2

PROGRAMACIÓN MULTIHILO

OBJETIVOS DE LA UNIDAD:

- Conocer las técnicas básicas para desarrollar aplicaciones multihilo en Java
- Crear y lanzar varios hilos que compartan información
- Depurar aplicaciones multihilo
- Usar los métodos de sincronización para procesos y subprocesos
- Compartir información entre los hilos de un proceso
- Aprender acerca de los problemas de acceso a memoria compartida
- Usar diferentes técnicas de programación para sincronizar la ejecución de los threads.

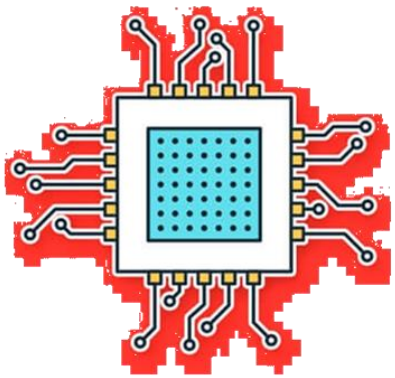
2. Programación Multihilo

INTRODUCCIÓN

Los hilos son elementos cruciales en el mundo de la programación concurrente, ya que permiten que un programa **realice múltiples tareas** de forma simultánea **dentro de un mismo proceso**.

La ejecución de un proceso **comienza con un único hilo**, pero se pueden crear más sobre la marcha. Los distintos hilos de un mismo proceso comparten:

- El **espacio de memoria** asignado al proceso
- La información de **acceso a ficheros** (incluyendo stdin, stdout y stderr).



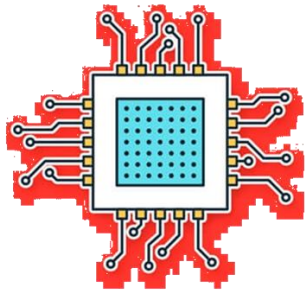
2. Programación Multihilo

INTRODUCCIÓN

Vamos a utilizar los **threads** para realizar **programación concurrente** dentro de un proceso. Aunque los hilos se ejecutan en el contexto de un proceso, cada uno tiene su:

- **TCB (Tread Control Block)** que es sensiblemente más pequeño que el PCB (Process Control Block) porque entre los hilos comparten gran parte de ese PCB.

Por eso veremos que a los hilos también se le llama **lightweight processes (procesos ligeros)** y por tanto los cambios de contexto en el procesador son mucho menos costosos para los hilos que para los procesos.



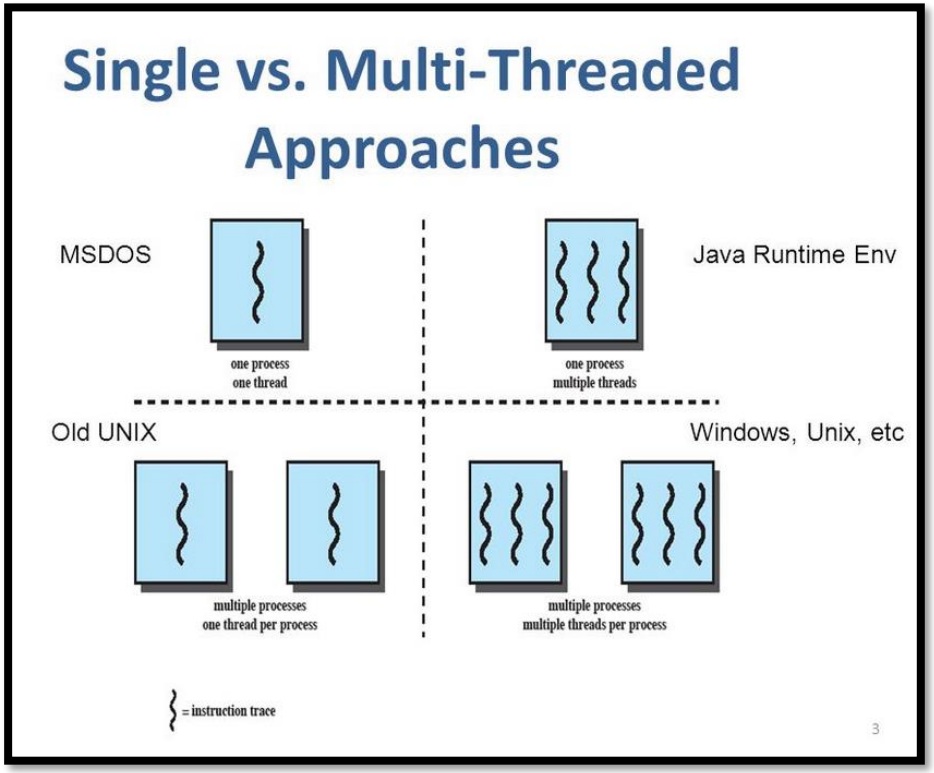
2.1 Clases Java para la gestión de hilos

2.1.1. Funcionamiento de los hilos en Java

Cuando un **programa Java se lanza (se convierte en un proceso)** empieza a ejecutarse por su **método main()** que lo ejecuta el thread principal (main), un hilo especial creado por la Java VM para ejecutar la aplicación.

Desde un proceso se pueden **crear e iniciar tantos threads como necesites**.

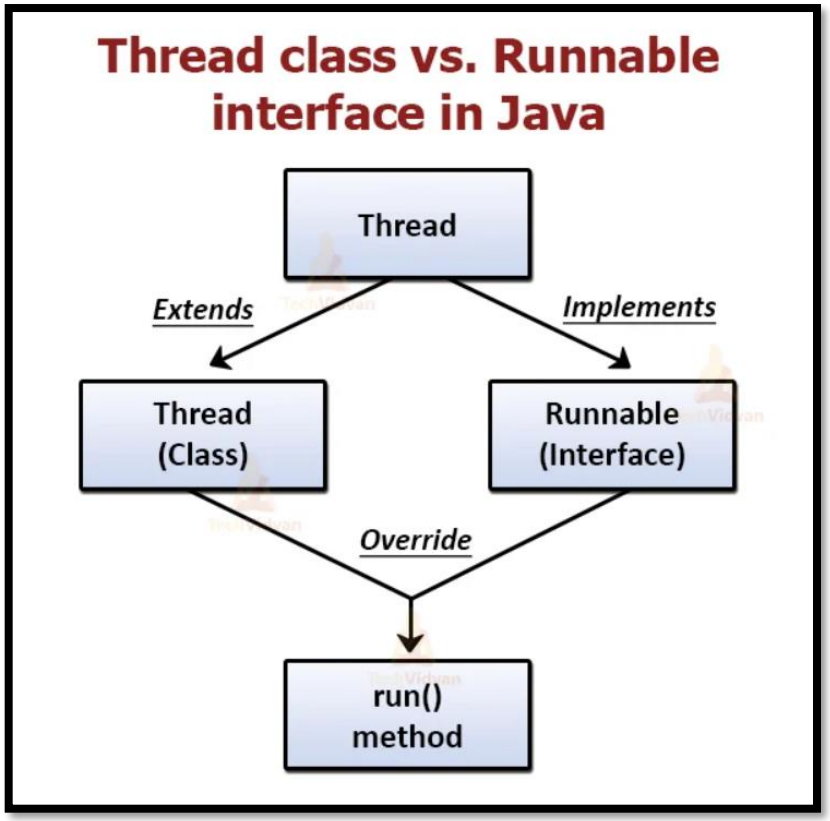
Estos hilos ejecutarán partes del código de la **aplicación en paralelo con el thread principal.**



2.1 Clases Java para la gestión de hilos

2.1.2. Formas de trabajar con Threads

En Java podemos trabajar de **dos maneras** en el manejo de Threads y que se muestra en el siguiente gráfico:



- **Implementar** la **interfaz Runnable**
  Se tiene que implementar el método run() de la interfaz.

- **Extender la clase Thread**
  Se tiene que sobreescribir el método run() de la clase Thread.

La funcionalidad del hilo se programa en el método run

2.1 Clases Java para la gestión de hilos

2.1.3. Ejemplo interfaz Runnable

En el siguiente código se muestra un **ejemplo básico** para implementar la interfaz y crear un hilo.

```
1   public class Principal
2   {
3       public static void main(String[] args)
4           {
5               Hilo hi=new Hilo();
6               Thread hilo=new Thread(hi);
7               hilo.start();
8
9               //Ejecución del hilo padre
10              for(int i=0;i<10;i++)
11              {
12                  System.out.println("Hilo padre");
13              }
14          }
15  }
16
```

Comienza el hilo

Código del método run

```
public class Hilo implements Runnable
{
    @Override
    public void run()
    {
        // Ejecución del hilo hijo.
        for(int i=0;i<10;i++)
        {
            System.out.println("Hilo hijo");
        }
    }
}
```

2.1 Clases Java para la gestión de hilos

2.1.4. Thread subclass

Además de implementando la interfaz Runnable, la segunda forma que tenemos de indicar a un thread el **código a ejecutar es creando una subclase de java.lang.Thread y sobrescribiendo el método run()**.

La clase Thread implementa de **forma implícita la interfaz Runnable**. Al igual que con Runnable, el método run() contiene el código que ejecutará un thread cuando se llame al **método start().**

La especificación de la clase Thread **(java.Lang.Thread)** la puedes consultar **aquí**

Module java.base
Package java.lang

Class Thread

java.lang.Object
    java.lang.Thread

2.1 Clases Java para la gestión de hilos

2.1.4. Ejemplo Clase que hereda de Thread

Vamos a ver un ejemplo de creación de una clase que **herede de Thread**

```
1   public class Principal
2   {
3       public static void main(String[] args)
4       {
5           Hilo hilo=new Hilo();
6           hilo.start();
7
8           //Ejecución del hilo padre
9           for(int i=0;i<10;i++)
10          {
11              System.out.println("Hilo padre");
12          }
13      }
14  }
```

Comienza el hilo

Código del método run

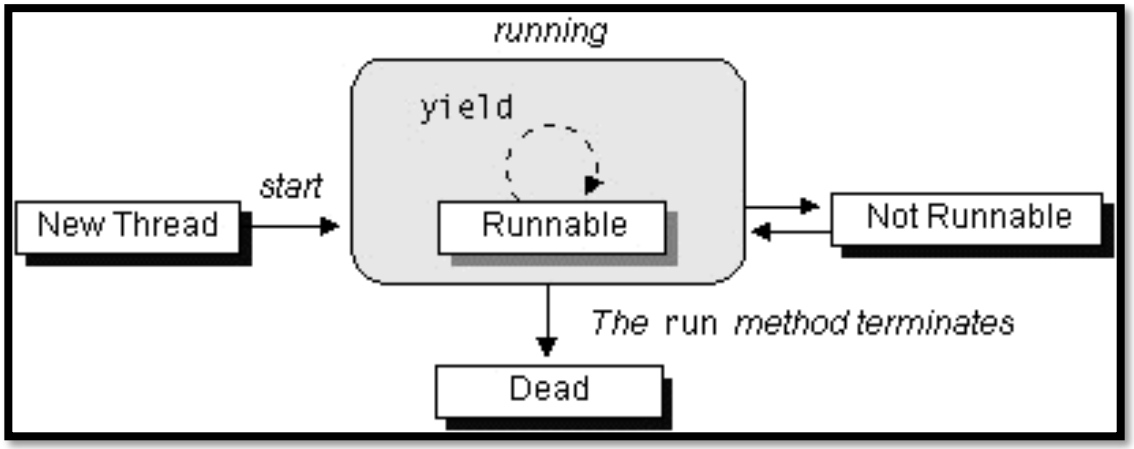
```
1   public class Hilo extends Thread
2   {
3       @Override
4       public void run()
5       {
6           // Ejecución del hilo hijo.
7           for(int i=0;i<10;i++)
8           {
9               System.out.println("Hilo hijo");
10          }
11      }
12  }
13
```

2.1 Clases Java para la gestión de hilos

2.1.5. Método start() y método run()

La llamada al **método start() devuelve el control al thread principal** en cuanto el hilo asociado se inicia.

El método **run() se ejecutará en un hilo de ejecución diferente**, posiblemente por un procesador diferente, entrando en la cola de procesos para competir por las unidades de procesamiento del sistema.



Ciclo de vida de un thread

2.1 Clases Java para la gestión de hilos

2.1.6. Ejemplo de Thread subclass

```java
public class LanzaHilo extends Thread {
    private int countDown = 10;
    private static int taskCount = 0;
    private final int id = taskCount;

    public LanzaHilo()
    {
        taskCount++;
    }

    public LanzaHilo(int countDown)
    {
        taskCount++;
        this.countDown = countDown;
    }

    @Override
    public void run() {
        while (countDown > 0) {
            System.out.println("#" + id + " (" + countDown + ")" );
            countDown--;
        }
        System.out.println("Lanzamiento (" + id + ")");
    }

    public static void main(String[] args) {
        LanzaHilo launch = new LanzaHilo(5);
        LanzaHilo launch1 = new LanzaHilo(3);
        launch.start();
        launch1.start();
        System.out.println("Comienza la cuenta atrás!");
    }
}
```

Desde el momento en que se hace el start() el control **vuelve al hilo principal (main)** que continúa ejecutando las líneas de código que hay en el método main.

La creación de un hilo, aunque menos que la de un proceso, también tiene un coste de recursos y temporal, por lo que el hilo tarda unos instantes en empezar a ejecutarse.

Por eso **el hilo principal tiene tiempo de ejecutar** la siguiente instrucción y mostrar el mensaje.

Los hilos se ejecutan de manera concurrente y no hay dos salidas iguales.

2. Hilos

ACTIVIDAD

Enunciado

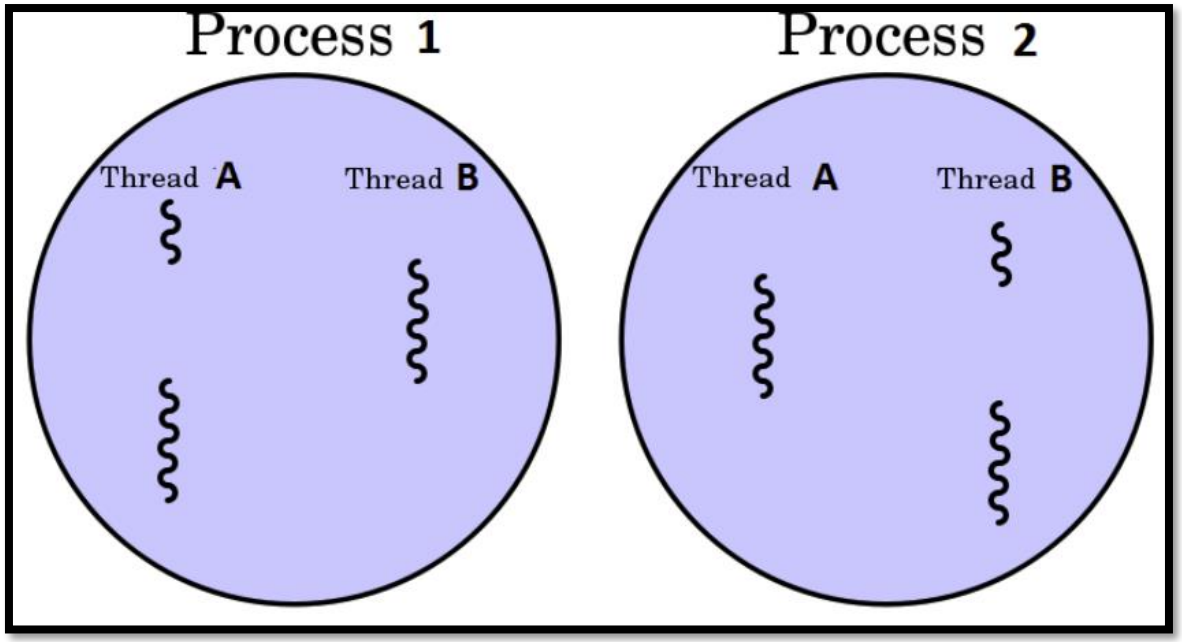
Realiza las siguientes tareas

- Divide el programa anterior para que el código lo tengamos en dos clases: La clase principal y la clase del hilo.
- Una vez dividido el código en dos clases, codifica el programa anterior haciendo uso de la implementación de la interfaz Runnable.
- Crea nuevos hilos.
- Investiga como puedo mostrar la salida de cada uno de los hilo en diferentes colores.

2.1 Clases Java para la gestión de hilos

2.1.7. ¿Cuándo termina un proceso?

Cuando un **proceso tiene mas hilos**, la norma es que el proceso **no finaliza su ejecución hasta que el último de los hilos haya terminado.** Así que podemos encontrarnos, como en el ejemplo de la cuenta atrás, el main principal acaba y el proceso sigue en ejecución, porque los hilos no han acabado.



2.1 Clases Java para la gestión de hilos

2.1.8. Métodos de la clase java.lang.Thread

Si miramos a la definición de **la clase Thread** veremos que tiene muchos métodos. Debemos tener cuidado ya que algunos de estos métodos como stop(), suspend(), resume() and destroy() han sido marcados como **obsoletos(deprecated).**

Method	Description
start()	HAce que un nuevo thread ejecute el código del método run()
boolean isAlive()	Comprueba si un thread está vivo o no
sleep(long ms)	Cambia el estado del thread a bloqueado durante los ms indicados
run()	Es el código que el thread ejecuta. Es llamado por el método start(). Representa el ciclo de vida de un thread.
String toString()	Devuelve una representación legible de un thread [nombre, priority, nombre_del_grupo
long getId()	Devuelve el identificador del thread (es un id asignado por el proceso)

2.1 Clases Java para la gestión de hilos

2.1.8. Métodos de la clase java.lang.Thread

void yield()	Hace que el hilo deje de ejecutarse en el momento volviendo a la cola y permitiendo que se ejecuten otros hilos
void join()	Se llama desde otro thread y hace que el thread que lo invoca se bloquee hasta que el thread termine. Es parecido a p.waitFor() para los procesos
String getName()	Obtiene el nombre del thread
String setName(String name)	Cambia el nombre del thread
int getPriority()	Obtiene la prioridad del thread
setPriority(int p)	Modifica la prioridad del thread

2.1 Clases Java para la gestión de hilos

2.1.8. Métodos de la clase java.lang.Thread

void interrupt()	Interrumpe la ejecución del thread provocando que salte una excepción de tipo InterruptedException
boolean interrupted()	Comprueba si un thread ha sido interrumpido
Thread.currentThread()	Método estático de la clase Thread que devuelve una referencia al hilo que está ejecutando el código
boolean isDaemon()	Comprueba si un hilo es un servicio/demonio. Un proceso/hilo de baja prioridad que se ejecuta de forma independiente de su proceso padre. Un proceso puede finalizar aunque un hilo *daemon* esté todavía ejecutándose.
setDaemon(boolean on)	Convierte un hilo en un demonio/servicio. Por defecto todos los hilos se crean como hilos de usuario.
int activeCount()	Devuelve el número de hilos pertenecientes a un grupo que siguen activos.
Thread.State getState()	Devuelve el estado actual del hilo. Los posibles valores son NEW, RUNNABLE, BLOCKED, WAITING, TIMED_WAITING or TERMINATED.

2.1 Clases Java para la gestión de hilos

2.1.8. Métodos de la clase java.lang.Thread

La clase Thread también tiene **unos 9 constructores**, la mayoría de ellos están duplicados permitiendo recibir un objeto Runnable como parámetro

Constructores de la clase Thread
Thread()
Thread(Runnable target)
Thread(String name)
Thread(ThreadGroup group, String name)
Thread(Runnable target, String name)
Thread(ThreadGroup group, Runnable target)
Thread(ThreadGroup group, Runnable target, String name)
Thread(ThreadGroup group, Runnable target, String name, long stackSize)
Thread(ThreadGroup group, Runnable target, String name, long stackSize, boolean inheritThreadLocals)

2.1 Clases Java para la gestión de hilos

2.1.8. Ejemplo práctico de uso de todos estos métodos.

```
1   public class EjemplosMetodos extends Thread {
2
3       EjemplosMetodos (ThreadGroup group, String name) {
4       // Call to parent class constructor with group and thread name
5       super(group, name);
6       }
7
8       @Override
9       public void run() {
10          String threadName = Thread.currentThread().getName();
11          System.out.println("["+threadName+"] " + "Inside the thread");
12          System.out.println("["+threadName+"] " + "Priority: "
13            + Thread.currentThread().getPriority());
14          Thread.yield();
15          System.out.println("["+threadName+"] " + "Id: "
16            + Thread.currentThread().getId());
17          System.out.println("["+threadName+"] " + "ThreadGroup: "
18            + Thread.currentThread().getThreadGroup().getName());
19          System.out.println("["+threadName+"] " + "ThreadGroup count: "
20            + Thread.currentThread().getThreadGroup().activeCount());
21      }
22
```

2.1 Clases Java para la gestión de hilos

2.1.8. Ejemplo práctico de uso de todos estos métodos.

```
23      public static void main(String[] args) {
24          // main thread
25          Thread.currentThread().setName("Main");
26          System.out.println(Thread.currentThread().getName());
27          System.out.println(Thread.currentThread().toString());
28
29          ThreadGroup even = new ThreadGroup("Even threads");
30          ThreadGroup odd = new ThreadGroup("Odd threads");
31
32          Thread localThread = null;
33          for (int i=0; i<3; i++) {
34              localThread = new EjemplosMetodos((i%2==0)?even:odd, "Thread"+i);
35              localThread.setPriority(i+1);
36              localThread.start();
37          }
38
39          try {
                localThread.join(); // --> Will wait until last thread ends
41                                  // like a waitFor() for processes
42          } catch (InterruptedException ex) {
                ex.printStackTrace();
44              System.err.println("The main thread was interrupted while waiting for "
                  + localThread.toString() + "to finish");
46          }
47          System.out.println("Main thread ending");
48      }
49
50  }
```

2.1 Clases Java para la gestión de hilos

2.1.8. Ejemplo práctico de uso de todos estos métodos.

En el ejemplo anterior podemos ver cómo tenemos que ayudarnos el **método estático Thread.currentThread()** para saber qué hilo está ejecutándose en cada momento, ya que hay muchos hilos ejecutando el mismo código al mismo tiempo.

Hemos creado **una única clase para los hilos y para el hilo principal**. **No debería ser una práctica común** más allá de los ejemplos. Es mejor separar el código del objeto que hereda de Thread o que implementa Runnable en una clase aparte.

También es importante hacer ver que **la clase Thread (o Runnable) puede tener sus propios constructores, propiedades y métodos**, más allá del método run que están obligadas a sobrescribir. También puede invocar a los constructores de la superclase haciendo uso de super().

2.1 Clases Java para la gestión de hilos

2.1.9. SIMULACIÓN DE SISTEMAS REALES

Simulación de sistema reales

Este es un método que vamos a utilizar exhaustivamente en las actividades para simular períodos de tiempo y acelerar las simulaciones.

Por ejemplo, podemos hacer un ajuste para que cada hora *real* se reduzca a un segundo. De esta forma podremos simular un día completo en tan solo 24 segundos.

También es interesante su uso para utilizar períodos de tiempo aleatorios en la ejecución de cada hilo, permitiendo así una simulación realista de los eventos en un sistema real.

2.1 Clases Java para la gestión de hilos

2.1.10. GENERACIÓN DE NÚMEROS ALEATORIOS

En Java podemos generar **números aleatorios** en el rango de los enteros, long, float y double. Tenemos tres métodos:

- **Metodo 1:** Usando la clase Random
- **Metodo 2:** Usando Math.Random
- **Metodo 3:** Usar ThreadLocalRandom

ACTIVIDAD

Crea un programa donde se muestre un **numero aleatorio entre 0 y 9**, usando los tres métodos.

2.1 Clases Java para la gestión de hilos

2.1.11. PAUSAR UN HILO

Un thread puede pausar su propia ejecución llamando al método estático **Thread.sleep().** El método sleep() recibe como parámetro el **número de milisegundos** que quiere estar pausado antes de volver a ponerse como listo para ejecución.

No es un método 100% preciso (menos aun si utilizamos la versión que recibe ms y ns), pero aún así es bastante preciso. A continuación tenemos un **ejemplo de un thread** que se pausa durante 3 segundos (3000ms) llamando al método sleep():

```
try {
    Thread.sleep(3000L);
} catch (InterruptedException e) {
    e.printStackTrace();
}
```

2.1 Clases Java para la gestión de hilos

2.1.12 GESTIÓN DE LA PRIORIDAD DE LOS HILOS

Los hilos **heredan la prioridad** del padre en Java, pero este valor puede ser cambiado con el método **setPriority() y con getPriority()** podemos saber la prioridad de un hilo.

El valor de la prioridad **varía entre 1 y 10**. Cuanto más alto es el valor, mayor es la prioridad. La clase Thread define las siguientes constantes:

- **MIN_PRIORITY** (valor 1)
- **MAX_PRIORITY** (valor 10)
- **NORM_PRIORITY** (valor 5).

El **planificador** elige el hilo en función de su prioridad. Si dos hilos tienen la misma prioridad realiza un **round-robin**, es decir de forma cíclica va alternando los hilos.

2.1 Clases Java para la gestión de hilos

2.1.13 GESTIÓN DE LA PRIORIDAD DE LOS HILOS

El hilo de **mayor prioridad** seguirá funcionando hasta que ceda el control. El control lo puede ceder en los siguientes casos:

- Cede el control llamando al **método yield().**
- **Deja de ser ejecutable** (por muerte o por bloqueo)
- Aparece un **hilo de mayor prioridad**, por ejemplo si se encontraba en estado dormido por una operación de E/S o bien es desbloqueado por otro con los métodos **notifyAll() / notify().**

2.1 Clases Java para la gestión de hilos

2.1.13 EJEMPLO PRIORIDADES HILOS

```
1   class Hilo extends Thread {
2     private int c = 0;
3     public int getContador () {
4       return c;
5     }
6     public void pararHilo() {
7         interrupt();
8     }
9
10    @Override
11    public void run() {
12      //while (!stopHilo) c++;
13      while (!Thread.currentThread().isInterrupted()) c++;
14    }
```

2.1 Clases Java para la gestión de hilos

2.1.13 EJEMPLO PRIORIDADES HILOS

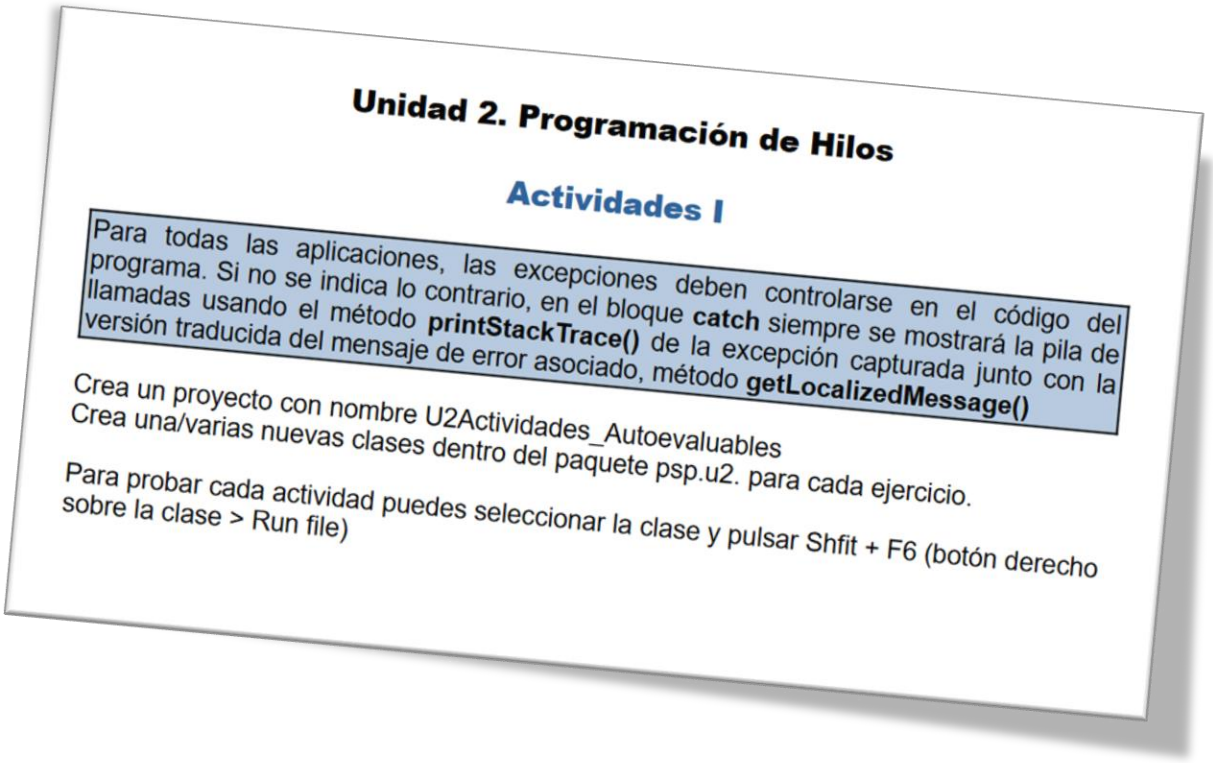
```
1   public class PrincipalPrioridad {
2     public static void main(String args[]) {
3       Hilo h1 = new Hilo();
4       Hilo h2 = new Hilo();
5       Hilo h3 = new Hilo();
6
7       h1.setPriority(Thread.NORM_PRIORITY);
8       h2.setPriority(Thread.MAX_PRIORITY);
9       h3.setPriority(Thread.MIN_PRIORITY);
10
11      h1.start();
12      h2.start();
13      h3.start();
14
15      try {
16           Thread.sleep(3000);
17          } catch (InterruptedException ex) {
18            System.out.println("Error hilo");
19          }
20
21      h1.pararHilo();
22      h2.pararHilo();
23      h3.pararHilo();
24
25      System.out.println("h2 (Prio. Máx: " + h2.getContador());
26      System.out.println("h1 (Prio. Normal: "+ h1.getContador());
27      System.out.println("h3 (Prio. Mínima: "+ h3.getContador());
28    }
29  }
```

2. Hilos

ACTIVIDAD

Enunciado

Realiza los ejercicios iniciales sobre hilos que tienes planteados en la Moodle, siguiendo las instrucciones indicadas.



Unidad 2. Programación de Hilos

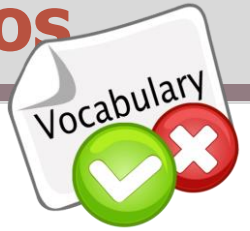
Actividades I

Para todas las aplicaciones, las excepciones deben controlarse en el código del programa. Si no se indica lo contrario, en el bloque **catch** siempre se mostrará la pila de llamadas usando el método **printStackTrace()** de la excepción capturada junto con la versión traducida del mensaje de error asociado, método **getLocalizedMessage()**

Crea un proyecto con nombre U2Actividades_Autoevaluables
Crea una/varias nuevas clases dentro del paquete psp.u2. para cada ejercicio.

Para probar cada actividad puedes seleccionar la clase y pulsar Shfit + F6 (botón derecho sobre la clase > Run file)

2.2 Sincronización y comunicación de hilos



2.2.0 VOCABULARIO

- **Condición de carrera:** Situación en que el correcto funcionamiento de un programa depende del **orden en que se intercale la ejecución de las instrucciones** de sus diferente hilos. Esto ocurre cuando uno o más hilos acceden a información compartida de forma concurrente e intentan modificarla a la vez.

- **Deadlock:** Situación en que **dos o más hilos están bloqueados mutuamente**, todos ellos esperando para conseguir el bloqueo sobre objetos bloqueados por otros hilos, de manera que ninguno de ellos podrá continuar nunca.

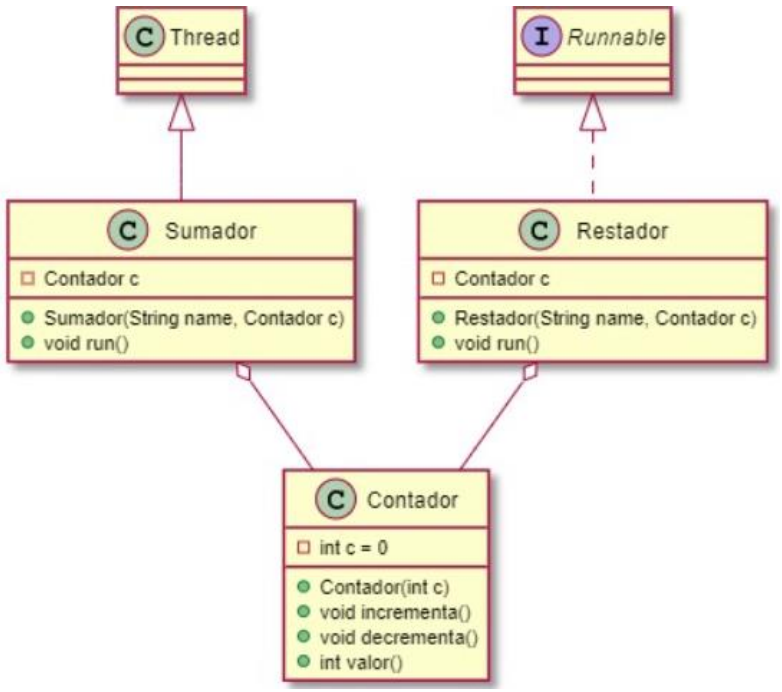
- **Sección crítica:** Fragmento de un programa que **no puede ejecutar de manera simultánea (concurrentemente)** más de un hilo del programa, es decir, que distintos hilos deben ejecutar en exclusión mutua

- **Thread-safe:** Se dice de **una clase** cuyos métodos implementan los mecanismos de sincronización necesarios para el **uso concurrente de sus objetos** por parte de distintos hilos, de manera que no es necesario ningún mecanismo de sincronización externo a la propia clase.

2.2 Sincronización y comunicación de hilos

2.2.1 MEMORIA COMPARTIDA

Los hilos necesitan comunicarse. Para ello **comparten un objeto**.
Ejemplo:



La clase Sumador y la clase Restador comparten el mismo objeto Contador. Sumador suma 1 y Restador resta 1.

- **¿Qué tipo de relación existe entre Contador, Sumador y Restador?**
- **¿Qué crees que va a ocurrir?**

2.2 Sincronización y comunicación de hilos

2.2.1 MEMORIA COMPARTIDA

```java
public class Hilos {
  public static void main(String[] args) {
    // Inicializar el objeto Contador
    Contador c = new Contador(100);

    // Crear y lanzar 2 hilos (Sumador + Restador)
    Sumador suma = new Sumador("Sumador", c);
    Restador rest = new Restador("Restador", c);
    Thread resta = new Thread(rest);

    suma.start();
    resta.start();

      try {
          // El hilo principal espera a que los hilos suma y resta terminen
          suma.join();
          resta.join();
      } catch (InterruptedException ex) {
          System.out.println("Error Hilos");
      }
    System.out.println("El valor final de c es " + c.valor());

  }
}
```

Clase Principal

```java
public class Contador {
private int c = 0;

public Contador(int c) {
  this.c = c;
}

public void incrementa() {
  c++;
}
public void decrementa() {
  c--;
}

public int valor() {
  return c;
}
}
```

Clase Contador

2.2 Sincronización y comunicación de hilos

2.2.1 MEMORIA COMPARTIDA

```java
public class Sumador extends Thread {
  private final Contador c;
  public Sumador(String name, Contador c) {
    // To set the thread name we can access the
    // Thread class constructor
    super(name);
    this.c = c;
  }

  @Override
  public void run() {
    // Ejecutar 30 veces con espera entre 50ms y 150ms
    String rojo = "\u001B[31m";
    for (int i = 0; i < 300; i++) {
      try {
        c.incrementa();
        System.out.println(rojo + Thread.currentThread().getName()
            + " " + c.valor());
        Thread.sleep((long) (Math.random() * 100 + 50));
      } catch (InterruptedException ex) {
        System.out.println("Error");
      }
    }
  }
}
```

Clase Sumador

```java
public class Restador implements Runnable {
  private final Contador c;
  private final String name;
  public Restador(String name, Contador c) {
    // Restador doesn't extend Thread,
    //so it cannot call the Thread constructor super(name);
    this.name= name;
    this.c = c;
  }

  @Override
  public void run() {
    Thread.currentThread().setName(this.name);
    String verde = "\u001B[32m";
    // Ejecutar 30 veces con espera entre 50ms y 150ms
    for (int i = 0; i < 300; i++) {
      try {
        c.decrementa();
        System.out.println(verde + Thread.currentThread().getName()
            + " " + c.valor());
        Thread.sleep((long) (Math.random() * 100 + 50));
      } catch (InterruptedException ex) {
        System.out.println("Error");
      }
    }
  }
}
```

Clase Restador

2.2 Sincronización y comunicación de hilos

2.2.1 MEMORIA COMPARTIDA

Si ejecutamos el programa, **la mayoría de veces obtendremos el resultado esperado, 100.** Sin embargo, hay ocasiones en que podemos encontrar otros valores tales como **99, 101 o cualquier otro.**

Para evitar problemas de sincronización (son problemas como hemos visto aleatorios y muy difíciles de detectar), **necesitamos que los hilos estén sincronizados entre sí.**

```
1    public void incrementa() {
2      c++;
3    }
4    public void decrementa() {
5      c--;
6    }
```

Necesitamos que este código no se pueda ejecutar por las dos clases al mismo tiempo, o sea, no puede ejecutarse de **manera concurrente**.

2.2 Sincronización y comunicación de hilos

2.2.2 SINCRONIZACIÓN

La sincronización en Java se realiza usando **monitores**. Es una propiedad que proporciona la **clase Object**, por lo tanto todas nuestras clases Java, directa o indirectamente, heredan esta propiedad de Object. Este mecanismo permite a un único thread a la vez ejecutar la **sección de código protegida por el monitor**.

Un monitor no es más que un **bloqueo sobre un objeto**. Cada objeto tiene un y sólo un **bloqueo (candado)** interno asociado. El bloqueo de un objeto solamente puede ser adquirido por un thread en cada momento.

Conceptos:

- **Exclusión mutua (sólo un hilo puede disponer de un monitor a la vez).** Por lo tanto, la sincronización utilizando monitores significa que cuando un hilo accede a una sección protegida por un monitor, ningún otro hilo puede acceder a esa o a cualquier otra sección protegida por ese mismo monitor, hasta que el hilo salga de la sección protegida

- La sincronización también asegura que **las escrituras en memoria** realizadas por un thread dentro de un bloque protegido por un monitor son accesibles al resto de threads que accedan a los bloques protegidos por ese mismo monitor.

2.2 Sincronización y comunicación de hilos

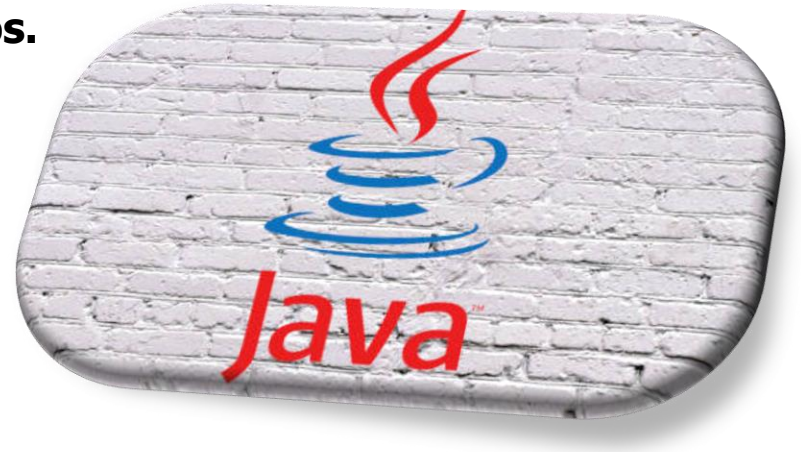
2.2.2 SINCRONIZACIÓN

Secciones críticas

En Java la palabra reservada **synchronized** sirve para hacer que un bloque de código o un método sea **protegido por el cerrojo del objeto**. Para ejecutar un bloque o un método sincronizado, los hilos deben conseguir previamente el bloqueo (candado).

La palabra reservada synchronized puede aplicarse en **distintos tipos de bloques de código** y, en cada caso, se utilizará un objeto de bloqueo distinto.

- **Métodos no estáticos**
- **Métodos estáticos**
- **Bloques de código dentro de los métodos.**



2.2 Sincronización y comunicación de hilos

2.2.2 SINCRONIZACIÓN

Métodos no estáticos

Para métodos no estáticos se añade la palabra reservada **synchronized** a la definición del método

```
1  public class Counter {
2   private int count = 0;
3   public synchronized void add(int value){
4     this.count += value;
5   }
6   public synchronized void sub(int value){
7     this.count -= value;
8   }
9  }
```

El bloqueo se aplica **sobre el objeto** sobre el que se ejecuta el método (this). En este caso **dos hilos no podrían ejecutar a la vez dos métodos del mismo objeto** marcados como synchronized.

2.2 Sincronización y comunicación de hilos

2.2.2 SINCRONIZACIÓN

Métodos estáticos

En este caso el **bloqueo se realiza sobre la clase a la que pertenece el método**. Como sólo hay una instancia de cada objeto clase en la JVM, sólo un hilo a la vez podrá adquirir el monitor y ejecutar el código protegido de una clase estática.

Bloques de código dentro de los métodos

La sincronización no tiene porqué **realizarse sobre todo un método**. A veces es preferible sincronizar sólo una parte de un método. Otras no es posible sincronizar el método completo. Para sincronizar un bloque de código usamos la palabra reservada **synchronized seguida, entre paréntesis, del objeto del que usaremos el monitor**.

```
1  public void add(int value){
2    synchronized(this){
3      this.count += value;
4    }
5  }
```

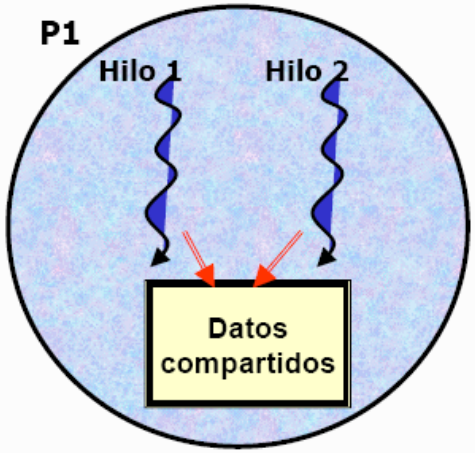
2.2 Sincronización y comunicación de hilos

2.2.2 SINCRONIZACIÓN

Sincronización y actualización de la información.

En la programación multihilo, y dentro de la **sincronización**, un bloque protegido por un monitor nos garantiza que:

- Cuando un hilo entra **en un bloque synchronized se actualizará el valor de todas las variables** visibles para el hilo.
- Cuando un hilo salga de un bloque synchronized **todos los cambios realizados por el hilo se actualizarán** en la memoria principal.



2.2 Sincronización y comunicación de hilos

2.2.2 EJEMPLO "MONITOR IMPLEMENTA UN CONTADOR"

El siguiente ejemplo muestra un **monitor que implementa un contador:**

Clase Contador

```
class Contador {
  // monitor contador
  private int actual;
  public Contador(int inicial)
  {
    actual = inicial;
  }
  public synchronized void inc() { actual++; }
  public synchronized void dec() { actual--; }
  public synchronized int valor() { return actual;}
}
```

Clase Usuario (Hilo)

```
class Usuario extends Thread {
  // clase hilo usuario
  private final Contador cnt;
  private final String color;
  public Usuario(String nombre, Contador cnt, String color) {
    super(nombre);
    this.cnt = cnt;
    this.color=color;

  }
  public void run() {
    for (int i = 0; i < 5; i++) {
      cnt.inc();
      System.out.println(color + "Hola, soy " +
      this.getName() + ", mi contador vale " + cnt.valor());
    }
  }
}
```

2.2 Sincronización y comunicación de hilos

2.2.2 EJEMPLO "MONITOR IMPLEMENTA UN CONTADOR"

El siguiente ejemplo muestra un **monitor** que implementa un contador y se puede observar que los contadores no se repiten en ningún caso**:**

```
class EjemploContador {
  // principal
  final static int N_HEBRAS = 3;
  public static void main(String[] args) {
    // metodo principal
    final Contador cont1 = new Contador(10);
    Usuario hebra[] = new Usuario[N_HEBRAS];
    for (int i = 0; i < N_HEBRAS; i++) {
      //crea hebras y color.
      String color="\u001B[3" + i + "m";
      hebra[i] = new Usuario("la hebra-" + i, cont1,color);
      // lanza hebras
      hebra[i].start(); }
    }
  }
```

Clase Ejemplo Contador

Usar final con objetos de tipo monitor

Un objeto usado como monitor, o como memoria compartida entre hilos, debería ser de tipo **final**, porque si se le asigna un nuevo valor quedan si efecto todos los bloqueos que existan sobre dicho objeto. Un objeto de tipo final una vez que se ha creado y se le ha asignado un valor no se le puede asignar un nuevo valor.

2. Hilos

ACTIVIDAD

Enunciado

Realiza un **programa concurrente** donde se simule el funcionamiento de un cajero, según las siguientes condiciones:

- Luis y Manuel, van a realizar reintegros de una misma cuenta.
- Lo primero que hace cada uno de ellos es comprobar el saldo. Si el saldo es suficiente para el retiro, este se realiza con éxito, de lo contrario el sistema indica al usuario que no hay dinero suficiente.
- Cada vez que se saque dinero, o no se pueda se tiene que mostrar el saldo restante.
- Las 2 personas van a poder hacer reintegros prácticamente al mismo tiempo.
- Van a sacar cantidades aleatorias entre 50€ y 200€. Cada uno hace 5 reintegros.
- El saldo inicial de la cuenta es de 1000€.
- Al finalizar el programa se mostrará cuánto dinero ha sacado Luís y cuánto ha sacado Manuel y cuánto queda en la cuenta.

2. Hilos

ACTIVIDAD

Ejemplo de salida

```
run:
Manuel ha realizado un reintegro de 59 Euros
Luis ha realizado un reintegro de 115 Euros
Saldo actual en cuenta: 826 Euros
Saldo actual en cuenta: 826 Euros
Luis ha realizado un reintegro de 61 Euros
Manuel ha realizado un reintegro de 159 Euros
Saldo actual en cuenta: 606 Euros
Luis ha realizado un reintegro de 126 Euros
Saldo actual en cuenta: 480 Euros
Saldo actual en cuenta: 606 Euros
Luis ha realizado un reintegro de 62 Euros
Saldo actual en cuenta: 289 Euros
Luis ha realizado un reintegro de 116 Euros
Manuel ha realizado un reintegro de 129 Euros
Saldo actual en cuenta: 173 Euros
Saldo actual en cuenta: 173 Euros
Manuel no ha podido realizar un reintegro de 174 Euros
Saldo actual en cuenta: 173 Euros
Manuel no ha podido realizar un reintegro de 196 Euros
Saldo actual en cuenta: 173 Euros
Luis ha retirado 480 Euros de la cuenta
Manuel ha retirado 347 Euros de la cuenta
En la cuenta quedan 173 Euros
BUILD SUCCESSFUL (total time: 0 seconds)
```

2. Hilos

ACTIVIDAD

Enunciado Banco versión 1

Modifica el programa del banco con las siguientes condiciones:

- Luis y Manuel, van a poder realizar también ingresos en la cuenta.
- El saldo inicial será de 500 Euros
- De manera aleatoria se elegirá entre un ingreso o un reintegro.
- No se podrá realizar un reintegro si no hay saldo. Siempre se podrá hacer un ingreso.
- Cada vez que se saque dinero, o no se pueda se tiene que mostrar el saldo restante.
- Cada vez que se ingrese dinero, se tiene que mostrar el saldo restante.
- Van a sacar/ingresar cantidades aleatorias entre 50€ y 200€. Cada uno hace 5 reintegros.
- El saldo inicial de la cuenta es de 500€.
- Al finalizar el programa se mostrará cuánto dinero ha sacado e ingresado Luís, cuánto ha sacado e ingresado Manuel y cuánto queda en la cuenta.

2. Hilos

ACTIVIDAD

Ejemplo de salida

```
Luis ha realizado un ingreso de 52 Euros
Manuel ha realizado un reintegro de 189 Euros
Saldo actual en cuenta: 363 Euros
Saldo actual en cuenta: 363 Euros
Manuel ha realizado un reintegro de 50 Euros
Luis ha realizado un reintegro de 127 Euros
Saldo actual en cuenta: 186 Euros
Saldo actual en cuenta: 186 Euros
Manuel ha realizado un ingreso de 80 Euros
Saldo actual en cuenta: 434 Euros
Luis ha realizado un ingreso de 168 Euros
Manuel ha realizado un reintegro de 170 Euros
Saldo actual en cuenta: 264 Euros
Saldo actual en cuenta: 264 Euros
Luis ha realizado un reintegro de 52 Euros
Saldo actual en cuenta: 148 Euros
Manuel ha realizado un reintegro de 64 Euros
Saldo actual en cuenta: 148 Euros
Luis no ha podido realizar un reintegro de 196 Euros
Saldo actual en cuenta: 148 Euros
Luis ha retirado 179 Euros de la cuenta
Luis ha ingresado 220 Euros de la cuenta
Manuel ha retirado 473 Euros de la cuenta
Manuel ha ingresado 80 Euros de la cuenta
En la cuenta quedan 148 Euros
BUILD SUCCESSFUL (total time: 0 seconds)
```

2. Hilos

ACTIVIDAD

Enunciado Venta de entradas Concierto

Diseñar e implementar un programa que simule la venta de entradas para un concierto, según las condiciones que puedes ver detalladas en la Moodle.

2. Hilos

ACTIVIDAD

Enunciado Adivina número

Diseñar e implementar un programa que simule el juego de adivinar un número que previamente se ha pensado según las condiciones indicadas en la Moodle.

2.2 Sincronización y comunicación de hilos

2.2.3 SINCRONIZACIÓN ENTRE HILOS

En función del estado de los recursos, cada uno de los hilos podrá realizar determinadas acciones o no, permitiendo que los hilos **se queden a la espera de un cambio de estado** que podrá ser notificado por otros hilos.

Además de un mecanismo de **bloqueo sobre los recursos compartidos**, será necesario un **mecanismo de espera** para que, en el caso de que el estado de los recursos compartidos no permita a un hilo realizar una acción, la ejecución del hilo **quede en suspenso** a la espera de que esa condición se cumpla. El mecanismo es **espera no activa, es decir, no se consume ni procesador ni recursos del sistema.**

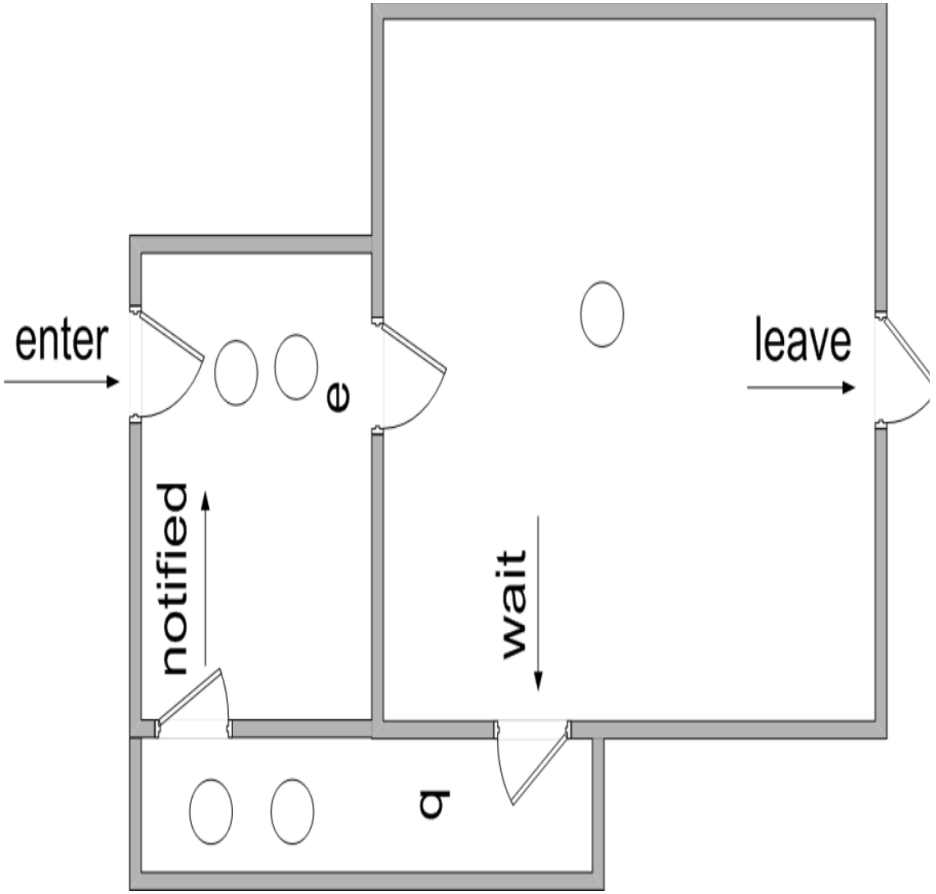
Para resolver este tipo de situaciones volvemos a utilizar **métodos de la clase Object**, accesibles para cualquier objeto.

- **wait():** interrumpe la ejecución del hilo actual.
- **notify():** desbloquea uno de los hilos que están esperando sobre un objeto tras haber ejecutado el método wait()., de manera que pueda continuar su ejecución
- **notifyAll():** desbloquea todos los hilos que están esperando sobre un objeto de bloqueo tras haber ejecutado el método wait(), de manera que puedan continuar su ejecución.

2.2 Sincronización y comunicación de hilos

2.2.3 SINCRONIZACIÓN ENTRE HILOS

Cuando se llama **al método wait(),** el hilo estará dentro de un bloque sincronizado, por lo tanto tendrá el bloqueo del monitor. En ese momento el hilo **libera el bloqueo de ese monitor** y se queda en una **cola** (perteneciente al objeto) de hilos en espera de ser notificados, diferente a la de los hilos que están esperando por el bloqueo.



Cuando se **desbloquea un hilo** porque otro ha llamado a **notify()/notifyAll(),** el hilo vuelve **al punto donde hizo el wait()**, por lo tanto sigue dentro de un bloque sincronizado. Para poder continuar con la ejecución tendrá que pasar a la cola de hilos esperando por el bloqueo y esperar a ser seleccionado para seguir ejecutándose.

2.2 Sincronización y comunicación de hilos

2.2.3 EJEMPLO LIBRERIA

Un autor está escribiendo **un libro** y hasta que no lo termina, **los lectores no lo podrán leer**. En este caso hay un **autor y dos posibles lectores.**

```java
// This is our main class which will create object of above
// classes and run it.
public class Principal {

  public static void main(String args[])
  {
    // Book object on which wait and notify method will be called
    Book book=new Book("El Alquimista");

    BookReader johnReader=new BookReader("Juan",book);
    BookReader arpitReader=new BookReader("Alberto",book);

    johnReader.start();
    arpitReader.start();

    // To ensure both readers started waiting for the book
    try {
      Thread.sleep(3000);
    } catch (InterruptedException e) {
      e.printStackTrace();
    }

    //BookWriter thread which will notify once book get completed
    BookWriter bookWriter=new BookWriter("Autor",book);
    bookWriter.start();
  }
}
```

Clase Principal

2.2 Sincronización y comunicación de hilos

2.2.3 EJEMPLO LIBRERIA

```java
// It is the common java class on which thread
// will act and call wait and notify method.
public class Book {
  String title;
  boolean isCompleted;

  public Book(String title) {
    super();
    this.title = title;
  }
  public String getTitle() {
    return title;
  }
  public void setTitle(String title) {
    this.title = title;
  }
  public boolean isCompleted() {
    return isCompleted;
  }
  public void setCompleted(boolean isCompleted) {
    this.isCompleted = isCompleted;
  }
```

```java
  public synchronized void escribir()
  {
      System.out.println("Autor está escribiendo el libro: " +
      getTitle() );
      try {
        Thread.sleep(1000);
      } catch (InterruptedException e) {
        e.printStackTrace();
      }
      setCompleted(true);
      System.out.println("El libro ha sido escrito");

      notify();
      System.out.println("Se comunica a un lector");
  }

  public synchronized void leer()
  {
      System.out.println(Thread.currentThread().getName()
      +" está esperando a que el libro se escriba: "+getTitle());
      try {
        wait();
        // Ha terminado de leer el libro. Se lo dice al otro lector.
      } catch (InterruptedException e) {
        e.printStackTrace();
      }
      System.out.println(Thread.currentThread().getName()
      +": El libro ha sido terminado: Ahora lo puedo leer");
    }
}
```

Clase Libro (Recurso compartido)

2.2 Sincronización y comunicación de hilos

2.2.3 EJEMPLO LIBRERIA

```
public class BookWriter extends Thread{
  Book book;

  public BookWriter(String name, Book book) {
    super(name);
    this.book = book;
  }

  @Override
  public void run() {
      book.escribir();
    }
}
```

Clase Escritor

```
public class BookReader extends Thread{
  Book book;

  public BookReader(String name, Book book) {
    super(name);
    this.book = book;
  }

  @Override
  public void run() {
     book.leer();
  }
}
```

Clase Lector

2.2 Sincronización y comunicación de hilos

2.2.3 COMENTARIOS SOBRE EL CÓDIGO

Se hacen 2 wait(), uno por cada lector y solo un notify(). Por lo tanto, uno de los lectores queda sin notificar y se queda esperando indefinidamente. **¿Cómo solucionarlo?**

Dos soluciones:

- Desde el escritor usar **notifyAll()** para notificar a todos los lectores. Los lectores serían notificados y podrán acceder al libro de manera excluyente pues book es un objeto sincronizado. El orden ya no lo sabemos, pero no se interrumpirán.
- Un lector cuando acabe de leer el libro, lance un **notify() para que el siguiente lector pueda leer el libro.**

Otro problema. ¿Qué pasa si el orden es diferente?. O sea que BookWriter acabe y lance notify(), pero sin haber aún lectores. En ese caso se notifica a **NADIE…..** Cuando los lectores se activen no habrá nadie que les envie un notify() y se quedarán colgados.

Solución:
Los lectores **sólo se deben bloquear** cuando el libro se esté escribiendo.

```
1  try {
2    if (!book.isCompleted())
3      book.wait();
4  } catch (InterruptedException e) {
```

2. Hilos

ACTIVIDAD

Sumador-Restador

Modifica el ejemplo Sumador-Restador sincronizado

Haz las modificaciones necesarias en las clases del **proyecto Sumador – Restador** para que:

- El primer hilo que haga una operación sobre el contador **sea un Sumador**
- Después de un **Sumador siempre se ejecute un Restador** y después de un Restador siempre se **ejecute un Sumador**, haciendo una secuencia Sumador-Restador-Sumador-Restador-...

ACTIVIDAD

Banco Reintegro – Ingreso con espera

Modificar el programa del Banco teniendo en cuenta ahora las siguientes cuestiones:

- Cuando Luis o Manuel quieran sacar dinero **comprobarán el saldo**. Si el reintegro es menor o igual al saldo, podrá hacer el reintegro. Si no, esperará que haya saldo.

2. Hilos

ACTIVIDAD

Banco Reintegro – Ingreso con espera y limite de saldo.

Sobre el ejemplo anterior, ahora vamos a poner **límites al uso de la cuenta**. En la cuenta nunca podremos **tener más de 2500€ ni menos de 200€**.

Cada hilo mostrará mensajes indicando si en algún momento no puede realizar una operación por falta / exceso de saldo en la cuenta y **se quedará esperando a que la situación cambie** para intentar completar la operación.

Al finalizar el programa se mostrará cuánto dinero ha sacado e ingresado Luís, cuánto ha sacado e ingresado Manuel y cuánto queda en la cuenta.

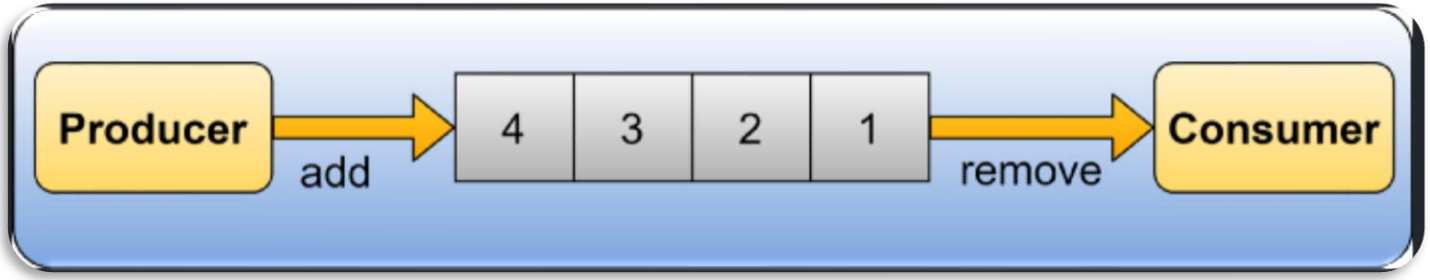
** Puede que algún hilo se quede esperando indefinidamente. Idead algún mecanismo de liberación de todos los hilos que se hayan quedado bloqueados.

2.3 Modelo productor-consumidor

2.3.1 ESQUEMA DE SINCRONIZACIÓN Y COMUNICACIÓN DE HILOS

La **sincronización de threads** es muy importante para que nos se produzcan bloqueo de hilos. Para ello hay que operar de forma correcta con **los recursos compartidos** por hilos concurrentes.

En este punto vamos a ver el **modelo Productor-Consumidor**.



2.3 Modelo productor-consumidor

2.3.2 CLASE PRINCIPAL

En esta clase se **declara el objeto o propiedad que van a compartir el productor y el consumidor**. Este objeto es a través del que se realiza la comunicación, sincronización e intercambio de información entre los hilos.

```
1  import java.util.logging.Level;
2  import java.util.logging.Logger;
3
4  public class ClasePrincipal {
5
6      public static void main(String[] args) {
7          ClaseCompartida objetoCompartido = new ClaseCompartida();
8          Productor p  = new Productor(objetoCompartido);
9          Consumidor c  = new Consumidor(objetoCompartido);
10         p.start();
11         c.start();
12
13         try {
14             // No es obligatorio, pero en ocasiones puede interesar que la ClasePrincipal
15             // espere a que acaben los hilos
16             p.join();
17             c.join();
18            // Acciones a realizar una vez hayan acabado el productor y el consumidor
19
20         } catch (InterruptedException ex) {
21             Logger.getLogger(ClasePrincipal.class.getName()).log(Level.SEVERE, null, ex);
22         }
23     }
24 }
```

Número de hilos por tipo

En el ejemplo se crea un hilo de cada tipo. Puede haber más productores y más consumidores.

2.3 Modelo productor-consumidor

2.3.3 CLASE PRODUCTOR Y CONSUMIDOR

Por otro lado vamos a tener la clase del **productor y del consumidor** que se encargan de realizar las llamadas necesarias **a los métodos del objeto compartido** que reciben como parámetro.

```
1   public class Consumidor extends Thread {
2       private final ClaseCompartida objetoCompartido;
3
4       Consumidor(ClaseCompartida objetoCompartido) {
5           this.objetoCompartido = objetoCompartido;
6       }
7
8       @Override
9       public void run() {
10          // La ejecución del método run estará normalmente gestionada por un bucle
11          // que controlará el ciclo de vida del hilo y se adaptará al problema.
12          // En el caso de simulaciones se harán esperas proporcionales.
13          try {
14              // Código que hace el hilo consumidor
15              objetoCompartido.accionDeConsumir();
16              // La espera es opcional
17              Thread.sleep((long)(Math.random()*1000+1000));
18          } catch (InterruptedException ex) {
19
20          }
21      }
22  }
```

2.3 Modelo productor-consumidor

2.3.3 CLASE PRODUCTOR Y CONSUMIDOR

Por otro lado vamos a tener la clase del **productor y del consumidor** que se encargan de realizar las llamadas necesarias **a los métodos del objeto compartido** que reciben como parámetro.

```
1   public class Productor extends Thread {
2       private final ClaseCompartida objetoCompartido;
3
4       Productor(ClaseCompartida objetoCompartido) {
5           this.objetoCompartido = objetoCompartido;
6       }
7
8       @Override
9       public void run() {
10          // La ejecución del método run estará normalmente gestionada por un bucle
11          // que controlará el ciclo de vida del hilo y se adaptará al problema.
12          // En el caso de simulaciones se harán esperas proporcionales.
13          try {
14              // Código que hace el hilo productor
15              objetoCompartido.accionDeProducir();
16              // La espera es opcional
17              Thread.sleep((long)(Math.random()*1000+1000));
18          } catch (InterruptedException ex) {
19
20          }
21      }
```

2.3 Modelo productor-consumidor

2.3.4 CLASE COMPARTIDA

El modelo se completa con la **clase compartida**. Aquí vamos a crear los métodos a los que acceden productores y consumidores y, además, vamos a realizar la sincronización entre hilos para que no se produzcan **condiciones de carrera**.

```
1   class ClaseCompartida {
2       int valorAccedidoSimultaneamente;
3
4       ClaseCompartida() {
5           // Se inicializa el valor
6           this.valorAccedidoSimultaneamente = 0;
7       }
8
9       public synchronized void accionDeConsumir() {
10          // Si no se cumple la condición para poder consumir, el consumidor debe esperar
11          while (valorAccedidoSimultaneamente == 0) {
12              try {
13                  System.out.println("Consumidor espera...");
14                  wait();
15              } catch (InterruptedException ex) {
16                  // Si es necesario se realizará la gestión de la Interrupción
17              }
18          }
19
20          // Cuando se ha cumplido la condición para consumir, el consumidor consume
21          valorAccedidoSimultaneamente--;
22          System.out.printf("Se ha consumido: %d.\n", valorAccedidoSimultaneamente);
23
24          // Se activa a otros hilos que puedan estar en espera
25          notifyAll();
26      }
27
```

2.3 Modelo productor-consumidor

2.3.4 CLASE COMPARTIDA

El modelo se completa con la **clase compartida**. Aquí vamos a crear los métodos a los que acceden productores y consumidores y, además, vamos a realizar la sincronización entre hilos para que no se produzcan **condiciones de carrera**.

```
28      public synchronized void accionDeProducir () {
29          // Si no se cumple la condición para poder producir, el productor debe esperar
30          while (valorAccedidoSimultaneamente > 10) {
31              try {
32                  System.out.println("Productor espera...");
33                  wait();
34              } catch (InterruptedException ex) {
35                  // Si es necesario se realizará la gestión de la Interrupción
36              }
37          }
38
39          // Cuando se ha cumplido la condición para producir, el productor produce
40          valorAccedidoSimultaneamente++;
41          System.out.printf("Se ha producido: %d.\n", valorAccedidoSimultaneamente);
42
43          // Se activa a otros hilos que puedan estar en espera
44          notifyAll();
45      }
46  }
```

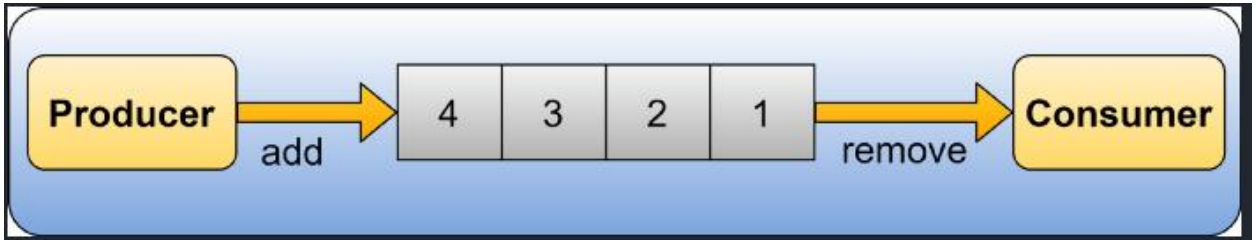
2.3 Modelo productor-consumidor

2.3.5 RESUMEN MODELO PRODUCTOR - CONSUMIDOR

El modelo **Productor-Consumidor** original trabaja con un **buffer** en el que el Productor va **depositando información y el Consumidor la va sacando**, de forma que el buffer nunca se llene ni se pueda leer si está vacío. En nuestro ejemplo, lo hemos simplificado al uso de **una variable** que nunca puede exceder **el valor de 10 ni ser inferior a 0**.

Esa variable puede ser **cualquier tipo de dato**, y el código de las clases variará en función de ello, para **adaptarlo al problema** y al control del tipo de dato utilizado.

Además, las **condiciones o estados** que se utilizan para las esperas y las actualizaciones será lo que nosotros, como programadores, **tengamos que adaptar** al modelo para hacerlo funcionar en situaciones diferentes.



2. Hilos

ACTIVIDAD

Parking

Se tiene un aparcamiento con **50 plazas**, numeradas de 1 a 50, al que intentan acceder coches continuamente. Los coches llegan con una frecuencia entre 5 y 15 minutos.

Cada coche se identifica con una matrícula aleatoria en **formato [9999 LLL]** (4 números y 3 letras) que queda asociada a la plaza de aparcamiento libre que se le asigna, si la hay.Si el parking está lleno, los coches esperan en la entrada a que haya plaza. El tiempo de permanencia de los coches en el parking está entre **30 y 60 minutos.**

Realiza **varias simulaciones** para ver si el parking se va llenando, y en cuento tiempo lo hace, o si por el contrario nunca llega a llenarse. Muestra información detallada de lo que va pasando, informando de **cuántas plazas están ocupadas** y cuántas quedan libres en cada momento.

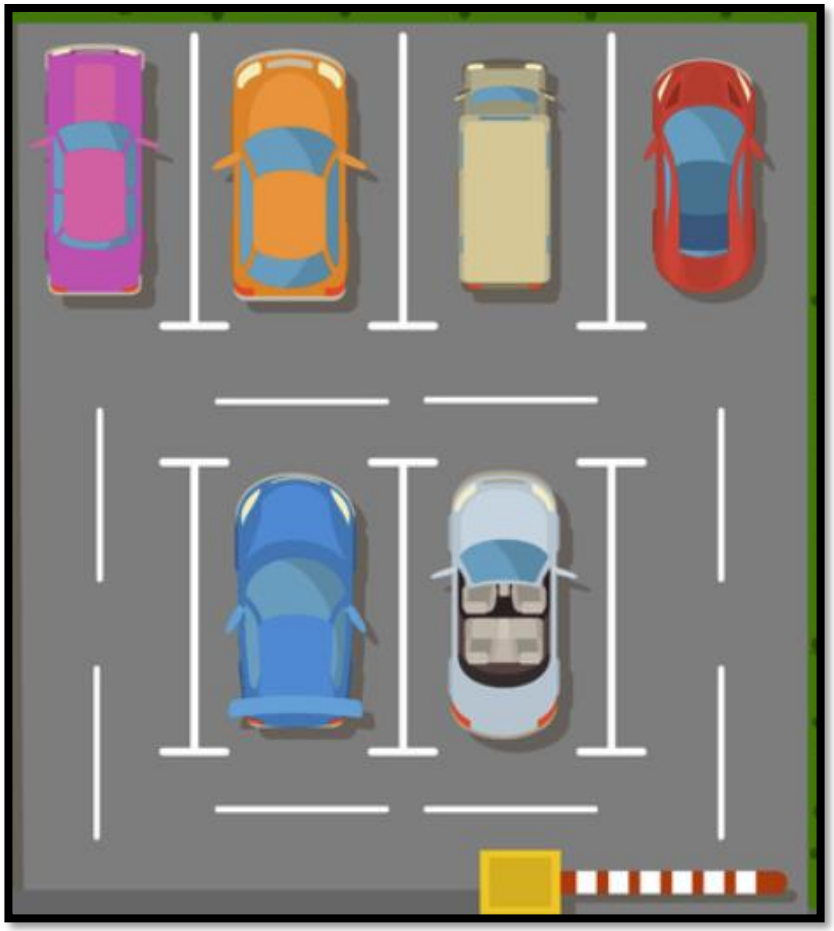
En la siguiente diapositiva tienes **un ejemplo de salida**, con 5 plazas de aparcamiento.

2. Hilos

ACTIVIDAD

Parking (Ejemplo de salida. 5 plazas de aparcamiento)

```
run:
[0807 FIH] aparca en la plaza 0 -> Quedan 4 plazas libres
[0904 QOK] aparca en la plaza 1 -> Quedan 3 plazas libres
[7622 CIF] aparca en la plaza 2 -> Quedan 2 plazas libres
[9467 LMG] aparca en la plaza 3 -> Quedan 1 plazas libres
[5560 JJY] aparca en la plaza 4 -> Quedan 0 plazas libres
[6052 EAO] espera a que queden plazas libres
[2328 RBQ] espera a que queden plazas libres
[5221 SCZ] espera a que queden plazas libres
[7176 HHU] espera a que queden plazas libres
[8492 ZSL] espera a que queden plazas libres
[3556 NXF] espera a que queden plazas libres
[7936 UWJ] espera a que queden plazas libres
[7232 AIQ] espera a que queden plazas libres
[4351 LKV] espera a que queden plazas libres
[6243 EUN] espera a que queden plazas libres
[9813 COJ] espera a que queden plazas libres
[6110 OBK] espera a que queden plazas libres
[4354 YWF] espera a que queden plazas libres
[0673 XGP] espera a que queden plazas libres
[6126 VMG] espera a que queden plazas libres
[3591 LRL] espera a que queden plazas libres
[9675 EXV] espera a que queden plazas libres
[7666 HNP] espera a que queden plazas libres
[1998 AMG] espera a que queden plazas libres
[3089 HML] espera a que queden plazas libres
[5977 UVY] espera a que queden plazas libres
[2729 QNQ] espera a que queden plazas libres
[3910 VFL] espera a que queden plazas libres
[3616 TRL] espera a que queden plazas libres
[4064 DIH] espera a que queden plazas libres
[9467 LMG] deja la plaza 3 -> Quedan 1 plazas libres
[6052 EAO] aparca en la plaza 3 -> Quedan 0 plazas libres
```



2. Hilos

ACTIVIDAD

Puente

Necesitamos simular un sistema que controla **el paso de personas por un puente**, siempre en la misma dirección, para que se cumplan las siguientes restricciones.

- No pueden pasar más de tres personas a la vez.
- No puede haber más de 200kg de peso en ningún momento.

El **tiempo de llegada** entre dos personas está entre 1 y 30 minutos y para atravesar el puente se tarda también un tiempo entre 10 y 50 minutos. Las personas tienen un peso entre 40 y 120kg.

Realizar la simulación usando una **escala de tiempo proporcional**. Muestra cuántas personas hay en el puente en cada momento, cuánto peso está soportando el puente y cuánto tiempo les queda a cada uno para terminar de cruzar el puente.

En la siguiente diapositiva tienes **un ejemplo de salida**.

2. Hilos

ACTIVIDAD

Puente

```
run:
[Persona 0] cruza el puente
Personas en el puente: 1 - Peso en el puente: 106
[Persona 1] cruza el puente
Personas en el puente: 2 - Peso en el puente: 183
[Persona 2] con peso 108kg. espera a cruzar el puente
Personas en el puente: 2 - Peso en el puente: 183
[Persona 3] con peso 74kg. espera a cruzar el puente
Personas en el puente: 2 - Peso en el puente: 183
[Persona 4] con peso 70kg. espera a cruzar el puente
Personas en el puente: 2 - Peso en el puente: 183
[Persona 5] con peso 74kg. espera a cruzar el puente
Personas en el puente: 2 - Peso en el puente: 183
Persona 0 me quedan 10 minutos para terminar de cruzar
Persona 1 me quedan 18 minutos para terminar de cruzar
[Persona 6] con peso 77kg. espera a cruzar el puente
Personas en el puente: 2 - Peso en el puente: 183
[Persona 7] con peso 46kg. espera a cruzar el puente
Personas en el puente: 2 - Peso en el puente: 183
[Persona 8] con peso 61kg. espera a cruzar el puente
Personas en el puente: 2 - Peso en el puente: 183
[Persona 9] con peso 81kg. espera a cruzar el puente
Personas en el puente: 2 - Peso en el puente: 183
[Persona 10] con peso 97kg. espera a cruzar el puente
Personas en el puente: 2 - Peso en el puente: 183
Persona 0 me quedan 0 minutos para terminar de cruzar
[Persona 0] sale del puente
Personas en el puente: 1 - Peso en el puente: 77
[Persona 2] cruza el puente
Personas en el puente: 2 - Peso en el puente: 185
[Persona 10] con peso 97kg. espera a cruzar el puente
Personas en el puente: 2 - Peso en el puente: 185
[Persona 9] con peso 81kg. espera a cruzar el puente
```

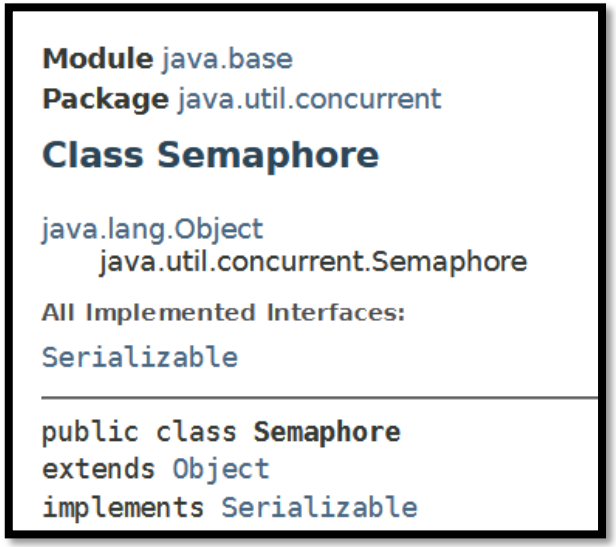
2.4 Mecanismos alternativos de sincronización

2.4.1 SEMÁFOROS

Un semáforo es un **mecanismo para permitir, o restringir**, el acceso a recursos compartidos en un entorno de multiprocesamiento, con varios hilos ejecutándose de forma concurrente.

Los semáforos se emplean para permitir el acceso a **diferentes partes de programas (llamados secciones críticas)** donde se manipulan variables o recursos que deben ser accedidos de forma especial. Según **el valor con que son inicializados** se permiten a **más o menos procesos** utilizar el recurso de forma simultánea.

Especificación de
java.util.concurrent.Semaphore



Module java.base
Package java.util.concurrent

Class Semaphore

java.lang.Object
    java.util.concurrent.Semaphore

All Implemented Interfaces:

Serializable

```
public class Semaphore
extends Object
implements Serializable
```

2.4 Mecanismos alternativos de sincronización

2.4.1 SEMÁFOROS

El funcionamiento de los semáforos se basa en **el uso de dos métodos**, así como en el valor inicial **permits** con el que se crea el semáforo:

- **release():** Ejecutado por un hilo para **liberar el semáforo** cuando el hilo **ha terminado de ejecutar la sección crítica**. Por defecto se incrementa la variable permits en 1, aunque puede recibir un valor e incrementarla en esa cantidad.

- **acquire():** Ejecutado por un hilo para **acceder al semáforo**. Para que un hilo pueda tomar el control del semáforo y no quedarse bloqueado, **la variable permits debe tener un valor mayor que cero**. También puede recibir un valor, por lo que permits tendrá que ser mayor que dicho valor.

- **permits:** Se inicializa **a la cantidad de recursos existentes** o hilos que queramos que puedan acceder simultáneamente. Así, cada proceso, al ir solicitando un recurso, **verificará que el valor del semáforo sea mayor de 0**; si es así es que existen recursos libres, seguidamente acaparará el recurso y restará el valor del semáforo. Cuando el semáforo alcance el valor 0, significará que todos los recursos están siendo utilizados, y los procesos que quieran solicitar un recurso deberán esperar a que el semáforo sea positivo (algún hilo haga un release).

2.4 Mecanismos alternativos de sincronización

2.4.1 EJEMPLO PRODUCTOR-CONSUMIDOR CON SEMÁFOROS

```java
import java.util.logging.Level;
import java.util.logging.Logger;
import java.util.concurrent.Semaphore;

public class Almacen {

  private final int MAX_LIMITE = 20;
  private int producto = 0;
  private final Semaphore productor = new Semaphore(MAX_LIMITE);
  private final Semaphore consumidor = new Semaphore(0);
  private final Semaphore mutex = new Semaphore(1);

  public void producir(String nombreProductor) {
    System.out.println(nombreProductor + " intentando almacenar un producto");
    try {
        // En el ejemplo, hasta 20 productores pueden acceder a la vez
        productor.acquire();
        // Sin embargo, sólo 1 (consumidor/productor) a la vez podrá actualizar
        mutex.acquire();

        producto++;
        System.out.println(nombreProductor + " almacena un producto. "
            + "Almacén con " + producto + (producto > 1 ? " productos." : " producto."));
        mutex.release();

        Thread.sleep(500);

    } catch (InterruptedException ex) {
      Logger.getLogger(Almacen.class.getName()).log(Level.SEVERE, null, ex);
    } finally {
      // El productor permite que un consumidor pueda acceder
      consumidor.release();
    }

  }
```

2.4 Mecanismos alternativos de sincronización

2.4.1 EJEMPLO PRODUCTOR-CONSUMIDOR CON SEMÁFOROS

```
37      public void consumir(String nombreConsumidor) {
38        System.out.println(nombreConsumidor + " intentando retirar un producto");
39        try {
40            // En el ejemplo siempre tiene que llegar un consumidor antes que un productor
41            consumidor.acquire();
42            // Sin embargo, sólo 1 (consumidor/productor) a la vez podrá actualizar
43            mutex.acquire();
44
45            producto--;
46            System.out.println(nombreConsumidor + " retira un producto. "
47                + "Almacén con " + producto + (producto > 1 ? " productos." : " producto."));
48            mutex.release();
49
50            Thread.sleep(500);
51        } catch (InterruptedException ex) {
52          Logger.getLogger(Almacen.class.getName()).log(Level.SEVERE, null, ex);
53        } finally {
54          // El consumidor avisa para que un productor pueda volver a dejar productos.
55          productor.release();
56
57        }
58      }
59
60  }
```

2.4 Mecanismos alternativos de sincronización

2.4.2 MECANISMO DE ALTO NIVEL

Java, en su **paquete java.util.concurrent** proporciona varias clases **thread-safe** que nos permiten acceder a los elementos de colecciones y tipos de datos sin preocuparnos de la concurrencia.

Es un paquete muy amplio que contiene **multitud de clases** que podemos utilizar en nuestros desarrollos multihilo para **simplificar la complejidad de los mismo**

- **Colas concurrentes:** La interfaz BlockingQueue define una cola FIFO que bloquea hilos.
- **Colecciones concurrentes:** ConcurrentMap es una subinterfaz de java.util.Map.
- **Variables atomáticas:** El paquete java.util.concurrent.atomic incluye clases que proporcionan acciones atómicas.

2.4 Mecanismos alternativos de sincronización

2.4.3 EXECUTORS, CALLABLES Y FUTURE

Existen muchas aproximaciones y librerías que permiten el uso y gestión de hilos desde un programa.

- **Executors** nos va a permitir definir un pool de threads (un conjunto de hilos) que se encargarán de ejecutar las tareas, pero con un límite en cuanto al número de hilos creados. **Executors:** **Ejemplo supermercado**

- **Callable** viene a poner solución a uno de los problemas que tenemos con la interfaz Runnable, la posibilidad de devolver un valor desde este método.

- **Future** es una interfaz que implementa el objeto que devuelve el resultado de la ejecución de un Callable. Se puede seguir ejecutando una aplicación hasta que necesite obtener el resultado del hilo Callable, momento en el que se invoca el método get() en la instancia Future. Si el resultado ya está disponible se recoge y en caso contrario se bloqueará en la llamada hasta que su método call() devuelva el resultado.