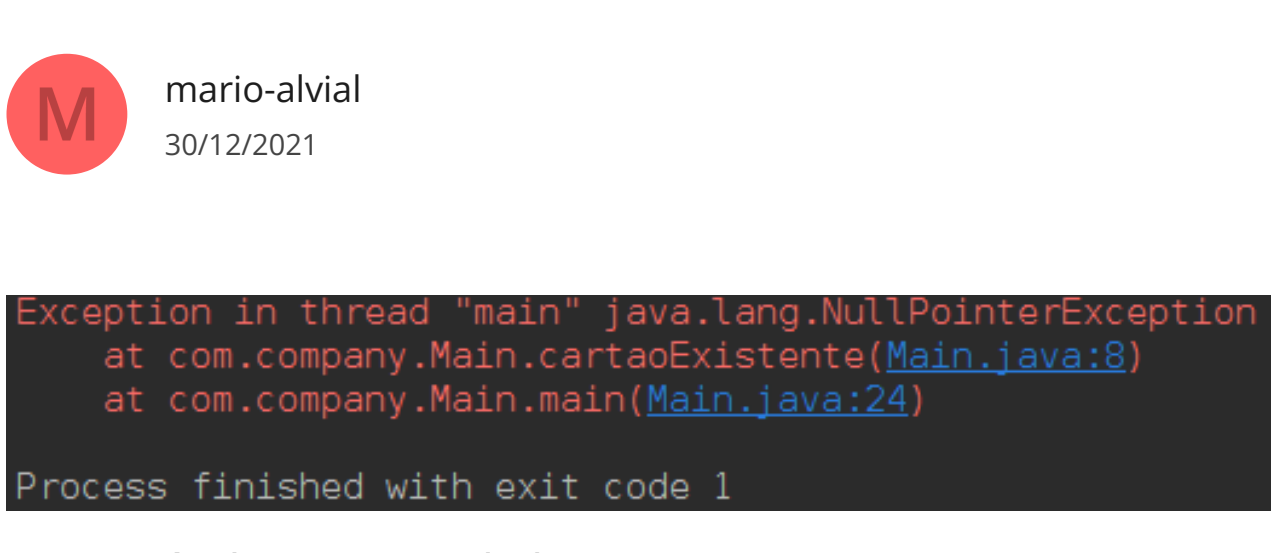
Evite NullPointerException en Java

mario-alvial
30/12/2021



```
Exception in thread "main" java.lang.NullPointerException
    at com.company.Main.cartaoExistente(Main.java:8)
    at com.company.Main.main(Main.java:24)

Process finished with exit code 1
```

Este artículo es parte de la **Formación Java y Orientación a Objetos** Quizás el error más común al que se enfrentan los desarrolladores, especialmente cuando están dando sus primeros pasos en el mundo de la programación, es el famoso NullPointerException (NPE para íntimos).Sin embargo, con algunas precauciones, a las que podemos llamar programación defensiva, logramos evitar esta excepción.

Para ayudarnos a comprender mejor cómo prevenir una NPE, tomemos un ejemplo que realiza pagos. En él, tenemos una clase que se encarga de guardar la tarjeta de crédito del usuario:

```
public class TarjetaDeCredito {
    private String titular;
    private String numero;
    private String cvv;
    private LocalDate fechaDeVencimiento;
    *//getters y setters*
}
```

También tenemos una clase que representa al usuario de este sistema:

```
public class Usuario {
    private String nomeCompleto;
    private String cpf;
    private List<CartaoDeCredito> cartoes;
    *//getters e setters*
}
```

¿Me creerían si les dijera que es probable que este código ya reciba un NPE? Antes de explicar por qué, entendamos mejor esta excepción.

NullPointerException

NullPointerException es una excepción que indica que la aplicación intentó usar una referencia a un objeto que tenía un valor nulo.Extiende la clase **RuntimeException**, que a su vez incluye excepciones que se lanzan en tiempo de ejecución. Además, NPE es un *unchecked exception*, por lo que no es necesario manejarlo y el compilador no informa un error en tiempo de compilación.Las *unchecked exceptions* representan fallas en el código creado por el programador (sí, es nuestra culpa). Por lo general, estos son errores de aplicación que podrían haberse evitado si el desarrollador hubiera tenido más cuidado al programar.

Pero, ¿por qué tomamos NPE? ¿Cuáles son las causas? ¿Qué hace que el código sea propenso a esta excepción?

Si tuviéramos que enumerar las acciones que causan **NullPointerException** tendríamos algo como esto:

Llamar a un método de una referencia de objeto que tiene un valor nulo

- Acceder o modificar un atributo de una referencia de objeto nulo
- En el caso de array o colecciones, use métodos de elementos nulos
- Registrar un *NullPointerException*
- Usar un unboxing en un objeto de valor nulo
- Intenta obtener el tamaño de una array nula

Volviendo al código que se muestra al principio, te demostraré que es muy probable que ese código reciba una NPE.

Bueno, tenemos un método que, dada una nueva tarjeta que el usuario está intentando registrar, verifica por número si esa tarjeta ya está registrada en la lista de tarjetas del usuario. Podemos ver el código del método a continuación:

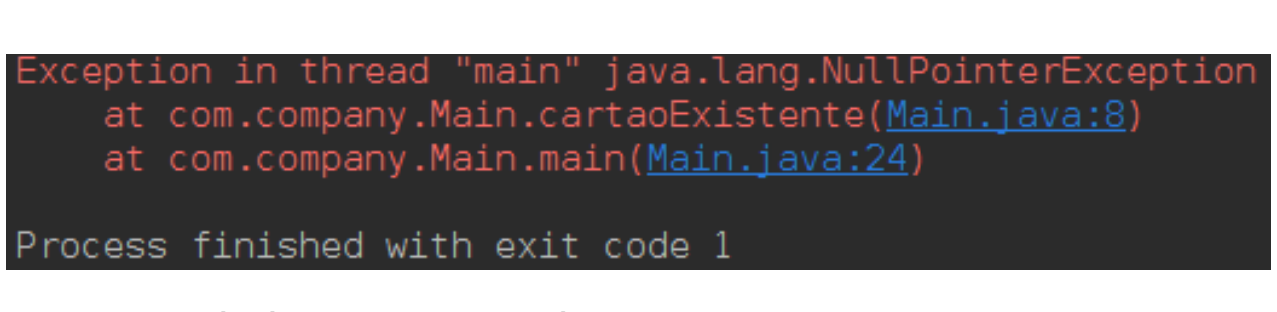
```
public static boolean tarjetaExistente(TarjetaDeCredito tarjetaDeCredito, Usua
    for (TarjetaDeCredito tarjeta : usuario.getTarjetas()) {
  }
    return true;
}
```

Para mostrarte cómo funciona este método, voy a crear un nuevo usuario e intentar validar una nueva tarjeta de crédito.

Como acabamos de crear el usuario, en teoría el resultado debería ser true porque aún no tenemos ninguna tarjeta registrada. Para las pruebas, usaremos el siguiente código:

```
public static void main(String[] args) {
    CartaoDeCredito cartaoDeCredito = new CartaoDeCredito("Mário E Alvial", "5
    Usuario usuario = new Usuario("Mário Sérgio Esteves Alvial", "75475962820"
    System.out.println(cartaoExistente(cartaoDeCredito, usuario));
}
```

Al ejecutar este código, tenemos el siguiente resultado:



```
Exception in thread "main" java.lang.NullPointerException
    at com.company.Main.cartaoExistente(Main.java:8)
    at com.company.Main.main(Main.java:24)

Process finished with exit code 1
```

Hay más de lo que esperaba. Hacemos una NPE, pero ¿por qué? Completé toda la información, tanto para la tarjeta como para el usuario, ¿qué es nulo?

¿Recuerda que dije que solo esa declaración de clase dejaba abierta la posibilidad de NPE? Entonces, ella fue la causa del problema. Todos los atributos de esas clases se declararon, pero no se inicializaron, por lo que, como no son de tipos primitivos, el valor predeterminado de estos atributos es nulo.

Entonces, cuando fuimos a iterar a través de la lista de tarjetas del usuario para ver si ya había una tarjeta con ese número, en realidad iteramos sobre una lista nula y fue entonces cuando tomamos la NPE.

Para arreglar este código podemos encapsular nuestro for dentro de uno if que comprueba que la lista de tarjetas no sea nula. Algo así:

```
public static boolean tarjetaExistente(TarjetaDeCredito tarjetaoDeCredito, Usu
    if (usuario.getTarjetas() != null) {
        for (TarjetaDeCredito tarjeta : usuario.getTarjetas()) {
            if (tarjetaDeCredito.getNumero().equals(tarjeta.getNumero()))
            {
                return false;
            }
        }
    }
    return true;
}
```

Problema de lista nula resuelto. Pero imagina tener que seguir haciendo este if cada vez que tienes la oportunidad, algo es nulo? Además del trabajo, el código se verá extremadamente desaliñado. Así que intentemos cortarlo de raíz de una vez por todas.

Codificando Defensivamente

Una gran práctica en el mundo de la programación que disminuye en gran medida las posibilidades de tomar NPE es inicializar los atributos de su clase. Entonces, su valor predeterminado nunca será nulo. En nuestro ejemplo, podemos hacerlo así:

```
public class TarjetaDeCredito {
    private String titular = "";
    private String numero = "";
    private String cvv = "";
    private LocalDate fechaDeVencimiento = LocalDate.now();
}

public class Usuario {
    private String nombreCompleto = "";
    private String nit = "";
    private List<TarjetaDeCredito> tarjetas = new ArrayList<>();
}
```

Listo, iniciados. Con eso, podemos eliminar ese if que verificaba si la lista de tarjetas del usuario era nula, porque sabemos que no lo es.

Otro punto importante es evitar tanto como sea posible, incluso diría no hacer, definitivamente no hacerlo, asignar nulo a alguna variable.

Entiendo que a veces quieres tener una referencia nula para asignarle valor luego, por ejemplo, ahora el proceso de creación de usuarios se ha vuelto un poco más complejo, necesitamos saber si el usuario es hombre o mujer, porque el proceso de creación ambos son diferentes, para eso tenemos este código:

```
public static Usuario creaUsuario(boolean isMulher){
    Usuario usuario = null;
    if(isMujer){
        usuario = creaUsuarioMujer();
    }else{
        usuario = creaUsuarioHombre();
    }
    return usuario;
}
```

Este código, por sí mismo, no lanza una NPE, pero nos deja en alerta porque cualquier método que llamemos usando la referencia usuario con valor nulo, activará una NPE. Una forma de evitar esto es haciendo un **early return** de esa forma:

```
public static Usuario creaUsuario(boolean isMujer){
    if(isMujer) {
        return creaUsuarioMujer();
    }
    return creaUsuarioHombre()
}
```

El código se vuelve aún más simple, ¿no crees? Mi punto es que realmente no asignes nulo a nada.

Valide siempre los datos "no confiables". Podemos asignar este nombre a datos provenientes de fuentes externas a su aplicación, ya sea que provengan del usuario o de una aplicación externa. Los llamo datos “no confiables” porque no sabes lo que contienen, no los enviaste, no hay forma de saberlo.

Lo mismo ocurre con las validaciones, ya sea usando anotaciones o validando para ver si los campos que recibimos no son nulos o están en el formato incorrecto, no importa. Lo importante es validar los datos desconocidos.

Y por último, pero no menos importante, haga todo lo posible por utilizar métodos que se refieran a objetos que esté seguro de que no son nulos.

Por ejemplo, volviendo a nuestro método que verifica si la tarjeta ya existe en la lista de tarjetas de usuario:

```
public static boolean tarjetaExistente(TarjetaDeCredito tarjetaDeCredito, Usua
    for (TarjetaDeCredito tarjeta : usuario.getTarjetas()) {
        if(tarjetaDeCredito.getNumero().equals(tarjeta.getNumero())){
            return false;
        }
    }
    return true;
}
```

Pensemos que acabamos de inicializar el atributo que representa la lista de tarjetas de usuario, el resto sigue siendo nulo. Además, ya tenemos tarjetas válidas guardadas en nuestra lista.

Ahora, está de acuerdo conmigo en que si el usuario no completa su número de tarjeta, tomaremos un NPE en esta parte de nuestro código de método:

```
if (tarjetaDeCredito.getNumero().equals(tarjeta.getNumero())) {
    return false;
}
```

Pues el objeto tarjetaDeCrédito está llamando al método getNumero()que devuelve el valor del atributo número que, en este caso, es nulo. Entonces, quién llama al método equals()es nulo y, a su vez, recibiremos un NPE.Para evitar esto, podemos validar informaciones provenientes de una fuente externa. De acuerdo, podemos, pero también podemos en lugar de llamar al equals()por el objeto completado por el usuario, llamarlo por la tarjeta iterada por el for. De esa forma:

```
if (tarjeta.getNumero().equals(tarjetaDeCredito.getNumero())) {
    return false;
}
```

De acuerdo, no tomaremos más NPE por la razón por la que lo estábamos tomando, porque sabemos que el atributo número de las tarjetas que ya están en la lista del usuario están completadas, por lo que el método equals()no se llamará por un objeto nulo.

La clase Optional

Con Java 8 vino una nueva clase llamada **Optional**, que a su vez trae un conjunto de métodos para manejar bloques de código críticos. Optional se puede pensar como algo que puede tener valor o no. Si no tiene valor, decimos que está vacío.Para explicar esta clase de manera consistente, tendría que escribir una publicación completa, ya que el tema no es pequeño. **Existen varias ventajas en el buen uso de Optional, entre ellas la protección contra NPE**, pero también, es más difícil entender su uso de inmediato, por lo que es necesario un estudio profundo de esta clase.

Puedes leer también:

- Verificar si es letra o número en Java
- Convertir int a String en Java

ARTÍCULOS DE TECNOLOGÍA > PROGRAMACIÓN

alura LATAM
POWERED BY grupo alura
AOVS Sistemas de Informática S.A
CNPJ 05.555.382/0001-33
SÍGUENOS EN NUESTRAS REDES SOCIALES

NAVEGACIÓN
PLANES
INSTRUCTORES
BLOG
POLÍTICA DE PRIVACIDAD
TÉRMINOS DE USO
SOBRE NOSOTROS
PREGUNTAS FRECUENTES

¡CONTÁCTANOS!
¡QUIERO ENTRAR EN CONTACTO!

BLOG
PROGRAMACIÓN
FRONT END
DATA SCIENCE
INNOVACIÓN Y GESTIÓN
DEVOPS

NOVEDADES Y LANZAMIENTOS
Email*
INSCRÍBETE

ALIADOS
Empresa participante do SCALE UP ENDEAVOR
En Alura somos unas de las Scale-Ups seleccionadas por Endeavor, programa de aceleración de las empresas que más crecen en el país.
Google for Startups Alumni 2021
Fuimos unas de las 7 startups seleccionadas por Google For Startups en participar del programa Growth Academy en 2021

CURSOS
Cursos de Programación — Lógica de Programación | Java
Cursos de Front End — HTML y CSS | JavaScript | React
Cursos de Data Science — Data Science | Machine Learning | Excel | Base de Datos | Data Visualization | Estadística
Cursos de DevOps — Docker | Linux
Cursos de Innovación y Gestión — Productividad y Calidad de Vida | Transformación Ágil | Marketing Analytics | Liderazgo y Gestión de Equipos | Startups y Emprendimiento

Alura
EDUCAÇÃO EM TECNOLOGIA: FIAP, Casa do Código, PM3 - Cursos de Produto
MAIS ALURA: START BY Alura, Alura Língua, Alura Para Empresas, Alura LATAM
COMUNIDADE: Tech Guide, 7 days of code, Hipsters ponto Jobs
PODCASTS: Hipsters ponto Tech, Dev sem Fronteiras, Layers ponto Tech