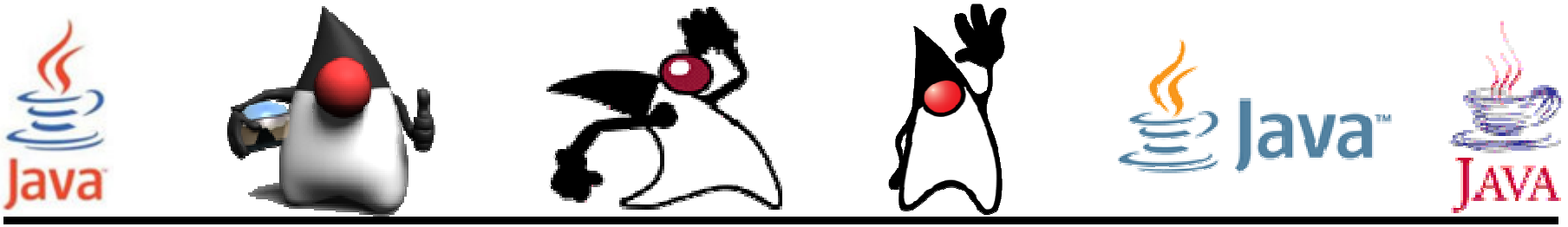


Threads (Hilos de ejecución)

Pedro Corcuera

Dpto. Matemática Aplicada y
Ciencias de la Computación

Universidad de Cantabria

corcuerp@unican.es

Objetivos

- Conocer los detalles de los hilos de ejecución (threads)
- Aprender a programar aplicaciones concurrentes basadas en threads

Índice

- Concepto de thread
- Estados de un Thread
- Prioridades de un Thread
- Clase Thread
- Formas de crear threads en Java
  - Extending Thread class
  - Implementing Runnable interface
- ThreadGroup
- Sincronización
- Comunicación Inter-thread
- Asignación de tarea mediante Timer y TimerTask

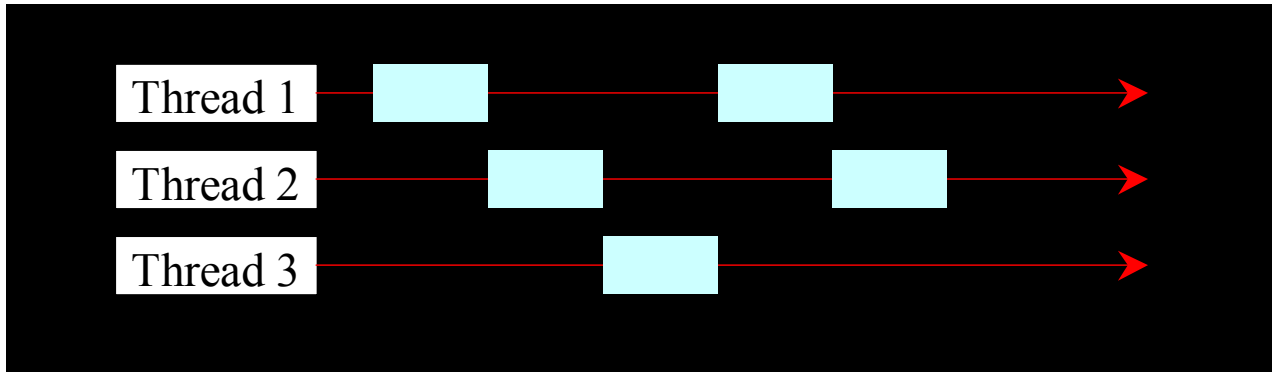
Concepto de Thread

- Un thread o hilo de ejecución es la ejecución secuencial de una serie de instrucciones dentro de un programa
- De forma simple se puede pensar que los threads son procesos ejecutados por un programa
- La mayoría de lenguajes de programación son single-threaded
- El desarrollo del hardware ha llevado el concepto de thread a la CPU: multinúcleo

Concepto de Thread

Threads múltiples en múltiples CPUs

Threads múltiples compartiendo una CPU



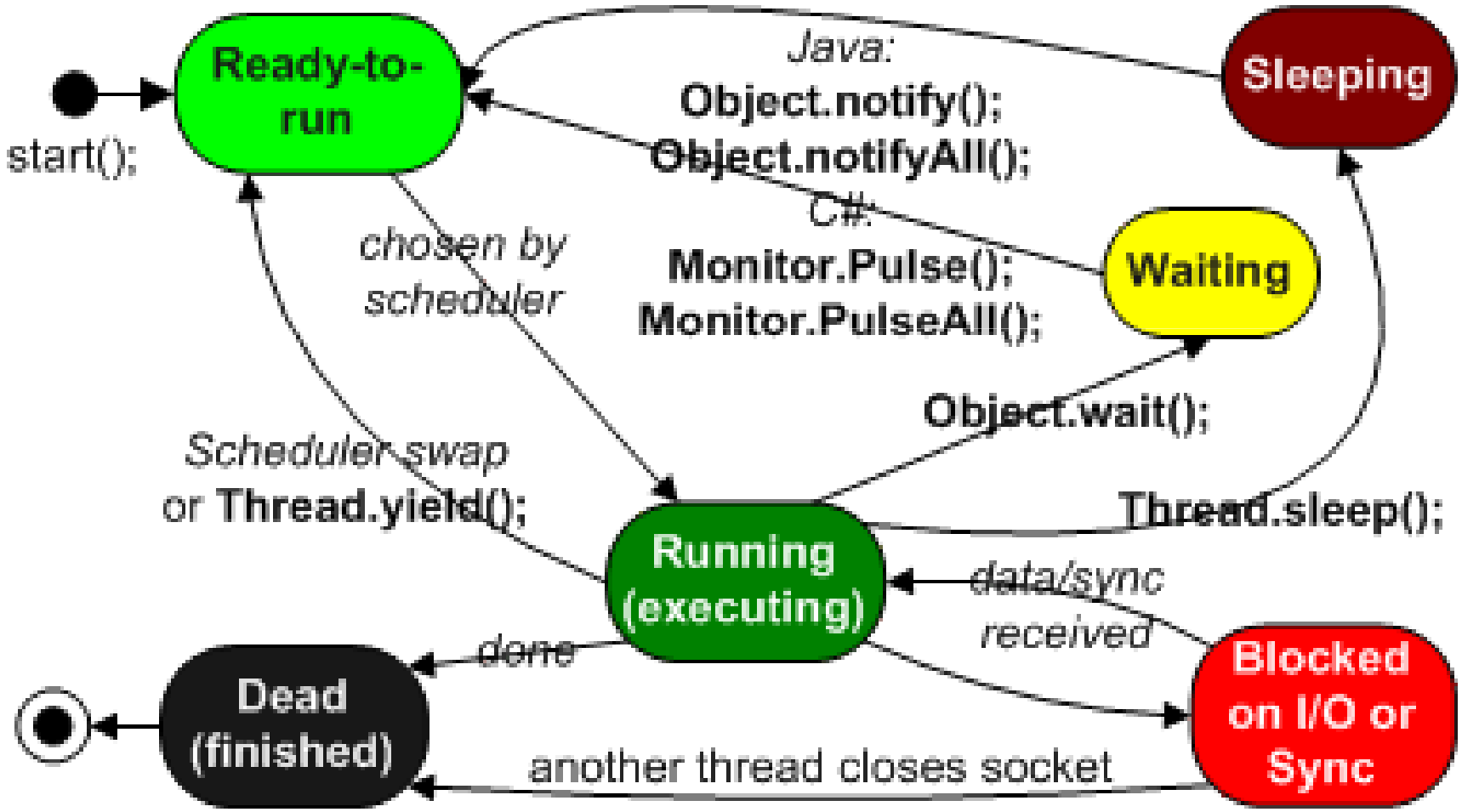
Multi-threading en la plataforma Java

- Cada aplicación tiene un thread, llamado el thread principal. Este thread tiene la habilidad de crear threads adicionales
- La programación multi-threaded es más compleja:
  - acceso compartido de objetos
  - riesgo de condición de carrera

Estados de un thread

- Un thread puede estar en uno de los siguientes estados:
  - Running
    - En curso de ejecución
    - En control de la CPU
  - Ready to run
    - Puede ejecutarse pero no se le ha dado la orden
  - Resumed
    - Ready to run después de haber estado suspendido o bloqueado
  - Suspended
    - Voluntariamente permite que otros threads se ejecuten
  - Blocked
    - Esperando por algún recurso o que ocurra un evento

Estados de un thread



Prioridades de un thread

- Las prioridades determinan que thread recibe el control de la CPU y consiga ser ejecutado primero
- En Java viene definidas por un valor entero de 1 a 10
- A mayor valor de prioridad, mayor la oportunidad de ser ejecutado primero
- Si dos threads tienen la misma prioridad, la ejecución depende del sistema operativo

Prioridades de un thread

Constante	Prioridad	Nombre del Thread
MAX_PRIORITY	10	Reference Handler
	8	Finalizer
	6	AWT-EventQueue-0, methods actionPerformed(), keyPressed(), mouseClicked(), y windowClosing().
NORM_PRIORITY	5	main
		Signal dispatcher
		AWT-Windows
		SunToolkit.PostEventQueue-0
	4	Screen Updater
MIN_PRIORITY	1	-

La clase Thread - Constructores

Resumen de Constructores

Thread()
Allocates a new Thread object.

Thread(Runnable target)
Allocates a new Thread object.

Thread(Runnable target, String name)
Allocates a new Thread object.

Thread(String name)
Allocates a new Thread object.

Thread(ThreadGroup group, Runnable target)
Allocates a new Thread object.

Thread(ThreadGroup group, Runnable target, String name)
Allocates a new Thread object so that it has target as its run object, has the specified name as its name, and belongs to the thread group referred to by group.

Thread(ThreadGroup group, Runnable target, String name, long stackSize)
Allocates a new Thread object so that it has target as its run object, has the specified name as its name, belongs to the thread group referred to by group, and has the specified *stack size*.

Thread(ThreadGroup group, String name)
Allocates a new Thread object.

La clase Thread - Constantes

Resumen de Campos

static int **MAX_PRIORITY**
The maximum priority that a thread can have.

static int **MIN_PRIORITY**
The minimum priority that a thread can have.

static int **NORM_PRIORITY**
The default priority that is assigned to a thread.

La clase Thread – algunos métodos

Method Summary

static Thread **currentThread**() Returns a reference to the currently executing thread object.
long **getId**() Returns the identifier of this Thread.
String **getName**() Returns this thread's name.
Thread.State **getState**() Returns the state of this thread.
ThreadGroup **getThreadGroup**() Returns the thread group to which this thread belongs.
void **interrupt**() Interrupts this thread.
void **run**() If this thread was constructed using a separate Runnable run object, then that Runnable object's run method is called; otherwise, this method does nothing and returns.
void **setContextClassLoader**(ClassLoader cl) Sets the context ClassLoader for this Thread.
void **setName**(String name) Changes the name of this thread to be equal to the argument name.
void **setPriority**(int newPriority) Changes the priority of this thread.
static void **sleep**(long millis) Causes the currently executing thread to sleep (temporarily cease execution) for the specified number of milliseconds, subject to the precision and accuracy of system timers and schedulers.
void **start**() Causes this thread to begin execution; the Java Virtual Machine calls the run method of this thread.

Formas de crear e iniciar un Thread

- Método A: Extendiendo la clase *Thread*
  - Derivar de la clase Thread
  - Override el método run()
  - Crear un thread con new MyThread(...)
  - Iniciar el thread invocando el método start()
- Método B: Implementando la interface *Runnable*
  - Implementar la interface Runnable
  - Override el método run()
  - Crear un thread con new Thread(runnable)
  - Iniciar el thread invocando el método start()

Extendiendo la clase Thread

- La subclase extiende la clase *Thread*
  - La subclase sobreescribe el método *run()* de la clase *Thread*
- Se puede crear un objeto instanciando la subclase
- Invocando el método *start()* del objeto instanciado inicia la ejecución del thread
  - El entorno runtime de Java inicia la ejecución del thread mediante la llamada del método *run()* del objeto

Esquemas de inicio de un thread desde una subclase

- Esquema 1: El método *start()* no está en el constructor de la subclase
  - El método *start()* requiere ser invocado explícitamente después de crear la instancia del objeto para que se inicie el thread
- Esquema 2: El método *start()* está en el constructor de la subclase
  - La creación de la instancia del objeto iniciará el thread

Ejemplo 1: método start() no está en el constructor

```
public class PrintNameThread extends Thread {
    PrintNameThread(String name) {
        super(name);
    }
    // Override el metodo run() de la clase Thread
    // se ejecuta cuando el metodo start() se invoca
    public void run() {
        System.out.println("metodo run() del thread "
                + this.getName() + " invocado" );
        for (int i = 0; i < 10; i++) {
            System.out.print(this.getName());
        }
    }
}
```

Ejemplo 1: método start() no está en el constructor

```
public class ExtendThreadClassTest1 {
    public static void main(String args[]) {
        System.out.println("Crea objeto PrintNameThread..");
        PrintNameThread pnt1 = new PrintNameThread("A");
        // Inicio del thread invocando el metodo start()
        System.out.println("Llamada del metodo start() de " +
                           pnt1.getName() + " thread");
        pnt1.start();
        System.out.println("Crea objeto PrintNameThread..");
        PrintNameThread pnt2 = new PrintNameThread("B");
        System.out.println("Llamada del metodo start() de " +
                           pnt2.getName() + " thread");
        pnt2.start();
    }
}
```

Ejemplo 2: método start() no está en el constructor

```
public class Contador1 extends Thread {
    protected int count, inc, delay;
    public Contador1( int  init,  int  inc,  int  delay ) {
      this.count = init; this.inc = inc; this.delay = delay;
    }
    public void run() {
        try {
            for (;;) {
                System.out.print(count + " ");
                count += inc; sleep(delay);
            }
        } catch (InterruptedException e) {}
    }
    public static void main(String[] args) {
        new Contador1(0, 1, 33).start();
        new Contador1(0, -1, 100).start();
    }
}
```

Ejemplo 3: método start() está en el constructor

```
public class PrintNameThread1 extends Thread {
    PrintNameThread1(String name) {
        super(name);
        // metodo start() dentro del constructor
        start();
    }
    public void run() {
        String name = getName();
        for (int i = 0; i < 10; i++) {
            System.out.print(name);
        }
    }
}
```

Ejemplo 3: método start() está en el constructor

```
public class ExtendThreadClassTest2 {
    public static void main(String args[]) {
        PrintNameThread1 pnt1 =
                new PrintNameThread1("A");
        PrintNameThread1 pnt2 =
                new PrintNameThread1("B");
    }
}
-------Versión sin instanciar objetos----------------
public class ExtendThreadClassTest2 {
    public static void main(String args[]) {
       new PrintNameThread1("A");
       new PrintNameThread1("B");
    }
}
```

Implementando la interface Runnable

- La interface *Runnable* debe implementarse por una clase cuyas instancias se intentan ejecutar como thread
- La clase debe definir el método run() sin argumentos
  - El método *run()* equivale a main() del nuevo thread
- Proporcionar los medios para que la clase sea activa sin derivar *Thread*
  - Una clase que implementa *Runnable* puede ejecutarse sin derivar de *Thread* instanciando un Thread y pasándolo como parámetro

Esquemas de inicio de un thread para una clase que implementa Runnable

- Esquema 1: El thread invocante crea un objeto Thread y lo inicia explícitamente después de crear un objeto de la clase que implementa la interface Runnable
  - El método *start()* del objeto Thread require ser invocado explícitamente despueés que el objeto se crea
- Esquema 2: El objeto Thread se crea y se inicia dentro del método constructor de la clase que implementa la interface Runnable
  - El thread invocante sólo necesita crear instancias de la clase Runnable

Ejemplo 1: creación de un objeto thread e inicio explícito

```
// La clase implementa la interface Runnable
class PrintNameRunnable implements Runnable {
    String name;
    PrintNameRunnable(String name) {
        this.name = name;
    }
    // Implementacion de run() definido en la
    // interface Runnable .
    public void run() {
        for (int i = 0; i < 10; i++) {
            System.out.print(name);
        }
    }
}
```

Ejemplo 1: creación de un objeto thread e inicio explícito

```
public class RunnableThreadTest1 {
   public static void main(String args[]) {
      PrintNameRunnable pnt1 = new
                                  PrintNameRunnable("A");
      Thread t1 = new Thread(pnt1);
      t1.start();
      PrintNameRunnable pnt2 = new
                                  PrintNameRunnable("B");
      Thread t2 = new Thread(pnt2);
      t2.start();
      PrintNameRunnable pnt3 = new
                                  PrintNameRunnable("C");
      Thread t3 = new Thread(pnt3); t3.start();
    }
}
```

Ejemplo 2: creación de objeto thread e inicio dentro del constructor

```
class PrintNameRunnable1 implements Runnable {
    Thread thread;
    PrintNameRunnable1(String name) {
        thread = new Thread(this, name);
        thread.start();
    }
    // Implementacion de run() definido en la
    // interface Runnable .
    public void run() {
        String name = thread.getName();
        for (int i = 0; i < 10; i++) {
            System.out.print(name);
        }
    }
}
```

Ejemplo 2: creación de objeto thread e inicio dentro del constructor

```
public class RunnableThreadTest2 {
    public static void main(String args[]) {
  // Como el constructor del objeto PrintNameRunnable
  // crea un objeto Thread y lo inicia,
  // no hay necesidad de hacerlo aqui.
        new PrintNameRunnable1("A");
        new PrintNameRunnable1("B");
        new PrintNameRunnable1("C");
    }
}
```

Comparativa extensión de Thread e implementación interface Runnable

- Seleccionar entre las dos formas de crear un thread es cuestión de gusto
- Atributos de la extensión de la clase Thread
  - Fácil de implementar
  - La clase no puede extender otra clase
- Atributos de implementar la interface Runnable
  - Puede tomar más trabajo debido a la declaración de Thread e invocación de los métodos Thread
  - La clase puede extender otras clases

Clase ThreadGroup

- Un grupo de threads representa un conjunto de threads
- Adicionalmente, un grupo de threads puede incluir otros grupos de threads
  - Los grupos de thread forman un árbol en el que cada grupo de treads excepto el inicial tiene un padre
- Dentro de un grupo se permite que un thread tenga acceso a la información del grupo, pero no a los demás grupos ni el del padre

Ejemplo Clase ThreadGroup

```
public class SimpleThread extends Thread {
    public SimpleThread(String str) {
        super(str);
    }
    public void run() {
        for (int i = 0; i < 5; i++) {
       // System.out.format("%d %s%n", i, getName());
            try {
                sleep((long)(Math.random() * 1000));
            } catch (InterruptedException e) {}
        }
        System.out.format("HECHO! %s%n", getName());
    }
}
```

Ejemplo Clase ThreadGroup

```
public class ThreadGroupTest {
    public static void main (String[] args) {
        // Inicia tres threads primero.
        new SimpleThread("Boston").start();
        new SimpleThread("New York").start();
        new SimpleThread("Seoul").start();
        // Obtiene un ThreadGroup y muestra numero
        ThreadGroup group = Thread.currentThread().getThreadGroup();
        System.out.println("Numero de threads activos en el grupo  = " +
                           group.activeCount());
        // Muestra nombres de threads en ThreadGroup.
        Thread[] tarray = new Thread[10];
        int actualSize = group.enumerate(tarray);
        for (int i=0; i<actualSize;i++){
            System.out.println("Thread " + tarray[i].getName()
                               + " en thread group " + group.getName());
        }

    }
}
```

Sincronización: condición de carrera

- Una condición de carrera ocurre cuando varios threads, que se ejecutan sin sincronizar, acceden al mismo objeto (llamado recurso compartido) retornando resultados inesperados (error)
- Ejemplo:
  - Con frecuencia los Threads comparten un recurso común, pe un fichero, con un thread leyendo del fichero y otro escribiendo en el fichero
- Esta situación se puede evitar sincronizando los threads que acceden al recurso común

Ejemplo de thread sin sincronizar

```
public class UnsynchronizedExample2 extends Thread {
    static int n = 1;
    public void run() {
        for (int i = 1; i <= 10; i++) {
            try {
                 Thread.sleep(500);
            } catch (InterruptedException ie) { }
            System.out.println(n);
            n++;
        }
    }
    public static void main(String args[]) {
        Thread thr1 = new UnsynchronizedExample2();
        Thread thr2 = new UnsynchronizedExample2();
        thr1.start();
        thr2.start();
    }
}
```

Sincronización: bloqueando un objeto

- La operación de sincronización es un mecanismo de exclusión mutua de threads, es decir, no puede ser interrumpida (operación atómica)
- Un thread se sincroniza cuando se convierte en propietario del monitor del objeto (bloqueo)
- Formas para bloquear (sincronizar) un thread
  - Opción 1: Usar el método *synchronized*
  - Opción 2: Usar la sentencia *synchronized* sobre un objeto común

Sincronización: bloqueando un objeto

- Método sincronizado:

```
class MyClass{
    synchronized void aMethod(){
        statements
    }
}
```

- Bloque sincronizado :

```
synchronized(exp){
    statements
}
```

Ejemplo de threads sincronizados

```
public class SynchronizedExample2 extends Thread {
    static int n = 1;
    public void run() { syncmethod(); }
    synchronized static void syncmethod() {
        for (int i = 1; i <= 10; i++) {
              try {
                 Thread.sleep(500);
              } catch (InterruptedException ie) {
              }
            System.out.println(n);  n++;
        }
    }
    public static void main(String args[]) {
        Thread thr1 = new SynchronizedExample2();
        Thread thr2 = new SynchronizedExample2();
        thr1.start();  thr2.start();
    }
}
```

Ejemplo de threads sincronizados

```
public class SynchronizedExample1 extends Thread {
    Operation ope ;
    SynchronizedExample1 (Operation op){
      this.ope = op;
    }
    public void run() {
       synchronized (ope) {
           ope.method();
       }
    }
    public static void main(String args[]) {
        Operation op = new Operation();
        Thread thr1 = new SynchronizedExample1(op);
        Thread thr2 = new SynchronizedExample1(op);
        thr1.start();
        thr2.start();
    }
}
```

Ejemplo de threads sincronizados

```
class Operation {
   static int n = 1;
   void method() {
     for (int i = 1; i <= 10; i++) {
           try {
              Thread.sleep(250);
           } catch (InterruptedException ie) {
           }
         System.out.println(n);
         n++;
     }
  }
}
```

Métodos de la clase Object para la comunicación Inter-thread

Resumen de métodos
void notify() Wakes up a single thread that is waiting on this object's monitor.
void notifyAll() Wakes up all threads that are waiting on this object's monitor.
void wait() Causes the current thread to wait until another thread invokes the notify() method or the notifyAll() method for this object.
void wait(long timeout) Causes the current thread to wait until either another thread invokes the notify() method or the notifyAll() method for this object, or a specified amount of time has elapsed.
void wait(long timeout, int nanos) Causes the current thread to wait until another thread invokes the notify() method or the notifyAll() method for this object, or some other thread interrupts the current thread, or a certain amount of real time has elapsed.

Métodos wait(), notify(), and notifyAll()

- Para facilitar la comunicación entre threads se usan los métodos wait(), notify(), y notifyAll() de la clase Object
- Estos métodos deben ser invocados en un método o bloque sincronizado del objeto invocante de los métodos y deben usarse dentro de un bloque try/catch
- El método wait() provoca que el thread espere hasta que ocurra alguna condición, en cuyo caso se usan los métodos notify() o notifyAll() para notificar a los threads en espera para continuar con la ejecución

Método wait() de la Clase Object

- El método wait() provoca que un thread libere el bloqueo que mantiene sobre un objeto, permitiendo que otro thread se ejecute
- El método wait() está definido en la clase Object
- wait() sólo se puede invocar desde el interior de un código sincronizado y siempre debe estar en un bloque try porque genera IOExceptions
- wait() sólo puede ser invocado por un thread que es propietario del bloqueo del objeto

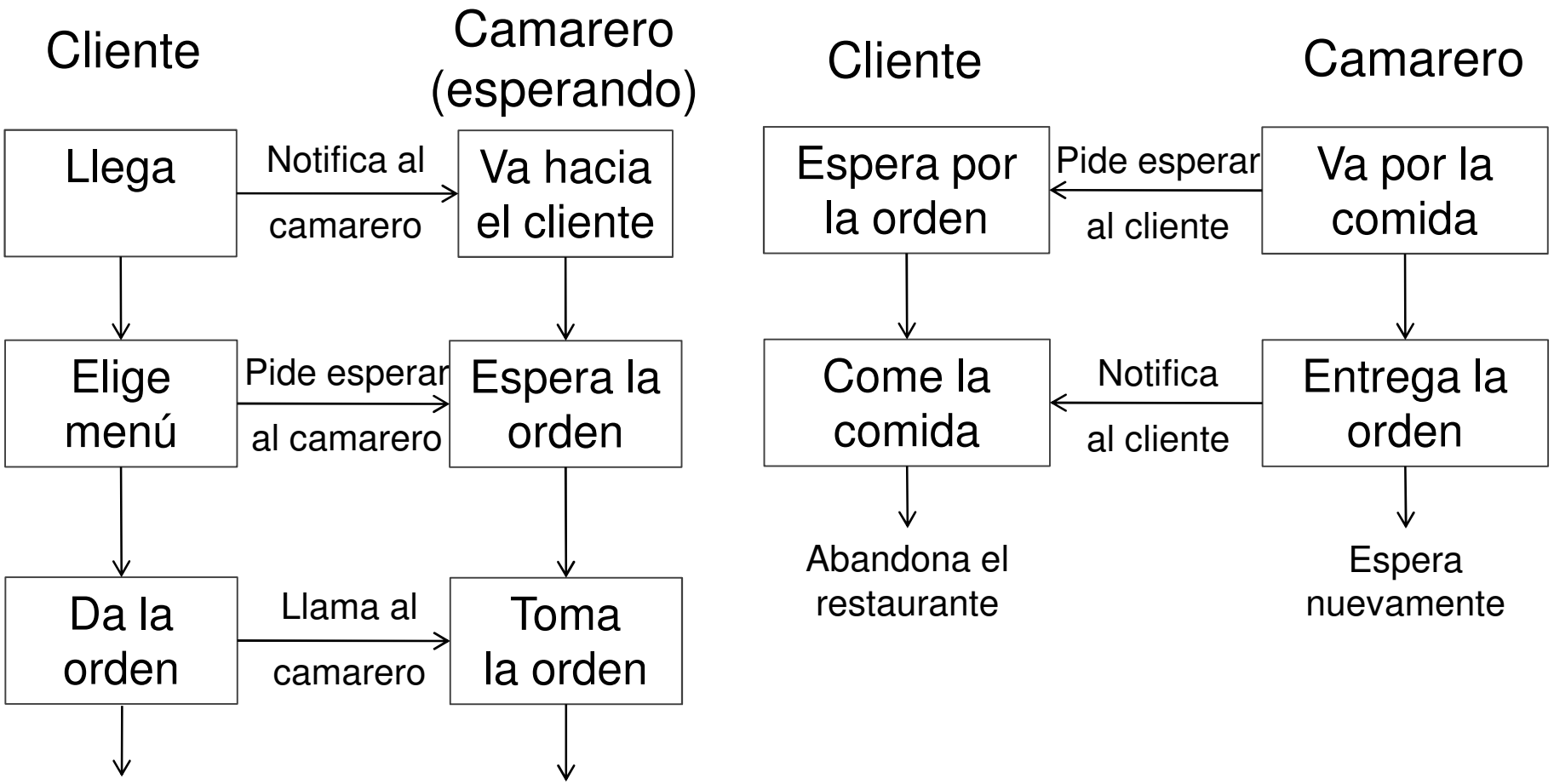
Método wait() de la Clase Object

- Cuando se invoca el método wait(), el thread se incapacita para el planificador de tareas y permanece inactivo hasta que ocurre uno de los eventos:
  - otro thread invoca el método notify() para este objeto y el planificador escoje arbitrariamente la ejecución del thread
  - otro thread invoca el método notifyAll() para este objeto
  - otro thread interrumpe este thread
  - el tiempo especificado por wait() ha transcurrido
- Después, el thread pasa a estar disponible otra vez y compite por el bloqueo del objeto y si lo consigue continúa como si no hubiera ocurrido la suspensión

Método notify()

- Despierta un único thread que está esperando por este monitor del objeto
  - Si varios threads esperan por este objeto, uno de ellos es seleccionado para ser despertado
  - La selección es arbitraria y ocurre a discreción de la implementación
- Sólo puede usarse dentro de un código sincronizado
- El thread despertado no podrá continuar hasta que el thread en curso renuncie al bloqueo sobre el objeto

Comunicación entre threads: Ejemplo práctico



Comunicación entre threads: Ejemplo Productor-Consumidor

- Suponer un escenario en el que existe dos threads distintos operando ambos sobre una única área de datos compartida
- Un thread, el Productor, inserta información en el área de datos mientras que el otro thread, el Consumidor, extrae información de la misma área
- Para que el Productor inserte información en el área de datos debe haber suficiente espacio
- La única función del Productor es insertar datos en el área de datos, no elimina ningún dato

Comunicación entre threads: Ejemplo Productor-Consumidor

- Para que el Consumidor pueda extraer información debe existir información en el área de datos
- La única función del Consumidor es extraer datos desde el área de datos
- La solución del problema Productor-Consumidor se consigue elaborando un adecuado protocolo de comunicación entre los threads para intercambiar información y es el principal factor para que este problema sea interesante en términos de sistemas concurrentes

Productor-Consumidor sin sincronizar
Area de datos: CubbyHole.java

```
public class CubbyHole {
    private int contents;
    private boolean available = false;

    public int get() {
        available = false;
        return contents;
    }

    public void put(int value) {
        contents = value;
        available = true;
    }
}
```

Productor-Consumidor sin sincronizar
Producer.java

```
public class Producer extends Thread {
    private CubbyHole cubbyhole;
    private int number;

    public Producer(CubbyHole c, int number) {
        cubbyhole = c;
        this.number = number;
    }

    public void run() {
      for (int i = 0; i < 10; i++) {
        cubbyhole.put(i);
        System.out.println("Producer #" + this.number
                                + " put: " + i);
        try {
              sleep((int)(Math.random() * 100));
            } catch (InterruptedException e) { }
        }
    }
}
```

Productor-Consumidor sin sincronizar
Consumer.java

```
public class Consumer extends Thread {
    private CubbyHole cubbyhole;
    private int number;

    public Consumer(CubbyHole c, int number) {
        cubbyhole = c;
        this.number = number;
    }

    public void run() {
        int value = 0;
        for (int i = 0; i < 10; i++) {
            value = cubbyhole.get();
            System.out.println("Consumer #" + this.number
                                + " got: " + value);
        }
    }
}
```

Productor-Consumidor sin sincronizar
Programa principal

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

        CubbyHole c = new CubbyHole();

        Producer p1 = new Producer(c, 1);
        Consumer c1 = new Consumer(c, 1);

        p1.start();
        c1.start();
    }
}
```

Productor-Consumidor sin sincronizar Resultado

```
>java  ProducerConsumerUnsynchronized
Producer #1 put: 0
Consumer #1 got: 0
Consumer #1 got: 0
Consumer #1 got: 0
Consumer #1 got: 0
Consumer #1 got: 0
Consumer #1 got: 0
Consumer #1 got: 0
Consumer #1 got: 0
Consumer #1 got: 0
Consumer #1 got: 0
Consumer #1 got: 0
Producer #1 put: 1
Producer #1 put: 2
Producer #1 put: 3
Producer #1 put: 4
Producer #1 put: 5
Producer #1 put: 6
Producer #1 put: 7
Producer #1 put: 8
Producer #1 put: 9
```

Los resultados son impredecibles

– Se puede leer un número antes de que se haya producido

– Se pueden producir varios números con solo uno o dos leídos

Productor-Consumidor sincronizado
Area de datos: CubbyHole.java

```
public class CubbyHole {
    private int contents;
    private boolean available = false;

    public synchronized int get(int who) {
        while (available == false) {
            try {
                wait();
            } catch (InterruptedException e) { }
        }
        available = false;
        System.out.format("Consumer %d got: %d%n", who,
                          contents);
        notifyAll();
        return contents;
    }
```

Productor-Consumidor sincronizado
Area de datos: CubbyHole.java

```
    public synchronized void put(int who, int value) {
        while (available == true) {
            try {
                wait();
            } catch (InterruptedException e) { }
        }
        contents = value;
        available = true;
        System.out.format("Producer %d put: %d%n", who,
                          contents);
        notifyAll();
    }
}
```

Productor-Consumidor sincronizado
Programa principal

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

        CubbyHole c = new CubbyHole();

        Producer p1 = new Producer(c, 1);
        Consumer c1 = new Consumer(c, 1);

        p1.start();
        c1.start();
    }
}
```

Productor-Consumidor sincronizado Resultado

```
>java  ProducerConsumerSynchronized
Producer 1 put: 0
Consumer 1 got: 0
Producer 1 put: 1
Consumer 1 got: 1
Producer 1 put: 2
Consumer 1 got: 2
Producer 1 put: 3
Consumer 1 got: 3
Producer 1 put: 4
Consumer 1 got: 4
Producer 1 put: 5
Consumer 1 got: 5
Producer 1 put: 6
Consumer 1 got: 6
Producer 1 put: 7
Consumer 1 got: 7
Producer 1 put: 8
Consumer 1 got: 8
Producer 1 put: 9
Consumer 1 got: 9
```

Asignación de tareas mediante Clases Timer y TimerTask

- La clase Timer facilita a los threads la programación de ejecución futura de tareas en un thread de fondo
- La tareas se pueden programar para ejecutarse una vez o de forma repetida en intervalos regulares
- A cada objeto Timer le corresponde un thread de fondo que se usa para ejecutar todas las tareas del timer secuencialmente
- Los tiempos de las tareas deben completarse rápidamente

Asignación de tareas mediante Clases Timer y TimerTask

- La clase TimerTask es una clase abstracta con un método abstracto llamado run()
- La clase concreta debe implementar el método abstracto run()

Ejemplo de uso
Clases Timer y TimerTask

```
import java.util.Timer;
import java.util.TimerTask;

/** Demo simple que usa java.util.Timer para ejecutar una tarea en 5 segundos */
public class TimerReminder {
    Timer timer;

    public TimerReminder(int seconds) {
        timer = new Timer();
        timer.schedule(new RemindTask(), seconds*1000);
    }
    class RemindTask extends TimerTask {
        public void run() {
            System.out.println("Se acabo el tiempo!");
            timer.cancel(); //Termina el tiempo del thread
        }
    }
    public static void main(String args[]) {
        System.out.println("Se programa la tarea ReminderTask para 5 segundos");
        new TimerReminder(5);
        System.out.println("Tarea programada.");
    }
}
```

Problemas a resolver en la programación concurrente

- La programación concurrente debe evitar problemas que pueden surgir entre los recursos disponibles por los threads:
  - Acaparamiento (starvation)
  - Inactividad (dormancy)
  - Abrazo mortal (deadlock)
  - Terminación prematura
- Para ello debe usarse correctamente la sincronización