Optional

Класс Optional

Optional - новый класс в пакете java.util, является контейнером (оберткой) для значений которая также может безопасно содержать null. Впервые появился в Java 1.8. Используется для уменьшения объемов кода и увеличения надежности при работе с null.

Какой класс задач решается с помощью Optional

Класс Optional применяется в тех случаях когда из метода может быть возвращена ссылка значение которой равно null и это в последствии может привести к NullPointerException. Классическим примером подобного подхода является задача поиска. В случае успеха возвращается ссылка на найденный объект. Но возникает вопрос что вернуть в случае неудачи поиска. Обычно использовали два подхода:

- Возврат null с последующей проверкой в коде.
- Генерация исключения с последующим перехватом и обработкой.

Optional является классом оболочкой для возвращаемой из такого метода ссылки. И работа с ним значительно облегчает обработку подобных ситуаций

Модельный класс для дальнейшего пояснения

```java
public class Cat {

	private String name;
	private String color;

	public Cat(String name, String color) {
		super();
		this.name = name;
		this.color = color;
	}
	public Cat() {
		super();
	}

	public String getName() {
		return name;
	}

	public void setName(String name) {
		this.name = name;
	}

	public String getColor() {
		return color;
	}

	public void setColor(String color) {
		this.color = color;
	}

	@Override
	public String toString() {
		return "Cat [name=" + name + ", color=" + color + "]";
	}
}
```

Пример метода для поиска с возвратом null

```
public static Cat findCatByName(String name, Cat[] cats) {
    for (Cat cat : cats) {
        if (cat.getName().equals(name)) {
            return cat;
        }
    }
    return null;
}
```

В примере показан метод который производит поиск объекта класса Cat в массиве подобных объектов. В случае неудачи метод возвращает null. В таком случае нужны будут дополнительные проверки значения возвращенной ссылки.

Пример метода поиска с генерацией исключения

```
public static Cat findByNameEx(String name, Cat[] cats) throws NoSuchElementException {
    for (Cat cat : cats) {
        if (cat.getName().equals(name)) {
            return cat;
        }
    }
    throw new NoSuchElementException();
}
```

В примере показан метод который производит поиск объекта класса Cat в массиве подобных объектов. В случае неудачи метод генерирует исключение NoSuchElementException. В вызывающем коде необходимо обрабатывать подобное исключение.

Использование Optional

```
public static Optional<Cat> findCatByNameOptional(String name, Cat[] cats) {
    Cat result = null;
    for (Cat cat : cats) {
        if (cat.getName().equals(name)) {
            result = cat;
            break;
        }
    }
    return Optional.ofNullable(result);
}
```

В примере показан метод который производит поиск объекта класса Cat в массиве подобных объектов. В качестве возвращаемого значения используется Optional<Cat>. Именно в него «упаковывается» результат который и был возвращен.

Статические методы класса Optional и их использование

У Optional есть ряд статических методов каждый из них используется для упаковки результата в следующих случаях. Все они возвращают Optional.

Метод	Когда использовать
Optional.empty()	Объект точно отсутствует
Optional.of(T obj)	Объект присутствует и он точно не null
Optional.ofNullable(T obj)	Объект присутствует, но возможно он null

Используемый вариант по большей мере определяется логикой метода. Указанное выше не более чем рекомендация.

Какие преимущества дает применение Optional

Возврат значения типа Optional упрощает решения проблем указанных выше. Так, как теперь в качестве типа возвращаемого значения указан Optional то это сразу указывает на то, что результат поиска может быть и отрицательным. Теперь разработчику использующему этот метод нужно будет в явном виде проверять наличие или отсутствие результатов поиска. Таким образом код становиться яснее и безопаснее в использовании.

Проверка наличия или отсутствия не null ссылки в Optional

В простейшем случае для проверки наличия не null ссылки в Optional используется метод isPresent() и если данные метод вернул true то хранимая ссылка не null и для ее получения стоит вызвать метод get().

```
Optional<Cat> result = findCatByNameOptional("Vaska", cats);

if (result.isPresent()) {

    Cat cat = result.get();
    System.out.println(cat);
}else {
    System.out.println("Cat not found");
}
```

Проверка на наличие не null ссылки

Получение ссылки из Optional

Если не выполнить проверку и вызвать метод get() и при этом ссылка в Optional равна null будет создано исключение NoSuchElementException.

Возврат значения по умолчанию

Как вы уже убедились получение значения ссылки Optional довольно простое. Для этого нужно использовать метод get(). В ряде случаев возникает потребность извлечь ссылку на объект по умолчанию, если ссылка в Optional равна null. Вы можете вернуть ссылку или уже на существующий объект, или сначала создать объект а потом его вернуть.

Метод	Когда использовать
T orElse (T other)	Если значение ссылки равно null и нужно вернуть ссылку на существующий объект по умолчанию
T orElseGet(Supplier<? extends T> supplier)	Если значение ссылки равно null и нужно сначала создать объект по умолчанию и потом вернуть ссылку на него

Пример использования orElse

Ссылка на объект по умолчанию

```
Cat defaultCat = new Cat("Default name", "Default color");

Optional<Cat> result = findCatByNameOptional("Vaska", cats);

Cat cat = result.orElse(defaultCat);

System.out.println(cat);
```

Получение ссылки из Optional

Теперь в случае неудачного поиска (т. е. если ссылка хранимая в Optional равна null) метод orElse вернет ссылку которая использовалась в качестве его параметра. Т.е. ссылка cat в случае успешного поиска будет указывать на найденный объект, а в случае неудачи на объект по умолчанию (на него указывает ссылка defaultCat).

Пример использования orElseGet

```
Optional<Cat> result = findCatByNameOptional("Vaska", cats);

Cat getCat = result.orElseGet(Cat::new);
```

Создание объекта и возврат ссылки на него

В случае неудачного поиска (т. е. если ссылка хранимая в Optional равна null) метод orElseGet запустит процесс генерации объекта (в примере использована ссылка на конструктор по умолчанию класс Cat) и вернет ссылку на него. Т.е. ссылка cat в случае успешного поиска будет указывать на найденный объект, а в случае неудачи на объект по умолчанию (был создан вызовом конструктора по умолчанию).

Генерация исключения в случае «пустого» Optional

При работе с кодом написанным ранее (т.е. до введения Optional) может оказаться, что основные методы требуют результаты работы метода в виде ссылки. Если она равна null то должна происходить генерация исключения. По сути если унаследованный код опирается на прием когда в случае неудачного поиска или отсутствия объекта нужно генерировать исключение то это можно выполнить с помощью следующих методов:

Метод	Версия Java	Когда использовать
orElseThrow(Supplier<? extends X> exceptionSupplier)	1.8	Если храниться null то генерировать исключение полученное с помощью параметра метода.
orElseThrow()	1.11	Если храниться null то генерировать исключение NoSuchElementException

Пример использования orElseThrow(Supplier<? extends X> exceptionSupplier)

Этот метод позволяет сгенерировать вам любое исключение в случае если в Optional храниться null. Ссылку на конструктор этого исключения нужно указать в качестве параметра этого метода. Правда стоит отметить, что чаще всего в таких случаях генерируют NoSuchElementException.

```
Optional<Cat> result = findCatByNameOptional("Vaska", cats);

Cat cat = result.orElseThrow(NoSuchElementException::new);

System.out.println(cat)
```

Ссылка на конструктор нужного исключения

Пример использования orElseThrow()

По причине того что NoSuchElementException является наиболее часто генерируемым исключением в Java 11 появился метод упрощающий этот процесс. Теперь если вызвать orElseThrow() без параметров, то будет сгенерированно именно это исключение.

```
Optional<Cat> result = findCatByNameOptional("Vaska", cats);

Cat resultCat = result.orElseThrow();

System.out.println(resultCat);
```

Получение ссылки или генерация исключения

При таком применении если ссылка в Optional не равна null то она будет извлечена и присвоена ссылке resultCat. В противном случае будет возбужденно исключение NoSuchElementException.

Использование Optional для преобразования значений

Когда на основе искомых объектов нужно создать новые данные можно использовать методы трансформации Optional. Но перед их изучением сначала стоит рассмотреть что значит генерация новых данных на основании получаемого объекта.

```
Optional<Cat> result = findCatByNameOptional("Vaska", cats);

Cat cat = result.get();

String catName = cat.getName();
```

Получение новых данных на основании полученного объекта

На основании искомого объекта класса Cat созданы новые данные в виде строки.

Использование Optional для преобразования значений

Для упрощения реализаций задачи подобной описанной выше стоит использовать следующие методы Optional:

Метод	Когда использовать
Optional<T> filter(Predicate<? super T> predicate)	Если реализация Predicate вернет true, то вернеться Optinal со значением, в противном случае Optional со значением null.
Optional<U> map(Function<? super T,? extends U> mapper)	Если значение ссылки в Optional не равно null то создается новый Optional на основании значения ссылки. В противном случае вернется пустой Optional.
Optional<U> flatMap(Function<? super T,? extends Optional<? extends U>> mapper)	Если значение ссылки в Optional не равно null то создается новый Optional на основание текущего Optional. В противном случае вернется пустой Optional.
Optional<T> or(Supplier<? extends Optional<? extends T>> supplier)	Если значение ссылки в Optional равно null то вернется новый Optional созданный на основе параметра.

Пример использования метода map

```
Optional<Cat> result = findCatByNameOptional("Vaska", cats);

String catColor = result.map(a -> a.getName()).get();
```

Создание нового Optional

В этом примере мы получаем Optional<Cat> в качестве результата поиска. В дальнейшем при вызове метода map происходит создание Optional<String>. В качестве параметра методу map передана лямбда функция которая используется в качестве реализации функционального интерфейса Function и создает строку на основании объекта класса Cat. Полученная строка пакуется в Optional. Т.е. метод map на основе одного Optional генерирует другой Optional.

Пример использования метода or

```
String[] names = new String[] { null, null, "Vaska", null, "Barsik" };

Optional<String> result = Optional.ofNullable(names[0]);

for (String n : names) {
    Optional<String> temp = Optional.ofNullable(n);

    result = result.or(() -> temp);
}

System.out.println(result.get());
```

Замена произойдет если в result null, а в temp нет.

Тут реализована идея поиска первого элемента массива не равного null и упаковку его в новый Optional. Метод or работает следующим образом если в result храниться null, а temp при этом не null, то result будет равен текущему temp. Остальные значения при этом пропустяться.

Пример использования filter

```
String[] names = new String[] { null, null, "Vaska", null, "Barsik" };

Optional<String> result = Optional.ofNullable(names[0]);

for (String n : names) {

    Optional<String> temp = Optional.ofNullable(n).filter(a -> a.startsWith("B"));

    result = result.or(() -> temp);
}

System.out.println(result.get());
```

Проверка на то, что элемент начинается с «B»

Метод filter вызывается в случае если содержимое Optional не равно null. Если реализация Predicate вернет true то будет возвращен Optional с этим элементом. В примере в качестве реализации Predicate использована лямбда функция. Так как элементом Optional является String то эта реализация вернет true только в случае если эта строка начинается на «B».

Пример использования flatMap

```
String[] names = new String[] { null, null, "Vaska", null, "Barsik" };

Optional<String> result = Optional.ofNullable(names[0]);

for (String n : names) {
     Optional<String> temp = Optional.ofNullable(n).flatMap(a ->  Optional.of(a.toUpperCase()));
     result = result.or(() -> temp);
}
System.out.println(result.get());
```

Генерация на основе содержимого Optional нового Optional

В этом примере мы получаем Optional<String> в качестве результата поиска первого отличного от null значения в массиве. После чего метод flatMap создает новый Optional куда пакует значение строки в верхнем регистре.

В чем разница между map и flatMap

Довольно проблематично отличить разницу между методами map и flatMap ведь оба возвращают Optional. Разница заключается в том:

- map — на основе значения генерирует новое значение. Стоит использовать когда нужно сослаться на метод который принимает значение и возвращает значение.
- flatMap — на основе значения генерируется новый Optional со значением. Стоит использовать когда нужно сослаться на метод который принимает значение и возвращает Optional со значением.

По сути метод map для работы с кодом в котором не используется Optional, flatMap для работы с кодом в котором методы его активно используют.

В чем разница между map и flatMap

```
public class Main {

     public static void main(String[] args) {
          Cat cat1 = new Cat("Vaska", "Black");
          Optional<Cat> result = Optional.of(cat1);

          Optional<Cat> repCat1 = result.map(Main::repaintCat);

          Optional<Cat> repCat2 = result.flatMap(Main::repaintCatOptinal);
     }

     public static Cat repaintCat(Cat cat) {
          Cat repaintCat = new Cat(cat.getName(), "white");
          return repaintCat;
     }

     public static Optional<Cat> repaintCatOptinal(Cat cat) {
          Cat repaintCat = new Cat(cat.getName(), "white");
          Optional<Cat> repaintCatInOptional = Optional.of(repaintCat);
          return repaintCatInOptional;
     }

}
```

Используем первый метод для map

Используем второй метод для flatMap

Метод принимает Cat и возвращает Cat

Метод принимает Cat и возвращает Optional<Cat>

Задача выполнения каких либо действий при отличной от null ссылки в Optional

Довольно частой является ситуация когда в случае успешного поиска или получения объекта нужно произвести то или иное действие. Если же объект не был получен то действий производить не нужно было. Ниже приведен пример как это реализовывали в случае если в результате поиска могла быть возвращена ссылка на объект или null.

```
Cat cat = findCatByName("Vaska", cats);

if (cat != null) {
    System.out.println(cat);
}
```

Если возвращенная ссылка не равна null. Выполнить действие.

Задача выполнения каких либо действий при отличной от null ссылки в Optional

Класс Optional предлагает ряд методов которые упрощают реализацию данной задачи.

Метод	Когда использовать
void ifPresent(Consumer<? super T> action)	Если значение ссылки равно не равно null то выполнить действие с использованием ссылки, в противном случае ничего не делать.
void ifPresentOrElse(Consumer<? super T> action, Runnable emptyAction)	Если значение ссылки не равно null то выполнить действие с использованием ссылки, в противном случае выполнить альтернативное действие.

Пример применения ifPresent

```
Optional<Cat> result = findCatByNameOptional("Vaska", cats);

result.ifPresent(a -> System.out.println(a));
```

Если в Optional не null. Выполнить действие.

В данном примере Optional<Cat> является результатом метода поиска. В случае если в Optional не null то будет выполнен метод подставленный в качестве параметра (реализация Consumer). В данном случае в качестве реализации используется лямбда функция.

Пример применения ifPresentOrElse

```
Optional<Cat> result = findCatByNameOptional("Vaska", cats);

result.ifPresentOrElse(System.out::println, () -> System.out.println("Not found"));
```

Выполнить если ссылка не null

Выполнить если ссылка равна null

В данном примере Optional<Cat> является результатом метода поиска. В случае если в Optional не null то будет выполнен метод подставленный в качестве первого параметра (реализация Consumer). В данном случае в качестве реализации используется ссылку на метод System.out.println. Если же ссылка равна null то будет вызван метод подставленный в качестве второго параметра (реализация Runnable). В примере лямбда функция приводящая в выводу на экран надписи «Not found».

Метод stream()

В классе Optional присутствует метод **Stream<T> stream()**. Если ссылка не равна null то будет возвращен Stream с этой ссылкой, в противном случае пустой Stream. Рассмотрение потоков (Stream) и методов работы с ними будет рассмотрено позже.

Где стоит использовать Optional и где нет

Использовать Optional **стоит** в:

- Методы поиска
- Методы получения объекта
- Если множественные параметры метода могут иметь значение null, для их передачи стоит использовать Optional

Использовать Optional **не стоит** в:

- Качестве полей класса (не сериализуется)
- В качестве параметра метода установки

Список литературы

1) Герберт Шилдт Java 8. Полное руководство 9-е издание ISBN 978-5-8459-1918-2.
2) https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/util/Optional.html