Java – Notatki

Spis treści

[1. OOP 1](#_Toc79743524)

[2. Wyjątki 2](#_Toc79743525)

[2.1. Try carch finally 2](#_Toc79743526)

[2.2. Try-with-resource 2](#_Toc79743527)

[2.3. Rodzaje wyjątków 2](#_Toc79743528)

[2.4. Throws 3](#_Toc79743529)

[2.5. Rzucanie wyjątków 4](#_Toc79743530)

[2.6. Tworzenie swoich wyjątków 4](#_Toc79743531)

[2.7. Wiązanie wyjątków (chaining exceptions) 4](#_Toc79743532)

[2.8. Dobre praktyki 5](#_Toc79743533)

[3. Kolekcje 7](#_Toc79743534)

## OOP

<https://stackify.com/oops-concepts-in-java/>

### Abstraction

Abstraction - Using simple things to represent complexity. In Java, abstraction means simple things like objects, classes and variables represent more complex underlying code and data. This is important because it lets you avoid repeating the same work multiple times.

Data Abstraction may also be defined as the process of identifying only the required characteristics of an object ignoring the irrelevant details. The properties and behaviours of an object differentiate it from other objects of similar type and also help in classifying/grouping the objects.

### Encapsulation

The practice of keeping fields within a class private, then providing access to those fields via public methods. Encapsulation is a protective barrier that keeps the data and code safe within the class itself. We can then reuse objects like code components or variables without allowing open access to the data system-wide.

Another way to think about encapsulation is, it is a protective shield that prevents the data from being accessed by the code outside this shield.

* Technically in encapsulation, the variables or data of a class is hidden from any other class and can be accessed only through any member function of own class in which they are declared.
* As in encapsulation, the data in a class is hidden from other classes, so it is also known as **data-hiding**.
* Encapsulation can be achieved by Declaring all the variables in the class as private and writing public methods in the class to set and get the values of variables.

### Inheritance

A special feature of Object-Oriented Programming in Java, Inheritance lets programmers create new classes that share some of the attributes of existing classes

### Polymorphism

Allows programmers to use the same word in Java to mean different things in different contexts.

### SOLID principles

* Single Responsibility – class should only have one responsibility
* Open/Closed - Open for Extension, Closed for Modification

Classes should be open for extension but closed for modification. In doing so, we stop ourselves from modifying existing code and causing potential new bugs in an otherwise happy application.

Of course, the one exception to the rule is when fixing bugs in existing code.

Aby przestrzegać zasady otwarty/zamknięty najlepiej jest użyć wzorców projektowych. Dzięki nim pośrednio skorzystamy także z dziedziczenia i polimorfizmu, a nasze rozwiązanie będzie zrozumiałe dla innych programistów. Najlepszymi wzorcami, które mogą okazać się pomocne, są: metoda fabrykująca, budowniczy, metoda szablonowa, strategia i most. Wszystkie z nich pozwalają dopisać dodatkową implementację, bez zmiany klas bazowych.

* Liskov Substitution

Any subclass can be used in place of the base class (all methods match).

Example

Interface Animal with method run. We are calling run method for classes which implement Animal interface. If at least class (i.e. Fish) couldn’t invoke run method the Liskov principle is broken.

* Interface Segregation

The interfaces should be as small as possible. So that they do not provide functionality not needed in the class. Big interfaces should be splitted into smaller ones.

* Dependency Inversion

Both, high-level and low-level modules should depend on abstraction. Polymorphism Interfaces or abstract class (abstraction) should be used instead of normal classes wherever it is possible.

## Wyjątki

## Try carch finally

try {

Normalny kod, który może rzucić wyjątek.

Jeśli wystąpi wyjątek to od razu kończy wykonywanie kodu z tego bloku i przechodzi do catch.

} catch {

Obsługa błędu, działa tylko jeśli wrzucony został zadeklarowany wyjątek

Można użyć kilku bloków catch do obsługi różnych wyjątków. Lepiej umieścić bardziej szczegółowe wyjątki wyżej, żeby w razie czego wykonały się zamiast tych mniej szczegółowych.

Może być tylko blok finally – bez catch

} finally {

Cleanup

Close file, database, etc

}

## Try-with-resource

Konstrukcja wygląda jak try/catch z tym, że przez blokiem objętym try możemy zainicjalizować zmienne, które zostaną automatycznie zamknięte. Kompilator widząc konstrukcję try-with-resources wygeneruje za nas kod zbliżony do tego z pierwszego przykładu (w podobny sposób generuje np. domyślne konstruktory).

Może być używany z typami, które implementują interfejs AutoCloseable

try (BufferedReader reader = new BufferedReader(new FileReader(filename))) {

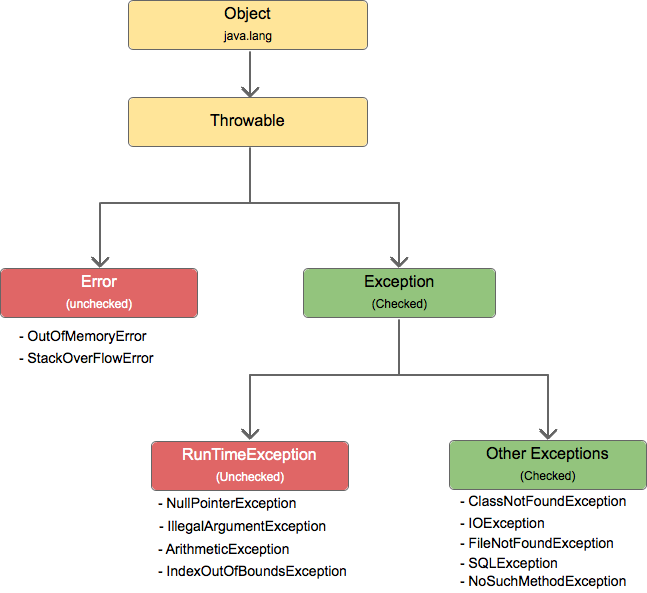
} catch {

}

Jeśli wystąpi błąd to reader zostanie automatycznie zamknięty.

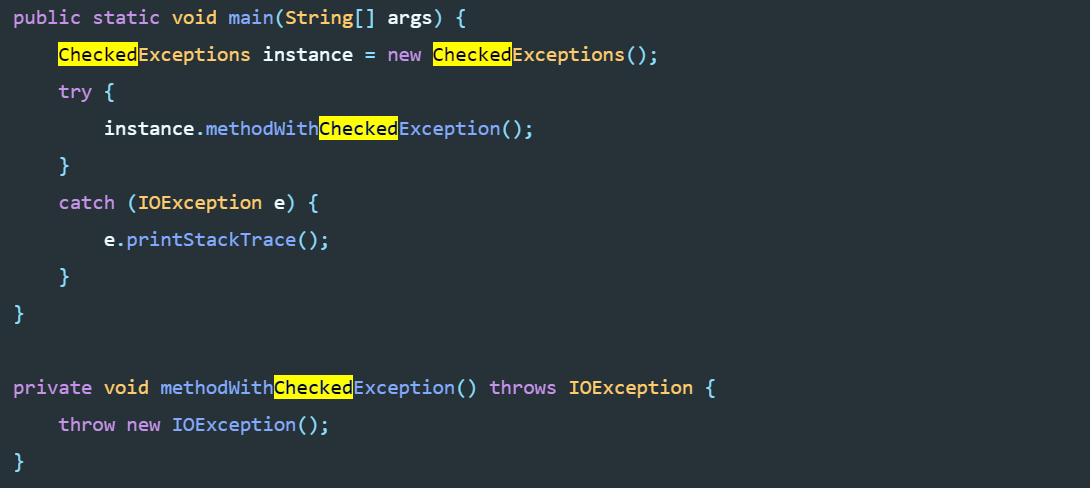
## Rodzaje wyjątków

Wyróżniamy dwa rodzaje wyjątków, tak zwane “checked exceptions” oraz “unchecked exceptions”. Różnica między nimi sprowadza się do tego, że te pierwsze muszą być obsłużone przez programistę, wymaga tego kompilator. Przykładowym wyjątkiem typu unchecked jest IllegalArgumentException, natomiast IOException jest wyjątkiem typu checked.

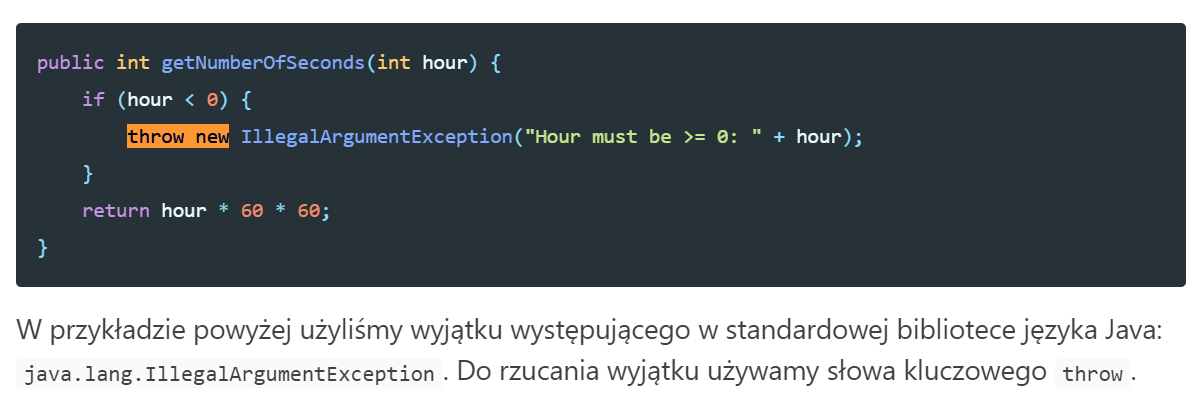


## Throws

Wyjątek można obsłużyć na dwa sposoby. Jeden już znasz, to otoczenie fragmentu kodu blokami try/catch. Drugi sprowadza się do “zepchnięcia” odpowiedzialności obsłużenia wyjątku o poziom niżej, do metody wywołującej. Służy do tego klauzula throws, którą dodajemy do deklaracji metody.

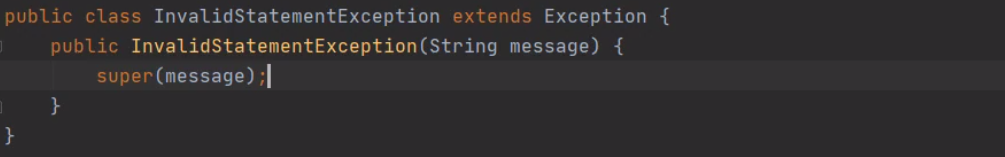


## Rzucanie wyjątków



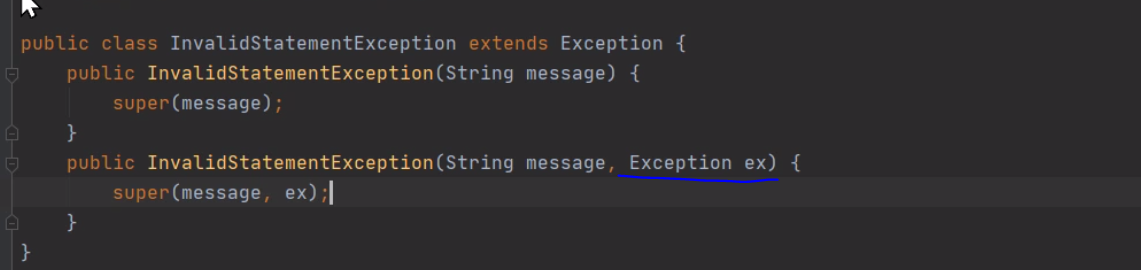
## Tworzenie swoich wyjątków

Tworzymy nową klasę dziedziczącą po klasie Exception albo innej dziedzczącej po niej. Powinniśmy dodać odpowiedni konstruktor, żeby umożliwić tworzenie wyjątków.

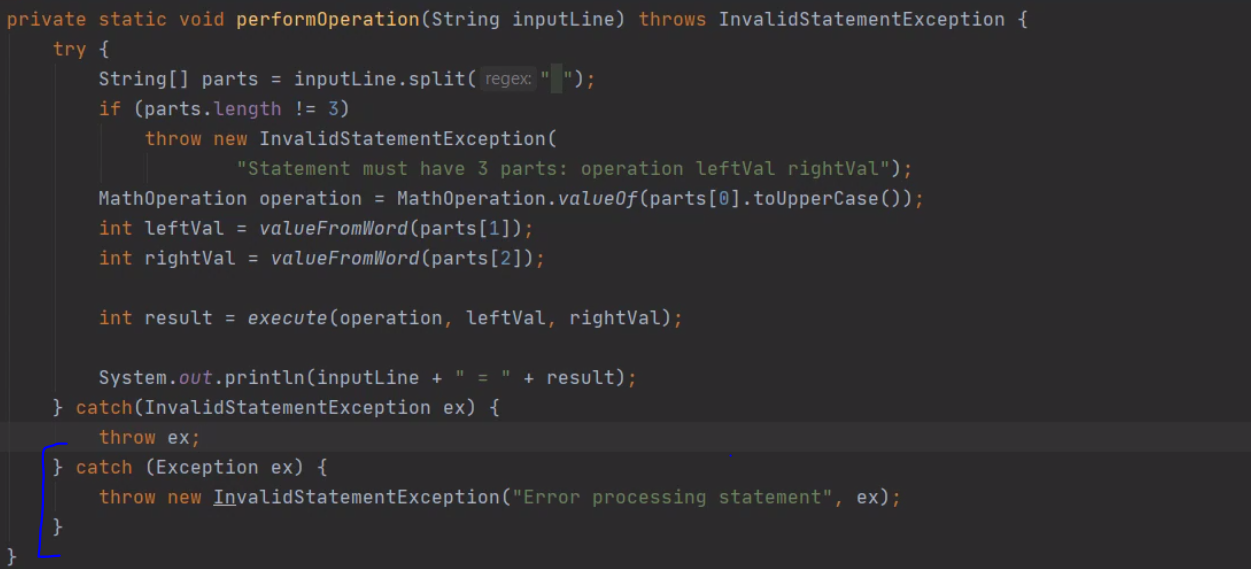


## Wiązanie wyjątków (chaining exceptions)

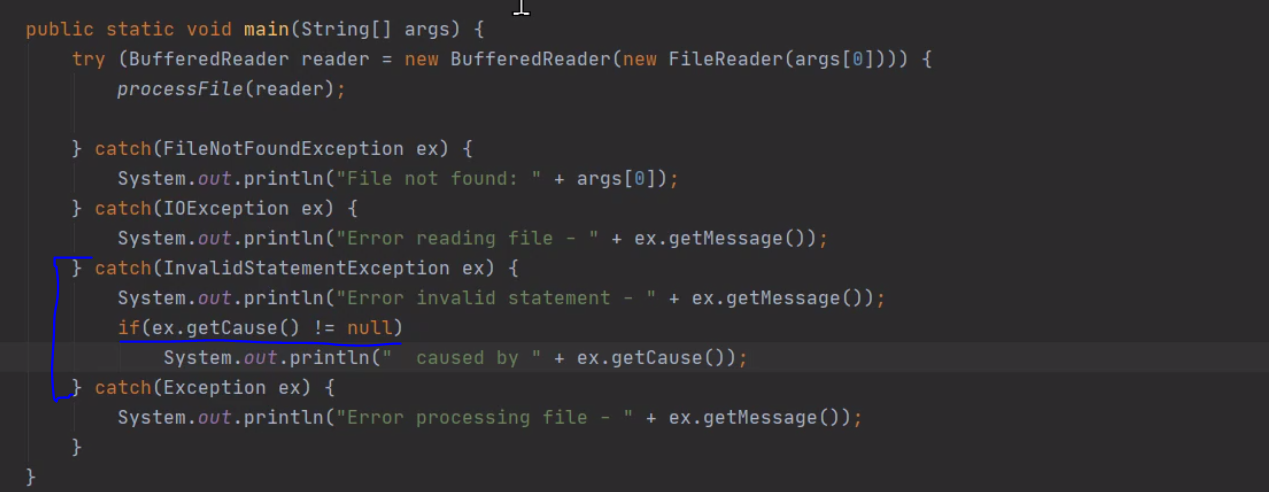
Żeby utworzyć wyjątek powiązany z innym musimy do klasy wyjątku dodać konstruktor z argumentem typu Throwable/Exception. A następnie tworząc wyjątek przekazać orginalny do konstruktora. Poźniej za pomocą funkcji getCause możemy pobrać orginalny wyjątek.



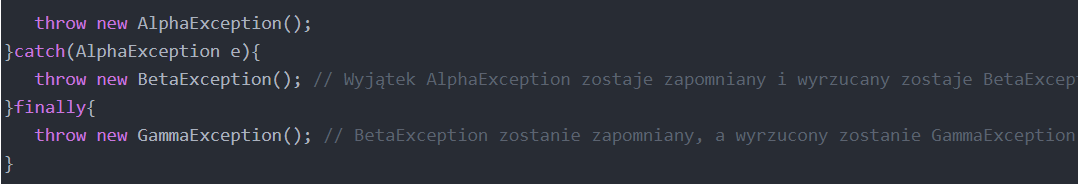
Poniżej jeśli wystąpi jakiś nieoczekiwany wyjątek zwracamy go jako InvalidStatementException.



Za pomocą funkcji getCause sprawdzamy wyjątek wystąpił przez rzucenie wyjątku innego typu.

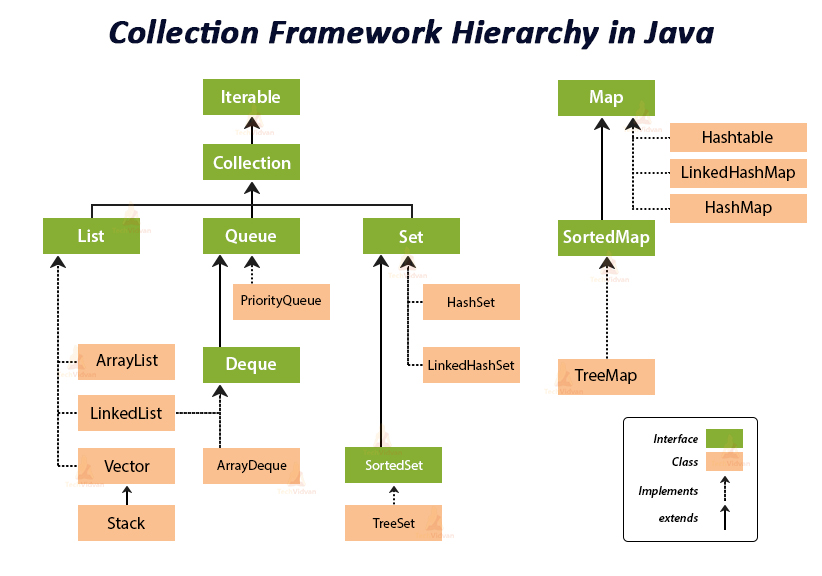


## Dobre praktyki

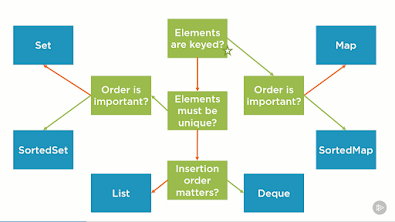
* Powinniśmy zdecydować się na jedną rzecz albo loguje do logów, albo przekazuje wyjątek dalej
* Złą praktyką jest przepychanie wyjątków w górę stosu z metody na metodę. Wyjątek powinien być, w miarę szybko obsłużony, a jeżeli tak nie jest to może powinien być wyjątkiem typu RuntimeException
* Problem polega na tym, że zostaje wykonany tylko pierwszy blok catch, który pasuje do wyjątku. Tak więc, jeśli najpierw złapiesz wyjątek IllegalArgumentException, nigdy nie osiągniesz bloku catch, który powinien obsłużyć bardziej specyficzny wyjątek NumberFormatException, ponieważ jest to podklasa wyjątku IllegalArgumentException.
* Uważaj na wyjątki wyrzucane w blokach catch i finally. Kiedy nowy wyjątek zostanie zgłoszony w bloku catch lub w finally, który będzie propagował z tego bloku, wówczas obecny wyjątek zostanie przerwany (i zapomniany), ponieważ nowy wyjątek jest propagowany na zewnątrz.
* Iinformacja o wyjątku ma dwóch odbiorców: użytkownika i programistę. Dla użytkownika powinniśmy zwracać prostą informację o tym jaki błąd i gdzie wystąpił. Dla programisty szczegółową informację. To co dla użytkownika, stack trace. Zapisać to do sytemu logowania.

## Kolekcje

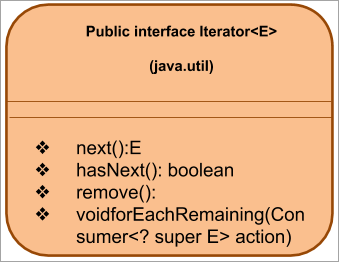
## Collection Framework - hierarchia



## Której kolekcji użyć?



## Interface Iterator



Graphical user interface, text, application, email

Description automatically generated

## Sortowanie, Comparator

## List

## Map