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

La interfaz 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. El resultado solo se puede recuperar utilizando el método get cuando el cálculo se ha completado, bloqueando si es necesario hasta que esté listo.También, se proporcionan métodos adicionales para determinar si la tarea se completó normalmente o se canceló.

Una vez que se ha completado un cálculo, el cálculo no se puede cancelar.

Ejemplo :

interface ArchiveSearcher { String search(String target); }

class App {

ExecutorService executor = ...

ArchiveSearcher searcher = ...

void showSearch(final String target)throws InterruptedException {

Future<String> future = executor.submit(new Callable<String>() {

public String call() {

return searcher.search(target);

}});

displayOtherThings(); // do other things while searching

try {

displayText(future.get()); // use future

} catch (ExecutionException ex) { cleanup(); return; }

}

}

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

La clase CompletableFuture, proporciona una extensión muy poderosa de Future, que puede ayudarnos a simplificar la complejidad de la programación asincrónica y proporcionar capacidades de programación funcional, que pueden pasarse a través de devoluciones de llamada. La forma de lidiar con los resultados del cálculo, y proporciona un método para convertir y combinar CompletableFuture.

La clase CompletableFuture implementa la interfaz Future, por lo que aún puede pasarla como antesgetObtenga resultados bloqueando o sondeando, pero no se recomienda este método.

CompletableFuture y FutureTask pertenecen a la clase de implementación de la interfaz Future, y ambos pueden obtener el resultado de ejecución del hilo.

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

**whenComplete :** Puede manejar resultados de cálculo normales y anormales, y excepcionalmente maneja situaciones anormales.

public static void main(String[] args) throws ExecutionException, InterruptedException {

CompletableFuture future = CompletableFuture.supplyAsync(new Supplier<Object>() {

@Override

public Object get() {

System.out.println(Thread.currentThread().getName() + "\t completableFuture");

int i = 10 / 0;

return 1024;

}

}).whenComplete(new BiConsumer<Object, Throwable>() {

@Override

public void accept(Object o, Throwable throwable) {

System.out.println("-------o=" + o.toString());

System.out.println("-------throwable=" + throwable);

}

}).exceptionally(new Function<Throwable, Object>() {

@Override

public Object apply(Throwable throwable) {

System.out.println("throwable=" + throwable);

return 6666;

}

});

System.out.println(future.get());}

**thenAccept y thenRun**: muy similares al whenComplete, ejecutaran el lambda una vez se complete el futuro. El primero recibe un resultado, y el segundo no. Son equivalentes al supplyAsync y runAsync respectivamente.

// thenAccept

CompletableFuture<String> futureAsync = CompletableFuture.supplyAsync(() -> {

LOGGER.info("Comenzando supplyAsync for thenAccept...");

Sleep.sleepSeconds(2);

LOGGER.info("Terminado supplyAsync for thenAccept!");

return "Terminado";

}, executor);

futureAsync.thenAcceptAsync(s -> {

LOGGER.info("Comenzando thenAccept...");

Sleep.sleepSeconds(2);

LOGGER.info("Terminado thenAccept!");

LOGGER.info("Resultado: {}", s);

}, executor);

// thenRun

CompletableFuture<Void> futureRun = CompletableFuture.runAsync(() -> {

LOGGER.info("Comenzando runAsync for thenRun...");

Sleep.sleepSeconds(2);

LOGGER.info("Terminado runAsync for thenRun!");

}, executor);

futureRun.thenRunAsync(() -> {

LOGGER.info("Comenzando thenRun...");

Sleep.sleepSeconds(2);

LOGGER.info("Terminado thenRun!");

}, executor);

**thenCompose**: Lo que hace es una cadena de futuro también. Por ejemplo, aquí llamamos a ‘thenCompose’ con una lambda que a su vez es otro futuro:

CompletableFuture<String> future1 = CompletableFuture.supplyAsync(() -> {

LOGGER.info("Comenzando supplyAsync for thenCompose...");

Sleep.sleepSeconds(2);

LOGGER.info("Terminado supplyAsync for thenCompose!");

return "Terminado";

}, executor);

CompletableFuture<String> fCompose =

future1.thenComposeAsync(s -> CompletableFuture.supplyAsync(() -> {

LOGGER.info("Comenzando thenCompose...");

Sleep.sleepSeconds(2);

LOGGER.info("Terminado thenCompose!");

return s.concat(" + Terminado other");

}, executor),

executor);

fCompose.whenCompleteAsync((s, e) -> LOGGER.info("Resultado thenCompose: {}", s),

executor);

**thenCombine**: En este caso, en lugar de una cadena de futuros, espera a que terminen dos futuros, para luego hacer algo. En este caso la lambda tendrá dos parámetros, que son el resultado de cada uno de los dos futuros:

CompletableFuture<String> future1 = CompletableFuture.supplyAsync(() -> {

LOGGER.info("Comenzando future1 for thenCombine...");

Sleep.sleepSeconds(2);

LOGGER.info("Terminado future1 for thenCombine!");

return "Terminado";

}, executor);

CompletableFuture<String> future2 = CompletableFuture.supplyAsync(() -> {

LOGGER.info("Comenzando future2 for thenCombine...");

Sleep.sleepSeconds(1);

LOGGER.info("Terminado future2 for thenCombine!");

return "Terminado other";

}, executor);

CompletableFuture<String> fCombine =

future1.thenCombineAsync(future2, (s1, s2) -> {

LOGGER.info("En el thenCombine, recibidos results: {}, {}", s1, s2);

return s1 + s2;

}, executor);

fCombine.whenCompleteAsync((s, e) -> LOGGER.info("Resultado thenCombine: {}", s),

executor);

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

Tenemos varias maneras de gestionar las excepciones de futuros con la clase CompletableFuture, usando estos métodos:

**exceptionally**: registra un callback para gestionar la excepción. Recibe una lambda que solo tiene de parámetro la excepción, debe retornar un valor del mismo tipo que el futuro en el que se originó la excepción.

**handle**: registra un callback para gestionar el resultado o excepción. Recibe una lambda que tiene dos parámetros, el resultado y la excepción. Si la excepción no es nula, es que ha habido una excepción. También deber retornar un valor del tipo del futuro que lanzo la excepción.

**whenComplete**: con este método que ya hemos explicado podemos hacer algo parecido al ‘handle’, dado que la lambda que registra tiene también los dos parámetros.

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

El Manifiesto Reactivo es un [documento](https://www.reactivemanifesto.org/) que define los principios básicos de la [programación reactiva](https://hub.packtpub.com/what-is-functional-reactive-programming/) .

**Los 4 principios del Manifiesto Reactivo:**

Los sistemas reactivos deben responder: el sistema debe responder de manera oportuna. Los sistemas receptivos se enfocan en brindar tiempos de respuesta rápidos y consistentes, por lo que brindan una calidad de servicio constante.

### Los sistemas reactivos deben ser resistentes: en caso de que el sistema se enfrente a alguna falla, debe seguir respondiendo. La resiliencia se logra mediante la replicación, la contención, el aislamiento y la delegación. Las fallas están contenidas dentro de cada componente, aislando los componentes entre sí, por lo que cuando se produce una falla en un componente, no afectará a los otros componentes ni al sistema en su conjunto.

### Los sistemas reactivos deben ser elásticos: Los sistemas reactivos pueden reaccionar a los cambios y seguir respondiendo a diferentes cargas de trabajo. Logran elasticidad de manera rentable en plataformas de hardware y software de productos básicos.

### Los sistemas reactivos deben estar impulsados ​​por mensajes: Impulsado por mensajes: para establecer el principio de resiliencia, los sistemas reactivos deben establecer un límite entre los componentes basándose en el paso de mensajes asíncronos.

Esos son los principios fundamentales detrás de la programación reactiva presentados por el manifiesto.

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

**Reactive programming:** es un subconjunto de la programación asincrónica y un paradigma donde la disponibilidad de nueva información impulsa la lógica hacia adelante en lugar de tener un flujo de control impulsado por un hilo de ejecución.

Admite la descomposición del problema en varios pasos discretos donde cada uno puede ejecutarse de forma asincrónica y sin bloqueo, y luego componerse para producir un flujo de trabajo, posiblemente sin límites en sus entradas o salidas.

Los principales beneficios de la programación reactiva son:

* Una mayor utilización de los recursos informáticos en hardware multinúcleo y multiprocesador;
* Un mayor rendimiento al reducir los puntos de serialización según la Ley de Amdahl y, por extensión, la Ley de escalabilidad universal de Günther.

Un beneficio secundario es el de la productividad del desarrollador, ya que todos los paradigmas de programación tradicionales han luchado por proporcionar un enfoque sencillo y fácil de mantener para lidiar con la computación y E / S asincrónicas y sin bloqueo. La programación reactiva resuelve la mayoría de los desafíos aquí, ya que generalmente elimina la necesidad de una coordinación explícita entre los componentes activos.

**Reactive systems:** son un conjunto de principios de diseño arquitectónico para construir sistemas modernos que están bien preparados para satisfacer las crecientes demandas a las que se enfrentan las aplicaciones en la actualidad.

La base de un sistema reactivo es el paso de mensajes , que crea un límite temporal entre los componentes que permite desacoplarlos en el tiempo, lo que permite la concurrencia, y el espacio, lo que permite la distribución y la movilidad. Este desacoplamiento es un requisito para el aislamiento total entre componentes y constituye la base tanto de la resiliencia como de la elasticidad .

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

Un mensaje son algunos datos enviados a una dirección específica. En los sistemas controlados por mensajes, cada componente tiene una dirección única a la que otros componentes pueden enviar mensajes. Cada uno de estos componentes, o destinatarios, espera mensajes y reacciona a ellos.

Los componentes de un sistema controlado por mensajes generalmente tienen una cola en la que los mensajes entrantes se pueden almacenar en caso de un pico de carga.

Un evento son algunos datos emitidos por un componente para que cualquiera que esté escuchando los consuma.

En los sistemas controlados por eventos, los componentes anuncian una ubicación donde exponen sus eventos. Esta conocida ubicación se implementa mediante una cola ordenada. A veces, la cola es indexable para que los consumidores puedan realizar un seguimiento de los eventos ya consumidos y los eventos pendientes.

**8. ¿Que es** [**http://reactivex.io/**](http://reactivex.io/)**?**

Es una API que facilita el manejo de flujos de datos y eventos, a partir de una combinación de **el patrón Observer**, el patrón **Iterator**, y características de la **Programación Funcional**.

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** (mediante el uso de **Observables** y operadores) 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.

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

**RxJava** es la implementación de ReactiveX para Java.Las 2 clases principales son: **Observable** y **Subscriber**.

En RxJava, **Observable** es una clase que **emite un flujo de datos o eventos**.

Y **Subscriber** es una clase que **actúa sobre** los elementos emitidos.

El funcionamiento estándar es el siguiente:

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

Específicamente hablando:

* 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**

Observable integerObservable = Observable.create(new Observable.OnSubscribe() {

@Override

public void call(Subscriber subscriber) {

subscriber.onNext(1);

subscriber.onNext(2);

subscriber.onNext(3);

subscriber.onCompleted();

}

});

Tenemos este Observable que emite los enteros 1, 2, y 3, y luego finaliza.

Ahora vamos a:

Crear un primer Subscriber, de forma que podamos **actuar ante estas emisiones**.

Subscriber integerSubscriber = new Subscriber() {

@Override

public void onCompleted() {

System.out.println("Complete!");

}

@Override

public void onError(Throwable e) {

}

@Override

public void onNext(Integer value) {

System.out.println("onNext: " + value);

}

};

Una vez que hemos creado un Observable y un Subscriber debemos conectarlos con el método Observable.subscribe().

integerObservable.subscribe(integerSubscriber);

// Y esta es la salida:

// onNext: 1

// onNext: 2

// onNext: 3

// Complete!