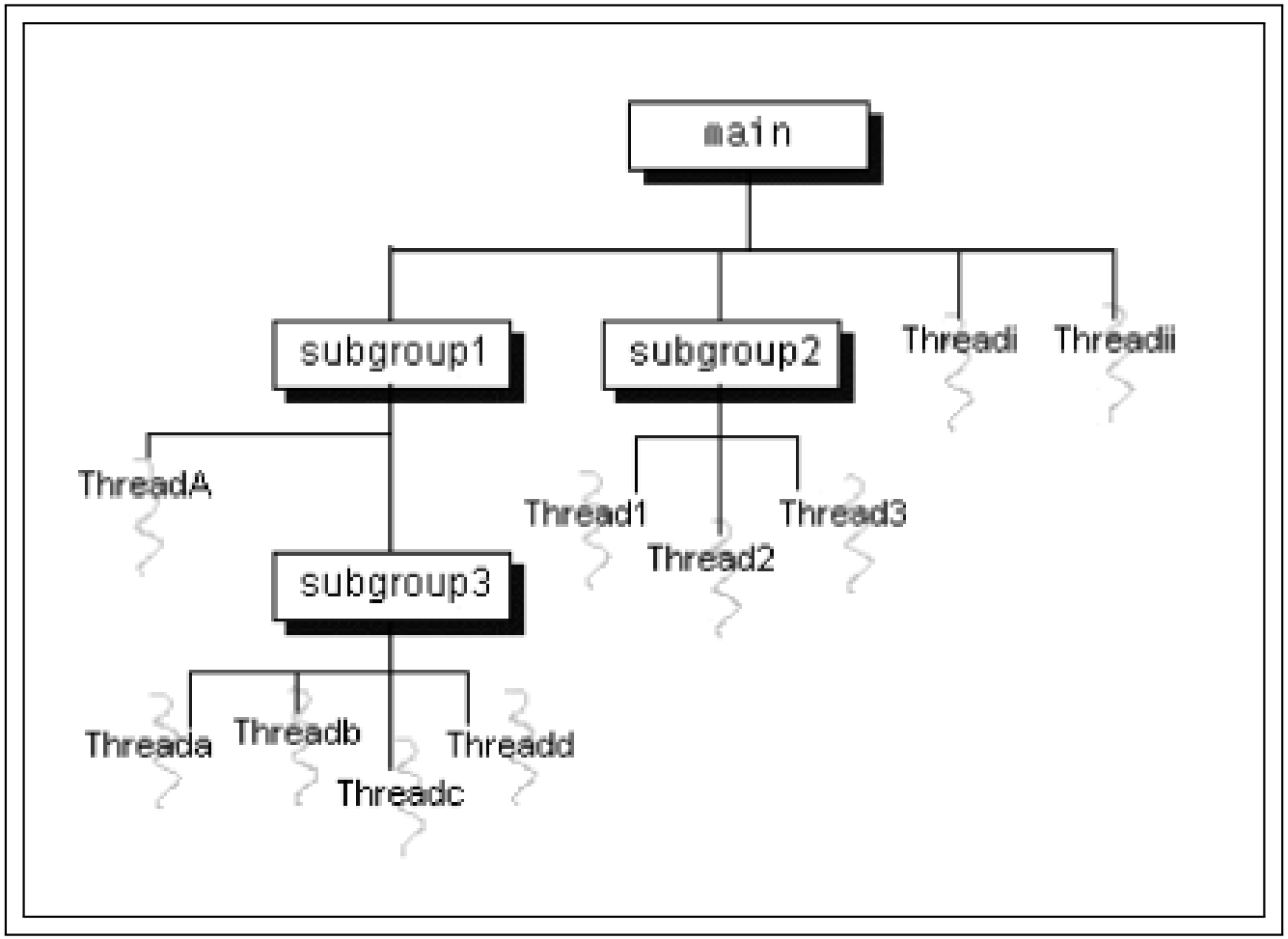

Figura 3: Representación de los grupos de Threads.

Threads

Grupos de threads

Grupos de threads

Para conseguir que un **thread** pertenezca a un grupo concreto, hay que indicarlo al crear el nuevo thread, según uno de los siguientes constructores:

public Thread (**ThreadGroup** grupo, **Runnable** destino)

public Thread (**ThreadGroup** grupo, **String** nombre)

public Thread (**ThreadGroup** grupo, **Runnable** destino, **String** nombre)

Threads

Grupos de threads

Grupos de threads

Para conseguir que un **thread** pertenezca a un grupo concreto, hay que indicarlo al crear el nuevo thread, según uno de los siguientes constructores:

public Thread (**ThreadGroup** grupo, **Runnable** destino)

public Thread (**ThreadGroup** grupo, **String** nombre)

public Thread (**ThreadGroup** grupo, **Runnable** destino, **String** nombre)

Threads

Grupos de threads

Grupos de threads

Para conseguir que un **thread** pertenezca a un grupo concreto, hay que indicarlo al crear el nuevo thread, según uno de los siguientes constructores:

public Thread (**ThreadGroup** grupo, **Runnable** destino)

public Thread (**ThreadGroup** grupo, **String** nombre)

public Thread (**ThreadGroup** grupo, **Runnable** destino, **String** nombre)

Threads

Grupos de threads

Grupos de threads

Para conseguir que un **thread** pertenezca a un grupo concreto, hay que indicarlo al crear el nuevo thread, según uno de los siguientes constructores:

public Thread (**ThreadGroup** grupo, **Runnable** destino)

public Thread (**ThreadGroup** grupo, **String** nombre)

public Thread (**ThreadGroup** grupo, **Runnable** destino, **String** nombre)

Threads

Grupos de threads

Grupos de threads

A su vez, un **ThreadGroup** debe pertenecer a otro **ThreadGroup**. Como ocurría en el caso anterior, si no se especifica ninguno, el nuevo grupo pertenecerá al **ThreadGroup** desde el que ha sido creado (por defecto al grupo **main**). La clase **ThreadGroup** tiene dos posibles constructores:

public ThreadGroup (**ThreadGroup** parent, **String** nombre)

public ThreadGroup (**String** nombre)

Threads

Grupos de threads

Grupos de threads

A su vez, un **ThreadGroup** debe pertenecer a otro **ThreadGroup**. Como ocurría en el caso anterior, si no se especifica ninguno, el nuevo grupo pertenecerá al **ThreadGroup** desde el que ha sido creado (por defecto al grupo **main**). La clase **ThreadGroup** tiene dos posibles constructores:

public ThreadGroup (**ThreadGroup** parent, **String** nombre)

public ThreadGroup (**String** nombre)

Threads

Grupos de threads

Grupos de threads

A su vez, un **ThreadGroup** debe pertenecer a otro **ThreadGroup**. Como ocurría en el caso anterior, si no se especifica ninguno, el nuevo grupo pertenecerá al **ThreadGroup** desde el que ha sido creado (por defecto al grupo **main**). La clase **ThreadGroup** tiene dos posibles constructores:

public ThreadGroup (**ThreadGroup** parent, **String** nombre)

public ThreadGroup (**String** nombre)

Preguntas

Preguntas ?

Threads

Ejercicio

Ejercicio

Simular el proceso de cobro de un supermercado; unos clientes van con un carro lleno de productos y una cajera les cobra los productos, pasándolos uno a uno por el escáner de la caja registradora. En este caso la cajera debe de procesar la compra cliente a cliente, es decir que primero le cobra al cliente 1, luego al cliente 2 y así sucesivamente. Para ello vamos a definir una clase "**Cajera**" y una clase "**Cliente**" el cual tendrá un "array de enteros" que representarán los productos que ha comprado y el tiempo que la cajera tardará en pasar el producto por el escáner; es decir, que si tenemos un array con **[1,3,5]** significará que el cliente ha comprado 3 productos y que la cajera tardara en procesar el producto 1 **'1 segundo'**, el producto 2 **'3 segundos'** y el producto 3 en **'5 segundos'**, con lo cual tardará en cobrar al cliente toda su compra **'9 segundos'**.

Threads

Ejercicio

Ejercicio

Simular el proceso de parking o estacionamiento que poseé una única entrada y una única salida. La idea es controlar el ingreso y egreso de autómoviles, de tal manera que no sea factible el ingreso de un nuevo vehículo si el estacionamiento está lleno y no es factible sacar un autómovil si el estacionamiento está vacío. Considere como **región crítica** el número actual de vehículos en el estacionamiento, por lo cual debe utilizar **bloqueos/sincronización de thread**.

Threads

Ejercicio

Ejercicio

Simular el proceso de deposito y giro de dinero de una cuenta corriente. La idea es controlar el ingreso y egreso de dinero, de tal manera que no sea factible girar dinero si la cuenta está en valor negativo y no sea factible mantener más de $100.000 (cien mil pesos) en la cuenta. Cada transación debe esperar un número aletorio de segundos (entre 1 y 5) y el monto a depositar o girar también deberá ser aleatorio entre $1.000 (mil) y $10.000 (diez mill). Considere como **región crítica** la clase que gestiona la cuenta corriente (el saldo), por lo cual debe utilizar **bloqueos/sincronización de thread**.