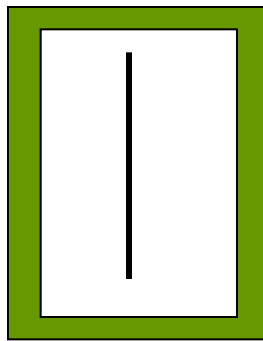




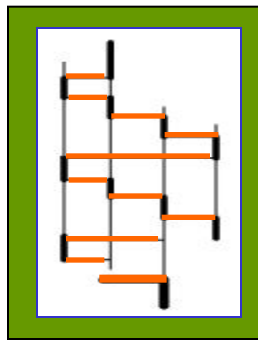
Concurrencia Threads

Threads

- Un **thread** es un flujo de control secuencial dentro de un proceso. A los threads también se los conoce como **procesos livianos** (requiere menos recursos crear un thread nuevo que un proceso nuevo) ó **contextos de ejecución**.
- Un **thread** es similar a un programa secuencial: tiene un comienzo, una secuencia de ejecución, un final y en un instante de tiempo dado hay un único punto de ejecución. Sin embargo, un thread no es un programa. Un **thread** se ejecuta adentro de un programa.
- Lo novedoso en **threads** es el uso de múltiples threads adentro de un mismo programa, ejecutándose simultáneamente y realizando tareas diferentes:



Programa *singleThread*



Programa *multiThread*

Thread en ejecución
Transferencia del control
Thread *bloqueado*



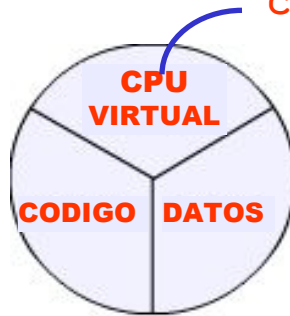
En un programa *multithread*, más de un thread se ejecuta en forma concurrente. El control de ejecución es transferido entre los diferentes threads, cada uno de los cuáles es responsable de distintas tareas.

- En el modelo de **multithreading** la CPU asigna a cada thread un tiempo para que se ejecute; cada thread “tiene la percepción” que dispone de la CPU constantemente, sin embargo el tiempo de CPU está dividido entre todos los threads.

Threads

- Un **thread** se ejecuta dentro del contexto de un programa (o proceso) y comparte los recursos asignados al programa. A pesar de esto, los **threads** toman algunos recursos del ambiente de ejecución del programa como propios: tienen su propia pila de ejecución, contador de programa, código y datos. Como un **thread** solamente se ejecuta dentro de un contexto, a un thread también se lo llama **contexto de ejecución**.
- La plataforma JAVA soporta programas **multithreading** a través del lenguaje, de librerías y del sistema de ejecución. A partir de la versión 5.0, la plataforma JAVA incluye librerías de concurrencia de más alto nivel

Múltiples-threads comparten el mismo código, cuando se ejecutan a partir de instancias de la misma clase.



CPU virtual que ejecuta código y utiliza datos

Múltiples-threads comparten datos, cuando acceden a objetos comunes (podría ser a partir de códigos diferentes).

- La **clase Thread** forma parte del paquete **java.lang** y provee una implementación de threads independiente del sistema de ejecución. Hay dos estrategias para usar objetos **Threads**:
 - **Directamente controlar la creación y el gerenciamiento** instanciando un Thread cada vez que la aplicación requiere iniciar una tarea concurrente.
 - **Abstraer el gerenciamiento** de threads pasando la tarea concurrente a un **ejecutor** para que la administre y ejecute.

Creación y Gerenciamiento de Threads

- La **clase Thread** provee el comportamiento genérico de los threads JAVA: arranque, ejecución, interrupción, asignación de prioridades, etc.
- El método **run()** es el más importante de la clase **Thread**, implementa la funcionalidad del thread, es el código que se ejecutará “simultáneamente” con otros threads del programa. El método **run()** predeterminado provisto por la clase Thread no hace nada.
- La plataforma JAVA es **multithread**: siempre hay un thread ejecutándose junto con las aplicaciones de los usuarios, por ejemplo el **garbage collector** es un thread que se ejecuta en background; las GUI's **recolectan los eventos** generados por el usuario en threads separados, etc..
- Una aplicación JAVA siempre se ejecuta en un **thread**, llamado **main thread**. Este **thread** ejecuta secuencialmente las sentencias del cuerpo del método **main()** de la clase. En otros programas JAVA, como **applets** y **servlets**, que no tienen método **main()**, la ejecución del **main thread** comienza con el método **main()** de su contenedor, que es el encargado de invocar los métodos del ciclo de vida de dichas componentes.

El método run() de la clase Thread

Es un método estándar de la clase **Thread**. Es el lugar donde el **thread** comienza su ejecución.

```
public class SimpleThread extends Thread {
    private int contador=10;
    public SimpleThread(int nro) {
        super("" +nro); —Invoca al constructor de
                        Thread con un argumento String
                        que es el nombre del thread
    }
    public String toString() {
        return "#" + getName() + ":" + contador--;
    }

    public void run() {
        for (int i = contador; i > 0; i --)
            System.out.println(this);

        System.out.println("Termino!" + this );
    }
}

Un thread termina cuando finaliza
el método run()
```

```
public class TestCincoThread {
    private static int nroThread=0;
    public static void main(String[] args) {
        for (int i=0; i<5;i++)
            new SimpleThread(++ nroThread).start();
    }
}
```

Inicializa el objeto Thread e invoca al método run(). El thread pasa a estado "vivo"

Se recupera el nombre del thread con el método getName() de la clase Thread

Cuando el método start() retorna, hay 2 threads ejecutándose en paralelo: el thread que invocó al start(), en nuestro caso el main thread y el thread que está ejecutando el método run().

Tenemos 5 tareas concurrentes, cada una de ellas imprime en pantalla 10 veces su nombre. Además tenemos el **main thread**.

¿Cuál es la salida del programa TestCincoThread?

La salida de una ejecución del programa podría ser diferente a la salida de otra ejecución del mismo programa, ya que el mecanismo de **scheduling** de threads no es determinístico.

Sleep Métodos de la clase Thread

Suspende temporariamente la ejecución del **thread** que se está ejecutando. Afecta solamente al **thread** que ejecuta el **sleep()**, no es posible decirle a otro thread que "se duerma". Es un método de clase. El tiempo de suspensión se expresa en milisegundos.

```
import java.util.concurrent.TimeUnit;
public class SimpleThread extends Thread {
    private int contador=10;
    public SimpleThread(int nro) {
        super("" +nro);
    }
    public String toString() {
        return "#" + getName()+ ":" + contador--;
    }
    public void run() {
        for (int i = contador; i > 0; i --){
            System.out.println(this);
            try {
                //Antes de JSE 5:
                //sleep(100);
                //Estilo JSE 5:
                TimeUnit.MILLISECONDS.sleep(100);
            } catch (InterruptedException e){
                throw new RuntimeException();
            }
            System.out.println("Termino!" + this );
        }
    }
}
```

- El método **sleep()** está encerrado en un bloque **try/catch**, dado que el thread podría recibir una solicitud de interrupción antes que el tiempo se agote (se invoca al método **interrupt()** sobre el objeto thread).
- sleep()** es un método sobrecargado, que permite especificar el tiempo de espera en milisegundos y en nanosegundos. En la mayoría de las implementaciones de la JVM, este tiempo se redondea a la cantidad de milisegundos más próxima (en general un múltiplo de 20 milseg o 50 milseg).
- Los threads se ejecutan en cualquier orden. El método **sleep()** no permite controlar el orden de ejecución de los threads; suspende la ejecución del thread por un tiempo dado.
- En nuestro ejemplo, la única garantía que se tiene es que el thread suspenderá su ejecución por al menos 100 milisegundos, pero podría tomar más tiempo antes de retomar la ejecución.

Join

Métodos de la clase Thread

El método **join()** permite que un **thread** espere a que otro termine de ejecutarse. El objetivo del método **join()** es esperar por un evento específico: la terminación de un **thread**. El **thread** que invoca al **join()** sobre otro **thread** se bloquea hasta que dicho **thread** termine su método **run()**. Una vez que el **thread** completa el **run()**, el método **join()** retorna inmediatamente.

```
public class SimpleThreadTest2 {
    public static void main(String args[]) {
        SimpleThread t=new SimpleThread(1);
        t.start();
        while (t.isAlive()) {
            System.out.println("esperando...");
            try {
                t.join();
            } catch (InterruptedException e) {
                System.out.println(getName() + "join
interrumpido");
            }
            System.out.println(getName() + " join
completado");
        }
    } // fin del while
}
```

El main thread se suspende hasta que el thread t termine (isAlive() devuelve false)

```
thrd.start();
while (thrd.isAlive()){
    try{
        thrd.join(2000);
        System.out.print(".");
    } catch (InterruptedException e) { }
} System.out.println(" Listo!");
```

Este segmento de código inicia al thread thrd y cada 2 segundos imprime un "." mientras thrd continúa ejecutándose

- En este caso el **main thread** se bloquea en espera que el **thread t** termine de ejecutarse.
- El método **join()** es sobrecargado, permite especificar el tiempo de espera. Sin embargo, de la misma manera que el **sleep()**, no se puede asumir que este tiempo sea preciso. Como el método **sleep()**, el **join()** responde a una interrupción terminando con una **InterruptedException**

Métodos de la clase Thread

Yield

Permite indicarle al mecanismo de scheduling que el thread ya hizo suficiente trabajo y que podría cederle tiempo de CPU a otro thread. Su efecto es dependiente del SO sobre el que se ejecuta la JVM. Permite implementar **multithreading cooperativo**.

```
public class SimpleThread extends Thread {
    private int contador=10;
    public SimpleThread(int nro) {
        super("'" +nro);
    }
    public String toString() {
        return "#" + getName()+ ":" +contador--;
    }
    public void run()    {
        for (int i = contador; i > 0; i --){
            System.out.println(this);
            Thread.yield();
        }
        System.out.println("Termino!" + this );
    }
}
```

SimpleThread de esta manera
realizaría un procesamiento
mejor distribuido entre varias
tareas SimpleThread.

La interface Runnable

- Es posible escribir **threads** implementando la interface **Runnable**.
- La interface **Runnable** solamente especifica que se debe implementar el método **run()**.

```
package java.lang;

public interface Runnable {
    abstract public void run();
}
```

```
package java.lang;

public class Thread implements Runnable {
    //código JAVA
}
```

- Si una clase implementa la **interface Runnable** simplemente significa que tiene un método **run()**, pero NO tiene ninguna habilidad de **threading**. Para crear un thread a partir de un objeto **Runnable** es necesario crear un **objeto Thread** y pasarle el objeto **Runnable** en el constructor. Luego, se invoca al método **start()** sobre el **thread** creado, NO sobre el objeto **Runnable**.

Identifica el thread ejecutándose

```
public class SimpleThread implements Runnable{
    private int contador=10;
    public String toString() {
        return "#" + Thread.currentThread().getName() + ":" + contador--;
    }
    public void run(){
        for (int i = contador; i > 0; i --)
            System.out.println(this);
        System.out.println("Termino!" + this );
    }
}
```

Se obtiene una
referencia al thread en
ejecución

```
public class TestCincoThread {
    private static int nroThread=0;
    public static void main(String[] args) {
        for (int i=0; i<5;i++)
            new Thread(new SimpleThread(), ""+i).start();
    }
}
```

Métodos de la clase Thread

Interrupt

Es un pedido de interrupción. El thread que lo recibe se interrumpe a si mismo de una manera conveniente. El pedido causa que los métodos de bloqueo (**sleep()**, **join()**, **wait()**) disparen la excepción `InterruptedException` y además setea un *flag* en el **thread** que indica que al thread se le ha pedido que se interrumpa. Se usa el método **isInterrupted()** para preguntar por este *flag*.

```
public class SimpleThreadInterrupt {
    private int contador = 10;
    private class Mensaje implements Runnable {
        public void run() {
            for (int i = contador; i > 0; i--) {
                System.out.println( i);
                try {
                    TimeUnit.MILLISECONDS.sleep(4000);
                } catch (InterruptedException e) {
                    System.out.println("El thread " + this + " no puede terminar");
                }
            }
            System.out.println("Termino!" + this);
        }
    }
    // Fin de la clase Mensaje
    public static void main(String[] args) throws InterruptedException {
        SimpleThreadInterrupt s=new SimpleThreadInterrupt();
        Thread t = new Thread(s.new Mensaje());
        t.start();
        while (t.isAlive()) {
            System.out.println("Esperando.....");
            t.join(1000);
            if (((System.currentTimeMillis()) > 1000 * 60 * 60) && t.isAlive()) {
                System.out.println("Cansado de esperar!");
                t.interrupt();
                t.join();
            }
        }
        System.out.println("Fin!");
    }
    // Fin de la clase SimpleThreadInterrupt
```

La interface Runnable

- Una ventaja de implementar la **interface Runnable** es que todo el código pertenece a la misma clase y de esta manera es simple combinar la clase base con otras interfaces. Es posible acceder a cualquier objeto y métodos de la clase evitándose mantener referencias en objeto separados.
- Los objetos **Runnable** pueden extender a otras clases en lugar de Thread.
- La **interface Runnable** permite separar la implementación de una tarea del thread que la ejecuta. Es más flexible.
- También es importante considerar que JAVA provee un conjunto de clases que gerencian **multithreading** (por ej. pool de threads): la **interface Runnable** es aplicable a APIs para gerenciamiento de threads.

Ejemplo

```
import java.awt.Graphics;
import java.util.*;
import java.text.DateFormat;
import java.applet.Applet;

public class Reloj extends Applet implements Runnable {
    private Thread relojThread = null;

    public void start() {
        if (relojThread == null) {
            relojThread = new Thread(this, "Reloj");
            relojThread.start();
        }
    }

    public void run() {
        Thread miThread = Thread.currentThread();
        while (relojThread == miThread) {
            repaint();
            try {
                TimeUnit.MILLISECONDS.sleep(1000);
            } catch (InterruptedException e){}
        }
    }

    public void paint(Graphics g) {
        Calendar cal = Calendar.getInstance();
        Date fecha = cal.getTime();
        DateFormat fechaFormateada = DateFormat.getTimeInstance();
        g.drawString(fechaFormateada.format(fecha), 5, 10);
    }

    public void stop() {
        relojThread = null;
    }
} // Fin de la clase Reloj
```

Implementación de la **interface Runnable**.
Esto indica que se implementa el método **run()**, no se hereda ninguna habilidad de **threading**

Se crea una instancia de Thread, **relojThread**.
Estado **NEW THREAD**

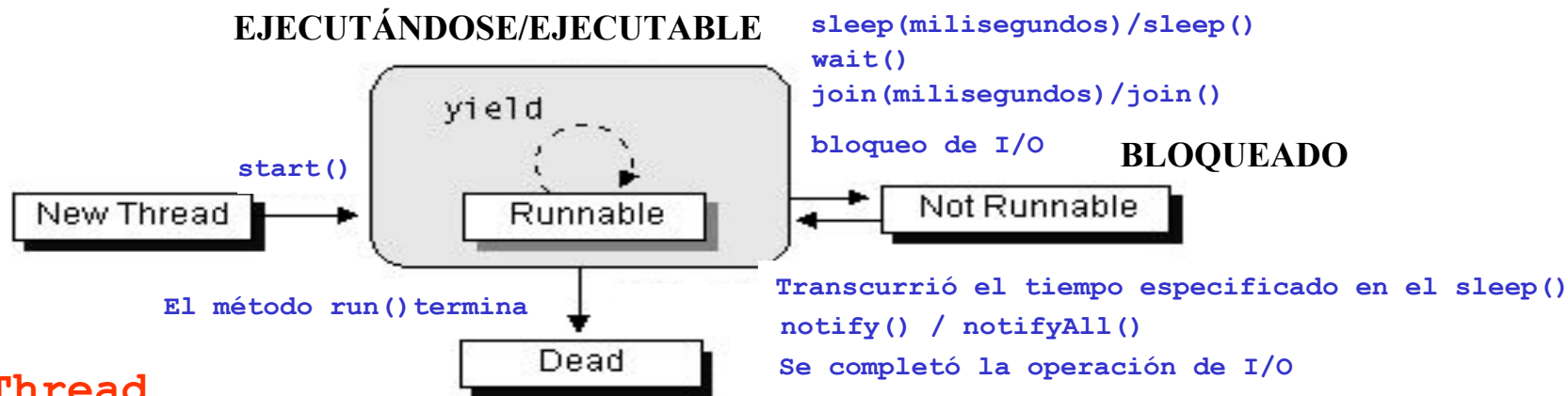
Crea los recursos para ejecutar el thread, organiza la ejecución del thread e **invoca al método run()**. Estado **RUNNABLE**

Durante un segundo el thread está en estado **NOT RUNNABLE**

Esta asignación hace que la condición de continuación del **run()** deje de cumplirse y de esta manera el thread finaliza.
Pasa a estado **DEAD**

Se crea un objeto **Thread** y se le provee de un objeto **Runnable** en el constructor. Este objeto es el que implementará el método **run()**.

Ciclo de vida de un Thread



Estado New Thread

Inmediatamente después que un thread es creado pasa a estado **New Thread**, pero aún no ha sido iniciado, por lo tanto no puede ejecutarse. Se debe invocar al método **start()**.

Estado Running (Ejecutándose)/Runnable (Ejecutable)

Después de ejecutarse el método **start()** el thread pasa al estado **Runnable o Ejecutable**. Un thread arrancado con **start()** podría o no comenzar a ejecutarse. No hay nada que evite que el thread se ejecute. La JVM implementa una estrategia (scheduling) que permite compartir la CPU entre todos los threads en estado Runnable.

Estado Not Runnable o Blocked (Bloqueado)

Un thread pasa a estado **Not Runnable o Bloqueado** cuando ocurren algunos de los siguientes eventos: se invoca al método **sleep()**, al **wait()**, **join()** ó **el thread está bloqueado en espera de una operación de I/O, el thread invoca a un método synchronized sobre un objeto y el lock del objeto no está disponible**. Cada entrada al estado **Not Runnable** tiene una forma de salida correspondiente. Cuando un thread está en estado bloqueado, el *scheduler* lo saltea y no le da ningún *slice* de CPU para ejecutarse.

Estado Dead

Los **threads** definen su finalización implementando un **run()** que termine naturalmente.

Prioridades en Threads

- En las configuraciones de computadoras en las que se dispone de una única CPU, los threads se ejecutarán de a uno a la vez simulando concurrencia. Uno de los principales beneficios del modelo de **threading** es que permite abstraernos de la configuración de procesadores.
- Cuando múltiples threads quieren ejecutarse, es el SO el que determina a cuál de ellos le asignará CPU. Los programas JAVA pueden influir, sin embargo la decisión final es del SO.
- Se llama **scheduling** a la estrategia que determina el orden de ejecución de múltiples threads sobre una única CPU.
- La JVM soporta un algoritmo de **scheduling simple** llamado **scheduling de prioridad fija**, que consiste en determinar el orden en que se ejecutarán los threads de acuerdo a la prioridad que ellos tienen.
- La prioridad de un thread le indica al **scheduler** cuán importante es.
- Cuando se crea un thread, éste hereda la prioridad del thread que lo creó (NORM_PRIORITY). Es posible modificar la prioridad de un thread después de su creación usando el método **setPriority(int)**. Las prioridades de los threads son números enteros que varían entre las constantes MIN_PRIORITY y MAX_PRIORITY (definidas en la clase Thread).

Prioridades en Threads

- El sistema de ejecución de JAVA elige para ejecutar entre los threads que están en estado **Runnable** aquel que tiene prioridad más alta. Cuando éste thread finaliza, cede el procesador o pasa a estado **Bloqueado**, comienza a ejecutarse un thread de más baja prioridad.
- El **scheduler** usa una estrategia **round-robin** para elegir entre dos threads de igual prioridad que están esperando por la CPU. El thread elegido se ejecuta hasta que un thread de más alta prioridad pase a estado **Runnable**, ceda la CPU a otro thread, finalice el método run() ó, expire el tiempo de CPU asignado (time-slicing). Luego, el segundo thread tiene la posibilidad de ejecutarse.
- El algoritmo de **scheduling** también es **preemptive**: cada vez que un thread con mayor prioridad que todos los threads que están en estado **Runnable** pasa a estado **Runnable**, el sistema de ejecución elige el nuevo thread de mayor prioridad para ejecutarse.

Ejemplo de Threads con Prioridades

```
import java.awt.*;
import java.applet.Applet;
public class RaceApplet extends Applet implements Runnable {
    private Runner[] runners = new Runner[2];
    private Thread updateThread = null;
    public void init() {
        runners[0] = new Runner();
        runners[1] = new Runner();
        if (raceType.compareTo("unfair") == 0){
            runners[0].setPriority(Thread.NORM_PRIORITY);
            runners[1].setPriority(Thread.MIN_PRIORITY);
        } else {
            runners[0].setPriority(Thread.NORM_PRIORITY);
            runners[1].setPriority(Thread.NORM_PRIORITY);
        }
        if (updateThread == null) {
            updateThread = new Thread(this, "Thread Race");
            updateThread.setPriority(Thread.MAX_PRIORITY);
        }
    }
    //Código JAVA
    public void run() {
        Thread myThread = Thread.currentThread();
        while (updateThread == myThread) {
            repaint();
            try {TimeUnit.MILLISECONDS.sleep(1000);}
            catch (InterruptedException e) {}
        }
    }
} // Fin de la clase RaceApplet
```

Se inicializan las prioridades de los dos threads "runners"

Se inicializa la prioridad del thread que dibuja en MAX_PRIORITY

```
public class Runner extends Thread {
    public int tick = 1;
    public void run() {
        while (tick < 400000)
            tick++;
    }
}
```


Ejecutores

Los Ejecutores simplifican la programación concurrente. Se incorporaron en JSE 5.

- Los **EJECUTORES** proveen una capa de indirección entre un cliente y la ejecución de una tarea. Es un objeto intermedio que ejecuta la tarea, desligando al cliente de la ejecución de la misma.
- Los **EJECUTORES** son objetos que encapsulan la creación y administración de **threads**, permitiendo **desacoplar** la tarea concurrente del mecanismo de ejecución. Entre sus responsabilidades están la creación, el uso y el *scheduling* de threads.
- Los **EJECUTORES** permiten modelar programas como una serie de tareas concurrentes asincrónicas, evitando los detalles asociados con **threads**: simplemente se crea una tarea que se pasa al ejecutor apropiado para que la ejecute.
- Un **EJECUTOR** es normalmente usado en vez de crear explícitamente **threads**:

Con threads creados por el programador:

```
Runnable r= new RunnableTask();  
new Thread(r).start();
```

Con EJECUTORES:

```
Executor e= unEjecutor;  
Runnable r= new RunnableTask();  
e.execute(r);
```

- Un **EJECUTOR** es un objeto que implementa la interface **Executor**.

```
package java.util.concurrent;  
public interface Executor {  
    public void execute();  
}
```

Construye el
contexto apropiado
para ejecutar objetos
Runnable

Ejecutores

Con threads creados por el programador:

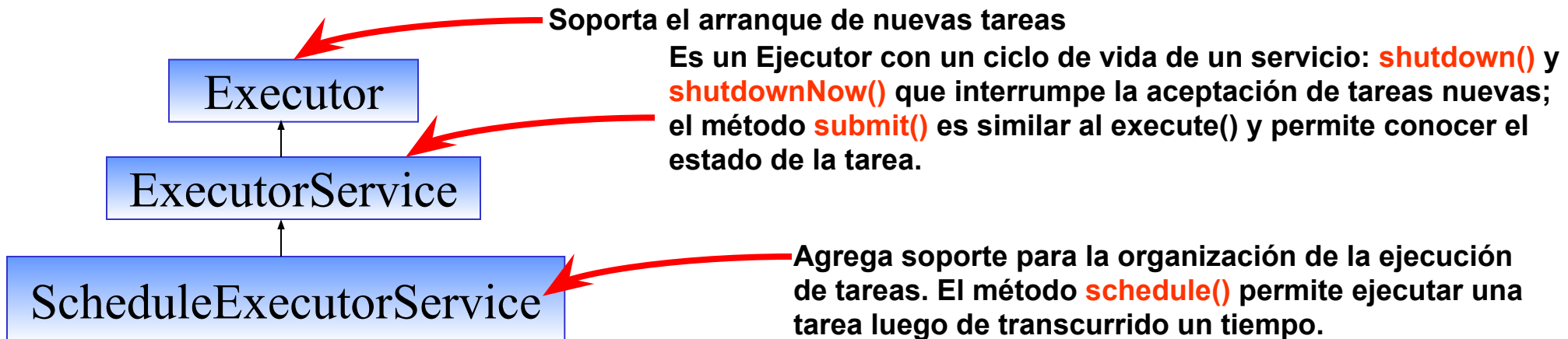
```
Runnable r= new RunnableTask();  
new Thread(r).start();
```

Con EJECUTORES:

```
Executor e= unEjecutor;  
Runnable r= new RunnableTask();  
e.execute(r);
```

El comportamiento del método **execute()** es menos específico que el usado con **Threads**, siendo los **threads** creados y lanzamos inmediatamente. Dependiendo de la implementación del **Executor** el método **execute()** podría hacer lo mismo, o usar un **thread** existente disponible para ejecutar **r** o encolar **r** hasta que haya un **thread** disponible para ejecutar la tarea.

El paquete **java.util.concurrent** define tres interfaces Executor:



Ejecutores & Pool de Threads

Típicamente las implementaciones de **EJECUTORES** del paquete **java.util.concurrent** usan *pool de threads*. Estos threads existen independientemente de las tareas Runnablees que ejecutan y generalmente ejecutan múltiples tareas.

El pool de threads minimiza la sobrecarga causada por la creación de nuevos threads=> **reuso de threads.**

Aumenta la *performance* de aplicaciones que ejecutan muchos **threads** simultáneamente. El pool adquiere un rol crucial en configuraciones donde se tienen más threads que CPUs => **programas más rápidos y eficientes.**

Para crear un **EJECUTOR** que **use una pool de threads** se puede invocar a los siguientes métodos de clase de la clase **java.util.concurrent.Executors**:

```
ExecutorService exec = Executors.newFixedThreadPool(int nThreads);
```

```
ExecutorService exec = Executors.newFixedThreadPool(int nThreads, ThreadFactory threadFact);
```

Crea un pool de **threads** que reusa un conjunto finito de **threads**. En el 2do método se usa el objeto **ThreadFactory** para crear los **threads** necesarios.

```
ExecutorService exec = Executors.newCachedThreadPool();
```

```
ExecutorService exec = Executors.newCachedThreadPool(ThreadFactory threadFact);
```

Crea un pool de **threads** que crea **threads** nuevos a medida que los necesita y reusa los construidos que están disponibles. El 2do método usa el objeto **ThreadFactory** para crear los nuevos **threads**

Ejemplo 1

```
package concurrentes;
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;
public class CachedThreadPool {
    public static void main(String[ ] args) {
        ExecutorService exec = Executors.newCachedThreadPool();
        for ( int i = 0; i < 5; i++)
            exec.execute(new SimpleThread());
        exec.shutdown();
    }
}
```

Tarea Específica

ExecutorService exec = Executors.newFixedThreadPool(10);

Ejemplo 2

```
package concurrentes;
import java.util.concurrent.*;

public class SingleThreadExecutor {
    public static void main(String[] args){
        ExecutorService exec = Executors.newSingleThreadExecutor();
        for ( int i = 0; i < 5; i++)
            exec.execute(new SimpleThread());
        exec.shutdown();
    }
}
```

En este ejemplo una tarea se completada en el mismo orden en que es recibida y antes de comenzar una nueva.

Un **SingleThreadExecutor** es similar a **FixedThreadPool** con un pool de un único thread. Es útil para tareas que se quieran ejecutar continuamente, por ej: escuchan conexiones entrantes, tareas que actualizan logs remotos o locales o que hacen *dispatching* de eventos. Otro ej: tareas que usan el filesystem evitando manejar la sincronización

Compartir Recursos

Condición de Carrera

```
public class Counter
```

```
{
```

```
    private int c = 0;  
    public void increment()  
    {
```

```
        c++;
```

```
    }
```

```
    public void decrement()  
    {
```

```
        c--;
```

```
    }
```

```
    public int value()  
    {
```

```
        return c;
```

```
    }
```

```
}
```

Si un mismo objeto **Counter** es **referenciado** por múltiples threads (por ejemplo A y B), la interferencia entre estos threads provocaría que el comportamiento de los métodos **increment()** y **decrement()** NO sea el esperado

Recuperar el valor actual de c.
Incrementarlo/Decrementarlo en 1.
Guardar en c el nuevo valor

¿Qué pasa si el thread A invoca al increment() al mismo tiempo que el thread B invoca decrement() sobre la misma instancia de Counter?

Si el valor inicial de c es 0, podría ocurrir lo siguiente:

Thread A: Recupera c. (c=0)

Thread B: Recupera c. (c=0)

Thread A: Incrementa el valor recuperado; resultado es c=1.

Thread B: Decrementa el valor recuperado; resultado es c=-1 (lo hace antes que A guarde el valor).

Thread A: Guarda el resultado en c; c=1

Thread B: Guarda el resultado en c; c=-1

Condición de Carrera

Compartir Recursos

- Hasta ahora vimos ejemplos de threads asincrónicos que no comparten datos ni necesitan coordinar sus actividades.
- Con multithreading hay situaciones en que dos o más threads intentan acceder a los mismos recursos en el mismo momento. Se debe evitar este tipo de colisión sobre los recursos compartidos (durante períodos críticos): acceder a la misma cuenta bancaria en el mismo momento, imprimir en la misma impresora, etc. Ejemplo de la clase Counter
- Para resolver el problema de colisiones, todos los esquemas de multithreading establecen *un orden para acceder al recurso compartido*. En general se lleva a cabo usando una cláusula que *bloquea* (lock) el código que accede al recurso compartido y así solamente de a un thread a la vez se accede al recurso. Esta cláusula implementa **exclusión mutua**.
- Java provee soporte para **exclusión mutua** mediante la palabra clave **synchronized**.

Compartir Recursos

- Cada objeto contiene un **lock** único llamado **monitor**. Cuando invocamos a un **método synchronized**, el objeto es “bloqueado” (locked) y ningún otro método **synchronized** sobre el mismo objeto puede ejecutarse hasta que el primer método termine y libere el **lock del objeto**.
- El **lock** del objeto es único y compartido por todos los métodos y **bloques synchronized** del mismo objeto. Este **lock** evita que el recurso común sea modificado por más de thread a la vez.

```
public class Recurso {  
    public synchronized int f() {}  
    public synchronized void g() {}  
}
```

Si el método f() es invocado sobre un objeto Recurso, el método g() no puede ejecutarse sobre el mismo objeto, hasta que f() termine y libere el lock.

- Es posible definir un bloque **synchronized**: **synchronized (unObjeto) {}**
- Un thread puede adquirir el **lock** de un objeto múltiples veces. Esto ocurre si un método invoca a un segundo método **synchronized** sobre el mismo objeto, quién a su vez invoca a otro método **synchronized** sobre el mismo objeto, etc. La JVM mantiene un contador con el número de veces que el objeto fue bloqueado (lock). Cuando el objeto es desbloqueado, el contador toma el valor cero. Cada vez que un thread adquiere el lock sobre el mismo objeto, el contador se incrementa en uno y cada vez que abandona un método **synchronized** el contador se decrementa en uno, hasta que el contador llegue a cero, liberando el lock para que lo usen otros threads. La adquisición del lock múltiples veces sólo es permitida para el thread que lo adquirió en el primer método **synchronized** que invocó.

Compartir Recursos

La cláusula synchronized

```
public class SynchronizedCounter
{
    private int c = 0;
    public synchronized void increment()
    {
        c++;
    }
    public synchronized void decrement()
    {
        c--;
    }
    public synchronized int value()
    {
        return c;
    }
}
```

Ejemplo

Producer/Consumidor

```
public class Productor extends Thread {  
    private Bolsa bolsa;  
    private int numero;  
  
    public Productor(Bolsa c, int numero) {  
        bolsa = c;  
        this.numero = numero;  
    }  
  
    public void run() {  
        for (int i = 0; i < 10; i++) {  
            bolsa.put(i);  
            System.out.println("Productor #" + this.numero  
                + " escribió: " + i);  
            try {  
                TimeUnit.MILLISECONDS.sleep((int)(Math.random() * 100));  
            } catch (InterruptedException e) {}  
        }  
    }  
}
```

Objeto Compartido

```
public class Consumidor extends Thread {  
    private Bolsa bolsa;  
    private int numero;  
  
    public Consumidor(Bolsa c, int numero) {  
        bolsa = c;  
        this.numero = numero;  
    }  
  
    public void run() {  
        int valor = 0;  
        for (int i = 0; i < 10; i++) {  
            valor = bolsa.get();  
            System.out.println("Consumidor#" +  
                this.numero + " leyó: " + valor);  
        }  
    }  
}
```

Ejemplo

Productor/Consumidor

Para evitar que el Productor y el Consumidor colisionen sobre el objeto compartido Bolsa, esto es, que intenten guardar y leer simultáneamente dejando al objeto en un estado inconsistente, los métodos `get()` y `put()` se declaran **synchronized**

```
public class Bolsa {  
    private int contenido;  
    private boolean disponible = false;  
    public synchronized int get() {  
        //El objeto bolsa fue bloqueada por el Consumidor  
  
        .....  
        //el objeto bolsa fue desbloqueada por el Consumidor  
    }  
  
    public synchronized void put(int value) {  
        // El objeto bolsa fue bloqueada por el Productor  
  
        .....  
        //el objeto bolsa fue desbloqueado por el Productor  
    }  
}
```

Secciones críticas

Productor/Consumidor

¿Qué sucede si el Productor es más rápido que el Consumidor y genera dos números antes que el consumidor pueda consumir el primero de ellos?

.....

Consumidor #1 leyó: 3

Productor #1 escribió: 4

Productor #1 escribió: 5

Consumidor #1 leyó: 5

← El Consumidor perdió el 4

¿Qué sucede si el Consumidor es más rápido que el Productor y consume dos veces el mismo valor?

.....

Productor #1 escribió: 4

Consumidor #1 leyó: 4

Consumidor #1 leyó: 4

Productor #1 escribió: 5

← El Consumidor obtiene el 4 dos veces

En ambos casos el resultado es erróneo dado que el Consumidor debe leer cada uno de los números producidos por el Productor exactamente una vez.

Cooperación entre Threads

¿Cómo podemos hacer para que el **Productor** y el **Consumidor** cooperen entre ellos?

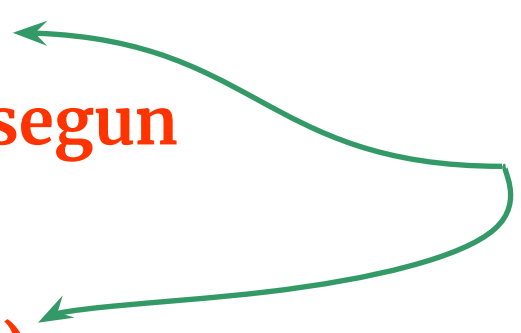
El **Productor** debe indicarle al **Consumidor** de una manera sencilla que el valor está listo para ser leído y el **Consumidor** debe tener alguna forma de indicarle al **Productor** que el valor ya fue leído.

Además, si no hay nada para leer, el **Consumidor** debe esperar a que el **Productor** escriba un nuevo valor y, el **Productor** debe esperar a que el **Consumidor** lea antes de escribir un valor nuevo.

Para este propósito la clase **Object** provee los siguientes métodos: **wait()**, **wait(milisegundos)**, **notify()** y **notifyAll()**.

Los métodos **wait()**, **wait(milisegundos)**, **notify()** y **notifyAll()** deben usarse adentro de un método o bloque **synchronized**.

wait()
wait(milisegundos)
notify()
notifyAll()



El método **wait()** suspende la ejecución del thread y libera el *lock* del objeto, y así permite que otros métodos **synchronized** sobre el mismo objeto puedan ejecutarse.

El método **notifyAll()** “despierta” a todos los threads esperando (**wait()**), compiten por el *lock* y el que lo obtiene retoma la ejecución. El método **notify()** despierta a un thread.

Productor/Consumidor

```
public class Bolsa {  
    private int contenido;  
    private boolean disponible = false;  
    public synchronized int get() {  
        while (disponible == false) {  
            try { wait(); }  
            catch (InterruptedException e) { .. }  
        }  
        disponible = false;  
        notifyAll();  
        return contenido;  
    }  
    public synchronized void put(int value) {  
        while (disponible == true) {  
            try { wait(); }  
            catch (InterruptedException e) { .. }  
        }  
        contenido = value;  
        disponible = true;  
        notifyAll();  
    }  
}
```

Libera el lock (del objeto Bolsa) tomado por el Consumidor, permitiendo que el Productor agregue un dato nuevo en la Bolsa y, luego espera ser notificado por el Productor.

El Consumidor notifica al Productor que ya leyó, dándole la posibilidad de producir un nuevo valor.

Libera el lock (del objeto Bolsa) tomado por el Productor, permitiendo que el Consumidor lea el valor actual antes de producir un nuevo valor.

El Productor notifica al Consumidor cuando agregó un dato nuevo en la Bolsa

Productor/Consumidor

```
public class ProductorConsumidorTest {  
    public static void main(String[] args) {  
        Bolsa c = new Bolsa();  
        Productor p1 = new Productor(c, 1);  
        Consumidor c1 = new Consumidor(c, 1);  
  
        p1.start();  
        c1.start();  
    }  
}
```

Productor #1 escribió: 0
Consumidor #1 leyó: 0
Productor #1 escribió: 1
Consumidor #1 leyó: 1
Productor #1 escribió: 2
Consumidor #1 leyó: 2
Productor #1 escribió: 3
Consumidor #1 leyó: 3
.....
Productor #1 escribió: 9
Consumidor #1 leyó: 9

Salida del programa
ProductorConsumidorTest