Podstawy Java – dzień 3

v3.1

Plan

- Pakiety, importy
- Wyjątki

Pakiety, importy

Pakiety

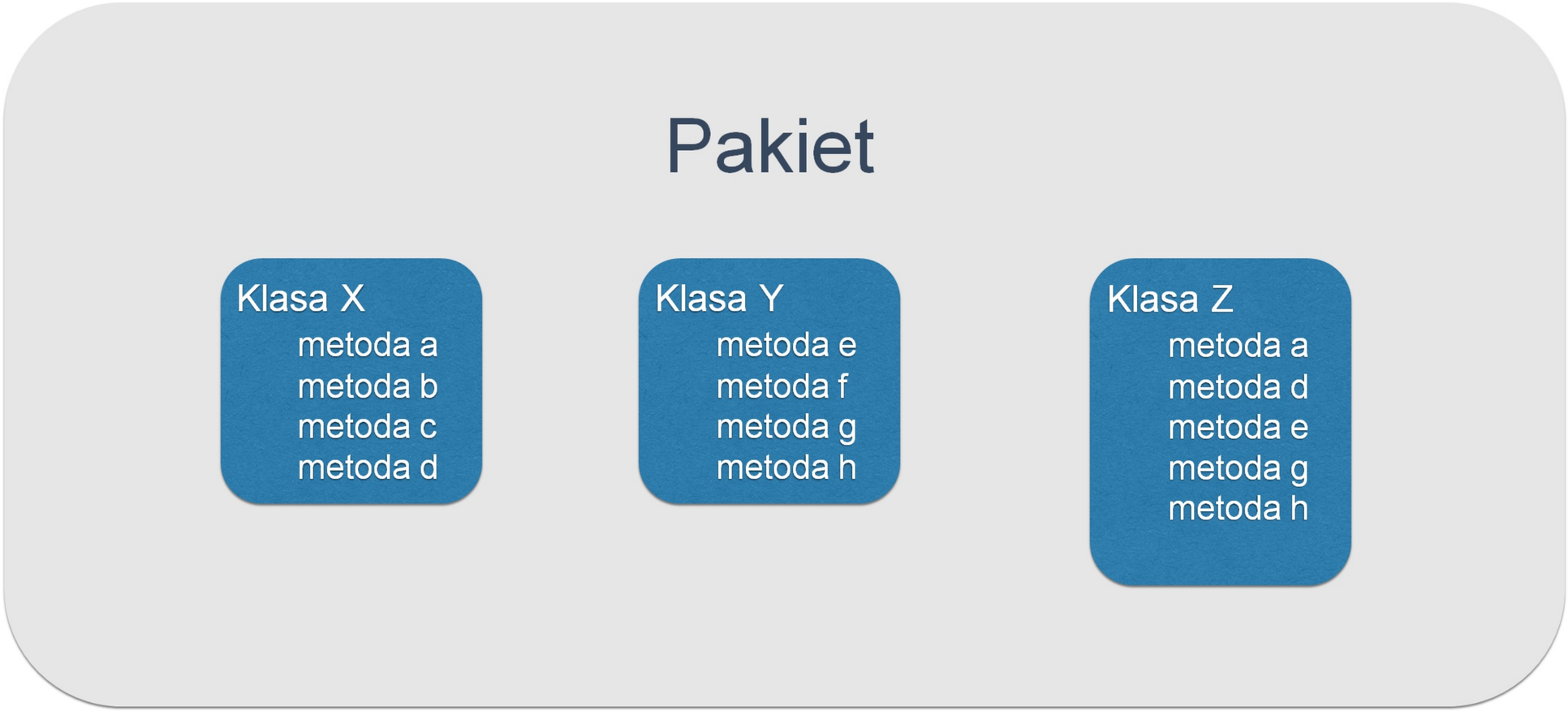
Grupowanie klas w pakiety pozwala na:

- podzielenie/ułożenie/gromadzenie ich wg znaczenia,
- uniknięcie konfliktów nazw.

Jeśli nie zadeklarujemy pakietu, klasy zostają automatycznie umieszczone w pakiecie domyślnym.

Pakiety są elementem grupującym wiele klas odpowiadających za pewną część aplikacji lub biblioteki.

Pakiet


Pakiet
Klasa X
metoda a
metoda b
metoda c
metoda d
Klasa Y
metoda e
metoda f
metoda g
metoda h
Klasa Z
metoda a
metoda d
metoda e
metoda g
metoda h

Pakiety

- Pakiety posiadają strukturę hierarchiczną (każdy z pakietów może zawierać dowolną liczbę innych katalogów oraz klas).
- Nazwy pakietów zwyczajowo piszemy małą literą, oddzielając kolejne elementy nazwy znakiem kropki (.).
- Pakiety stanowią na dysku strukturę zagnieżdżonych w sobie katalogów.

Pakiet określamy umieszczając w pierwszej linii naszej klasy instrukcję:

```
package {nazwa pakietu};
```

Przykład:

```
package pl.coderslab;
```

Pakiety

Zaleca się nazywać pakiety wg następującej konwencji:

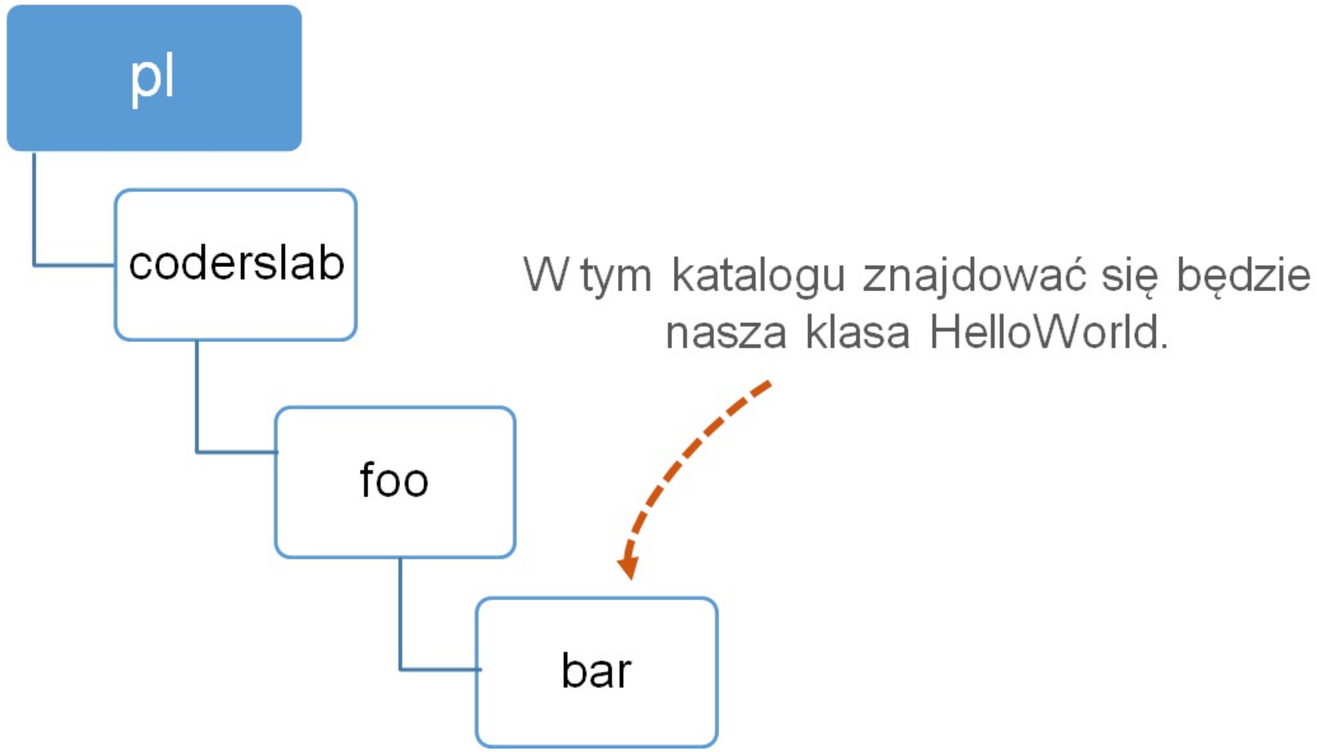
[odwrócona nazwa domeny].[nazwa aplikacji].[element aplikacji]

```
package pl.coderslab.foo.bar;

public class HelloWorld {
    public static void main (String[] args) {
    }
}
```

Pakiety

Dla powyższego przykładu zostanie utworzona na dysku komputera następująca struktura folderów:



Pakiet java.lang

Wszystkie klasy z pakietu **`java.lang`** są automatycznie importowane do każdego pliku naszego programu.

Zawiera on definicje podstawowych klas języka Java, np.:

```
java.lang.Math;
java.lang.String;
java.lang.StringBuilder;
java.lang.Integer;
java.lang.System;
```

Część z nich będziemy omawiać na kursie.

Pełna, **kwalifikowana** nazwa (uwzględniająca pakiet), zapisywana jest następująco:

```
java.lang.System.out.println("Hello");
```

Importy

Jeżeli zachodzi potrzeba użycia klas znajdujących się w innych pakietach, można to zrobić na dwa sposoby:

- podając nazwę klasy, poprzedzoną nazwą pakietu w jakim klasa się znajduje (tzw. nazwa kwalifikowana):

```
java.util.Scanner scan = new java.util.Scanner(System.in);
```

- wykorzystując instrukcję importu:

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

w następujący sposób:

```
Scanner scanner = new Scanner(System.in);
```

Importy

Deklaracja importu pozwala na używanie krótkich nazw klas podczas tworzenia obiektów:

```
package pl.coderslab.foo.bar;

import java.util.Scanner;

public class ScannerCoder {
    public static void main (String[] args) {
        Scanner scan = new Scanner(System.in);
    }
}
```

Importy

Deklaracja importu pozwala na używanie krótkich nazw klas podczas tworzenia obiektów:

```
package pl.coderslab.foo.bar;

import java.util.Scanner;

public class ScannerCoder {
    public static void main (String[] args) {
        Scanner scan = new Scanner(System.in);
    }
}
```

Instrukcję importu umieszczamy zaraz za instrukcją **package** (o ile ona istnieje), ale przed definicją klasy.

Importy

Deklaracja importu pozwala na używanie krótkich nazw klas podczas tworzenia obiektów:

```
package pl.coderslab.foo.bar;

import java.util.Scanner;

public class ScannerCoder {
    public static void main (String[] args) {
        Scanner scan = new Scanner(System.in);
    }
}
```

Instrukcję importu umieszczamy zaraz za instrukcją **`package`** (o ile ona istnieje), ale przed definicją klasy.

Dzięki instrukcji importu możemy posługiwać się uproszczoną nazwą klasy **`Scanner`**.

Importy

Można importować dowolną liczbę klas:

```
import java.io.File;
import java.io.FileReader;
import java.io.FileWriter;
import java.io.InputStream;
import java.io.PrintStream;
import java.io.PrintWriter;
import java.io.StringReader;
import java.io.StringWriter;
```

Jeśli chcemy zimportować wszystkie klasy z danego pakietu, możemy posłużyć się symbolem gwiazdki "*":

```
import java.io.*;
```

Importowanie klas nie oznacza dodawania ich do naszego programu – nie zwiększamy więc jego rozmiaru.

Import statyczny

Instrukcja **`import static`** pozwala na import statycznych pól i metod z wybranej klasy wg poniższego schematu:

```
import static pakiet.Klasa.nazwa;
```

Podobnie jak podczas importowania klas, możemy użyć symbolu ("*").

```
import static pakiet.Klasa.*;
```

Import statyczny

```
package pl.coderslab.foo.bar;
import static java.lang.System.out;
import static java.lang.Math.PI;
public class ScannerCoder {
    public static void main(String[] args) {
        out.println("text");
        out.println(PI);
    }
}
```

Import statyczny

```
package pl.coderslab.foo.bar;
import static java.lang.System.out;
import static java.lang.Math.PI;
public class ScannerCoder {
    public static void main(String[] args) {
        out.println("text");
        out.println(PI);
    }
}
```

Taka forma importu pozwala nam posługiwać się skróconym zapisem:
`out.println` zamiast **`System.out.println`**
`PI` zamiast **`Math.PI`**

Zadania

Wyjątki

Obsługa błędów – wyjątki

Podczas działania programu zdarzyć się może sytuacja niepożądana lub taka, której nie przewidzieliśmy, np. próba pobrania nieistniejącego elementu, wywołanie metody na nieistniejącym obiekcie, brak pliku itd.

W języku Java do informowania o błędach służą klasy, które są pochodnymi klasy **`Throwable`**, pozwalające na zgłoszenie i obsłużenie **wyjątku**.

Wyjątek – jest to informacja o tym, że pojawił się błąd podczas działania programu.

Potocznie mówimy, że wyjątek został **rzucony** lub **zwrócony**.

Dzieje się tak wtedy, gdy program w trakcie swojego działania natrafia na wyjątkową sytuację, którą komunikuje określonym wyjątkiem (w zależności od tego jaki problem wystąpił).

Obsługa błędów – wyjątki

Do obsługi wyjątków w naszym programie należy użyć konstrukcji:
`try–catch–finally`

Możemy również przenieść obsługę wyjątku w inne miejsce za pomocą instrukcji **`throws`**.

Przyczyn wystąpienia wyjątków może być bardzo wiele.
Przykładowe wyjątki:

- **`ArrayIndexOutOfBoundsException`** – podczas pobierania elementu tablicy z nieistniejącego indeksu,
- **`ArithmeticException`** – przy próbie dzielenia przez 0,
- **`NullPointerException`** – podczas odwołania się do obiektu, którego referencja wskazuje na null.

Instrukcja try–catch

Obsługa błędu w miejscu jego wystąpienia:

```
try {
    // kod programu
}
catch (ExceptionType e) {
    // kod obsługi wyjątku
}
```

Instrukcja try–catch

Obsługa błędu w miejscu jego wystąpienia:

```
try {
    // kod programu
}
catch (ExceptionType e) {
    // kod obsługi wyjątku
}
```

Fragment kodu, w którym może wystąpić potencjalny wyjątek.

Instrukcja try–catch

Obsługa błędu w miejscu jego wystąpienia:

```
try {
    // kod programu
}
catch (ExceptionType e) {
    // kod obsługi wyjątku
}
```

Fragment kodu, w którym może wystąpić potencjalny wyjątek.

Jeżeli wystąpi wyjątek typu **`ExceptionType`**, wywołany zostanie kod obsługi wyjątku, natomiast w zmiennej **`e`** będziemy mieli obiekt tego wyjątku.

Instrukcja try–catch–finally

Rozszerzona forma instrukcji:

```
try {
    // kod programu
}
catch (ExceptionType e) {
}
catch (ExceptionType1 e1) {
}
finally {
}
```

Instrukcja try–catch–finally

Rozszerzona forma instrukcji:

```
try {
    // kod programu
}
catch (ExceptionType e) {
}
catch (ExceptionType1 e1) {
}
finally {
}
```

Możliwe jest sprawdzanie wystąpienia wielu wyjątków na raz.

Instrukcja try–catch–finally

Rozszerzona forma instrukcji:

```
try {
    // kod programu
}
catch (ExceptionType e) {
}
catch (ExceptionType1 e1) {
}
finally {
}
```

Możliwe jest sprawdzanie wystąpienia wielu wyjątków na raz.

Blok **`finally`** jest opcjonalny – kod, który w nim umieścimy wykona się zawsze.

Instrukcja try–catch

Jeśli chcemy obsłużyć kilka wyjątków w taki sam sposób, to zamiast pisać kilka bloków **`catch`**, możemy to również zrobić wg poniższego schematu:

```
try {
      // kod programu
    }
    catch(ExceptionType | ExceptionType1 e) {
      // obsługa błędów
    }
```

Instrukcja try–catch – obsługa wyjątku

Przykład 1

Wyjątek **`ArithmeticException`**:

```
int a = 1, b = 0, c;
try {
    c = a / b;
} catch (ArithmeticException e) {
    System.out.println("Nie dziel przez zero!!");
}
```

Instrukcja try–catch – obsługa wyjątku

Przykład 1

Wyjątek **`ArithmeticException`**:

```
int a = 1, b = 0, c;
try {
    c = a / b;
} catch (ArithmeticException e) {
    System.out.println("Nie dziel przez zero!!");
}
```

Jeżeli wykryta zostanie próba dzielenia przez zero, program nie zakończy swojego działania, tylko przekaże sterowanie do bloku **`catch`**, a następnie wyświetli umieszczony tam komunikat.

Instrukcja try–catch

Przykład 2

Wyjątek **`ArrayIndexOutOfBoundsException`**:

```
int tab[] = { 1, 2, 3, 4, 5 };
try {
    System.out.println(tab[1]);
    System.out.println(tab[5]);
} catch (ArrayIndexOutOfBoundsException e) {
    System.out.println("Niepoprawny indeks");
}
```

Instrukcja try–catch

Przykład 2

Wyjątek **`ArrayIndexOutOfBoundsException`**:

```
int tab[] = { 1, 2, 3, 4, 5 };
try {
    System.out.println(tab[1]);
    System.out.println(tab[5]);
} catch (ArrayIndexOutOfBoundsException e) {
    System.out.println("Niepoprawny indeks");
}
```

Jak już doskonale zdajemy sobie sprawę, taki indeks nie istnieje w tablicy.

Instrukcja try–catch

Przykład 2

Wyjątek **`ArrayIndexOutOfBoundsException`**:

```
int tab[] = { 1, 2, 3, 4, 5 };
try {
    System.out.println(tab[1]);
    System.out.println(tab[5]);
} catch (ArrayIndexOutOfBoundsException e) {
    System.out.println("Niepoprawny indeks");
}
```

Jak już doskonale zdajemy sobie sprawę, taki indeks nie istnieje w tablicy.

Odwołanie do nieistniejącego elementu tablicy powoduje wystąpienie wyjątku **`ArrayIndexOutOfBoundsException`**.

Wyjątki są obiektami

Klasy wyjątków posiadają kilka przydatnych metod informowania o błędzie. W poniższym przykładzie próbujemy przekształcić napis, który nie jest liczbą, na typ **`int`**. Jest to niemożliwe, zatem zostanie zwrócony wyjątek.

```
try {
    int num = Integer.parseInt("abc");
} catch (NumberFormatException e) {
    System.out.println(e.getMessage());
    System.out.println(e);
    e.printStackTrace();
}
```

Wyjątki są obiektami

Klasy wyjątków posiadają kilka przydatnych metod informowania o błędzie. W poniższym przykładzie próbujemy przekształcić napis, który nie jest liczbą, na typ **`int`**. Jest to niemożliwe, zatem zostanie zwrócony wyjątek.

```
try {
    int num = Integer.parseInt("abc");
} catch (NumberFormatException e) {
    System.out.println(e.getMessage());
    System.out.println(e);
    e.printStackTrace();
}
```

Zmienna **`e`** – to obiekt klasy **`NumberFormatException`**.

Wyjątki są obiektami

Klasy wyjątków posiadają kilka przydatnych metod informowania o błędzie. W poniższym przykładzie próbujemy przekształcić napis, który nie jest liczbą, na typ **`int`**. Jest to niemożliwe, zatem zostanie zwrócony wyjątek.

```
try {
    int num = Integer.parseInt("abc");
} catch (NumberFormatException e) {
    System.out.println(e.getMessage());
    System.out.println(e);
    e.printStackTrace();
}
```

Zmienna **`e`** – to obiekt klasy **`NumberFormatException`**.
Zwraca komunikat wyjątku.

Wyjątki są obiektami

Klasy wyjątków posiadają kilka przydatnych metod informowania o błędzie. W poniższym przykładzie próbujemy przekształcić napis, który nie jest liczbą, na typ **`int`**. Jest to niemożliwe, zatem zostanie zwrócony wyjątek.

```
try {
    int num = Integer.parseInt("abc");
} catch (NumberFormatException e) {
    System.out.println(e.getMessage());
    System.out.println(e);
    e.printStackTrace();
}
```

Zmienna **`e`** – to obiekt klasy **`NumberFormatException`**.
Zwraca komunikat wyjątku.
Zwraca informacje o wyjątku – jego typ i komunikat wyjątku.

Wyjątki są obiektami

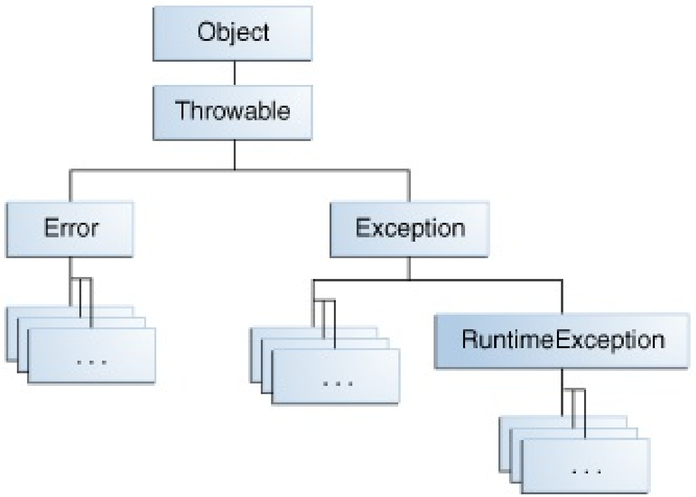
Klasy wyjątków posiadają kilka przydatnych metod informowania o błędzie. W poniższym przykładzie próbujemy przekształcić napis, który nie jest liczbą, na typ **`int`**. Jest to niemożliwe, zatem zostanie zwrócony wyjątek.

```
try {
    int num = Integer.parseInt("abc");
} catch (NumberFormatException e) {
    System.out.println(e.getMessage());
    System.out.println(e);
    e.printStackTrace();
}
```

Zmienna **`e`** – to obiekt klasy **`NumberFormatException`**.
Zwraca komunikat wyjątku.
Zwraca informacje o wyjątku – jego typ i komunikat wyjątku.
Wyświetla pełną informację o metodach, których wywołanie spowodowało wystąpienie wyjątku.

Hierarchia klas wyjątków

Klasy wyjątków tworzą rozbudowaną hierarchię.



Error: odpowiada za błędy zgłaszane przez JVM, np. błąd kompilacji.

`RuntimeException:` tutaj znajdują się m.in. poznane już przez nas wyjątki:

- **`NullPointerException`**
- **`ArrayIndexOutOfBoundsException`**

Więcej o strukturze klas Java dowiemy się omawiając zagadnienia programowania obiektowego.

Podział wyjątków

Klasy wyjątków możemy również podzielić na:

- **wyjątki niekontrolowane** – możemy ale nie musimy ich obsługiwać w programie (są to klasy **`Error`** i **`RuntimeException`**, jak również wszystkie ich podklasy).
- **wyjątki kontrolowane** – to wyjątki, których obsługę musimy zapewnić w naszym programie.

Większość wyjątków, które będziemy obsługiwać to pochodne klasy **`Exception`** – ich listę znajdziemy pod adresem:

https://docs.oracle.com/javase/8/docs/api/java/lang/Exception.html

Instrukcja throws

Jeżeli nie chcemy obsługiwać wyjątku w danej metodzie, można użyć w jej nagłówku klauzuli **throws**, po której następuje nazwa wyjątków.

Instrukcja **throws** oznacza przesunięcie obsługi wyjątku do kodu wywołującego.

```
[deklaracja metody] throws [lista wyjątków] {
        //...
}
```

Instrukcja throws

Tworząc metodę definiujemy, że może ona zwrócić wyjątek:

```
public static void divide(int a, int b)
    throws ArithmeticException {
    int c = a / b;
}
```

Samo umieszczenie klauzuli **throws** nie zapewni poprawnej obsługi naszego wyjątku!

Dlatego – przy wywołaniu metody **divide(a,b)** – zapewniamy obsługę wyjątku.

```
public static void main(String[] args) {
    try {
        divide(4, 2);
    } catch (ArithmeticException e) {
    System.out.println(e.getMessage());
    }
  }
```

Zasady

Stosując kilka klauzul **`catch`** powinniśmy zaczynać od najbardziej szczegółowych do najbardziej ogólnych, zgodnie z hierarchią.

Jeśli nie zastosujemy się do tej zasady pisząc np.:

```
try {
    int num = Integer.parseInt("abc");
} catch (Exception e) {
} catch (NumberFormatException e) {
}
```

otrzymamy błąd kompilacji:

Unreachable catch block for NumberFormatException. It is already handled by the catch block for Exception.

To dlatego, że klasa **`NumberFormatException`** jest pochodną klasy **`Exception`** – w związku z tym drugi blok **`catch`** nigdy nie zostanie wykonany.

Popularne wyjątki

Kilka wyjątków oraz opis możliwych okoliczności ich wystąpienia:

- **`NullPointerException`** – podczas próby wywołania metody na nieistniejącym obiekcie,
- **`InputMismatchException`** – przy próbie odczytania danych w nieprawidłowym formacie, np. podczas korzystania z klasy Scanner,
- **`IllegalArgumentException`** – kiedy przekazywany argument jest z jakiegoś powodu nieprawidłowy,
- **`IOException`** – np. podczas pracy z plikami, problemów z systemem wejścia/wyjścia,
- **`NumberFormatException`** – przy nieudanej próbie zmiany na liczbę,
- **`IndexOutOfBoundException`** – gdy odwołujemy się do nieistniejącego elementu tablicy.

Zadania