

Mecanismos de sincronización

► MONITORES

- Conjunto de métodos atómicos que **proporcionan de forma sencilla exclusión mutua a un recurso**. Los métodos indicados permiten que cuando un hilo ejecute uno de los mismos, solamente ese hilo pueda estar ejecutando un método del monitor.
- Mientras que **un monitor no puede ser utilizado incorrectamente**, los semáforos dependen del programador ya que debe proporcionar la correcta secuencia de operaciones para no bloquear el sistema.
- Para utilizar un monitor en Java se utiliza la palabra clave ***synchronized*** sobre una región de código para indicar que se debe ejecutar como si de una sección crítica se tratase. Existen dos formas de utilizarlo:
 - **Métodos sincronizados.**
 - **Sentencias sincronizadas.**



Monitores. Métodos sincronizados. Cajero

```
Class cuenta{
    private int saldo;
    cuenta(int s) {saldo=s;} //Inicializa el saldo actual
    int getsaldo() {return saldo;} //devuelve el saldo
    void restar(int cantidad) { //se resta la cantidad al saldo
        saldo=saldo-cantidad; }
    void retirardinero(int cant, String nom){
        if (getsaldo() >=cant){
            System.out.println (" se retira saldo " + getsaldo() );
            try {
                Thread.sleep(500);
            }catch (InterruptedException ex) {}
            restar(cant);
            System.out.println(nom+ " retira " + cant + " actual(" + getsaldo() +")" );
        } else {
            System.out.println(nom+ " sin saldo suficiente " + getsaldo() );
        }
        if (getsaldo()<0) { System.out.println("Saldo negativo " + getsaldo() );}
    } //retirar dinero y Cuenta
}

Class sacardinero extends Thread{
    private cuenta c;
    String nom;
    public sacardinero(String n,Cuenta c){ super(n); this.c=c;}
    public void run() {
        for (int x=1;x<=4; x++) { c.retirardinero(10,getname()); } } //run
}
```

```
Public class compartirdinero{
    public static void main(String[] args){
        cuenta c=new cuenta(40);
        sacardinero h1= new sacardinero("Pepe", c);
        sacardinero h2=new sacardinero("Maria",c);
        h1.start();
        h2.start();
    }
}
```

Resultado???

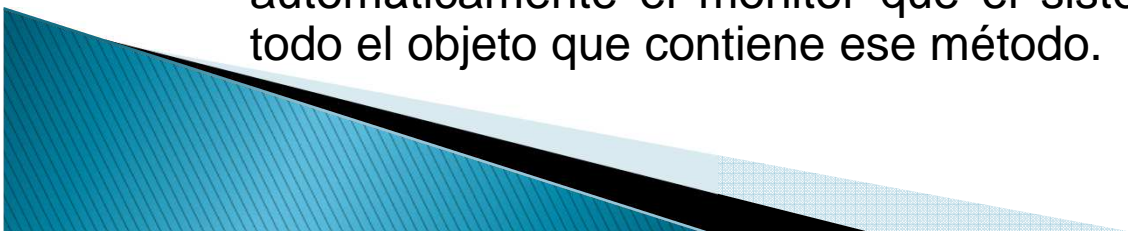
Mecanismos de sincronización. Monitores

▶ MÉTODOS SINCRONIZADOS

- Mecanismo para construir una sección crítica de forma sencilla. La ejecución por parte de un hilo de un método sincronizado de un objeto en Java imposibilita que se ejecute a la vez otro método sincronizado del mismo objeto por parte de otro hilo.
- Para crear un método sincronizado, solo es necesario añadir la palabra clave *synchronized* en la declaración del método, sabiendo que los constructores ya son síncronos por defecto.

```
public synchronized void increment() {  
    c++;  
}
```

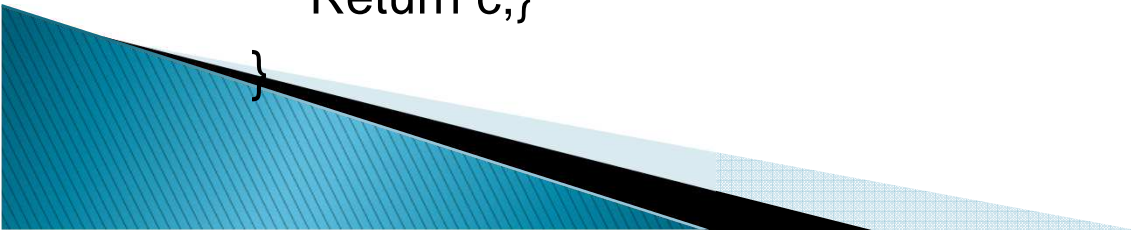
- Cuando un hilo invoca un método sincronizado, adquiere automáticamente el monitor que el sistema crea específicamente para todo el objeto que contiene ese método.



Mecanismos de sincronización. Monitores

► MÉTODOS SINCRONIZADOS

```
public class Contador{  
    private int c=0;  
    public void Contador(int num){  
        this.c=num;  
    }  
    public synchronized void increment(){  
        c++; }  
    public synchronized void decrement(){  
        c--; }  
    public synchronized int value(){  
        Return c;}  
}
```



Monitores. Métodos sincronizados. Cajero

```
Class cuenta{
    private int saldo;
    cuenta(int s) {saldo=s;} //Inicializa el saldo actual
    int getsaldo() {return saldo;} //devuelve el saldo
    void restar(int cantidad) { //se resta la cantidad al saldo
        saldo=saldo-cantidad; }
    void retirardinero(int cant, String nom){
        if (getsaldo() >=cant){
            System.out.println (" se retira saldo " + getsaldo() );
            try {
                Thread.sleep(500);
            }catch (InterruptedException ex) {}
            restar(cant);
            System.out.println(nom+ " retira " + cant + " actual(" + getsaldo() +")" );
        } else {
            System.out.println(nom+ " sin saldo suficiente " + getsaldo() );
        }
        if (getsaldo()<0) { System.out.println("Saldo negativo " + getsaldo() );}
    } //retirar dinero y Cuenta
}

Class sacardinero extends Thread{
    private cuenta c;
    String nom;
    public sacardinero(String n,Cuenta c){ super(n); this.c=c;}
    public void run() {
        for (int x=1;x<=4; x++) { c.retirardinero(10,getname()); } } //run
}
```

```
Public class compartirdinero{
    public static void main(String[] args){
        cuenta c=new cuenta(40);
        sacardinero h1= new sacardinero("Pepe", c);
        sacardinero h2=new sacardinero("Maria",c);
        h1.start();
        h2.start();
    }
}
```

Resultado???

Monitores. Métodos sincronizados. Cajero

```
Class cuenta{
    private int saldo;
    cuenta(int s) {saldo=s;} //Inicializa el saldo actual
    int getsaldo() {return saldo;} //devuelve el saldo
    void restar(int cantidad) { //se resta la cantidad al saldo
        saldo=saldo-cantidad; }
    synchronized void retirardinero(int cant, String nom){
        if (getsaldo() >=cant){
            System.out.println (" se retira saldo " + getsaldo() );
            try {
                Thread.sleep(500);
            }catch (InterruptedException ex) {}
            restar(cant);
            System.out.println(nom+ " retira " + cant + " actual(" + getsaldo() +")" );
        } else {
            System.out.println(nom+ " sin saldo suficiente " + getsaldo() );
        }
        if (getsaldo()<0) { System.out.println("Saldo negativo " + getsaldo() );}
    } //retirar dinero y Cuenta
}

Class sacardinero extends Thread{
    private cuenta c;
    String nom;
    public sacardinero(String n,Cuenta c){ super(n); this.c=c;}
    public void run() {
        for (int x=1;x<=4; x++) { c.retirardinero(10,getname()); } } //run
}
```

```
Public class compartirdinero{
    public static void main(String[] args){
        cuenta c=new cuenta(40);
        sacardinero h1= new sacardinero("Pepe", c);
        sacardinero h2=new sacardinero("Maria",c);
        h1.start();
        h2.start();
    }
}
```

Resultado???

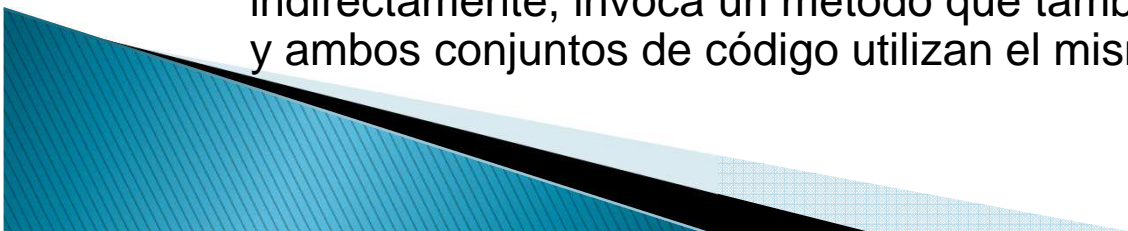
Mecanismos de sincronización. Monitores

▶ SENTENCIAS SINCRONIZADAS

- Permite una sincronización de grano fino al sincronizar únicamente una región específica de código.
- Esta funcionalidad permite especificar el objeto que proporciona el monitor en vez de ser el objeto por defecto que se está ejecutando como ocurre en métodos sincronizados.

```
public void increment() {  
    synchronized(objeto1) {  
        GlobalVar.c1++;  
    }  
}
```

- **Sincronización reentrante:** permitir que un hilo pueda adquirir un monitor que ya tiene. El hilo está ejecutando código sincronizado, que, directa o indirectamente, invoca un método que también contiene código sincronizado, y ambos conjuntos de código utilizan el mismo monitor.



Sentencias sincronizadas. Monitores

```
class GlobalVar{
public static int c1=0;
public static int c2=0;
}

class dosobjetos extends Thread {
private Object objeto1;
private Object objeto2;

dosobjetos(Object obj1,Object obj2){
objeto1=obj1;
objeto2=obj2;
}

public void inc1(){
synchronized(objeto1){
int tmp= GlobalVar.c1;
tmp++;
try{
sleep(1000);
} catch (InterruptedException e) {
System.out.println("Interrupci3n del hilo hijo");
}
GlobalVar.c1=tmp;
//GlobalVar.c1++;
} }

public void inc2(){
synchronized(objeto2){

int tmp= GlobalVar.c2;
tmp++;
try{
sleep(1000);
} catch (InterruptedException e) {
System.out.println("Interrupci3n del hilo hijo" );
}
GlobalVar.c2=tmp;
//GlobalVar.c2++;
} }
}
```

```
public void run(){
System.out.println ("soy el hilo " + this.getName() + " y
escribo en c1");
inc1();
System.out.println ("soy el hilo " + this.getName() + " y he
escrito en c1");

System.out.println ("soy el hilo " + this.getName() + " y
escribo en c2");
inc2();
System.out.println ("soy el hilo " + this.getName() + " y he
escrito en c2");
}
}

public class MonitorExclusionMutua {

public static void main(String[] args) throws
InterruptedException {
Object objeto1=new Object();
Object objeto2=new Object();
int N=4; //int N=Integer.parseInt(args[0]);
dosobjetos hilos[];
System.out.println ("Creando " + N + " hilos");

hilos= new dosobjetos[N];

for (int i=0;i<N;i++){
hilos[i]=new dosobjetos(objeto1,objeto2);
hilos[i].start();
}

for (int i=0;i<N;i++){
hilos[i].join();
}

System.out.println ("C1= " + GlobalVar.c1);
System.out.println ("C2= " + GlobalVar.c2);
}
}
```


Mecanismos de sincronización. Monitores

► CONDICIONES

- A veces el hilo que se encuentra dentro de una sección crítica no puede continuar, al no cumplirse una **condición**. Sin embargo, esta condición solo puede ser cambiada por otro hilo desde dentro de su correspondiente sección crítica.
 - Es necesario que el hilo que no puede continuar libere temporalmente el cerrojo que protege la sección crítica mientras espera a que alguien le avise de la ocurrencia de dicha condición.
- Para implementar condiciones se utiliza:
 - **wait**: El hilo espera hasta que otro hilo invoque notify o notifyAll()..
 - **notify**: implementa signal. Avisa de la ocurrencia de la condición por la que otro hilo espera Desbloquea un hilo que esté esperando en el método wait().
 - **notifyAll**. Avisa a todos los hilos en espera para que continuen su ejecución. Todos los hilos esperando en wait() reanudan su ejecución.



Monitores. Condiciones

- ▶ **wait()** : Una hebra que invoca el método wait() del objeto entonces queda suspendida hasta que alguien la despierte.
- ▶ **wait(tiempo)** Igual, pero se queda dormida un tiempo máximo.
- ▶ **notify()** Una hebra que tiene el cerrojo de un objeto puede invocar el método notify() del objeto y despierta una hebra suspendida en el objeto.
- ▶ **notifyAll()** Igual, pero despierta a todas las hebras.

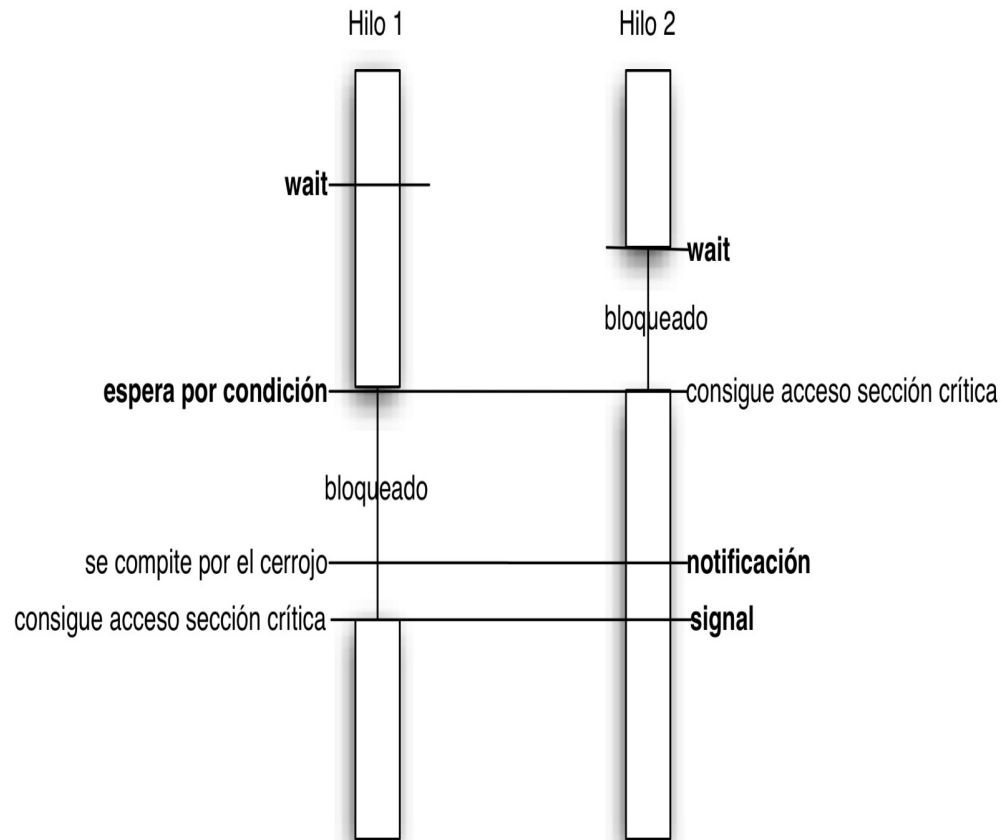


Clase Object

Las condiciones en Java se implementan utilizando la clase object. Estos métodos se deben ejecutar desde bloques sincronizados.

Método	Tipo Retorno	Descripción
wait()	void	Implementa la operación wait. El hilo espera hasta que otro hilo invoque notify o notifyAll().
notify()	void	Implementa la operación signal. Desbloquea un hilo que esté esperando en el método wait()
notifyAll()	void	Despierta a todos los hijos que estén esperando para que continúen con su ejecución. Todos los hilos esperando por wait reanudan su ejecución. Por ejemplo, si hay un proceso escritor, escribiendo datos, cuando finalice puede avisar a todos los procesos lectores para que continúen su ejecución a la vez (ya que pueden operar todos a la vez).

Mecanismos de sincronización. Monitores



```
synchronized public void  
comprobacion_ejecucion()
```

```
{  
    // Seccion critica  
    while (condicion) //no pueda continuar  
    {  
        wait();  
    }  
    // Seccion critica  
}
```

```
synchronized public void aviso_condicion()
```

```
{  
    // Seccion critica  
    if (condicion se cumple)  
        notify();  
    // Seccion critica  
}
```

Condiciones. Monitores. Caso Práctico

▶ Caso Práctico:

- Alumnos llegan a clase y esperan al profe.
- Profe llega, Saluda y comienza la clase.
- Los alumnos Saludan al profe.



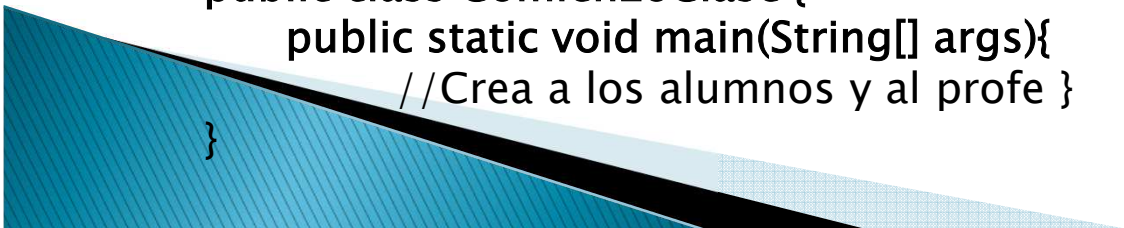
Condiciones. Monitores. Caso Práctico

```
class Bienvenida {  
    public Bienvenida(){  
        //Constructor Inicializa el estado la clase a sin comenzar  
        public synchronized void saludarProfesor (String nombreA)  
        //cuando el profe comience la clase  
  
        public synchronized void llegadaProfesor(String nombre){  
        //llega el profe saluda y comienza la clase
```

```
class Alumno extends Thread{  
    public Alumno (String nombre,Bienvenida mensajealumno){ }  
    public void run(){  
        //hilo de cada alumno para saludar }
```

```
class Profesor extends Thread{  
    public Profesor(String nombre,Bienvenida mensajeprofesor){ }  
    public void run(){  
        // Hilo del Profe cuando llega y saluda}  
    }
```

```
public class ComienzoClase {  
    public static void main(String[] args){  
        //Crea a los alumnos y al profe }  
    }
```



Condiciones. Monitores. Caso Práctico

```
class Bienvenida {
    boolean clase_comenzada;
    public Bienvenida(){
        this.clase_comenzada=false;
    }
    //hasta que el profe no salude no empieza la clase
    //por lo que los alumnos esperan con un wait
    public synchronized void saludarProfesor (String nombreA){
        try{
            while(clase_comenzada==false){
                wait();
            }
            System.out.println("Soy " + nombreA + ". Buenos días, profesor.");
        } catch (InterruptedException ex) {
            ex.printStackTrace();
        }
    }
    //Cuando el profe saluda avisa a los alumnos con notifyall de su
    llegada
    public synchronized void llegadaProfesor(String nombre){
        System.out.println("Buenos días a todos. soy el profe " + nombre);
        this.clase_comenzada=true;
        notifyAll();
    }
}

class Alumno extends Thread{
    String nombre;
    Bienvenida saludo;
    public Alumno (String nombre,Bienvenida mensajealumno){
        this.nombre=nombre;
        this.saludo = mensajealumno;
    }
}
```

Profe Saluda
Y Alumnos Saludan
Comienza la clase.

```
public void run(){
    System.out.println("Alumno" + this.nombre + " llega.");
    try{
        Thread.sleep(1000);
        saludo.saludarProfesor(nombre);
    } catch (InterruptedException ex){
        System.err.println ("Thread alumno interrumpido");
        System.exit(-1);
    }
}

class Profesor extends Thread{
    String nombre;
    Bienvenida saludo;

    public Profesor(String nombre,Bienvenida mensajeprofesor){
        this.nombre=nombre;
        this.saludo=mensajeprofesor;
    }

    public void run(){
        System.out.println(nombre + " llega.");
        try {
            Thread.sleep(1000);
            saludo.llegadaProfesor(nombre);
        } catch (InterruptedException ex){
            System.err.println ("Thread profesor interrumpido ");
            System.exit(-1);
        }
    }
}

public class ComienzoClase {
    public static void main(String[] args){
        //Objeto compartido
        Bienvenida b=new Bienvenida();
        String nombrealumno;
        int n_alumnos= 10; //int n_alumnos = Integer.parseInt(args[0]);
        for (int i=0; i<n_alumnos;i++){
            nombrealumno="alumno-"+i;
            new Alumno(nombrealumno,b).start();
        }
        Profesor profesor =new Profesor ("Jose Manuel", b);
        profesor.start();
    }
}
```


Programación de aplicaciones multihilo

- ▶ La **programación multihilo** permite la ejecución de varios hilos al mismo tiempo, tantos hilos como núcleos tenga el procesador, para realizar una tarea común.
- ▶ A la hora de realizar un programa multihilo cooperativo, se deben seguir las fases:
 - **Descomposición funcional.** Es necesario identificar previamente las diferentes tareas que debe realizar la aplicación y las relaciones existentes entre ellas.
 - **Partición.** La comunicación entre hilos se realiza principalmente a través de memoria. Los tiempos de acceso son muy rápidos por lo que no supone una pérdida de tiempo significativa. Ojo: Problemas de sincronización.
 - **Implementación.** Se utiliza la clase *Thread* o la interfaz *Runnable* como punto de partida. Utilizar mecanismos de sincronización.

