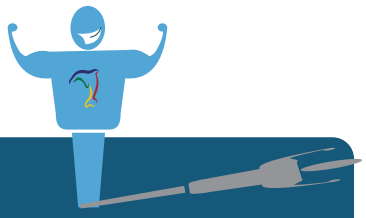
autentia

Avenida de Castilla,1 - Edificio Best Point - Oficina 21B
28830 San Fernando de Henares (Madrid)
tel./fax: +34 91 675 33 06
info@autentia.com - **www.autentia.com**

¿**Qué ofrece** Autentia Real Business Solutions S.L?

Somos su empresa de **Soporte a Desarrollo Informático**.
Ese apoyo que siempre quiso tener...

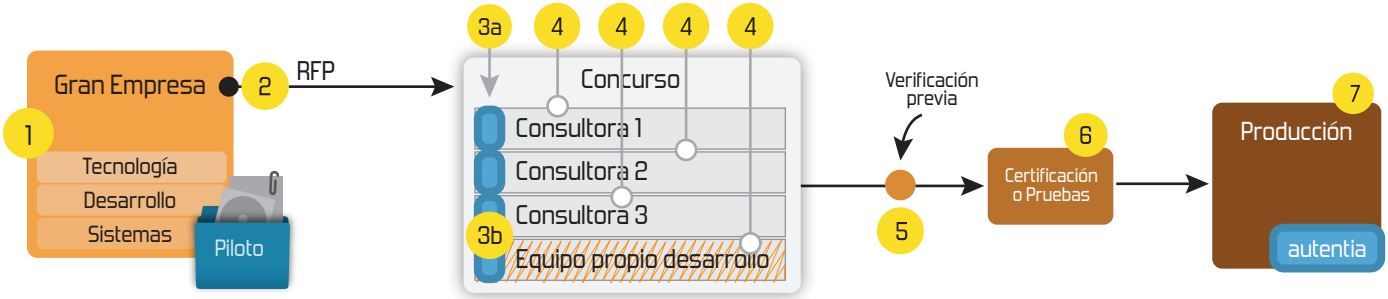
1. Desarrollo de componentes y proyectos a medida



2. Auditoría de código y recomendaciones de mejora

3. Arranque de proyectos basados en nuevas tecnologías

1. Definición de frameworks corporativos.
2. Transferencia de conocimiento de nuevas arquitecturas.
3. Soporte al arranque de proyectos.
4. Auditoría preventiva periódica de calidad.
5. Revisión previa a la certificación de proyectos.
6. Extensión de capacidad de equipos de calidad.
7. Identificación de problemas en producción.



4. Cursos de formación (impartidos por desarrolladores en activo)

- Spring MVC, JSF-PrimeFaces /RichFaces, HTML5, CSS3, JavaScript-jQuery
- Gestor portales (Liferay)
- Gestor de contenidos (Alfresco)
- Aplicaciones híbridas
- Control de autenticación y acceso (Spring Security)
- UDDI
- Web Services
- Rest Services
- Social SSO
- SSO (Cas)
- JPA-Hibernate, MyBatis
- Motor de búsqueda empresarial (Solr)
- ETL (Talend)
- Dirección de Proyectos Informáticos.
- Metodologías ágiles
- Patrones de diseño
- TDD
- Tareas programadas (Quartz)
- Gestor documental (Alfresco)
- Inversión de control (Spring)
- BPM (jBPM o Bonita)
- Generación de informes (JasperReport)
- ESB (Open ESB)

Compartimos nuestro conocimiento en:
www.adictosaltrabajo.com

Para más información visítenos en:
www.autentia.com

Daniel Diaz Suarez

Desarrollador Web en Autentia

Puedes encontrarme en Autentia: Ofrecemos servicios de soporte a desarrollo, factoría y formación

Somos expertos en Java/JEE

Ver todos los tutoriales del autor

Fecha de publicación del tutorial: 2015-03-02 Tutorial visitado 1 veces Descargar en PDF

Jugando con la clase Optional en Java 8

0. Índice de contenidos.

1. Introducción

Java 8 ha añadido muchas cosas interesantes al lenguaje, sin duda la más importante han sido las famosas lambdas. Las lambdas van más allá del mero "azúcar sintáctico" y suponen cambios profundos en como usaremos Java de aquí a unos años, además de abrirnos muchas posibilidades, una de ellas es el patrón Option plasmado en la clase de la JDK8 `Optional` .

2. Historia

Por todos es conocida la famosa frase de Tony Hoare llamando a `null` como el error del billon de dólares. Muchos de nosotros hemos sufrido alguna Null Pointer Exception, a veces en partes del código donde parece imposible que sucedan, posiblemente ese NPE se ha estado fraguando desde otras partes lejanas de la aplicación.

Para corregir estos errores, algunos lenguajes han decidido eliminar por completo los temidos `null` , pero para aquellos lenguajes que en su momento lo incluyeron, no es tan fácil. De ahí la existencia de alternativas cómo el patrón **Option**, el cual nos permite mostrar de manera explicita (mediante el sistema de tipos) la posibilidad de que un método pueda no devolver el valor deseado. Esto nos obliga a controlar la posible ausencia de valor de manera explicita, permitiéndonos elegir un valor alternativo en caso de dicha ausencia o simplemente realizar otra acción.

3. Optional vs Exceptions

En Java las `Exception` pueden usarse para avisar de un error inesperado en un método. Estas, pueden ser de dos tipos, checked y unchecked, las *checked exceptions* obligan a quien llame a la exception a *capturarla*, sin embargo las *unchecked* pueden ser lanzadas sin avisarlo previamente en la *signatura* del método. Esto puede provocar errores silenciosos, y no deberían ser usadas excepto para errores insalvables.

El problema de las *checked exceptions* es que son bastante pesadas de utilizar, además de que un método puede lanzar varias dependiendo del problema, y acaban convirtiéndose más en un problema que en una solución.

`Optional` sin embargo parece una solución más ligera, sobre todo usado junto a otros patrones importados de la programación funcional que vamos a ver a continuación.

4. Clase Optional en Java 8

En Java 8 este patrón se encapsula en la clase Optional, la cual incluye muchos de los métodos necesarios para trabajar con este patrón. Vamos a hacer un breve repaso de la clase antes de lanzarnos a como usarla. *En algunos casos he simplificado la signatura del método para centrarnos en lo importante*

```
public final class Optional<T>{}
```

Lo más importante es la *signatura* de la clase, en la cual podemos ver que es una clase genérica que nos permite que el objeto que contenga (o no) sea de cualquier clase.

```
public static<T> Optional<T> empty()
public static <T> Optional<T> ofNullable(T value)
public static <T> Optional<T> of(T value)
```

El siguiente punto que vamos a ver, es cómo crear la clase, Optional tiene un constructor privado, y nos proporciona tres métodos factoría estáticos para instanciar la clase. Siendo el método `.of` el que nos permite recubrir cualquier objeto en un optional.

```
String nombre = "Daniel";
Optional<String> oNombre = Optional.of(nombre);
```

Los otros dos métodos nos permiten recubrir un valor nulo o devolver un objeto `Optional` vacío en caso de que queramos avisar de la ausencia de valor. La opción de recubrir un nulo viene dada principalmente para permitirnos trabajar con APIs que hacen uso de nulos para avisar de estas ausencias de valor.

```
public boolean isPresent()
public T get()
```

El método `isPresent` método es el equivalente a `variable == null` y cómo el propio nombre indica nos dice si el objeto `Optional` contiene un valor o está vacío. Este método se usa principalmente si trabajamos de manera imperativa con `Optional` . Y el método `get` es el encargado de devolvernos el valor, devolviendo una excepción si no estuviera presente.

```
public Optional<T> filter(Function f)
public<U> Optional<U> map(Function f)
public<U> Optional<U> flatMap(Function f)
```

Aquí es donde viene lo bueno, estos tres métodos hacen que trabajar con *Optional* sea verdaderamente interesante, nos da la posibilidad de encadenar distintas operaciones que devuelvan `Optional` sin tener que estar comprobando si el valor está presente después de cada operación. Más adelante podremos verlas en acción.

```
public T orElse(T other)
public T orElseGet(Function f)
public <X extends Throwable> T orElseThrow(Function f)
```

Finalmente estos métodos nos permiten finalizar una serie de operaciones, teniendo tres maneras: - La primera `orElse` nos devuelve el valor o si no devolverá el valor que le demos. - `orElseGet` , nos devolverá el valor si está presente y si no, invocará la función que le pasemos por parámetro y devolverá el resultado de dicha función. - Y finalmente `orElseThrow` , nos devolverá el valor si está presente y si invocará la función que le pasemos, la cual tiene que devolver una excepción y lanzará dicha excepción. Esto nos ayudará a trabajar en conjunción con APIs que todavía usen excepciones.

5. Usando Optional de manera imperativa

Vamos a ver como podemos usar `Optional` de manera imperativa, esto desde mi punto de vista es un anti-patrón, aunque siempre será mejor que devolver un null silencioso.

Pongamos el caso siguiente, tenemos un método que obtiene un disco a partir de un nombre, puede darse el caso de que no se encuentre ningún disco con ese nombre, sin `Optional` tendríamos dos opciones: - Devolver `null` en caso de que no encontrásemos el disco. - Lanzar una excepción indicando que no se ha encontrado el disco.

Con `Optional` se nos abre la tercera opción:

```
public Optional<Album> getAlbum(String artistName)
```

A la hora de usar este método, de la manera imperativa haríamos lo siguiente.

```
Album album;
Optional<Album> albumOptional = getAlbum("Random Memory Access");
if(albumOptional.isPresent()){
    album = albumOptional.get();
}else{
    // Avisar al usuario de que no se ha encontrado el album
}
```

Esto ya es un avance respecto a los `null` , ya que estamos indicando explícitamente al usuario de la API de que es posible que no se encuentre el album y de que es necesario que actúe en caso de error.

El problema de esto viene cuando queremos ejecutar varias operaciones consecutivas que devuelvan null. Para ilustrar este caso imaginémonos que después de obtener el Album queremos obtener las canciones del album y finalmente obtener la duración total del album.

```
private static Optional<Double> getDurationOfAlbumWithName(String name) {
    Album album;
    Optional<Album> albumOptional = getAlbum(name);
    if (albumOptional.isPresent()) {
        album = albumOptional.get();
        Optional<List<Track>> tracksOptional = getAlbumTracks(album.getName());
        double duration = 0;
        if (tracksOptional.isPresent()) {
            List<Track> tracks = tracksOptional.get();
            for (Track track : tracks) {
                duration += track.getDuration();
            }
            return Optional.of(duration);
        } else {
            return Optional.empty();
        }
    } else {
        return Optional.empty();
    }
```

Como podemos observar esto se nos puede ir de las manos muy rápidamente, cada operación sucesiva que hagamos sobre un método que puede devolver un valor vacío se convierte en un nivel más de anidación.

Llegados a este punto podemos pensar en usar excepciones, las cuales al menos nos permiten tener todas las acciones a la misma altura dentro de un `try` y resolver los distintos errores en el `catch` .

Últimos Tutoriales del Autor

» Introducción a React

» Analiza el código de tu aplicación Android con SonarQube

» Introducción a Typescript

» Primeros pasos con Elasticsearch

» Haciendo un cliente de Twitter en Android.

6. Usando Optional de manera funcional.

El patrón Option es un patrón nacido en los lenguajes funcionales (sobre todo Scala), por lo que usarlo de una manera imperativa hace que no sea la más eficiente como hemos podido demostrar anteriormente. A continuación, vamos a resolver el mismo problema, pero esta vez usando distintas construcciones de programación funcional, que gracias a las lambdas ahora también son posibles en Java8.

```
Optional<Double> getDurationOfAlbumWithName(String name) {
    Optional<Double> duration = getAlbum(name)
            .flatMap((album) -> getAlbumTracks(album.getName()))
            .map((tracks) -> getTracksDuration(tracks));
    return duration;
}
```

¡Usando las funciones `map` y `flatMap` acortaríamos un código relativamente complicado a solo 3 lineas!, vamos a repasar paso por paso donde está el truco.

Para ello vamos a ver el método `map` de la clase `Optional` que es lo que hace:

```
public<U> Optional<U> map(Function<? super T, ? extends U> mapper) {
    Objects.requireNonNull(mapper);
    if (!isPresent())
        return empty();
    else {
        return Optional.ofNullable(mapper.apply(value));
    }
}
```

Dentro de esa *signatura* complicada se encuentra algo bastante sencillo, es un método que admite una función, la cual a su vez admite un `Optional`, esta función ha de devolver un valor del tipo que acepta . Lo que hace la función es más fácil de ver, comprueba si el Optional está vacío, si lo está devuelve un `Optional` vacío y si no, **aplica** la función que le hemos pasado por parámetro, pasándole el valor del `Optional`.

Es decir, si el `Optional` está vacío, **el método map no hace nada**, esto es primordial para poder concatenar operaciones sin necesidad de comprobar a cada momento si el `Optional` está vacío.

Antes de seguir con ejemplo, comentar el uso del método `flatMap`, y es que cuando queremos encadenar distintas operaciones que devuelvan `Optional`, es necesario usar `flatMap` ya que si no acabaríamos teniendo un `Optional<Optional<Double>>`.

Si lo extraemos paso a paso, podemos ver que no hay magia por ningún lado:

```
Optional<Album> albumOptional = getAlbum(name);
Optional<List<Track>> listOptional = albumOptional.flatMap((album) -> getAlbumTracks(albu
Optional<Double> durationOptional = listOptional.map((tracks) -> getTracksDuration(tracks
```

7. Acabar con la cadena de Optionals

Podríamos seguir devolviendo `Optional` por toda la aplicación, pero de primeras no parece una buena idea, por ello en algún momento tenemos que decidirnos a que hacer en caso de que el valor que queremos no estuviera presente, para ello vamos a hacer uso de las diferentes opciones que hemos comentado anteriormente.

```
private static double getDurationOfAlbumWithName(String name) {
    return getAlbum(name)
            .flatMap((album) -> getAlbumTracks(album.getName()))
            .map((tracks) -> getTracksDuration(tracks))
            .orElse(0.0);
}
```

De esta manera usamos un valor alternativo, esta puede ser una buena opción si lo único que queremos es pintar esta información en la UI, de manera que la ausencia de valor tiene sentido y se expresa con `0.0`.

El `orElseGet` es un poco más especifico, y suele usarse cuando queremos intentar primero una computación rápida y dejamos la computación lenta como *opción B*.

```
private static Album getAlbum(String albumName) {
    Album album = getAlbumFromCache(albumName)
            .orElseGet(() -> getAlbumFromDisk(albumName));
    return album;
}
```

Y finalmente `orElseThrow`, si consideramos que es un valor necesario podríamos usar una excepción para para que la capturase el método que ha realizado la llamada.

```
private static Album getAlbum() throws Exception {
    return getAlbumFromCache("RAM")
            .orElseThrow(() -> new Exception("Album Not Found"));
}
```

8. Más allá de Optional

`Optional` es un paso adelante para evitar usar `nulls`, pero no soluciona todos los problemas. Una de las principales cosas en las que se quedar corto, es que no nos ofrece la posibilidad de que decir que es lo que ha salido mal en caso de que no haya valor. Lo cual hace difícil su uso en métodos en los que pueden salir varias cosas mal.

Para ello, otros lenguajes cómo Scala existen alternativas cómo la clase Validation y Either.

8.1 Either

La clase `Either` tiene dos posibilidades al igual que la clase `Optional`, la diferencia es que la manera de modelar el error no es con la ausencia de valor, si no con una clase propia que encapsula el error producido, ambas posibilidades se encapsulan en las clases :

- `Left` : Esta es la posibilidad de error
- `Right` : Esta es la posibilidad de éxito

Lo bueno de `Either` es que nos permite incluir un error en el caso `Left`, a diferencia de `Optional` que solo nos dejaba avisar de la ausencia de valor.

8.2 Validation

Puede darse un caso en el que queramos realizar las operaciones subsecuentes aun habiendo ocurrido un error, cómo podría ser el caso de la validación de un formulario, para ello, en otros lenguajes se inventó el concepto de `Validation`. Al igual que `Optional` y `Either`, se modelan dos casos `Success` y `Failure`, a diferencia de `Either` es que el `Failure` puede incluir uno o más errores.

- En caso de `Success` se incluye el valor.
- En caso de `Failure` se incluye el error o errores.

Existen implementaciones de ambas clases en Java, aunque no están en la JDK, por lo que su seguramente no se extienda tanto como el de `Optional`

9. Conclusiones

Como hemos podido ver Java8 ha incluido varios patrones funcionales, lo que nos permite escribir código más conciso y más resistente a errores sin perder todo el valor que aporta la orientación a objetos. Evitar los `null` es solo el primer paso para mejorar nuestro código, y `Option` se convierte en una de las mejores maneras de lograrlo.