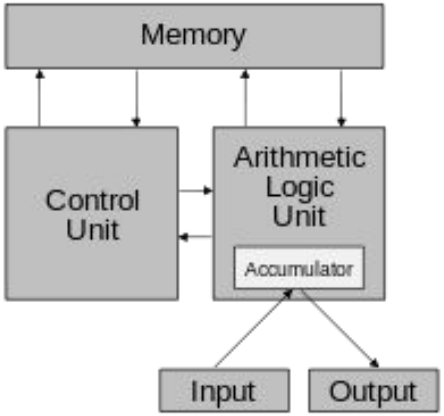
Programación Concurrente - Threads

¿Qué se entiende por un Computador?

Un dispositivo que tiene capacidad de procesamiento y dispositivos de E/S.



El mismo requiere para poder realizar **cálculos** guardar **datos en almacenamiento volátil** (RAM y caché)

Ese es el concepto tradicional de **computadora de Von Neumann**.

Este concepto con el tiempo se fue complejizando con el objetivo de obtener un mejor rendimiento del computador.

Programa vs. Proceso

Programa (o algoritmo)

Es una **secuencia de instrucciones**, escritas para realizar una tarea específica en un computador.

Es un elemento estático que reside en un almacenamiento no volátil a la espera de un pedido de ejecución.

Proceso

Es un **programa** en **ejecución.**

El **sistema operativo le asigna** ALU, RAM y algunos **recursos** que requiera el programa.

Es un elemento dinámico ya que se carga el programa en la memoria volátil y la misma cambia durante la ejecución.

Modelos de Programación

A medida que las computadoras lograron mayor complejidad fueron surgiendo nuevos modelos de programación.

Analizaremos algunos:

- Modelos secuencial o lineal.
- Modelo concurrente.
- Modelo distribuido.

Ejercicio 1 - Programación Secuencial

- Descargar el paquete **project-concurrency.zip** del módulo en el campus.
- Descomprimir e incorporar al IDE.
- Se debe implementar el servicio ***GenericService*** en la clase ***GenericServiceImpl*** asegurándose de que les corran los test de unidad ubicados en ***GenericServiceTest***.

Modelo de Programación Secuencial

Modelo de Programación Secuencial

En el modelo secuencial se corre cada proceso “completo”.

Es el modelo natural de un procesador que va ejecutando las instrucciones del proceso en el orden que él mismo determina.

Procesamiento unicore secuencial

A) Supongamos un scheduler que recibe 3 procesos y los ejecuta en orden de llegada. ¿Cuánto tardaría en terminar cada proceso?

Proceso ID	Tiempo que tarda en completar ejecución
P1	120 seg
P2	60 seg
P3	30 seg

Respuesta:

- **P1: 120 segundos**
- **P2: 180 segundos**
- **P3: 210 segundos**

Procesamiento unicore secuencial

B) Suponiendo un scheduler que recibe 3 procesos y los ejecuta en orden de tiempo de ejecución. ¿Cuánto tardaría en terminar cada proceso?

Proceso ID	Tiempo que tarda en completar ejecución
P3	30 seg
P2	60 seg
P1	120 seg

Respuesta:

- **P1: 210 segundos**
- **P2: 90 segundos**
- **P3: 30 segundos**

Procesamiento unicore secuencial

Si votaran P1, P2 y P3 teniendo en cuenta la demora en la que se les asigna CPU ¿Cuál de los 2 schedulers saldría elegido?

Comparando:

- **P1** demora con **A 0** seg. y con **B 90** seg.
- **P2** demora con **A 120** seg. y con **B 30** seg.
- **P3** demora con **A 180** seg. y con **B 0** seg.

Procesamiento unicore secuencial

Por supuesto esa estrategia requeriría que todos los procesos lleguen al scheduler antes de que los asigne, lo cual no es una situación realista.

Procesamiento unicore intercalando procesos

Con el tiempo los sistemas operativos comenzaron a implementar estrategias de “**time slice**” y “**context switching**”.

Time slice	Context switching
El sistema divide el tiempo de procesador en “slices” pequeños que va repartiendo entre los procesos.	**Cada vez que el slice se acaba (o el proceso "actual" cede su tiempo), el S.O. se encarga de disponibilizar los recursos y el estado del proceso que va a usar el siguiente slice.**

Esto permite una **operación multiproceso** que da la sensación de que los programas están corriendo "al mismo tiempo".

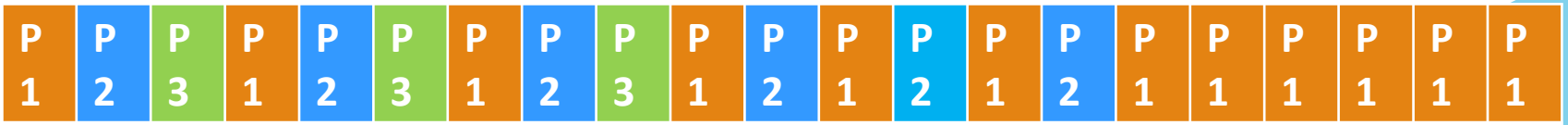
Procesamiento unicore intercalando procesos

C) Suponiendo un scheduler que recibe 3 procesos y los intercale en el orden de llegada. ¿Cuánto tardaría en terminar cada proceso?

Proceso ID	Tiempo que tarda en completar ejecución
P3	30 seg
P2	60 seg
P1	120 seg

Respuesta:

- **P1: 210 segundos**
- **P2: 150 segundos**
- **P3: 90 segundos**



Procesamiento unicore intercalando procesos

Si votaran P1, P2 y P3 teniendo en cuenta la demora en la que se les asigna CPU ¿ Cuál de los 2 schedulers saldría elegido?

Comparando:

- **P1** demora con **A 0** seg**.** con **B 90** seg. y con **C 0** seg**.**
- **P2** demora con **A 120** seg. con **B 30** seg. y con **C 10** seg.
- **P3** demora con **A 180** seg. con **B 0** seg. y con **C 20** seg.

Procesamiento unicore intercalando procesos

¿ Y si analizamos el promedio de demora?

- con A fue de 100 seg
- con B fue de 40 seg
- con C fue de 10 seg

Procesamiento unicore intercalando procesos

Este modelo trae obvios beneficios, cada proceso:

- Comienza antes
- Puede obtener resultados parciales en tiempos razonables
- Puede beneficiarse de tiempos cedidos por otros procesos.
- En general la mayoría de los procesos pueden finalizar antes

Estos beneficios son para varios procesos corriendo al mismo momento.

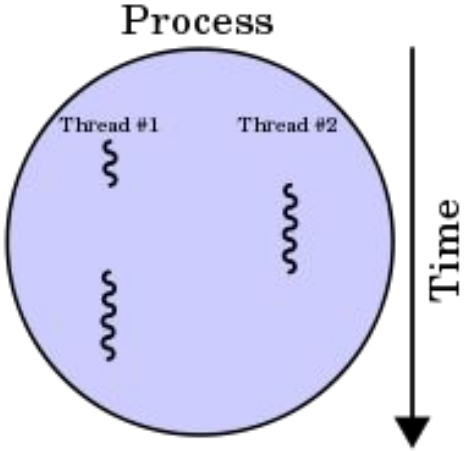
Un solo proceso no puede beneficiarse del intercalamiento.

Modelo de Programación Concurrente

Modelo de Programación Concurrente

Con este modelo se aprovecha el surgimiento de los **Threads** y de los **procesadores multicore.**

- Para esto el **algoritmo** identifica **zonas de código** que se pueden **correr** en forma **concurrente**.
- Esas **zonas** se definen **dentro de un Thread** que pueden correr de manera **concurrente** (y semi independiente) al Proceso principal.
- En caso de tener **procesadores multicore** el Thread podría **correr paralelamente.**

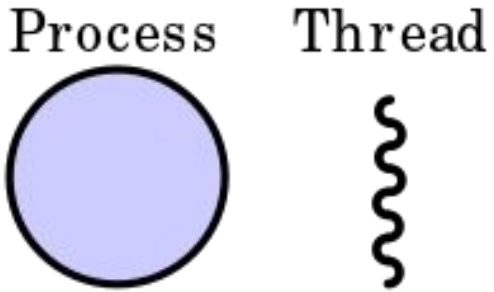


PARALELO != CONCURRENTE

Threads vs. Procesos

Un proceso puede tener threads que son subunidades de ejecución que pueden correr independientemente del proceso principal (y de otros Threads)..

- Los **Threads de un proceso comparten** el espacio de memoria, los Files Descriptors y demás **recursos** del proceso principal.
- **Pueden comunicarse entre** los **Threads** de un proceso y hasta **afectar el comportamiento** de uno desde otro. Los **procesos** solo se comunican vía **IPC**.
- La **creación** y switcheo de contexto de **Threads es mucho más simple** y requiere mucho menos tiempo y recursos que la creación de Procesos.
- **El contro**l de inicio y fin de **Threads es responsabilidad del programador** en contrapartida con los procesos que los maneja el sistema operativo



Threads vs. Procesos

Al tener menos overhead los Threads son más eficientes para correr tareas por lo cual son muy utilizados aunque requieren ciertos cuidados al utilizarlos

Threads en Java - Thread

Para poder utilizar Threads en Java se utiliza la clase **Thread**.

```
public class HelloThread extends Thread {
      @Override
      public void run() {
            System.out.println("Hello from a thread!");
      }

      public static void main(String[] args) {
            Thread thread = new HelloThread();

            thread.start();
      }
}
```

Implementar RUN

Ejecutar usando a START

Esta es la forma menos utilizada ya que Java no tiene multi herencia y extender de la clase Thread limita la programación.

Threads en Java - Runnable

Un poco más genérico es implementar la interfaz **Runnable**.

```
public class HelloRunnable implements Runnable {
      @Override
      public void run() {
            System.out.println("Hello from a thread!");
      }
      public static void main(String[] args) {
            Thread helloThread = new Thread(new HelloRunnable());

            helloThread.start();
      }
}
```

Implementar RUN

Ejecutar mediante a START

Se sigue utilizando **Thread**, solo que se lo construye con el **Runnable**.

Métodos estáticos en Threads

La clase **Thread** tiene métodos estáticos de utilidad:

`Thread currentThread()`	Retorna una instancia del thread actual
`void sleep(long millis)`	Provoca que el thread actual quede suspendido
`boolean interrupted()`	Indica si el thread actual fue interrumpido
`void yield()`	Indica que el thread actual puede liberar el procesador por el momento
`boolean holdsLock(Object obj)`	Informa si el thread actual tiene un lock sobre el objeto dado.

Métodos estáticos en Threads

Thread que se duerme un segundo antes de terminar

```
public class SleeperRunnable implements Runnable {
	@Override
	public void run() {
		for (int i = 0; i < 10; i++) {
			try {
				System.out.println("Siesta numero: " + i);
				Thread.sleep(1000);
			} catch (InterruptedException e) {
				System.out.println("Interrupted");
				return;
			}
		}
	}
}
```

Métodos de instancia en Threads

La clase **Thread** tiene métodos de instancia útiles:

`String getName()`	Retorna el nombre del thread
`void interrupt()`	Interrumpe al Thread. Si el mismo está suspendido en un wait o join, él mismo recibe una InterruptedException, si no se setea el “interrupted” flag
`boolean isAlive()`	Indica si el Thread está vivo, esto quiere decir iniciado pero no terminado
`void join()`	Suspende la ejecución del **Thread que invoca** hasta que el Thread sobre el cual se llama termine

Métodos de instancia en Threads

Clase que se encarga de lanzar los “sleepy” threads

```
public class SleepyThreads {
      public static void main(String[] args) throws InterruptedException {
            final Thread[] ts = new Thread[2];
            for (int i = 0; i < ts.length; i++) {
                  Thread thread = new Thread(new SleeperRunnable(), "sl-" + i);
                  thread.start();
                  ts[i] = thread;
            }
            ts[1].interrupt();
            ts[0].join();
      }
}
```

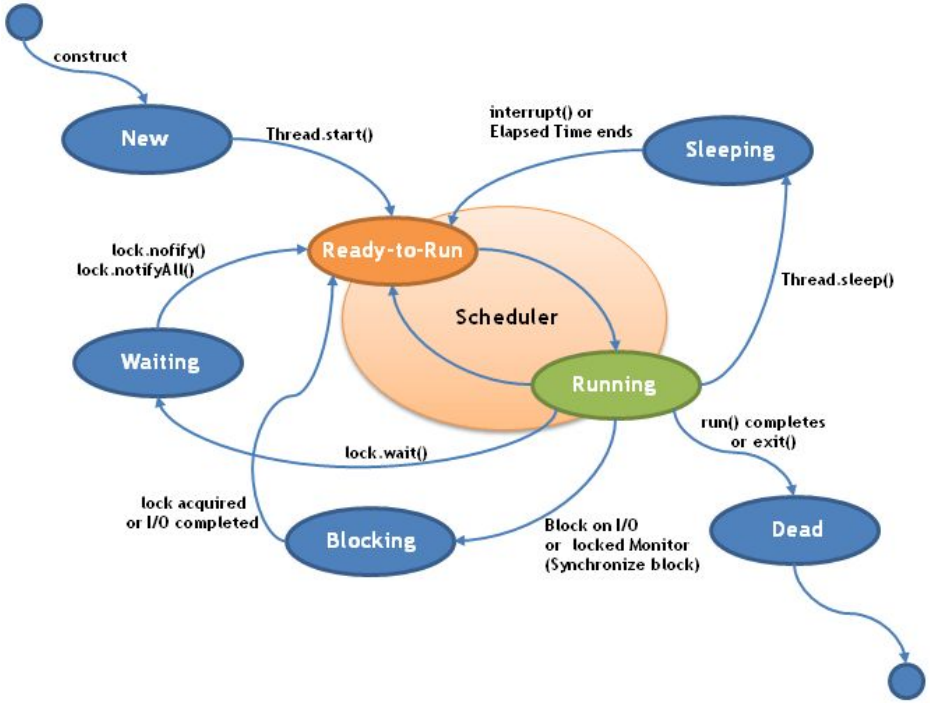
Recordar que **Runnable** es una interfaz funcional.

Ejercicio 2 - Threads

- Incorporar al proyecto **concurrency** un Test (ThreadTest) que se encargue de instanciar un GenericService pasárselo a un Thread que realice 5 visitas e imprima en pantalla el conteo de visitas final.
- En la misma clase, escribir dos nuevos test que realicen el mismo proceso usando un Runnable y una Lambda respectivamente.

Ciclo de vida de los Threads

Durante el ciclo de vida de un Thread el mismo puede tener los siguientes estados:



Threads en Java 5

Threads en Java 5

Java 5 agrega unas abstracciones y servicios para realizar las operaciones con threads de manera más eficiente, con mayor poder.

La semántica previa era lanzar un Thread y que sea “fire and forget”. O sea no importa el resultado, ni cuando termina.

Si importara se pueden usar elementos, como variables compartidas o colas para obtener el resultado, pero no es lo más cómodo.

Threads en Java 5

En Java 5 se incorporan los siguientes elementos:

- La interfaz **Callable**.
- La interfaz **Future**.
- La interfaz **ExecutorService**.

A partir de ellos se puede coordinar y obtener respuestas entre Threads de manera simple

Threads en Java 5 - Callable

Callable representa una tarea a ejecutar, al igual que Runnable, pero la diferencia es que la tarea **retorna un valor de respuesta**.

```
@FunctionalInterface
public interface Callable<V> {
      V call() throws Exception;
}
```

Se puede tener un Callable<Void> que sería equivalente a Runnable

Threads en Java 5 - Future

La interfaz **Future** que representa **el estado de una tarea asincrónica** que se mandó a ejecutar.

El principal objetivo es obtener la respuesta mediante el método “get”, pero como la tarea puede estar “corriendo” también permite preguntar si la tarea ya terminó o no.

```
public interface Future<V> {
      boolean cancel(boolean var1);

      boolean isCancelled();

      boolean isDone();

      V get() throws InterruptedException, ExecutionException;

      V get(long var1, TimeUnit var3) throws InterruptedException,
      ExecutionException, TimeoutException;
}
```

Threads en Java 5 - ExecutorService

La interfaz **ExecutorService** es la que permite lanzar los Thread (de Runnable y Callable) y que los mismos se wrapeen con un Future para que se pueda utilizar esa semántica para acceder al resultado del Thread.

```
public interface ExecutorService extends Executor {
      …
      <T> Future<T> submit(Callable<T> var1);
      <T> Future<T> submit(Runnable var1, T var2);
      Future<?> submit(Runnable var1);
      …
}
```

Threads en Java 5 - ExecutorService

ExecutorService además provee métodos para lanzar varios threads en simultáneo y obtener el resultado del primero que termina.

```
public interface ExecutorService extends Executor {
      …
      <T> List<Future<T>> invokeAll(Collection<? extends Callable<T>> var1) throws
      InterruptedException;

      <T> List<Future<T>> invokeAll(Collection<? extends Callable<T>> var1, long var2,
      TimeUnit var4) throws InterruptedException;

      <T> T invokeAny(Collection<? extends Callable<T>> var1) throws
      InterruptedException, ExecutionException;
      …
}
```

Threads en Java 5 - ExecutorService

ExecutorService también permite mantener el control sobre el ciclo de vida del mismo y los Futures que se están ejecutando.

```
public interface ExecutorService extends Executor {
      …
      void shutdown();
      List<Runnable> shutdownNow();
      boolean isShutdown();
      boolean isTerminated();
      boolean awaitTermination(long var1, TimeUnit var3) throws InterruptedException;
      …
}
```

shutdown and **shutdownNow** envían señales de **interrupción** a las tareas y a partir de **awaitTermination** and **isTerminated** se puede controlar el estado (más allá de preguntarle a cada Future).

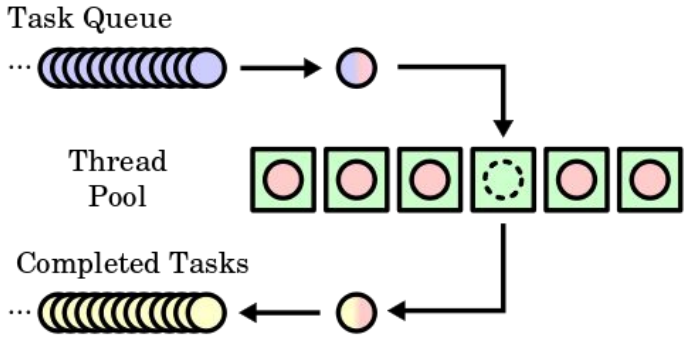
Executors - Shutdown

La manera recomendada de cerrar un ExecutorService es utilizar ambos shutdown y shutdown now.

```
executorService.shutdown();
try {
      if (!executorService.awaitTermination(800, TimeUnit.MILLISECONDS)) {
      executorService.shutdownNow();
      }
} catch (InterruptedException e) {
      executorService.shutdownNow();
}
```

ExecutorService - Thread Pool

A pesar de que la creación de Threads es más eficiente que un proceso sigue consumiendo tiempo y recursos.



Para mejorar esto se puede generar un **Thread Pool** que están creados y a medida que los mismos terminan sus tareas se les asigna una nueva tarea (Thread) a ejecutar.

Existen implementaciones de ExecutorServices que:

- se encargan de este manejo como **newFixedThreadPool y newCachedThreadPool**.
- Que permite correr **tareas con delay o repetitivamente** por ejemplo **ScheduledExecutorService**.

Executors - Construcción

Para hacer más simple la creación de estos ExecutorService la clase Executors provee métodos estáticos de construcción.

```
public class Executors {
  ...
  ExecutorService newCachedThreadPool(ThreadFactory threadFactory)
  ExecutorService newFixedThreadPool(int nThreads)
  ScheduledExecutorService newScheduledThreadPool(int corePoolSize)
  ExecutorService newSingleThreadExecutor()
  ScheduledExecutorService  newSingleThreadScheduledExecutor()
  ...
}
```

Mirar la implementación de estos métodos permite ver cómo instanciar ExecutorService en caso de necesitar alguno especial

Ejemplo de uso de Futures

Un caso donde se quiere bajar algo de Internet, por lo cual puede tardar mucho.

Se utiliza el Future para iniciar la descarga y realizar otras tareas mientras se descarga.

```
private final ExecutorService pool = Executors.newFixedThreadPool(10);

public Future<String> startDownloading(final URL url) throws IOException {
   return pool.submit(new Callable<String>() {
       @Override
       public String call() throws Exception {
           try (InputStream input = url.openStream()) {
               return IOUtils.toString(input, StandardCharsets.UTF_8);
           }
       }
   });
}
```

Ejemplo de uso de Futures

Teniendo el código anterior el uso del future sería:

```
final Future<String> contentsFuture = startDownloading(new
                                      URL("http://www.example.com"));

while (!contentsFuture.isDone()) {
   askUserToWait();
   doSomeComputationInTheMeantime();
}

String rta = contentsFuture.get();
```

Ejercicio 3 - Executor

- Agregar al proyecto un Test (ExecutorTest).
- Generar un test que utilizando un CachedThreadPool y un GenericService genere un Future que registre 5 visitas y retorne el conteo.
- El test debe realizar la assertion correspondiente al valor esperado de visitas

Takeaways

La programación concurrente permite aprovechar mejor los recursos del sistema y acelerar tiempos de respuesta.
Esto es aún mejor **en ambientes multicore**, ya que **hay ejecución paralela**.
Como contrapartida **aparecen nuevos potenciales problemas** que tendremos en cuenta a la hora de programar y analizaremos más adelante.

Referencias y para profundizar

- Sun - Java Tutorial de Concurrencia
- **Concurrent Programming in Java**: Design Principles and Pattern (2nd Edition) by Doug Lea. A comprehensive work by a leading expert, who's also the architect of the Java platform's concurrency framework.
- **Java Concurrency in Practice** by Brian Goetz, Tim Peierls, Joshua Bloch, Joseph Bowbeer, David Holmes, and Doug Lea. A practical guide designed to be accessible to the novice.
- **Effective Java Programming Language Guide (2nd Edition)** by Joshua Bloch. Though this is a general programming guide, its chapter on threads contains essential "best practices" for concurrent programming.

CREDITS

Content of the slides:

- POD - ITBA

Images:

- POD - ITBA
- Or obtained from: commons.wikimedia.org

Slides theme credit:

Special thanks to all the people who made and released these awesome resources for free:

- Presentation template by SlidesCarnival
- Photographs by Unsplash