Concurrencia en Java. Tareas

Samsung
TECH INSTITUTE

Contenido

- Tareas. Creación y estados
- Sincronización entre tareas
- Cooperación

Concurrencia en Java. Tareas

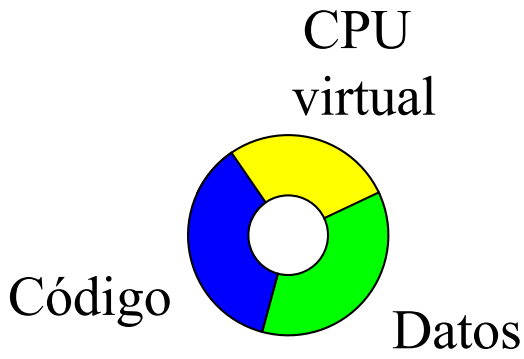
- Un Proceso
  - Contiene su propio espacio virtual de direcciones
    - Es distinto para cada proceso
  - Contiene una o más hebras (*threads, tareas*) de ejecución
    - La tarea es la unidad básica de ejecución.
    - Las tareas se planifican siguiendo los criterios clásicos
      - disponibilidad de recursos
      - prioridades
      - justicia
      - ....
    - Cada tarea tiene su propia pila de ejecución, variables locales y contador de programa.
    - Los cambios de contexto son muy rápidos.

Planificación de tareas

- Prioridades
  - Cada tarea tiene una prioridad. Se ejecuta la tarea de mayor prioridad
  - De 1 (min.) a 10 (máx.)
- Pre-emptivo
  - Si cuando se está ejecutando una tarea aparece otra de mayor prioridad, le quita el control y pasa a ejecutarse.
- Time-slicing
  - Cada tarea tiene un tiempo máximo de ejecución. Cuando se cumple, deja el control a otra de su misma prioridad.

Partes de una tarea

- CPU virtual
  - Simula una CPU con sus registros y estados.
- Código
  - que se va a ejecutar en la tarea
- Datos
  - sobre los que va a actuar la tarea.



- Un objeto de la clase **`Thread`** simula la CPU virtual.
- Un objeto (que implemente la interfaz **`Runnable`**) define el código a ejecutar y los datos a manipular.

Cómo crear una tarea 1/2

- Creamos una clase que implemente la interfaz **`Runnable`**
  - Este interfaz obliga a definir un método

    void run()
    - que debe implementar el cuerpo de ejecución de la tarea

```
package prTema7;

class MiCodigo implements Runnable {
        int tope;
        MiCodigo(int t) {
                tope = t;
        }
        public void run() {
                int i = 0;
                while (i < tope) {
                        System.out.println(Thread.currentThread().getName() + i);
                        i++;
                }
        }
}
```

- Los datos a manipular por la tarea son **`tope`** e **`i`** aunque puede actuar sobre cualquier referencia accesible.

Cómo crear una tarea 2/2

- Creamos una instancia de la clase el código y los datos

```
MiCodigo mc = new MiCodigo(1000);
```

- Creamos una instancia de la CPU virtual para que ejecute este código

```
Thread mt = new Thread(mc);
```

- La tarea está creada y preparada para ejecutar el código que hemos indicado en el método **`run()`**

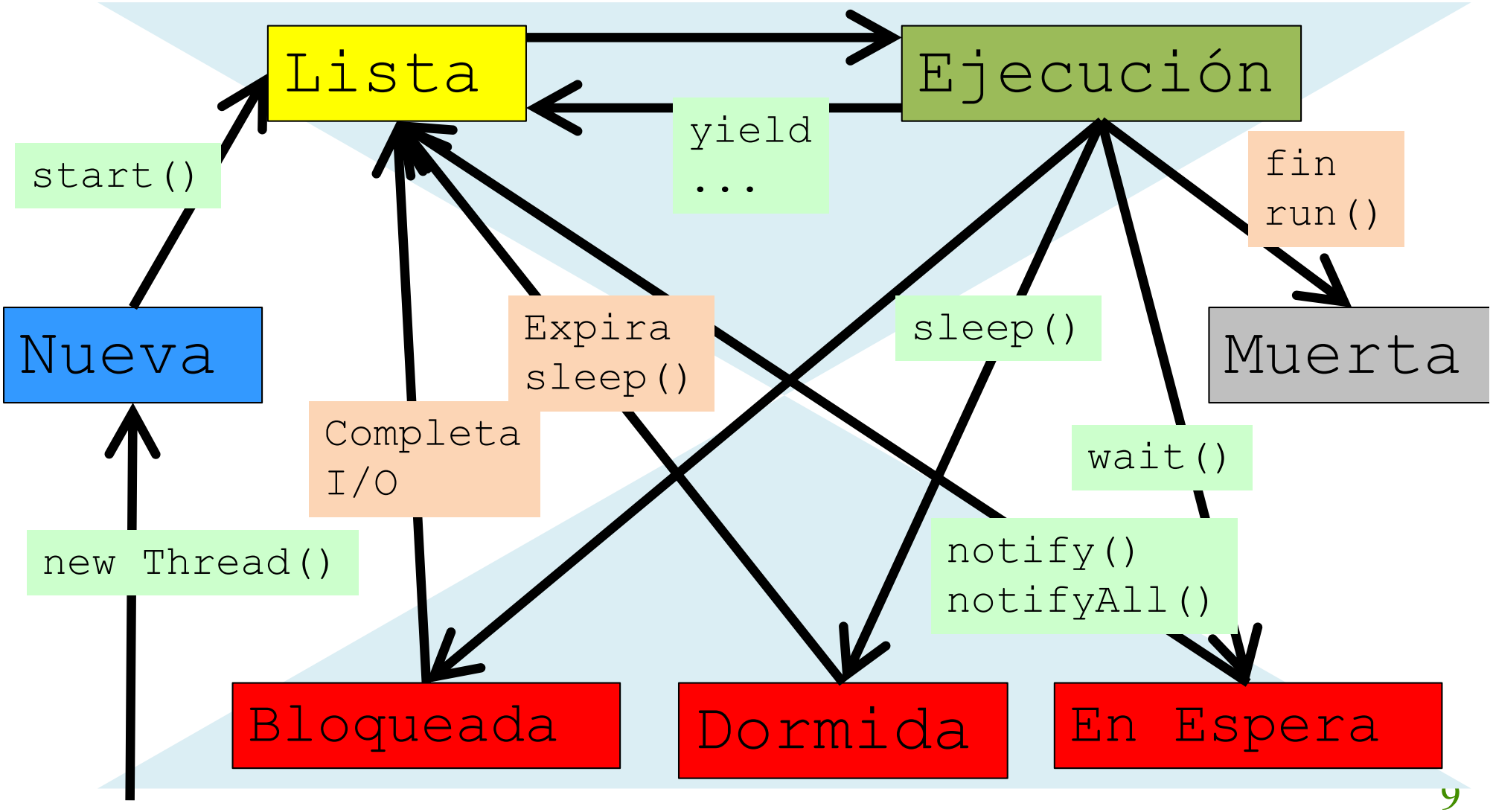
- Puede darse un nombre en la creación

```
Thread mt = new Thread(mc,"MiTarea");
String nombre = mt.getName();
```

Manejo de tareas

- **`void start()`**
  - Pone a la tarea en la cola de tareas listas para ejecutar
- **`static void sleep(int)`**
  - Duerme a una tarea un número de milisegundos. Cuando pasa ese tiempo, pasa al estado de lista.
    - Hay que capturar la interrupción **`InterrutedException`**
- **`static void yield()`**
  - Coloca a la tarea que está ejecutándose como última de su prioridad
- **`static Thread currentThread()`**
  - Devuelve la tarea que se está ejecutado ahora mismo
- **`void join()`**
  - Bloquea a quién lo llama hasta que finalice la tarea.
    - Hay que capturar la interrupción **`InterrutedException`**

Estados de una tarea



Cómo crear una tarea 1/2

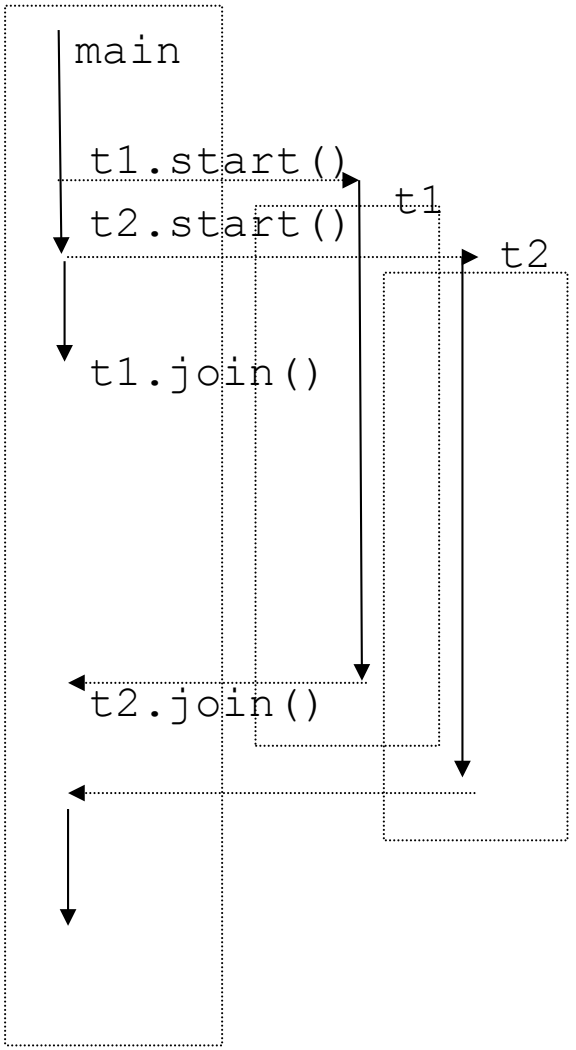
- Creamos una clase que cree las tareas y espere a que terminen.

```
package prTema7;
public class MainMiCodigo {
	public static void main(String [] args) {
		Thread tarea1 = new Thread(new MiCodigo(1000),"A");
		Thread tarea2 = new Thread(new MiCodigo(1000),"B");
		tarea1.start();
		tarea2.start();
		try {
			tarea1.join();
			tarea2.join();
		} catch (InterruptedException e) {
		}
	}
}
```

Ejemplo de tareas

```
class MiCodigo implements Runnable {
    int tope;
    public MiCodigo(int t) {
          tope = t;
    }
    public void run() {
          int i = 0;
          while (i < tope) {
             System.out.println(Thread.currentThread().getName()+i);
             i++;
          }
    }
}
```

```
public class LanzaderaMiCodigo {
    static public void main(String [] args) {
          MiCodigo mc1 = new MiCodigo(30);
          Thread   t1  = new Thread(mc1, "A");
          MiCodigo mc2 = new MiCodigo(50);
          Thread   t2  = new Thread(mc2, ”B");
          t1.start();
          t2.start();
          for (int i = 0 ; i < 20; i++) {
             System.out.println(“Ruido “+i);
          }
          try {
               t1.join();
               t2.join();
          } catch (InterruptedException e) {
          }
          System.out.println(“Fin“);
     }
}
```



Otra forma de crear una tarea

- Directamente heredando el código de la clase **`Thread`**

  ya que la clase **`Thread`** implementa la interfaz **`Runnable`**
  - Heredar
    - Ventajas
      - Los métodos de clase de **`Thread`** pueden llamarse sin hacer referencia a la clase (**`sleep`**, **`yield`**)
    - Inconvenientes
      - Ya no se puede heredar de otra clase
  - Interfaz
    - Ventajas
      - Es más lógico
      - Permite herencia de otra clase

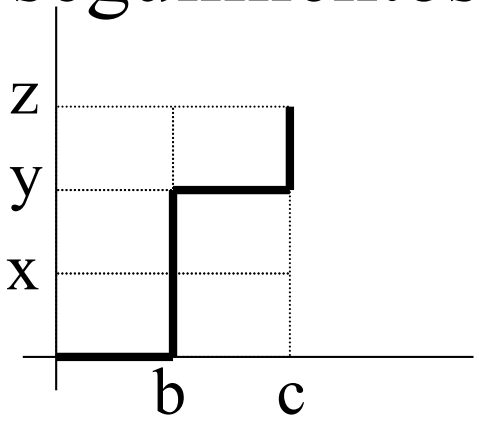
```
class MiTarea extends Thread {
      int tope;
      public MiTarea(int t, String nombre) {
            super(nombre);
            tope = t;
      }
      public void run() {
            int i = 0;
            while (i < tope) {
               System.out.println(
                     Thread.currentThread().getName()+i);
               i++;
            }
      }
}

class LanzaderaMiTarea {
      static public void main(String [] args) {
            Thread   t1  = new MiTarea(1000, "A");
            Thread   t2  = new MiTarea(1000, ”B");
            t1.start();
            t2.start();
            for (int i = 0 ; i < 1000; i++) {
               System.out.println(“Ruido “+i);
            }
            try {
                 t1.join();
                 t2.join();
            } catch (InterruptedException e) {
            }
            System.out.println(“Fin“);
       }
}
```

Interacciones entre tareas

- Sincronización
  - Relación de seguimientos
- Comunicación
  - Intercambio de datos
- Modo de interacción de tareas
  - Competencia: Uso exclusivo de recursos
    - Sincronización
  - Cooperación: Trabajos sobre partes de un problema.
    - Sincronización y Comunicación

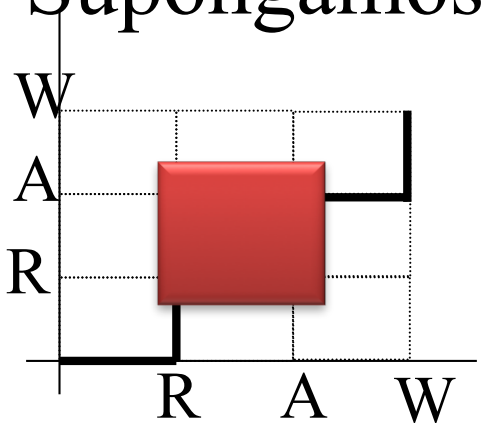
Sincronización de tareas

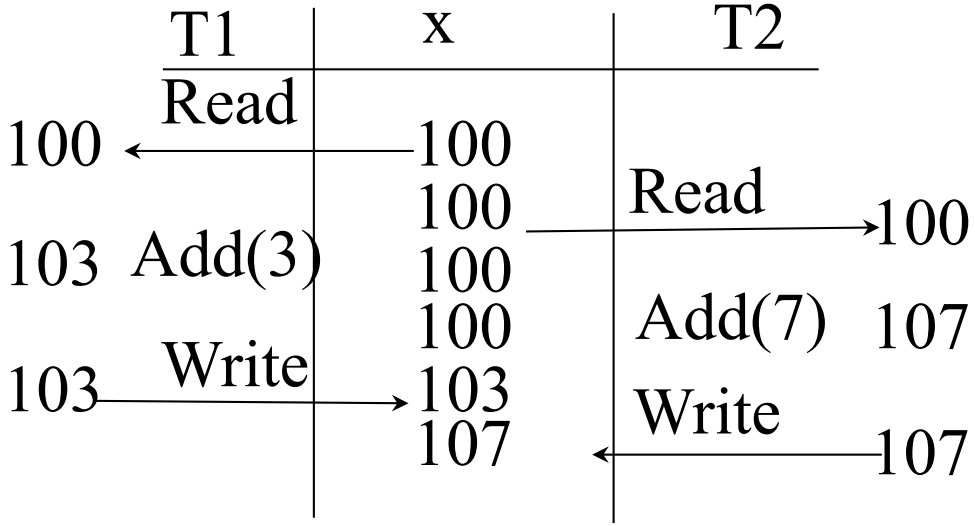
- Cuando dos tareas se ejecutan, necesitan repartirse el tiempo de la CPU
  - Sea A : b c
  - Sea W: x y z
- Relación de seguimientos



Ejemplo

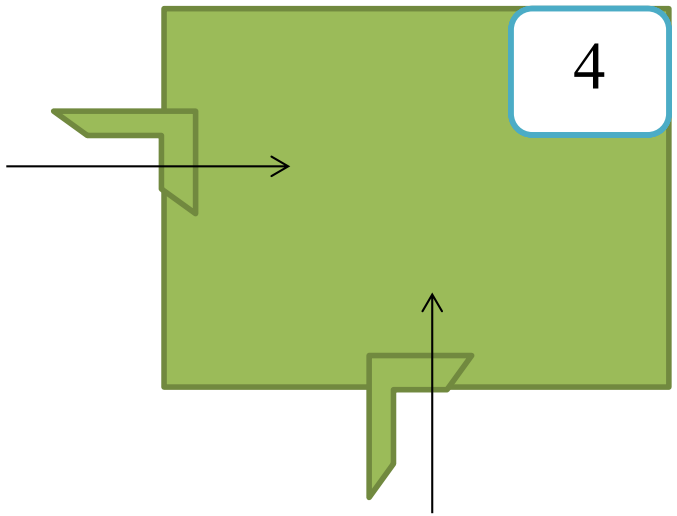
- Dos tareas quieren incrementar una variable x. La primera en 3 y la segunda en 7.
  - Sea T1: Read(x) Add(3) Write(x)
  - Sea T2: Read(x) Add(7) Write(x)
- Supongamos que la relación de seguimientos es





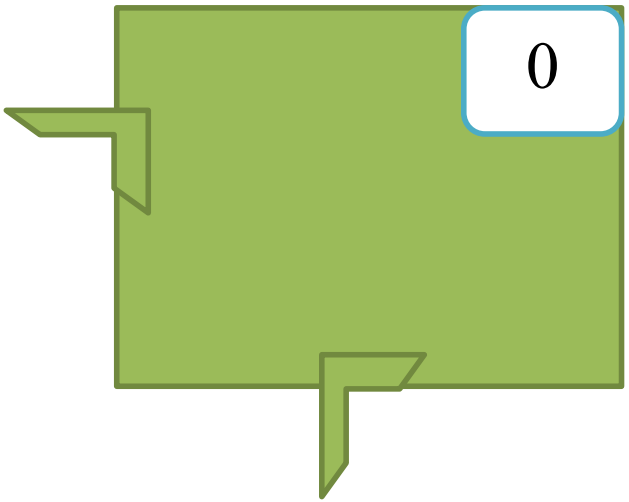
Sincronización de tareas

- El problema de la visita al jardín



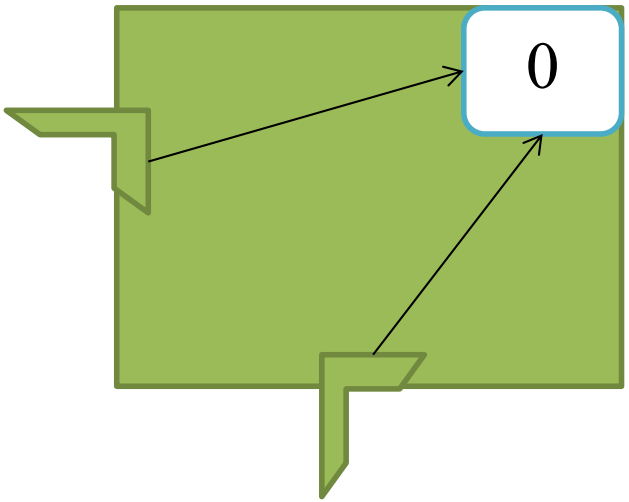
Sincronización de tareas

```
package prJardin;

public class Contador {
	private int contador;

	public Contador() {
		this.contador = 0;
	}

	public int valor() {
		return contador;
	}

	public  void incrementa() {
		contador++;
	}
}
```



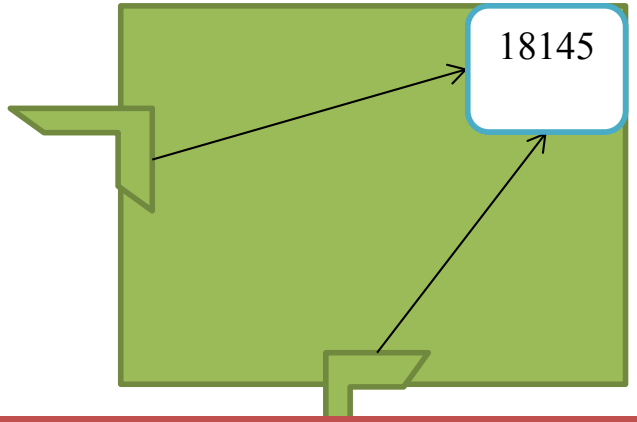
Sincronización de tareas

```
package prJardin;

public class Puerta implements Runnable {
	private Contador contador;
	private int visitas;

	public Puerta(Contador c, int v) {
		this.contador = c;
		this.visitas = v;
	}

	@Override
	public void run() {
		System.out.println(Thread.currentThread().getName()
				+ " arranca con visitas = " + visitas);
		for (int i = 0; i < visitas; i++) {
			contador.incrementa();
		}
	}
}
```



Sincronización de tareas

```
import prJardin.Contador;
import prJardin.Puerta;

public class Main {
          public static void main(String[] args) {
                    Contador c = new Contador();
                    int iteraciones = 10000;
                    Puerta p1 = new Puerta(c, iteraciones);
                    Puerta p2 = new Puerta(c, iteraciones);
                    Thread thP1 = new Thread(p1);
                    Thread thP2 = new Thread(p2);

                    long tIni = System.nanoTime();
                    thP1.start();
                    thP2.start();
                    try {
                              thP1.join();
                              thP2.join();

                    } catch (InterruptedException e) {
                              System.out.println(Thread.currentThread().getName()
                                        + " ha sido interrumpida.");
                    }
                    System.out.printf("Visitas por puerta: %,d\n", iteraciones);
                    System.out.printf("Contador: %,d\n", c.valor());
                    System.out.printf("Diferencia: %,d\n", c.valor() - 2 * iteraciones);
                    System.out.printf("Tiempo total %,d\n",
                                        (System.nanoTime() - tIni) / 1000);
          }
}
```




```
Thread-0 arranca con visitas = 10000
Thread-1 arranca con visitas = 10000
Visitas por puerta: 10.000
Contador: 18.145
Diferencia: -1.855
Tiempo total 31.529
```

Sincronización de tareas

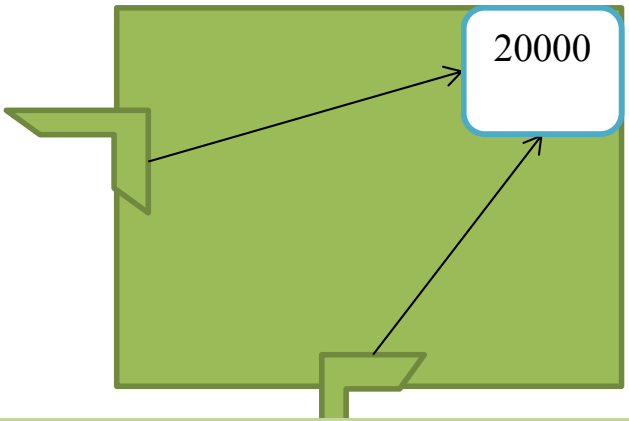
- Cada objeto dispone de un flag de bloqueo y una cola para tareas pendiente de tomar el flag de bloqueo
- Un objeto puede definir zonas de código sincronizadas.
  - Para acceder a ella se debe adquirir el flag de bloqueo
  - Sólo una tarea puede disponer del flag de bloqueo.
    - El resto de tareas que desean ese flag quedan bloqueadas y encoladas.
  - Cuando la tarea que posee el flag lo libera, la siguiente de la cola puede tomarlo.
- Se pueden sincronizar
  - un bloque de código (hay que indicar el objeto del que se desea obtener el flag)
  - o un método (bloquea el receptor)

Sincronización de tareas

```
package prJardin;

public class Contador {
	private int contador;

	public Contador() {
		this.contador = 0;
	}

	public int valor() {
		return contador;
	}

	synchronized public  void incrementa() {
		contador++;
	}
}
```

Sincronización de tareas

```
import prJardin.Contador;
import prJardin.Puerta;

public class Main {
          public static void main(String[] args) {
                    Contador c = new Contador();
                    int iteraciones = 10000;
                    Puerta p1 = new Puerta(c, iteraciones);
                    Puerta p2 = new Puerta(c, iteraciones);
                    Thread thP1 = new Thread(p1);
                    Thread thP2 = new Thread(p2);

                    long tIni = System.nanoTime();
                    thP1.start();
                    thP2.start();
                    try {
                              thP1.join();
                              thP2.join();

                    } catch (InterruptedException e) {
                              System.out.println(Thread.currentThread().getName()
                                        + " ha sido interrumpida.");
                    }
                    System.out.printf("Visitas por puerta: %,d\n", iteraciones);
                    System.out.printf("Contador: %,d\n", c.valor());
                    System.out.printf("Diferencia: %,d\n", c.valor() - 2 * iteraciones);
                    System.out.printf("Tiempo total %,d\n",
                                        (System.nanoTime() - tIni) / 1000);
          }
}
```




```
Thread-1 arranca con visitas = 10000
Thread-0 arranca con visitas = 10000
Visitas por puerta: 10.000
Contador: 20.000
Diferencia: 0
Tiempo total 38.958
```