UT1 Concurrencia.
Programación multihilo en Java.

Módulo - Programación de Servicios y Procesos
Ciclo - Desarrollo de Aplicaciones Multiplataforma
IES María Ana Sanz

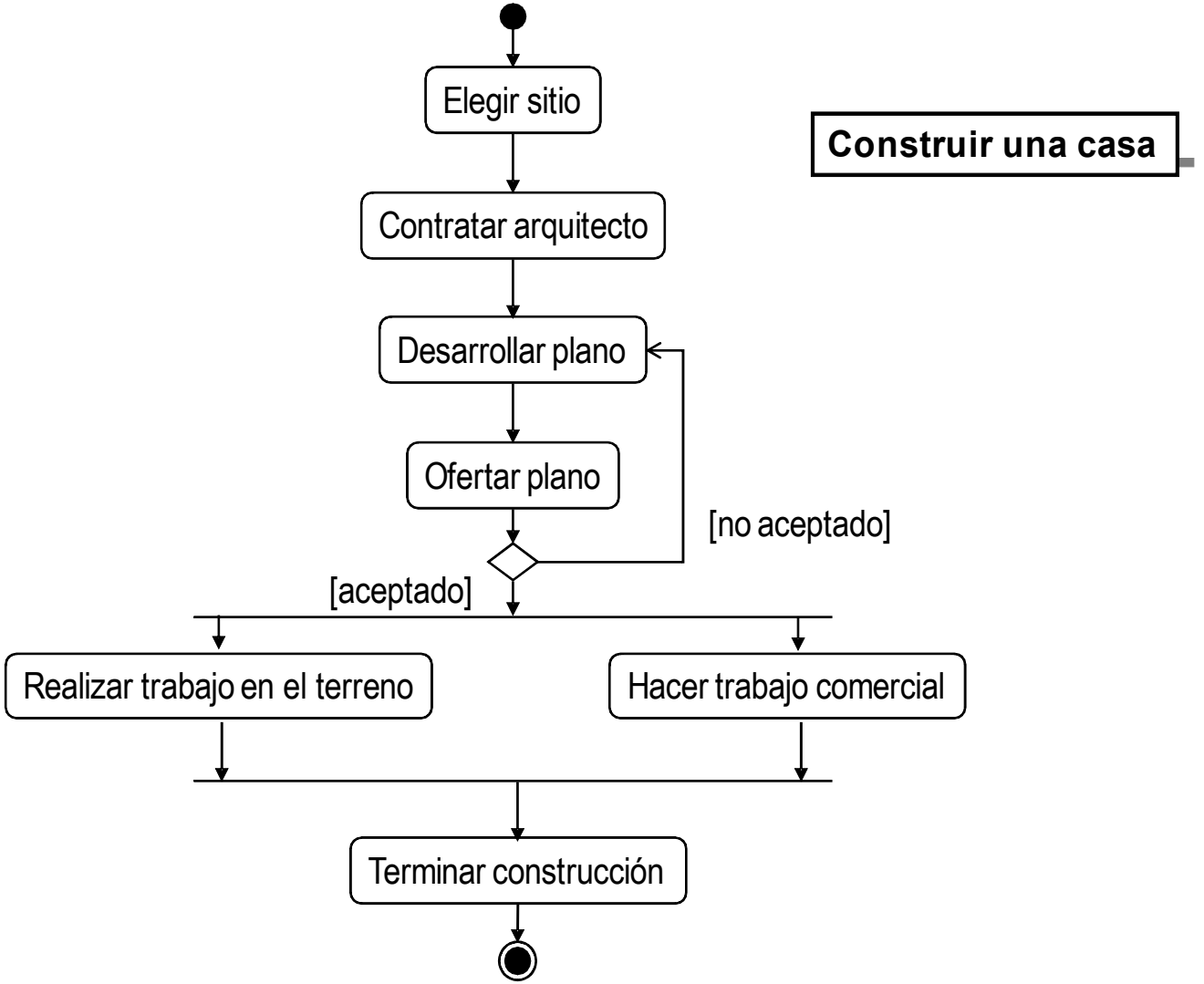
Resumen de contenidos

- Programación concurrente
  - procesos / hilos
- Hilos en Java
  - la clase Thread
  - el interfaz Runnable
  - cómo crear y ejecutar hilos?
- Ciclo de vida de un hilo
- Sincronización de hilos
  - exclusión mutua
  - cooperación de hilos (wait() / notify()
    - ➔ productor/consumidor
- Animaciones
  - clase javax.swing.Timer
  - hilos y animaciones
- Hilos y Swing

Concurrencia

- Programación concurrente
  - Ejecución de varias actividades a la vez, en paralelo
- Objetivo
  - Optimizar los recursos del sistema
    - ➔ Si un proceso (una aplicación) está esperando la finalización de una operación de E/S, otros procesos pueden aprovechar el procesador del sistema que en ese momento no se usa

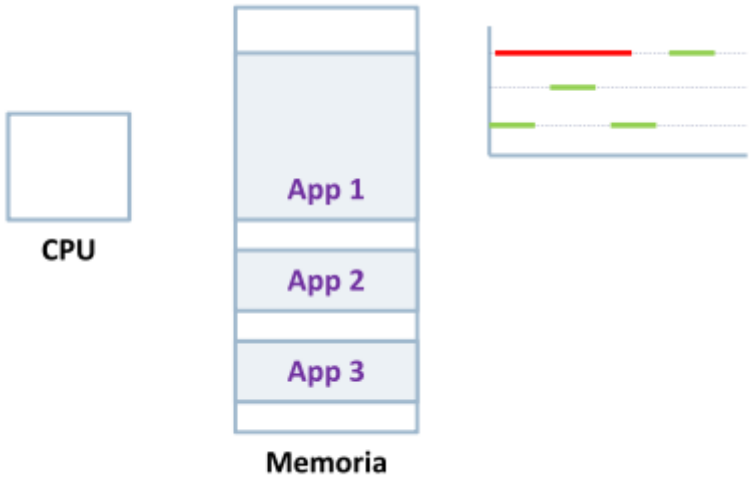
Concurrencia en la vida real



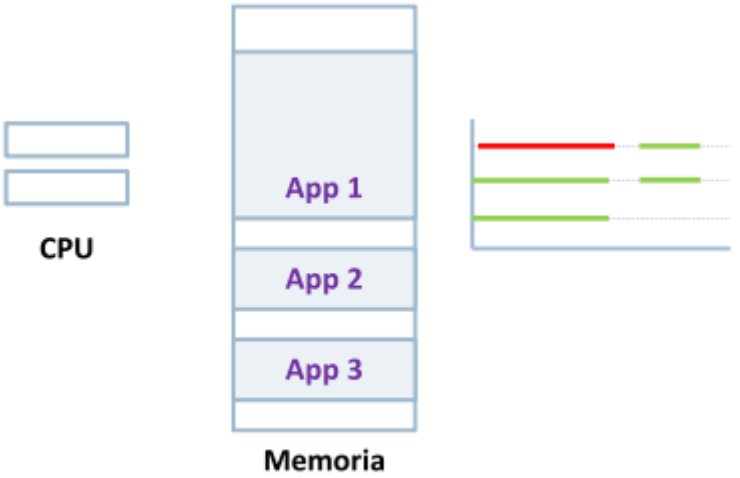
Procesos vs hilos

- Son las unidades básicas de ejecución en la programación concurrente
- Representan un flujo de control en ejecución
- En el contexto de un Sistema Operativo
  - Un proceso es un programa que está siendo ejecutado
    - → **Proceso = programa + datos + recursos**
- En el contexto de un programa concurrente
  - Un hilo es cada uno de los flujos secuenciales de control independientes especificados en el programa

Procesos



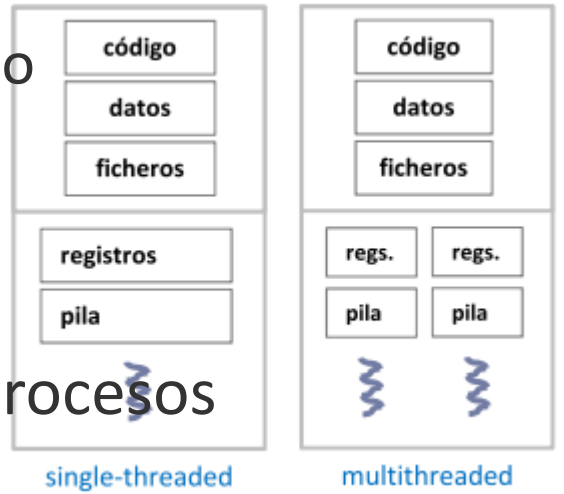
Varios procesos y una CPU
Se reparte el uso del procesador



Varios procesos y varias CPU
Hay paralelismo real

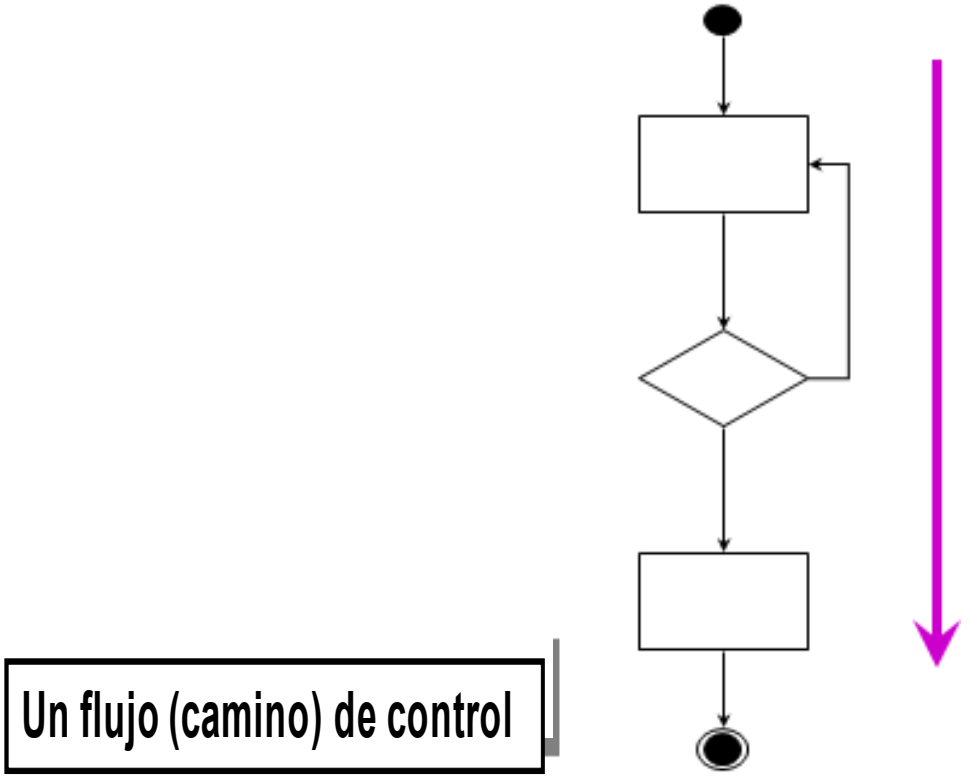
Procesos vs hilos

- **Procesos**
  - disponen de su propio espacio de memoria
  - pueden comunicarse entre ellos a través de pipes y sockets
- **Hilos**
  - flujos de ejecución dentro de un mismo proceso
  - *Thread* – hilo (hebra) – procesos ligeros
  - su espacio de memoria es compartido
- Java pone más énfasis en los hilos que en los procesos
  - la JVM es un único proceso con varios hilos



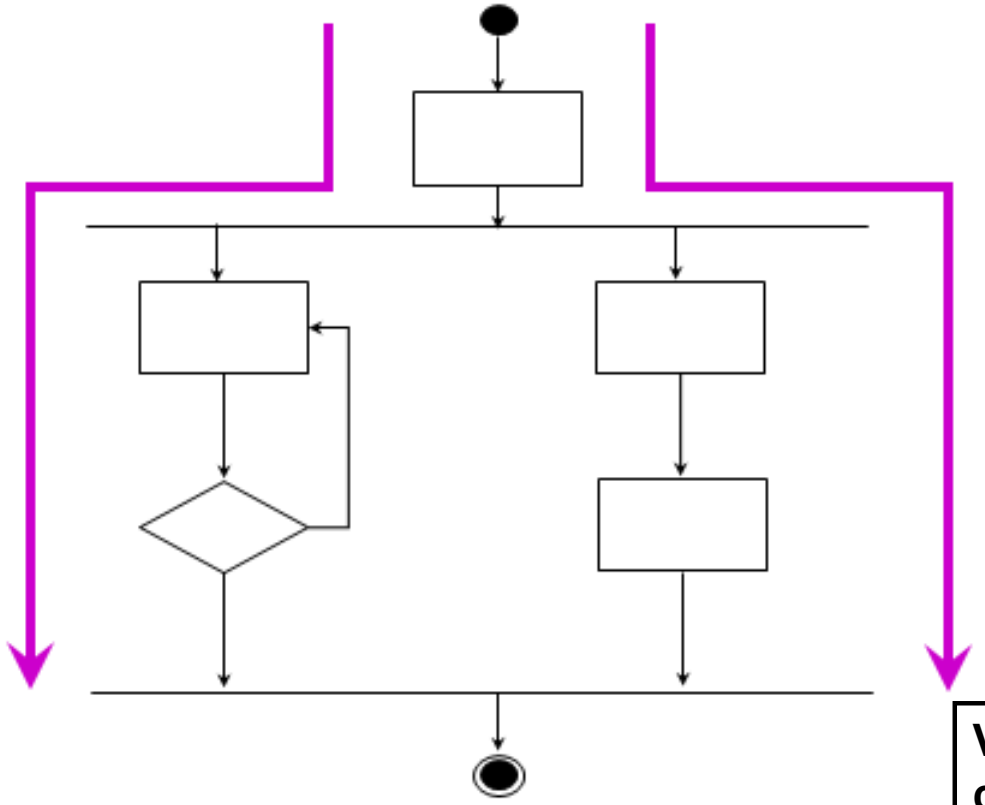
Proceso secuencial

- Proceso que posee un único hilo de control
  - se corresponde con un programa secuencial
    - ➔ lo visto hasta ahora



Proceso concurrente

- Un programa concurrente da lugar durante su ejecución a un proceso con varios hilos de ejecución

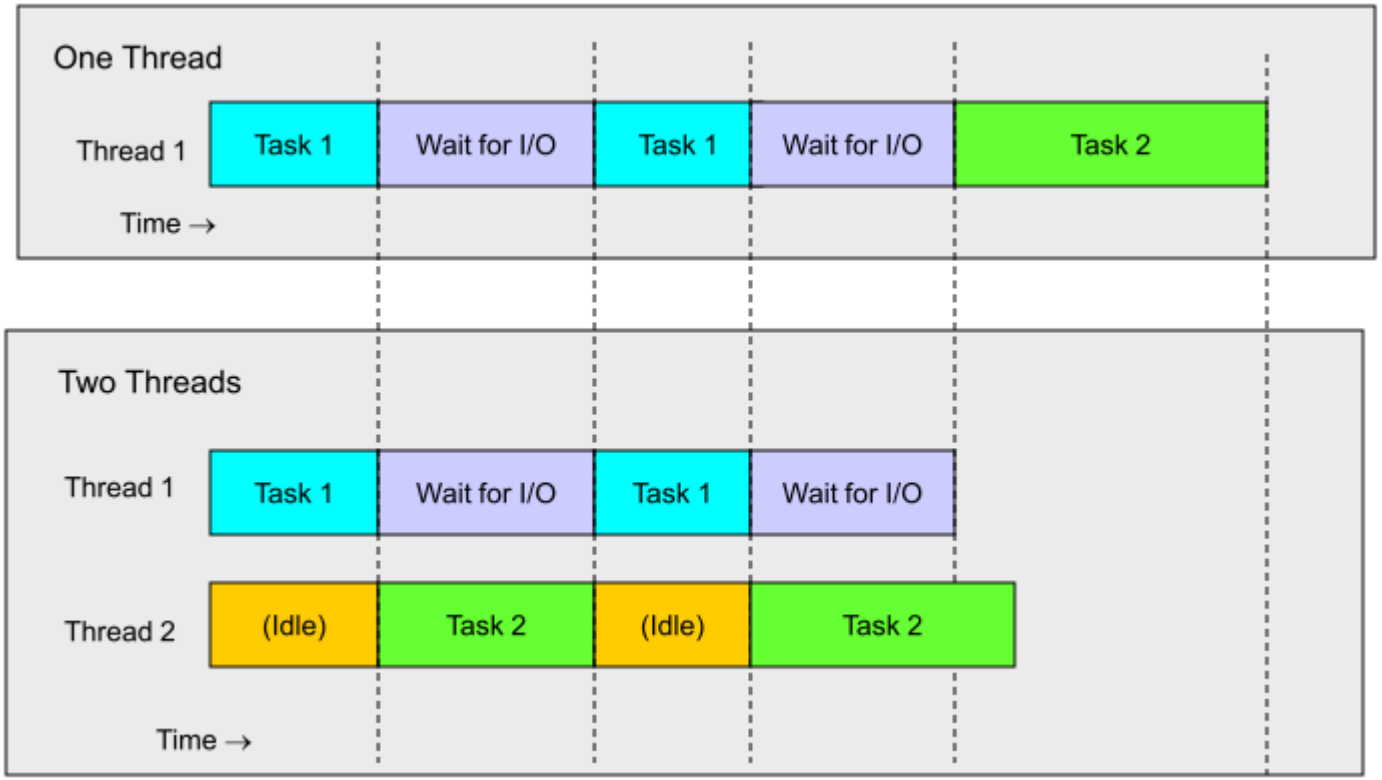


Varios flujos (caminos) de control

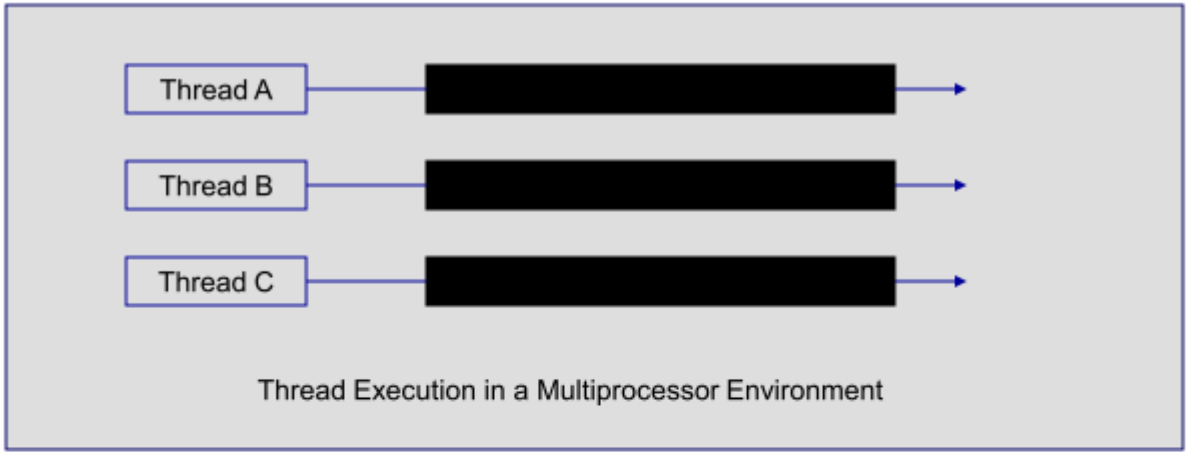
¿Qué es un hilo?

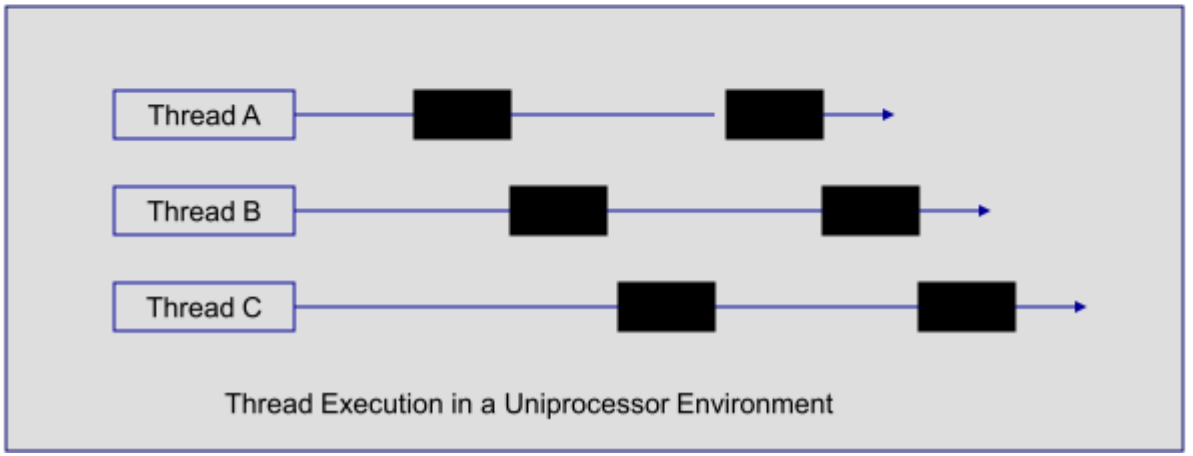
- Flujo secuencial de ejecución independiente dentro de un programa
- Todo programa tiene al menos un hilo
- Un programa con más de un hilo es un programa multihilo
- Todos los hilos comparten el procesador (si solo hay uno)
- El SO decide cuándo cada hilo consigue la CPU
  - *scheduling* (planificación de hilos)
- La programación multihilo puede ser complicada sobre todo cuando se requiere sincronización entre hilos

¿Qué es un hilo?



¿Qué es un hilo?

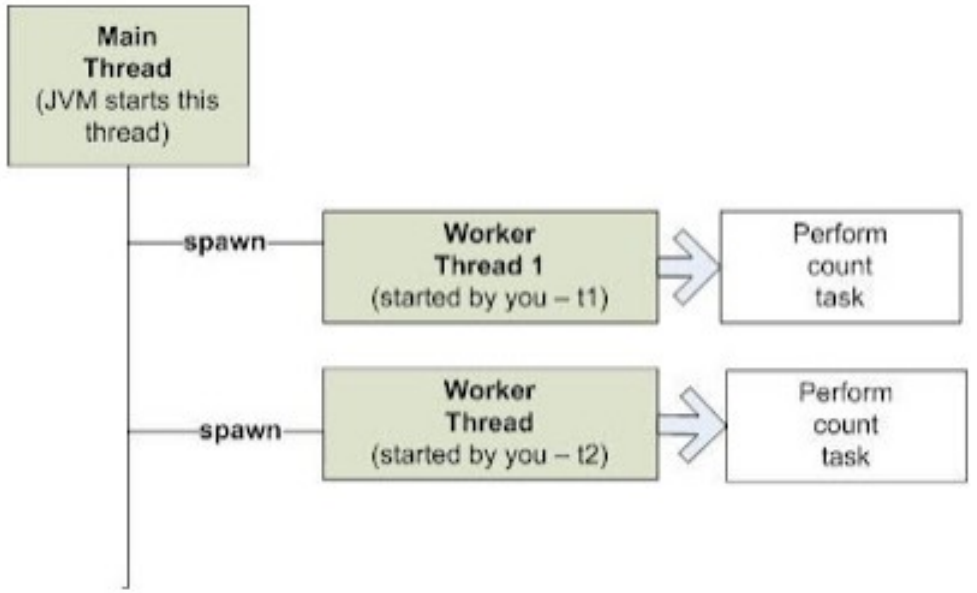




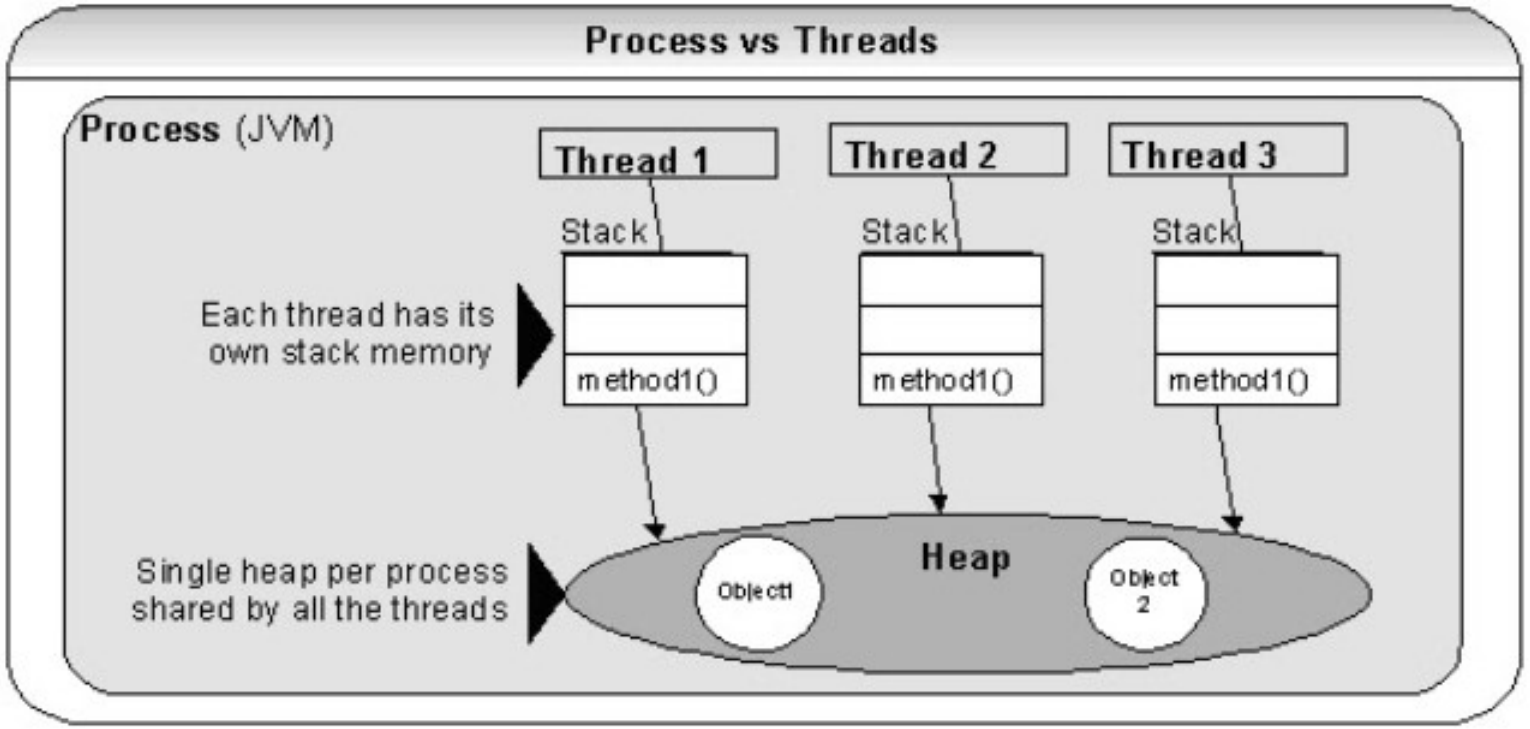
Hilos en Java

- Cuando se inicia un programa en Java
  - la máquina virtual crea un hilo principal
  - este hilo se encarga de invocar al método main() de la clase que se comience a ejecutar
  - el hilo termina cuando finaliza el método main()
  - si el hilo principal crea otros hilos, éstos se ejecutan de forma concurrente
  - cuando no queda ningún hilo activo el programa termina
- Todos los hilos se ejecutan en la misma máquina virtual (mismo proceso)

Hilos en Java



Hilos en Java



- Los hilos comparten el **heap** y tienen su propia **pila**
  - la llamada a un método desde un hilo y sus variables locales son ***thread safe***
  - el heap no es ***thread safe*** y debe sincronizarse

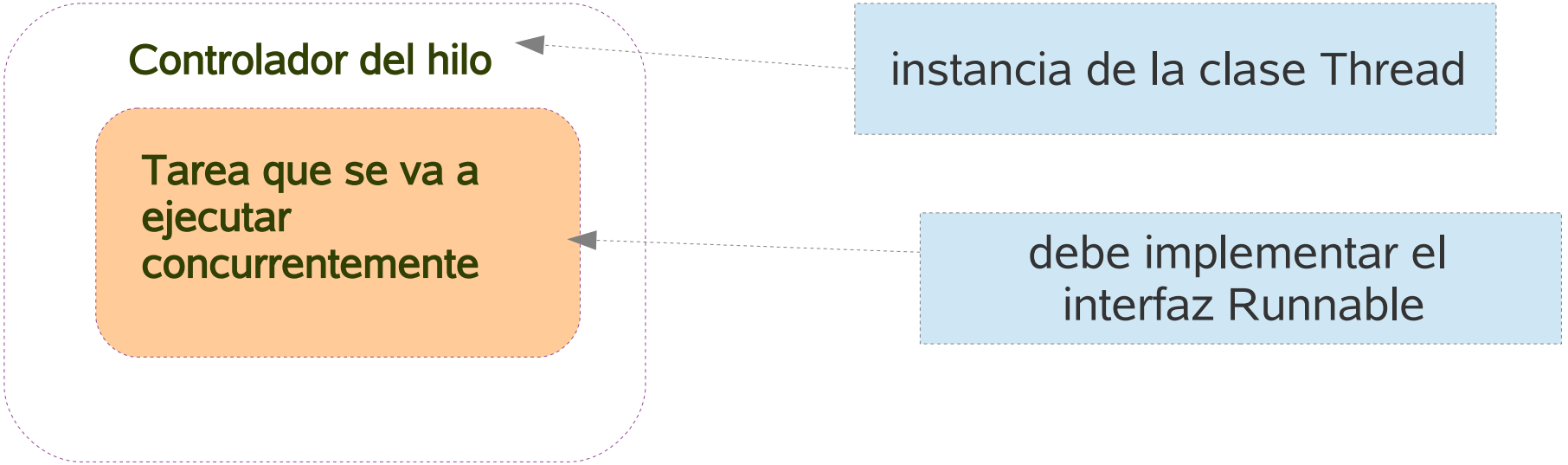
Planificador de hilos (Thread Scheduler)

- Habitualmente hay más hilos que CPU
- El ***Thread Scheduler*** es el responsable de asignar la CPU de alguna manera entre todos los hilos que compiten por ella
- Es parte del SO
- Los SO asignan a cada hilo una pequeña cantidad de tiempo de procesador llamada *quantum* o *time slice*. Cuando este tiempo termina el hilo deja de ejecutarse y el SO asigna a otro hilo nuevamente tiempo de procesador
- En los SO modernos se utiliza *preemtive scheduling*
  - los hilos más prioritarios se ejecutan antes
    http://www.javamex.com/tutorials/threads/thread_scheduling.shtml

Aplicaciones de los hilos

- Para mejorar el rendimiento de aplicaciones que hacen un uso extensivo de operaciones de IO
- Para mejorar la respuesta de las aplicaciones con interfaz gráfico
  - una aplicación con GUI puede quedar “congelada” en un programa con un único hilo si al pulsar un botón, por ejemplo, comienza a realizar un cálculo que le ocupe mucho tiempo. La GUI no responde hasta que el cálculo no termina.
- Para atender las peticiones de dos o más clientes simultáneamente en aplicaciones basadas en servidor
  - servidor web, un chat
- Para efectuar animaciones

Crear hilos en Java



Idea de cómo crear un hilo en Java gráficamente

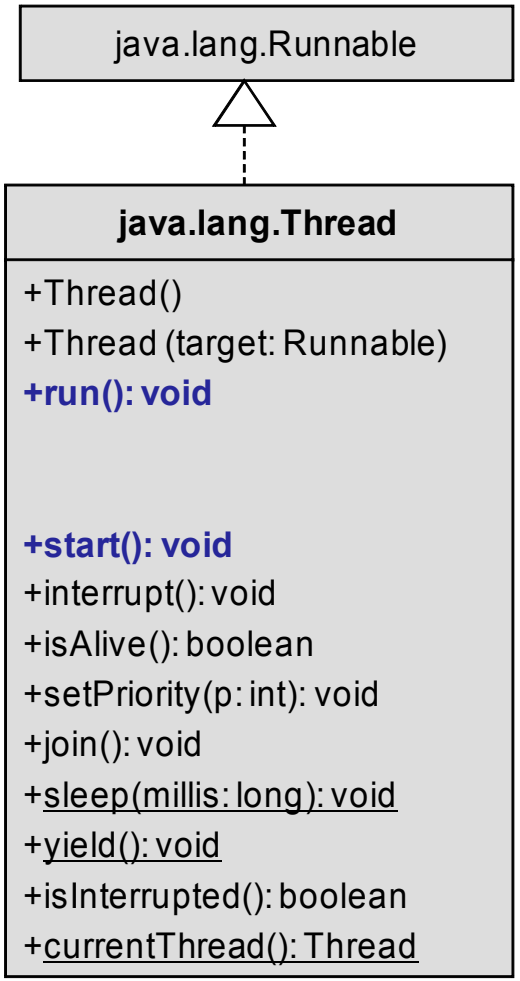
- identificar la tarea a ejecutar concurrentemente
- el controlador del hilo, el objeto que facilita la ejecución de la tarea

Crear hilos en Java

- Dos formas básicas
  - Extendiendo la clase Thread
  - Implementando el interface Runnable
- Thread y Runnable en *java.lang*

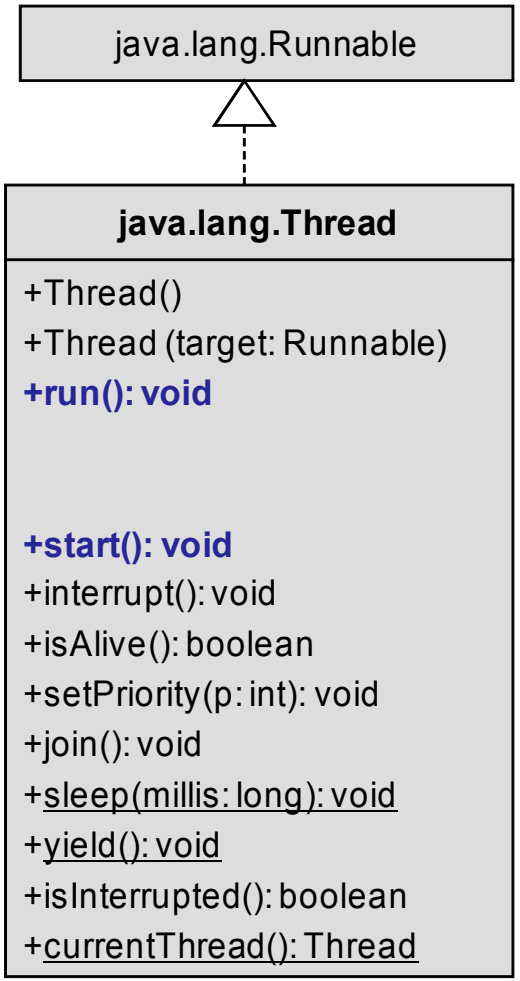
- Otra posibilidad
  - a través del Executor framework (usando un *thread pool*)

La clase Thread



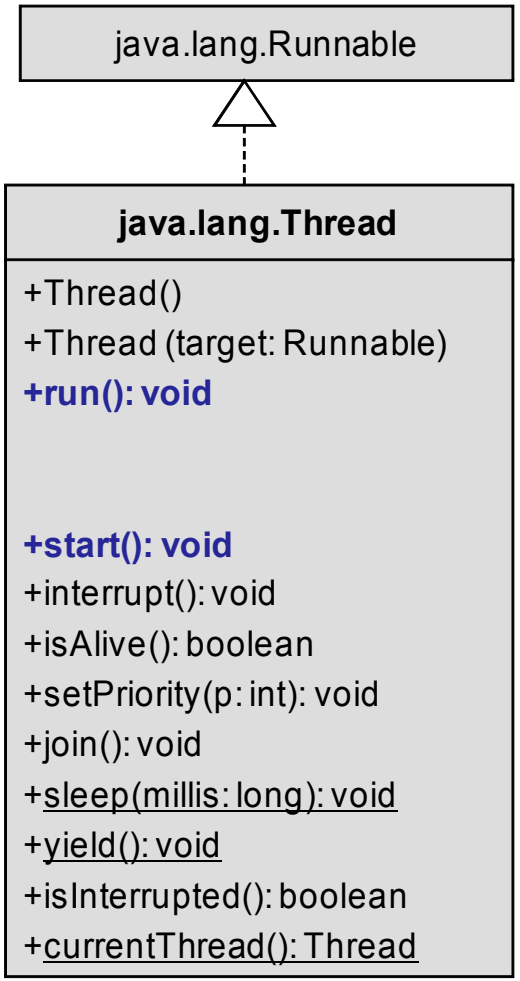
Constructor	Descripción
Thread()	crea un hilo por defecto
Thread(Runnable)	crea un hilo a partir de cualquier objeto que implemente el interface Runnable
Thread(String)	crea un hilo con un nombre específico
Thread(Runnable, String)	crea un hilo a partir de cualquier objeto que implemente el interface Runnable y con un nombre

La clase Thread



Método	Descripción
run()	Invocado por la JVM para ejecutar el hilo. Se debe redefinir este método y proporcionar el código que queramos que nuestro hilo ejecute. Nunca se invoca este método directamente.
start()	crea un hilo a partir de cualquier objeto que implemente el interface Runnable. Sitúa el hilo en su estado Runnable (puede ser elegido por el planificador)
getName()	devuelve el nombre del hilo
currentThread()	Método estático, devuelve una referencia al hilo ejecutándose actualmente
yield()	Método estático, el hilo actualmente en ejecución se pausa para que otros hilos puedan ejecutarse

La clase Thread



Método	Descripción
sleep(long)	Método estático, pone a dormir al hilo actualmente en ejecución (lo pasa al estado bloqueado) durante el nº determinado de milisegundos. Otros hilos pueden así ejecutarse
interrupt()	Interrumpe un hilo
isInterrupted()	Devuelve true si un hilo ha sido interrumpido
setPriority(int)	Establece la prioridad del hilo a n (rango de 1 a 10)
isAlive()	Testea si el hilo actual está ejecutándose
setDaemon(boolean)	Marca un hilo como hilo demonio

El interfaz Runnable

```
public interface Runnable
{
    public abstract void run();
}
```

- las clases que implementen este interfaz deberán proporcionar código al método run()
- la clase Thread implementa Runnable

Creando un hilo extendiendo la clase Thread

- crear una clase que herede de Thread
- redefinir el método run() para ejecutar la tarea deseada
  - el código de este método define lo que va a hacer el hilo durante su ejecución
- crear el hilo instanciando un objeto de esta clase
- llamar al método start() sobre el hilo creado en el paso anterior
  - este método se encarga de llamar a run()

Creando un hilo extendiendo la clase Thread

```
public class Saludo extends Thread
{

   public Saludo()
   {

   }

   public void run()
   {
      for (int i = 1; i <= 10; i++)
      {
         System.out.println(this.getName() + " " + i + "Hola");
         Thread.yield(); // damos oportunidad de que se ejecuten otros hilos
      }
   }
}
```

lo que el hilo va a hacer

Proyecto Hola Adios con Thread

Creando un hilo extendiendo la clase Thread

```
public class Despedida extends Thread
{

   public Despedida()
   {

   }


   public void run()
   {
      for (int i = 1; i <= 10; i++)
      {
         System.out.println(this.getName() + " " + i + "Adiós");
         Thread.yield(); // damos oportunidad de que se ejecuten otros hilos
      }
   }
}
```

Creando un hilo extendiendo la clase Thread

```
public class TestSaludoDespedida
{

   public static void main(String[] args)
   {
       // creo el hilo Saludo
       Thread hilo1 = new Saludo(); // también Saludo hilo1 = new Saludo();
       hilo1.start();  // inicio el hilo

       // creo el hilo Despedida
       Thread hilo2 = new Despedida(); // también Despedida hilo2 = new Despedida();
       hilo2.start();  // inicio el hilo
       System.out.println("Fin del hilo main()");

   }
}
```

No llamar directamente a run()

Si llamamos a run() se ejecuta su código pero en el hilo actual, es decir, el hilo main() y no en el nuevo hilo

posible ejecución

```
Fin del hilo main()
Thread-9 1 Adiós
Thread-9 2 Adiós
Thread-9 3 Adiós
Thread-9 4 Adiós
Thread-9 5 Adiós
Thread-9 6 Adiós
Thread-9 7 Adiós
Thread-8 1 Hola
Thread-9 8 Adiós
Thread-8 2 Hola
Thread-9 9 Adiós
Thread-8 3 Hola
Thread-9 10 Adiós
Thread-8 4 Hola
Thread-8 5 Hola
Thread-8 6 Hola
Thread-8 7 Hola
Thread-8 8 Hola
Thread-8 9 Hola
Thread-8 10 Hola
```

Ejercicios

Creando un hilo implementando Runnable

- crear una clase que implemente el interface Runnable
- implementar el método run() para ejecutar la tarea deseada
  - el código de este método define lo que va a hacer el hilo durante su ejecución
- crear el hilo proporcionando una instancia de tipo Runnable al constructor de Thread
- llamar al método start() sobre el hilo creado en el paso anterior
  - este método se encarga de llamar a run()

Creando un hilo implementando Runnable

```
public class Saludo implements Runnable
{
    public void run()
    {
        Thread t = Thread.currentThread();
        for (int i = 1; i <= 10; i++)
        {
            System.out.println(t.getName() + " " + i + " Hola");
            Thread.yield(); // damos oportunidad de que se ejecuten otros hilos
        }
    }
}
public class Despedida implements Runnable
{
    public void run()
    {
        Thread t = Thread.currentThread();
        for (int i = 1; i <= 10; i++)
        {
            System.out.println(t.getName() + " " + i + " Adiós");
            Thread.yield(); // damos oportunidad de que se ejecuten otros hilos
        }
    }
}
```

la tarea a realizar de forma concurrente

Proyecto Hola Adios con Runnable

Creando un hilo implementando Runnable

```
public class TestSaludoDespedida
{
   public static void main(String[] args)
   {

      Thread hilo1 = new Thread(new Saludo()); // pasamos al constructor el objeto runn
      hilo1.start();  // inicio el hilo

      Thread hilo2 = new Thread(new Despedida());
      hilo2.start();  // inicio el hilo
      System.out.println("Fin del hilo main()");

   }
}
```

el objeto Runnable

Forma recomendable de trabajar con hilos (implementando Runnable).

Más flexible, permite combinar el interfaz con lo que se herede de otra clase
public MiPanel extends JPanel implements Runnable

posible ejecución

```
Thread-2 1 Hola
Fin del hilo main()
Thread-2 2 Hola
Thread-2 3 Hola
Thread-2 4 Hola
Thread-2 5 Hola
Thread-2 6 Hola
Thread-2 7 Hola
Thread-2 8 Hola
Thread-2 9 Hola
Thread-2 10 Hola
Thread-3 1 Adiós
Thread-3 2 Adiós
Thread-3 3 Adiós
Thread-3 4 Adiós
Thread-3 5 Adiós
Thread-3 6 Adiós
Thread-3 7 Adiós
Thread-3 8 Adiós
Thread-3 9 Adiós
Thread-3 10 Adiós
```

Otra forma utilizando Runnable

```
public class Saludo implements Runnable
{
   public Saludo()
   {
      Thread hilo = new Thread(this);
      hilo.start();
   }

   public void run()
   {
       Thread t = Thread.currentThread();
       for (int i = 1; i <= 10; i++)
       {
          System.out.println(t.getName() + " " + i + " Hola");
          Thread.yield(); // damos oportunidad de que se ejecuten otros hilos
       }
   }
}
```

```
public class TestSaludoDespedida
{
   public static void main(String[] args)
   {
       new Saludo();
       new Despedida();


   }
}
```

Proyecto Hola Adios otras alternativas

Otra forma con clase anónima

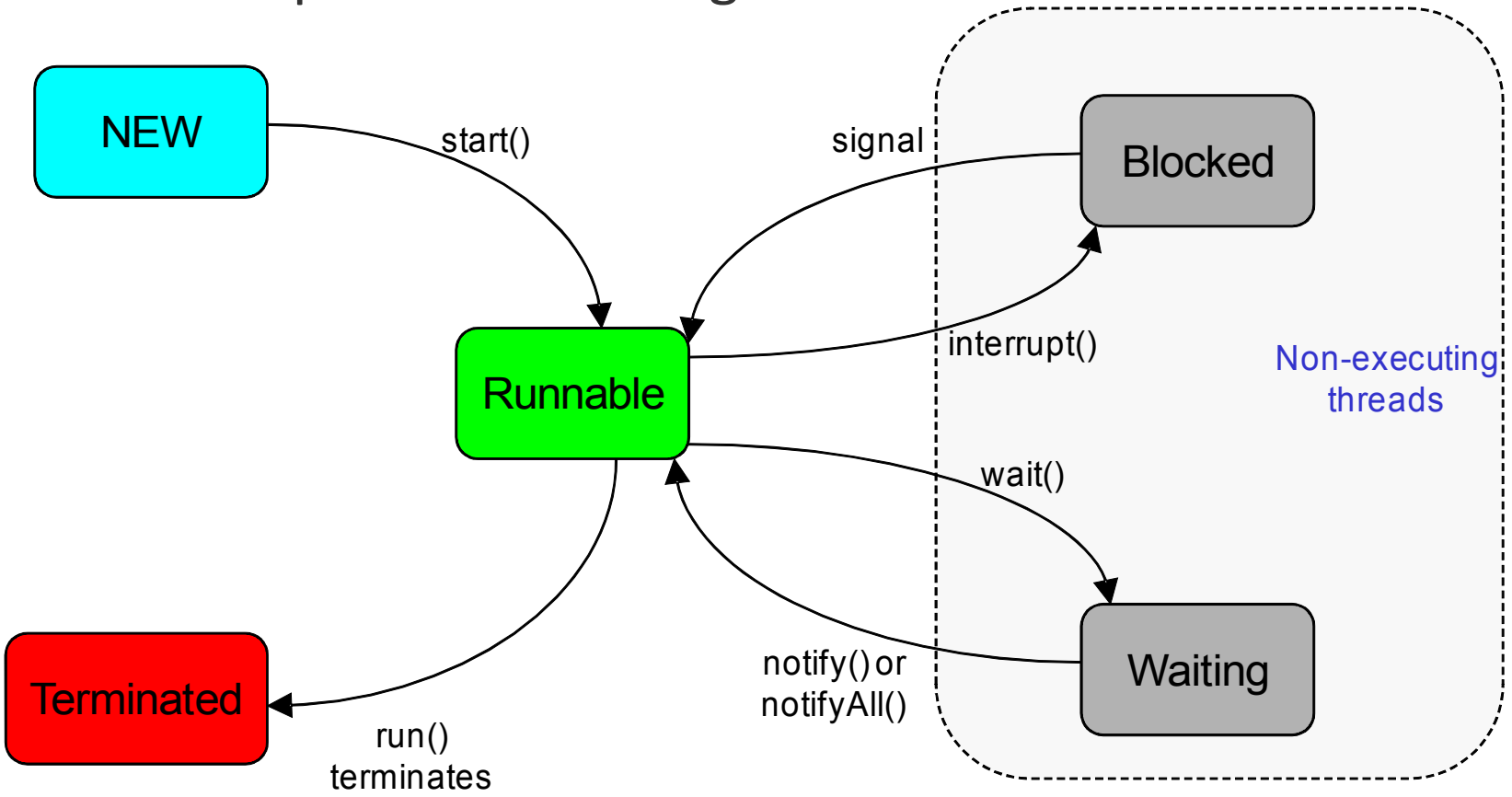
```
Thread hilo = new Thread(new Runnable()
        {
            public void run()
            {
                System.out.println("Empieza el hilo que saluda");
                for (int i = 1; i <= 10; i++)
                {
                    System.out.println("Hola ....");
                    try
                    {
                        Thread.sleep(1000) ;
                    }
                    catch (InterruptedException ex)
                    {
                        ex.printStackTrace();
                    }

                }
            }
        }
    );
    hilo.start();
```

Proyecto Hola Adios otras alternativas

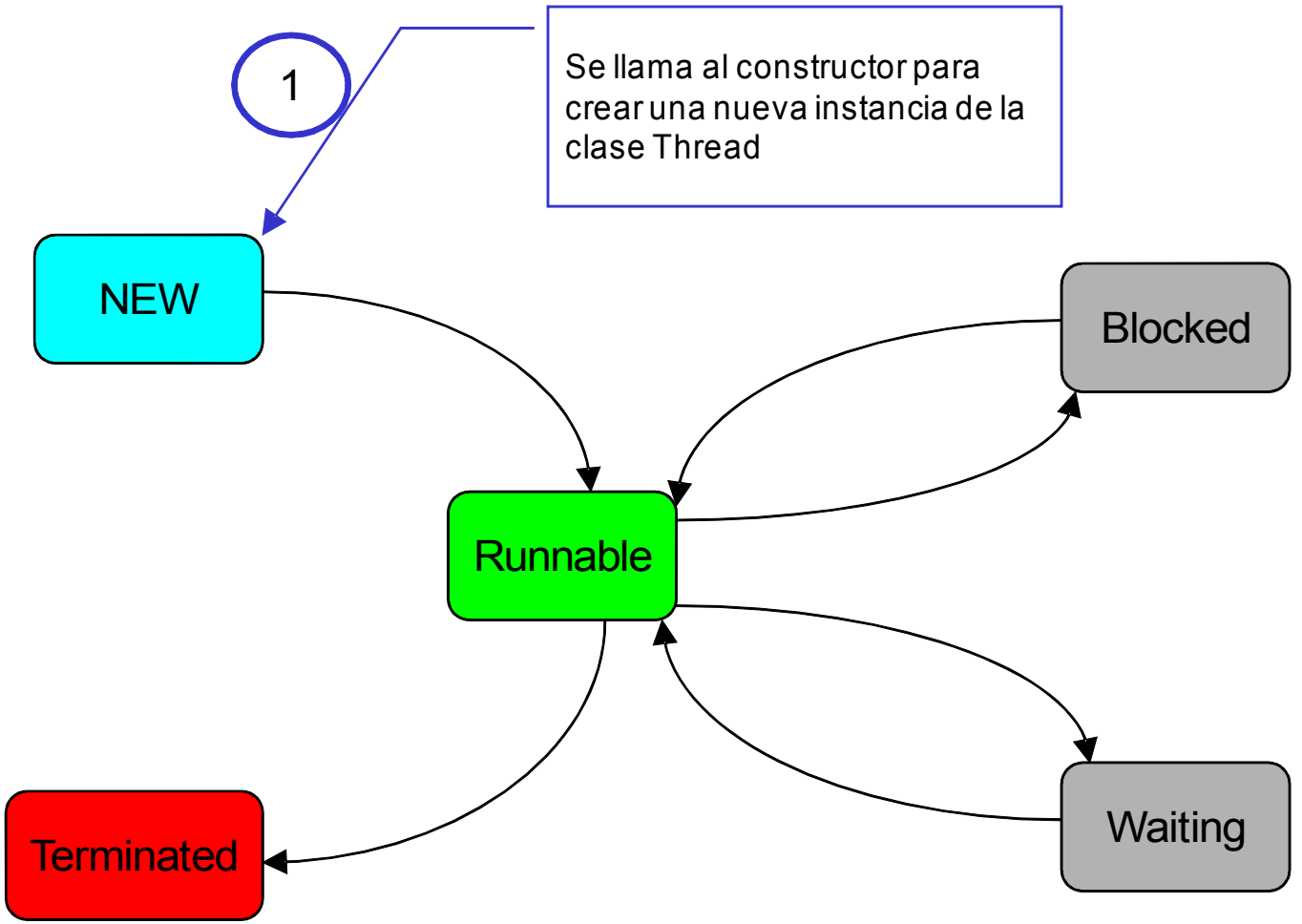
Ejercicios

Ciclo de vida de un hilo

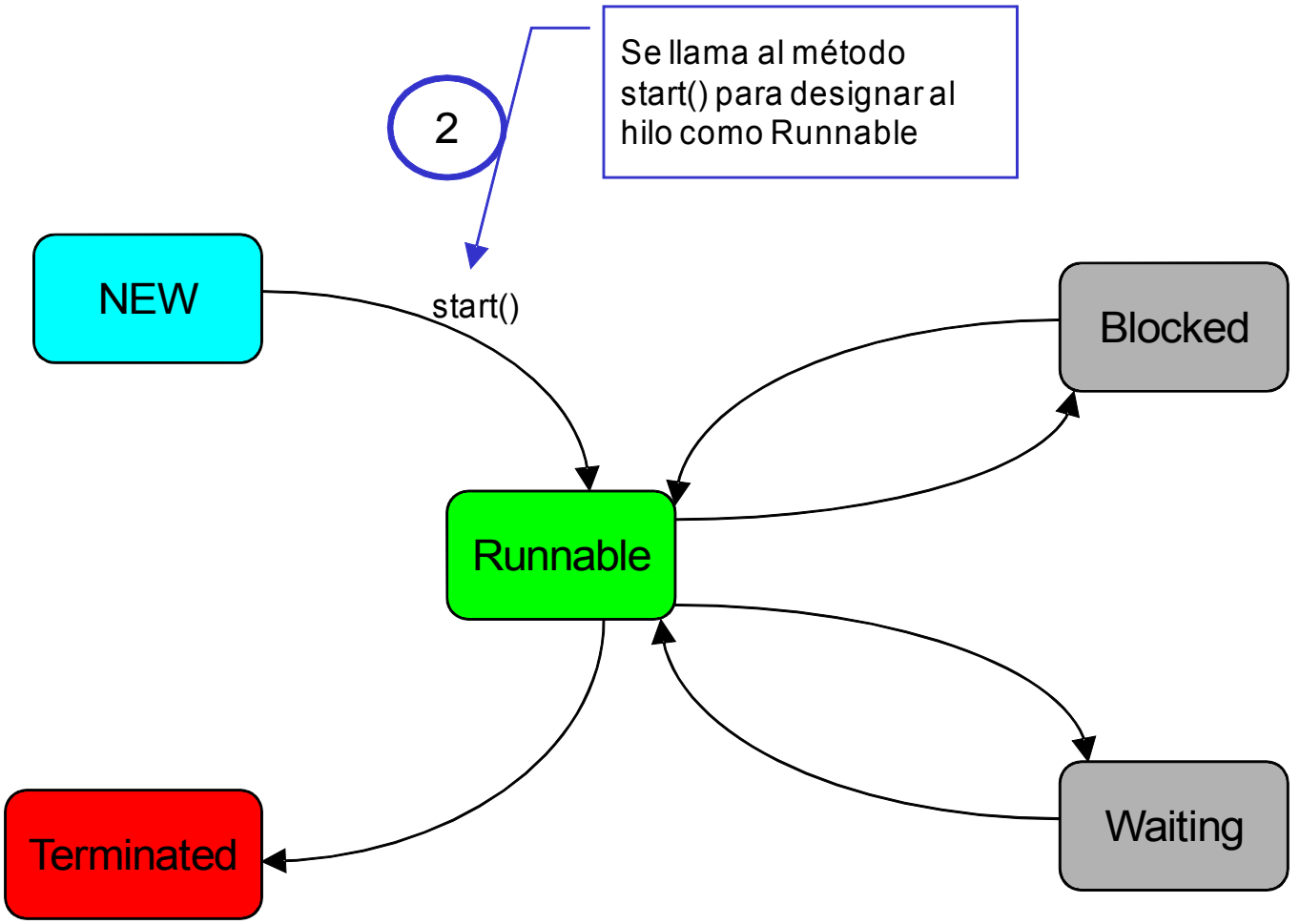
- En un momento determinado un hilo se encuentra en alguno de los estados que muestra la figura



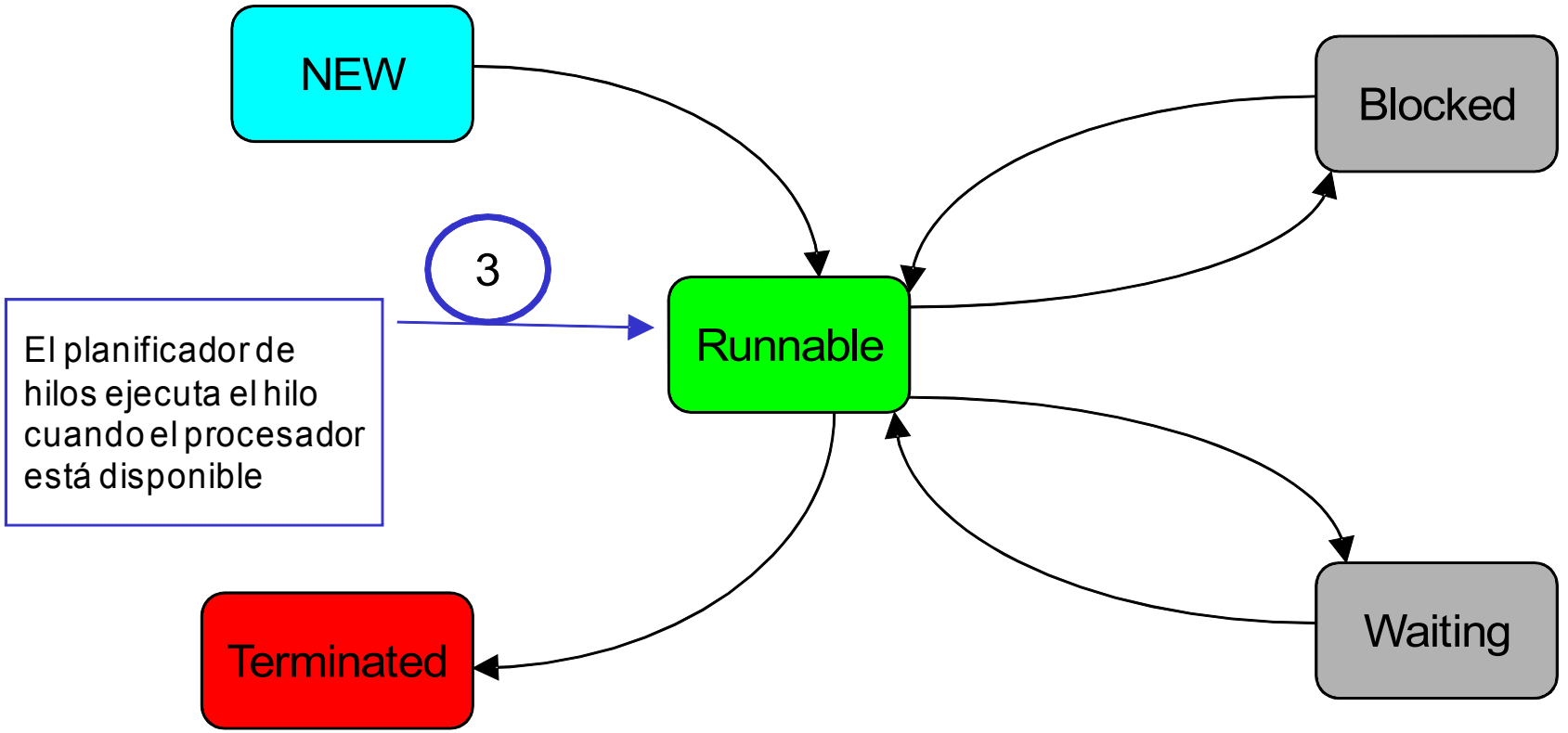
Ciclo de vida de un hilo



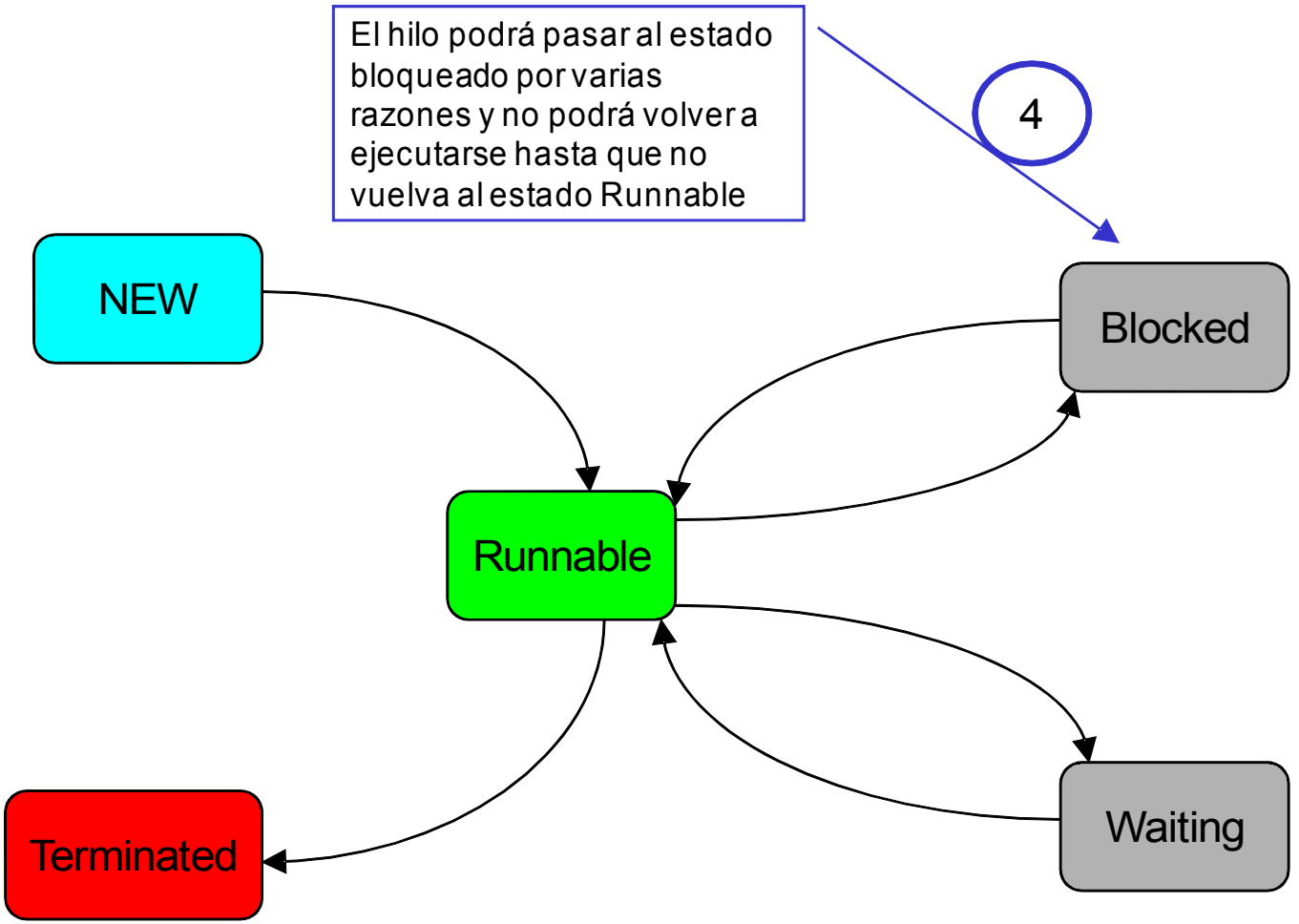
Ciclo de vida de un hilo



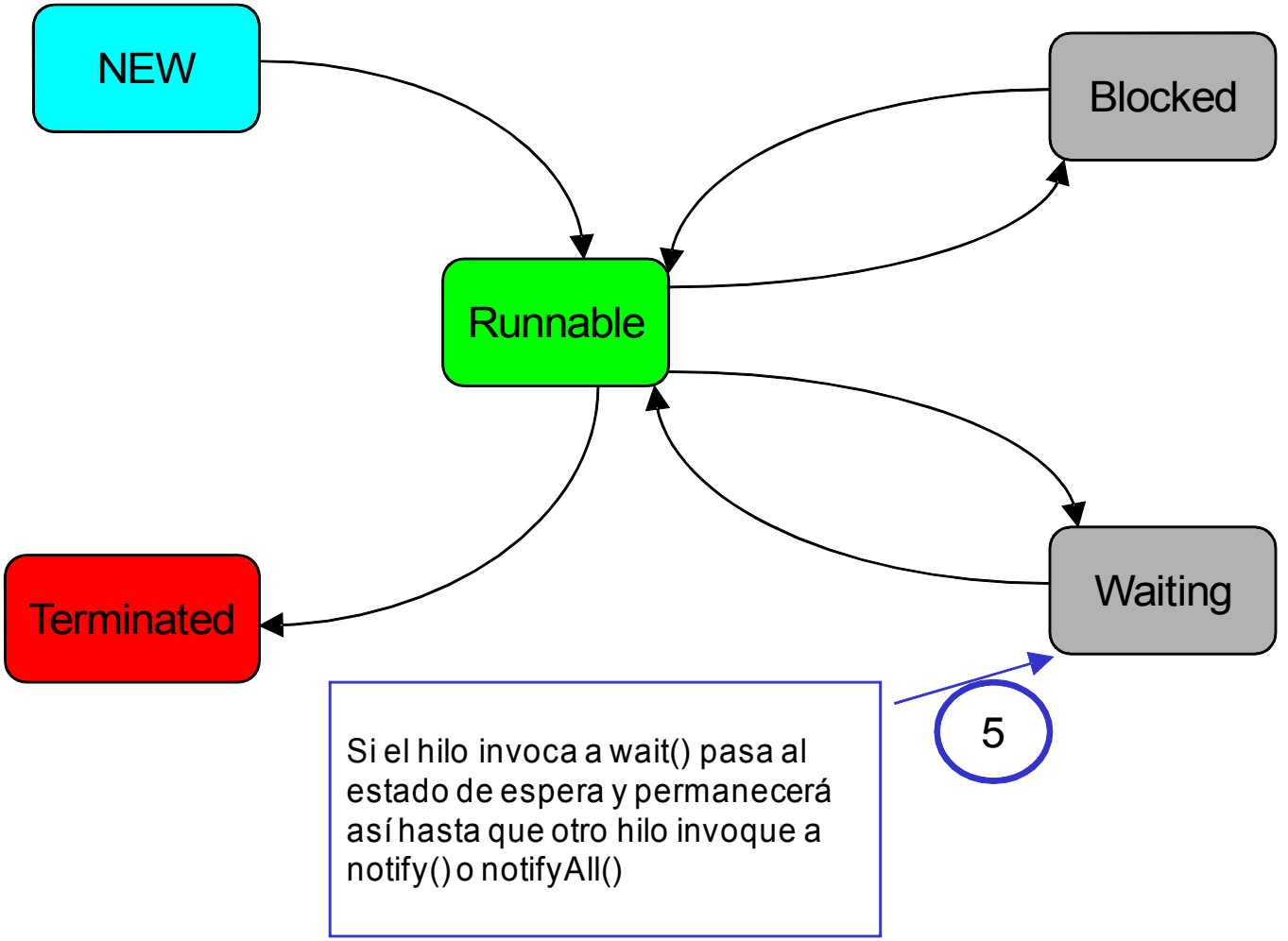
Ciclo de vida de un hilo



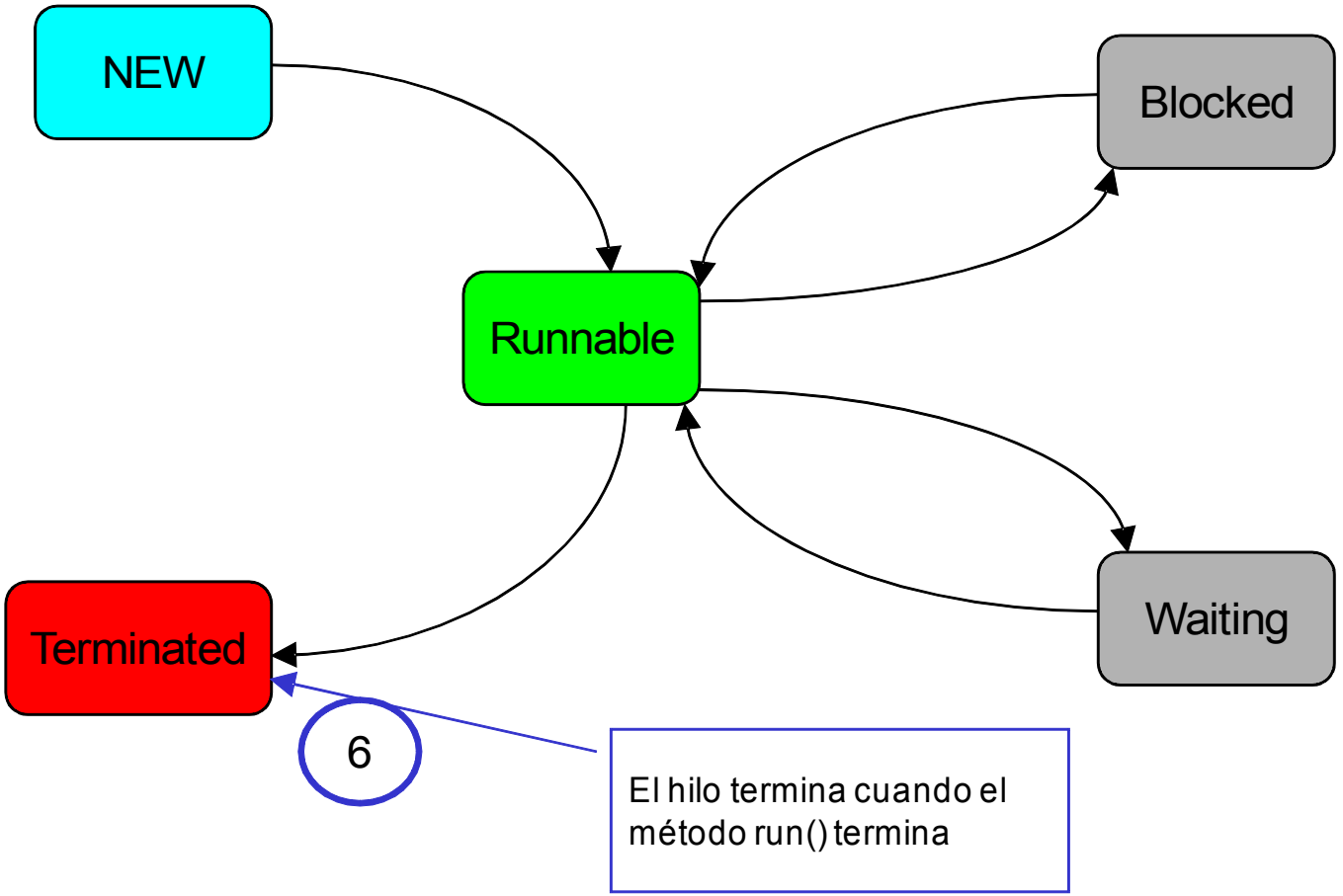
Ciclo de vida de un hilo



Ciclo de vida de un hilo



Ciclo de vida de un hilo

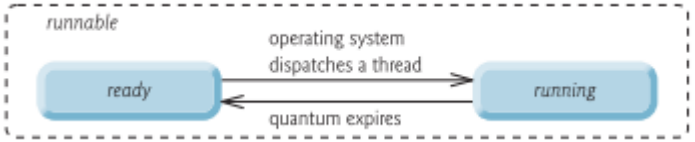


Estados de un hilo

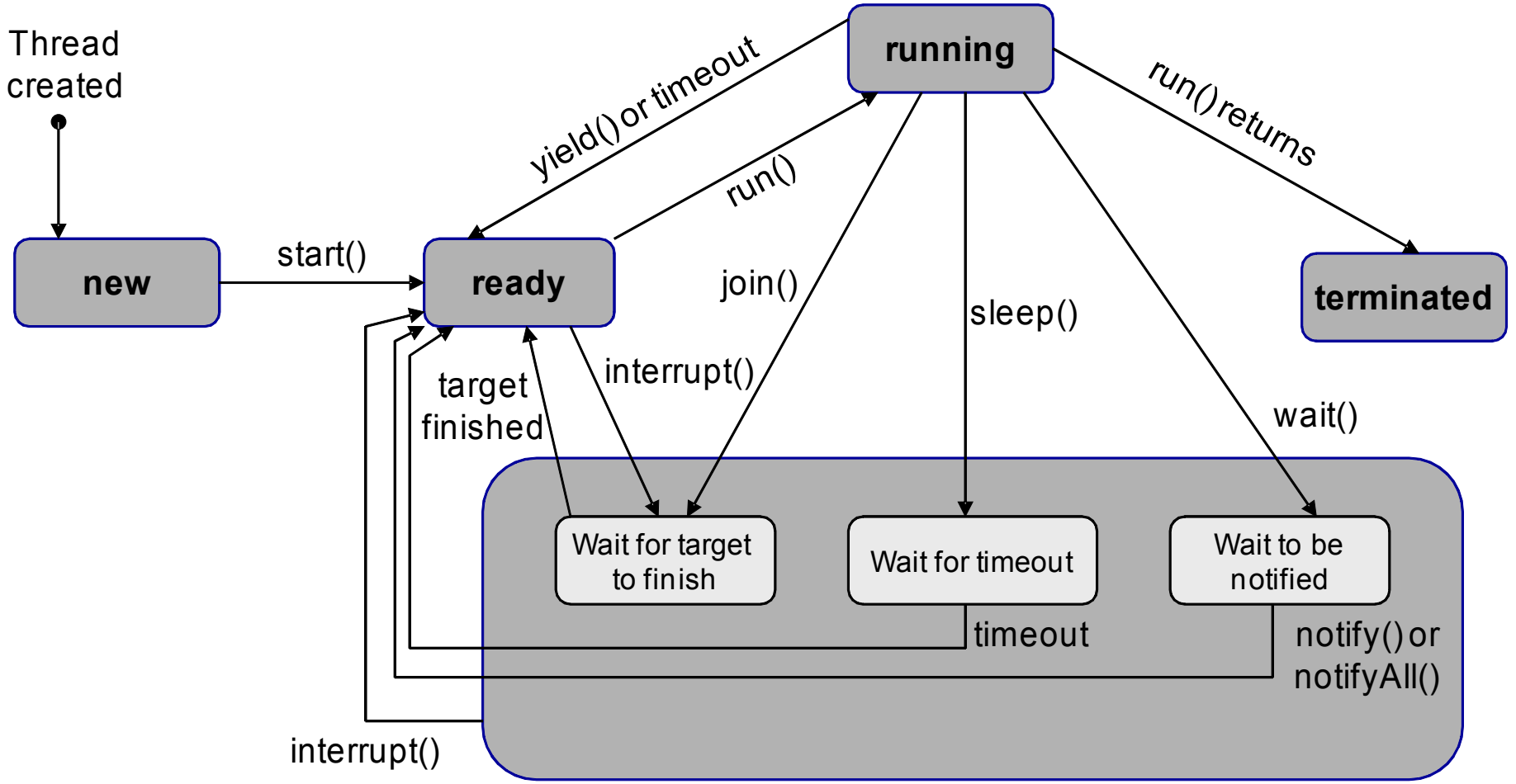
Estado	Descripción
New	El hilo ha sido creado (se ha llamado a su constructor) pero todavía no se ha iniciado
Runnable	Se ha llamado al método start() y el hilo está listo para el *thread scheduler* de la JVM. Esto no significa que empieza a ejecutarse inmediatamente. Un hilo en este estado puede estar ejecutándose o puede estar esperando en una cola de hilos a poder ejecutarse (no hay un estado separado para el hilo que está preparado y puede ejecutarse y el que está realmente ejecutándose). Un hilo que está ejecutándose puede dejar de hacerlo (y seguir en modo Runnable) si acaba su tiempo de CPU o se invoca a yield()
Blocked	El hilo no está en estado Runnable por lo que no puede ejecutarse, el *thread scheduler* no puede elegirlo, es un hilo inactivo. Ocurre cuando se ha llamado al método sleep() o cuando el hilo está esperando a que termine una operación de IO o cuando se invoca a join() o si el hilo pide un *lock* (cerrojo) sobre un objeto que ya lo tiene otro hilo. Cuando la condición cambia (por ejemplo, la operación de IO termina o el tiempo que durmió el hilo ha pasado) el hilo vuelve al estado Runnable

Estados de un hilo

Estado	Descripción
Waiting	El hilo ha invocado a su método wait() para que otros hilos puedan acceder a un objeto. Permanece en estado Waiting hasta que otro hilo llama al método notify() o notifyAll(). Este estado entra en juego cuando los hilos han de coordinar sus actividades
Terminated	El método run() ha finalizado y por tando el hilo termina, deja de estar vivo



Ciclo de vida de un hilo - Otra vista



Prioridad de un hilo - setPriority()

- setPriority(int)
  - cambia la prioridad del hilo a un valor entre 1 y 10
- Tres constantes estáticas en la clase Thread asociadas a prioridades
  - MAX_PRIORITY – prioridad máxima de un hilo – 10
  - MIN_PRIORITY – prioridad mínima de un hilo – 1
  - NORM_PRIORITY – prioridad por defecto – 5
- Por defecto a cada hilo se le da la prioridad del hilo que lo creó
  - si se creó desde el main() su prioridad será 5

Prioridad de un hilo - setPriority()

- El planificador de hilos de Java decide qué hilo ejecutar en base a la prioridad
  - los hilos de más alta prioridad se ejecutan primero
- Si dos o más hilos tienen la misma prioridad el planificador determinará cuál ejecutar
  - se les da a cada uno un tiempo de CPU
- La planificación de hilos depende en última instancia del SO
  - las prioridades no garantizan el orden en que los hilos se ejecutarán, no hay que confiar en ellas

Poner un hilo a dormir - sleep()

- public static void sleep(long n)
  throws InterruptedException
  - duerme el hilo durante al menos n milisegundos
    - el hilo pasa al estado Blocked, transcurrido el tiempo especificado vuelve al estado Runnable, el planificador lo ejecutará cuando lo considere oportuno
  - puede generar la excepción indicada que hay que capturar o propagar
  - efecto
    - hacer una pausa en el programa
    - permite también dar oportunidad a otros hilos de que se ejecuten

Poner un hilo a dormir - sleep()

```
import java.util.Date;
public class DateConSleep
{

   public static void main(String[] args)
   {
      Date d1 = new Date();
      System.out.println("Hilo main se pone a dormir: " + d1.toString());
      try
      {
         Thread.sleep(5000) ; // pausar durante 5 segundos
      }
      catch (InterruptedException ex)
      {
      }
      Date d2 = new Date();
      System.out.println("Hilo main se despierta: " + d2.toString());
   }
}
```

Proyecto Main con sleep mostrar hora

Poner un hilo a dormir - sleep()

Proyecto Contador con sleep

```
public class Contador implements Runnable
{
   private int limite;
   public Contador(int limite)
   {
      this.limite = limite;
   }
   public void run()
   {
      for (int i = 1; i <= limite; i++)
      {
         System.out.println("Valor del contador: " + i);
         try
         {
            Thread.sleep(1000) ; // duerme un segundo entre sucesivas escrituras
         }
         catch (InterruptedException ex)
         {
            ex.printStackTrace();
         }

      }
   }
   public static void main(String[] args)
   {
      System.out.println("Creando un hilo para contar ");
      Contador tarea = new Contador(6);
      new Thread(tarea).start();
   }
}
```

Esperar a que un hilo termine - join()

- public final void join() throws InterruptedException
  - permite que un hilo espere a que otro acabe
    - hay veces en que un hilo debe esperar a que otro acabe para que él pueda continuar (un hilo A prepara un fichero, un hilo B no puede continuar hasta que el fichero no está listo). Forma sencilla de sincronizar hilos
  - t.join()
    - el hilo actual pausa su ejecución (pasa a estado Blocked) hasta que el hilo t termina
    - el código detrás de t.join() no se ejecuta hasta que no termina el hilo t

Esperar a que un hilo termine - join()

```
public class Worker extends Thread
{

   public void run()
   {
      int suma = 0;
      for (int i = 1; i <= 1000; i++)
      {
         suma += i;
         if (i % 10 == 0)
           System.out.println(
               Thread.currentThread().getName() + " " + i);
      }
   }
```

```
public static void main() throws InterruptedException
  {
     Worker uno = new Worker();
     Worker dos = new Worker();
     System.out.println("Empezando ...");
     uno.start();
     dos.start();
     uno.join();
     dos.join();
     System.out.println("Todo realizado ...");

  }
}
```

Proyecto Varios hilos con join

yield() / isAlive()

- public static void yield() - (no se suele usar)
  - pausa temporalmente el hilo lo que permite ceder el control a otro Thread.yield()
  - no se bloquea sino que sigue en estado Runnable (aunque no ejecutándose)
  - el planificador se activa y carga otro hilo que podría ser el mismo
- public final boolean isAlive()
  - devuelve *true* si el hilo está vivo, es decir, si ha sido iniciado con start() y aún no ha muerto

¿Cómo parar un hilo?

- Por ejemplo, si estamos en un bucle infinito (*while (true) ….*), algo habitual en los hilos, cómo parar el hilo de una forma segura?

- No usar Thread.stop() -
  - en desuso y totalmente desaconsejado
- Para parar un hilo de forma segura
  - organizaremos el método run() para poder salir de él
- Dos opciones
  - utilizar en el hilo una variable *stop* definida como *volatile*
  - llamar a interrupt() de la clase Thread

Parar un hilo - con una variable volatile

```
private volatile boolean stop = false;

public void run()
 {
        while (!stop)
        {
         …
        }
}

public void parar()
{
        stop = true;
}
```

volatile – indica que el valor de la variable será modificada por varios hilos

Declarar una variable *volatile* indica:

a) el valor de la variable no se guardará nunca en la caché local del hilo sino en la memoria principal

b) el acceso a la variable se realiza como si estuviese en un bloque sincronizado

Parar un hilo - con una variable volatile

```
public class Tarea implements Runnable
{
   private volatile boolean stop;
   private Thread th;

   public Tarea()
   {
      stop = false;
      th = new Thread(this);
      th.start();
   }

   public void run()
    {
      Thread th = Thread.currentThread();
      while (!stop)
      {
         try
         {
            Thread.sleep(3000);

         }
         catch (Exception exc)
         {

         }
      }
      System.out.println(Thread.currentThread().getName() +
                         " saliendo, se ha parado el hilo.");
    }
    public void parar()
    {
      stop = true;
    }
}
```

Proyecto Parar un hilo con volatile

Parar un hilo con interrupt()

- hay que solicitárselo mediante interrupt()
  - esto pone un flag *boolean* presente en cada hilo a *true*
- el hilo debería verificar periódicamente si lo quieren parar
  - comprobar si el hilo actual ha recibido una interrupción
    - ➔ if (Thread.interrupted())
  - comprobar si otro hilo ha recibido una interrupción
    - ➔ if (t.isInterrupted())
- si el hilo está bloqueado con sleep(), por ej, y es interrumpido
  - el flag no se pone a *true*
  - se genera una excepción InterruptedException

Parar un hilo con interrupt()

```
public void run()
{
     th = Thread.currentThread();
    while (!th.isInterrupted())
    {
        try
        {
            Thread.sleep(3000);
            System.out.println("En hilo " + th.getName());

        }
        catch (Exception exc)
        {
             System.out.println("Hilo interrumpido mientras dormía");
             th.interrupt();
        }
    }
    System.out.println(Thread.currentThread().getName() + " saliendo, se ha
                                                    parado el hilo.");
}
```

```
public void interrumpir()
 {
    th.interrupt();
 }
```

si el hilo se interrumpe mientras duerma el flag no se pone a true, salta la excepción y ahí es donde interrumpimos para poner el flag a true

Proyecto Parar un hilo con interrupt

Demonios

- Hilo **demonio** - public void setDaemon(boolean)
  - llamados **servicios** o **hilos en segundo plano**
  - se ejecutan con prioridad baja
  - proporcionan un servicio básico al programa
  - el *garbage collector* (recolector de basura) es un demonio
  - el hilo main() nunca es un demonio
  - los hilos de usuario por defecto tampoco
    - hilo.setDaemon(true) si queremos que lo sea
  - la JVM termina cuando todos los hilos no demonios terminan, en este momento cualquier demonio también acaba

Ejercicios

Necesidad de sincronizar hilos

- Los ejemplos vistos hasta ahora
  - utilizaban hilos asíncronos, hilos independientes
    - la tarea que realizaba un hilo no dependía de la que realizaba otro
  - los hilos no utilizaban recursos compartidos
  - los hilos no cooperaban para conseguir un objetivo común
    - en algún ejemplo había cooperación sencilla con join()
      - suma de un array con varios hilos, contar palabras de varios ficheros, ... (sincronización muy simple)

Necesidad de sincronizar hilos

- En muchos casos los hilos tienen que interactuar, relacionarse entre ellos
  - compartiendo un recurso (objeto) común
    - varios hilos acceden a una cuenta corriente para hacer ingresos y reintegros
    - varios hilos utilizan un mismo contador para incrementar / decrementar
  - cooperando y coordinando sus actividades
    - un hilo lee datos de un fichero pero previamente otro hilo ha tenido que escribir datos en el fichero
- Es preciso **sincronizar** los hilos para
  - evitar inconsistencias (problemas de concurrencia en general)

Sincronización en Java

- El mecanismo que utiliza Java para la sincronización es el monitor
  - **monitor** – objeto destinado a ser usado sin peligro por más de un hilo de ejecución
- El monitor en Java soporta dos clases de sincronización
  - **exclusión mutua** – permite a múltiples hilos ejecutarse independientemente sobre datos compartidos sin interferirse, se protege secciones críticas de código
    - soportada vía *object locks (cerrojos)*
  - **cooperación** – soportada vía wait() / notify()
    - permite a varios hilos trabajar juntos para conseguir un objetivo común (un hilo debe esperar a que otro termine su trabajo para poder continuar)

Race condition ("condición de carrera")

- Dos o más hilos no sincronizados acceden a un mismo recurso compartido, un objeto y lo actualizan
  - se pueden producir resultados inesperados (actualizaciones que no se reflejan, por ej.)
  - es el problema conocido como ***race condition***
- Race condition
  - el resultado de la ejecución del programa (correcto o incorrecto) depende del orden en que se hayan ejecutado sus hilos

Race condition ("condición de carrera")

- Dos o más hilos no sincronizados acceden a un mismo recurso compartido, un objeto y lo actualizan
  - se pueden producir resultados inesperados (actualizaciones que no se reflejan, por ej.)
  - es el problema conocido como ***race condiction***
- Race condition
  - el resultado de la ejecución del programa (correcto o incorrecto) depende del orden en que se hayan ejecutado sus hilos

Race condition ("condición de carrera")

```
public class Secuencia
 {
        private int siguiente = 0;
        public int getSiguiente()
        {
                siguiente = siguiente + 1;
                return siguiente;
        }
}
```

- Secuencia es el objeto compartido por más de un hilo (en el heap)
- en un momento dado *siguiente* vale 3
- hay dos hilos ejecutando en paralelo getSiguiente()
- Qué valor tendrá *siguiente* después de que ambos hilos hayan completado el método?
- el valor correcto debería ser 5

Race condition ("condición de carrera")

```
public class Secuencia
 {
        private int siguiente = 0;
        public int getSiguiente()
        {
                siguiente = siguiente + 1;
                return siguiente;
        }
}
```

t1	t2
lee 3	
incrementa siguiente	lee 3
	incrementa siguiente
escribe siguiente – 4!!	escribe siguiente – 4!!

El valor correcto después de ejecutar t1 y t2 debería ser 5 pero es 4. Se ha perdido una actualización de uno de los hilos

Race condition

- Qué ha ocurrido?
  - La operación de incremento no era atómica: se divide en pasos (leer – incrementar – escribir)
  - las operaciones atómicas se terminan completamente y no pueden ser interrumpidas por otro hilo
- Problemas con la interferencia de hilos
  - produce resultados incorrectos
  - muy difícil de detectar
- Solución
  - hacer el objeto compartido *thread safe*
  - asegurarnos de que solo un hilo puede ejecutar el método en un momento dado
  - asegurarnos de que el método se ejecuta atómicamente y los resultados están disponibles a otros hilos

Race condition - Solución

- la solución para la interferencia es la exclusión mutua
  - asegurarnos de que solo un hilo está en un momento dado dentro de una sección crítica de código
    - ➔ sección crítica – parte del código que accede a un recurso compartido y que no debería ser accedido por más de un hilo a la vez
- nuestro método getSiguiente() es una sección crítica
  - está permitiendo ahora que más de un hilo cambie a la vez el valor de siguiente
-

```
public synchronized int getSiguiente()
{
        siguiente = siguiente + 1;
        return siguiente;
}
```

Solución – la palabra clave synchronized en la declaración del método – Ahora se garantiza que solo un hilo ejecutará este método en un momento dado

Después de la sincronización

La situación ahora
es una de estas dos

t1	t2
lee 3	
incrementa siguiente	
escribe siguiente – 4!!	
	lee 4
	incrementa siguiente
	escribe siguiente 5!!

t1 empieza primero

t1	t2
	lee 3
	incrementa siguiente
	escribe siguiente – 4!!
lee 4	
incrementa siguiente	
escribe siguiente 5!!	

t2 empieza primero

Sincronización en Java

- Cada objeto en Java tiene asociado un **lock** (*cerrojo, candado*)
  - normalmente este *lock* no se usa
- Cuando un método se declara sincronizado
  - el *lock* se activa
  - un hilo necesita adquirir el *lock* primero, antes de que pueda llamar a métodos sincronizados del objeto compartido
- Si otro hilo quiere llamar a un método sincronizado sobre el mismo objeto
  - la JVM verifica si el *lock* del objeto está disponible
  - si lo está el hilo adquiere el *lock* y ejecuta el método
  - si el *lock* no está disponible (otro hilo está ejecutando el método y tiene ese *lock*) ha de esperar a que el *lock* se libere

Sincronización en Java

- El *lock* es por objeto, no por método sincronizado
  - si un objeto tiene dos métodos sincronizados no puede haber dos hilos ejecutando cualquiera de los métodos sincronizados a la vez
  - sin embargo otros hilos puede ejecutar métodos no sincronizados sobre el objeto

Nuestro ejemplo Secuencia

```
public class SecuenciaThread extends Thread
{
	private Secuencia secuencia = null;
	public SecuenciaThread(Secuencia secuencia)
	{
		this.secuencia = secuencia;
	}
	public void run()
	{
		for (int i = 0; i <= 10; i++)
			System.out.println(secuencia.getSiguiente());
	}
}
```

Proyecto Sincronizacion Secuencia

¿Qué hay que sincronizar?

- Si una clase tiene más de un método que lee / escribe sobre variables compartidas todos esos métodos deberían ser sincronizados

```
public class ContadorSincronizado
{
   private int c = 0;

   public synchronized void incrementar()
   {
      c++;
   }
   public synchronized void decrementar()
   {
      c--;
   }
   public synchronized int getValor()
   {
      return c;
   }
}
```

qué pasaría si getValor() no fuese sincronizado?

Ámbito del lock

- Se puede incrementar el rendimiento limitando el ámbito del *lock* al mínimo
  - grado más fino de sincronización

```
class  X
{
   public synchronized void x1()
   {
          //sentencias
          //…..
          //<comienzo sección crítica>
          c++;
          //<fin sección crítica>
          //más sentencias
          //…….
   }
}
```

el lock se mantiene más tiempo por el hilo, todo el método

```
class X
{
   public void x1()
   {
          //sentencias
          //…..
          synchronized (this)
          {
                 //<comienzo sección crítica>
                 c++;
                 //<fin sección crítica>
          }
          //más sentencias

   }
}
```

el lock se reduce a la sección crítica, se mantiene menos tiempo. This se refiere al objeto actual

Hilos que cooperan - Sincronización vía wait()/notify()

- La sincronización vía wait() / notify() permite que
  - varios hilos trabajen coordinadamente para conseguir un objetivo común
    - ➔ un hilo no puede avanzar en su tarea hasta que otro no haya terminado su trabajo
  - los hilos se comunican entre sí para poder coordinar sus operaciones
- Se implementa a través de monitores y condiciones de sincronización

Monitores

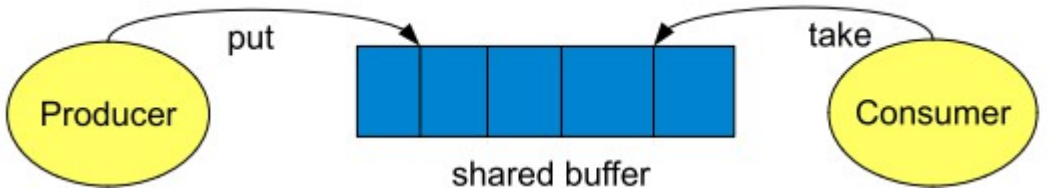
- Características de los lenguajes para la programación concurrente
- Un **monitor** en Java es
  - una clase que encapsula datos (private) y tiene
  - métodos sincronizados (lo que proporciona acceso exclusivo, exclusión mutua)
- Más formalmente
  - *un monitor es un módulo de un programa concurrente que encapsula un recurso compartidos (objeto = datos + sus operaciones) con mecanismos específicos para la sincronización (variables condición) y cuyas operaciones se ejecutan por definición en exclusión mutua*

Monitores y condiciones de sincronización

- Los monitores soportan condiciones de sincronización para asegurar que el acceso a los datos que encapsulan es mutuamente exclusivo
- Las condiciones de sincronización permiten a un monitor bloquear hilos hasta que se cumpla una condición (hay un nuevo valor en un buffer, el buffer ya está vacío, ...)

Problema del Productor Consumidor

- Problema clásico de sincronización en programación concurrente
- Se tienen dos hilos
  - hilo productor e hilo consumidor
    - ambos comparten un recurso común, un buffer de tamaño fijo (por ejemplo)
  - el productor genera datos que pone en el buffer
  - el consumidor toma datos del buffer y los consume



muchos problemas concurrentes
se ajustan
a este modelo

Condiciones de sincronización

- ¿Puede el productor ejecutarse siempre?
  - no lo puede hacer cuando el buffer esté lleno
    - ➔ el productor debe bloquearse hasta que una condición sobre el buffer cambie
- ¿Puede el consumidor ejecutarse siempre?
  - no lo puede hacer cuando el buffer esté vacío
    - ➔ el consumidor debe bloquearse hasta que una condición sobre el buffer cambie
- Es lo que se llama condiciones de sincronización

Implementando el buffer

- Identificamos lo que serán los hilos
  - entidades activas que realizarán las acciones, hilo productor e hilo consumidor, añaden y borran del buffer
- Identificamos lo que será el monitor
  - entidad pasiva que responde a las acciones de los hilos
  - será el buffer compartido sobre el que los hilos añaden y borran
    - ➔ estas acciones serán los métodos del monitor

Implementando el buffer

```
    //put() llamado por el hilo productor
public synchronized void put(Object item)
{
        while (elementos == BUFFER_SIZE)
                Thread.yield();
        elementos++;
        buffer[in] = item;
        in = (in + 1) % BUFFER_SIZE;
}
```

```
//get() llamado por el hilo consumidor
public synchronized Object get()
{
        Object  item;
        while (elementos == 0)
                Thread.yield();
        elementos--;
        item = buffer[out];
        out = (out + 1) % BUFFER_SIZE;
        return item;
}
```

primer intento de implementar las condiciones de sincronización

Implementación del buffer como monitor en Java (solo se muestran los métodos put() / get())

Proyecto Problema del productor Consumidor completo

Abrazo mortal

- put() / take() son sincronizados para que solo un hilo (productor o consumidor) se ejecute en cada momento y evitar el *race condition* sobre *elementos*
- sin embargo, el código anterior puede causar un problema, el abrazo mortal, debido a cómo se ha implementado la condición de sincronización
- asumimos que el *buffer* está lleno y el hilo productor se está ejecutando – adquiere el lock sobre el *buffer* y entra en el método put() pero no puede continuar porque el *buffer* está lleno, así que llama a yield() - ahora el hilo consumidor puede ejecutarse

Abrazo mortal

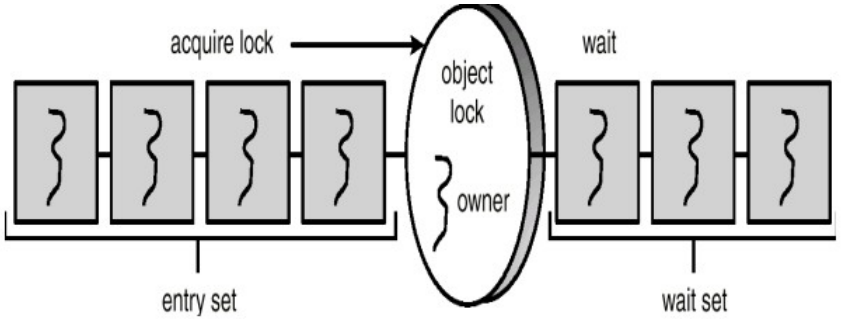
- el hilo consumidor empieza a ejecutarse pero no puede acceder al *buffer* porque el *lock* no ha sido liberado por el productor (lo posee éste todavía)
- el productor está esperando a que el consumidor borre algún elemento del *buffer* y el consumidor está esperando a que el productor libere el *lock* sobre el *buffer*
- ninguno de los dos hilos puede progresar, ambos están en un **abrazo mortal**
- el abrazo mortal aquí está causado por el método yield() que no libera el *lock* sobre *buffer*

Evitando el abrazo mortal – wait() / notify()

- la situación anterior se puede evitar utilizando los métodos wait() / notify() de la clase Object
- cada objeto además del *lock* tiene lo que se denomina un **wait set**
  - conjunto de hilos que está esperando a que alguna condición cambie sobre el objeto / recurso compartido
- cuando un hilo entra en un método sincronizado de un objeto adquiere el *lock* sobre el objeto. Pero el hilo puede que no pueda continuar porque no se cumple una condición
- usando wait() se permite a un hilo liberar el *lock* y esperar hasta que la condición para que pueda continuar su trabajo se cumpla. Como el *lock* está ahora liberado otros hilos lo pueden adquirir y cambiar la condición en el objeto compartido

El método wait()de la clase Object

- cuando un hilo llama a **wait()**
  - el hilo libera el *lock* sobre el objeto
    - ➔ así otros hilos bloqueados pueden ejecutarse
  - el hilo pasa a estado de Waiting
    - ➔ hasta que otro hilo llama a notify() o notifyAll()
  - el hilo se sitúa en el *wait set*



Avisando utilizando notify() / notiyAll()de la clase Object

- el productor llama a put(), ve que el *buffer* está lleno y llama a wait(). Esto libera el *lock*, bloquea al productor y ésta pasa al *wait set* del objeto compartido

- liberar el *lock* permite al consumidor entrar en el método get() y liberar espacio en el *buffer* para el productor

- Cómo avisa ahora el hilo consumidor de que el productor puede ahora entrar de nuevo al método put()?
  - al final de put() y get() , **notify() / notifyAll()** deben ser invocados

Avisando utilizando notify() / notiyAll()de la clase Object

- Cuando un hilo llama a **notify()**
  - se selecciona un hilo arbitrario *t* del *wait set*
  - se mueve *t* al *entry set*
  - *t* pasa a estado Runnable
  - *t* ahora puede competir por el *lock*

- Cuando un hilo llama a **notifyAll() - preferible**
  - se borran todos los hilos del *wait set y*
  - se sitúan en el *entry set* (pasa a Runnable)
  - el planificador decide qué hilo se selecciona

Implementación definitiva del buffer

```
   //put() llamado por el hilo productor
public synchronized void put(Object item)
                    throws InterruptedException
{
          while (elementos == BUFFER_SIZE)
                    wait();
          elementos++;
          buffer[in] = item;
          in = (in + 1) % BUFFER_SIZE;
          notifyAll();
}
```

utilizamos notifyAll() , es mejor cuando hay varios productores y consumidores

```
//get() llamado por el hilo consumidor
public synchronized Object get()
                                  throws
InterruptedException
{
          Object  item;
          while (elementos == 0)
                    wait();
          elementos--;
          item = buffer[out];
          out = (out + 1) % BUFFER_SIZE;
          notifyAll();
          return item;
}
```

Implementación del buffer como monitor en Java (solo se muestran los métodos put() / get())

Resumen wait() / notify()

Método	Descripción
Así se implementan las condiciones de sincronización en Java, con estos métodos de la clase Object	
public final void wait() throws InterruptedException	Espera a ser notificado por otro hilo. El hilo que espera libera el lock asociado con el objeto compartido/monitor
public final void notify()	Despierta a un único hilo que está esperando en el wait set del objeto (lo pasa a estado Runnable)
public final void notifyAll()	Despierta a todos los que están esperando en el wait set del objeto (los pasa a estado Runnable)
Los tres métodos solo pueden usarse dentro de un método sincronizado. Si se llama a uno de estos métodos desde un método no sincronizado se lanza la excepción **IllegalMonitorStateException**	

Significado de thread-safe

- http://en.wikipedia.org/wiki/Thread_safety
  - *Una pieza de código es segura – **thread safe** - si maneja estructuras de datos compartidas de forma que garantice la ejecución segura por varios hilos de ejecución al mismo tiempo*
- Código que se ejecuta de forma segura en un entorno multihilo
- Es importante escribir código seguro y evitar los problemas que pueden producirse con la concurrencia
  - race condition
  - el abrazo mortal (dead lock – interbloqueo o punto muerto)
  - inanición (starvation)
  - interbloqueo activo (livelock)
- No quiere decir que haya que sincronizar todo
  - la sincronización afecta al rendimiento de los programas

Puntos a recordar para escribir código thread safe

- Los objetos inmutables (clase String) son seguros, su estado no puede ser modificado una vez se crean
- Las variables *final* o de solo lectura también son *thread safe* en Java
- Las variables *static* si no se sincronizan pueden dar problemas
- El uso de locking es una de las formas de escribir código seguro en Java
- Las variables locales son seguras
- Las operaciones atómicas en Java son seguras
  - leer un valor int de 32 bits es una operación segura, que no interfiere con otros hilos

Puntos a recordar para escribir código thread safe

- Java tiene clases que son seguras
  - String, StringBuffer, ...
  - Vector, HashTable, ConcurrentHashMap – seguras pero son colecciones en desuso
    - ➔ la clase Collections proporciona métodos estáticos para envolver una colección y obtener su versión sincronizada
      - ✔ pubic static Map synchronizedMap(Map m)

La clase javax.swing.Timer - Animaciones

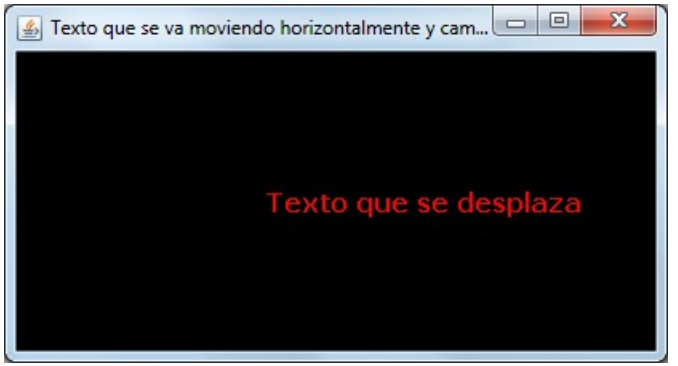
- uno de los usos básicos de los hilos es ejecutar una tarea periódicamente, cada cierto intervalo de tiempo
- hay una clase especializada para ello
  - javax.swing.Timer
  - un objeto Timer genera un evento de tipo ActionEvent a intervalos regulares de tiempo
  - no tiene una representación visual
  - detrás de un Timer subyace el uso de un hilo
  - utilizando *timers* (temporizadores) podemos
    - hacer algo después de un retardo
    - crear animaciones – ilusión de crear movimiento a través de imágenes o figuras

La clase javax.swing.Timer - Animaciones

Método o constructor	Descripción
Timer(int, ActionListener)	Crea un Timer que genera un ActionAvent a intervalos regulares, especificado por el retardo expresado en milisegundos. El evento se manejará por el oyente especificado.
void addActionListener(ActionListener oyente)	Añade un oyente al temporizador
void setDelay(int retardo)	Establece el retardo del temporizador
boolean isRunning()	Devuelve *true* si el timer está funcionando
void start()	Inicia el temporizador causando que se generen eventos de tipo ActionEvent
void stop()	Para el temporizador
void setRepeats(boolean)	Si el parámetro es *false* el evento se genera una sola vez al principio

Ejemplo Timer

```
public class PanelTexto extends JPanel implements
                                                ActionListener
{
    private Timer timer;
    private int x;
    private int y;
    private boolean rojo;;

    /**
     * Constructor de la clase PanelTexto
     */
    public PanelTexto(int ancho, int alto)
    {
        this.setPreferredSize(new Dimension(ancho, alto));

        rojo = true;
        this.x = this.getWidth() / 2;
        this.y = this.getHeight() / 2;
        this.timer = new Timer(100, this);
        timer.start();

    }
```



crear el timer, el panel es el oyente, se generará un evento cada 100 milisegundos

iniciar el temporizador

Proyecto Timer texto que se desplaza y cambia de color

Ejemplo Timer

```
public void paintComponent(Graphics g)
{
    super.paintComponent(g);
    this.setBackground(Color.black);
    if (rojo)
       g.setColor(Color.red);
    else
       g.setColor(Color.white);
    rojo = !rojo;
    g.setFont(new Font("Verdana", Font.BOLD, 16));
    g.drawString("Texto que se desplaza", x, 100);
}
public void actionPerformed(ActionEvent e)
{
    actualizar();
     repaint();

}
```

paintComponent(g) nunca debe invocarse directamente, hay que hacerlo a través de **repaint()** (public void repaint()). Si el panel necesita volver a ser dibujado porque ha habido cambios se lo comunicaremos al sistema llamando a repaint();

código que se ejecuta cada vez que se genera un evento

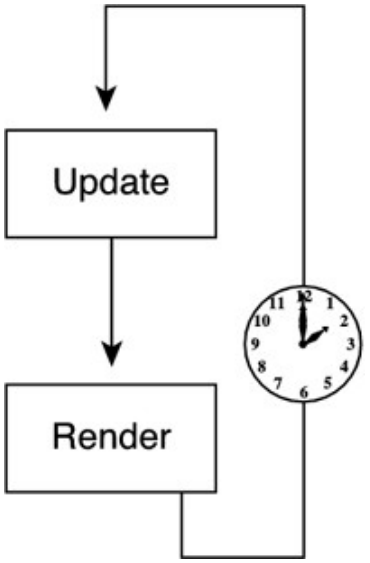
```
private void actualizar()
{
    if (this.x > this.getWidth())
       x = 0;
    x += 5;
}
}
```

Ejercicios

Hilos y animaciones

- Para crear una animación con hilos

  hay que crear un **bucle de animación**
- Este bucle incluye código para
  - actualizar (update) – calcular la nueva posición de la figura o la nueva imagen a mostrar
  - visualizar (render) – dibujar la figura o la imagen
  - esperar un periodo corto de tiempo antes de repetir el proceso (dormir el hilo)
    - ➔ establecemos frecuencia animación y
    - ➔ permitimos que otros hilos se ejecuten



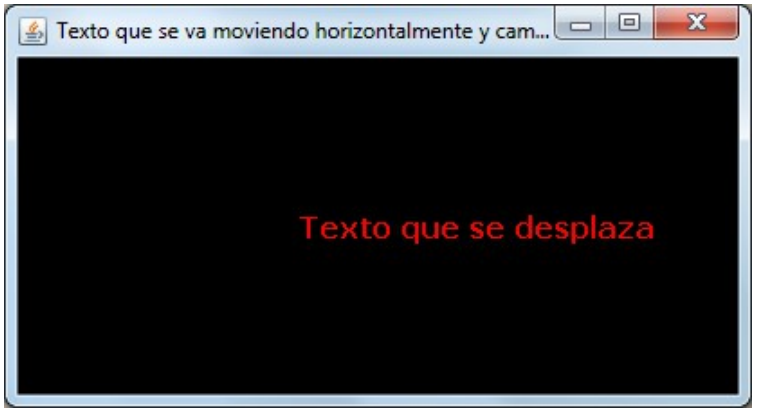
Hilos y animaciones

```
public void run()
{
       while (true)
      {
          actualizar();
          repaint();
          try
           {
              Thread.sleep(DELAY); // pause
          }
           catch(InterruptedException ie)
           {

          }
      }
  }
```

```
public void run()
{
       running = true;
       while (running)
      {
          actualizar();
          repaint();
          try
           {
              Thread.sleep(DELAY); // pause
          }
           catch(InterruptedException ie)
            {
              running = false;
          }
      }
  }
```

Hilos y animaciones - Ejemplo

```
public class PanelTexto extends JPanel implements Runnable
{
   private Thread th;
   private int x;
   private int y;
   private boolean rojo;;

   public PanelTexto(int ancho, int alto)
   {
      this.setPreferredSize(new Dimension(ancho, alto));
      rojo = true;
      this.x = this.getWidth() / 2;
      this.y = this.getHeight() / 2;
      th = new Thread(this);
      th.start();

   }
```



crear e iniciar el hilo

Proyecto Animacion texto que se desplaza con hilos

Hilos y animaciones - Ejemplo

```
public void paintComponent(Graphics g)
 {
    super.paintComponent(g);
    ..........................
 }

 public void run()
 {
    while (true)
    {
       actualizar();  // update posición
       repaint();     // render
       try
       {
          Thread.sleep(100);
       }
       catch (InterruptedException ex) {  }
    }

 }
```

código que se ejecuta concurrentemente

```
private  void actualizar()
  {
     if (this.x > this.getWidth())
        x = 0;
     x += 5;
  }
```

Hilos y animaciones - Otro ejemplo

```
public class PanelPersonaje extends JPanel  implements
                                Runnable, ActionListener
{
    private static final int X = 20;
    private static final int Y = 20;

    private BufferedImage imagenes[];
    private static final int NUM_FRAMES = 5;
    private int imagenActual = 0;
    private int retrasoAnimacion = 50;
    private volatile boolean running;

    private JButton btnIniciar;
    private JButton btnParar;

    private Thread th;
```

Mostrar una secuencia de imágenes. La animación se produce al mostrar cada una de las imágenes un tiempo específico.
La animación se puede parar y reanudar.

Proyecto Animacion Personaje

Hilos y animaciones - Otro ejemplo

```
public PanelPersonaje()
 {
    ..............
    imagenes = new BufferedImage[NUM_FRAMES];
    try
    {
       for (int i = 0; i < NUM_FRAMES; i++)
          imagenes[i] = ImageIO.read(getClass().getResource("images/Stick" + i + ".GIF"));
    }
    catch (IOException ex)
    {
    }
    running = false;

 }
```

cada una de las imágenes que formarán la secuencia animada

para poder cargar imágenes desde un fichero jar o un fichero regular

```
public void paintComponent(Graphics g)
 {
    super.paintComponent(g);
    setBackground(Color.white);
    g.drawImage(imagenes[imagenActual], this.getWidth() / 2,
                                        this.getHeight() / 2, this);

 }
```

se dibuja una imagen en el panel

Hilos y animaciones - Otro ejemplo

```
public void run()
{
        while (running)
        {
            actualizar();
            repaint();
            try
            {
                Thread.sleep(retrasoAnimacion); // pausa
            }
            catch(InterruptedException ie)
            {
                pararAnimacion();
            }
        }
    }
```

```
public void actionPerformed(ActionEvent ev)
    {
        if (ev.getSource() == btnIniciar)
        {
            iniciarAnimacion();
        }
        else if (ev.getSource() == btnParar)
        {

            pararAnimacion();
        }

    }
```

Hilos y animaciones - Otro ejemplo

```
public  void actualizar()
{
    imagenActual = (imagenActual + 1) % NUM_FRAMES;
}

public  void iniciarAnimacion()
{
    if (th == null || !running)
    {
        th = new Thread(this);
        th.start();
        running = true;
    }
}

public  void pararAnimacion()
{
    running = false;
}
```

Ejercicios

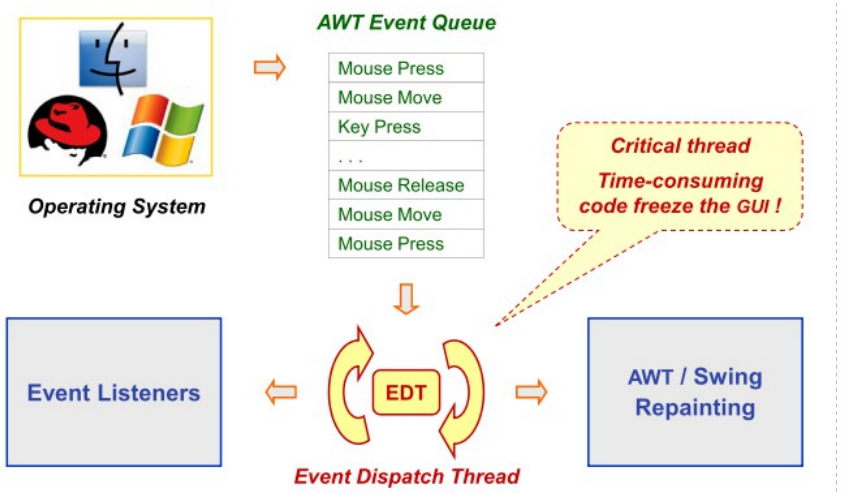
Hilos y Swing - EL EDT (Event Dispatch Thread)

- Cada aplicación Java empieza con el método main() que se ejecuta en el hilo *main*
- En un programa Swing (con una GUI) dentro del método main() se hace lo siguiente
  - llamar al constructor que crea y sitúa los componentes en una ventana (JFrame)
  - llamar al método setVisible() sobre el JFrame
- Al hacer visible la ventana se crea un segundo hilo, el **EDT - event dispatch thread** (*hilo gestor de eventos*)
- El hilo *main* se mantiene ejecutándose hasta que termina el método main()

Hilos y Swing - EL EDT (Event Dispatch Thread)

- El EDT es el hilo que
  - ejecuta las instrucciones de dibujo de los componentes
  - se encarga de la gestión de los eventos asociados a los componentes (ejecuta el código de los gestores de eventos)

Swing Event--Dispatch Model



- La librería Swing no es segura (thread-safe)
  - trabajar con componentes y métodos Swing desde otros hilos diferentes al EDT puede causar conflictos

Hilos y Swing - EL EDT (Event Dispatch Thread)

- Si necesitamos ejecutar código en el *Event Dispatch Thread* podemos utilizar
  - invokeLater()
  - invokeAndWait()
  - ambos son métodos estáticos en javax.swing.SwingUtilities

invokeLater()

- puede ser llamado desde cualquier hilo para ejecutar ciertas instrucciones en el EDT
- el código a ejecutar se incluye en un método run() de un objeto que implementa Runnable
  - invokeLater(Runnable r)
- después de ser llamado retorna inmediatamente sin esperar a que las instrucciones de run() se hayan ejecutado en el EDT

```
public static void main(String[] args)
{
    SwingUtilities.invokeLater(new Runnable()
                        {
                            public void run()
                            {
                                new GuiPersonajeConHilos();
                            }
                        }
                        );
}
```

código a ejecutar en el
Event Dispatch Thread

Hasta ahora nuestras GUI habían sido lanzadas desde el método main – hilo main

invokeAnWait()

- se utiliza de la misma forma que invokeLater()
  - invokeAndWait(Runnable r)
- se diferencia en que después de ser llamado no retorna hasta que el EDT ha ejecutado el código incluido en run()
  - es bloqueante
- preferible usar invokeLater()

```
public static void main(String[] args)
{
   SwingUtilities.invokeAndWait(new Runnable()
                        {
                            public void run()
                            {
                                new GuiPersonajeConHilos();
                            }
                        }
                        );
}
```

código a ejecutar en el Event Dispatch Thread

GUI que no responde (unresponsive GUI)

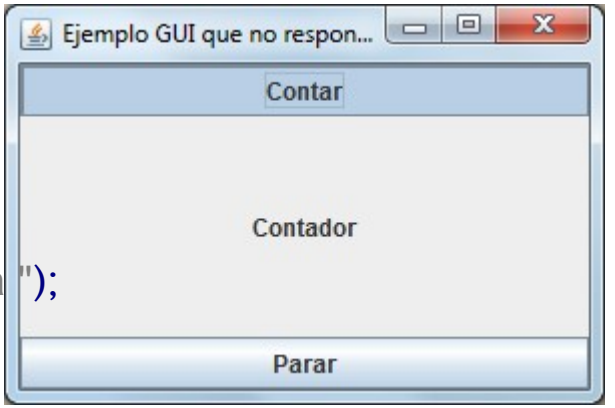
- Hay situaciones cuando se trabaja con GUI en las que es indispensable crear hilos adicionales para realizar ciertas tareas sobre todo si estas son tareas de larga duración
  - si no se hiciese así la gestión de los eventos dentro del EDT quedaría bloqueada y la interfaz no reaccionaría a las acciones del usuario
  - tendríamos lo que se denomina una ***unresponsive GUI*** (se congela el interfaz y no responde)
- A esos hilos se les denomina hilos trabajadores (worker threads o *background threads*)

Las tareas que consumen mucho tiempo o realizan IO deben ser ejecutadas en hilos fuera del EDT

Los componentes Swing solo deberían ser accedidos desde el EDT

GUI que no responde - Ejemplo

```
public class GuiDosBotonesQueNoResponde extends JFrame
{
    ..............................
    private void crearGui()
    {
        this.setTitle("Ejemplo GUI que no responde - se queda congelada ");

         btnContar.addActionListener(new ActionListener()
                                {
                                    public void actionPerformed(ActionEvent ev)
                                    {
                                        conta++;
                                        for (int i = 1; i <= 10000000; i++)
                                        {
                                            lblResultado.setText(conta + "");
                                        }
                                    }
                                }
                                );
```



mientras hace este cálculo el EDT no puede procesar otros eventos, por ejemplo, el click en el botón Parar

Proyecto GuiDosBotones no responde

GUI que sí responde - Ejemplo

```
btnContar.addActionListener(new ActionListener()
        {
            public void actionPerformed(ActionEvent ev)
            {
                new Thread(new Runnable()
                    {
                        public void run()
                        {
                            for (int i = 1; i <= 10000000; i++)
                            {
                                conta++;
                                SwingUtilities.invokeLater(new
                                                        Runnable()
                                    {
                                        public void run()
                                        {
                                            lblResultado.setText(conta + "");
                                        }
                                    }
                                );
                                                                try
                                                                {
                                                                    Thread.sleep(10);
                                                                }
                                                                catch (InterruptedException ex) {}
                                                            }

                                                        } // fin del for
                                                    }).start();
                                                }
                                            }
```

hilo trabajador para la tarea intensiva

Ejercicios