Exceptions

Spis treści

Zapiski prowadzącego Karola Rogowskiego i uczestnika Bootcampu Zajavka Bartek Borowczyk aka Samuraj Programowania.

Co to i po co to?

Nareszcie przechodzimy do czegoś wyjątkowego! To znaczy sytuacji... wyjątkowych. To znaczy takich, które nie są normalne, czyli są wyjątkowe. Chyba taki był powód, czemu to zostało tak nazwane.

Powodów powstania wyjątku w kodzie może być dużo:

- próbujemy wczytać plik z dysku, ale plik nie istnieje,
- staramy się podzielić przez 0,
- próbujemy pobrać jakąś informację z jakiejś strony internetowej, ale Internet przestał działać,
- staramy się dostać w tablicy do indeksu, który nie istnieje.

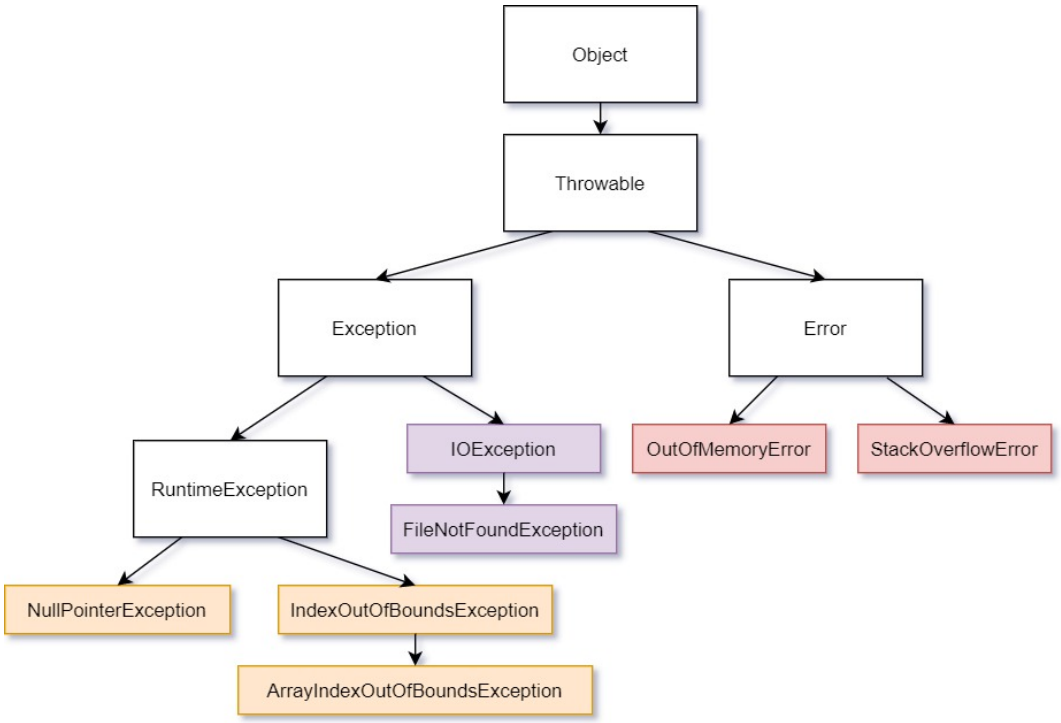
Może przypadek dzielenia przez 0 jest skrajny, ale w trakcie skomplikowanych obliczeń, może przypadkiem taka sytuacja wystąpić. Należy wtedy przygotować się na jej obsługę w kodzie.

Sam mechanizm wyjątków powstał po to, żeby dać programiście możliwość obsłużenia sytuacji, które odstają od normy, ale chcemy, żeby nasz program dalej działał, jak należy. Inaczej mówiąc, wiemy wcześniej, że taka sytuacja może wystąpić i chcemy być na nią odpowiednio przygotowani.

Możliwe też jest, że przygotujemy się na taką obsługę, a sytuacja wyjątkowa nie wystąpi. Określamy taką sytuację jako ***happy path***.

W Javie mamy 2 podejścia, jeżeli chodzi o obsługę wyjątków. Albo może to zrobić metoda, w której taki wyjątek wystąpił, albo metoda taka może powiedzieć, "*ja nie wiem, niech to obsłuży metoda, która mnie wywołuje*" i przekazać taki wyjątek do metody, która jest źródłem wywołania, tzw. caller.

Hierarchia wyjątków



Obraz 1. Hierarchia wyjątków

Na rysunku wyżej, w kolorze innym niż biały zostały wypisane przykładowe klasy należące do jakiejś kategorii. W praktyce jest ich o wiele więcej i nie należy uczyć się tego na pamięć. Przejdźmy do konkretów.

Wyróżniamy 3 główne kategorie sytuacji wyjątkowych:

- **Error** - oznacza, że stało się coś tak niewybaczalnie złego, że nie ma już sensu próbować się z tego podnosić i otrzepywać, możemy jedynie rozłożyć ręce. Z punktu widzenia naszego programu - np. pamięć się skończyła i program nie jest w stanie rezerwować więcej miejsca w pamięci na zmienne,
- **Checked Exception** - są to wyjątki, które dziedziczą (tak, wyjątki są klasami i mogą się dziedziczyć) bezpośrednio lub pośrednio z klasy Exception. Wyjątki takie charakteryzują się tym, że na etapie kompilacji następuje sprawdzenie, czy taki wyjątek został obsłużony w metodzie lub uwzględniony w jej sygnaturze. Oznacza to, że jeżeli wiemy, że jakiś fragment kodu wyrzuca wyjątek **checked** (czyli ten sprawdzany na etapie kompilacji) to musimy tak napisać kod, żeby ten wyjątek został albo obsłużony w metodzie, którą właśnie piszemy, albo żeby ta metoda przekazała ten wyjątek do metody ją wywołującej,
- **Runtime Exception (Unchecked Exception)** - są to wyjątki, które dziedziczą bezpośrednio lub pośrednio z klasy RuntimeException. Te wyjątki nie są sprawdzane na etapie kompilacji. Programista może napisać ich obsługę, ale nie musi. Jeżeli jej nie napisze, może stać się coś, czego się nie spodziewamy, dlatego warto mieć też tego świadomość.

W dzisiejszych czasach panuje tendencja do używania wyjątków **UNchecked** (Runtime). Oczywiście jest to kwestia filozoficzna, natomiast w praktyce, jeżeli stosujemy wyjątki Runtime - nasz kod robi się czytelniejszy. Minus jest taki, że używając jakiejś metody, nie mamy świadomości o tym, że może ona wyrzucić jakiś wyjątek, bo w przypadku wyjątków Checked, taka informacja pojawiłaby się w sygnaturze metody.

Własny wyjątek

Na początek stwórzmy sobie swój własny wyjątek:

```
public class MyCheckedException extends Exception {

    public MyCheckedException() {
        super("My exception was thrown");
    }

    public MyCheckedException(String message) {
        super(message);
    }

    /* Wszystkie wyjątki rozszerzają klasę Throwable
    (zobacz jeszcze raz wykres hierarchii wyjątków).
    Możemy zatem odnosić się do wyjątków przez klasę,
    która jest rodzicem ich wszystkich (przypomnij sobie polimorfizm).
    Istnieje też konstruktor, który pozwala nam stworzyć wyjątek z innego wyjątku.
    W praktyce nazywa się to opakowaniem wyjątku, stąd taka konstrukcja jak pod spodem.
    Oznacza ona, że tworząc nasz wyjątek, przekazujemy mu inny wyjątek,
    który często jest źródłem problemu. */
    public MyCheckedException(String message, Throwable cause) {
        super(message, cause);
    }
}
```

Wywołania `super` służą do wywołania określonego konstruktora z klasy `Exception`, a nasz wyjątek `MyCheckedException` może być stworzony na 3 różne sposoby, bo ma 3 konstruktory. Wyjątek jest klasą, zatem ma konstruktor, nie jest to nic nadzwyczajnego.

Jak wyrzucić swój własny wyjątek

Wiemy już jak stworzyć swój własny wyjątek, jeszcze trzeba być w stanie go wyrzucić i go złapać. Z innymi wyjątkami jest podobnie, ale skupmy się na naszym. Wyobraźmy sobie metodę, która sprawdza, czy wiek, który jej przekazujemy, jest większy od 20, jeżeli jest, to drukuje na ekranie "OK", ale jeżeli nie jest, to wyrzuca nasz wyjątek z wiadomością "Sorry, ale jesteś za młody". Kod, który to robi, mógłby wyglądać w ten sposób:

```
public void ageChecker(int age) {
    if (age > 20) {
        System.out.println("OK");
    } else {
        // niżej wyjaśniam czemu tutaj dostajemy błąd
        throw new MyCheckedException("Sory, ale jesteś za młody");
    }
}
```

Fragment `throw new MyCheckedException("Sorry, ale jesteś za młody");` odpowiada za faktyczne wyrzucenie wyjątku w momencie, gdy program dojdzie do wykonania tego fragmentu kodu. Wyjątek tworzymy jak obiekt przez słówko `new` i dlatego cały czas mówię wyrzucać, bo pojawia nam się słówko `throw`.

Jeżeli przepiszesz ten kod do Intellij, to zobaczysz też, że nie kompiluje się on poprawnie. Wynika to z tego, że nasz wyjątek `MyCheckedException` jest wyjątkiem **Checked Exception**, bo dziedziczy z klasy `Exception`. Checked Exception są klasą wyjątków, dla których na etapie kompilacji kompilator sprawdza, czy obsłużyliśmy ten wyjątek jakoś po wyrzuceniu. Jak możesz zauważyć, w tym przykładzie tylko wyrzuciliśmy wyjątek, ale nie napisaliśmy żadnej obsługi do niego, dlatego też dostajemy błąd kompilacji. Wyjątek taki możemy po wyrzuceniu natomiast obsłużyć na 2 sposoby.

Blok try-catch

Pierwszym sposobem na obsłużenie wyjątku jest klauzula `try-catch`. Czyli fragment kodu, mówiący, że jeżeli jakaś metoda wyrzuca wyjątek, to my go próbujemy (`try`) złapać (`catch`) w siatkę (nie ma w Javie słowa na siatkę). A jak już go złapiemy, to staramy się coś z nim zrobić. Możemy go zignorować, ale możemy też wydrukować na ekranie, że taki wyjątek wystąpił. Ważne jest to, że jeżeli taki wyjątek obsługujemy, to nasz program nie przestanie działać. Mówię, o tym, dlatego, że jeżeli ten wyjątek nie byłby **checked**, tylko był `RuntimeException` (czyli **unchecked**), to wcale nie musielibyśmy tego wyjątku obsługiwać, i wtedy nasz program mógłby przerwać swoje wykonywanie w połowie i zakończyć się w niespodziewany sposób. Ale przejdźmy do kodu, który jest w stanie taki wyjątek obsłużyć:

```
private static void ageChecker(int age) {
    if (age > 20) {
        System.out.println("OK");
    } else {
        try {
            throw new MyCheckedException("Sory, ale jesteś za młody");
        } catch (Exception e) {
            System.out.println("I caught the exception");
        }
    }
}
```

W powyższym fragmencie kodu dodano klauzulę `try-catch`. W bloku kodu `try {}` umieszczamy fragment kodu, który potencjalnie może wyrzucić wyjątek. Następnie dopisujemy `catch`, które określa jaki wyjątek staramy się złapać. W tym przykładzie można było równie dobrze napisać `catch (MyCheckedException e)`. Ważne jest to, że jeżeli mamy wyjątek Checked i piszemy `try-catch` to musimy albo złapać nasz konkretny wyjątek, albo możemy złapać jego superklasę. Dlatego w tym przykładzie łapiemy `Exception`, a nie `MyCheckedException`.

Następnie możemy przejść do napisania, co zrobimy z wyjątkiem, który złapaliśmy. Blok kodu za `catch` może być pusty, tzn. `catch(Exception e) {}`. Wtedy ze złapanym wyjątkiem nic nie zrobimy. Należy tu też dodać, że łapiąc taki wyjątek, określamy nazwę zmiennej, pod którą złapany wyjątek będzie dostępny. Czyli mamy referencję do złapanego wyjątku, tutaj nazywa się ona `e`, ale może mieć dowolną nazwę.

W praktyce, po złapaniu takiego wyjątku następuje jego logowanie (nie mylić z logowaniem użytkownika w jakimś systemie). Logowanie w skrócie polega na zapisywaniu kroków wykonania aplikacji oraz wyrzucanych przez aplikację błędów do plików, aby później móc analizować błędy, które pojawiają się w systemie. Natomiast na ten moment, pomijamy aspekt logowania, wrócimy do niego w przyszłości.

My natomiast po złapaniu takiego wyjątku wydrukowaliśmy na ekranie "*I caught the exception*". Możemy również wydrukować na ekranie Stringa, który został zdefiniowany na etapie tworzenia

wyjątku:

```
// blok try
catch (Exception e) {
    System.out.println("I caught the exception: " + e.getMessage());
}
```

Możemy również wydrukować na ekranie **StackTrace**. Wydrukujemy go w ten sposób:

```
public class StackTraceExample {

    public static void main(String[] args) {
        runAgeChecker();
    }

    private static void runAgeChecker() {
        ageChecker(20);
    }

    private static void ageChecker(int age) {
        if (age > 20) {
            System.out.println("OK");
        } else {
            try {
                throw new MyCheckedException("Sorry, ale jesteś za młody");
            } catch (Exception e) {
                e.printStackTrace();
            }
        }
    }
}
```

Po wydrukowaniu **StackTrace** na ekranie, ukaże się nam coś takiego:

```
pl.zajavka.MyCheckedException: Sory, ale jesteś za młody
    at pl.zajavka.StackTraceExample.ageChecker(StackTraceExample.java:20)
    at pl.zajavka.StackTraceExample.runAgeChecker(StackTraceExample.java:12)
    at pl.zajavka.StackTraceExample.main(StackTraceExample.java:8)
```

Możesz zwrócić uwagę, że czytając taki **StackTrace** od dołu, możemy konkretnie prześledzić, w którym miejscu zaczął się nasz program, a w którym miejscu nastąpiło wyrzucenie błędu. Linijki pokazują konkretne miejsca (nazwy metod oraz numery linijek w klasie), gdzie wywołanie programu przechodziło przez kolejne metody i wchodziło coraz głębiej. Na końcu (czyli na górze, bo StackTrace czyta się od dołu) jest wydrukowana nazwa wyrzuconego wyjątku oraz wiadomość, która często określa, co jest przyczyną wyrzucenia wyjątku. Kolejne zagnieżdżenia wywołań metod staną się jeszcze bardziej jasne, jak zobaczysz materiał dotyczący modelu pamięci w Javie.

Klauzula throws

Powiedzieliśmy, że jednym ze sposobów obsłużenia wyjątku jest `try-catch`. Drugim sposobem jest powiedzenie w metodzie "*Ja nie chcę tego wyjątku obsługiwać, daj go wyżej*". Wyżej oznacza do metody,

która wywołuje metodę, w której obecnie się znajdujemy i w której został wyrzucony wyjątek. Oczywiście możemy przekazać informację, o tym, że taki wyjątek jest wyrzucany wyżej, ale w końcu ktoś go musi obsłużyć, czyli:

```
public class StackTraceExample {

    public static void main(String[] args) {
        runAgeChecker(); // tutaj dostaniemy błąd kompilacji
    }

    private static void runAgeChecker() throws MyCheckedException {
        ageChecker(20);
    }

    private static void ageChecker(int age) throws MyCheckedException {
        if (age > 20) {
            System.out.println("OK");
        } else {
            throw new MyCheckedException("Sorry, ale jesteś za młody");
        }
    }
}
```

Metoda `ageChecker()` wyrzuca wyjątek `MyCheckedException`. Zauważ, że po jej liście parametrów pojawił się zapis `throws MyCheckedException`. Jest to informacja dla kogoś, kto będzie chciał tę metodę wywołać, że wyrzuca ona wyjątek `MyCheckedException` i należy go obsłużyć. Albo wykorzystując ten sam mechanizm, podać go wyżej (czyli do metody wywołującej). Metoda `runAgeChecker()` zrobiła to samo, czyli zamiast obsługiwać wyjątek `MyCheckedException`, też uznała, że nie chce tego robić (a co, bo może ☺) i przekazała ten wyjątek wyżej. W przykładzie powyżej mamy błąd kompilacji, bo metoda `main()`, musi albo ten wyjątek obsłużyć, bo to ona wywołuje metodę `runAgeChecker()`, albo przekazać go wyżej w ten sam sposób, przez `throws MyCheckedException`.

```
public class StackTraceExample {

    public static void main(String[] args) throws MyCheckedException {
        // teraz już nie dostaniemy błędu kompilacji bo przekazaliśmy ten wyjątek wyżej
        runAgeChecker();
        // bo wyjątek nie został obsłużony aż do początku działania programu,
        // zatem program jest na tym etapie przerwany
        System.out.println("Ta linijka nigdy się nie wydrukuje");
    }

    private static void runAgeChecker() throws MyCheckedException {
        ageChecker(20);
    }

    private static void ageChecker(int age) throws MyCheckedException {
        if (age > 20) {
            System.out.println("OK");
        } else {
            throw new MyCheckedException("Sorry, ale jesteś za młody");
        }
    }
}
```

W przykładzie powyżej nikt nie obsłużył tego błędu i tak jak zostało napisane w komentarzu, `System.out.println("Ta linijka nigdy się nie wydrukuje");` nie zostanie wydrukowane na ekranie, bo program nigdy nie dojdzie do tego fragmentu kodu, bo przerwie działanie po wywołaniu poprzedniej metody. Stanie się tak, bo metoda `runAgeChecker()` wyrzuca wyjątek, którego `main()` nie obsługuje, tylko przekazuje dalej, zatem program przerywa wywołanie.

Gdybyśmy chcieli doprowadzić do sytuacji, w której `System.out.println("Ta linijka nigdy się nie wydrukuje");` się jednak wydrukuje, moglibyśmy napisać to tak:

```
public class StackTraceExample {

    public static void main(String[] args) {
        try {
            runAgeChecker();
        } catch (MyCheckedException e) {
            System.out.println("Handling exception somehow: " + e.getMessage());
        }
        System.out.println("Ta linijka nigdy się nie wydrukuje");
    }

    private static void runAgeChecker() throws MyCheckedException {
        ageChecker(20);
    }

    private static void ageChecker(int age) throws MyCheckedException {
        if (age > 20) {
            System.out.println("OK");
        } else {
            throw new MyCheckedException("Sorry, ale jesteś za młody");
        }
    }
}
```

Po wykonaniu przykładu powyżej, na ekranie zostanie wydrukowane:

```
Handling exception somehow: Sory, ale jesteś za młody
Ta linijka nigdy się nie wydrukuje
```

Czyli obsługując wyrzucony wyjątek, doprowadziliśmy do sytuacji, w której program wykonuje się dalej. Jednocześnie wydrukowaliśmy na ekranie wiadomość, która została wpisana do wyrzucanego wyjątku.

RuntimeExceptions (Unchecked Exceptions)

Wszystkie pokazane kombinacje alpejskie są wymuszane na nas przez kompilator, gdy działamy na wyjątkach Checked. Czyli tych, które są sprawdzane na etapie kompilacji programu. Jeżeli chcielibyśmy uniknąć pisania w 145 różnych metodach stwierdzenia `throws MyCheckedException` wystarczyłoby zdefiniować `MyCheckedException` jako `RuntimeException`. Czyli wyjątek, który nie jest sprawdzany na etapie kompilacji, możemy go wtedy albo obsłużyć, albo nie.

Jeżeli stworzę teraz kolejny wyjątek:

```
public class MyRuntimeException extends RuntimeException {

    public MyRuntimeException() {
        super("My exception was thrown");
    }

    public MyRuntimeException(String message) {
        super(message);
    }

    public MyRuntimeException(String message, Throwable cause) {
        super(message, cause);
    }
}
```

To mogę wtedy ten sam kod napisać w ten sposób:

```
public class StackTraceExample {

    public static void main(String[] args) {
        runAgeChecker();
        System.out.println("Ta linijka nigdy się nie wydrukuje");
    }

    private static void runAgeChecker() {
        ageChecker(20);
    }

    private static void ageChecker(int age) {
        if (age > 20) {
            System.out.println("OK");
        } else {
            throw new MyRuntimeException("Sorry, ale jesteś za młody");
        }
    }
}
```

Wyrzucając wyjątek `MyRuntimeException`, który jest **unchecked**, nie muszę na etapie kompilacji dodawać do sygnatur metod deklaracji `throws MyRuntimeException`. Mogę, nie jest to błąd kompilacji, ale nie muszę. Plusem tego podejścia jest to, że nie muszę w 132 miejscach pisać tego samego, czyli `throws MyRuntimeException`. Minusem natomiast to, że często łatwo jest zapomnieć, że taki wyjątek może zostać gdzieś głęboko wewnątrz wywołań wyrzucony. Dlatego jest to kwestia filozoficzna i często na różnych projektach są różne podejścia do tematu, najczęściej jednak wykorzystuje się wyjątki **unchecked**.

Blok try-catch-finally

Nie wspomniałem jeszcze, że blok try-catch można rozszerzyć do try-catch-finally (nie mylić z final). Na podstawie przykładu:

```
public class StackTraceExample {

    public static void main(String[] args) {
        try {
            runAgeChecker();
        } catch (Exception e) {
            System.out.println("Handling: " + e.getMessage());
        } finally {
            System.out.println("Ta linijka zawsze się wydrukuje, nieważne czy poleci wyjątek czy nie");
        }
    }

    private static void runAgeChecker() {
        ageChecker(20);
    }

    private static void ageChecker(int age) {
        if (age > 20) {
            System.out.println("OK");
        } else {
            throw new MyRuntimeException("Sorry, ale jesteś za młody");
        }
    }
}
```

Zwróć uwagę na dodanie bloku:

```
finally {
    System.out.println("Ta linijka zawsze się wydrukuje, nieważne czy poleci wyjątek czy nie");
}
```

Blok `finally` robi to, co zostało napisane, nieważne czy wyjątek zostanie wyrzucony, czy nie, blok `finally` wykona się zawsze. Może on być używany, jeżeli potrzebujemy na koniec wywołania bloku `try-catch` zamknąć jakieś zasoby, nieważne czy udało nam się to, co chcieliśmy czy nie. To jest takie bycie kulturalnym, nieważne czy udało Ci się załatwić sprawę, zamknij za sobą drzwi ☺.

Wiedząc już, że mamy i `try` i `catch` i `finally`, warto wspomnieć, że nie wszystkie są wymagane. Dopuszczalne kombinacje to:

```
try {
    // kod rzucający wyjątek
} catch (Exception e) {
    // obsługa wyjątku
}
```

oraz

```
try {
    // kod rzucający wyjątek
} finally { // tutaj nie ma catch
    // obsługa finally
}
```

oraz oczywiście

```
try {
    // kod rzucający wyjątek
} catch (Exception e) {
    // obsługa catch
} finally {
    // obsługa finally
}
```

Muszą być one również w podanej kolejności (`try-catch-finally`), nie można napisać np. `try-finally-catch`.

Na ten moment trzeba też wiedzieć, że nie można napisać samego `try`. W przyszłości się dowiemy, że jest jeden przypadek, gdzie można, ale na teraz przyjmijmy, że nie można.

Multi catch

W po jednym bloku `try` możemy napisać kilka bloków `catch`. Nie możemy napisać pod sobą kilku `try` i nie możemy napisać kilku `finally`. Tylko `catch` może wystąpić więcej niż raz.

```
public class StackTraceExample {

    public static void main(String[] args) {
        try {
            runAgeChecker();
        } catch (MyCheckedException tymRazemNazweZmiennaInaczej) {
            System.out.println("Here I'm handling MyCheckedException: "
                + tymRazemNazweZmiennaInaczej.getMessage());
        } catch (Exception toTezMozeSieNazywacJakChce) {
            System.out.println("Here I'm handling Exception: "
                + toTezMozeSieNazywacJakChce.getMessage());
        }
        System.out.println("Ta linijka nigdy się nie wydrukuje");
    }

    private static void runAgeChecker() throws MyCheckedException {
        ageChecker(20);
    }

    private static void ageChecker(int age) throws MyCheckedException {
        if (age > 20) {
            System.out.println("OK");
        } else {
            throw new MyCheckedException("Sorry, ale jesteś za młody");
        }
    }
}
```

Sytuacja taka jest wykorzystywana wtedy, gdy chcemy w inny sposób obsłużyć różne rodzaje wyjątków. Przykładowo, jeżeli łączymy się do jakiegoś zewnętrznego serwera i serwer kompletnie nie działa, możemy dostać wyjątek `ServiceUnavailableException` i w jego przypadku chcemy, aby program przestał działać, bo dalej kompletnie nie ma sensu wykonywać naszego programu. Może też wystąpić taka

sytuacja, że serwer wyrzuci nam wyjątek `TimeoutException` (timeout występuje wtedy, jeżeli serwer nie wyrobił się z odpowiedzią w założonym przez nas czasie, np. 5 sekund to wtedy my przerywamy czekanie na odpowiedź i mamy wyjątek `TimeoutException`). My możemy wtedy chcieć obsłużyć ten wyjątek inaczej i ponowić próbę połączenia z serwerem, bo przecież działa, tylko się nie wyrobił, więc spróbujmy jeszcze raz. Do takich sytuacji możemy wykorzystać **multi-catch**, czyli w zależności od rodzaju wyjątku, który został wyrzucony, robimy co innego.

Ważne jest natomiast aby pamiętać, że w przypadku stosowania **multi-catch** istnieje pewna hierarchia, w której możemy te wyjątki łapać. W przykładzie wyżej, najpierw łapiemy wyjątek bardziej szczegółowy (subklasę), a później bardziej ogólny (superklasę). Na tym ta zasada polega, że nie możemy zrobić odwrotnie. Bo gdybyśmy najpierw złapali wyjątek jako `Exception`, to nie byłoby takiej możliwości, żeby złapać wyjątek określony jako `MyCheckedException`, bo zostałby on już wyłapany wcześniej. Czyli tak nie wolno:

```
public static void main(String[] args) {
    try {
        runAgeChecker();
    } catch (Exception tenTezMozeSieNazywacJakChce) {
        System.out.println("Here I'm handling Exception: "
            + tenTezMozeSieNazywacJakChce.getMessage());
    // tu dostaniemy błąd kompilacji "Exception 'MyCheckedException' has already been caught"
    } catch (MyCheckedException tymRazemNazweTenWyjatekInaczej) {
        System.out.println("Here I'm handling MyCheckedException: "
            + tymRazemNazweTenWyjatekInaczej.getMessage());
    }
    System.out.println("Ta linijka nigdy się nie wydrukuje");
}
```

Rethrowing exceptions

Zabawa dalej polega na tym, że mamy możliwość podczas obsługiwania wyjątku wyrzucić wyjątek. To samo możemy zrobić w `finally`, czyli w trakcie wywoływania `finally` również możemy wyrzucić wyjątek, jeżeli pojawia się sytuacja, która tego wymaga. Przykład:

```
public class MyAnotherException extends RuntimeException {

    public MyAnotherException() {
        super("My exception was thrown");
    }

    public MyAnotherException(String message) {
        super(message);
    }

    public MyAnotherException(String message, Throwable cause) {
        super(message, cause);
    }
}
// gdzieś w kodzie
public static void main(String[] args) {
    try {
        runAgeChecker();
    } catch (MyCheckedException tymRazemNazweTenWyjatekInaczej) {
        throw new MyAnotherException("Wrapping previous exception", tymRazemNazweTenWyjatekInaczej);
    } catch (Exception tenTezMozeSieNazywacJakChce) {
        System.out.println("Here I'm handling Exception: " + tenTezMozeSieNazywacJakChce.getMessage());
    }
    System.out.println("Ta linijka nigdy się nie wydrukuje");
}
```

W pierwszym bloku catch wrapujemy wyrzucony wyjątek w kolejny wyjątek `MyAnotherException`. Teraz przywołuję komentarz z początku notatek o wyjątkach, gdzie pokazywałem trzeci konstruktor, który przyjmował `Throwable`. To jest właśnie ten przypadek, gdzie to się robi potrzebne. Czyli możemy stworzyć nowy wyjątek i go wyrzucić, przekazując do niego poprzedni wyjątek. Jeżeli uruchomimy ten program, na ekranie zostanie wydrukowany taki **StackTrace**:

Pierwszy bez komentarzy

```
Exception in thread "main" pl.zajavka.MyAnotherException: Wrapping previous exception
    at pl.zajavka.StackTraceExample.main(StackTraceExample.java:11)
Caused by: pl.zajavka.MyCheckedException: Sory, ale jesteś za młody
    at pl.zajavka.StackTraceExample.ageChecker(StackTraceExample.java:26)
    at pl.zajavka.StackTraceExample.runAgeChecker(StackTraceExample.java:19)
    at pl.zajavka.StackTraceExample.main(StackTraceExample.java:9)
```

Drugi z komentarzami

```
// wiadomość przekazana podczas tworzenia wyjątku MyAnotherException
Exception in thread "main" pl.zajavka.MyAnotherException: Wrapping previous exception
    // nazwa metody i linijka w kodzie, gdzie MyAnotherException został wyrzucony
    at pl.zajavka.StackTraceExample.main(StackTraceExample.java:11)
// wyjątek, który został przekazany do MyAnotherException wraz z jego własnym StackTrace
Caused by: pl.zajavka.MyCheckedException: Sory, ale jesteś za młody
    at pl.zajavka.StackTraceExample.ageChecker(StackTraceExample.java:26)
    at pl.zajavka.StackTraceExample.runAgeChecker(StackTraceExample.java:19)
    at pl.zajavka.StackTraceExample.main(StackTraceExample.java:9)
```

Tak jak pokazane zostało to w przykładzie, można te poziomy zagnieżdżać nawet parę razy, czyli opakowywać wyjątki w wyjątki i w praktyce często się to robi. Wtedy w naszym **StackTrace**, na jego

górze będzie wyjątek najbardziej opakowujący, a na samym dole tzw. root cause. Czyli pierwotna przyczyna wyrzucenia wyjątku.

Popularne wyjątki, z jakimi możemy się spotkać

Wyjątki typu `RuntimeException`:

- **NullPointerException** - kto próbuje wykonywać metodę na referencji będącej `null`em ten wie ☺. Zostaje wyrzucony, jak próbujemy wykonać metodę, na referencji, która jest `null`em,
- **NumberFormatException** - zostaje wyrzucony przykładowo w takiej sytuacji `Long.parseLong("a");`, czyli gdy chcemy stworzyć numer ze Stringa, który nie jest numerem,
- **IllegalArgumentException** - gdy do metody przekażemy nieprawidłowy argument,
- **ClassCastException** - gdy spróbujemy rzutować referencję na obiekt, na który może się to udać, bo oba z nich mają tego samego rodzica, ale w praktyce pod referencją kryje się inna implementacja, przykładowo:

```
Animal animal = new Dog(); // zarówno Dog i Monkey dziedziczą z Animal
Monkey monkey = (Monkey) animal; // tutaj dostaniemy ClassCastException
```

- **ArrayIndexOutOfBoundsException** - gdy staramy się odwołać do miejsca, w tablicy, które w niej nie istnieje.

Podam również często spotykane wyjątki typu *Checked Exception*, natomiast na ten moment ich nie dotykaliśmy, gdyż dotyczą one operacji na plikach. O nich będzie w przyszłości:

- **IOException** - generalny problem z odczytem lub zapisem do pliku na dysku,
- **FileNotFoundException** - staramy się odwołać do pliku na dysku, który nie istnieje.

I mamy jeszcze przecież **Errory**:

- **StackOverflowError** - Error wyrzucany przez JVM, gdy ta metoda zaczyna wywoływać samą siebie i nigdy nie przerywa. Następuje wtedy przepełnienie Stosu. Czym jest Stos, dowiemy się z filmu o modelu pamięci w Javie.
- **OutOfMemoryError** - Error wyrzucany przez JVM, gdy skończy nam się pamięć i JVM nie jest w stanie zarezerwować miejsca na utworzenie kolejnego obiektu.