Edwin Maraví
emaravi@cjavaperu.com

CJAVA

Misión

Nuestro equipo trabaja para integrar la tecnología Java en la sociedad como solución a todas sus necesidades.

CJAVA

siempre para apoyarte

Visión

Poder aportar al desarrollo del País usando tecnología Java.

CJAVA
siempre para apoyarte

Servicios Académicos

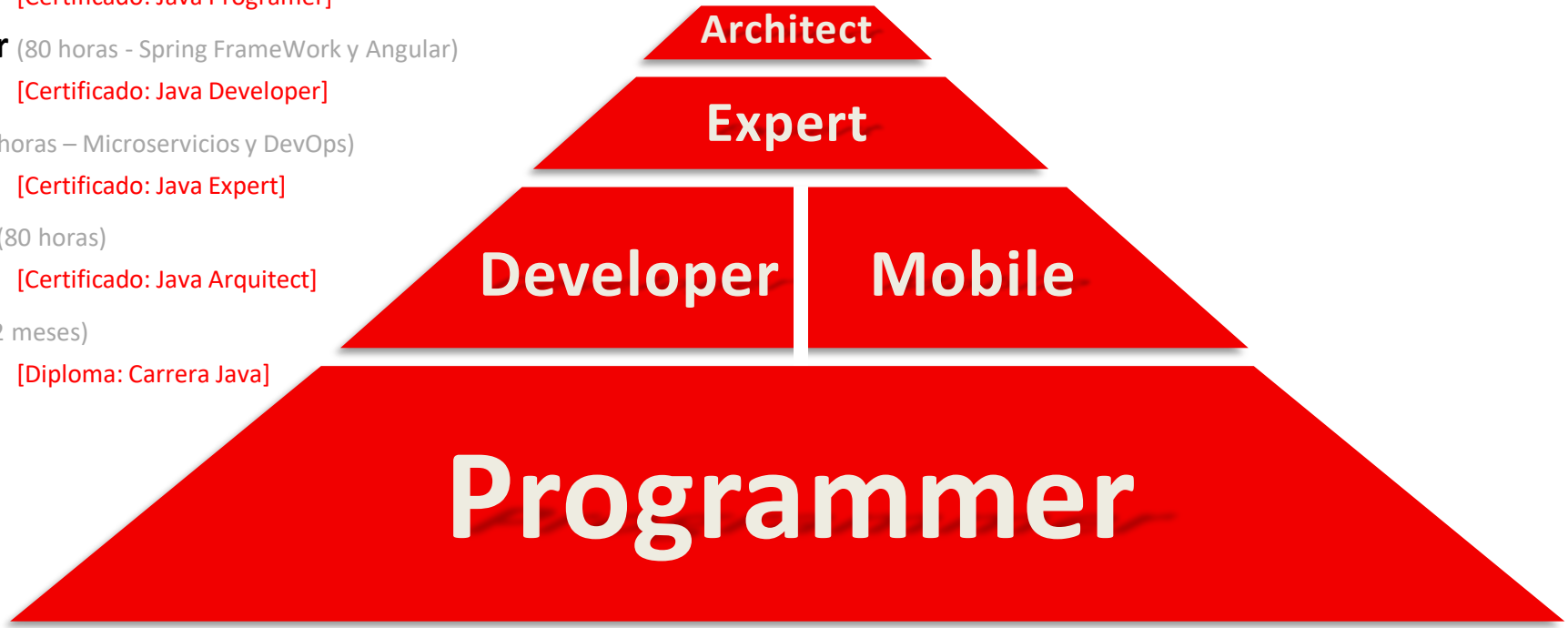
Programer (80 horas - Certificación Java 11)
[Certificado: Java Programer]

Developer (80 horas - Spring FrameWork y Angular)
[Certificado: Java Developer]

Expert (80 horas – Microservicios y DevOps)
[Certificado: Java Expert]

Architect (80 horas)
[Certificado: Java Arquitect]

Carrera (12 meses)
[Diploma: Carrera Java]



CJAVA
siempre para apoyarte

Quienes Somos

Somos una organización orientada a desarrollar, capacitar e investigar tecnología JAVA a través de un prestigioso staff de profesionales a nivel nacional.

Contáctenos

Av. Arenales 395 oficina 405
Santa Beatriz - Lima 01 - Perú

Teléfono: 433-6948

RPC / WhatsApp: 932 656 459

Email: info@cjavaperu.com

Síguenos

/cjava.peru.1

/cjava_peru

/cjavaperu

Hilos con JAVA

CJAVA
siempre para apoyarte

Objetivos

Al finalizar esta lección, debería estar capacitado para:

- Describir la programación de tareas del sistema operativo
- Definir un thread
- Crear threads
- Gestionar threads
- Sincronizar threads que acceden a datos compartidos
- Identificar posibles problemas de threads

CJAVA
siempre para apoyarte

Programación de tareas

Los sistemas operativos modernos utilizan una multitarea preferente para asignar el tiempo de CPU a las aplicaciones.

Existen dos tipos de tareas que se pueden programar para la ejecución:

- Procesos: un proceso es un área de la memoria que contiene tanto código como datos. Un proceso tiene un thread de ejecución que se programa para recibir porciones de tiempo de CPU.
- Thread: un thread es la ejecución programada de un proceso. Es posible que existan threads simultáneos. Todos los threads de un proceso comparten la misma memoria de datos, pero es posible que sigan diferentes rutas de acceso a través de una sección de código.

CJAVA
siempre para apoyarte

Importancia de los threads

Para ejecutar un programa tan rápido como sea posible, debe evitar cuellos de botella de rendimiento. Algunos de estos cuellos de botella son:

- Contención de recursos: dos o más tareas esperan uso exclusivo de un recurso
- Operaciones de E/S de bloqueo: no realizar ninguna acción a la espera de las transferencias de datos de disco o de red
- Infrautilización de CPU: una aplicación de thread único utiliza solo una CPU única

CJAVA
siempre para apoyarte

Clase Thread

La clase Thread se utiliza para crear e iniciar threads. El código que va a ejecutar un thread se debe colocar en una clase, lo que:

- Amplía la clase Thread
  - Código más simple
- Implanta la interfaz Runnable
  - Más flexible
  - extends es aún libre

CJAVA
siempre para apoyarte

Ampliación de Thread

Ampliar java.lang.Thread y sustituir el método run:

```
public class ExampleThread extends Thread {
    @Override
    public void run() {
        for(int i = 0; i < 100; i++) {
            System.out.println("i:" + i);
        }
    }
}
```

Inicio de Thread

Tras crear un nuevo Thread, se debe iniciar llamando al método Thread start:

```
public static void main(String[] args) {
    ExampleThread t1 = new ExampleThread();
    t1.start();
}
```

Programa el método run que se va a llamar.

Implantación de Runnable

Implantar java.lang.Runnable y el método run:

```
public class ExampleRunnable implements Runnable {
    @Override
    public void run() {
        for(int i = 0; i < 100; i++) {
            System.out.println("i:" + i);
        }
    }
}
```

Ejecución de instancias Runnable

Tras crear un nuevo Runnable, se debe transferir a un constructor de Thread. El método Thread start comienza la ejecución:

```
public static void main(String[] args) {
    ExampleRunnable r1 = new ExampleRunnable();
    Thread t1 = new Thread(r1);
    t1.start();
}
```

Runnable con datos compartidos

Los threads comparten potencialmente campos estáticos y de instancia.

```
public class ExampleRunnable implements Runnable {
    private int i;

    @Override
    public void run() {
        for(i = 0; i < 100; i++) {
            System.out.println("i:" + i);
        }
    }
}
```

Variable compartida potencialmente

Un ejecutable: varios threads

Varios threads que hacen referencia a un objeto pueden producir que se acceda de forma simultánea a los campos de instancia.

```
public static void main(String[] args) {
    ExampleRunnable r1 = new ExampleRunnable();
    Thread t1 = new Thread(r1);
    t1.start();
    Thread t2 = new Thread(r1);
    t2.start();
}
```

Una única instancia Runnable

Problemas con datos compartidos

A los datos compartidos se debe acceder con cuidado. Campos de instancia y estáticos:

- Se crean en un área de memoria conocida como espacio de pila.
- Cualquier thread los puede compartir de forma potencial.
- Varios threads los pueden cambiar de forma simultanea.
  - No hay ningún compilador o advertencias de IDE.
  - El acceso “de forma segura” a campos compartidos es su responsabilidad.

Las diapositivas anteriores pueden producir lo siguiente:

i:0,i:0,i:1,i:2,i:3,i:4,i:5,i:6,i:7,i:8,i:9,i:10,i:12,i:11 ...

El cero se ha producido dos veces

Fuera de la secuencia

Datos no compartidos

Algunos tipos de variables nunca se comparten. Los siguientes tipos siempre tienen protección de thread:

- Variables locales
- Parámetros del método
- Parámetros de manejador de excepciones

Operaciones atómicas

Las operaciones atómicas funcionan como una operación única. Una única sentencia en el lenguaje Java no siempre es atómica.

- i++;
  - Crea una copia temporal del valor en i.
  - Incrementa la copia temporal.
  - Anota el nuevo valor en i.
- l = 0xffff_ffff_ffff_ffff;
  - Es posible acceder a variables de 64 bits mediante dos operaciones de 32 bits independientes.

¿Qué inconsistencias podrían encontrar dos threads al incrementar el mismo campo?

¿Qué ocurre si dicho campo es largo?

Ejecución desordenada

- Las operaciones realizadas en un thread puede parecer que no se ejecutan en orden al observar los resultados desde otro thread.
  – La optimización de código puede dar como resultado una operación desordenada.
  – Los threads funcionan en copias almacenadas en caché de variables compartidas.
- Para garantizar el comportamiento consistente en los threads, debe sincronizar sus acciones.
  – Necesita una forma de establecer que una acción ocurra antes que otra.
  – Necesita una forma de vaciar de nuevo los cambios en variables compartidas en la memoria principal.

CJAVA
siempre para apoyarte

Palabra clave volatile

Un campo puede tener el modificador volatile aplicado:

```
public volatile int i;
```

- La lectura o escritura de un campo volatile hará que un thread sincronice su memoria de trabajo con la memoria principal.
- volatile no significa atómico.
  – Si i es volatile, i++ todavía no es una operación con protección de thread.

Parada de un thread

Un thread se para al terminar su método run.

```
public class ExampleRunnable implements Runnable {
    public volatile boolean timeToQuit = false;

    @Override
    public void run() {
        System.out.println("Thread started");
        while(!timeToQuit) {
            // ...
        }
        System.out.println("Thread finishing");
    }
}
```

Variable volátil compartida

CJAVA
siempre para apoyarte

Parada de un thread

```
public static void main(String[] args) {
    ExampleRunnable r1 = new ExampleRunnable();
    Thread t1 = new Thread(r1);
    t1.start();
    // ...
    r1.timeToQuit = true;
}
```

CJAVA
siempre para apoyarte

Palabra clave volatile

La palabra clave synchronized se utiliza para crear bloques de código con protección de thread. Un bloque de código synchronized:

- Hace que un thread escriba todos sus cambios en la memoria principal cuando se alcanza el final del bloque.
  - Similar a volatile
- Se usa para agrupar bloques de código para una ejecución exclusiva.
  - Bloque de threads hasta que obtienen acceso exclusivo
  - Resuelve el problema atómico

CJAVA
siempre para apoyarte

Métodos synchronized

```
public class ShoppingCart {
    private List<Item> cart = new ArrayList<>();
    public synchronized void addItem(Item item) {
        cart.add(item);
    }
    public synchronized void removeItem(int index) {
        cart.remove(index);
    }
    public synchronized void printCart() {
        Iterator<Item> ii = cart.iterator();
        while(ii.hasNext()) {
            Item i = ii.next();
            System.out.println("Item:" + i.getDescription());
        }
    }
}
```

CJAVA
siempre para apoyarte

Bloques synchronized

```
public void printCart() {
    StringBuilder sb = new StringBuilder();
    synchronized (this) {
        Iterator<Item> ii = cart.iterator();
        while (ii.hasNext()) {
            Item i = ii.next();
            sb.append("Item:");
            sb.append(i.getDescription());
            sb.append("\n");
        }
    }
    System.out.println(sb.toString());
}
```

CJAVA
siempre para apoyarte

Bloqueo de supervisión de objeto

Cada objeto en Java se asocia a una supervisión, que un thread puede bloquear o desbloquear.

- Los métodos synchronized utilizan la supervisión para el objeto this.
- Los métodos static synchronized utilizan la supervisión de clases.
- Los bloques synchronized deben especificar qué supervisión del objeto bloquear o desbloquear.

```
synchronized ( this ) { }
```

- Los bloques synchronized pueden ser anidados.

Detección de interrupción

La interrupción de un thread es otra posible forma de solicitar que un thread pare la ejecución.

```
public class ExampleRunnable implements Runnable {
    @Override
    public void run() {
        System.out.println("Thread started");
        while(!Thread.interrupted()) {
            // ...
        }
        System.out.println("Thread finishing");
    }
}
```

Método Thread estático

Interrupción de un thread

Cada thread tiene un método interrupt() e isInterrupted().

```
public static void main(String[] args) {
    ExampleRunnable r1 = new ExampleRunnable();
    Thread t1 = new Thread(r1);
    t1.start();
    // ...
    t1.interrupt();
}
```

Interrupción de un thread

Thread.sleep()

Un Thread puede realizar una ejecución durante un tiempo.

```
long start = System.currentTimeMillis();
try {
    Thread.sleep(4000);
} catch (InterruptedException ex) {
    // What to do?
}
long time = System.currentTimeMillis() - start;
System.out.println("Slept for " + time + " ms");
```

interrupt() se llama durante la suspensión

CJAVA
siempre para apoyarte

Métodos Thread adicionales

- Existen muchos más Thread y métodos relacionados con thread:
  - setName(String), getName() y getId()
  - isAlive(): ¿Ha finalizado un thread?
  - isDaemon() y setDaemon(boolean): JVM puede cerrarse mientras los threads del daemon están en ejecución.
  - join(): un thread actual espera que otro thread finalice.
  - Thread.currentThread(): las instancias Runnable pueden recuperar la instancia Thread que actualmente esté en ejecución.
- La clase Object también tiene métodos relacionados con thread:
  - wait(), notify() y notifyAll(): los threads se pueden suspender durante un tiempo indeterminado, reactivándose solo cuando el Object esperado recibe una notificación de reactivación.

CJAVA
siempre para apoyarte

Métodos a evitar

Se deben evitar algunos métodos Thread:

- setPriority(int) y getPriority()
  - Es posible que no causen ningún impacto o que provoquen problemas
- Los siguientes métodos están anticuados y nunca se deben usar:
  - destroy()
  - resume()
  - suspend()
  - stop()

Interbloqueo

El interbloqueo se produce cuando dos o más threads se bloquean para siempre, en espera el uno del otro.

```
synchronized(obj1) {
        synchronized(obj2) {
        }
}
```

El thread 1 realiza una pausa después de bloquear la supervisión de obj1.

```
synchronized(obj2) {
        synchronized(obj1) {
        }
}
```

El thread 2 realiza una pausa después de bloquear la supervisión de obj2.

CJAVA
siempre para apoyarte

Resumen

En esta lección, debe haber aprendido a hacer lo siguiente:

- Describir la programación de tareas del sistema operativo
- Definir un thread
- Crear threads
- Gestionar threads
- Sincronizar threads que acceden a datos compartidos
- Identificar posibles problemas de threads

Simultaneidad

CJAVA
siempre para apoyarte

Objetivos

Al finalizar esta lección, debería estar capacitado para:

- Usar variables atómicas
- Usar ReentrantReadWriteLock
- Usar recopilaciones java.util.concurrent
- Describir las clases de sincronizador
- Usar ExecutorService para ejecutar tareas de forma simultánea
- Aplicar el marco Fork-Join

CJAVA
siempre para apoyarte

Paquete java.util.concurrent

Java 5 introdujo el paquete java.util.concurrent, que contiene clases que son útiles en la programación simultánea.

Sus funciones incluyen:

- Recopilaciones simultáneas
- Alternativas de sincronización y bloqueo
- Pools de threads
  - Pools de recuento de threads dinámicos y fijos disponibles
  - Metodología "divide y vencerás" paralela (Fork-Join), nueva en Java 7

Paquete java.util.concurrent.atomic

El paquete java.util.concurrent.atomic contiene clases que soportan la programación con protección de thread y bloqueo libre en variables únicas.

```
AtomicInteger ai = new AtomicInteger(5);
if(ai.compareAndSet(5, 42)) {
    System.out.println("Replaced 5 with 42");
}
```

Una operación atómica garantiza que el valor actual sea 5 y, a continuación, se defina en 42.

Paquete java.util.concurrent.locks

El paquete java.util.concurrent.locks es un marco para bloquear y esperar condiciones que es distinto de las supervisiones y sincronización incorporadas.

```
public class ShoppingCart {
    private final ReentrantReadWriteLock rwl =
        new ReentrantReadWriteLock();

    public void addItem(Object o) {
        rwl.writeLock().lock();
        // modify shopping cart
        rwl.writeLock().unlock();
    }
```

Bloqueo de único escritor, varios lectores

Write Lock

java.util.concurrent.locks

```
public String getSummary() {
        String s = "";
        rwl.readLock().lock();
        // read cart, modify s
        rwl.readLock().unlock();
        return s;
    }
    public double getTotal() {
        // another read-only method
    }
}
```

Read Lock

Todos los métodos de solo lectura se pueden ejecutar de forma simultánea.

CJAVA
siempre para apoyarte

Recopilaciones con protección de thread

Las recopilaciones de java.util no tienen protección de thread. Para usar recopilaciones en modo de protección de thread:

- Usar bloques de código sincronizados para todos los accesos a una recopilación si se realizan escrituras
- Crear un envoltorio sincronizado mediante métodos de biblioteca, como java.util.Collections.synchronizedList(List<T>)
- Usar recopilaciones java.util.concurrent

Nota: el hecho de que una Collection se cree con protección de thread, no hace que sus elementos tengan protección de thread.

CJAVA
siempre para apoyarte

Sincronizados

El paquete java.util.concurrent proporciona cinco clases que ayudan a las expresiones de sincronización con un objetivo especial común.

Clase	Descripción
Semaphore	Semaphore es una herramienta de simultaneidad clásica.
CountDownLatch	Utilidad todavía muy simple y muy común para bloquear hasta que se contenga un número determinado de señales, eventos o condiciones.
CyclicBarrier	Punto de sincronización multidireccional reajustable útil en algunos estilos de programación paralela.
Phaser	Proporciona una forma más flexible de barrera que se puede usar para controlar el cálculo en fases entre varios threads.
Exchanger	Permite a dos threads intercambiar objetos en un punto de encuentro y es útil en distintos diseños de pipeline.

java.util.concurrent.CyclicBarrier

CyclicBarrier es un ejemplo de categoría de sincronizador de clases proporcionada por java.util.concurrent.

```
final CyclicBarrier barrier = new CyclicBarrier(2);

new Thread() {
    public void run() {
        try {
            System.out.println("before await - thread 1");
            barrier.await();
            System.out.println("after await - thread 1");
        } catch (BrokenBarrierException|InterruptedException ex) {

        }
    }
}.start();
```

Dos threads deben permanecer en espera antes de que se puedan desbloquear.

No se puede alcanzar.

CJAVA
siempre para apoyarte

Alternativas de threads de alto nivel

Puede resultar difícil usar las API relacionadas con el Thread tradicional de forma correcta. Las alternativas incluyen:

- java.util.concurrent.ExecutorService, mecanismo de mayor nivel usado para ejecutar tareas
  - Puede crear y volver a usar objetos de Thread para el usuario.
  - Permite ejecutar el trabajo y comprobar los resultados en el futuro.
- Marco Fork-Join, servicio ExecutorService de extracción de trabajo especializado, nuevo en Java 7

CJAVA
siempre para apoyarte

java.util.concurrent.ExecutorService

ExecutorService se utiliza para ejecutar tareas.

- Elimina la necesidad de crear y gestionar threads de forma manual.
- Las tareas **se pueden** ejecutar en paralelo según la implantación de ExecutorService.
- Las tareas pueden ser:
  - java.lang.Runnable
  - java.util.concurrent.Callable
- La implantación de instancias se puede obtener con Executors.

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

java.util.concurrent.Callable

La interfaz Callable:

- Define una tarea ejecutada en ExecutorService.
- Es similar en naturaleza a Runnable, pero puede:
  - Devolver un resultado mediante genéricos.
  - Devolver una excepción comprobada.

```
package java.util.concurrent;
public interface Callable<V> {
    V call() throws Exception;
}
```

CJAVA
siempre para apoyarte

java.util.concurrent.Future

La interfaz Future se utiliza para obtener resultados de un método V call() de Callable.

```
Future<V> future = es.submit(callable);
//submit many callables
try {
    V result = future.get();
} catch (ExecutionException|InterruptedException ex) {

}
```

ExecutorService controla cuándo se ha realizado el trabajo.

Obtiene el resultado del método call de Callable (bloquea si es necesario).

Si Callable devuelve una Exception.

Cierre de ExecutorService

El cierre de ExecutorService es importante porque sus threads son threads de no daemons y evitarán que JVM se cierre.

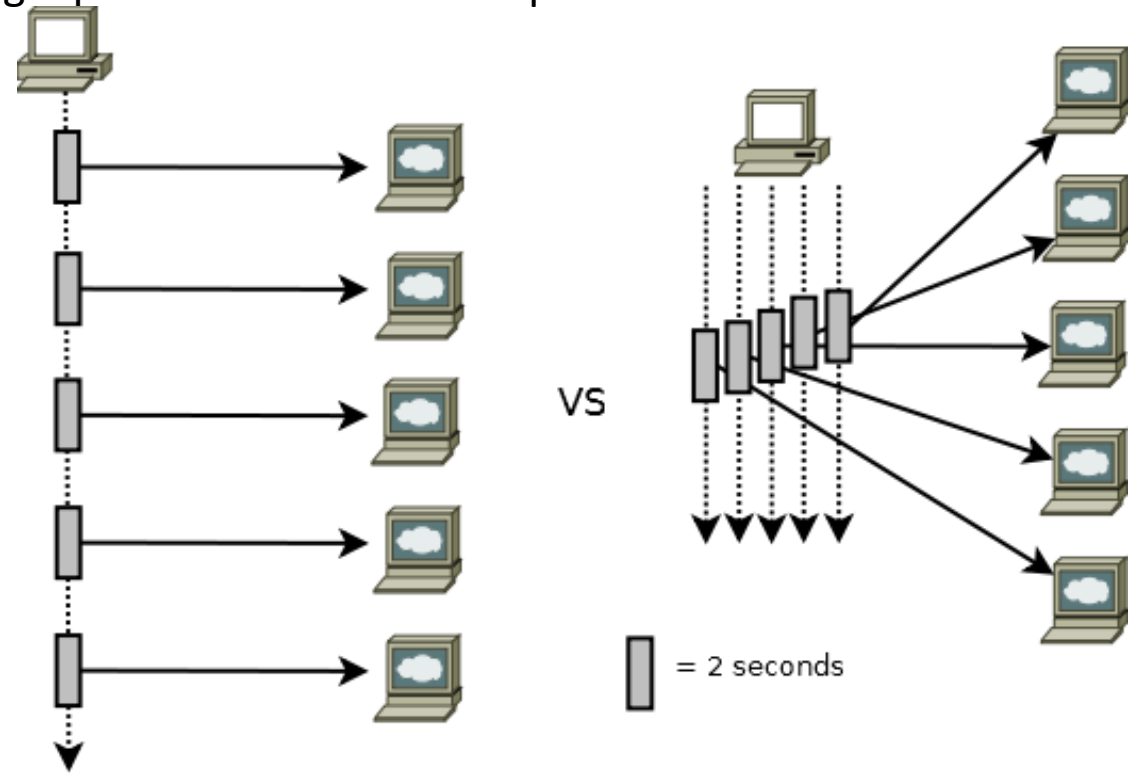
```
es.shutdown();

try {
    es.awaitTermination(5, TimeUnit.SECONDS);
} catch (InterruptedException ex) {
    System.out.println("Stopped waiting early");
}
```

Pare la aceptación de nuevos Callable.

Si desea esperar que las acciones Callable finalicen.

E/S simultánea

Las llamadas de bloqueo secuencial se ejecutan en una duración de tiempo más larga que las llamadas de bloqueo simultáneo.



Cliente de red de thread único

```
public class SingleThreadClientMain {
    public static void main(String[] args) {
        String host = "localhost";
        for (int port = 10000; port < 10010; port++) {
            RequestResponse lookup =
             new RequestResponse(host, port);
            try (Socket sock = new Socket(lookup.host, lookup.port);
                 Scanner scanner = new Scanner(sock.getInputStream());){
                lookup.response = scanner.next();
                System.out.println(lookup.host + ":" + lookup.port + " " +
                    lookup.response);
            } catch (NoSuchElementException|IOException ex) {
                System.out.println("Error talking to " + host + ":" +
                    port);
            }
        }
    }
}
```

CJAVA
siempre para apoyarte

Cliente de red multithread (parte 1)

```
public class MultiThreadedClientMain {
    public static void main(String[] args) {
        //ThreadPool used to execute Callables
        ExecutorService es = Executors.newCachedThreadPool();
        //A Map used to connect the request data with the result
        Map<RequestResponse,Future<RequestResponse>> callables =
            new HashMap<>();

        String host = "localhost";
        //loop to create and submit a bunch of Callable instances
        for (int port = 10000; port < 10010; port++) {
            RequestResponse lookup = new RequestResponse(host, port);
            NetworkClientCallable callable =
                new NetworkClientCallable(lookup);
            Future<RequestResponse> future = es.submit(callable);
            callables.put(lookup, future);
        }
```

Cliente de red multithread (parte 2)

```
//Stop accepting new Callables
        es.shutdown();

        try {
            //Block until all Callables have a chance to finish
            es.awaitTermination(5, TimeUnit.SECONDS);
        } catch (InterruptedException ex) {
            System.out.println("Stopped waiting early");
        }
```

Cliente de red multithread (parte 3)

```
for(RequestResponse lookup : callables.keySet()) {
            Future<RequestResponse> future = callables.get(lookup);
            try {
                lookup = future.get();
                System.out.println(lookup.host + ":" + lookup.port + " " +
                    lookup.response);
            } catch (ExecutionException|InterruptedException ex) {
                //This is why the callables Map exists
                //future.get() fails if the task failed
                System.out.println("Error talking to " + lookup.host +
                    ":" + lookup.port);
            }
        }
    }
}
```

Cliente de red multithread (parte 4)

```
public class RequestResponse {
    public String host; //request
    public int port; //request
    public String response; //response

    public RequestResponse(String host, int port) {
        this.host = host;
        this.port = port;
    }

    // equals and hashCode

}
```

Cliente de red multithread (parte 5)

```
public class NetworkClientCallable implements Callable<RequestResponse> {
    private RequestResponse lookup;

    public NetworkClientCallable(RequestResponse lookup) {
        this.lookup = lookup;
    }

    @Override
    public RequestResponse call() throws IOException {
        try (Socket sock = new Socket(lookup.host, lookup.port);
             Scanner scanner = new Scanner(sock.getInputStream());) {
            lookup.response = scanner.next();
            return lookup;
        }
    }
}
```

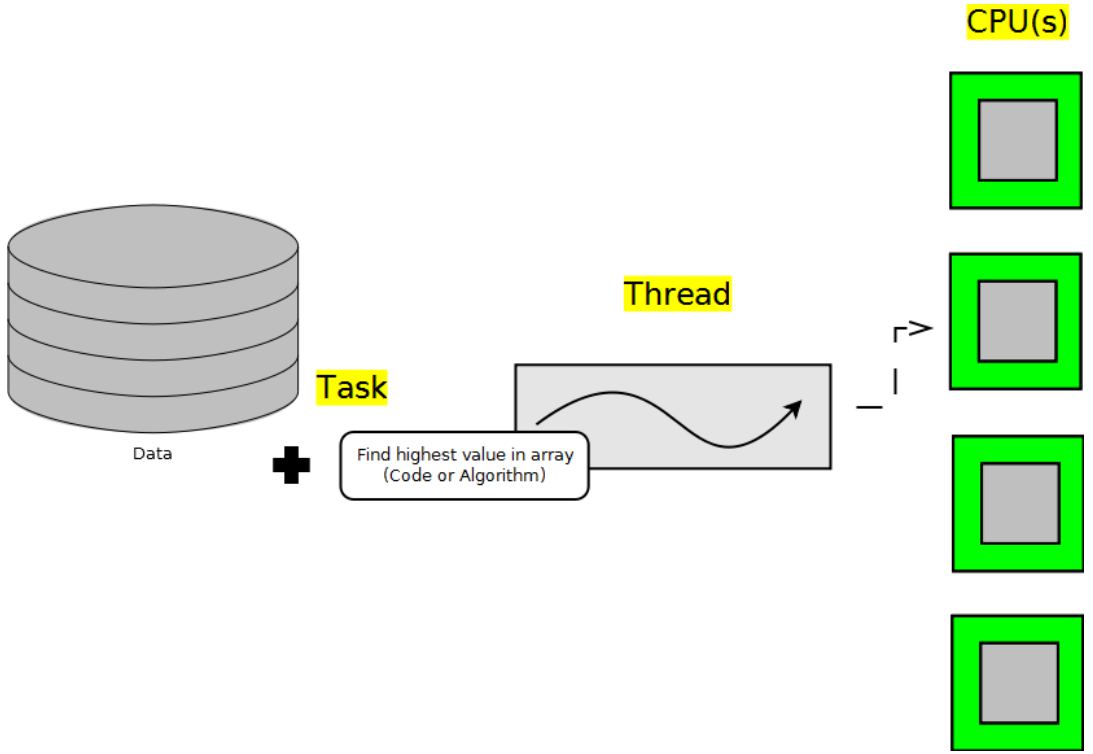
Paralelismo

Los sistemas modernos contienen varias CPU. Para sacar partido de la potencia de procesamiento en un sistema es preciso ejecutar tareas en paralelo en varias CPU.

- Divide y vencerás: una tarea se debe dividir en subtareas. Debe intentar identificar aquellas subtareas que se puedan ejecutar en paralelo.
- Puede ser difícil ejecutar algunos problemas como tareas paralelas.
- Algunos problemas son más sencillos. Los servidores que soportan varios clientes pueden usar una tarea independiente para manejar cada cliente.
- Tenga cuidado con el hardware. La programación de demasiadas tareas paralelas puede afectar de forma negativa al rendimiento.

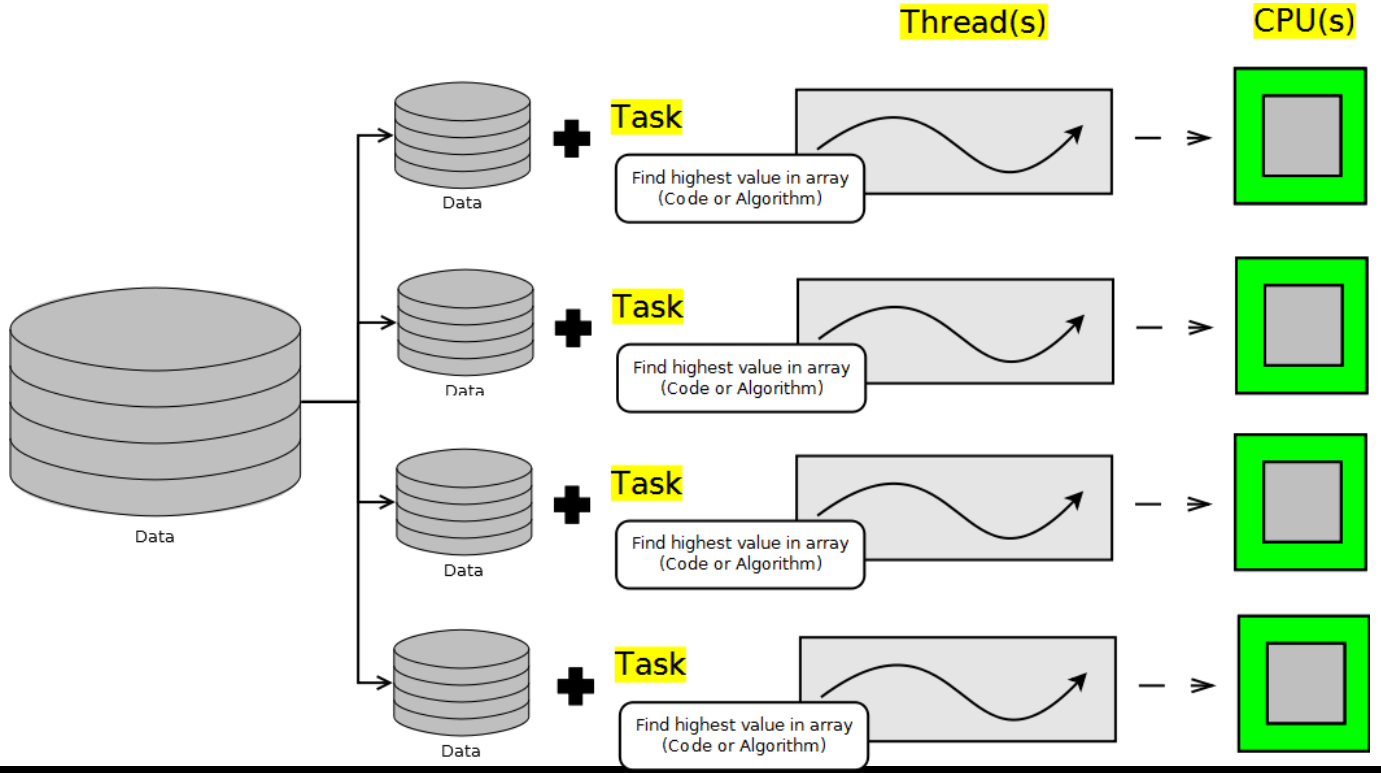
Sin paralelismo

Los sistemas modernos contienen varias CPU. Si no aprovecha los threads de alguna forma, solo se utilizará una parte de la potencia de procesamiento del sistema.



Paralelismo Naive

Una solución paralela simple divide los datos que se van a procesar en varios juegos.
Un juego de datos para cada CPU y un thread para procesar cada juego de datos.

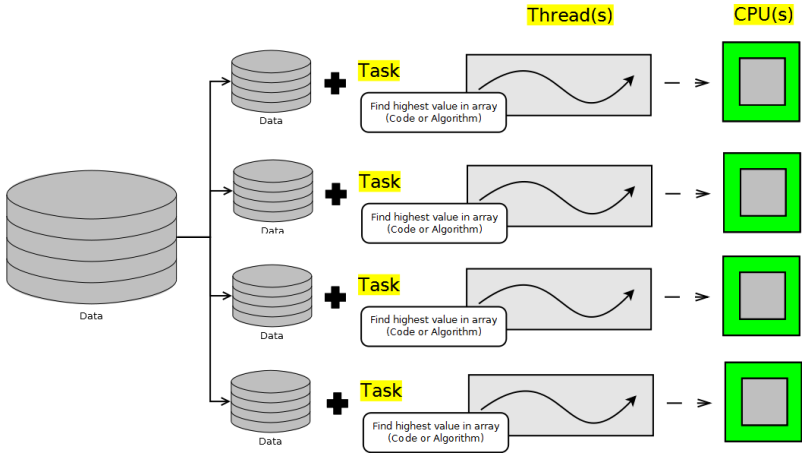


La necesidad de un marco Fork-Join

La división de juegos de datos en subjuegos con el mismo tamaño para cada thread que se va a procesar tiene un par de problemas.

Lo ideal es que todas las CPU se utilicen completamente hasta que la tarea finalice pero:

- Las CPU se pueden ejecutar a diferentes velocidades.
- Las tareas que no son de Java requieren tiempo de CPU y pueden reducir el tiempo del que dispone un thread de Java para la ejecución en una CPU.
- Los datos que se analizan pueden requerir diferentes cantidades de tiempo para el proceso.

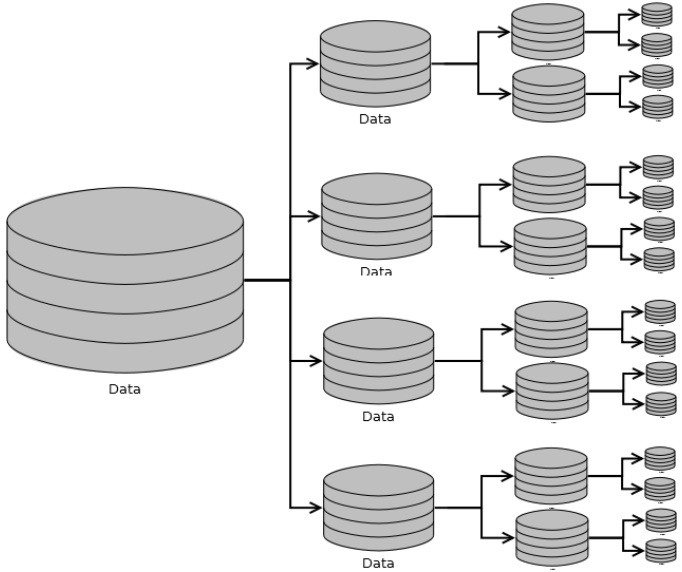


Extracción de trabajo

Para mantener varios threads ocupados:

- Divida los datos que se van a procesar en un gran número de subjuegos.
- Asigne los subjuegos de datos a una cola de procesamiento de threads.
- Cada thread tendrá muchos subjuegos en cola.

Si un thread finaliza todos sus subjuegos pronto, puede “extraer” subjuegos de otro thread.



Ejemplo de thread único

```
int[] data = new int[1024 * 1024 * 256]; //1G

for (int i = 0; i < data.length; i++) {
    data[i] = ThreadLocalRandom.current().nextInt();
}

int max = Integer.MIN_VALUE;
for (int value : data) {
    if (value > max) {
        max = value;
    }
}
System.out.println("Max value found:" + max);
```

Juego de datos muy grande.

Llenar la matriz con valores.

Buscar de forma secuencial la matriz para el valor mayor.

CJAVA
siempre para apoyarte

java.util.concurrent.ForkJoinTask<V>

Un objeto ForkJoinTask representa una tarea que se va a ejecutar.

- Una tarea contiene el código y los datos que se van a procesar. Similar a Runnable o Callable.
- Un número pequeño de threads en un pool Fork-Join crea y procesa un gran número de tareas.
  - ForkJoinTask normalmente crea más instancias

  ForkJoinTask hasta que los datos que se van procesar se subdividen de forma adecuada.
- Los desarrolladores normalmente utilizan las siguientes subclases:
  - RecursiveAction: si una tarea no tiene que devolver un resultado.
  - RecursiveTask: si una tarea tiene que devolver un resultado.

Ejemplo de RecursiveTask

```
public class FindMaxTask extends RecursiveTask<Integer> {
    private final int threshold;
    private final int[] myArray;
    private int start;
    private int end;

    public FindMaxTask(int[] myArray, int start, int end,
int threshold) {
        // copy parameters to fields
    }
    protected Integer compute() {
        // shown later
    }
}
```

Tipo de resultado de la tarea.

Datos a procesar.

Dónde se realiza el trabajo. Observe el tipo de devolución genérica.

Estructura de compute

```
protected Integer compute() {
    if DATA_SMALL_ENOUGH {
        PROCESS_DATA
        return RESULT;
    } else {
        SPLIT_DATA_INTO_LEFT_AND_RIGHT_PARTS
        TASK t1 = new TASK(LEFT_DATA);
        t1.fork();
        TASK t2 = new TASK(RIGHT_DATA);
        return COMBINE(t2.compute(), t1.join());
    }
}
```

Ejecución asíncrona

Proceso en el thread actual

Bloquear hasta que se termine

Ejemplo de compute (por debajo del umbral)

```
protected Integer compute() {
    if (end - start < threshold) {
        int max = Integer.MIN_VALUE;
        for (int i = start; i <= end; i++) {
            int n = myArray[i];
            if (n > max) {
                max = n;
            }
        }
        return max;
    } else {
        // split data and create tasks
    }
}
```

Umbral decidido por el usuario

Rango en la matriz

Ejemplo de compute (por encima del umbral)

```
protected Integer compute() {
    if (end - start < threshold) {
        // find max
    } else {
        int midway = (end - start) / 2 + start;
        FindMaxTask a1 =
    new FindMaxTask(myArray, start, midway, threshold);
        a1.fork();
        FindMaxTask a2 =
    new FindMaxTask(myArray, midway + 1, end, threshold);
        return Math.max(a2.compute(), a1.join());
    }
}
```

Tarea para la mitad izquierda de los datos

Tarea para la mitad derecha de los datos

Ejemplo de ForkJoinPool

ForkJoinPool se utiliza para ejecutar ForkJoinTask. Crea un thread para cada CPU en el sistema por defecto.

```
ForkJoinPool pool = new ForkJoinPool();
FindMaxTask task =
 new FindMaxTask(data, 0, data.length-1, data.length/16);
Integer result = pool.invoke(task);
```

El método compute de la tarea se llama automáticamente.

CJAVA
siempre para apoyarte

Recomendaciones del marco Fork-Join

- Evite operaciones de bloqueo o E/S.
  – Solo se crea un thread por CPU por defecto. Las operaciones de bloqueo evitarán el uso de todos los recursos de CPU.
- Conozca el hardware.
  – Una solución Fork-Join se ejecutará de forma más lenta en un sistema de una CPU que en una solución secuencial estándar.
  – Algunas CPU aumentan la velocidad solo cuando usan un único núcleo, lo que podría compensar de forma potencial cualquier aumento de rendimiento proporcionado por Fork-Join.
- Conozca el problema.
  – Muchos de los problemas tienen una sobrecarga adicional si se ejecutan en paralelo (ordenación paralela, por ejemplo).

CJAVA
siempre para apoyarte

Resumen

En esta lección, debe haber aprendido a hacer lo siguiente:

- Usar variables atómicas
- Usar ReentrantReadWriteLock
- Usar recopilaciones java.util.concurrent
- Describir las clases de sincronizador
- Usar ExecutorService para ejecutar tareas de forma simultánea
- Aplicar el marco Fork-Join

Java Date/Time API

CJAVA
siempre para apoyarte

Objectives

After completing this lesson, you should be able to:

- Create and manage date-based events
- Create and manage time-based events
- Combine date and time into a single object
- Work with dates and times across time zones
- Manage changes resulting from daylight savings
- Define and create timestamps, periods, and durations
- Apply formatting to local and zoned dates and times

CJAVA
siempre para apoyarte

Why Is Date and Time Important?

In the development of applications, programmers often need to represent time and use it to perform calculations:

- The current date and time (locally)
- A date and/or time in the future or past
- The difference between two dates/time in seconds, minutes, hours, days, months, years
- The time or date in another country (time zone)
- The correct time after daylight savings time is applied
- The number of days in the month of February (leap years)
- A time duration (hours, mins, secs) or a period (years, months, days)

CJAVA
siempre para apoyarte

Previous Java Date and Time

Disadvantages of `java.util.Date` (`Calendar`, `TimeZone` & `DateFormat`):

- Does not support fluent API approach
- Instances are mutable – not compatible with lambda
- Not thread-safe
- Weakly typed calendars
- One size fits all

CJAVA
siempre para apoyarte

Java Date and Time API: Goals

- The classes and methods should be straightforward.
- The API should support a fluent API approach.
- Instances of time/date objects should be immutable. (This is important for lambda operations.)
- Use ISO standards to define date and time.
- Time and date operations should be thread-safe.
- The API should support strong typing, which makes it much easier to develop good code first. (The compiler is your friend!)
- `toString` will always return a human-readable format.
- Allow developers to extend the API easily.

CJAVA
siempre para apoyarte

Working with Local Date and Time

The `java.time` API defines two classes for working with local dates and times (without a time zone):

- `LocalDate`:
  - Does not include time
  - A year-month-day representation
  - `toString` – ISO 8601 format (YYYY-MM-DD)
- `LocalTime`:
  - Does not include date
  - Stores hours:minutes:seconds.nanoseconds
  - `toString` – (HH:mm:ss.SSSS)

CJAVA
siempre para apoyarte

Working with `LocalDate`

`LocalDate` is a class that holds an event date: a birth date, anniversary, meeting date, and so on.

A date is a label for a day.

`LocalDate` uses the ISO calendar by default.

`LocalDate` does not include time, so it is portable across time zones.

You can answer the following questions about dates with `LocalDate`:

- Is it in the future or past?
- Is it in a leap year?
- What day of the week is it?
- What is the day a month from now?
- What is the date next Tuesday?

LocalDate: Example

```
import java.time.LocalDate;
import static java.time.temporal.TemporalAdjusters.*;
import static java.time.DayOfWeek.*;
import static java.lang.System.out;

public class LocalDateExample {

  public static void main(String[] args) {
    LocalDate now, bDate, nowPlusMonth, nextTues;
    now = LocalDate.now();
    out.println("Now: " + now);
    bDate = LocalDate.of(1995, 5, 23); // Java's Birthday
    out.println("Java's Bday: " + bDate);
    out.println("Is Java's Bday in the past? " + bDate.isBefore(now));
    out.println("Is Java's Bday in a leap year? " + bDate.isLeapYear());
    out.println("Java's Bday day of the week: " + bDate.getDayOfWeek());
    nowPlusMonth = now.plusMonths(1);
    out.println("The date a month from now: " + nowPlusMonth);
    nextTues = now.with(next(TUESDAY));
    out.println("Next Tuesday's date: " + nextTues);
  }
}
```

`next` method

`TUESDAY`

`LocalDate` objects are immutable – methods return a new instance.

Working with `LocalTime`

`LocalTime` stores the time within a day.

- Measured from midnight
- Based on a 24-hour clock (13:30 is 1:30 PM.)
- Questions you can answer about time with `LocalTime`
  - When is my lunch time?
  - Is lunch time in the future or past?
  - What is the time 1 hour 15 minutes from now?
  - How many minutes until lunch time?
  - How many hours until bedtime?
  - How do I keep track of just the hours and minutes?

LocalTime: Example

```
import java.time.LocalTime;
import static java.time.temporal.ChronoUnit.*;
import static java.lang.System.out;

public class LocalTimeExample {
  public static void main(String[] args) {
    LocalTime now, nowPlus, nowHrsMins, lunch, bedtime;
    now = LocalTime.now();
    out.println("The time now is: " + now);
    nowPlus = now.plusHours(1).plusMinutes(15);
    out.println("What time is it 1 hour 15 minutes from now? " + nowPlus);
    nowHrsMins = now.truncatedTo(MINUTES);
    out.println("Truncate the current time to minutes: " + nowHrsMins);
    out.println("It is the " + now.toSecondOfDay()/60 + "th minute");
    lunch = LocalTime.of(12, 30);
    out.println("Is lunch in my future? " + lunch.isAfter(now));
    long minsToLunch = now.until(lunch, MINUTES);
    out.println("Minutes til lunch: " + minsToLunch);
    bedtime = LocalTime.of(21, 0);
    long hrsToBedtime = now.until(bedtime, HOURS);
    out.println("How many hours until bedtime? " + hrsToBedtime);
  }
}
```

HOURS, MINUTES

CJAVA
siempre para apoyarte

Working with LocalDateTime

`LocalDateTime` is a combination of `LocalDate` and `LocalTime`.

- `LocalDateTime` is useful for narrowing events.
- You can answer the following questions with `LocalDateTime`:
  - When is the meeting with corporate?
  - When does my flight leave?
  - When does the course start?
  - If I move the meeting to Friday, what is the date?
  - If the course starts at 9 AM on Monday and ends at 5 PM on Friday, how many hours am I in class?

LocalTimeDate: Example

```
import java.time.*;
import static java.time.Month.*;
import static java.time.temporal.ChronoUnit.*;
import static java.lang.System.out;

public class LocalDateTimeExample {
  public static void main(String[] args) {
    LocalDateTime meeting, flight, courseStart, courseEnd;
    meeting = LocalDateTime.of(2014, MARCH, 21, 13, 30);
    out.println("Meeting is on: " + meeting);
    LocalDate flightDate = LocalDate.of(2014, MARCH, 31);
    LocalTime flightTime = LocalTime.of(21, 45);
    flight = LocalDateTime.of(flightDate, flightTime);
    out.println("Flight leaves: " + flight);
    courseStart = LocalDateTime.of(2014, MARCH, 24, 9, 00);
    courseEnd = courseStart.plusDays(4).plusHours(8);
    out.println("Course starts: " + courseStart);
    out.println("Course ends:   " + courseEnd);
    long courseHrs = (courseEnd.getHour() - courseStart.getHour()) *
                     (courseStart.until(courseEnd, DAYS) + 1);
    out.println("Course is:     " + courseHrs + " hours long.");
  }
}
```

LocalDateTime, LocalDate, LocalTime

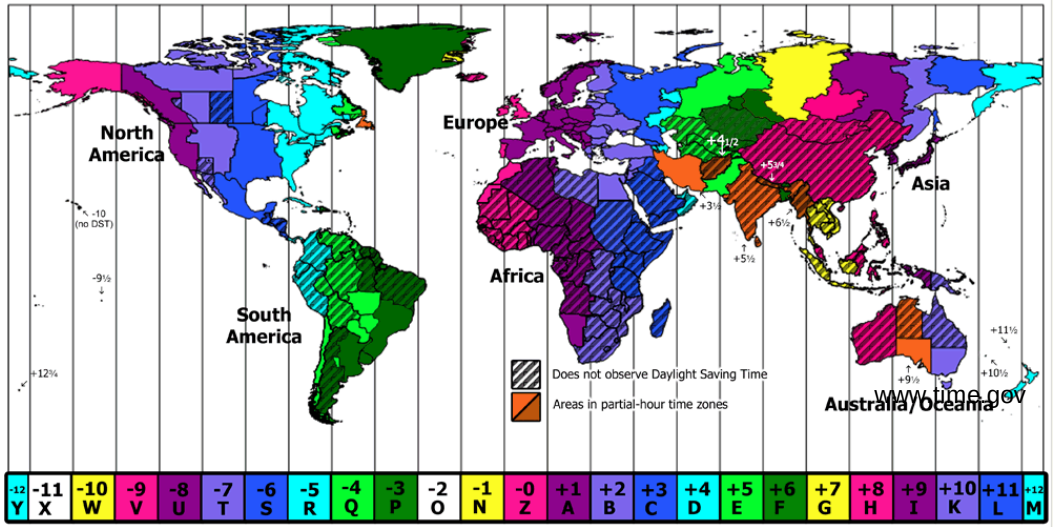
MARCH

Combine `LocalDate` and `LocalTime` objects.

Working with Time Zones

Time zones are geographic, but the time in a specific location is defined by the government in that location.

When a country (and sometimes a state) observes changes (for daylight savings) varies.



Daylight Savings Time Rules

Time changes result in a local hour gap/overlap:

Sunday, March 9, 2014 (New York)	Local time	UTC Offset
	1:59:58 AM	UTC-5h EST
	1:59:59 AM	UTC-5h EST
	2:00:00 -> 3:00:00	UTC-4h EDT
	3:00:01 AM	UTC-4h EDT

Starting DST causes a one hour gap.

Sunday, November 2, 2014 (New York)	Local time	UTC Offset
	1:59:58 AM	UTC-4h EST
	1:59:59 AM	UTC-4h EST
	2:00:00 -> 1:00:00	UTC-5h EDT
	1:00:01 AM	UTC-5h EDT

Ending DST causes a one hour overlap.

Modeling Time Zones

- `ZoneId`: Is a specific location or offset relative to UTC

```
ZoneId nyTZ = ZoneId.of("America/New_York");
ZoneId EST = ZoneId.of("US/Eastern");
ZoneId Romeo = ZoneId.of("Europe/London");
```

- `ZoneOffset`: Extends `ZoneId`; specifies the actual time difference from UTC

```
ZoneOffset USEast = ZoneOffset.of("-5");
ZoneOffset Nepal = ZoneOffset.ofHoursMinutes(5, 45);
ZoneId EST = ZoneId.ofOffset("UTC", USEast);
```

- `ZoneRules`: Is the class used to determine offsets

Creating ZonedDateTime Objects

Stores `LocalDateTime`, `ZoneId`, and `ZoneOffset`

```
ZoneId USEast = ZoneId.of("America/New_York");
LocalDate date = LocalDate.of(2014, MARCH, 23);
LocalTime time = LocalTime.of(9, 30);
LocalDateTime dateTime = LocalDateTime.of(date, time);
ZonedDateTime courseStart = ZonedDateTime.of(date, time, USEast);
ZonedDateTime hereNow = ZonedDateTime.now(USEast).truncatedTo(MINUTES);
System.out.println("Here now:          " + hereNow);
System.out.println("Course start:      " + courseStart);
ZonedDateTime newCourseStart = courseStart.plusDays(2).minusMinutes(30);
System.out.println("New Course Start: " + newCourseStart);
```

```
Here now:          2014-02-19 T 17:00 -05:00[America/New_York]
Course start:      2014-03-23 T 09:30 -04:00[America/New_York]
New Course Start: 2014-03-25 T 09:00 -04:00[America/New_York]
```

Space added to make the fields more clear

Working with ZonedDateTime Gaps/Overlaps

Given a meeting date the day before daylight savings (2AM on March 9th), what happens if the meeting is moved out by a day?

```
// DST Begins March 9th, 2014
LocalDate meetDate = LocalDate.of(2014, MARCH, 8);
LocalTime meetTime = LocalTime.of(16, 00);
ZonedDateTime meeting = ZonedDateTime.of(meetDate, meetTime, USEast);
System.out.println("meeting time:     " + meeting);
ZonedDateTime newMeeting = meeting.plusDays(1);
System.out.println("new meeting time: " + newMeeting
```

```
meeting time:     2014-03-08 16:00 -05:00[America/New_York]
new meeting time: 2014-03-09 16:00 -04:00[America/New_York]
```

The local time is not changed, and the offset is managed correctly.

ZoneRules

- Each time zone (`ZoneId`) has a set of rules that are part of the JDK.
- Date or times that land on time changes can be determined by using the rules.

```
// Ask the rules if there was a gap or overlap
ZoneId USEast = ZoneId.of("America/New_York");
LocalDateTime lateNight = LocalDateTime.of(2014, MARCH, 9, 2, 30);
ZoneOffsetTransition zot = USEast.getRules().getTransition(lateNight);
if (zot != null) {
  if (zot.isGap()) System.out.println("gap");
  if (zot.isOverlap()) System.out.println("overlap");
}
```

- Given the code above, what will print?

Working Across Time Zones

The `OffsetDateTime` class stores a `LocalDateTime` and `ZoneOffset`.

- This is useful for determining `ZonedDateTimes` across time zones.

```
LocalDateTime meeting = LocalDateTime.of(2014, JUNE, 13, 12, 30);
ZoneId SanFran = ZoneId.of("America/Los_Angeles");
ZonedDateTime staffCall = ZonedDateTime.of(meeting, SanFran);
OffsetDateTime = staffCall.toOffsetDateTime();
```

- The offset is used to calculate date/time using zone rules:

```
ZoneId London = ZoneId.of("Europe/London");
OffsetDateTime staffCallOffset = staffCall.toOffsetDateTime();
ZonedDateTime staffCallUK = staffCallOffset.atZoneSameInstant(London);
System.out.println("Staff call (Pacific) is at: " + staffCall);
System.out.println("Staff call (UK) is at:      " + staffCallLondon);
```

CJAVA
siempre para apoyarte

Date and Time Methods

Prefix	Example	Use
`now`	`today = LocalDate.now()`	Creates an instance using the system clock
`of`	`meet = LocalTime.of(13, 30)`	Creates an instance by using the parameters passed
`get`	`today.get(DAY_OF_WEEK)`	Returns part of the state of the target
`with`	`meet.withHour(12)`	Returns a copy of the target object with one element changed
`plus, minus`	`nextWeek.plusDays(7)` `sooner.minusMinutes(30)`	Returns a copy of the object with the amount added or subtracted
`to`	`meet.toSecondOfDay()`	Converts this object to another type. Here returns `int` seconds.
`at`	`today.atTime(13, 30)`	Combines this object with another; returns a `LocalDateTime` object
`until`	`today.until`	Calculates the amount of time until another date in terms of the unit
`isBefore, isAfter`	`today.isBefore(lastWeek)`	Compares this object with another on the timeline
`isLeapYear`	`today.isLeapYear()`	Checks if this object is a leap year

Date and Time Amounts

`Instant` – Stores an instant in time on the time-line

- Useful for: timestamps, e.g. login events
- Stored as seconds (`long`) and nanoseconds (`int`)
- Methods used to compare before and after

```
Instant now = Instant.now();
Thread.sleep(0,1); // long milliseconds, int nanoseconds
Instant later = Instant.now();
System.out.println("now is before later? " + now.isBefore(later));
System.out.println("Now:   " + now);
System.out.println("Later: " + later);
```

```
now is before later? true
Now:   2014-02-21 T 16:11:34.788 Z
Later: 2014-02-21 T 16:11:34.789 Z
```

`toString` includes nanoseconds to three digits

Period

`Period` is a class that holds a date-based amount.

- Years, months, and days based on the ISO-8601 calendar
- Plus and minus work with a conceptual day, thus preserving daylight savings changes

```
Period oneDay = Period.ofDays(1);
System.out.println("Period of one day: " + oneDay);
LocalDateTime beforeDST = LocalDateTime.of(2014, MARCH, 8, 12, 00);
ZonedDateTime newYorkTime =
  ZonedDateTime.of(beforeDST, ZoneId.of("America/New_York"));
System.out.println("Before: " + newYorkTime);
System.out.println("After:  " + newYorkTime.plus(oneDayYear));
```

```
Period of one day: P1D
Before: 2014-03-08 T 12:00 -05:00[America/New_York]
After:  2014-03-09 T 12:00 -04:00[America/New_York]
```

The time is preserved, because only "days" are added.

Duration

`Duration` is a class that stores a time-based amount.

Time is measured in actual seconds and nanoseconds.

Days are treated as 24 hours, and daylight savings is ignored.

```
Duration one24hourDay = Duration.ofDays(1);
System.out.println("Duration of one day: " + one24hourDay);
beforeDST = LocalDateTime.of(2014, MARCH, 8, 12, 00);
newYorkTime = ZonedDateTime.of(beforeDST, ZoneId.of("America/New_York"));
System.out.println("Before: " + newYorkTime);
System.out.println("After:  " + newYorkTime.plus(one24hourDay));
```

The time is not preserved because 24 hours are added.

```
Duration of one day: PT24H
Before: 2014-03-08 T 12:00 -05:00[America/New_York]
After:  2014-03-09 T 13:00 -04:00[America/New_York]
```

Calculating Between Days

`TemporalUnit` is an interface representing a unit of time.

Implemented by the `enum` class `ChronoUnit`

```
import static java.time.temporal.ChronoUnit.*;

LocalDate christmas = LocalDate.of(2014, DECEMBER, 25);
LocalDate today = LocalDate.now();
long days = DAYS.between(today, christmas);
System.out.println("There are " + days + " shopping days til Christmas");
```

`Period` also provides a `between` method

```
Period tilXMas = Period.between(today, christmas);
System.out.println("There are " + tilXMas.getMonths() +
                   " months and " + tilXMas.getDays() +
                   " days til Christmas");
```

Making Dates Pretty

`DateTimeFormatter` produces formatted date/times

- Using predefined constants, patterns letters, or a localized style

```
ZonedDateTime now = ZonedDateTime.now();
DateTimeFormatter formatter = DateTimeFormatter.ISO_LOCAL_DATE;
System.out.println(now.format(formatter));
formatter = DateTimeFormatter.ISO_ORDINAL_DATE;
System.out.println(now.format(formatter));
formatter = DateTimeFormatter.ofPattern("EEEE, MMMM dd, yyyy G, hh:mm a VV");
System.out.println(now.format(formatter));
formatter = DateTimeFormatter.ofLocalizedDateTime(FormatStyle.MEDIUM);
System.out.println(now.format(formatter));
```

Predefined `DateTimeFormatter` constants

String pattern

Format style

```
2014-02-21
2014-052-05:00
Friday, February 21, 2014 AD, 03:51 PM America/New_York
Feb 21, 2014 3:51:51 PM
```

Year and day of the year

`FormatStyle.MEDIUM`

Using Fluent Notation

One of the goals of JSR-310 was to make the API fluent.

Examples:

```
 // Not very readable - is this June 11 or November 6th?
LocalDate myBday = LocalDate.of(1970, 6, 11);

// A fluent approach
myBday = Year.of(1970).atMonth(JUNE).atDay(11);

// Schedule a meeting fluently
LocalDateTime meeting = LocalDate.of(2014, MARCH, 25).atTime(12, 30);

// Schedule that meeting using the London timezone
ZonedDateTime meetingUK = meeting.atZone(ZoneId.of("Europe/London"));

// What time is it in San Francisco for that meeting?
ZonedDateTime earlyMeeting =
  meetingUK.withZoneSameInstant(ZoneId.of("America/Los_Angeles"));
```

CJAVA
siempre para apoyarte

Summary

- In this lesson, you should have learned how to:
  - Create and manage date-based events
  - Create and manage time-based events
  - Combine date and time into a single object
  - Work with dates and times across time zones
  - Manage changes resulting from daylight savings
  - Define and create timestamps, periods and durations
  - Apply formatting to local and zoned dates and times

Contáctenos

Av. Arenales 395 oficina 405
Santa Beatriz - Lima 01 - Perú

Teléfono: 433-6948

RPC / WhatsApp: 932 656 459

Email: info@cjavaperu.com

Síguenos

/cjava.peru.1

/cjava_peru

/cjavaperu

CJAVA

siempre para apoyarte

#CJavaNoPara

Gracias