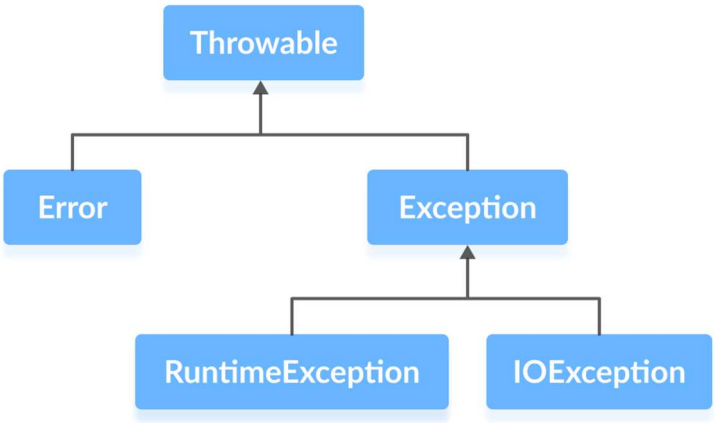
PROGRAMOWANIE OBIEKTOWE JAVA – LABORATORIUM

WYJĄTKI

Wyjątek (ang. exception) jest specjalną klasą. Jest ona specyficzna ponieważ w swoim łańcuchu dziedziczenia ma klasę java.lang.Throwable. Instancje, które w swojej hierarchii dziedziczenia mają tę klasę mogą zostać „rzucone" (ang. throw) przerywając standardowe wykonanie programu.

Hierarchia wyjątków w Javie



Klasa Throwable jest klasą główną w hierarchii i dzieli się na Error oraz Exception.

Error - reprezentuje nieodwracalne warunki, np. brak pamięci, wyciek pamięci, błędy przepełnienia stosu, niezgodność bibliotek, nieskończona rekursja itp.

Exception - mogą być przechwytywane i obsługiwane przez program. Gdy w metodzie występuje wyjątek, tworzymy obiekt i jest on nazywany obiektem wyjątku. Zawiera informacje o wyjątku, takie jak nazwa i opis wyjątku oraz stan programu, w którym wystąpił wyjątek.

Przykładem może być walidacja argumentów metody. Załóżmy, że nasza metoda jako argument przyjmuje liczbę godzin i zwraca liczbę sekund, odpowiadających przekazanemu argumentowi. Możemy założyć, że akceptujemy wyłącznie argumenty dodatnie lub 0. Innymi słowy jeśli metoda zostanie wywołana z argumentem mniejszym od 0 możemy uznać to za nieprawidłowe wywołanie i zasygnalizować taką sytuację rzucając wyjątek.

```
package PO_UR.Lab08;

public class ExceptionExamle01 {
    public int getNumberOfSeconds(int hour) {
        if (hour < 0) {
            throw new IllegalArgumentException("Hour must be >= 0: " +
hour);
        }
        return hour * 60 * 60;
    }
}
```

Powyżej użyto wyjątku występującego w standardowej bibliotece języka Java: **java.lang.IllegalArgumentException**. Do rzucania wyjątku używam się słowa kluczowego **throw**.

Rzucenie wyjątku i co dalej?

Przeanalizujemy poniższy przykład:

```java
package PO_UR.Lab08;


public class StackTraceExample {
    public static void main(String[] args) {
        StackTraceExample example = new StackTraceExample();
        example.method1();
    }

    public void method1() {
        method2();
    }

    public void method2() {
        method3();
    }

    public void method3() {
        throw new RuntimeException("BUM! BUM! BUM!");
    }
}
```

W metodzie main tworzymy instancję klasy StackTraceExample i na instancji wywołujemy metodę method1, metoda ta wywołuje z kolei metodę method2. method2 wywołuje method3, która rzuca wyjątek java.lang.RuntimeException (kolejny wyjątek z biblioteki standardowej). Tą listę metod wywołujących siebie nawzajem nazywamy stosem wywołań. W naszym przypadku stos wygląda następująco:

1. main
2. method1
3. method2
4. method3

Po uruchomieniu tego programu zostanie rzucony wyjątek, a programista zobaczy stos wywołań metod (ang. stacktrace), jak w przykładzie poniżej:

```
C:\Users\ezesl\.jdks\openjdk-17.0.1\bin\java.exe "-javaagent:C:\Program Files\JetBrains\IntelliJ IDEA
Community Edition 2021.3\lib\idea_rt.jar=63657:C:\Program Files\JetBrains\IntelliJ IDEA Community
Edition 2021.3\bin" -Dfile.encoding=UTF-8 -classpath
C:\Users\ezesl\IdeaProjects\EZ_example\out\production\untitled104 PO_UR.Lab08.StackTraceExample

Exception in thread "main" java.lang.RuntimeException: BUM! BUM! BUM!

        at PO_UR.Lab08.StackTraceExample.method3(StackTraceExample.java:19)

        at PO_UR.Lab08.StackTraceExample.method2(StackTraceExample.java:15)

        at PO_UR.Lab08.StackTraceExample.method1(StackTraceExample.java:11)

        at PO_UR.Lab08.StackTraceExample.main(StackTraceExample.java:7)

Process finished with exit code 1
```

W pracy programisty umiejętność czytania tego typu komunikatów jest bardzo istotna. Stacktrace to nic innego jak odwrócony stos wywołań metod od rozpoczęcia programu do miejsca w którym został rzucony wyjątek.

Pierwsza linijka mówi o tym jaki wyjątek został rzucony, kolejne linijki to metody, które były wywoływane. Każda linia składa się z nazwy klasy wraz z pakietem, w nawiasach znajduje się nazwa pliku oddzielona dwukropkiem od numeru linii w tym pliku. W naszym przypadku wyjątek RuntimeException został rzucony po wywołaniu metody p at PO_UR.Lab08.StackTraceExample.method3(StackTraceExample.java:19), która znajduje się w 19 linijce pliku StackTraceExample.java.

OBSŁUGA WYJĄTKÓW

Wyjątek jest obsługiwany, jeśli reagujemy na jego wystąpienie i próbujemy "naprawić" program w trakcie jego działania. Możemy też powiedzieć, że łapiemy wyjątek. Do obsługi wyjątków służy blok try/catch. Proszę spójrz na przykład poniżej:

```
try {
    // code
}
catch(Exception e) {
    // code
}
```

Przykład 1.

```
package PO_UR.Lab08;

public class ExceptionExample02 {
    public static void main(String[] args) {
        try {
            // code that generate exception
            int divideByZero = 5 / 0;
            System.out.println("Rest of code in try block");
        }

        catch (ArithmeticException e) {
            System.out.println("ArithmeticException => " + e.getMessage());
        }
    }
}
```

W powyższym przykładzie próbujemy podzielić liczbę przez 0, aby obsłużyć wyjątek kod dzielenia przez 0 umieszczamy w bloku try. W przypadku wystąpienia wyjątków pozostała cześć w bloku try jest pomijalna. Blok catch przechwytuje wyjątek i wykonuje zawarte w nim istrukcję. W przypadku gdy żadna z instrukcji w bloku try nie generuje wyjątku, blok catch jest pomijalny.

JAVA FINALLY BLOCK

W javie ostatni blok jest zawsze wykonywany bez względu na to, czy istnieje wyjątek, czy nie. Jest on opcjonalny i dla każdego bloku try może występować tylko jeden blok finalny. Podstawowa składania:

```
try {
    //code
}
catch (ExceptionType1 e1) {
    // catch block
}
finally {
    // finally block always executes
}
```

Przykład 2

```
package PO_UR.Lab08;

public class ExceptionExample02 {
    public static void main(String[] args) {
        try {
            // code that generate exception
            int divideByZero = 5 / 0;
            System.out.println("Rest of code in try block");
        }

        catch (ArithmeticException e) {
```

```
            System.out.println("ArithmeticException => " + e.getMessage());
        }
        finally {
            System.out.println("This is the finally block");
        }
    }
}
```

W powyższym przykładzie dzielimy liczbę przez 0 w bloku try. Tutaj ten kod generuje ArithmeticException. Wyjątek jest przechwytywany przez blok catch. I wtedy wykonywany jest ostatni blok.

JAVA THROW AND THROWS

Wyróżniamy dwa typy wyjątków Unchecked Exceptions i Checked Exceptions.

- **Unchecked Exceptions:** wyjątki, które nie są sprawdzane w czasie kompilacji, ale w czasie wykonywania. Do grupy tej zaliczamy m.in. wyjątki: ArithmeticException, NullPointerException, ArrayIndexOutOfBoundsException, exceptions z klasy Error, etc.
- **Checked Exceptions:** wyjątki sprawdzane w czasie kompilacji, np.: IOException, InterruptedException, etc.

Najpopularniejsze wyjątki i ich zastosowanie

Poniżej przedstawiono kilka najpopularniejszych wyjątków oraz bardzo krótki opis kiedy są używane.

- NullPointerException — rzucany kiedy próbujesz wywołać metodę na zmiennej, której wartość to null
- IllegalArgumentException — rzucany, kiedy przekazywany argument jest z jakiegoś powodu nieprawidłowy (walidacja wewnątrz metod)
- IOException (wyjątki po nim dziedziczące) — rzucany w przypadku problemów z systemem wejścia/wyjścia, czyli najogólniej rzecz ujmując, kiedy wystąpi problem przy pracy z plikami lub z transmisją danych za pośrednictwem Internetu
- NumberFormatException — rzucany, kiedy próbujemy zamienić na liczbę np. obiekt typu String, który zawiera nie tylko cyfry
- IndexOutOfBoundException — rzucany, kiedy próbujemy się odwołać do nieistniejącego elementu tablicy lub listy

DOBRE PRAKTYKI PRZY UŻYWANIU WYJĄTKÓW

- Pierwsza i najważniejsza zasada, blok try powinien być jak najmniejszy. Takie podejście bardzo ułatwia znajdowanie błędów w bardziej skomplikowanych programach. Dzięki małemu blokowi try także możemy napisać lepszy kod do obsługi wyjątku – wiemy dokładnie z którego miejsca wyjątek może zostać rzucony więc wiemy także jak najlepiej na niego zareagować.
- Blok finally bardzo często jest niezbędny. Szczególnie jeśli operujemy na instancjach, które wymagają “zamknięcia”.
- Używaj klas wyjątków, które idealnie pasują do danej sytuacji. Jeśli nie ma takiego wyjątku w bibliotece standardowej utwórz własną klasę wyjątku.
- Tworząc instancję wyjątków podawaj możliwie najdokładniejszy opis w treści wyjątku. Pozwala to na dużo łatwiejsze znajdowanie błędów w programie jeśli komunikat wyjątku jest szczegółowy.
- Nie zapominaj o używaniu wyjątków typu checked. Chociaż wymagają trochę więcej kodu i generują często irytujące błędy kompilacji ich używanie jest czasami wskazane.

Zadania do samodzielnego rozwiązania:

Zadanie 1.

Napisz program, który pobierze od użytkownika liczbę i wyświetli jej pierwiastek. Do obliczenia pierwiastka możesz użyć istniejącej metody java.lang.Math.sqrt(). Jeśli użytkownik poda liczbę ujemną rzuć wyjątek java.lang.IllegalArgumentException. Obsłuż sytuację, w której użytkownik poda ciąg znaków, który nie jest liczbą.

Zadanie 2.

Napisz metodę, która będzie zwracać silnię podanej jako argument liczby. Metoda powinna rzucać wyjątek rodzaju Checked zdefiniowanego przez Ciebie typu BlednaWartoscDlaSilniException, gdy jej argument będzie ujemny. Skorzystaj z tej metody w main, obsługując potencjalny wyjątek.

Zadanie 3.

Napisz program z klasą Adres, która będzie miała podane poniżej pola, które będą ustawiane w konstruktorze klasy Adres. Konstruktor powinien sprawdzić wszystkie podane wartości i rzucić wyjątek NieprawidlowyAdresException rodzaju Checked, jeżeli któraś z wartości będzie nieprawidłowa. Uwaga: komunikat rzucanego wyjątku powinien zawierać informację o wszystkich nieprawidłowych wartościach przekazanych do konstruktora – dla przykładu, jeżeli ulica i miasto będą miały wartość null, to komunikat wyjątku powinien być następujący: "Ulica nie może być nullem. Miasto nie może być nullem". Pola klasy: String ulica – wartość nieprawidłowa to null, int numerDomu – wartość nieprawidłowa to liczba <= 0, String kodPocztowy – wartość nieprawidłowa to null, String miasto – wartość nieprawidłowa to null.