

# Obsługa wyjątków w try catch

## Czego się dowiesz

Czym są wyjątki,  
jaka jest hierarchia wyjątków w Javie,  
jak działa blok try catch i multi-catch,  
jak obsługiwać kilka wyjątków w jednym bloku catch,  
jak działa blok finally.

## Wstęp

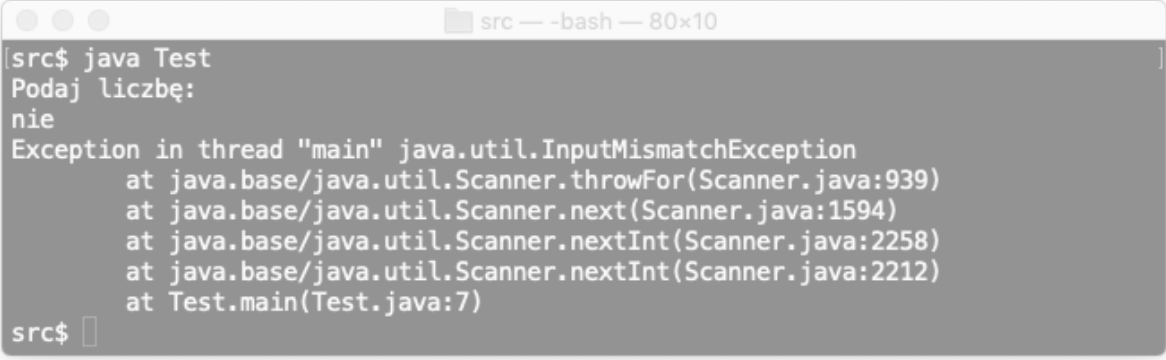
Do tej pory nie zwracaliśmy większej uwagi na to, że nasz program może trafić na pewne sytuacje, które spowodują błędy i zakończenie programu. Przykładowo jeśli od użytkownika odbieraliśmy dane za pomocą obiektu klasy Scanner, to gdyby wprowadził on zamiast liczby jakiś znak, albo do liczby int próbował przypisać liczbę zmiennoprzecinkową wystąpiłby błąd niezgodności typów. W Javie takie sytuacje nazywamy **wyjątkami** (eng. exception). W celu zobrazowania problemu zaczniemy również od przykładu:

*Test.java*

```
import java.util.Scanner;

class Test {
    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        System.out.println("Podaj liczbę: ");
        int number = sc.nextInt();
        System.out.println("Podałeś: " + number);
        sc.close();
    }
}
```

W ogólności program jest bardzo prosty, jeśli jednak trafimy na użytkownika-buntownika, który postanowi przetestować odporność naszego programu na błędy, to zamiast liczby, może wprowadzić słowo:

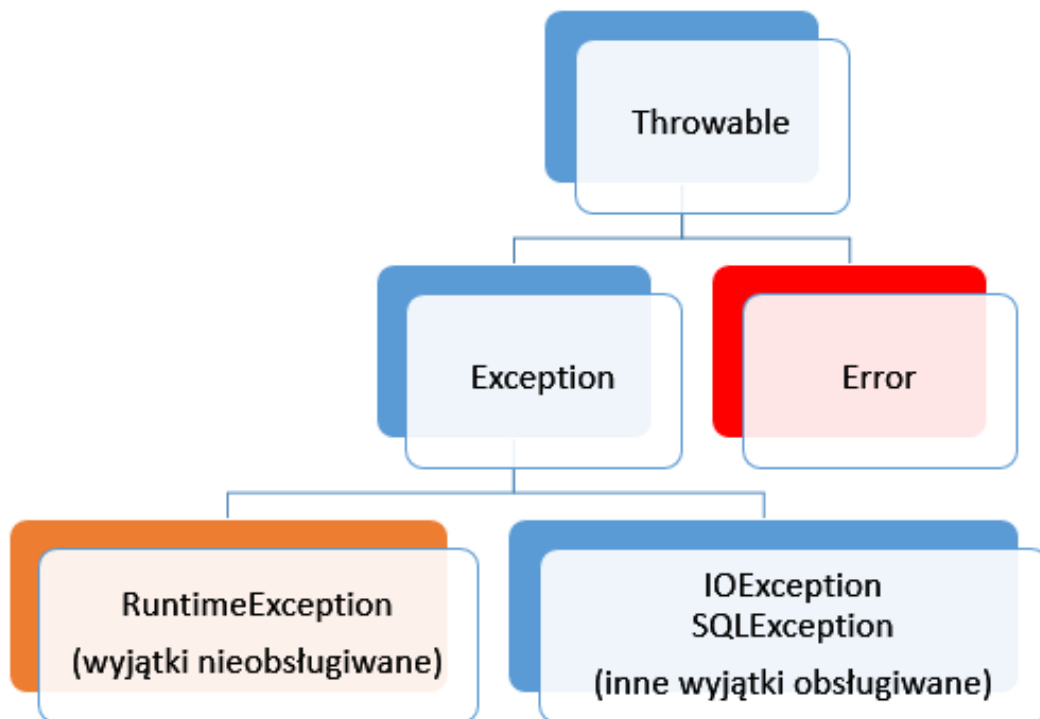
A screenshot of a terminal window with a title bar that says 'src — -bash — 80x10'. The terminal shows a Java program execution. The user enters 'src\$ java Test'. The program prompts 'Podaj liczbę:' and the user enters 'nie'. This causes an 'Exception in thread "main" java.util.InputMismatchException' to be thrown. The stack trace shows the exception originates from 'Test.main' at line 7, propagating through 'Scanner.nextInt' and 'Scanner.throwFor' in the Java base classes.

```
[src$ java Test
Podaj liczbę:
nie
Exception in thread "main" java.util.InputMismatchException
    at java.base/java.util.Scanner.throwFor(Scanner.java:939)
    at java.base/java.util.Scanner.next(Scanner.java:1594)
    at java.base/java.util.Scanner.nextInt(Scanner.java:2258)
    at java.base/java.util.Scanner.nextInt(Scanner.java:2212)
    at Test.main(Test.java:7)
src$ ]
```

Jak widzisz nasz zbuntowany tester aplikacji wprowadził słowo "nie" przez co program **rzucił wyjątek** typu `InputMismatchException`. Jest on spowodowany tym, że metoda `nextInt()` z klasy `Scanner` oczekiwała liczby całkowitoliczbowej, natomiast wyraz liczbą zdecydowanie nie jest.

Pomimo, że dopiero teraz mówimy o wyjątkach, to już spotkaliśmy się z nimi w lekcjach poświęconym tablicom. Pamiętasz może, gdy ktoś próbował odwołać się do elementu tablicy, który wykraczał poza jej zakres? Rzucany był wtedy wyjątek `ArrayIndexOutOfBoundsException`. Innym typem wyjątku, który spotkaliśmy to popularny w Javie `NullPointerException`.

W Javie istnieje dość dużo zdefiniowanych typów wyjątków, które pozwalają na obsługę najbardziej popularnych sytuacji i możliwych do wystąpienia błędów. Hierarchia dziedziczenia klas wyjątków przedstawia się tak jak poniżej:



Jak widzisz klasą wspólną dla wszystkich wyjątków jest Throwable. Jej subklasami są Exception oraz Error. Różnią się one tym, że wyjątki pochodne klasy Error nie mają sensu obsługi i niekoniecznie występują z winy programisty, ale np. z powodu problemu maszyny wirtualnej. Pochodne klasy Exception często dają się w sensowny sposób obsłużyć i naprawić lub przynajmniej nie doprowadzić do zakończenia działania aplikacji.

Klasa Exception posiada wiele klas pochodnych, które w ogólności dzielą się na dwie grupy:

**kontrolowane** (eng. checked exceptions) -

weryfikowane na etapie kompilacji, muszą być obsługiwane w kodzie programu,

**niekontrolowane** (eng. unchecked exceptions) - błędy fazy wykonania (kiedy program jest już uruchomiony) - nie muszą one być jawnie obsługiwane w kodzie programu.

Do wyjątków niekontrolowanych zaliczają się m.in. wcześniej wspomniane `IndexOutOfBoundsException` oraz `InputMismatchException`. Jeżeli jakaś metoda, czy obiekt będzie potrzebował obsługi wyjątku kontrolowanego (np. z sytuacją taką spotkamy się przy zapisie i odczycie z plików), to poinformuje cię o tym kompilator lub środowisko programistyczne podkreślając odpowiedni fragment kodu, który będzie wymagał twojej uwagi.

## Blok try catch

Pierwszym sposobem obsługi wyjątków jest zastosowanie bloku **try catch**. Opisowo konstrukcję tę można opisać jako "spróbuj przechwycić wyjątek w bloku try i jeśli on faktycznie wystąpi to wykonaj kod w bloku catch".

Jeżeli chcielibyśmy obsłużyć wyjątek `InputMismatchException` z naszego wcześniejszego przykładu, to można to zrobić tak:

*Test.java*

```
import java.util.InputMismatchException;
import java.util.Scanner;

class Test {
    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        System.out.println("Podaj liczbę: ");
        int number = 0;
        try {
            number = sc.nextInt();
            System.out.println("Wczytano poprawną liczbę");
        } catch (InputMismatchException ex) {
            System.err.println("Podana wartość nie jest liczbą całkowitą");
            ex.printStackTrace();
        }
        System.out.println("Podałeś: " + number);
        sc.close();
    }
}
```

Najpierw zadeklarowaliśmy i zainicjowaliśmy zmienną `number`. Następnie w bloku **try{ }** umieszczony został kod, który może spowodować problemy, czyli odebranie nieprawidłowej wartości od użytkownika. Jeżeli użytkownik wprowadzi nieprawidłową wartość i wystąpi wyjątek `InputMismatchException`, to sterowanie programu trafia do bloku **catch**. Możesz go sobie wyobrazić jako metodę, która przyjmuje argument z obiektem wyjątku. W naszym przypadku jest to argument o nazwie `ex` typu `InputMismatchException`.

Na ekranie wyświetlamy informację o tym, że podana wartość jest nieprawidłowa, a następnie wyświetlamy ślad stosu wywołując metodę `printStackTrace()` dziedziczoną z klasy `Throwable`. `Stacktrace` pojawia się standardowo przy wystąpieniu wyjątku, jednak, gdy wyjątku nie obsłużymy, to jednocześnie program kończy swoje działanie. Jako strumień wyjścia zastosowaliśmy `System.err` - działa on analogicznie do `System.out`, jednak kod w konsoli będzie wyróżniony kolorem czerwonym. Ślad stosu pozwala dojść krok po kroku do miejsca, w którym wystąpił problem. W konsoli zobaczymy stos wywołań metod i informację w którym wierszu, której klasy wyjątek ma swoje źródło i jak był później propagowany.

Najważniejsze jest jednak to, że dzięki obsłużeniu wyjątku program kontynuuje swoją pracę i pomimo, że wyświetla błędną wartość, bo 0, to jednak kończy swoją pracę.

---

Podaj liczbę:

nie

Podana wartość nie jest liczbą całkowitą

`java.util.InputMismatchException`

at java.base/java.util.Scanner.throwFor(Scanner.java:939)

at java.base/java.util.Scanner.next(Scanner.java:1594)

at java.base/java.util.Scanner.nextInt(Scanner.java:2258)

at java.base/java.util.Scanner.nextInt(Scanner.java:2212)

at Test.main(Test.java:10)

Podałeś: 0

Zauważ, że w konsoli nie wyświetlił się tekst *"Wczytano poprawną liczbę"*. Ponieważ w wierszu 10 naszego programu wystąpił wyjątek, to sterownie programu zostało natychmiast skierowane do bloku catch. Program po wyświetleniu stosu wywołań nie został jednak przerwany i końcowa metoda `System.out.println()` także została wykonana. Gdybyśmy nie zastosowali bloku try catch, działanie programu zostałoby natychmiast przerwane i tekst *"Podałeś 0"* nie zostałby wyświetlony.

Program można jeszcze usprawnić i prosić użytkownika o podanie wartości, dopóki nie będzie ona poprawna. W tym celu dodajemy pętlę z prostą flagą typu boolean, która pozwoli określić, czy już poprawnie wczytano liczbę (flagą w programowaniu nazywa się zmienną, która oznacza pewien stan aplikacji, np. poprawny/niepoprawny).

*Test.java*

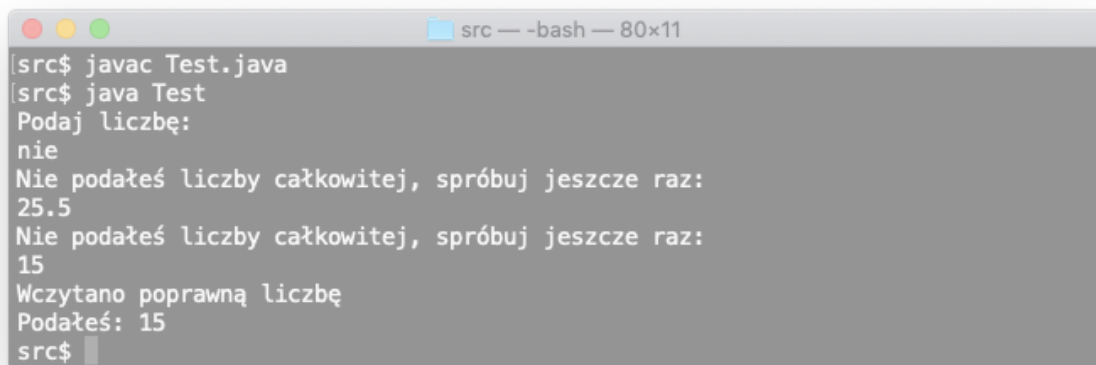
```
import java.util.InputMismatchException;
import java.util.Scanner;

public class Test {
    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        System.out.println("Podaj liczbę: ");
        int number = 0;
        boolean error = true;
        do {
            try {
                number = sc.nextInt();
                error = false;
                System.out.println("Wczytano poprawną liczbę");
            } catch (InputMismatchException e) {
                System.out.println("Nie podałeś liczby całkowitej,
spróbuj jeszcze raz: ");
                sc.nextLine();
            }
        } while (error);
        System.out.println("Podałeś: " + number);
        sc.close();
    }
}
```

Oprócz zmiennej `number` zainicjowaliśmy zmienną `error`. Ponieważ korzystamy z pętli `while` to pętla wykona się przynajmniej raz. Jeżeli ktoś poda prawidłową wartość, to ustawiamy zmienną `error` na `false` i pętla nie wykona się ponownie. Jeśli ktoś poda np. napis zamiast liczby, to wiersz `error = false;` się nie wykona, bo sterowanie zostanie przeniesione do bloku `catch`. Tym samym w przypadku podania błędnej wartości zmienna `error` cały czas będzie miała wartość `true` i pętla będzie się powtarzać.

W bloku `catch` oprócz wyświetlenia komunikatu o błędzie wywołujemy metodę `nextLine()`. Jest to konieczne, ponieważ chcemy pozbyć się z bufora konsoli błędnej wartości, którą wprowadził użytkownik.

W ten sposób program nie zwróci błędu, a nasz program stał się "idiotoodporny".



```
src — -bash — 80x11
[src$ javac Test.java
[src$ java Test
Podaj liczbę:
nie
Nie podałeś liczby całkowitej, spróbuj jeszcze raz:
25.5
Nie podałeś liczby całkowitej, spróbuj jeszcze raz:
15
Wczytano poprawną liczbę
Podajesz: 15
src$
```

# Try multi-catch

Jeżeli w bloku try{} umieścimy bardziej skomplikowany kod, to może się zdarzyć, że będzie on mógł wygenerować więcej niż 1 rodzaj wyjątku. W takiej sytuacji warto posłużyć się blokiem multi-catch, który pozwoli przechwycić każdy z wyjątków osobno i obsłużyć je w różny sposób.

Poprzedni przykład rozszerzmy o możliwość wprowadzenia dwóch liczb, które będą zapisywane do tablicy z rozmiarze 2, a następnie poprosimy użytkownika o wybór wartości do wyświetlenia.

*Test.java*

```
import java.util.InputMismatchException;
import java.util.Scanner;

class Test {
    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        int[] numbers = new int[2];
        boolean error = true;

        do {
            try {
                System.out.println("Podaj 1 liczbę: ");
                numbers[0] = sc.nextInt();
                sc.nextLine();
                System.out.println("Podaj 2 liczbę: ");
                numbers[1] = sc.nextInt();
                sc.nextLine();

                System.out.println("Którą wartość wyświetlić (1 lub 2)?");

                System.out.println("Liczba:" + numbers[sc.nextInt() - 1]);

                error = false;
            } catch (InputMismatchException ex) {
                System.out.println("Nie podałeś liczby całkowitej, spróbuj jeszcze raz: ");
                sc.nextLine();
            } catch (ArrayIndexOutOfBoundsException ex) {
                System.out.println("Miało być 1 lub 2, zacznijmy od nowa: ");
                sc.nextLine();
            }
        } while (error);
        sc.close();
    }
}
```

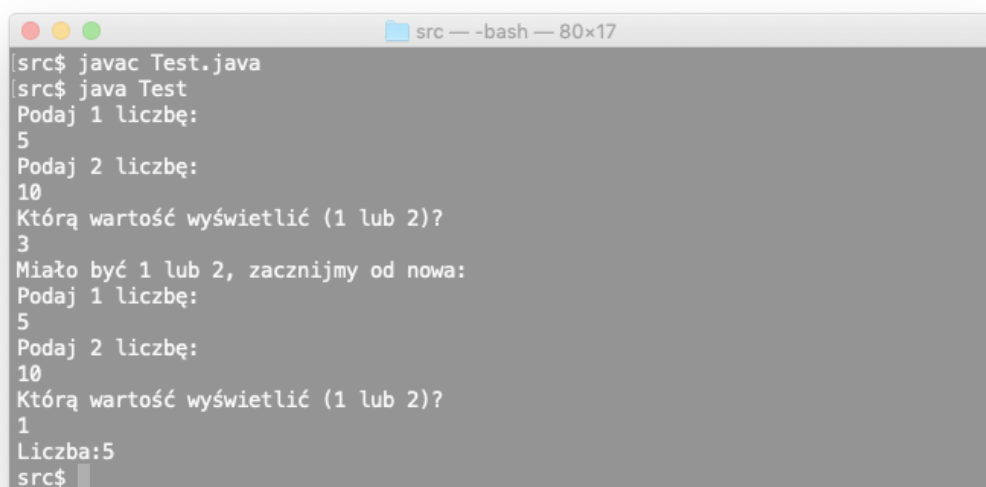


Wartości od użytkownika będą teraz przechowywane w tablicy numbers. W bloku try wczytujemy od użytkownika dwie liczby i umieszczamy je w tablicy. Następnie prosimy użytkownika podanie indeksu liczby, którą chce zobaczyć i wyświetlamy odpowiednią wartość z tablicy. Odwołujemy się do indeksu pomniejszonego o 1 względem liczby podanej przez użytkownika `numbers[sc.nextInt() - 1]` dzięki temu ktoś może podać liczbę w ludzkiej formie, a nie liczonej od 0.

W powyższym kodzie mogą wystąpić dwa wyjątki:

- jeśli użytkownik wprowadzi zamiast liczby np. tekst, rzucony będzie wyjątek `InputMismatchException`,
- jeśli użytkownik poda jako indeks liczbę inną niż 1 lub 2, to przy odwoływaniu się do tablicy otrzymamy wyjątek `ArrayIndexOutOfBoundsException`.

Cały problematyczny kod możemy objąć jednym blokiem try, ale dopisać do niego kilka bloków catch. W zależności od tego, który rodzaj wyjątku zostanie rzucony, to zostaniemy przeniesieni do odpowiedniego bloku catch.



```
src$ javac Test.java
src$ java Test
Podaj 1 liczbę:
5
Podaj 2 liczbę:
10
Którą wartość wyświetlić (1 lub 2)?
3
Miało być 1 lub 2, zacznijmy od nowa:
Podaj 1 liczbę:
5
Podaj 2 liczbę:
10
Którą wartość wyświetlić (1 lub 2)?
1
Liczba:5
src$
```

Od Javy 7 istnieje również inny zapis, który pozwala wszystkie rodzaje wyjątków przechwycić w jednym bloku `catch()`. Robimy to rozdzielając typy wyjątków pionową kreską:

```
try{
//kod z błędami
} catch(InputMismatchException | ArrayIndexOutOfBoundsException ex) {
    //dowolne instrukcje, np.:
    ex.printStackTrace();
}
```

Jeżeli chcemy obsłużyć kilka wyjątków w jednym bloku `catch`, to możemy także przechwycić typ, który jest nadrzędny dla typów, które chcemy obsłużyć. Przykładowo jeśli chcemy obsłużyć `ArrayIndexOutOfBoundsException` i `NullPointerException`, to możemy obsłużyć nadrzędny dla nich `RuntimeException` lub jeszcze wyżej `Exception`:

```
try{
//kod z błędami
} catch(Exception ex) {
    //dowolne instrukcje, np.:
    ex.printStackTrace();
}
```

W praktyce lepiej definiować osobne bloki `catch()` dla różnych typów wyjątków, bo dzięki temu jesteśmy w stanie dostarczyć użytkownikowi naszej aplikacji bardziej precyzyjną informację niż "wystąpił błąd".

## Blok finally

W bloku `try...catch` warto wspomnieć o możliwości dodania jeszcze jednej sekcji - **finally**. Jest to część kodu, która wykona się zawsze, niezależnie od tego, czy wyjątek wystąpi, czy też nie. W naszym kodzie w różnych blokach `catch` na końcu dodawaliśmy wywołanie metody `nextLine` na obiekcie `Scanner`, w celu pozbycia się znaku nowej linii z bufora. Zamiast zapisywać go kilka razy, dla różnych typów

wyjątków, lepiej jest przenieść to wywołanie do bloku

**finally:**

```
import java.util.InputMismatchException;
import java.util.Scanner;

class Test {
    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        int[] numbers = new int[2];
        boolean error = true;

        do {
            try {
                System.out.println("Podaj 1 liczbę: ");
                numbers[0] = sc.nextInt();
                sc.nextLine();
                System.out.println("Podaj 2 liczbę: ");
                numbers[1] = sc.nextInt();
                sc.nextLine();

                System.out.println("Którą wartość wyświetlić (1 lub 2)? ");
                System.out.println("Liczba:" + numbers[sc.nextInt() - 1]);
                error = false;
            } catch (InputMismatchException ex) {
                System.out.println("Nie podałeś liczby całkowitej, spróbuj
jeszcze raz: ");
            } catch (ArrayIndexOutOfBoundsException ex) {
                System.out.println("Miało być 1 lub 2, zaczniemy od nowa: ");
            } finally {
                sc.nextLine(); //wykona się zawsze, niezależnie, czy wyjątki
wystąpią, czy nie
            }
        } while (error);
        sc.close();
    }
}
```

Teraz `sc.nextLine()` zostanie wywołane niezależnie, czy wyjątek wystąpi (nie ma też znaczenia jakiego będzie typu), czy też nie.

Sekcję `finally` wykorzystuje się często do zamykania strumieni wejścia/wyjścia - np. jeżeli nasz powyższy program nie prosił użytkownika ponownie o podanie poprawnych danych, to wywołanie `sc.close()` również moglibyśmy przenieść do sekcji `finally`.