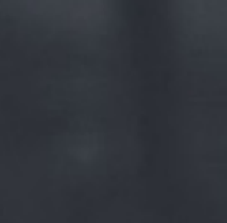
Ciencias de la Computación
FACULTAD DE CIENCIAS FÍSICAS Y MATEMÁTICAS
UNIVERSIDAD DE CHILE

Curso de **Programación** en **Java**

Juan Francisco Maldonado León
Arquitecto de Software

Ciencias de la Computación
FACULTAD DE CIENCIAS FÍSICAS Y MATEMÁTICAS
UNIVERSIDAD DE CHILE

Programación Concurrente



Juan Francisco Maldonado León
Arquitecto de Software

Threads

Un Thread en Java se pueden crear de dos formas: mediante la extensión de la clase **Thread** o implementando la interfaz **Runnable**

Extendiendo de la clase Threads

Al crear un hilo extendiendo de la clase Thread, es necesario sobreescribir (override) el método run(). el método run no recibe parámetros y no retorna nada.

Para iniciar un hilo, es necesario invocar al método start() de la clase Thread.

Threads

Programación **Concurrente**

```
public class EjemploHilo extends Thread {

    @Override
    public void run() {
        try {
            sleep(1000);
        } catch (InterruptedException ex) {
        }
        System.out.println("El nombre del thread es: " + getName());
    }

    public static void main(String args[]) {
        EjemploHilo thread=new EjemploHilo();
        thread.start();
        System.out.println("metodo main; El nombre del thread es: "+
                Thread.currentThread().getName());
    }
}
```

Resultado al ejecutar el programa

```
metodo main; El nombre del thread es: main
El nombre del thread es: Thread-0
```

Threads

Programación **Concurrente**

Inicia Thread Main

```
EjemploHilo thread=new EjemploHilo();
```

```
thread.start();
```

JVM genera un nuevo hilo e invoca al método run()

```
System.out.println("El nombre del thread es: "
+ getName());
```

El Thread Main continua la ejecución

```
System.out.println("metodo main; El nombre del thread es: "+
        Thread.currentThread().getName());
```

Method	Method Type	Short Description
`Thread currentThread()`	Static method	Returns reference to the current thread.
`String getName()`	Instance method	Returns the name of the current thread.
`int getPriority()`	Instance method	Returns the priority value of the current thread.
`void join()`, `void join(long)`, `void join(long, int)`	Overloaded instance methods	The current thread invoking join on another thread waits until the other thread dies. You can optionally give the timeout in milliseconds (given in `long`) or timeout in milliseconds as well as nanoseconds (given in `long` and `int`).
`void run()`	Instance method	Once you start a thread (using the `start()` method), the `run()` method will be called when the thread is ready to execute.
`void setName(String)`	Instance method	Changes the name of the thread to the given name in the argument.
`void setPriority(int)`	Instance method	Sets the priority of the thread to the given argument value.
`void sleep(long)` `void sleep(long, int)`	Overloaded static methods	Makes the current thread sleep for given milliseconds (given in long) or for given milliseconds and nanoseconds (given in `long` and `int`).
`void start()`	Instance method	Starts the thread; JVM calls the `run()` method of the thread.
`String toString()`	Instance method	Returns the string representation of the thread; the string has the thread's name, priority, and its group.

Implementando la interface Runnable

La misma clase Thread implementa la interfaz **Runnable**. En lugar de extender de la clase Thread, podemos implementar la interfaz Runnable la cual posee un solo método: run()::void

Implementando la interface Runnable

La clase Thread posee un constructor que recibe como parámetro un objeto que implemente la interfaz Runnable

Thread (Runnable runnable)

Threads

Programación **Concurrente**

```java
public class EjemploHilo implements Runnable {

    @Override
    public void run() {
        System.out.println("El nombre del thread es: "
        + Thread.currentThread().getName());
    }

    public static void main(String args[]) {
        Thread thread= new Thread( new EjemploHilo() );
        thread.start();
        System.out.println("metodo main; El nombre del thread es: "+
                Thread.currentThread().getName());
    }
}
```

Método Run

La clase Thread tiene la implementación predeterminada del método run (), por lo que si no se proporciona una definición, se utilizara la definida en la clase thread la cual no realiza ninguna acción.

Nombre y Prioridad

Cada Thread tiene un nombre, que se puede utilizar para identificar el hilo. Si usted no da un nombre explícitamente, un hilo obtendrá un nombre por defecto.

La prioridad puede variar de 1, el más bajo, a 10, la más alta.
La prioridad normal es de 5 por defecto.

Ejercicio

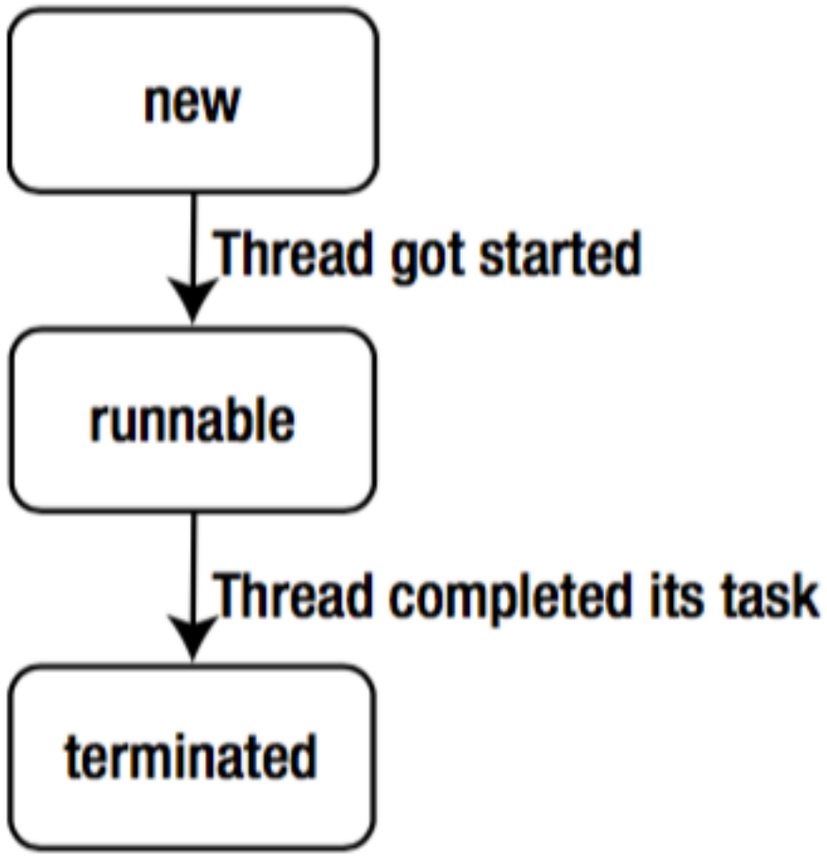
Supongamos que se desea implementar un temporizador de cuenta atrás para una bomba de tiempo, que cuenta de nueve a cero deteniéndose 1 segundo por cada cuenta. Después de llegar a cero, se debe imprimir "Boom !!!" Se debe implementar esta funcionalidad mediante la creación de un hilo para ejecutar la cuenta atrás. Con el fin de hacer una pausa para cada segundo, se puede llamar al método Thread.sleep.

Threads

Programación **Concurrente**

```
public class BombaDeTiempo extends Thread {

    private String[] numerosStr = { "Cero", "Uno", "Dos", "Tres", "Cuatro", "Cinco",
            "Seis", "Siete", "Ocho", "Nueve" };

    public void run() {
        for (int i = 9; i >= 0; i--) {
            try {
                System.out.println(numerosStr[i]);
                Thread.sleep(1000);
            } catch (InterruptedException ie) {
                ie.printStackTrace();
            }
        }
    }

    public static void main(String[] s) throws InterruptedException {
        BombaDeTiempo timer = new BombaDeTiempo();
        System.out.println("Comienza la  cuenta regresiva... ");
        timer.start();
        timer.join();
        System.out.println("Boom!!!");
    }
}
```

Estados de un Thread



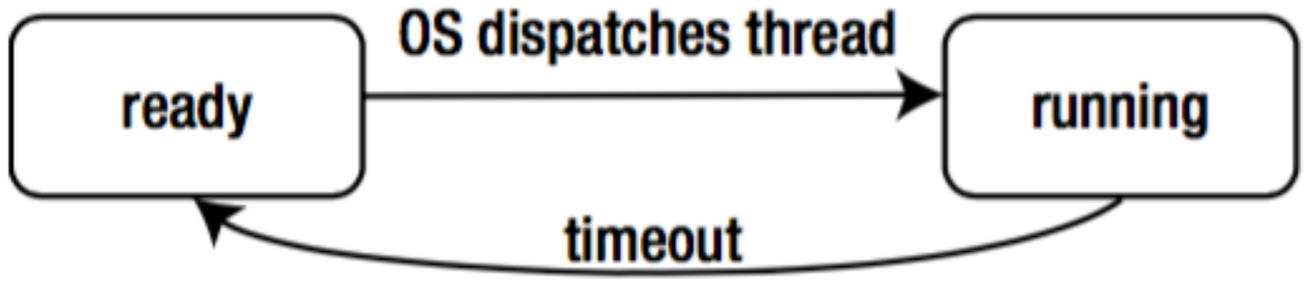
Threads

Programación **Concurrente**

```
BombaDeTiempo timer = new BombaDeTiempo();
System.out.println( "Luego de instanciar\n" + timer.getState() );
timer.start();
System.out.println( "Luego de inicar\n" + timer.getState() );
timer.join();
System.out.println( "Al terminar\n" + timer.getState() );
```

```
Luego de instanciar
NEW
Luego de inicar
RUNNABLE
Al terminar
TERMINATED
```

Estado de Ejecución de un Thread



Estado de Ejecución de un Thread

Una vez que un hilo hace que la transición de estado desde el estado NEW al estado RUNNABLE, se puede pensar del hilo, que tiene dos estados a nivel de sistema operativo: estado de servicio y estado de ejecución.

Estado de Ejecución de un Thread

Un hilo está en el estado listo cuando se está a la espera para el sistema operativo que se ejecute en el procesador. Cuando el sistema operativo realmente ejecuta en el procesador, que está en el estado de ejecución.

Data Races

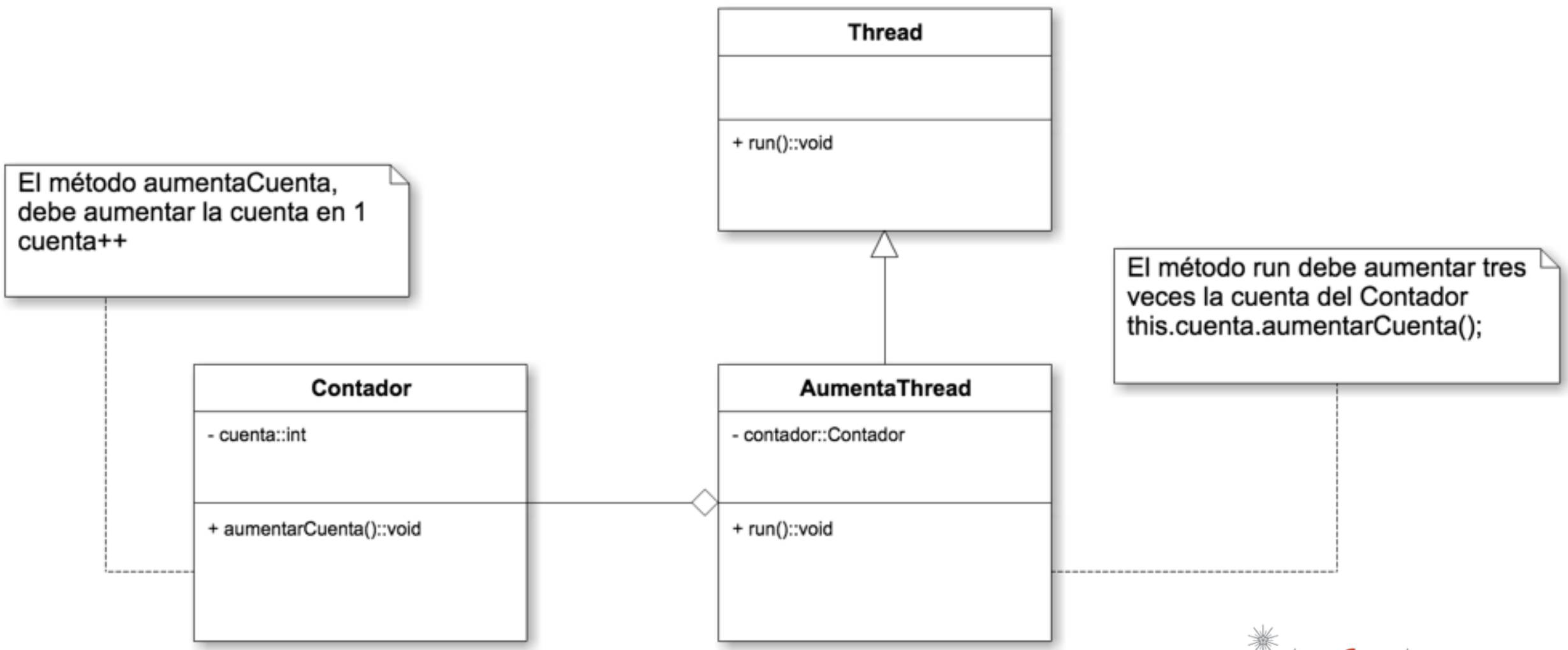
Problemas de Acceso Concurrente

Los Threads comparten memoria con otros objetos, por lo cual, pueden modificar el estado (atributos) de estos de forma concurrente.

Cuando dos o más hilos están tratando de modificar una variable, se genera un error conocido como Data Race

Data Races

Problemas de Acceso Concurrente



Threads

Programación **Concurrente**

```java
public class Contador {

    private int cuenta;

    /**
     * Aumenta la cuenta en 1
     */
    public void aumentarCuenta() {
        this.cuenta++;
        System.out.print(this.cuenta);
    }

    // getter - setters ...
```

Threads

Programación **Concurrente**

```
public class AumentaThread extends Thread {

    private Contador contador;

    public AumentaThread(Contador contador) {
        this.contador = contador;
    }

    @Override
    public void run() {
        for (int i = 0; i < 3; i++) {
            contador.aumentarCuenta();
        }
    }
}
```

Threads

Programación **Concurrente**

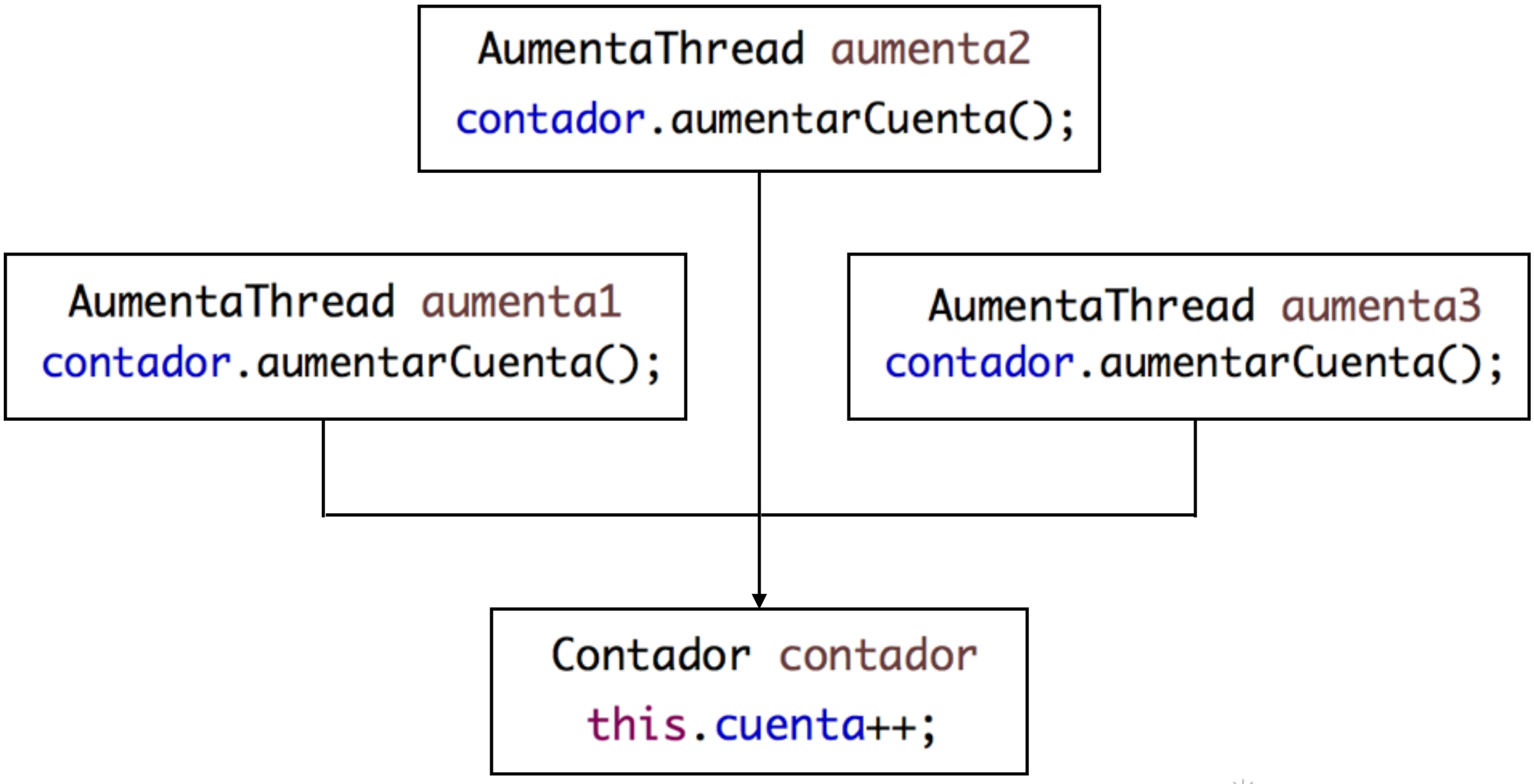
```
public class Programa {

    public static void main(String[] args) {
        Contador contador = new Contador();
        AumentaThread aumenta1 = new AumentaThread(contador);
        AumentaThread aumenta2 = new AumentaThread(contador);
        AumentaThread aumenta3 = new AumentaThread(contador);
        aumenta1.start();
        aumenta2.start();
        aumenta3.start();
    }

}
```

Resultado del Programa

245267389
245367289
145367289

Threads

Programación **Concurrente**



Para resolver este problema de acceso a datos, es necesario asegurarse de que solo un hilo pueda realizar la acción de escritura y lectura a la vez.

La parte del código que accede y modifica los datos se conoce como **sección critica.**

```
public void aumentarCuenta() {
    this.cuenta++;
    System.out.print(this.cuenta);
}
```

Sincronización de bloques

```
public void aumentarCuenta() {
    synchronized (this) {
        this.cuenta++;
        System.out.print(this.cuenta);
    }
}
```

Sincronización de métodos

```
public synchronized void aumentarCuenta() {
        this.cuenta++;
        System.out.print(this.cuenta);
}
```

You cannot declare constructors synchronized; it will result in a compiler error. For example, for

Sincronización de Constructores

No es posible declarar un constructor como synchronized; es un error de compilación

La JVM asegura que sólo un hilo puede invocar una llamada al constructor. Por lo tanto, no hay necesidad de declarar un constructor sincronizada.

Sin embargo, si se quiere, se puede utilizar bloques sincronizados dentro de constructores.

```
public synchronized Contador()
{
    this.cuenta = 0;
}
```

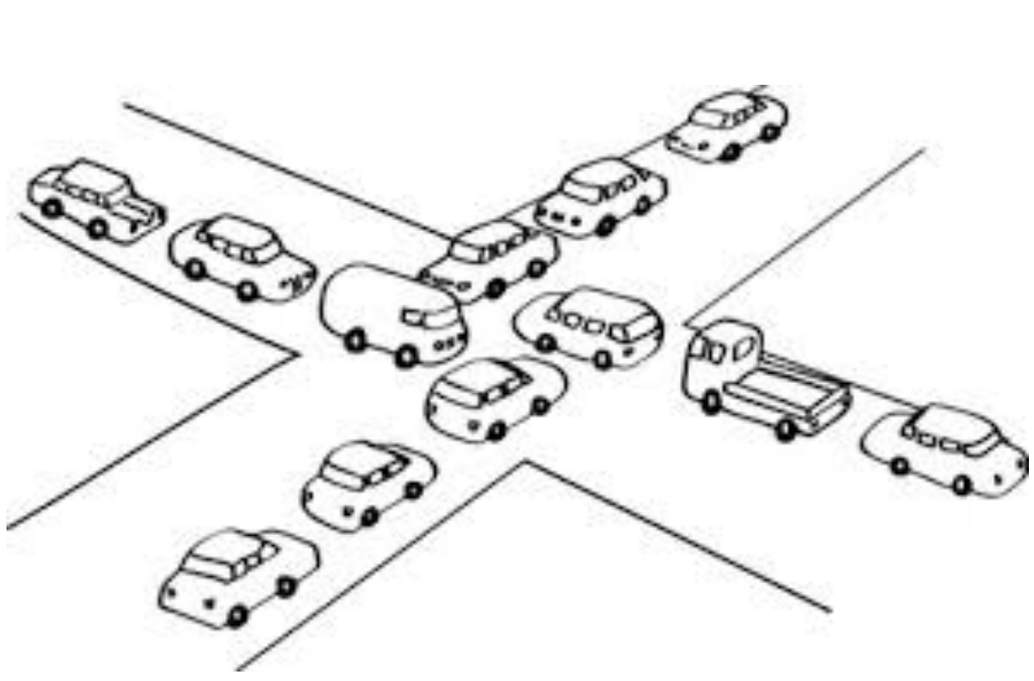
DeadLock

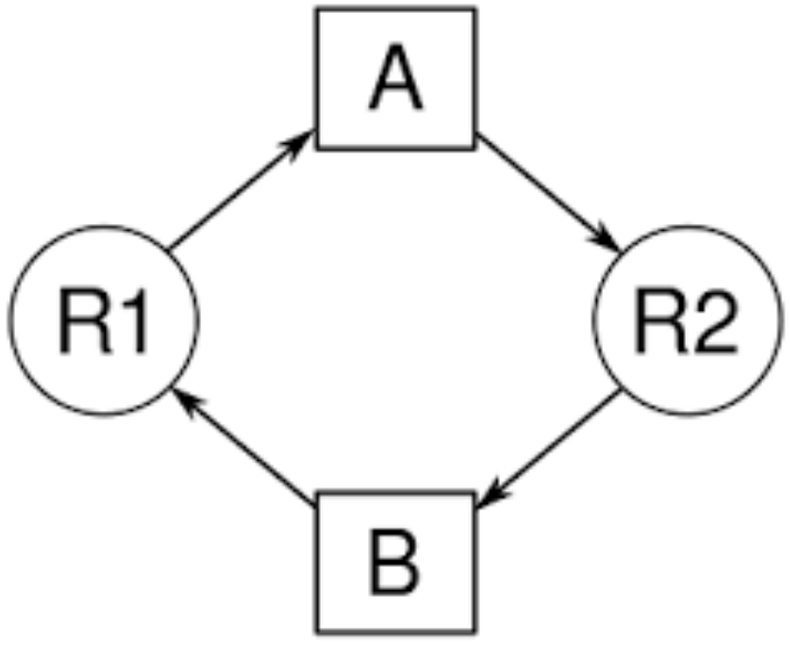
Problemas de Acceso Concurrente

DeadLock describe una situación en la que dos o más hilos se bloquean para siempre, a la espera de uno al otro.

DeadLock

Problemas de Acceso Concurrente





API Concurrencia

java.util.concurrent

Semaphore Controla el acceso a uno o mas recursos
Phaser is used to support a synchronization barrier.
CountDownLatch allows threads to wait for a countdown to complete.
Exchanger supports exchanging data between two threads.
CyclicBarrier enables threads to wait at a predefined execution point.

API Concurrencia

java.util.concurrent

Semaphore Controla el acceso a uno o mas recursos
Phaser is used to support a synchronization barrier.
CountDownLatch allows threads to wait for a countdown to complete.
Exchanger supports exchanging data between two threads.
CyclicBarrier enables threads to wait at a predefined execution point.