

Wyjątki - Instrukcja throw

Czego się dowiesz

- Do czego służy instrukcja throw,
- czym różni się throw od try catch,
- kiedy używać instrukcji throw a kiedy bloku try catch.

Wstęp

Istnieją dwa rodzaje sytuacji, w których raczej będziemy unikali stosowania konstrukcji try-catch:

- nie będzie nam się chciało (uwierz, że dosyć często będziesz się z tym spotykać),
- nie będziemy widzieli sensu w obsłudze wyjątku w konkretnym miejscu, uznając, że na tym konkretnym etapie tworzenia aplikacji nie ma to większego sensu i lepiej zrobić to później.

Niektórzy zamiast obsłużyć wyjątek, dodają jedynie blok try-catch, ale w bloku catch nie robią nic. Jest to najgorsza sytuacja jaka może wystąpić i już lepiej, żeby bloku catch nie było wcale, a aplikacja się po prostu "wysypała". Program, w którym udajemy obsługę wyjątku może prowadzić do kolejnych problemów i niespójności przetwarzanych danych.

Jeżeli w jakiejś metodzie wykorzystujemy fragmenty kodu mogące powodować wygenerowanie wyjątku, to możemy o tym fakcie poinformować korzystając ze słowa **throws** w sygnaturze metody. W ciele metody możemy natomiast rzucić wyjątek korzystając z instrukcji **throw**, której działanie jest podobne do return, ale będzie oznaczało nieoczekiwane przerwanie metody, a nie zwrócenie wyniku. Jeżeli będziemy rzucali wyjątek kontrolowany, to dodanie słowa throws do sygnatury metody jest wymagane, a w przypadku wyjątków niekontrolowanych jest to opcjonalne i najczęściej się tego nie zapisuje.

Jako przykład stworzymy klasę reprezentującą samochód spalinowy. Każdy taki samochód posiada bak paliwa, który można zatankować. Zdefiniujemy metodę, która będzie symulowała przejechanie 100km. Zastanowimy się co zrobić, gdy zabraknie nam paliwa.

Car.java

```
class Car {
    private static final double FUEL_CONSUMPTION = 8; //8l/100km
    private double fuel;

    public void refuel(double additionalFuel) {
        fuel += additionalFuel;
    }

    public void drive() {
        fuel -= FUEL_CONSUMPTION;
    }

    @Override
    public String toString() {
        return "Stan paliwa: " + fuel;
    }
}
```

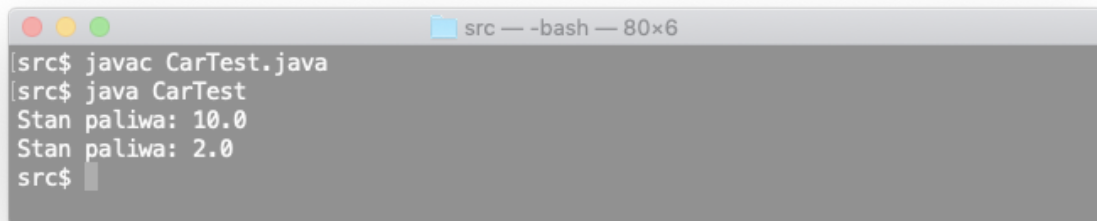
Klasa Car posiada pole fuel, czyli aktualny stan paliwa. Metoda refuel() pozwala zatankować samochód o wskazaną wartość paliwa, natomiast metoda drive() symuluje przejechanie 100km. Stała FUEL_CONSUMPTION oznacza ile paliwa samochód spala na 100km, my zakładamy, że 8 litrów.

W klasie testowej utworzymy obiekt Car, zatankujemy go, wyświetlmy jakieś informacje i spróbujemy przejechać kawałek.

CarTest.java

```
class CarTest {
    public static void main(String[] args) {
        Car car = new Car();
        car.refuel(10);
        System.out.println(car);
        car.drive();
        System.out.println(car);
    }
}
```

Po utworzeniu obiektu car tankujemy 10 litrów paliwa, następnie symulujemy przejechanie 100km. Po tankowaniu i przejechaniu dystansu wyświetlamy stan paliwa.



```
[src$ javac CarTest.java
[src$ java CarTest
Stan paliwa: 10.0
Stan paliwa: 2.0
src$ ]
```

W takim prostym przypadku wszystko działa ok, ale pomyślmy teraz o możliwych scenariuszach, w których coś może pójść nie tak.

1. Nie mamy wystarczająco dużo paliwa, żeby przejechać 100km. Stan baku robi się ujemny.
2. Samochód można tankować w nieskończoność, powinno być jakieś górne ograniczenie, które nie pozwoli zatankować np. więcej niż 50 litrów.

Ponieważ nasza klasa nie daje bezpośredniego dostępu do pola fuel, to na pierwszy rzut oka wystarczy dodać zabezpieczenie w postaci dwóch ifów w odpowiednich metodach klasy Car.

Car.java

```
class Car {
    private static final double FUEL_CONSUMPTION = 8; // 8l/100km
    private static final double MAX_FUEL = 50;
    private double fuel;

    public void refuel(double additionalFuel) {
        if (fuel + additionalFuel > MAX_FUEL)
            System.out.println("Nie możesz zatankować tyle paliwa. Zmieści się
jeszcze maksymalnie " + (MAX_FUEL - fuel));
        else
```

```
        fuel += additionalFuel;
    }

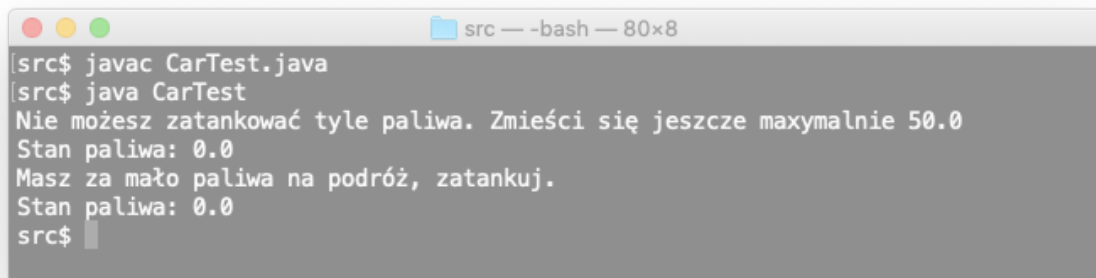
    public void drive() {
        if(fuel - FUEL_CONSUMPTION < 0)
            System.out.println("Masz za mało paliwa na podróż, zatankuj.");
        else
            fuel -= FUEL_CONSUMPTION;
    }

    @Override
    public String toString() {
        return "Stan paliwa: " + fuel;
    }
}
```

Jeśli teraz spróbujemy np. zatankować 60 litrów, to zobaczymy odpowiedni komunikat o niepowodzeniu.

CarTest.java

```
class CarTest {
    public static void main(String[] args) {
        Car car = new Car();
        car.refuel(60);
        System.out.println(car);
        car.drive();
        System.out.println(car);
    }
}
```



```
src — -bash — 80x8
[src$ javac CarTest.java
[src$ java CarTest
Nie możesz zatankować tyle paliwa. Zmieści się jeszcze maksymalnie 50.0
Stan paliwa: 0.0
Masz za mało paliwa na podróż, zatankuj.
Stan paliwa: 0.0
src$
```

Rozwiązanie takie działa, ale z punktu widzenia użyteczności kodu wprowadza to dużo ograniczeń. Po pierwsze w projekcie powinniśmy się starać oddzielać od siebie klasy, które przechowują dane, tzw. klasy modelu danych, od klas, które realizują obliczenia, czy też powalają się komunikować z użytkownikiem. U nas klasa Car jest modelem danych, reprezentuje samochód i pozwala nim w

pewien sposób sterować. Nie jest jej rolą to, żeby wyświetlać coś w konsoli, albo wczytywać dane od użytkownika. Nie powinny się w niej pojawiać wywołania metody `System.out.println()`.

Druga kwestia wymaga trochę wyobraźni. Na razie nasze aplikacje działają tylko w konsoli, ale wyobraź sobie, że teraz chcielibyśmy, żeby samochodem dało się sterować przy pomocy przycisków w jakimś graficznym interfejsie użytkownika. Nasza klasa w aktualnej formie będzie działała jedynie w aplikacjach konsolowych, jest mało uniwersalna i musielibyśmy ją tak naprawdę przepisać od nowa, żeby dostosować jej działanie do nowej aplikacji.

Obie sytuacje, które próbujemy tutaj rozwiązać, czyli próba zatankowania zbyt dużej ilości paliwa, albo próba podróżowania, gdy nie mamy wystarczająco paliwa to sytuacje, które są niepoprawne i są swego rodzaju błędami. Tak samo jak np. próba odwołania się do dziesiątego elementu w tablicy, która ma ich tylko pięć. Oba problemy da się rozwiązać przy pomocy wyjątków. Musimy więc w naszym programie stworzyć wyjątek, który będzie można następnie obsłużyć. Jeśli ktoś będzie próbował jechać samochodem, który nie ma paliwa, to program zostanie przerwany.

Wyjątki to w rzeczywistości obiekty, które rzucamy przy pomocy instrukcji `throw`. Działa to podobnie do zwracania obiektu poprzez instrukcję `return` jako wynik metody. Różnica polega na tym, że obiekt wyjątku możemy później złapać w bloku `try-catch`, a wartość zwróconą przez `return` przypisać np. do zmiennej.

```
Car.java
class Car {
    private static final double FUEL_CONSUMPTION = 8; // 8l/100km
```

```
private static final double MAX_FUEL = 50;
private double fuel;

void refuel(double additionalFuel) {
    if (fuel + additionalFuel > MAX_FUEL)
        throw new IllegalArgumentException("Nie możesz zatankować tyle
paliwa. Zmieści się jeszcze maksymalnie " + (MAX_FUEL - fuel));
    else
        fuel += additionalFuel;
}

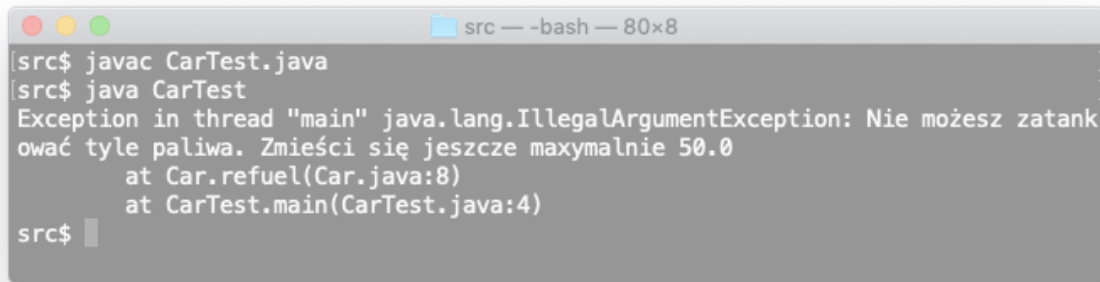
void drive() {
    if(fuel - FUEL_CONSUMPTION < 0)
        throw new IllegalStateException("Masz za mało paliwa na podróż,
zatankuj.");
    else
        fuel -= FUEL_CONSUMPTION;
}

@Override
public String toString() {
    return "Stan paliwa: " + fuel;
}
}
```

Zamiast wyświetlać komunikaty o błędach w metodzie `refuel()` tworzymy obiekt wyjątku `IllegalArgumentException` i rzucamy go przy pomocy instrukcji `throw`. W metodzie `drive()` robimy podobnie, ale rzucamy wyjątek typu `IllegalStateException`.

W Javie istnieje wiele klas wyjątków, my wykorzystaliśmy tylko dwie z nich. W praktyce nie ma znaczenia, czy rzucamy wyjątek `IllegalArgumentException`, czy `IllegalStateException`. Oba wpływają w taki sam sposób na działanie naszego programu, a to którego użyjemy ma służyć przede wszystkim czytelności kodu. W kolejnej lekcji nauczymy się też definiować własne wyjątki.

Jeśli spróbujesz teraz uruchomić aplikację, to w konsoli zobaczysz komunikat informujący o wyjątku `IllegalArgumentException` i program przestanie działać.

A screenshot of a terminal window with a title bar that says "src — -bash — 80x8". The terminal shows the following commands and output:

```
[src$ javac CarTest.java
[src$ java CarTest
Exception in thread "main" java.lang.IllegalArgumentException: Nie możesz zatankować tyle paliwa. Zmieści się jeszcze maksymalnie 50.0
    at Car.refuel(Car.java:8)
    at CarTest.main(CarTest.java:4)

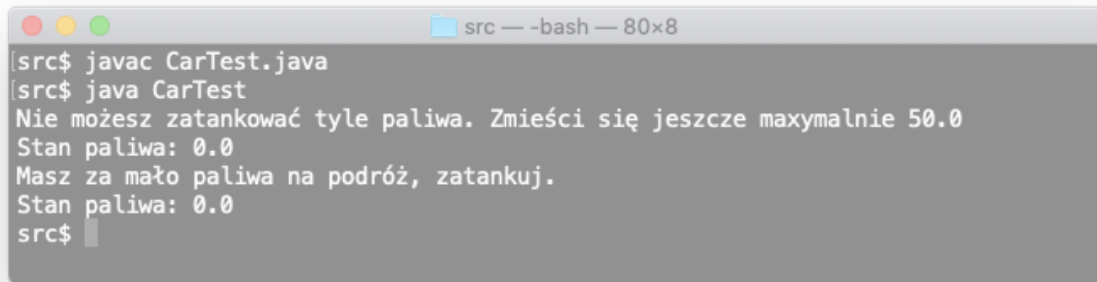
src$
```

W klasie `CarTest` musimy teraz złapać wyjątki, które rzuciliśmy w metodach klasy `Car`.

CarTest.java

```
class CarTest {
    public static void main(String[] args) {
        Car car = new Car();
        try {
            car.refuel(60);
        } catch (IllegalArgumentException ex) {
            System.err.println(ex.getMessage());
        }
        System.out.println(car);
        try {
            car.drive();
        } catch (IllegalStateException ex) {
            System.err.println(ex.getMessage());
        }
        System.out.println(car);
    }
}
```

Wywołania metod `refuel()` oraz `drive()` opakowaliśmy w bloki `try catch`. Ponieważ metoda `refuel()` wywołana z argumentem 60 jest niepoprawna, to obiekt wyjątku `IllegalArgumentException`, który stworzyliśmy i rzuciliśmy z jej wnętrza, zostanie przypisany do parametru `ex` klauzuli `catch`. Metoda `getMessage()` pozwala wyciągnąć z obiektu wyjątku wiadomość, którą ustawiliśmy przy wywoływaniu konstruktorów `IllegalArgumentException` lub `IllegalStateException` w klasie `Car`.

A terminal window titled 'src — -bash — 80x8' showing the execution of a Java program. The user enters 'javac CarTest.java' and 'java CarTest'. The program outputs: 'Nie możesz zatankować tyle paliwa. Zmieści się jeszcze maksymalnie 50.0', 'Stan paliwa: 0.0', and 'Masz za mało paliwa na podróż, zatankuj.' followed by 'Stan paliwa: 0.0' and a prompt 'src\$'.

```
[src$ javac CarTest.java
[src$ java CarTest
Nie możesz zatankować tyle paliwa. Zmieści się jeszcze maksymalnie 50.0
Stan paliwa: 0.0
Masz za mało paliwa na podróż, zatankuj.
Stan paliwa: 0.0
src$ ]
```

Doszliśmy do efektu takiego samego jak wcześniej z instrukcją if-else, jednak teraz nasze rozwiązanie jest bardziej uniwersalne. Rozwiązanie nie jest przywiązane tylko do konsoli. Przykładowo jeśli klasy Car zamiast w klasie CarTest użylibyśmy w projekcie z graficznym interfejsem użytkownika, to komunikaty o błędach moglibyśmy ustawiać w wygodny sposób jako etykiety lub inne elementy w okienku aplikacji zapisując schematycznie:

```
Car car = new Car();
try {
    car.refuel(60);
} catch (IllegalArgumentException ex) {
    oknoAplikacji.ustawEtykiety(ex.getMessage());
}
```