Programación Distribuida y Concurrente – TP 4 Async

Dimase, Matias; Isea, Maximiliano

1. **Explique la interfaz Future de Java, brinde usos, limitaciones y ejemplos.**

Un Future es un objeto que en algún momento contendrá el resultado de un método. Cuando se invoca devuelve inmediatamente un resultado (objeto Future) y sigue ejecutando de manera asíncrona. Cuando termina, retorna un resultado en el futuro. Un *Future* representa el resultado de un cálculo asincrónico.
Se proporcionan métodos para comprobar si el cálculo está completo, esperar a que se complete y recuperar el resultado del cálculo. Dicho resultado sólo se puede recuperar usando el método *get* cuando el cálculo se ha completado, bloqueando hasta que esté listo. La cancelación se realiza mediante el método *cancel* . Posee métodos adicionales para determinar si la tarea se completó o se canceló. Una vez completado un cálculo, este no se puede cancelar.

Método y Descripción	Modificador y Tipo
cancel(boolean mayInterruptIfRunning) Intenta cancelar la ejecución de la tarea.	boolean
get() Espera de ser necesario a que se completen los cálculos, luego retorna el resultado.	V
get(long timeout, TimeUnit unit) Espera de ser necesario el máximo tiempo proporcionado a que se completen los cálculos y retorna el resultado si está disponible.	V
isCanceled() Retorna *true* si la tarea fue cancelada antes de su finalización.	boolean
isDone() Retorna *true* si la tarea fue completada.	boolean

Limitaciones:

- No se puede completar manualmente.
- No tiene la capacidad de adjuntar una función de devolución de llamada al futuro y hacer que se llame automáticamente cuando el resultado del futuro esté disponible.
- No se pueden encadenar varios futuros
- No se pueden combinar varios futuros juntos: Digamos que tiene 10 futuros diferentes que desea ejecutar en paralelo y luego ejecutar alguna función después de que se completen todos.
- Future API no tiene ninguna construcción de manejo de excepciones.

Ejemplo:

```
//llamada al método asíncrono
Future<List<Client>> clients = clientRepository.finAll();

// ejecuto más instrucciones

List<Client> clientsResult = clients.get(); // en este punto se bloquea esperando el que Future termine
```

Fuentes:

https://docs.oracle.com/javase/7/docs/api/java/util/concurrent/Future.html

https://www.linkedin.com/pulse/java-8-future-vs-completablefuture-saral-saxena

2. **Explique la clase CompletableFuture, porque representa una ventaja sobre Future.**

   En Java 8, entre otras muchas cosas, se añadió la clase **CompletableFuture** .Esta nueva clase implementa la interfaz Future, pero aportándole más funcionalidad. Implementa también la nueva interfaz **CompletionStage** la cual contiene todos los nuevos métodos.

   CompletionStage:
   - Realiza una acción y retorna un valor cuando otro completion stage se completa.
   - Es un modelo para tareas que pueden desencadenar otras tareas.

   En resumen, es un elemento de una cadena. Cuando más de un thread intenta completar, completar excepcionalmente o cancelar un CompletableFuture, sólo uno de ellos lo logra.

   Puede usar el método CompletableFuture.complete () para completar manualmente, solucionando así una de las limitaciones de Future. Todos los clientes que esperan este futuro obtendrán el resultado especificado.

   Para ejecutar alguna tarea en segundo plano de forma asincrónica y no devolver nada de la tarea, existe el método CompletableFuture.runAsync (), el cual Toma un objeto Runnable y devuelve CompletableFuture <Void>.

   En caso de querer retornar algo, usamos CompletableFuture.supplyAsync (), el cual Toma un Proveedor <T> y devuelve CompletableFuture <T> donde T es el tipo de valor obtenido al llamar al proveedor dado.

   El método get() bloquea, es decir, el proceso espera hasta que se complete el futuro y devuelva el resultado. Pero para la creación de sistemas asíncronos debemos poder adjuntar un callback al CompletableFuture, que se llame automáticamente cuando se complete el futuro. Así, podemos implementar una promesa no bloqueante, esto con Future no lo podíamos realizar, por lo tanto, CompletableFuture llegó para solucionar las limitaciones que poseía Future.

   Fuentes: https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/CompletableFuture.html

   https://www.baeldung.com/java-completablefuture

   https://anotherdayanotherbug.wordpress.com/2016/03/07/futuros-en-java-parte-3-completablefuture-introduccion/

3. **Comente y brinde ejemplos de los metodos: whenComplete, thenAccept, thenRun, thenCompose y thenCombine de la clase CompletableFuture.**

   whenComplete(BiConsumer <? super T, ? super Throwable> action)

   whenComplete() no puede devolver resultados. Por lo tanto, se utiliza como devoluciones de llamada que no interfieren en la canalización de procesamiento de CompletionStages.

   Si hay una excepción no controlada proveniente de las etapas antes de la etapa 'whenComplete', esa excepción se pasa tal como está. En otras palabras, si el CompletionStage ascendente se completa excepcionalmente, el CompletionStage devuelto por whenComplete () también se completa excepcionalmente.

   Ejemplo:

   CompletableFuture<String> cf0 =

```
    CompletableFuture.failedFuture(new RuntimeException("Oops"));

CompletableFuture<String> cf1 =
    cf0.whenComplete((msg, ex) -> {
      if (ex != null) {
        System.out.println("Exception occurred");
      } else {
        System.out.println(msg);
      }
    });

try {
  cf1.join();
} catch (CompletionException e) {
  System.out.println("Error: " + e.getMessage());
}
```

thenAccept y thenRun

Están enfocados para los casos donde no queremos devolver nada de la función de devolución de llamada, sino que solo deseo ejecutar algún fragmento de código después de completar Future.

CompletableFuture.thenAccept() toma un Consumer <T> y regresa CompletableFuture<Void> y Tiene acceso al resultado del CompletableFuture al que está adjunto.

thenRun() no tiene acceso al resultado del Future. Toma un Runnable y regresa CompletableFuture<Void>

```
//Ejemplo thenAccept
CompletableFuture.supplyAsync(() -> {
        return ProductService.getProductDetail(productId);
}).thenAccept(product -> {
        System.out.println("Got product detail from remote service " + product.getName())
});

//ejemplo de thenRun
// thenRun() example
CompletableFuture.supplyAsync(() -> {
    // Run some computation
}).thenRun(() -> {
    // Computation Finished.
});
```

thenCompose y thenCombine

Si la devolución de llamada está devolviendo CompletableFuture, el resultado final ,en el caso anterior, es un CompletableFuture anidado. Si el resultado final es un Future de nivel superior y existe una dependencia entre ellos (uno debe ocurrir antes que el otro), uso el thenCompose().

Supongamos que deseo obtener los detalles de un usuario (de una API remota) y, una vez que los detalles del usuario están disponibles, obtener su calificación crediticia de otro servicio. implementaciones de getUserDetail() y getCreditRating()

```
CompletableFuture<User> getUsersDetail(String userId) {
        return CompletableFuture.supplyAsync(() -> {
                return UserService.getUserDetails(userId);
```

```
        });
}

CompletableFuture<Double> getCreditRating(User user) {
        return CompletableFuture.supplyAsync(() -> {
                return CreditRatingService.getCreditRating(user);
        });
}

CompletableFuture<Double> result = getUserDetail(userId)
.thenCompose(user -> getCreditRating(user));
```

Si bien thenCompose() se usa para combinar dos futuros donde un futuro depende del otro, thenCombine() se usa cuando desea que dos futuros se ejecuten de forma independiente y hagan algo después de que ambos estén completos.

Ejemplo de calculo de indice de masa corporal con thenCombine:

```
CompletableFuture<Double> weightInKgFuture = CompletableFuture.supplyAsync(() -> {
   try {
      TimeUnit.SECONDS.sleep(1);
   } catch (InterruptedException e) {
     throw new IllegalStateException(e);
   }
   return 65.0;
});

CompletableFuture<Double> heightInCmFuture = CompletableFuture.supplyAsync(() -> {
   try {
      TimeUnit.SECONDS.sleep(1);
   } catch (InterruptedException e) {
     throw new IllegalStateException(e);
   }
   return 177.8;
});

CompletableFuture<Double> combinedFuture = weightInKgFuture.thenCombine(heightInCmFuture,
(weightInKg, heightInCm) -> {
   Double heightInMeter = heightInCm/100;
   return weightInKg/(heightInMeter*heightInMeter);
});

System.out.println("Your BMI is - " + combinedFuture.get());
```

https://www.callicoder.com/java-8-completablefuture-tutorial/

4. **¿Cómo se gestionan las excepciones de Futuros CompletableFuture?**

Existen 3 formas de manejar excepciones en CompletableFuture, ellas son:

Exceptionally

El callback exceptionally brinda la oportunidad de recuperarse de los errores generados en el Future original. Puedo registrar la excepción y devolver un valor predeterminado. solo tiene acceso a la excepción, no al resultado. el error no se propagará más en la cadena de devolución de

llamada si lo maneja una vez. Si el futuro completable se completó con éxito, entonces se omitirá la lógica interna "exceptionally".
Ejemplo:

```
Integer age = -1;

CompletableFuture<String> maturityFuture = CompletableFuture.supplyAsync(() -> {
   if(age < 0) {
      throw new IllegalArgumentException("Age can not be negative");
   }
   if(age > 18) {
      return "Adult";
   } else {
      return "Child";
   }
}).exceptionally(ex -> {
   System.out.println("Oops! We have an exception - " + ex.getMessage());
   return "Unknown";
});
```

Handle

En el método handle(), se tiene acceso al resultado y la excepción del CompletableFuture actual como argumentos: puede transformar el resultado actual en otro resultado o recuperar la excepción.
Si ocurre una excepción, el res argumento será nulo; de lo contrario, el ex argumento será nulo.

```
Integer age = -1;

CompletableFuture<String> maturityFuture = CompletableFuture.supplyAsync(() -> {
   if(age < 0) {
      throw new IllegalArgumentException("Age can not be negative");
   }
   if(age > 18) {
      return "Adult";
   } else {
      return "Child";
   }
}).handle((res, ex) -> {
   if(ex != null) {
      System.out.println("Oops! We have an exception - " + ex.getMessage());
      return "Unknown!";
   }
   return res;
});
```

WhenComplete

Fue explicado en la pregunta 2.

Comparativa entre las 3 alternativas

Artículo	Handle()	whenComplete ()	exceptionally()

¿Acceso al éxito?	sí	sí	No
¿Acceso al fracaso?	sí	sí	sí
¿Puede recuperarse del fracaso?	sí	No	sí
¿Puede transformar el resultado de `T`a `U`?	sí	No	No
¿Desencadenar cuando ocurre éxito?	sí	sí	No
¿Desencadenar cuando falla?	sí	sí	sí
¿Tiene una versión asíncrona?	sí	sí	Sí (Java 12)

https://mincong.io/2020/05/30/exception-handling-in-completable-future/

5. **¿Qué es el manifiesto Reactivo?**

   El Manifiesto Reactivo es un documento que recoge una serie de intenciones e ideas a seguir con el objetivo de construir sistemas reactivos. estas ideas son:

   Responsivos: El sistema responde a tiempo en la medida de lo posible. La responsividad es la piedra angular de la usabilidad y la utilidad. Significa que los problemas pueden ser detectados rápidamente y tratados efectivamente. Los sistemas responsivos se enfocan en proveer tiempos de respuesta rápidos y consistentes, estableciendo límites superiores. Esto, simplifica el tratamiento de errores, aporta seguridad al usuario final y fomenta una mayor interacción.

   Resilientes: El sistema permanece responsivo frente a fallos. La resiliencia es alcanzada con replicación, contención, aislamiento y delegación. Los fallos son manejados dentro de cada componente, aislando cada componente de los demás, y asegurando que cualquier parte del sistema pueda fallar y recuperarse sin comprometer el sistema completo. La recuperación de cada componente se delega en otro componente (externo) y la alta disponibilidad se asegura con replicación.

   Elásticos: El sistema se mantiene responsivo bajo variaciones en la carga de trabajo. Los Sistemas Reactivos pueden reaccionar a cambios en la frecuencia de peticiones adaptando los recursos asignados. Esto implica diseños que no tengan cuellos de botella centralizados, resultando en la capacidad de dividir o replicar componentes y distribuir las peticiones entre ellos.

   Orientados a Mensajes: Los Sistemas Reactivos confían en el intercambio de mensajes asíncronos para establecer fronteras entre componentes, lo que asegura bajo acoplamiento, aislamiento y transparencia de ubicación. también proporcionan los medios para delegar fallos como mensajes. El intercambio de mensajes explícito posibilita la gestión de la carga, la elasticidad, y el control de flujo, gracias al modelado y monitorización de las colas de mensajes en el sistema. La mensajería basada en ubicaciones transparentes permite que la gestión de fallos pueda trabajar con los mismos bloques y semánticas a través de un cluster o dentro de un solo nodo. La comunicación No-bloqueante permite a los destinatarios consumir recursos sólo mientras estén activos, llevando a una menor sobrecarga del sistema.

https://www.reactivemanifesto.org/es

6. **Explique Reactive programming vs. Reactive systems**

Sistemas reactivos es un término que se usa para describir un estilo arquitectónico, el cual permite que las aplicaciones compuestas por múltiples microservicios trabajen juntas para reaccionar mejor a su entorno y entre sí. Están diseñados para ser elásticos, resistentes y receptivos mediante el uso de comunicación asincrónica basada en mensajes.

La motivación de este nuevo paradigma procede de la necesidad de responder a las limitaciones de escalado en los modelos de desarrollo actuales, que se caracterizan por su desaprovechamiento del uso de la CPU debido al I/O, el sobreuso de memoria (enormes thread pools) y la ineficiencia de las interacciones bloqueantes.Es un paradigma de programación que sigue los lineamientos planteados en el manifiesto reactivo y está enfocado en:

Manejo de flujos de datos asíncronos:
Basado en datos (finitos o infinitos) que fluyen continuamente, los sistemas reactivos reaccionan a los datos ejecutando una serie de eventos.
sigue el patrón Observer; cuando hay un cambio de estado en un objeto, los otros objetos son notificados y actualizados. Por lo tanto, en lugar de sondear eventos para los cambios, los eventos se realizan de forma asíncrona para que los observadores puedan procesarlos.

Uso eficiente de recursos:
utilizando E/S asíncrona, la idea es disminuir el uso ineficiente de recursos usando aquellos recursos que de lo contrario estarían inactivos, ya que permanecen a la espera de actividad de E/S. Los nuevos datos se notifican a los clientes en vez de tener que solicitarlos, debido a que la E/S asíncrona invierte el diseño normal del procesamiento. Este enfoque libera al cliente para hacer otras cosas mientras espera nuevas notificaciones.
Existe el riesgo de que demasiadas notificaciones desborden al cliente. Por ello, un aspecto fundamental del control de flujo en sistemas distribuidos es que el cliente debe ser capaz de rechazar el trabajo que no puede manejar.

https://blog.bi-geek.com/que-es-la-programacion-reactiva/

https://developer.ibm.com/es/series/learning-path-introduction-to-reactive-systems/

7. **Explique Event-driven vs. message-driven**

Mensajería: los mensajes transportan una carga útil y se conservan hasta que se consumen. Los consumidores de mensajes suelen estar directamente dirigidos y relacionados con el productor, a quien le importa que el mensaje se haya entregado y procesado.

Eventos: los eventos se conservan como un historial de transmisión reproducible. Los consumidores de eventos no están vinculados al productor. Un evento es un registro de algo que ha sucedido y, por lo tanto, no se puede cambiar.

Casos de uso de mensajería y transmisión de eventos

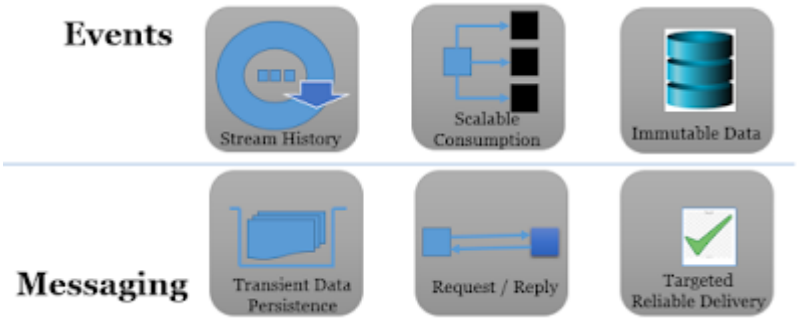
Dentro de una organización, hay muchos tipos diferentes de mensajes que fluyen, por ejemplo:

- Mensajes de pedido que representen una tarea para completar en una aplicación o sistema.
- Mensajes de respuesta que comunican la finalización de una solicitud
- Registro que nos informa del procesamiento de una solicitud
- Monitorización que nos informa del estado de un sistema

Se pueden proporcionar muchos ejemplos adicionales y se clasificarán en dos tipos. Mensajes que se centran en:

- Operaciones: representan el procesamiento actual o futuro. Por ejemplo, un minorista puede enviar una solicitud de pedido a su proveedor de pagos para su procesamiento.
- Eventos: representan el estado del sistema. Por ejemplo, un minorista puede emitir un evento por cada pedido completado. representan hechos y se emiten sin conocimiento de cómo se utilizarán.

Aunque podríamos enumerar cientos de casos de uso, nos centraremos en tres aspectos clave, para centrarnos en las diferencias.



Mensajes

Datos transitorios: los datos solo se almacenan hasta que un consumidor ha procesado el mensaje o caduca. No es necesario que los datos se conserven más de lo necesario.

Solicitar / Responder: las operaciones pueden dispararse y olvidarse, sin embargo, a menudo es deseable una interacción de solicitud / respuesta. Por lo tanto, cualquier solución debe respaldar este patrón de interacción.

Entrega confiable dirigida: cuando se envía un mensaje, se dirige a la entidad que procesará la solicitud o recibirá la respuesta. Esta orientación puede utilizar direccionamiento lógico y, en muchos casos, es preferible. Los mensajes también deben entregarse con un nivel adecuado de fiabilidad. Los diferentes mensajes tendrán diferentes niveles, pero el comportamiento transaccional y asegurado es fundamental.

Eventos

Historial de transmisiones: cuando se recuperan eventos, a menudo los consumidores están interesados en eventos históricos, no solo en los recientes, como recuperar las tendencias históricas de la disponibilidad de un sistema. Por lo tanto, cada evento debe adjuntarse y ponerse a disposición de los consumidores y, según la configuración, se almacenará un cierto número o volumen de eventos antes de su eliminación.

Consumo escalable: es probable que se emitan múltiples eventos para una sola operación a medida que pasa por la red. Por lo tanto, es probable que los requisitos de escala para manejar eventos sean órdenes de magnitud mayores que los requeridos para admitir mensajes operativos. La escalabilidad del requisito se limita a los productores que crean mensajes y al consumo. Un solo evento debe estar disponible para que lo consuman muchos consumidores con un impacto limitado a medida que aumenta el número de consumidores.

Datos inmutables: cuando se emite un evento, es una declaración de hecho y puede considerarse datos inmutables. Esto permite a los consumidores confiar en la suposición de una reproducción coherente y reduce el impacto de la replicación de los datos desde un punto de vista coherente.

Fuente: https://devaraj-durairaj.medium.com/message-driven-vs-event-driven-e07a18d40a95

8. **¿Qué es http://reactivex.io/?**

ReactiveX es una API que facilita el manejo de flujos de datos y eventos, a partir de una combinación de el patrón *Observer (Define relaciones de dependencia, uno a muchos, entre objetos, si el observado cambia, los dependientes reaccionan de forma automática)*, el patrón *Iterator (Forma de acceder secuencialmente a los elementos de una colección sin exponer su representación)*, y características de la *Programación Funcional (expresiones lambda) para recorrer conjuntos*.

El manejo de datos en tiempo real es una tarea común en el desarrollo de aplicaciones. Por lo tanto, tener una manera eficiente y limpia de lidiar con esta tarea es muy importante.

ReactiveX nos ofrece una API flexible para crear y actuar sobre los flujos de datos. Además, simplifica la programación asíncrona, como la creación de hilos y los problemas de concurrencia.

Es así que: *RxJava* es la implementación de *ReactiveX* para *Java*.

Tutorial de consulta:
https://programacionymas.com/blog/introduccion-rx-java-tutorial-android

9. **Brinde usos y ejemplos de RxJava.**

Las 2 clases principales son: Observable y Subscriber:

Observable es una clase que emite un flujo de datos o eventos. Mientras que, Subscriber es una clase que actúa sobre los elementos emitidos.

Un (objeto de la clase) Observable emite 1 o más elementos, y luego se completa (con éxito o con algún error).

Un Observable puede tener varios Subscribers, y cada elemento emitido por el Observable, será enviado al método Subscriber.onNext() para ser "usado en lo que se requiera".

Una vez que un Observable ha terminado de emitir elementos, invocará al método Subscriber.onCompleted(). O en caso que haya ocurrido algún error, el Observable invocará el método Subscriber.onError().

Ejemplo 1:

```
Observable<Long> observable1=Observable.interval(500, TimeUnit.MILLISECONDS).take(20);

Observable<Long> observable2=Observable.interval(200, TimeUnit.MILLISECONDS).take(20);

Observable.merge(observable1,observable2).subscribe(new Action1<Long>() {

 @Override

 public void call(Long arg0) {

        System.out.println(arg0);

 } });

 Thread.sleep(20000); }
```

construí dos observables. Cada uno de ellos emite 20 números a diferentes intervalos de tiempo (cada uno crea un Thread). Podríamos considerarlos como dos tareas asíncronas diferentes. uso, para fusionar ambas tareas, el método merge y tratarlas como si fuera una única.

Ejemplo 2:

```
Observable.just(1, 2, 3, 4, 5, 6) // números que recibe el Observable

    .filter(new Func1() { // filtramos para que solo emita valores impares

        @Override
```

```
            public Boolean call(Integer value) {
                return value % 2 == 1;
            }
        })
        .subscribe(new Subscriber() {
            @Override
            public void onNext(Integer value) {
                System.out.println("onNext: " + value);
            }

            @Override
            public void onCompleted() {
                System.out.println("Complete!");
            }

            @Override
            public void onError(Throwable e) {
            }
        });
// Salida:
// onNext: 1
// onNext: 3
// onNext: 5
// Complete!
```

El operador filter() permite definir una función que recibirá todos los valores que “pretende emitir” el Observable.

En esta función: Los valores que devuelvan false no son emitidos al Subscriber (por eso es que no se muestran en la salida).

https://www.arquitecturajava.com/introduccion-rxjava-observables/

https://www.programaenlinea.net/que-es-rxjava/