helion.pl
KSIĘGARNIA INTERNETOWA · WYDAWNICTWO

IDŹ DO
PRZYKŁADOWY ROZDZIAŁ
SPIS TREŚCI

KATALOG KSIĄŻEK
KATALOG ONLINE
ZAMÓW DRUKOWANY KATALOG

TWÓJ KOSZYK
DODAJ DO KOSZYKA

CENNIK I INFORMACJE
ZAMÓW INFORMACJE O NOWOŚCIACH
ZAMÓW CENNIK

CZYTELNIA
FRAGMENTY KSIĄŻEK ONLINE

Praktyczny kurs Java

Autor: Marcin Lis
ISBN: 83-7361-395-1
Format: B5, stron: 384



Poznaj tajniki najpopularniejszego języka programowania w erze Internetu

Chyba wszyscy użytkownicy internetu spotkali się z Javą, często nawet o tym nie wiedząc. W ciągu ostatnich 10 lat zyskała ona ogromną popularność, szczególnie wśród programistów aplikacji sieciowych. Jednakże kojarzenie jej z językiem przeznaczonym wyłącznie do tworzenia takich programów jest dużym błędem. Java to w pełni funkcjonalny i doskonale dopracowany język programowania, nadający się do tworzenia różnych aplikacji, a nie tylko apletów działających na stronach internetowych.

W Javie pisane są gry sieciowe, systemy bankowości elektronicznej, pakiety wspomagające sprzedaż i obsługę klienta, a nawet aplikacje działające w telefonach komórkowych i komputerach przenośnych. Podstawową zaletą języka Java jest przenośność kodu – raz napisany program można uruchomić na każdym urządzeniu, na którym zainstalowane jest odpowiednie środowisko uruchomieniowe, zwane JRE.

Książka „Praktyczny kurs Java" przeznaczona jest dla osób rozpoczynających swoją przygodę z programowaniem w tym języku. Opisuje podstawy języka, zasady programowania obiektowego i tworzenia własnych apletów i aplikacji.
Czytając kolejne rozdziały, dowiesz się:

- Jakie typy danych wykorzystywane są w Javie
- Jak deklarować zmienne i wyprowadzać ich wartości na ekran
- W jaki sposób sterować przebiegiem wykonywania programu
- Jakie zasady rządzą programowaniem obiektowym
- Czym są klasy, obiekty, argumenty i metody
- Co to są wyjątki i jak je obsługiwać w programie
- Jak wykorzystać zaawansowane techniki programowania obiektowego w swoich aplikacjach
- W jaki sposób uzyskiwać dostęp do systemu plików z poziomu swojej aplikacji
- Jak tworzyć aplety i samodzielne aplikacje

Zapoznaj się z podstawami programowania w Javie i naucz się zasad programowania obiektowego, a także dowiedz się, czym są wyjątki w Javie i stwórz własne aplety i aplikacje.

Wydawnictwo Helion
ul. Chopina 6
44-100 Gliwice
tel. (32)230-98-63
e-mail: helion@helion.pl

Spis treści

Wstęp **5**

Rozdział 1. Podstawy **7**
Lekcja 1. Struktura programu, kompilacja i wykonanie 7
Lekcja 2. Podstawy obiektowości i typy danych 9
Lekcja 3. Komentarze 12

Rozdział 2. Instrukcje języka **17**
Zmienne 17
Lekcja 4. Deklaracje i przypisania 18
Lekcja 5. Wyprowadzanie danych na ekran 21
Lekcja 6. Operacje na zmiennych 27
Instrukcje sterujące 39
Lekcja 7. Instrukcja warunkowa if...else 39
Lekcja 8. Instrukcja switch i operator warunkowy 45
Lekcja 9. Pętle 49
Lekcja 10. Instrukcje break i continue 56
Tablice 63
Lekcja 11. Podstawowe operacje na tablicach 64
Lekcja 12. Tablice wielowymiarowe 68

Rozdział 3. Programowanie obiektowe **79**
Podstawy 79
Lekcja 13. Klasy, pola i metody 80
Lekcja 14. Argumenty i przeciążanie metod 87
Lekcja 15. Konstruktory 98
Dziedziczenie 110
Lekcja 16. Klasy potomne 110
Lekcja 17. Specyfikatory dostępu i pakiety 118
Lekcja 18. Przesłanianie metod i składowe statyczne 132
Lekcja 19. Klasy i składowe i finalne 145

Rozdział 4. Wyjątki **153**
Lekcja 20. Blok try...catch 153
Lekcja 21. Wyjątki to obiekty 162
Lekcja 22. Własne wyjątki 169

Rozdział 5. Programowanie obiektowe II 181
Polimorfizm 181
Lekcja 23. Konwersje typów i rzutowanie obiektów 181
Lekcja 24. Późne wiązanie i wywoływanie metod klas pochodnych 190
Lekcja 25. Konstruktory oraz klasy abstrakcyjne 199
Interfejsy 209
Lekcja 26. Tworzenie interfejsów 209
Lekcja 27. Wiele interfejsów 216
Klasy wewnętrzne 226
Lekcja 28. Klasa w klasie 226
Lekcja 29. Rodzaje klas wewnętrznych i dziedziczenie 235
Lekcja 30. Klasy anonimowe i zagnieżdżone 244

Rozdział 6. System wejścia-wyjścia 253
Lekcja 31. Standardowe wejście 253
Lekcja 32. Standardowe wejście i wyjście 263
Lekcja 33. System plików 273
Lekcja 34. Operacje na plikach 283

Rozdział 7. Aplikacje i aplety 301
Aplety 301
Lekcja 35. Podstawy apletów 301
Lekcja 36. Czcionki i kolory 310
Lekcja 37. Grafika 319
Lekcja 38. Dźwięki i obsługa myszy 330
Aplikacje 340
Lekcja 39. Tworzenie aplikacji 340
Lekcja 40. Komponenty 359

Skorowidz 375

Rozdział 4.
Wyjątki

Praktycznie w każdym większym programie powstają jakieś błędy. Powodów jest bardzo wiele, może być to skutek niefrasobliwości programisty, założenia, że wprowadzone dane są zawsze poprawne, niedokładnej specyfikacji poszczególnych modułów aplikacji, użycia niesprawdzonych bibliotek czy nawet zwykłego zapomnienia o zainicjowaniu jednej tylko zmiennej. Na szczęście w Javie, tak jak i w większości współczesnych obiektowych języków programowania, istnieje mechanizm tzw. wyjątków, który pozwala na przechwytywanie błędów. Ta właśnie tematyka zostanie przedstawiona w kolejnych trzech lekcjach.

Lekcja 20. Blok try...catch

Lekcja 20. jest poświęcona wprowadzeniu w tematykę wyjątków. Zobaczymy, jakie są sposoby zapobiegania powstawaniu niektórych typów błędów w programach, dowiemy się, jak stosować przechwytujący błędy blok instrukcji `try...catch`. Poznamy bliżej wyjątek o nieco egzotycznej dla początkujących programistów nazwie `ArrayIndexOutOfBoundsException`, dzięki któremu będziemy mogli uniknąć błędów związanych z przekroczeniem dopuszczalnego zakresu indeksów tablic.

Sprawdzanie poprawności danych

Powróćmy na chwilę do rozdziału 2. i lekcji 11. Znajdował się tam przykład, w którym następowało odwołanie do nieistniejącego elementu tablicy (listing 2.38). Występowała tam sekwencja instrukcji:

```
int tab[] = new int[10];
tab[10] = 5;
```

Doświadczony programista od razu zauważy, że instrukcje te są błędne, jako że zadeklarowana została tablica dziesięcioelementowa, więc — ponieważ indeksowanie tablicy zaczyna się od zera — ostatni element tablicy ma indeks `9`. Tak więc instrukcja `tab[10] = 5` powoduje próbę odwołania się do nieistniejącego jedenastego elementu tablicy. Ten błąd jest jednak stosunkowo prosty do wychwycenia, nawet gdyby pomiędzy deklaracją tablicy a nieprawidłowym odwołaniem były umieszczone inne instrukcje.

Dużo więcej kłopotów mogłyby nam sprawić sytuacja, gdyby np. tablica była deklarowana w jednej klasie, a odwołanie do niej następowało w innej klasie. Taka przykładowa sytuacja została przedstawiona na listingu 4.1.

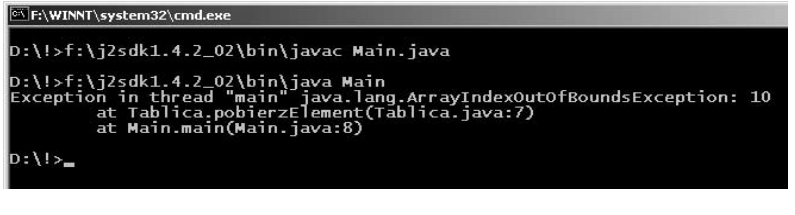
Listing 4.1.

```
public
class Tablica
{
  private int[] tablica = new int[10];
  public int pobierzElement(int indeks)
  {
        return tablica[indeks];
  }
  public void ustawElement(int indeks, int wartosc)
  {
        tablica[indeks] = wartosc;
  }
}

public
class Main
{
  public static void main (String args[])
  {
        Tablica tablica = new Tablica();
        tablica.ustawElement(5, 10);
        int liczba = tablica.pobierzElement(10);
        System.out.println(liczba);
  }
}
```

Powstały tu dwie klasy: `Tablica` oraz `Main`. W klasie `Tablica` zostało zadeklarowane prywatne pole typu tablicowego o nazwie `tablica`, któremu została przypisana dziesięcioelementowa tablica liczb całkowitych. Ponieważ pole to jest polem prywatnym (por. lekcja 17.), dostęp do niego mają jedynie inne składowe klasy `Tablica`. Dlatego też powstały dwie metody `pobierzElement` oraz `ustawElement` operujące na elementach tablicy. Metoda `pobierzElement` zwraca wartość zapisaną w komórce o indeksie przekazanym jako argument, natomiast `ustawElement` zapisuje wartość drugiego argumentu w komórce o indeksie wskazywanym przez argument pierwszy.

W klasie `Main` tworzymy obiekt klasy `Tablica` i wykorzystujemy metodę `ustawElement` do zapisania w piątej komórce wartości `10`. W kolejnej linii popełniamy drobny błąd. W metodzie `pobierzElement` odwołujemy się do nieistniejącego elementu o indeksie `10`. Musi to spowodować wystąpienie błędu w trakcie działania aplikacji (rysunek 4.1). Błąd tego typu bardzo łatwo popełnić, gdyż w klasie `Main` nie widzimy rozmiarów tablicy, nietrudno więc o pomyłkę.



Rysunek 4.1. *Odwołanie do nieistniejącego elementu w klasie Tablica*

Jak poradzić sobie z takim problemem? Pierwszym nasuwającym się sposobem jest sprawdzanie w metodach `pobierzElement` i `ustawElement`, czy przekazane argumenty nie przekraczają dopuszczalnych wartości. Jeśli takie przekroczenie wartości nastąpi, należy wtedy zasygnalizować błąd. To jednak prowokuje pytanie: w jaki sposób ten błąd sygnalizować? Pomysłem znanym programistom C/C++ jest np. zwracanie przez funkcję (metodę) wartości `-1` w przypadku błędu oraz wartości nieujemnej (najczęściej zero), jeśli błąd nie wystąpił[1]. Ta metoda będzie dobra w przypadku metody `ustawElement`, która wyglądałaby wtedy następująco:

```
public int ustawElement(int indeks, int wartosc)
{
  if(indeks >= tablica.length){
        return -1;
  }
  else{
        tablica[indeks] = wartosc;
        return 0;
  }
}
```

Wystarczyłoby teraz w klasie `Main` testować wartość zwróconą przez `ustawElement`, aby sprawdzić, czy nie przekroczyliśmy dopuszczalnego indeksu tablicy. Niestety, tej techniki nie można zastosować w przypadku metody `pobierzElement`, przecież zwraca ona wartość zapisaną w jednej z komórek tablicy. Czyli `-1` i `0` użyte przed chwilą do zasygnalizowania, czy operacja zakończyła się błędem, mogą być wartościami odczytanymi z tablicy. Trzeba zatem wymyślić inny sposób. Może być to np. wykorzystanie dodatkowego pola sygnalizującego w klasie `Tablica`. Pole to byłoby typu `boolean`. Ustawione na `true` oznaczałoby, że ostatnia operacja na klasie zakończyła się błędem, natomiast ustawione na `false`, że ostatnia operacja zakończyła się sukcesem. Klasa `Tablica` miałaby wtedy postać, jak na listingu 4.2.

Listing 4.2.

```
public
class Tablica
{
  private int[] tablica = new int[10];
  public boolean wystapilBlad = false;
  public int pobierzElement(int indeks)
  {
        if(indeks >= tablica.length){
              wystapilBlad = true;
              return 0;
        }
        else{
              wystapilBlad = false;
              return tablica[indeks];
        }
  }
```

[1] To tylko przykład. Równie często stosuje się zwrócenie wartości zero jako oznaczenie prawidłowego wykonania funkcji.

```
  public void ustawElement(int indeks, int wartosc)
  {
        if(indeks >= tablica.length){
              wystapilBlad = true;
        }
        else{
              tablica[indeks] = wartosc;
              wystapilBlad = false;
        }
  }
}
```

Do klasy dodaliśmy pole typu `boolean` o nazwie `wystapilBlad`. Jego początkowa wartość to `false`. W metodzie `pobierzElement` sprawdzamy najpierw, czy przekazany indeks nie przekracza dopuszczalnej maksymalnej wartości. Jeśli tak, ustawiamy pole `wystapilBlad` na `true` oraz zwracamy wartość zero. Oczywiście, w tym przypadku zwrócona wartość nie ma żadnego praktycznego znaczenia (przecież wystąpił błąd), niemniej coś musimy zwrócić. Użycie instrukcji `return` i zwrócenie wartości typu `int` jest bowiem konieczne, inaczej kompilator zgłosi błąd. Jeśli jednak argument przekazany metodzie nie przekracza dopuszczalnego indeksu tablicy, pole `wystapilBlad` ustawiamy na `false` oraz zwracamy wartość znajdującą się pod tym indeksem.

W metodzie `ustawElement` postępujemy podobnie. Sprawdzamy, czy przekazany indeks nie przekracza dopuszczalnej wartości. Jeśli tak, pole `wystapilBlad` ustawiamy na `true`, w przeciwnym przypadku dokonujemy przypisania wskazanej komórce tablicy i ustawiamy `wystapilBlad` na `false`. Po takiej modyfikacji obu metod w klasie `Main` możemy już bez problemów stwierdzić, czy operacje wykonywane na klasie `Tablica` zakończyły się sukcesem. Przykładowe wykorzystanie możliwości, jakie daje nowe pole, zostało przedstawione na listingu 4.3.

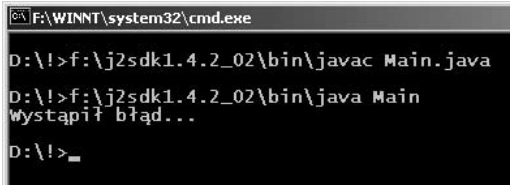
Listing 4.3.

```
public
class Main
{
  public static void main (String args[])
  {
        Tablica tablica = new Tablica();
        tablica.ustawElement(5, 10);
        int liczba = tablica.pobierzElement(10);
        if (tablica.wystapilBlad){
              System.out.println("Wystąpił błąd...");
        }
        else{
              System.out.println(liczba);
        }
  }
}
```

Podstawowe wykonywane operacje są takie same, jak w przypadku klasy z listingu 4.1. Po pobraniu elementu sprawdzamy jednak, czy operacja ta zakończyła się sukcesem i wyświetlamy odpowiedni komunikat na ekranie. Identyczne sprawdzenie można

wykonać również po wywołaniu metody `ustawElement`. Wykonanie kodu z listingu 4.3 spowoduje rzecz jasna wyświetlenie napisu `Wystąpił błąd` (rysunek 4.2).

Rysunek 4.2. *Efekt wykonania programu z listingu 4.3*



Sposobów poradzenia sobie z problemem przekroczenia indeksu tablicy można by zapewne wymyślić jeszcze kilka. Metody tego typu mają jednak poważną wadę: programiści mają tendencję do ich… niestosowania. Często możliwy błąd wydaje się zbyt banalny, aby się nim zajmować, czasami po prostu się zapomina się o sprawdzaniu pewnych warunków. Dodatkowo przedstawione wyżej sposoby, czyli zwracanie wartości sygnalizacyjnej czy dodatkowe zmienne, powodują niepotrzebne rozbudowywanie kodu aplikacji, co paradoksalnie może prowadzić do powstawania kolejnych błędów, czyli błędów powstałych przez napisanie kodu zajmującego się obsługą błędów… W Javie zastosowano więc mechanizm tak zwanych wyjątków, znany na pewno programistom C++ i Object Pascala[2].

Wyjątki w Javie

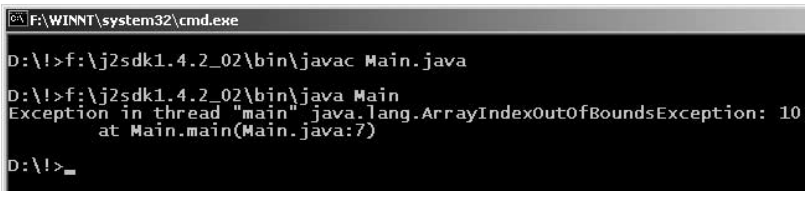
Wyjątek (ang. *exception*) jest to byt programistyczny, który powstaje w sytuacji wystąpienia błędu. Z powstaniem wyjątku spotkaliśmy się już w rozdziale 2. w lekcji 11. Był to wyjątek spowodowany przekroczeniem dopuszczalnego zakresu tablicy. Pokażmy go raz jeszcze. Na listingu 4.4 została zaprezentowana odpowiednio spreparowana klasa `Main`.

Listing 4.4.

```
public
class Main
{
  public static void main (String args[])
  {
         int tab[] = new int[10];
         tab[10] = 100;
  }
}
```

Deklarujemy tablicę liczb typu `int` o nazwie `tab` oraz próbujemy przypisać elementowi o indeksie `10` (czyli wykraczającym poza zakres tablicy) wartość `100`. Jeśli skompilujemy i uruchomimy taki program, na ekranie zobaczymy obraz widoczny na rysunku 4.3. Został tu wygenerowany wyjątek o nazwie `ArrayIndexOutOfBoundsException`, czyli wyjątek oznaczający, że indeks tablicy znajduje się poza jej granicami.

[2] Sama technika obsługi sytuacji wyjątkowych sięga jednak lat sześćdziesiątych ubiegłego stulecia (czyli wieku XX).



Rysunek 4.3. *Odwołanie do nieistniejącego elementu tablicy spowodowało wygenerowanie wyjątku*

Oczywiście, gdyby możliwości wyjątków kończyłyby się na wyświetlaniu informacji na ekranie i przerywaniu działania programu, ich przydatność byłaby mocno ograniczona. Na szczęście, wygenerowany wyjątek można przechwycić i wykonać własny kod obsługi błędu. Do takiego przechwycenia służy blok instrukcji `try...catch`. W najprostszej postaci wygląda on następująco:

```
try{
  //instrukcje mogące spowodować wyjątek
}
catch (TypWyjątku identyfikatorWyjątku){
  //obsługa wyjątku
}
```

W nawiasach klamrowych występujących po słowie `try` umieszczamy instrukcję, która może spowodować wystąpienie wyjątku. W bloku występującym po `catch` umieszczamy kod, który ma zostać wykonany, kiedy wyjątek wystąpi. W przypadku klasy `Main` z listingu 4.4 blok `try...catch` należałoby zastosować sposób przedstawiony na listingu 4.5.

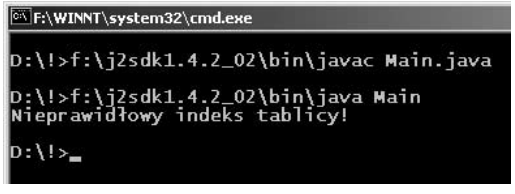
Listing 4.5.

```
public
class Main
{
  public static void main (String args[])
  {
        int tab[] = new int[10];
        try{
              tab[10] = 100;
        }
        catch(ArrayIndexOutOfBoundsException e){
              System.out.println("Nieprawidłowy indeks tablicy!");
        }
  }
}
```

Jak widać, wszystko odbywa się tu zgodnie z wcześniejszym ogólnym opisem. W bloku `try` znalazła się instrukcja `tab[10] = 100`, która — jak wiemy — spowoduje wygenerowanie wyjątku. W nawiasach okrągłych występujących po instrukcji `catch` został wymieniony rodzaj wyjątku, który zostanie wygenerowany: `ArrayIndexOutOfBounds➥Exception`, oraz jego identyfikator: `e`. Identyfikator to nazwa[3], która pozwala na wykonywanie operacji związanych z wyjątkiem, czym jednak zajmiemy się w kolejnej

[3] Dokładniej jest to nazwa zmiennej obiektowej, wyjaśnimy to bliżej w lekcji 21.

lekcji. W bloku po `catch` znajduje się instrukcja `System.out.println` wyświetlająca odpowiedni komunikat na ekranie. Tym razem po uruchomieniu kodu zobaczmy widok zaprezentowany na rysunku 4.4.



Rysunek 4.4. *Wyjątek został przechwycony w bloku try...catch*

Blok `try...catch` nie musi jednak obejmować tylko jednej instrukcji ani też tylko instrukcji mogących wygenerować wyjątek. Przypomnijmy ponownie listing 2.38. Blok `try` mógłby w tym wypadku objąć wszystkie trzy instrukcje, czyli klasa `Main` miałaby wtedy postać jak na listingu 4.6.

Listing 4.6.

```
class Main
{
  public static void main (String args[])
  {
        try{
              int tab[] = new int[10];
              tab[10] = 5;
              System.out.println("Dziesiąty element tablicy ma wartość: " + tab[10]);
        }
        catch(ArrayIndexOutOfBoundsException e){
               System.out.println("Nieprawidłowy indeks tablicy!");
        }
  }
}
```

Nie istnieje również konieczność obejmowania blokiem `try` instrukcji bezpośrednio generujących wyjątek, tak jak miało to miejsce w dotychczasowych przykładach. Wyjątek wygenerowany przez obiekt klasy `Y` może być bowiem przechwytywany w klasie `X`, która korzysta z obiektów klasy `Y`. Pokażemy to na przykładzie klas z listingu 4.1. Klasa `Tablica` pozostanie bez zmian, natomiast klasę `Main` zmodyfikujemy tak, aby miała wygląd zaprezentowany na listingu 4.7.

Listing 4.7.

```
public
class Main
{
  public static void main (String args[])
  {
        Tablica tablica = new Tablica();
        try{
              tablica.ustawElement(5, 10);
              int liczba = tablica.pobierzElement(10);
              System.out.println(liczba);
        }
```

```
        catch(ArrayIndexOutOfBoundsException e){
              System.out.println("Nieprawidłowy indeks tablicy!");
        }
  }
}
```

Spójrzmy, w bloku `try` wykonujemy trzy instrukcje, z których jedna: `int liczba = tablica.pobierzElement(10)` jest odpowiedzialna za wygenerowanie wyjątku. Wyjątek jest przecież jednak generowany w ciele metody `pobierzElement` klasy `Tablica`, a nie w klasie `Main`! Zostanie on jednak przekazany klasie `Main`, jako że wywołuje ona metodę klasy `Tablica`. Tym samym w klasie `Main` możemy zastosować blok `try...catch`.

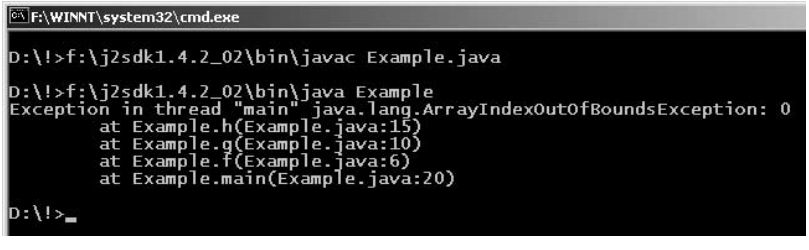
Z identyczną sytuacją będziemy mieli do czynienia w przypadku hierarchicznego wywołania metod jednej klasy. Czyli w sytuacji, kiedy metoda `f` wywołuje metodę `g`, która wywołuje metodę `h`, która z kolei generuje wyjątek. W każdej z wymienionych metod można zastosować blok `try...catch` do przechwycenia tego wyjątku. Dokładnie taki przykład jest widoczny na listingu 4.8.

Listing 4.8.

```
public
class Example
{
  public void f()
  {
        try{
              g();
        }
        catch(ArrayIndexOutOfBoundsException e){
              System.out.println("Wyjątek: metoda f");
        }
  }
  public void g()
  {
        try{
              h();
        }
        catch(ArrayIndexOutOfBoundsException e){
              System.out.println("Wyjątek: metoda g");
        }
  }
  public void h()
  {
        int[] tab = new int[0];
        try{
              tab[0] = 1;
        }
        catch(ArrayIndexOutOfBoundsException e){
              System.out.println("Wyjątek: metoda h");
        }
  }
  public static void main(String args[])
  {
        Example ex = new Example();
```

```
            try{
                  ex.f();
            }
            catch(ArrayIndexOutOfBoundsException e){
                  System.out.println("Wyjątek: metoda main");
            }
      }
}
```

Taką klasę skompilujemy bez żadnych problemów. Musimy jednak dobrze zdawać sobie sprawę, jak taki kod będzie wykonywany. Pytanie bowiem brzmi: które bloki `try` zostaną wykonane? Zasada jest następująca: zostanie wykonany blok najbliższy instrukcji powodującej wyjątek. Czyli w przypadku przedstawionym na listingu 4.8 jedynie blok obejmujący wywołanie metody `tab[0] = 1;` w metodzie `h`. Jeśli jednak będziemy usuwać kolejne bloki `try` najpierw z instrukcji `h`, następnie `g`, `f` i ostatecznie z `main`, zobaczymy, że faktycznie wykonywany jest zawsze blok najbliższy miejsca wystąpienia błędu. Po usunięciu wszystkich instrukcji `try` wyjątek nie zostanie obsłużony w naszej klasie i obsłuży go maszyna wirtualna Javy, co zaowocuje znanym nam już komunikatem na konsoli. Zwróćmy jednak uwagę, że w tym wypadku zostanie wyświetlona cała hierarchia metod, w których był propagowany wyjątek (rysunek 4.5).



Rysunek 4.5. *Przy hierarchicznym wywołaniu metod po wystąpieniu wyjątku otrzymamy ich nazwy*

Ćwiczenia do samodzielnego wykonania

20.1. Popraw kod klasy z listingu 4.2 tak, aby w metodach `pobierzElement` i `ustawElement` było również sprawdzane, czy przekazany indeks nie przekracza minimalnej dopuszczalnej wartości.

20.2. Zmień kod klasy `Main` z listingu 4.3 w taki sposób, aby było również sprawdzane, czy wywołanie metody `ustawElement` zakończyło się sukcesem.

20.3. Popraw kod z listingu 4.2 tak, aby do wychwytywania błędów był wykorzystywany mechanizm wyjątków zamiast instrukcji warunkowej `if`.

20.4. Napisz klasę `Example`, w której będzie się znajdowała metoda o nazwie `a`, która z kolei będzie wywoływała metodę o nazwie `b`. W metodzie `a` wygeneruj wyjątek `ArrayIndexOutOfBoundsException`. Napisz następnie klasę `Main` zawierającą metodę `main`, w której zostanie utworzony obiekt klasy `Example` i zostaną wywołane metody `a` oraz `b` tego obiektu. W metodzie `main` zastosuj bloki `try...catch`, przechwytujące powstały wyjątek.

Lekcja 21. Wyjątki to obiekty

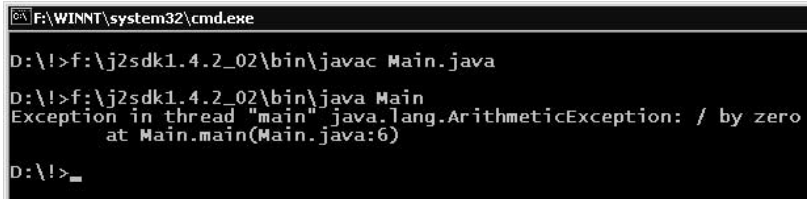
W lekcji 20. poznaliśmy wyjątek sygnalizujący przekroczenie dopuszczalnego zakresu tablicy. To oczywiście nie jedyny dostępny wyjątek, czas poznać również inne ich typy. W lekcji 21. dowiemy się więc, że wyjątki są tak naprawdę obiektami, a także, że tworzą one hierarchiczną strukturę. Pokażemy, jak przechwytywać wiele wyjątków w jednym bloku `try` oraz udowodnimy, że bloki `try...catch` mogą być zagnieżdżane. Okaże się, że jeden wyjątek ogólny może obsłużyć wiele błędnych sytuacji.

Dzielenie przez zero

Rodzajów wyjątków jest bardzo wiele. Wiemy już, jak reagować na przekroczenie zakresu tablicy. Poznajmy zatem inny typ wyjątku, powstający, kiedy zostanie podjęta próba wykonania nielegalnego dzielenia przez zero. W tym celu musimy spreparować odpowiedni fragment kodu. Wystarczy, że w metodzie `main` umieścimy przykładową instrukcję:

```
int liczba = 10 / 0;.
```

Kompilacja takiego kodu przebiegnie bez problemu, jednak próba wykonania musi skończyć się komunikatem o błędzie, widocznym na rysunku 4.6. Widzimy wyraźnie, że tym razem został zgłoszony wyjątek `ArithmeticException` (wyjątek arytmetyczny).



Rysunek 4.6. *Próba wykonania dzielenia przez zero*

Wykorzystując wiedzę z lekcji 20., nie powinniśmy mieć żadnych problemów z napisaniem kodu, który taki wyjątek przechwyci. Trzeba wykorzystać dobrze nam znaną instrukcję `try...catch` w postaci:

```
try{
  int liczba = 10 / 0;
}
catch(ArithmeticException e){
  //instrukcje do wykonania kiedy wystąpi wyjątek
}
```

Intuicja podpowiada nam już zapewne, że rodzajów wyjątków może być bardzo, bardzo dużo. I faktycznie tak jest, w klasach dostarczonych w JDK (w wersji SE) jest ich zdefiniowanych blisko 300. Aby sprawnie się nimi posługiwać, musimy dowiedzieć się, czym tak naprawdę są wyjątki.

Wyjątek jest obiektem

Wyjątek, który określaliśmy jako byt programistyczny, to nic innego jak obiekt, który powstaje, kiedy w programie wystąpi sytuacja wyjątkowa. Skoro wyjątek jest

obiektem, to wspominany wcześniej typ wyjątku (`ArrayIndexOutOfBoundsException`, `ArithmeticException`) to nic innego jak klasa opisująca tenże obiekt. Jeśli teraz spojrzymy ponownie na ogólną postać instrukcji `try...catch`:

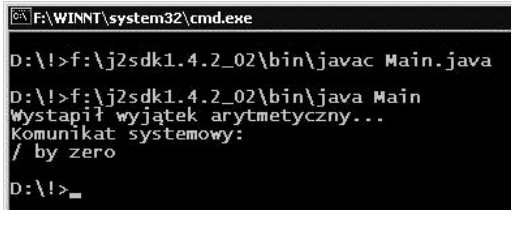
```
try{
  //instrukcje mogące spowodować wyjątek
}
catch(TypWyjątku identyfikatorWyjątku){
  //obsługa wyjątku
}
```

jasnym stanie się, że w takim razie *identyfikatorWyjątku* to zmienna obiektowa, wskazująca na obiekt wyjątku. Na tym obiekcie możemy wykonywać operacje zdefiniowane w klasie wyjątku. Możemy np. uzyskać systemowy komunikat o błędzie. Wystarczy wywołać metodę `getMessage()`. Zobaczmy to na przykładzie wyjątku generowanego podczas próby wykonania dzielenia przez zero. Jest on zaprezentowany na listingu 4.9.

Listing 4.9.

```
public
class Main
{
  public static void main (String args[])
  {
        try{
              int liczba = 10 / 0;
        }
        catch(ArithmeticException e){
              System.out.println("Wystapił wyjątek arytmetyczny...");
              System.out.println("Komunikat systemowy:");
              System.out.println(e.getMessage());
        }
  }
}
```

Wykonujemy tutaj próbę niedozwolonego dzielenia przez zero oraz przechwytujemy wyjątek klasy `ArithmeticException`. W bloku `catch` najpierw wyświetlamy nasze własne komunikaty o błędzie, a następnie komunikat systemowy. Po uruchomieniu kodu na ekranie zobaczymy widok zaprezentowany na rysunku 4.7.



Rysunek 4.7. *Wyświetlenie systemowego komunikatu obłędzie*

Istnieje jeszcze jedna możliwość uzyskania komunikatu o wyjątku, mianowicie umieszczenie w argumencie instrukcji `System.out.println` zmiennej wskazującej na obiekt wyjątku, czyli w przypadku listingu 4.9:

```
System.out.println(e);
```

W pierwszej chwili może się to wydawać nieco dziwne, bo niby skąd instrukcja `System.out.println` ma wiedzieć, co w takiej sytuacji wyświetlić? Zauważmy jednak, że jest to sytuacja analogiczna, jak w przypadku typów prostych (por. lekcja 5.). Skoro udawało się automatycznie przekształcać np. zmienną typu `int` na ciąg znaków, uda się również przekształcić zmienną typu obiektowego. Bliżej tym problemem zajmiemy się w rozdziale piątym.

Jeśli teraz z programie z listingu 4.9 zamienimy instrukcję:

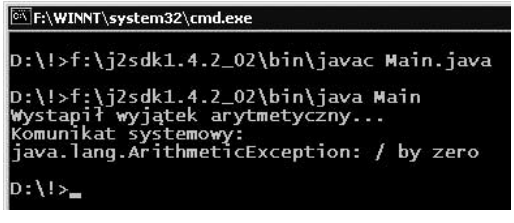
```
System.out.println(e.getMessage());
```

na:

```
System.out.println(e);
```

otrzymamy nieco dokładniejszy komunikat określający dodatkowo klasę wyjątku, tak jak jest to widoczne na rysunku 4.8.

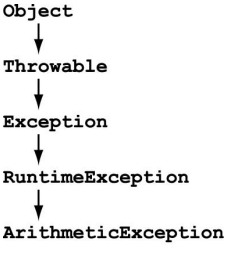
Rysunek 4.8.
Pełniejszy komunikat o typie wyjątku



Hierarchia wyjątków

Każdy wyjątek jest obiektem pewnej klasy. Klasy podlegają z kolei regułom dziedziczenia, zgodnie z którymi powstaje hierarchia klas. Kiedy zatem pracujemy z wyjątkami, musimy tę kwestię wziąć pod uwagę. Wszystkie standardowe wyjątki, które możemy przechwytywać w naszych aplikacjach za pomocą bloku `try...catch`, dziedziczą z klasy `Exception`, która z kolei dziedziczy z `Throwable` oraz `Object`. Hierarchia klas dla wyjątku `ArithmeticException`, który wykorzystywaliśmy we wcześniejszych przykładach, jest zaprezentowana na rysunku 4.9.

Rysunek 4.9.
Hierarchia klas dla wyjątku ArithmeticException



Wynika z tego kilka własności. Przede wszystkim, jeśli spodziewamy się, że dana instrukcja może wygenerować wyjątek typu `X`, możemy zawsze przechwycić wyjątek ogólniejszy, czyli wyjątek, którego typem będzie jedna z klas nadrzędnych do `X`. Jest to bardzo wygodna technika. Przykładowo z klasy `RuntimeException` dziedziczy bardzo wiele klas wyjątków odpowiadających najróżniejszym błędom. Jedną z nich jest `ArithmeticException`. Jeśli instrukcje, które obejmujemy blokiem `try...catch`, mogą

spowodować wiele różnych wyjątków, zamiast stosować wiele oddzielnych instrukcji przechwytujących konkretne typy błędów, często lepiej jest zastosować jedną przechwytującą wyjątek bardziej ogólny. Spójrzmy na listing 4.10.

Listing 4.10.

```
public
class Main
{
  public static void main (String args[])
  {
        try{
              int liczba = 10 / 0;
        }
        catch(RuntimeException e){
              System.out.println("Wystapił wyjątek czasu wykonania...");
              System.out.println("Komunikat systemowy:");
              System.out.println(e);
        }
  }
}
```

Jest to znany nam już program, generujący błąd polegający na próbie wykonania niedozwolonego dzielenia przez zero. Tym razem jednak zamiast wyjątku klasy `ArithmeticException` przechwytujemy wyjątek klasy nadrzędnej `RuntimeException`. Co więcej, nic nie stoi na przeszkodzie, aby przechwycić wyjątek jeszcze ogólniejszy, czyli wyjątek klasy nadrzędnej do `RuntimeException`. Jak widać na rysunku 4.9, byłaby to klasa `Exception`.

Przechwytywanie wielu wyjątków

W jednym bloku `try...catch` można przechwytywać wiele wyjątków. Konstrukcja taka zawiera wtedy jeden blok `try` i wiele bloków `catch`. Schematycznie wygląda ona następująco:

```
try{
  //instrukcje mogące spowodować wyjątek
}
catch(KlasaWyjątku1 identyfikatorWyjątku1){
  //obsługa wyjątku 1
}
  catch(KlasaWyjątku2 identyfikatorWyjątku2){
  //obsługa wyjątku 2
}
/*
... dalsze bloki catch ...
*/
catch(KlasaWyjątku n identyfikatorWyjątkun){
  //obsługa wyjątku n
}
```

Po wygenerowaniu wyjątku jest sprawdzane, czy jest on klasy *`KlasaWyjatku1`*, jeśli tak — są wykonywane instrukcje obsługi tego wyjątku i blok `try...catch` jest opuszczany. Jeżeli jednak wyjątek nie jest klasy *`KlasaWyjatku1`*, jest sprawdzane, czy jest on klasy *`KlasaWyjątku2`* itd.

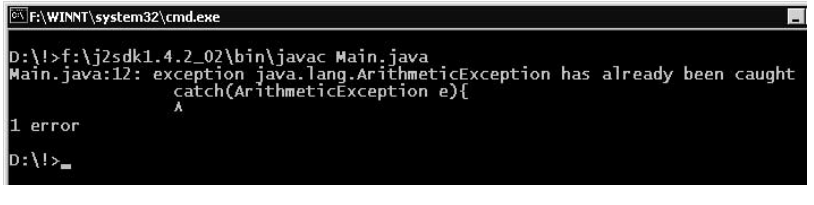
Przy tego typu konstrukcjach należy jednak pamiętać o hierarchii wyjątków, nie jest bowiem obojętne, w jakiej kolejności będą one przechwytywane. Ogólna zasada jest taka, że nie ma znaczenia kolejność, o ile wszystkie wyjątki są na jednym poziomie hierarchii. Jeśli jednak przechwytujemy wyjątki z różnych poziomów, najpierw muszą to być wyjątki bardziej szczegółowe, czyli stojące niżej w hierarchii, a dopiero po nich wyjątki bardziej ogólne, czyli stojące wyżej w hierarchii.

Nie można zatem najpierw przechwycić wyjątku `RuntimeException`, a dopiero nim wyjątku `ArithmeticException` (por. rysunek 4.9), gdyż skończy się to błędem kompilacji. Jeśli więc dokonamy próby kompilacji przykładowego programu przedstawionego na listingu 4.11, efektem będą komunikaty widoczne na rysunku 4.10.

Listing 4.11.

```
public
class Main
{
  public static void main (String args[])
  {
        try{
              int liczba = 10 / 0;
        }
        catch(RuntimeException e){
              System.out.println(e);
        }
        catch(ArithmeticException e){
              System.out.println(e);
        }
  }
}
```

Rysunek 4.10. *Błędna hierarchia wyjątków powoduje błąd kompilacji*



Dzieje się tak dlatego, że (można powiedzieć) błąd bardziej ogólny zawiera już w sobie błąd bardziej szczegółowy. Jeśli zatem przechwytujemy najpierw wyjątek `RuntimeException`, to tak jak byśmy przechwycili już wyjątki wszystkich klas dziedziczących z `RuntimeException`. Dlatego też kompilator protestuje.

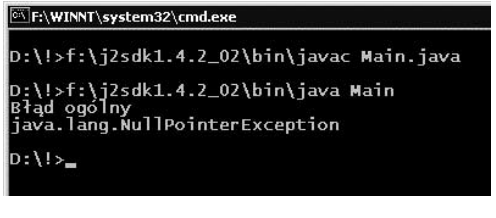
Kiedy jednak może przydać się sytuacja, że najpierw przechwytujemy wyjątek szczegółowy, a potem dopiero ogólny? Otóż, wtedy, kiedy chcemy w specyficzny sposób zareagować na konkretny typ wyjątku, a wszystkie pozostałe z danego poziomu hierarchii obsłużyć w identyczny, standardowy sposób. Taka przykładowa sytuacja jest przedstawiona na listingu 4.12.

Listing 4.12.

```
public
class Main
{
  public static void main (String args[])
  {
        Punkt punkt = null;
        int liczba;
        try{
              liczba = 10 / 0;
              punkt.x = liczba;
        }
        catch(ArithmeticException e){
              System.out.println("Nieprawidłowa operacja arytmetyczna");
              System.out.println(e);
        }
        catch(Exception e){
              System.out.println("Błąd ogólny");
              System.out.println(e);
        }
  }
}
```

Zostały zadeklarowane dwie zmienne: pierwsza typu `Punkt` o nazwie `punkt` oraz druga typu `int` o nazwie `liczba`. Zmiennej `punkt` została przypisana wartość pusta `null`, nie został zatem utworzony żaden obiekt klasy `Punkt`. W bloku `try` są wykonywane dwie błędne instrukcje. Pierwsza z nich to znane nam z poprzednich przykładów dzielenie przez zero. Druga instrukcja z bloku `try` to z kolei próba odwołania się do pola `x` nieistniejącego obiektu klasy `Punkt` (przecież zmienna `punkt` zawiera wartość `null`). Ponieważ chcemy w sposób niestandardowy zareagować na błąd arytmetyczny, najpierw przechwytujemy błąd typu `ArithmeticException` i, w przypadku kiedy wystąpi, wyświetlamy na ekranie napis `Nieprawidłowa operacja arytmetyczna`. W drugim bloku `catch` przechwytujemy wszystkie inne możliwe wyjątki, w tym także wyjątek `NullPointer➥Exception` występujący, kiedy próbujemy wykonać operacje na zmiennej obiektowej, która zawiera wartość `null`.

Po uruchomieniu kodu z listingu 4.12 na ekranie pojawi się zgłoszenie tylko pierwszego błędu. Dzieje się tak dlatego, że po jego wystąpieniu blok `try` został przerwany, a sterowanie zostało przekazane blokowi `catch`. Czyli jeśli w bloku `try` któraś z instrukcji spowoduje wygenerowanie wyjątku, dalsze instrukcje z bloku `try` nie zostaną wykonane. Nie miała więc szansy zostać wykonana nieprawidłowa instrukcja `punkt.x = liczba;`. Jeśli jednak usuniemy wcześniejsze dzielenie przez zero, przekonamy się, że i ten błąd zostanie przechwycony przez drugi blok `catch`, a na ekranie pojawi się stosowny komunikat (rysunek 4.11).



Rysunek 4.11. *Odwołanie do pustej zmiennej obiektowej zostało wychwycone przez drugi blok catch*

Zagnieżdżanie bloków try...catch

Bloki `try...catch` można zagnieżdżać. To znaczy, że w jednym bloku przechwytującym wyjątek `X` może istnieć drugi blok, który będzie przechwytywał wyjątek `Y`. Schematycznie taka konstrukcja wygląda następująco:

```
try{
  //instrukcje mogące spowodować wyjątek 1
  try{
        //instrukcje mogące spowodować wyjątek 2
  }
  catch (TypWyjątku2 identyfikatorWyjątku2){
        //obsługa wyjątku 2
  }
}
catch (TypWyjątku1 identyfikatorWyjątku1){
  //obsługa wyjątku 1
}
```

Zagnieżdżenie takie może być wielopoziomowe, czyli w już zagnieżdżonym bloku `try` można umieścić kolejny blok `try`, choć w praktyce takich piętrowych konstrukcji zazwyczaj się nie stosuje. Zwykle nie ma takiej potrzeby. Maksymalny poziom takiego bezpośredniego zagnieżdżenia z reguły nie przekracza dwóch poziomów. Aby na praktycznym przykładzie pokazać taką dwupoziomową konstrukcję, zmodyfikujemy przykład z listingu 4.12. Zamiast obejmowania jednym blokiem `try` dwóch instrukcji powodujących błąd, zastosujemy zagnieżdżenie, tak jak jest to widoczne na listingu 4.13.

Listing 4.13.

```
public
class Main
{
  public static void main (String args[])
  {
        Punkt punkt = null;
        int liczba;
        try{
              try{
                    liczba = 10 / 0;
              }
              catch(ArithmeticException e){
                    System.out.println("Nieprawidłowa operacja arytmetyczna");
                    System.out.println("Przypisuję zmiennejliczba wartość 10");
                    liczba = 10;
              }

              punkt.x = liczba;
        }
        catch(Exception e){
              System.out.println("Błąd ogólny");
              System.out.println(e);
        }
  }
}
```

Podobnie jak w poprzednim przypadku, deklarujemy dwie zmienne: `punkt` klasy `Punkt` oraz `liczba` typu `int`. Zmiennej `punkt` przypisujemy wartość pustą `null`. W wewnętrznym bloku `try` próbujemy wykonać nieprawidłowe dzielenie przez zero i przechwytujemy wyjątek `ArithmeticException`. Jeśli on wystąpi, zmienna `liczba` otrzymuje domyślną wartość równą `10`, dzięki czemu można wykonać kolejną operację, czyli próbę przypisania polu `x` obiektu `punkt` wartości zmiennej `liczba`. Rzecz jasna, przypisanie takie nie może zostać wykonane, gdyż zmienna `punkt` jest pusta, jest zatem generowany wyjątek `NullPointerException`, który jest przechwytywany przez zewnętrzny blok `try`. Widać więc, że zagnieżdżanie bloków `try` może być przydatne, choć warto zauważyć, że identyczny efekt można osiągnąć, korzystając również z niezagnieżdżonej postaci instrukcji `try...catch` (por. ćwiczenie 21.3).

Ćwiczenia do samodzielnego wykonania

21.1. Popraw kod z listingu 4.11 tak, aby przechwytywanie wyjątków odbywało się w prawidłowej kolejności.

21.2. Zmodyfikuj kod z listingu 4.12 tak, aby zostały zgłoszone oba typy błędów: `ArithmeticException` oraz `NullPointerException`.

21.3. Zmodyfikuj kod z listingu 4.5 w taki sposób, aby usunąć zagnieżdżenie bloków `try...catch`, nie zmieniając jednak efektów działania programu.

Lekcja 22. Własne wyjątki

Wyjątki możemy przechwytywać, aby zapobiec niekontrolowanemu zakończeniu programu w przypadku wystąpienia błędu. Tą technikę poznaliśmy w lekcjach 20. i 21. Okazuje się jednak, że można je również samemu zgłaszać, a także że można tworzyć nowe, nieistniejące wcześniej klasy wyjątków. Tej właśnie tematyce jest poświęcona bieżąca, 22., lekcja.

Zgłoś wyjątek

Wiemy, że wyjątki są obiektami. Skoro tak, zgłoszenie własnego wyjątku będzie polegało po prostu na utworzeniu nowego obiektu klasy opisującej wyjątek. Dokładniej za pomocą instrukcji `new` należy utworzyć nowy obiekt klasy, która pośrednio lub bezpośrednio dziedziczy z klasy `Throwable`. W najbardziej ogólnym przypadku będzie to klasa `Exception`. Tak utworzony obiekt musi stać się parametrem instrukcji `throw`. Jeśli zatem gdziekolwiek w pisanym przez nas kodzie chcemy zgłosić wyjątek ogólny, wystarczy, że napiszemy:

```
throw new Exception();
```

W specyfikacji metody musimy jednak zaznaczyć, że będziemy w niej zgłaszać wyjątek danej klasy. Robimy to za pomocą instrukcji `throws`, w ogólnej postaci:

```
specyfikator_dostepu [static] [final] typ_zwaracany nazwa_metody(argumenty)
throws KlasaWyjątku1, KlasaWyjątku2, ..., KlasaWyjatkun
{
  //treść metody
}
```

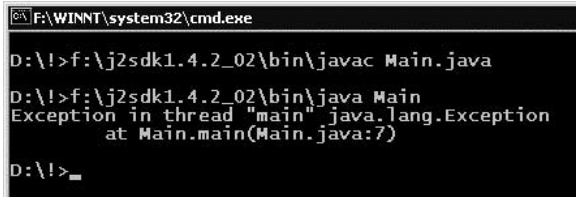
Zobaczmy, jak to wygląda w praktyce. Załóżmy, że mamy klasę `Main`, a w niej metodę `main`. Jedynym zadaniem tej metody będzie zgłoszenie wyjątku klasy `Exception`. Klasa taka jest widoczna na listingu 4.14.

Listing 4.14.

```
public
class Main
{
  public static void main (String args[])
  throws Exception
  {
        throw new Exception();
  }
}
```

W deklaracji metody `main` zaznaczyliśmy, że może ona generować wyjątek klasy `Exception` poprzez użycie instrukcji `throws Exception`. W ciele metody `main` została natomiast wykorzystana instrukcja `throw`, która jako parametr otrzymała nowy obiekt klasy `Exception`. Po uruchomieniu takiego programu na ekranie zobaczymy widok zaprezentowany na rysunku 4.12. Jest to najlepszy dowód, że faktycznie udało nam się zgłosić wyjątek.

Rysunek 4.12. *Zgłoszenie wyjątku klasy Exception*



Utworzenie obiektu wyjątku nie musi mieć miejsca bezpośrednio w instrukcji `throw`, można utworzyć go wcześniej, przypisać zmiennej obiektowej i dopiero tę zmienną wykorzystać jako parametr dla `throw`. Czyli zamiast pisać:

```
throw new Exception();
```

możemy równie dobrze zastosować konstrukcję:

```
Exception exception = new Exception();
throw exception;
```

W obu przedstawionych przypadkach efekt będzie identyczny, najczęściej korzystamy jednak z pierwszego zaprezentowanego sposobu.

Jeśli chcemy, aby zgłaszany wyjątek otrzymał komunikat, należy przekazać go jako parametr konstruktora klasy `Exception`, czyli napiszemy wtedy:

```
throw new Exception("komunikat");
```

lub:

```
Exception exception = new Exception("komunikat");
throw exception;
```

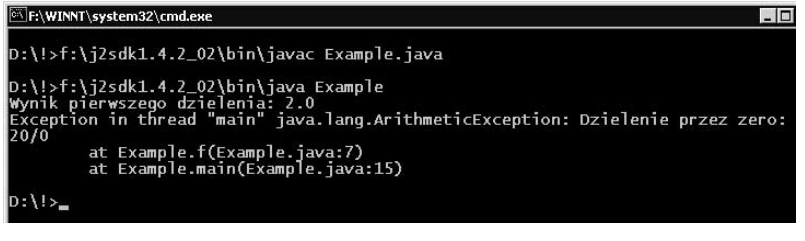
Oczywiście, można tworzyć obiekty wyjątków klas dziedziczących z `Exception`. Przykładowo: jeśli sami wykryjemy próbę dzielenia przez zero, być może zechcemy wygenerować nasz wyjątek, nie czekając, aż zgłosi go maszyna wirtualna. Spójrzmy na listing 4.15.

Listing 4.15.

```
public
class Example
{
  public double f(int liczba1, int liczba2)
  {
        if(liczba2 == 0)
              throw new ArithmeticException("Dzielenie przez zero: " + liczba1 +
                 "/" + liczba2);
        return liczba1 / liczba2;
  }
  public static void main(String args[])
  {
        Example example = new Example();
        int wynik = example.f(20, 10);
        System.out.println("Wynik pierwszego dzielenia: " + wynik);
        wynik = example.f(20, 0);
        System.out.println("Wynik drugiego dzielenia: " + wynik);
  }
}
```

W klasie `Example` jest zdefiniowana metoda `f`, która przyjmuje dwa argumenty typu `int`. Ma ona zwracać wynik dzielenia wartości przekazanej w argumencie `liczba1` przez wartość przekazaną w argumencie `liczba2`. Jest zatem jasne, że `liczba2` nie może mieć wartości zero. Sprawdzamy to, wykorzystując instrukcję warunkową `if`. Jeśli okaże się, że `liczba2` ma jednak wartość zero, za pomocą instrukcji `throw` wyrzucamy nowy wyjątek klasy `ArithmeticException`. W konstruktorze klasy przekazujemy komunikat informujący o dzieleniu przez zero. Podajemy w nim wartości argumentów metody `f`, tak by łatwo można było stwierdzić, jakie parametry spowodowały błąd.

W celu przetestowania działania metody `f` w klasie `Example` pojawiła się również metoda `main`. Tworzymy w niej nowy obiekt klasy `Example` i przypisujemy go zmiennej o nazwie `example`. Następnie dwukrotnie wywołujemy metodę `f`, raz przekazując jej argumenty równe `20` i `10`, drugi raz przekazując jej argumenty równe `20` i `0`. Spodziewamy się, że w drugim przypadku program zgłosi wyjątek `ArithmeticException` ze zdefiniowanym przez nas komunikatem. Faktycznie program zachowa się w taki właśnie sposób, co jest widoczne na rysunku 4.13.



Rysunek 4.13. *Zgłoszenie własnego wyjątku klasy ArithmeticException*

Ponowne zgłoszenie przechwyconego wyjątku

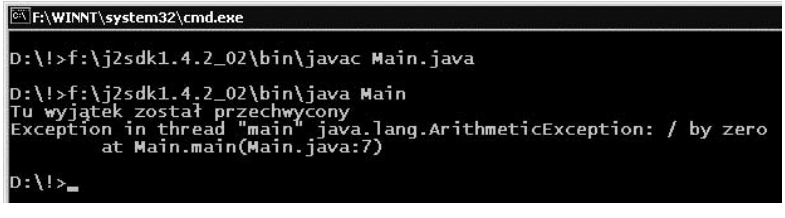
Wiemy już, jak przechwytywać wyjątki oraz jak samemu zgłaszać wystąpienie wyjątku. To pozwoli nam zapoznać się z techniką ponownego zgłaszania (potocznie: wyrzucania) już przechwyconego wyjątku. Jak pamiętamy, bloki `try...catch` można zagnieżdżać bezpośrednio, a także stosować je w przypadku kaskadowo wywoływanych metod. Jeśli jednak na którymkolwiek poziomie przechwytywaliśmy wyjątek, jego obsługa ulegała zakończeniu. Nie zawsze jest to korzystne zachowanie, czasami istnieje potrzeba, aby po wykonaniu naszego bloku obsługi wyjątek nie był niszczony, ale przekazywany dalej. Aby osiągnąć takie zachowanie, musimy zastosować instrukcję `throw`. Schematycznie wyglądałoby to następująco:

```
try{
  //instrukcje mogące spowodować wyjątek
}
catch(typTyjątku identyfikatorWyjątku){
  //instrukcje obsługujące sytuację wyjątkową
  throw indetyfikatorWyjątku
}
```

Listing 4.16 przedstawia, jak taka sytuacja wygląda w praktyce. W bloku `try` jest wykonywana niedozwolona instrukcja dzielenia przez zero. W bloku `catch` najpierw wyświetlamy na ekranie informację o przechwyceniu wyjątku, a następnie za pomocą instrukcji `throw` ponownie wyrzucamy (zgłaszamy) przechwycony już wyjątek. Ponieważ w programie nie ma już innego bloku `try...catch`, który mógłby przechwycić ten wyjątek, zostanie on obsłużony standardowo przez maszynę wirtualną. Dlatego też na ekranie zobaczymy widok zaprezentowany na rysunku 4.14.

Listing 4.16.

```
public
class Main
{
  public static void main (String args[])
  {
        try{
              int liczba = 10 / 0;
        }
        catch(ArithmeticException e){
              System.out.println("Tu wyjątek został przechwycony");
              throw e;
        }
  }
}
```



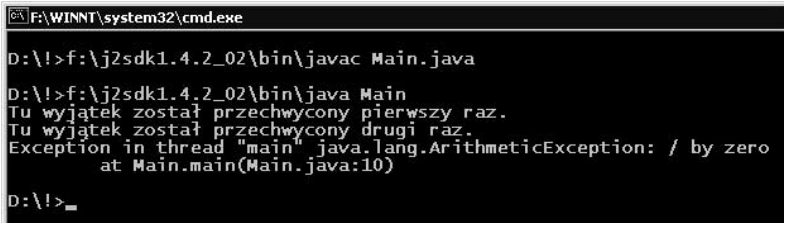
Rysunek 4.14. *Ponowne zgłoszenie raz przechwyconego wyjątku*

W przypadku zagnieżdżonych bloków try sytuacja wygląda analogicznie. Wyjątek przechwycony w bloku wewnętrznym i ponownie zgłoszony może być obsłużony w bloku zewnętrznym, w którym może być oczywiście zgłoszony kolejny raz itd. Jest to zobrazowane na listingu 4.17.

Listing 4.17.

```
public
class Main
{
  public static void main (String args[])
  {
        //tu dowolne instrukcje
        try{
              //tu dowolne instrukcje
              try{
                    int liczba = 10 / 0;
              }
              catch(ArithmeticException e){
                    System.out.println("Tu wyjątek został przechwycony pierwszy
                       raz.");
                    throw e;
              }
        }
        catch(ArithmeticException e){
              System.out.println("Tu wyjątek został przechwycony drugi raz.");
              throw e;
        }
  }
}
```

Mamy tu dwa zagnieżdżone bloki try. W bloku wewnętrznym zostaje wykonana nieprawidłowa instrukcja dzielenia przez zero. Zostaje ona w tym bloku przechwycona, a na ekranie zostaje wyświetlony komunikat o pierwszym przechwyceniu wyjątku. Następnie wyjątek jest ponownie zgłaszany. W bloku zewnętrznym następuje drugie przechwycenie, wyświetlenie drugiego komunikatu oraz kolejne zgłoszenie wyjątku. Ponieważ nie istnieje trzeci blok try...catch, ostatecznie wyjątek jest obsługiwany przez maszynę wirtualną. Po uruchomieniu zobaczymy widok zaprezentowany na rysunku 4.15.



Rysunek 4.15. *Przechwytywanie i ponowne zgłaszanie wyjątków*

W identyczny sposób będą zachowywały się wyjątki w przypadku kaskadowego wywoływania metod. Z sytuacją tego typu mieliśmy do czynienia w przypadku przykładu z listingu 4.8. W klasie Example były wtedy zadeklarowane cztery metody: main, f, g, h. Metoda main wywoływała metodę f, ta z kolei metodę g, a metoda g metodę h. W każdej

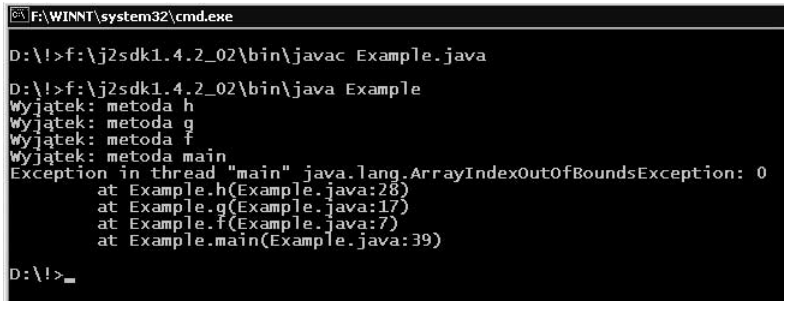
z metod znajdował się blok `try...catch`, jednak zawsze działał tylko ten najbliższy miejsca wystąpienia wyjątku (por. lekcja 20.). Zmodyfikujemy więc kod z listingu 4.8 tak, aby za każdym razem wyjątek był po przechwyceniu ponownie zgłaszany. Kod realizujący to zadanie jest przedstawiony na listingu 4.18.

Listing 4.18.

```
public
class Example
{
  public void f()
  {
        try{
              g();
        }
        catch(ArrayIndexOutOfBoundsException e){
              System.out.println("Wyjątek: metoda f");
              throw e;
        }
  }
  public void g()
  {
        try{
              h();
        }
        catch(ArrayIndexOutOfBoundsException e){
              System.out.println("Wyjątek: metoda g");
              throw e;
        }
  }
  public void h()
  {
        int[] tab = new int[0];
        try{
              tab[0] = 1;
        }
        catch(ArrayIndexOutOfBoundsException e){
              System.out.println("Wyjątek: metoda h");
              throw e;
        }
  }
  public static void main(String args[])
  {
        Example ex = new Example();
        try{
              ex.f();
        }
        catch(ArrayIndexOutOfBoundsException e){
              System.out.println("Wyjątek: metoda main");
              throw e;
        }
  }
}
```

Wszystkie wywołania metod pozostały niezmienione, w każdym bloku `catch` została natomiast dodana instrukcja `throw` ponownie zgłaszająca przechwycony wyjątek. Rysunek 4.16 pokazuje efekty uruchomienia przedstawionego kodu. Widać wyraźnie, jak wyjątek jest propagowany po wszystkich metodach, począwszy od metody `h` a skończywszy na metodzie `main`. Ponieważ w bloku `try...catch` metody `main` jest on ponownie zgłaszany, na ekranie jest także widoczna reakcja maszyny wirtualnej.

Rysunek 4.16. *Kaskadowe przechwytywanie wyjątków*



Tworzenie własnych wyjątków

Programując w Javie, nie musimy zdawać się na wyjątki systemowe, które dostajemy wraz z JDK. Nic nie stoi na przeszkodzie, aby tworzyć własne klasy wyjątków. Wystarczy więc, że napiszemy klasę pochodną pośrednio lub bezpośrednio z klasy `Throwable`, a będziemy mogli wykorzystywać ją do zgłaszania naszych własnych wyjątków. W praktyce jednak wyjątki wyprowadzamy z klasy `Exception` i klas od niej pochodnych. Klasa taka w najprostszej postaci będzie miała postać:

```
public
class nazwa_klasy extends Exception
{
  //treść klasy
}
```

Przykładowo możemy utworzyć bardzo prostą klasę o nazwie `GeneralException` (wyjątek ogólny) w postaci:

```
public
class GeneralException extends Exception
{
}
```

To w zupełności wystarczy. Nie musimy dodawać żadnych nowych pól i metod. Ta klasa jest pełnoprawną klasą obsługującą wyjątki, z której możemy korzystać w taki sam sposób, jak ze wszystkich innych klas opisujących wyjątki. Na listingu 4.19 jest widoczna przykładowa klasa `main`, która generuje wyjątek `GeneralException`.

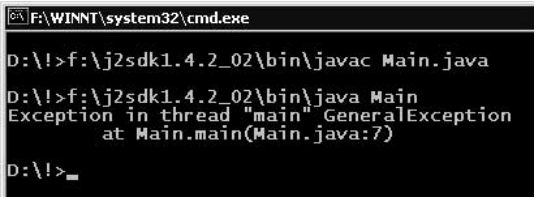
Listing 4.19.

```
public
class Main
{
  public static void main (String args[])
  throws GeneralException
```

```
  {
        throw new GeneralException();
  }
}
```

W metodzie `main` za pomocą instrukcji `throws` zaznaczamy, że metoda ta może zgłaszać wyjątek klasy `GeneralException`, sam wyjątek zgłaszamy natomiast przez zastosowanie instrukcji `throw`, dokładnie w taki sam sposób, jak we wcześniejszych przykładach. Na rysunku 4.17 jest widoczny efekt działania takiego programu, faktycznie zgłoszony został wyjątek nowej klasy: `GeneralException`.

Rysunek 4.17. *Zgłaszanie własnych wyjątków*



Własne klasy wyjątków można wyprowadzać również z klas pochodnych od `Exception`. Moglibyśmy np. rozszerzyć klasę `ArithmeticException` o wyjątek zgłaszany wyłącznie wtedy, kiedy wykryjemy dzielenie przez zero. Klasę taką nazwalibyśmy `DivideByZero➥Exception`. Miałaby ona postać widoczną na listingu 4.20.

Listing 4.20.

```
public
class DivideByZeroException extends ArithmeticException
{
}
```

Możemy teraz zmodyfikować program z listingu 4.15 tak, aby po wykryciu dzielenia przez zero był zgłaszany wyjątek naszego nowego typu, czyli `DivideByZeroException`. Klasa taka została przedstawiona na listingu 4.21.

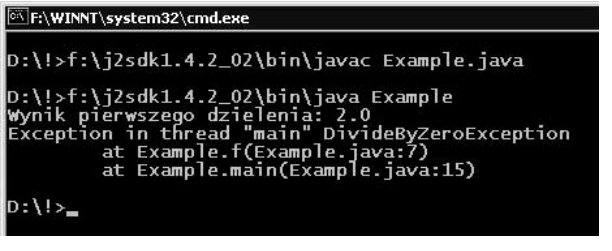
Listing 4.21.

```
public
class Example
{
  public double f(int liczba1, int liczba2)
  {
        if(liczba2 == 0)
              throw new DivideByZeroException();
        return liczba1 / liczba2;
  }
  public static void main(String args[])
  {
        Example example = new Example();
        double wynik = example.f(20, 10);
        System.out.println("Wynik pierwszego dzielenia: " + wynik);
```

```
            wynik = example.f(20, 0);
            System.out.println("Wynik drugiego dzielenia: " + wynik);
        }
    }
```

W stosunku do kodu z listingu 4.15 zmiany nie są duże, ograniczają się do zmiany typu zgłaszanego wyjątku w metodzie `f`. Zadaniem tej metody nadal jest zwrócenie wyniku dzielenia argumentu `liczba1` przez argument `liczba2`. W metodzie `main` dwukrotnie wywołujemy metodę `f`, pierwszy raz z prawidłowymi danymi i drugi raz z danymi, które spowodują wygenerowanie wyjątku. Efekt działania przedstawionego kodu jest widoczny na rysunku 4.18.

Rysunek 4.18. *Zgłoszenie wyjątku DivideByZeroException*



Zwróćmy jednak uwagę, że pierwotnie (listing 4.15) przy zgłaszaniu wyjątku parametrem konstruktora był komunikat (czyli wartość typu `String`). Tym razem nie mogliśmy jej umieścić, gdyż nasza klasa `DivideByZeroException` nie posiada konstruktora przyjmującego jako parametr obiektu typu `String`, a jedynie bezparametrowy konstruktor domyślny. Aby zatem można było przekazywać nasze własne komunikaty, należy dopisać do klasy `DivideByZeroException` odpowiedni konstruktor. Przyjmie ona wtedy postać widoczną na listingu 4.22.

Listing 4.22.

```
public
class DivideByZeroException extends ArithmeticException
{
  public DivideByZeroException(String str)
  {
        super(str);
  }
}
```

Teraz instrukcja `throw` z listingu 4.21 mogłaby przyjąć np. następującą postać:

```
throw new DivideByZeroException("Dzielenie przez zero: " + liczba1 + "/" + liczba2);
```

Sekcja finally

Do bloku `try` możemy dołączyć sekcję `finally`, która będzie wykonana zawsze, niezależnie od tego, co będzie działo się w bloku `try`. Schematycznie taka konstrukcja będzie wyglądała następująco:

```
try{
  //instrukcje mogące spowodować wyjątek
}
catch(){
  //instrukcje sekcji catch
}
finally{
  //instrukcje sekcji finally
}
```

Zgodnie z tym, co zostało napisane wcześniej, instrukcje sekcji `finally` są wykonywane zawsze, niezależnie od tego, czy w bloku `try` wystąpi wyjątek, czy nie. Obrazuje to przykład z listingu 4.23, który jest oparty na kodzie z listingów 4.20 i 4.21.

Listing 4.23.

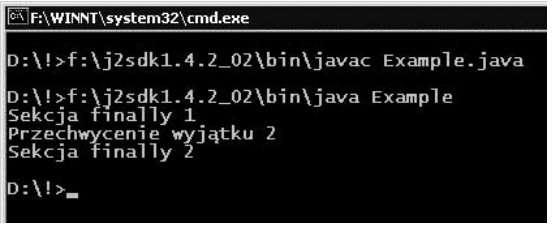
```
public
class Example
{
  public double f(int liczba1, int liczba2)
  {
        if(liczba2 == 0)
              throw new DivideByZeroException("Dzielenie przez zero: " + liczba1
                 + "/" + liczba2);
        return liczba1 / liczba2;
  }
  public static void main(String args[])
  {
        Example example = new Example();
        double wynik;
        try{
              wynik = example.f(20, 10);
        }
        catch(DivideByZeroException e){
              System.out.println("Przechwycenie wyjątku 1");
        }
        finally{
              System.out.println("Sekcja finally 1");
        }
        try{
              wynik = example.f(20, 0);
        }
        catch(DivideByZeroException e){
              System.out.println("Przechwycenie wyjątku 2");
        }
        finally{
              System.out.println("Sekcja finally 2");
        }

  }
}
```

Jest to znana nam klasa `Example` z metodą `f` wykonującą dzielenie przekazanych jej argumentów. Tym razem metoda `f` pozostała bez zmian w stosunku do wersji z listingu 4.21, czyli zgłasza błąd `DivideByZeroException`. Zmodyfikowaliśmy natomiast metodę

main. Oba wywołania metody zostały ujęte w bloki try...catch...finally. Pierwsze wywołanie nie powoduje powstania wyjątku, nie jest więc wykonywany pierwszy blok catch, jest natomiast wykonywany pierwszy blok finally. Tym samym na ekranie pojawi się napis Sekcja finally 1.

Drugie wywołanie metody f powoduje wygenerowanie wyjątku, zostaną zatem wykonane zarówno instrukcje bloku catch, jak i instrukcje bloku finally. Na ekranie pojawią się zatem dwa napisy: Przechwycenie wyjątku 2 oraz Sekcja finally 2. Ostatecznie wyniki działania całego programu będzie taki, jak ten zaprezentowany na rysunku 4.19.



Rysunek 4.19. *Blok finally jest wykonywany niezależnie od tego, czy pojawi się wyjątek*

Sekcję finally można zastosować również w przypadku instrukcji, które nie powodują wygenerowania wyjątku. Stosujemy wtedy instrukcję try...finally w postaci:

```
try{
  //instrukcje
}
finally{
  //instrukcje
}
```

Działanie jest takie samo jak w przypadku bloku try...catch...finally, to znaczy kod z bloku finally zostanie wykonany zawsze, niezależnie od tego, jakie instrukcje znajdą się w bloku try. Przykładowo: nawet jeśli w bloku try znajdzie się instrukcja return lub zostanie wygenerowany wyjątek, blok finally i tak zostanie wykonany. Obrazuje to przykład zaprezentowany na listingu 4.24.

Listing 4.24.

```
public
class Example
{
  public int f1()
  {
        try{
              return 0;
        }
        finally{
              System.out.println("Sekcja finally f1");
        }
  }
  public void f2()
  {
        try{
              int liczba = 10 / 0;
        }
```

```
            finally{
                  System.out.println("Sekcja finally f2");
            }
      }
      public static void main(String args[])
      {
            Example example = new Example();
            example.f1();
            example.f2();
      }
}
```

Ćwiczenia do samodzielnego wykonania

22.1. Napisz klasę `Example`, w której zostaną zadeklarowane metody `f` i `main`. W metodzie `f` napisz dowolną instrukcję generującą wyjątek `NullPointerException`. W metodzie `main` wywołaj metodę `f`, przechwyć wyjątek za pomocą bloku `try...catch`.

22.2. Zmodyfikuj kod z listingu 4.17 tak, aby generowany, przechwytywany i ponownie zgłaszany był wyjątek `ArrayIndexOutOfBoundsException`.

22.3. Napisz klasę o takim układzie metod, jak w przypadku klasy `Example` z listingu 4.18. W najbardziej zagnieżdżonej metodzie `h` wygeneruj wyjątek `ArithmeticException`. Przechwyć ten wyjątek w metodzie `g` i zgłoś wyjątek klasy nadrzędnej do `Arithmetic➥Exception`, czyli `RuntimeException`. Wyjątek ten przechwyć w metodzie `f` i zgłoś wyjątek nadrzędny do `RuntimeException`, czyli `Exception`. Ten ostatni wyjątek przechwyć w klasie `main`.

22.4. Napisz klasę wyjątku o nazwie `NegativeValueException` oraz klasę `Example`, która będzie z niego korzystać. W klasie `Example` napisz metodę o nazwie `f`, przyjmującą dwa argumenty typu `int`. Metoda `f` powinna zwracać wartość będącą wynikiem odejmowania argumentu pierwszego od argumentu drugiego. W przypadku jednak, gdyby wynik ten był ujemny, powinien zostać zgłoszony wyjątek `NegativeValueException`. Dopisz metodę `main`, która przetestuje działanie metody `f`.