CODE:ME

JAVA

OD PODSTAW

Materiały szkoleniowe

Części I - V

Paweł Apanasewicz

Spis treści

Rozdział 1.

Podstawy programowania w języku Java

1.1 Historia języka Java

Przyczyną powstawania nowych języków programowania najczęściej są zmiany w środowisku przetwarzania i zmiany w samym sposobie programowania. Właśnie te dwa powody legły u podstaw powstania Javy, a na jej sukces złożyły się zmiany w programowaniu z proceduralnego na obiektowe oraz szybko rosnąca popularność Internetu.

Java powstała w 1991 roku w firmie Sun Microsystems. Jej twórcami byli James Gosling, Patric Naughton, Chris Warth, Ed Franc i Mike Sheridan. Na początku dla Javy przyjęto nazwę Oak, zmieniono ją jednak w 1995 roku, ponieważ nazwa Oak była już używana przez firmę Oak Technology. Twórcom Javy przyświecał cel stworzenia języka programowania, który będzie umożliwiał tworzenie aplikacji uruchamialnych na różnych platformach – początkowo bowiem dedykowany był urządzeniom AGD. Dopiero po zwiększeniu popularności aplikacji o rozproszonej architekturze i upowszechnieniu Internetu, zauważono potencjał tego języka. Od tamtej pory Java i jej Aplety stały się bardzo popularne. Java stała się językiem, który zrewolucjonizował wytwarzanie aplikacji sieciowych oraz określił nowe standardy programowania.

1.2 Co jest sercem języka Java?

Sercem języka jest wirtualna maszyna Javy (JVM), która jest interpreterem kodu bajtowego. Kod bajtowy to zoptymalizowany zbiór instrukcji uruchamiania przez JVM. Wirtualna Maszyna Javy nie umożliwia programowi dostępu do zasobów komputera, na którym jest uruchamiany, co zapewnia bezpieczeństwo, natomiast kompilacja programów do kodu bajtowego zapewnia przenoszalność między platformami, to znaczy, na przykład, niezależność od systemu operacyjnego użytkownika.

Aby móc stworzyć program w języku Java, wymagane jest środowisko deweloperskie JDK (Java Developer Kit) oraz edytor tekstowy, najlepiej z funkcją wyróżniania składni. Na zajęciach używać będziemy Eclipse. Do uruchomienia skompilowanego programu wystarczy samo JRE (Java Runtime Environment), JRE zawarte jest już w JDK.

1.3 Podstawy języka Java

1.3.1 Pierwszy program

Poniżej przykład najprostszego programu w języku Java, który ma za zadanie wypisać na ekranie konsoli tekst „CODE:ME – Kurs Java SE”.

```
1 package pl.codeme;
2
3 public class First {
4
5    public static void main(String[] args) {
6        System.out.println("CODE:ME - Kurs Java SE");
7    }
8
9 }
```

Omówmy kod linia po linii:

Linia 1 Określenie w jakim pakiecie znajduje się uruchomiana klasa (do pakietów i klas wrócimy przy omawianiu programowania obiektowego)

Linia 3 Deklaracja klasy o nazwie `First` (klasy są podstawowym elementem w programowaniu obiektowym; na razie zapamiętajmy konstrukcję, reszta wyjaśni się przy omawianiu programowania obiektowego)

Linia 5 Deklaracja metody statycznej `main` z parametrem `args`, który jest tablicą parametrów podanych przy uruchomieniu. Klasy można uruchomić tylko jeżeli posiadają taką metodę

Linia 6 Ciało metody, tu wpisujemy kod, który chcemy uruchomić. W tym wypadku komenda `System.out.println(…)` wypisuje podany tekst na ekranie

Linia 7 Zakończenie deklaracji metody `main`

Linia 9 Zakończenie deklaracji klasy `First`

1.3.2 Zmienne i typy danych prymitywne

1.3.2.1 Zmienne

W programach do przechowywania danych używa się zmiennych. Zmienne to miejsca w pamięci oznaczone etykietą, którą nadaje programista podczas pisania programu. Np.

```
1 public static void main(String[] args) {
2     int liczba_calkowita = 10;
3     double liczba_zmiennoprzecinkowa;
4     byte zm1, zm2, zm3;
5     System.out.println(liczba_calkowita);
6 }
```

Od linii trzeciej do czwartej mamy przykłady deklaracji i deklaracji z definicją zmiennych.

Linia 2 Deklaracja i definicja zmiennej typu `int` i z nadaną etykietą `liczba_calkowita` oraz podstawienie pod nią wartości `10` (Deklaracja oznacza określenie typu zmiennej i nadanie jej etykiety, natomiast definicją jest podstawienie pod nią wartości)

Linia 3 Tylko deklaracja zmiennej typu `double` o nazwie `liczba_zmiennoprzecinkowa`

Linia 4 Deklaracja trzech zmiennych o etykietach `zm1`, `zm2` i `zm3` typu `byte`

1.3.2.2 Typy danych prymitywne

Typy liczbowe całkowite	

`Char`	Umożliwia przechowywanie znaków na 16 bitach (kodowanie UTF-16), czyli liczb od \u0000 (0) do \uffff (65 535). Każdy znak w tym kodowaniu ma przypisaną określoną wartość liczbową z tego przedziału.
	Przykład użycia:
	`char znak = 'A';`
	`char znak = 106; // litera j`

`Byte`	Służy do przechowywania liczb całkowitych z przedziału od -128 do 127 Przykład użycia: `byte liczba = 28;`

`Short`	Służy do przechowywania liczb całkowitych z przedziału od -32 768 do 32 767
	Przykład użycia:
	`short liczba = 3489;`
	`short liczba = 3_489;`

`Int`	Podstawowy typ całkowity z przedziału od -2 147 483 648 do 2 147 483 647; wszystkie operacje liczbowe na liczbach całkowitych zwracają ten typ
	Przykład użycia:
	`int liczba = 2567783;`
	`int liczba = 2_567_783;`

`Long`	Służy do przechowywania liczb całkowitych z przedziału od -9 223 372 036 854 775 808 do 9 223 372 036 854 775 807.
	Przykład użycia:
	`long liczba = 28456214098;`
	`long liczba = 28_456_214_098;`
	`long liczba = 284L; // rzutowanie na longa`

Typy liczbowe zmiennoprzecinkowe	

`Float`	Typ przeznaczony dla liczb zmiennoprzecinkowych 32 bitowych (IEEE 754) o pojedynczej precyzji
	Przykład użycia:
	`float liczba = 283;`
	`float liczba = 28.34F;`
	`float liczba = 4_123.452F;`

`Double`	Typ przeznaczony dla liczb zmiennoprzecinkowych 64 bitowych (IEEE 754) o podwójnej precyzji
	Przykład użycia:
	`double liczba = 134;`
	`double liczba = 28.31;`
	`double liczba = 2_376.348;`

Typy logiczne	

`Boolean`	Typ przyjmujący dwie wartości prawda (`true`) lub fałsz (`false`)
	Przykład użycia:
	`boolean warunek = true;`
	`boolean warunek = false;`

Liczby w języku Java można zapisywać nie tylko w postaci dziesiętnej, ale również w postaci szesnastkowej (int hex = 0x34ff;) i ósemkowej (int oct = 0146;).

W języku Java rozróżniane są także typy nieprymitywne, czyli tablicowe oraz obiektowe – będziemy o nich wspominać po zapoznaniu się z programowaniem obiektowym.

1.3.2.3 Rzutowanie typów danych

Rzutowaniem typów danych nazywamy konwersję jednego typu do drugiego. Istnieje konwersja niejawna i jawna.

Konwersja niejawna wykonuje się bez jawnej sygnalizacji (wymuszenia) i jest możliwa tylko jeżeli jest bezstratna, czyli dane po konwersji są takie same. Do konwersji niejawnej zaliczamy konwersję przy podstawianiu wartości, np.

```
1 public static void main(String[] args) {
2     int liczba1 = 10;
3     float liczba2 = liczba1; // niejawna konwersja int do float
4     System.out.println(liczba2);
5 }
```

W linii trzeciej jest konwersja niejawna (automatyczna) `int` do `float`. Jest to konwersja bezstratna, ponieważ liczbę 10 można zapisać zarówno w typie danych `int`, jak i `float`.

Konwersja w drugą stronę, z liczb zmiennoprzecinkowych na całkowite, jest konwersją stratną i nie może być wykonana automatycznie. Może natomiast być wymuszona, jeżeli możemy sobie pozwolić na stratę wartości po przecinku.

Przykład stratnej konwersji z liczby zmiennoprzecinkowej na liczbę całkowitą:

```
1 public static void main(String[] args) {
2     float liczba1 = 10.23F;
3     int liczba2 = (int)liczba1; // jawna konwersja float do int
4     System.out.println(liczba2);
5 }
```

W powyższym przykładzie po jawnej konwersji `liczba2` przyjęłaby wartość o 0,23 mniejszą, czyli 10.

1.3.3 Blok kodu i widoczność zmiennych

1.3.3.1 Blok kodu

Blokiem kodu w Javie nazywamy kod zapisany pomiędzy nawiasami klamrowymi (`{...blok kodu...}`). W programie może być wiele bloków kodu. Mogą być poprzedzane słowami kluczowymi i być częścią różnych konstrukcji i instrukcji w programie. Poniżej przykład kilku bloków kodu:

```
1  public class First { // start blok kodu 1
2
3      static { // start blok kodu 2
4          System.out.println("Blok kodu statycznego");
5      } // stop blok kodu 2
6
7      public static void main(String[] args) { // start blok kodu 3
8          for(int ix = 0; ix < 5; ix++) { // start blok kodu 4
9              if(ix == 3) { // start blok kodu 5
10                 System.out.println("Czwarta iteracja");
11             } // stop blok kodu 5
12         } // stop blok kodu 4
13
14         { // start blok kodu 6
15             System.out.println("Luzny blok kodu");
16         } // stop blok kodu 6
17     } // stop blok kodu 3
18
19 } // stop blok kodu 1
```

1.3.3.2 Widoczność zmiennych

Widoczność zmiennych uzależniona jest od miejsca ich deklaracji względem bloków kodu.

```
1  public static void main(String[] args) {
2      int ix = 3;
3      if(ix == 3) {
4          float zm = 1.2F;
5          System.out.println("1 - ix ma wartość " + ix);
6          System.out.println("1 - zm ma wartość " + zm);
7      }
8      System.out.println("2 - ix ma wartość " + ix);
9      System.out.println("2 - zm ma wartość " + zm); // błąd
10 }
```

Powyższy przykład pokazuje widoczność zmiennych w blokach kodu. W linii 2 i 4 są deklarowane i definiowane zmienne `ix` i `zm`. Zmienna `ix` jest deklarowana w bloku rozpoczynającym się w linii 1 i kończącym się w linii 10, natomiast zmienna `zm` w bloku rozpoczynającym się w linii 3 i kończącym się w linii 7. Zmienne zawsze są widoczne (czyli można ich używać w kodzie) w całym bloku, w którym zostały zadeklarowane oraz we wszystkich blokach stworzonych wewnątrz tego bloku. Analizując dalej przykład, zmienna `ix` jest widoczna w bloku od linii 1 do linii 10 oraz w bloku od linii 3 do linii 7, natomiast zmienna `zm` jest widoczna tylko w bloku od linii 3 do linii 7, ponieważ to w nim została zadeklarowana. Dlatego linia 9 jest oznaczona jako `błąd` – dla kompilatora zmienna `zm` nie istnieje w tym miejscu.

1.3.4 Operatory

Operatory arytmetyczne	
`+`	Suma (`int liczba = 2 + 2;`)
`-`	Różnica (`int liczba = 4 - 2;`)
`*`	Iloczyn (`int liczba = 2 * 2;`)
`/`	Iloraz (`int liczba = 2 / 2;`)
`%`	Modulo – reszta z dzielenia (`int liczba = 2 % 2;`)
`++`	Operator inkrementacji, zwiększa wartość o jeden (`zmienna++`)
`--`	Operator dekrementacji, zmniejsza wartość o jeden (`zmienna--`)
Operatory relacyjne (zwracają prawdę lub fałsz)	
`==`	Jest równe (`zmienna == 1`)
`!=`	Jest różne (`zmienna != 1`)
`>`	Jest większa (`zmienna > 1`)
`<`	Jest mniejsza (`zmienna < 1`)
`>=`	Jest większa równa (`zmienna >= 1`)
`<=`	Jest mniejsza równa (`zmienna <= 1`)
Operatory logiczne	
`&&`	`AND` iloczyn logiczny, wszystkie parametry muszą mieć wartość `true`
`||`	`OR` suma logiczna, wystarczy, że jeden z parametrów jest `true`
`!`	`NOT` negacja, zamienia na przeciwny
`^`	`XOR` alternatywa wykluczająca, zwraca `true`, jeżeli parametry są różne
Operatory podstawienia	
`=`	Operator podstawienia, `int liczba = 1;` podstawia pod zmienną wartość
`+=`	Operator sumy i podstawienia, `liczba += 1;` dodaje do zmiennej wartość
`-=`	Operator różnicy i podstawienia, `liczba -= 1;` odejmuje od zmiennej wartość
`*=`	Operator iloczynu i podstawienia, `liczba *= 1;` mnoży zmienną przez wartość
`/=`	Operator ilorazu i podstawienia, `liczba /= 1;` dzieli zmienną przez wartość
`%=`	Operator modulo i podstawienia, `liczba %= 3;` pod zmienną jest podstawiana reszta z dzielenia zmiennej przez wartość

1.3.5 Słowa kluczowe

`abstract`	Słowo używane przy deklaracji abstrakcyjnych klas i metod
`assert`	Tworzenie asercji
`boolean`	Określenie typu danych
`break`	Słowo używane w instrukcji `switch` oraz do przerywania pętli
`byte`	Określenie typu danych
`case`	Słowo używane w instrukcji `switch` dla wyróżnienia wartości testowanej zmiennej
`catch`	Słowo instrukcji `try...catch`
`char`	Określenie typu danych
`class`	Służy do deklaracji klasy
`const`	Nieużywane
`continue`	Pominięcie iteracji w pętlach
`default`	Ustawienie wartości domyślnej
`do`	Słowo rozpoczynające pętle `do...while`
`double`	Określenie typu danych
`else`	Słowo używane w instrukcji `if`
`enum`	Służy do deklaracji typu wyliczeniowego
`extends`	Słowo informujące o dziedziczeniu
`final`	Słowo używane przy klasach, parametrach metod, polach
`finally`	Słowo rozpoczyna blok zakończeniowy w instrukcji `try...catch`
`float`	Określenie typu danych
`for`	Rozpoczyna pętlę `for`
`goto`	Operacja już nieużywana
`if`	Rozpoczyna instrukcję `if`
`implements`	Dodawanie do klasy interfejsu
`import`	Importuje klasy
`instanceof`	Sprawdzanie typu zmiennych
`int`	Określenie typu danych
`interface`	Deklaracja interfejsu
`long`	Określenie typu danych
`native`	Deklaracja metody natywnej
`new`	Operator tworzący nową instancję klasy

`package`	Deklaracja przynależenia do pakietu
`private`	Kwalifikator dostępu
`protected`	Kwalifikator dostępu
`public`	Kwalifikator dostępu
`return`	Instrukcja
`short`	Określenie typu danych
`static`	Deklaracja elementów statycznych
`strictfp`	Obliczenia na liczbach zmiennoprzecinkowych zgodne z początkowymi wersjami
`super`	Odnoszenie się do obiektu po którym dziedziczymy
`switch`	Słowo rozpoczyna instrukcję `switch`
`synchronized`	Synchronizacja w programowaniu współbieżnym
`this`	Odnoszenie się obiektu
`throw`	Rzucanie wyjątków
`throws`	Deklaracja rzucanych wyjątków przez metodę
`transient`	Informacja o polu nieutrwalanym
`try`	Rozpoczęcie instrukcji `try…catch`
`void`	Typ zwracany przez metody, oznaczający nic
`volatile`	Określenie pola z niepewną wartością
`while`	Słowo używane przy pętlach `while` i `do…while`

Identyfikatory, czyli nazwy nadane klasom, interfejsom, typom wyliczeniowym (`enum`), metodom i zmiennym, nie mogą być takie same, jak słowa kluczowe. Nie mogą rozpoczynać się od liczby, ale, poza tym, liczby mogą w nazwie występować. W dowolnym miejscu mogą również zawierać znaki $ i _.

1.3.6 Biblioteki standardowe

Bibliotekami standardowymi nazywamy klasy, interfejsy itp. dołączone do języka Java. Są nimi np. `System.out`. Biblioteki standardowe umożliwiają tworzenie połączeń sieciowych, budowanie interfejsów graficznych czy też odczytywanie i zapisywanie plików.

Na kursie będziemy korzystać z kilku takich bibliotek. Ich liczba jest jednak zbyt duża, by omówienie wszystkich było możliwe. Zachęcamy do zapoznania się z pozostałymi dostępnymi bibliotekami, ich możliwościami i zastosowaniami.

Biblioteki są integralną częścią środowiska Javy, dlatego warto je znać. Do tego, jeśli myśli się o tworzeniu profesjonalnych programów, nie da się tego zrobić bez korzystania z bibliotek.

1.4 Instrukcje sterujące

1.4.1 Algebra Boole'a

Algebra Boole'a to algebra stosowana w logice, matematyce, informatyce i elektronice, a jej nazwa pochodzi od nazwiska angielskiego matematyka George'a Boole'a, który żył w pierwszej połowie XVIII wieku. Algebra ta wykonuje operacje na zbiorze dwuwartościowym (prawda, fałsz). W języku Java mocno związany jest z nią jest typ `boolean`. W języku Java udostępnione są dwie grupy operatorów logicznych: jednoargumentowe i dwuargumentowe.

1.4.1.1 Jednoargumentowe operatory logiczne

Do grupy jednoargumentowych operatorów logicznych zaliczany jest jeden operator negacji, który w języku Java oznaczany jest `!`. Poniższa tabela przedstawia wyniki jego stosowania:

A	!A
0	1
1	0

1.4.1.2 Dwuargumentowe operatory logiczne

Koniunkcja (iloczyn logiczny, AND), oznaczana w języku Java jako `&&`:

A	B	A && B
0	0	0
0	1	0
1	0	0
1	1	1

Alternatywa (suma logiczna, OR), oznaczana w języku Java jako `||`:

A	B	A || B
0	0	0
0	1	1
1	0	1
1	1	1

Suma modulo 2 (suma symetryczna, XOR), oznaczana w języku Java jako `^`:

A	B	A ^ B
0	0	0
0	1	1
1	0	1
1	1	0

Ekwiwalencja (równoważność), oznaczana w języku Java jako `==`:

A	B	A == B
0	0	1
0	1	0
1	0	0
1	1	1

Dysjunkcja (NAND) – operator niezaimplementowany bezpośrednio w języku Java, natomiast osiągnąć go można używając skrótowego zapisu `if` w postaci `a ? !b : true`, gdzie a i b są zmiennymi typu `boolean`.

A	B	A ? !B : 1
0	0	1
0	1	1
1	0	1
1	1	0

Binegacja (NOR) – operator niezaimplementowany bezpośrednio w języku Java, ale można go uzyskać stosując odpowiednią kombinację dostępnych operatorów logicznych `!(a || b)` lub `!a && !b`. Zmienne a i b są zmiennymi typu `boolean`.

A	B	!A && !B
0	0	1
0	1	0
1	0	0
1	1	0

1.4.2 Instrukcje warunkowe

Instrukcje warunkowe służą do „podejmowania” przez program decyzji, które operacje ma wykonać.

1.4.2.1 Instrukcja `if`

```
1  public static void main(String[] args) {
2      int liczba = 10;
3      if(liczba < 10) {
4          System.out.println("mniejsze niż 10");
5      } else if(liczba == 10) {
6          System.out.println("równe 10");
7      } else {
8          System.out.println("inny przypadek");
9      }
10 }
```

Powyżej przedstawiono konstrukcję instrukcji `if`, umożliwiającej sprawdzanie warunku logicznego podanego w nawiasie. Pozwala ona również na dodanie dowolnej liczby sekcji `else if`, służących sprawdzaniu kolejnych przypadków. Instrukcja ta daje także możliwość obsłużenia wariantu, w którym żaden ze sprawdzanych warunków
nie został spełniony – służy do tego sekcja `else`.

1.4.2.2 Instrukcja `switch`

```
1 public static void main(String[] args) {
2       int liczba = 10;
3       switch(liczba) {
4           case 10:
5               System.out.println("Równe 10");
6               break;
7           case 20:
8               System.out.println("Równe 20");
9               break;
10          case 30:
11          case 40:
12              System.out.println("Równe 30 lub 40");
13              break;
14          default:
15              System.out.println("inna wartość");
16      }
17  }
```

Instrukcja `switch` działa podobnie, jak `if`, ma inną składnię w postaci listy wartości, jakie może przyjąć testowana zmienna (w powyższym przykładzie o nazwie `liczba`).

Powyższy przykład sprawdza, jaką wartość przyjmuje zmienna `liczba` i wykonuje odpowiednią operację. Listę spodziewanych wartości podajemy po słowie `case`, słówko `break` informuje nas natomiast, w którym miejscu kończą się wykonywane instrukcje. W linii 10 i 11 jest przykład wykonania operacji, jeżeli któraś z podanych wartości
(30, 40) będzie równa zmiennej `liczba`. Sekcja `default` wykona się, jeżeli żadna z podanych wartości nie zostanie dopasowana.

Warto zapamiętać, że w instrukcji `switch` najważniejszą rolę odgrywa słowo `break`, ponieważ, jeżeli któryś `case` zostanie dopasowany, to zostaną wywołane wszystkie komendy zawarte w `swich`, aż do napotkania słówka `break` lub zakończenia instrukcji.

Instrukcje `if` i `switch` są wymienne, a wybór, której użyć, najczęściej podyktowany jest czytelnością kodu.

1.4.3 Pętle

Pętle są instrukcjami, które umożliwiają wielokrotne wykonanie sekwencji kodu, uzależniając liczbę takich wykonań od warunku logicznego. Język Java oferuje trzy pętle: `for`, `while` i `do...while`. Wszystkie pętle są zamienne, ale każda z nich daje pewne ułatwienia, które można wykorzystać.

1.4.3.1 Pętla `for`

Pętla `for` jest sterowana indeksem iteracji. W poniższym przykładzie jest to zmienna `ix`.

Blok sterujący pętlą `for` składa się z trzech elementów: zmiennej sterującej, warunku wykonania oraz iteracji. Zmienna sterująca `int ix = 0;` jest inicjalizacją i deklaracją zmiennej. Kolejny element `ix < 10` to warunek wykonywania pętli, a więc pętla będzie się wykonywać, dopóki zmienna `ix` będzie mniejsza od 10. Ostatni element to iteracja, czyli operacja, jaka ma się wykonać przy każdym uruchomieniu bloku kodu w pętli. `ix++` to użycie operatora inkrementacji na zmiennej sterującej, czyli przy każdym wykonaniu zwiększamy wartość zmiennej sterującej o jeden. Przy pierwszej iteracji zmienna `ix` będzie miała wartość 0 i zostanie zwiększona o 1 po wykonaniu bloku kodu. Przy kolejnej iteracji zmienna `ix` będzie miała wartość 1, zostanie zwiększona o jeden itd. aż osiągnie wartość 10 i pętla zostanie przerwana.

```
1 public static void main(String[] args) {
2     for(int ix = 0; ix < 10; ix++) {
3         System.out.println("Indeks: " + ix);
4     }
5 }
```

Efektem tego przykładu jest wypisanie dziesięciu linii tekstu:

```
Indeks: 1
Indeks: 2
...
```

Oprócz zaprezentowanej podstawowej wersji pętli `for`, mamy kilka wariacji:

Pętla `for` z dwiema zmiennymi sterującymi:

```
1 for(int i = 1, j = 10; i <= 10; i++, j--) {
2     System.out.println("Indeks: " + i + " i " + j);
3 }
```

Pętla z zewnętrzną zmienną sterującą:

```
1 int i = 0;
2 for(; i <= 10; i++) {
3     System.out.println("Indeks: " + i);
4 }
```

Pętla z iteracją wewnątrz pętli:

```
1 int i = 0;
2 for(; i <= 10;) {
3     System.out.println("Indeks: " + i);
4     i = i + 2;
5 }
```

Nieskończona pętla for:

```
1 for(;;) {
2     System.out.println("TEKST");
3 }
```

Można z tych wariacji korzystać w zależności od potrzeby. Istnieje jeszcze jeden wariant pętli `for`, używany dla Obiektów iterowalnych, ale wrócimy do tego po poznaniu programowania obiektowego.

1.4.3.2 Pętla `while`

Pętla `while` zbudowana jest z instrukcji `while`, warunku wykonania oraz bloku kodu. Poniższy przykład poniżej pokazuje, jak można wykorzystać zmienną typu `boolean` jako zmienną sterującą pętlą. Cały proces sterujący pętlą obsługuje instrukcja warunkowa `if`, rozpoczynająca się w linii 5, a jej zadaniem ustawianie odpowiedniej wartości dla zmiennej sterującej `czyWykonywac`.

```
1  public static void main(String[] args) {
2      boolean czyWykonywac = true;
3      int ix = 0;
4      while(czyWykonywac) {
5          if(ix > 10) {
6              czyWykonywac = false;
7              System.out.println("Kończę ix > 10");
8          } else {
8              System.out.println("Jeszcze wykonuję ix = " + ix);
10         }
11         ix++;
12     }
13 }
```

Aby uzyskać nieskończoną pętlę `while`, wystarczy zastąpić warunek logiczny wartością `true`.

1.4.3.3 Pętla do...while

Pętla do...while różni się od pętli for i while tym, że warunek w niej sprawdzany jest po wykonaniu bloku kodu, kiedy w dwóch poprzednich najpierw sprawdzany jest warunek, a dopiero, gdy jest spełniony, następuje wykonanie bloku kodu. Podsumowując, używając pętli do...while możemy być pewni, że blok kodu zostanie uruchomiony przynajmniej raz.

```
1    public static void main(String[] args) {
2        boolean czyWykonywac = false;
3        int ix = 0;
4        do {
5            switch(ix) {
6                case 0:
7                    czyWykonywac = true;
8                    System.out.println("Wykonuję się 1 raz");
9                    break;
10               case 5:
11                   czyWykonywac = false;
12                   System.out.println("Wykonuję się ostatni raz");
13                   break;
14               default:
15                   System.out.println("Wykonuję się jeszcze raz");
16           }
17           ix++;
18       } while(czyWykonywac);
19   }
```

Powyższy przykład pokazuje, jak można wysterować pętlą do...while instrukcją switch. W pierwszej iteracji zmienna ix ma wartość 0, natomiast zmienna sterująca wartość false, co, w przypadku pętli for czy while, nie spowodowałoby ich uruchomienia. Jednak pętla do...while uruchamia blok kodu zanim sprawdzi warunek, co oznacza, że dla ix równego 0 wartość zmiennej sterującej czyWykonywac zostaje zmieniona w switch-u na true, następnie zmienna ix jest zwiększana o 1 przy każdej iteracji, dzięki czemu dopiero przy piątej iteracji wpadamy do switch-a w linii 5, zmieniamy zmienną sterującą na false i kończymy wykonywanie się pętli.

1.4.3.4 Działanie słów `break` i `continue`

Słowo kluczowe `break`, oprócz poznanego zastosowania w instrukcji `switch` do ograniczania wykonywanego kodu w dopasowanych `case`-ach, używane jest w pętlach do natychmiastowego ich przerwania. Napotkane słowo `break` w bloku kodu pętli przerywa jej działanie.

Słówko `continue`, znajdujące się w bloku kodu pętli, powoduje przerwanie wykonywania kodu w obecnej iteracji i rozpoczęcie nowej iteracji.

Rozdział 2.

Praca w zespole

2.1 Maven

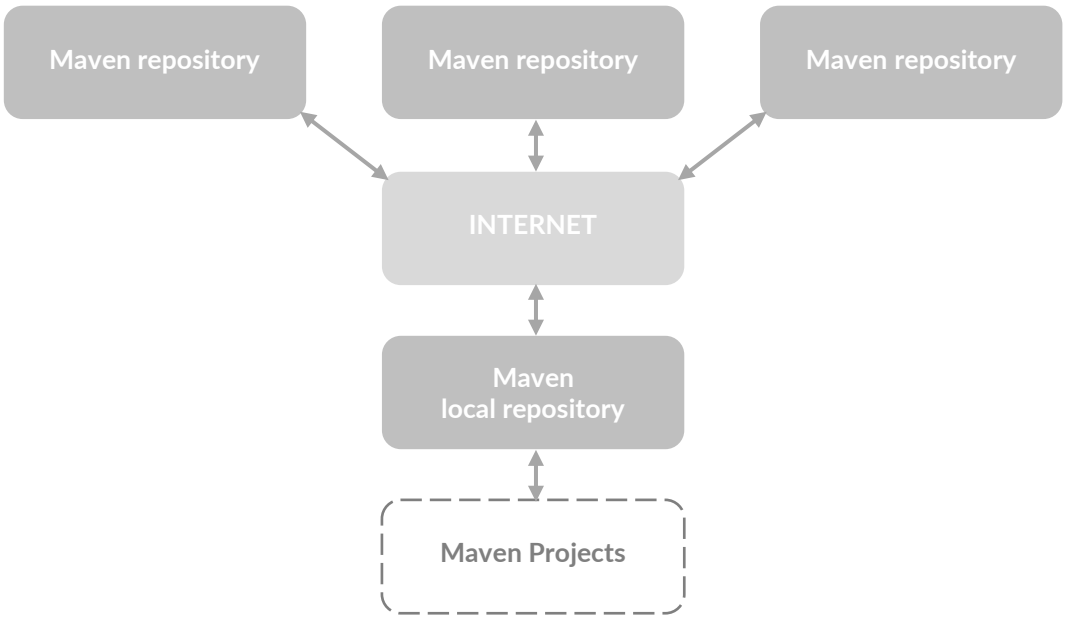
2.1.1 Do czego przydają się repozytoria bibliotek?

Wiele funkcjonalności, jak parsowanie XML-a czy różne rodzaje połączeń sieciowych, powtarza się w wielu aplikacjach. Tworzenie za każdym razem obszernych bloków kodu jest bardzo czasochłonne, a ze względów, na przykład, sprzętowych czy przez sposób działania połączeń sieciowych, funkcjonalności te będą za każdym razem wyglądały niemal identycznie. Dlatego pewne rozwiązania zostają zamknięte w biblioteki i udostępnione, często nieodpłatnie, na użytek programistów na całym świecie. Tak stworzone biblioteki, są następnie rozszerzane kolejnymi zestawami bibliotek o kolejne funkcjonalności. Korzystanie z gotowych rozwiązań znacznie przyspiesza pracę, jednak za każdym razem trzeba pamiętać, że biblioteki są aktualizowane, a te od nich zależne nie zawsze będą w pełni współpracowały z nowszymi wersjami. Ręczne gromadzenie i aktualizowanie w projektach potrzebnych bibliotek z czasem stało się mozolną, pochłaniającą cenny czas pracą.

By usprawnić tworzenie aplikacji w oparciu o biblioteki, powstały repozytoria bibliotek. Ich celem jest w prosty sposób udostępnić bibliotekę w pożądanej przez nas wersji
i dostosowanych do niej pozostałych, od których jest zależna.

2.1.2 Jak działa Maven?

Maven jest narzędziem, które, oprócz dostępu do repozytoriów, umożliwia również końcowe kompilacje aplikacji, umieszczanie tych aplikacji w lokalnych repozytoriach i ich wersjonowanie.



Tworząc projekt mavenowy, możemy czerpać z naszego lokalnego repozytorium, w którym znajdują się wszystkie ściągnięte przez nas biblioteki, oraz ze wszystkich naszych projektów. Projekty mavenowe mają dwie podstawowe cechy: specjalną strukturę katalogów oraz plik `pom`, w którym zdefiniowane są ustawienia naszego projektu, dołączone biblioteki oraz sposób kompilacji. Aby rozpocząć pracę z mavenem, wystarczy go pobrać i rozpakować. Bardzo często edytory kodu, takie jak Eclipse, wspierają Mavena i zawierają go w sobie.

Struktura katalogów w projekcie mavenowym została przedstawiona poniżej:

```
my-app
|-- pom.xml
`-- src
    |-- main
    |   |-- java
    |   |   `-- com
    |   |       `-- mycompany
    |   |           `-- app
    |   |               `-- App.java
    |   `-- resources
    |       `-- META-INF
    |           |-- application.properties
    `-- test
        |-- java
        |   `-- com
        |       `-- mycompany
        |           `-- app
        |               `-- AppTest.java
        `-- resources
            `-- test.properties
```

W głównym katalogu projektu przechowujemy plik `pom.xml`, natomiast kod aplikacji przechowywany jest w katalogu `src`. W katalogu `src` znajdują się dwa podkatalogi: `main` i `test`. W katalogu `main` trzymany jest kod aplikacji, natomiast w katalogu `test` kody testów jednostkowych. Każdy z tych katalogów zawiera dwa podkatalogi `java` i `resources`: w pierwszym trzymany jest kod javowy, natomiast w drugim pliki, które nie są kodem, ale są wykorzystywane w aplikacji, np. pliki konfiguracyjne.

Więcej szczegółów można doczytać na stronie projektu Maven pod adresem https://maven.apache.org/guides/getting-started/index.html.

Rozdział 3.

Programowanie obiektowe

3.1 Klasa

3.1.1 Czym jest klasa?

Klasa jest definicją typu obiektowego, a dokładniej opisem pól i metod, z których będzie można korzystać, jeśli na bazie tej klasy zostanie utworzona instancja obiektu. W języku Java jest to podstawowy element programu – każda klasa znajduje się w jednym pliku z rozszerzeniem .java, który nazywa się tak, jak klasa, np. klasa App będzie w pliku App.java.

Nową klasę definiujemy przez użycie słowa kluczowego `class`. Nazwę klasy rozpoczynamy wielką literą.

Poniżej przykład deklaracji pustej klasy ze statyczna metodą uruchamialną `main`:

```
1 public class App {
2
3     public static void main(String[] args) {
4         // TODO Auto-generated method stub
5     }
6 }
```

3.1.2 Pakiety i nazwa klasy

Pakiety w języku Java służą do utrzymywania porządku i ładu w tworzonej aplikacji, a więc dzielenie klas aplikacji na grupy związane z poszczególnymi funkcjonalnościami. Fizycznie każdy pakiet reprezentuje katalog na dysku stworzony w miejscu, gdzie umieszczamy źródła aplikacji i tak np. jeżeli naszą klasę `App` umieścimy w pakiecie `pl.codeme.oop`, a nasz katalog źródłowy aplikacji znajduje się pod ścieżką C:\codeme\src, to nasza klasa `App` w pakiecie `pl.codeme.oop` będzie na dysku pod ścieżką C:\codeme\src\pl\codeme\oop\App.java.

Należy pamiętać, że nazwy klas w języku Java zawsze są związane z pakietami, czyli nasza klasa `App` tak naprawdę nazywa się `pl.codeme.oop.App`. To pozwala nam na tworzenie klas o takiej samej nazwie (pliku) w różnych pakietach. Inaczej, po stworzeniu jednej klasy `App`, nie moglibyśmy stworzyć więcej klas o takiej nazwie. Do deklaracji, w jakim pakiecie leży klasa, służy słówko `package` – umieszczamy je zawsze na początku pliku.

```
1 package pl.codeme.oop;
2
3 public class App {
4
5     public static void main(String[] args) {
6         // TODO Auto-generated method stub
7     }
8 }
```

3.1.3 Enkapsulacja i składowe klasy

Enkapsulacja (hermetyzacja) to jedno z podstawowych zagadnień w programowaniu obiektowym. Celem enkapsulacji jest zamykanie danych i funkcjonalności, które są ze sobą powiązane. Dane są przechowywane w polach, natomiast funkcjonalności zamykane są w metodach. Klasa jest właśnie taką instrukcją, jakiego typu pola mogą być przechowywane i jakie czynności mogą być wykonywane. Dodatkowo określa się, które pola i metody są do użytku wewnętrznego, a z których mogą korzystać i do czego będą miały dostęp inne obiekty – o tym mówią kwalifikatory dostępu.

```
1  public class Auto {
2
3      float pojemnosc; // deklaracja pola
4      int masa = 1800; // deklaracja i definicja pola
5      int moc;
6      int predkosc;
7
8      void obliczPredkosc() { // deklaracja I definicja metody
9          … // kod metody
10     }
11
12     void napraw() {
13         … // kod metody
14     }
15 }
```

3.1.4 Kwalifikatory `public` i `private`

Kwalifikatory dostępowe `public` i `private` służą do określania, które elementy klasy są do użytku wewnętrznego (`private`), a z których mogą korzystać obiekty zewnętrzne. W języku Java występuje jeszcze jeden kwalifikator, `protected`, ale omówimy go w dalszej części, przy dziedziczeniu. Jeśli nie podamy żadnego kwalifikatora dostępu, element klasy będzie publiczny. Poniżej nasza klasa Auto z kwalifikatorami dostępu.

```
1  public class Auto {
2
3      private float pojemnosc; // deklaracja pola
4      private int masa = 1800; // deklaracja i definicja pola
5      private int moc;
6      private int predkosc;
7
8      private  void obliczPredkosc() { // definicja metody
9          … // kod metody
10     }
11
12     public void napraw() {
13         … // kod metody
14     }
15 }
```

Po dodaniu kwalifikatorów, widzimy, że wszystkie pola są prywatne i nie ma do nich dostępu z zewnątrz. Metoda `obliczPredkosc` również jest tylko do użytku wewnętrznego, natomiast metoda `napraw` jest dostępna dla obiektów zewnętrznych.

3.1.5 Przeciążanie metod

Przy definicji metody możemy ustalić parametry wchodzące do metody określając typ i nazwę każdego parametru, np. dodajmy do naszej klasy metodę `zatankuj(int paliwo)`, gdzie parametr `paliwo` określa, ile paliwa wlewamy do baku. Wyobraźmy sobie teraz sytuację tankowania do pełna – w takim przypadku ten parametr nie byłby nam potrzebny, ponieważ tankowalibyśmy dokładnie tylko, ile się zmieści. W języku Java nie ma możliwości definiowania parametrów metod z domyślną wartością po to, żeby ich nie podawać, ale możemy przeciążyć metodę, czyli zadeklarować i zdefiniować nową metodę o tej samej nazwie, lecz o innej liczbie parametrów lub o takiej samej liczbie, lecz innego typu. W tym celu stworzymy metodę `zatankuj()` bez parametrów i dzięki temu możemy tankować do pełna lub określoną ilość paliwa. Poniżej kod przedstawiający opisaną sytuację.

```
1 public void zatankuj(int paliwo) {
2     pojemnoscBaku += paliwo;
3 }
4
5 public void zatankuj() {
6     zatankuj(pojemnoscBaku - iloscPaliwa);
7 }
```

3.2 Różnica między klasą i obiektem

Podstawową różnicą miedzy klasą i obiektem jest to, że klasa to właściwie tylko informacja, jak ma wyglądać obiekt utworzony na jej podstawie. Można sobie wyobrazić to w ten sposób, że mamy opis techniczny, jak wyprodukować samochód, a więc klasę, a na bazie tego opisu wyprodukowaliśmy samochód, czyli obiekt utworzony na bazie klasy. Posługując się terminologią programowania obiektowego, opisując powyższą sytuację, powiedzielibyśmy, że stworzyliśmy nową instancję obiektu typu obiektowego `Auto`.

3.2.1. Konstruktor i operator `new`

Konstruktor jest specjalną metodą w klasie jest ona wywoływana przy tworzeniu nowej instancji obiektu na podstawie klasy. Definicja konstruktora różni się od definicji zwykłej metody w klasie, a mianowicie konstruktor to metoda nazywająca się tak samo jak klasa, nie zwracająca żadnej wartości, mogąca mieć dowolną liczbę parametrów, jak normalne metody. Poniżej przykładowy konstruktor naszej klasy Auto.

```
1 public Auto(int masa, float pojemnosc) {
2     this.pojemnoscBaku = 80;
3     this.masa = masa; // ustawienie pola z masą auta
4     this.pojemnosc = pojemnosc; // ustawienie pola z poj. auta
5     moc = (int)(masa / pojemnosc); // obliczenie mocy
6     obliczPredkosc(); // wywołanie metody obliczPredkosc
7 }
```

Aby stworzyć nową instancję obiektu z jakiejś klasy, używamy operatora `new`, przy użyciu którego wywoływany jest konstruktor. Poniżej przykład inicjalizacji nowego obiektu typu Auto i podstawienie go pod zmienną ford.

```
1 Auto ford = new Auto();
```

3.2.2 Referencja

Gdy tworzone są zmienne w oparciu o typy prymitywne, przekazanie tej zmiennej do metody jako parametr powoduje przekazanie tej zmiennej przez wartość, czyli wewnątrz metody jest tworzona kopia zmiennej i, jeżeli zmienimy wewnątrz metody wartość przekazanej zmiennej, nie wpłynie to na zmienną poza metodą, ponieważ są to dwie inne zmienne. Poniższy przykład pokazuje tę sytuację w linii 8 – przekazujemy do metody metoda zmienną `zm`, która ma wartość 2, następnie wewnątrz metody zmieniamy wartość przekazanego parametru na 4 (linia 2) lecz, gdy wypiszemy po tym wszystkim wartość zmiennej `zm` (linia 9), otrzymamy 2, ponieważ przy typach prymitywnych zmienna `zm` wewnątrz metody jest zupełnie inna zmienna niż w metodzie main.

```
1  public static void metoda(int zm) {
2      zm = 4;
3      System.out.println(zm); // na ekran 4
4  }
5
6  public static void main(String[] args) {
7      int zm = 2;
8      metoda(zm); // wypis na ekran z metody
9      System.out.println(zm); // na ekran 2
10 }
```

Przy zmiennych obiektowych przekazywanie nie odbywa się przez wartość, tylko przez referencję, to znaczy przekazując zmienną obiektową przekazujemy tylko adres tej zmiennej w pamięci, a nie jest tworzona nowa instancja tej zmiennej. Przekazany jest niejako „uchwyt" do niej i, jeżeli zmienimy wartość wewnątrz metody, to zmienimy wartość również poza tą metodą. Poniższy przykład pokazuje tę sytuację. Jako obiekt tworzymy tablicę jedno elementową.

```
1  public static void metoda(int[] zm) {
2      zm[0] = 4;
3      System.out.println(zm[0]); // na ekran 4
4  }
5
6  public static void main(String[] args) {
7      int[] zm = new int[1];
8      zm[0] = 2;
9      metoda(zm); // wypis na ekran z metody
10     System.out.println(zm[0]); // na ekran 4
11 }
```

Należy zapamiętać, że wszystkie zmienne typu obiektowego przekazywane są przez referencję, natomiast zmienne typu prymitywnego przez wartość. Wyjątkiem są jedynie zmienne typu String – String jest typem obiektowym, ale traktuje się go jako prymitywny.

3.2.3 Elementy statyczne i użycie `this`

Elementy statyczne

Pola i metody mogą być w klasie zadeklarowane jako statyczne. Podstawowe różnice między statycznymi a pozostałymi elementami klasy to:

1. Nie jest wymagane utworzenie instancji obiektów, aby można się do nich odwołać. Odwołujemy się po kropce po nazwie klasy: `Klasa.poleStatyczne` lub `Klasa.metodaStatyczna()`.
2. Pola statyczne są wspólne dla wszystkich obiektów stworzonych na bazie klasy. Jeśli pole statyczne jest publiczne, jest ono wspólne dla wszystkich obiektów w programie. W takim przypadku, jeżeli zadeklarowaliśmy pole statyczne publiczne, to każda zmiana wartości tego pola jest widoczna dla wszystkich obiektów
   w programie.
3. Metody statyczne mogą korzystać tylko z pól statycznych klasy. Wynika to z faktu, że niestatyczne pola i metody wymagają utworzenia instancji obiektu, aby można było się do nich odwołać.

Pola i metody statyczne mogą posiadać wszystkie kwalifikatory dostępu i wszystkie zależności, jakie język Java daje elementom niestatycznym, takie jak przesłanianie, przeciążenia itp.

Przykład użycia elementów statycznych w klasie:

```
1  public class App {
2
3      private static Auto auto;
4      public static String rejestracja;
5
6      static { // rozpoczęcie bloku statycznego
7          rejestracja = "GA 6783k";
8          auto = new FordMustang();
9      }
10
11     public static void jedziemyAutem() {
12         auto.jedzie();
13     }
14
15 }
```

W linii 6 rozpoczyna się blok statyczny. Wszystko, co w jego obszarze zostanie uruchomione po załadowaniu do programu pliku z tą klasą, można traktować jako swoisty konstruktor elementów statycznych.

Z elementami statycznymi związany jest też import elementów statycznych, który umożliwia korzystanie z elementów statycznych bez potrzeby pisania nazwy klasy, w której ten element statyczny się znajduje. Taki zapis ma postać `elementStatyczny` zamiast `Klasa.elementStatyczny`.

Poniższy przykład przedstawia import elementu statycznego:

```
import static pl.codeme.jse.Klasa.elementStatyczny;
```

Elementy obiektu i słowo `this`

Wszystkie elementy klasy niepoprzedzone słówkiem `static` są elementami, do których mamy dostęp tylko wtedy, gdy zostanie stworzona instancja obiektu na podstawie klasy, w której się znajdują.

Przed powstaniem obiektu, czyli przed wywołaniem konstruktora, możemy odnosić się tylko do elementów statycznych. Po stworzeniu obiektu mamy możliwość odniesienia się do pól i metod klasy, na podstawie której obiekt został stworzony. Każdy obiekt ma swoje niezależne pola i metody, które na tych polach operują.

Wewnątrz obiektu mamy możliwość odnoszenia się do samego siebie. Służy do tego słówko `this`, dzięki któremu mamy możliwość przekazania samego siebie jako parametru w metodach, odnoszenia się do pól obiektu, jeżeli zmienne lokalne nazywają się tak samo jak pola, oraz wewnątrz obiektu do jego elementów.

```
1  public class Kalkulator {
2
3      private Obliczenie obliczenie;
4      public int liczba1;
5      public int liczba2;
6
7      public Kalkulator(int liczba1, int liczba2) {
8          obliczenie = new Obliczenie(this);
9          this.liczba1 = liczba1;
10         this.liczba2 = liczba2;
11     }
12
13     public float oblicz(String operator) {
14         return obliczenie.oblicz(operator);
15     }
16
17 }
```

Powyższy przykład prezentuje użycie słowa `this` wewnątrz obiektu. W linii 8 mamy przekazanie samego siebie w parametrze do konstruktora obiektu `Obliczenie`, natomiast linie 9 i 10 to odwołanie się przez `this` do elementów obiektu.

3.2.4 Czas życia obiektu

Czas życia obiektu w programowaniu obiektowym to czas, kiedy obiekt jest używany w programie, tak więc obiekt „żyje” od wywołania konstruktora (utworzenie instancji obiektu) do wywołania destruktora, który ten obiekt „zniszczy”.

W języku Java nie istnieją destruktory jako metody niszczące obiekty. Ich funkcję przejął Garbage Collector, który według swoich algorytmów niszczy nieużywane już w programie obiekty.

Zawsze przed usunięciem obiektu przez Garbage Collector mamy możliwość uruchomienia specjalnej metody `finalize()`, a dokładniej jej nadpisania, ponieważ domyślnie metoda ta jest pusta.

Działający według swojego wewnętrznego algorytmu Garbage Collector usuwa obiekty, gdy przestają być używane bądź chwilę później, a więc istnieje ryzyko, że metoda `finalize` nie zostanie wywołana, a tym samym dane, które miały zostać zapisane przez tę metodę, zostaną utracone. Przez losowość działania Garbage Collectora nie zaleca się korzystania z metody `finalize` do wykonywania ważnych dla programu czy danych operacji.

3.3 Dziedziczenie i abstrakcyjność

3.3.1 Kompozycja i dziedziczenie

Dziedziczenie jest kolejnym podstawowym elementem programowania obiektowego. Jego celem jest rozszerzenie funkcjonalności klasy. W języku Java każda klasa może dziedziczyć tylko po jednej klasie. Załóżmy, że mamy klasę `Auto`:

```
1  public class Auto {
2
3      private int pojemnosc;
4      private int masa;
5      private int predkosc;
6
7      public void jedzie() {
8          System.out.println("Auto jedzie");
9      }
10
11     public void hamuje() {
12         System.out.println("Auto hamuje");
13     }
14
15 }
```

Klasa ta umożliwia ustawienie pól `pojemność` i `masa` przez konstruktor (linia 6) i ma dwie metody publiczne `jedzie()` i `hamuje()`. Jeśli teraz chcielibyśmy stworzyć klasę któregoś z producentów aut, np. `Forda`, to możemy założyć, że samochody `Ford` też mają funkcjonalności klasy `Auto` – pojemność i masę. Oczywiście, moglibyśmy stworzyć klasę `Ford` i zdefiniować w niej te same metody i pola, co w klasie `Auto`, ale byłoby to powielaniem kodu. W językach obiektowych, aby tego uniknąć, mamy możliwość dziedziczenia pól i metod od innych klas, jak w przykładzie, w którym klasa producenta `Ford` może dziedziczyć po klasie `Auto` i tym sposobem przejąć wszystkie pola i metody klasy `Auto`. Do dziedziczenia używamy słowa `extends`.

```java
public class Ford extends Auto {

    public void szyberdach() {
        System.out.println("Wieje w fordzie");
    }

}
```

Klasa `Ford` dziedziczy po klasie `Auto`, dzięki czemu obiekty stworzone na bazie klasy `Ford` będą miały metodę `szyberdach()` orz metody `jedzie()` i `hamuje()` oraz pola `pojemność`, `masa` i `prędkość`. Niestety, z obiektu `Ford` nie mamy dostępu do tych pól ponieważ mają kwalifikator dostępu `private`.

Kwalifikator dostępu `protected`

Aby uzyskać dostęp do elementów klasy, po której dziedziczymy, a przy okazji nie upubliczniać ich wszystkim, stworzono kwalifikator dostępu `protected`. Daje on możliwość dostępu do elementów klasy i obiektu pod warunkiem, że dziedziczymy po tej klasie, natomiast nie będą one widoczne na zewnątrz obiektu czy klasy.

```java
1 public class Auto {
2
3     private int pojemnosc;
4     private int masa;
5     protected int predkosc;

…
```

W linii 5 ustawiliśmy dla pola `prędkość` w klasie `Auto` dostęp chroniony (`protected`). Dzięki temu zabiegowi klasa `Ford` dziedzicząca po klasie `Auto` ma dostęp do tego pola, może mu zmieniać wartość, natomiast to pole nadal nie jest widoczne na zewnątrz obiektu.

Drugim sposobem rozszerzania funkcjonalności klasy jest kompozycja. Poniżej mamy klasę `Klimatyzacja` i chcielibyśmy mieć jej funkcjonalność w klasie `Ford`.

```
1  public class Klimatyzacja {
2
3      private int tOtoczenia;
4      private int tUstawiona;
5
6      public Klimatyzacja(int temperatura) {
7          this.tOtoczenia = temperatura;
8      }
9
10     public float ustawTermostat(int temperatura) {
11         this.tUstawiona = temperatura;
12
13         int rTemperatur = Math.abs(tOtoczenia - tUstawiona);
14         if(rTemperatur >= 8) {
15             return 0.15F;
16         } else if(rTemperatur < 8 && rTemperatur > 0) {
17             return 0.10F;
18         } else {
19             return 0;
20         }
21     }
22
23 }
```

Moglibyśmy po niej dziedziczyć, ale, jak pamiętamy możemy dziedziczyć, tylko po jednej klasie. Klasa `Auto` mogłaby dziedziczyć po `Klimatyzacja`, ale wtedy wszystkie klasy dziedziczące po `Auto` miałyby klimatyzację, co też nie jest przez nas pożądane. Pozostaje nam rozszerzenie funkcjonalności przez kompozycję, czyli stworzenie pola w klasie typu `Klimatyzacja` i dorobienie metod uruchamiających te funkcjonalności z `Klimatyzacja`, które chcemy wystawić na zewnątrz.

```
1  public class Ford extends Auto {
2
3      private Klimatyzacja klimatyzacja = new Klimatyzacja(18);
4
5      public void szyberdach() {
6          System.out.println("Wieje w fordzie");
7      }
8
9      public void uruchomKlimatyzacje(int temperatura) {
10         klimatyzacja.ustawTermostat(temperatura);
11     }
12
13 }
```

Powyżej mamy przykład, w którym rozszerzamy klasę `Ford` o funkcjonalność `uruchomKlimatyzację()` przez kompozycję. W linii 3 deklarujemy pole i od razu inicjalizujemy obiekt typu `Klimatyzacja`, dzięki czemu będziemy mieli wewnątrz obiektu `Ford` dostęp do funkcjonalności obiektu typu `Klimatyzacja`. W linii 9 definiujemy i deklarujemy metodę, która uruchamia funkcjonalność z obiektu `Klimatyzacja` i dzięki temu ta funkcjonalność jest dostępna na zewnątrz obiektu `Ford`.

3.3.2 Użycie słowa `super`

Dodajmy do naszej klasy `Auto` konstruktor, który ustawia pojemność i masę auta oraz wewnątrz oblicza prędkość auta:

```
1  public class Auto {
2
3      private int pojemnosc;
4      private int masa;
5      protected int predkosc;
6
7      public Auto(int pojemnosc, int masa) {
8          this.pojemnosc = pojemnosc;
9          this.masa = masa;
10         this.predkosc = (pojemnosc / masa) * 100;
11     }
12
13     public void jedzie() {
14         System.out.println("Auto jedzie");
15     }
16
17     public void hamuje() {
18         System.out.println("Auto hamuje");
19     }
20
21 }
```

Jeśli teraz chcielibyśmy, aby klasa `Ford` dziedziczyła po `Auto`, to w klasie `Ford` musimy wywołać konstruktor klasy `Auto`, ponieważ konstruktor wymaga dwóch parametrów, które muszą zostać podane. Poniższy kod pokazuje zmiany w klasie `Ford`:

```
1  public class Ford extends Auto {
2
3      private Klimatyzacja klimatyzacja;
4
5      public Ford(int pojemnosc, int masa) {
6          super(pojemnosc, masa);
7          klimatyzacja = new Klimatyzacja(18);
8      }

9
10     public void szyberdach() {
11         System.out.println("Wieje w fordzie");
12     }
13
14     public void uruchomKlimatyzacje(int temperatura) {
15         klimatyzacja.ustawTermostat(temperatura);
16     }
17
18     public void jedzie() {
19         super.jedzie();
20         System.out.println("Jadę Fordem");
21     }
22
23 }
```

W linii 6 widzimy konstrukcję `super(pojemnosc, masa)` – to wywołanie konstruktora z klasy, po której dziedziczymy, a `pojemność` i `masa` to przekazane parametry.

Słówko `super` ma jeszcze jedno zastosowanie. Dzięki niemu można wywołać, oprócz konstruktora, dowolną metodę z klasy, po której dziedziczymy.

W linii 18 mamy przykład przysłonienia, czyli nadpisania metody `jedzie()` z klasy `Auto`. Mówiąc dokładniej, jeżeli wywołamy metodę `jedzie()` z obiektu `Ford`, to wywołamy metodę `jedzie()` zdefiniowaną w klasie `Ford`, a nie w klasie `Auto`. Jeżeli musimy przysłonić metodę, bo ma ona funkcjonować inaczej niż w klasie, po której dziedziczymy, ale uruchomienie metody z klasy, po której dziedziczymy, jest niezbędne, to z pomocą przychodzi nam słówko `super`. Takie rozwiązanie mamy w linii 19: `super.jedzie()` wywołujemy metodę `jedzie()` z klasy `Auto`, którą przysłaniamy.

3.3.3 Klasy abstrakcyjne

Obecnie mamy już całkiem dobry mechanizm do tworzenia samochodów. Dzięki dziedziczeniu możemy tworzyć w prosty sposób producentów aut i wzbogacać podstawowe możliwości o nowe, co znajduje zastosowanie w klasie `Ford`. Załóżmy jednak, że prędkość aut nie może być wyliczana zawsze w taki sam sposób dla każdego producenta, ponieważ są przecież na rynku dostępne różne silniki. Najlepiej byłoby, aby każdy producent, czyli klasa dziedzicząca po `Auto`, sam określał metodę wyliczania prędkości – tu programowanie obiektowe wprowadza nam taką możliwość przez wprowadzenie abstrakcji, a dokładniej klas abstrakcyjnych.

Klasa abstrakcyjna to w założeniu normalna klasa – może posiadać pola i metody, może dziedziczyć i działają w niej wszystkie zasady, o których wspominaliśmy w poprzednich rozdziałach, ale odróżniają ją dwie rzeczy:

- jest przeznaczona tylko do dziedziczenia – nie można stworzyć na jej bazie obiektu,
- może posiadać metody abstrakcyjne, czyli deklaracje metod bez ich definicji.

Klasy abstrakcyjnej używamy wtedy, gdy zakładamy jakąś funkcjonalność, ale ustalenie jej działania zależy od nieznanych czynników. Klasy i metody abstrakcyjne deklarujemy przez użycie słówka `abstract`:

```
1  public abstract class Auto {
2
3      protected int pojemnosc;
4      protected int masa;
5      protected int predkosc;
6
7      public Auto(int pojemnosc, int masa) {
8          this.pojemnosc = pojemnosc;
9          this.masa = masa;
10         this.predkosc = obliczPredkosc();
11     }
12
13     protected abstract int obliczPredkosc();
14
15     public void jedzie() {
16         System.out.println("Auto jedzie");
17     }
18
19     public void hamuje() {
20         System.out.println("Auto hamuje");
21     }
22
23 }
```

Powyżej mamy kod klasy `Auto` przekształconej w klasę abstrakcyjną. Pierwsza zmiana jest w linii 1 – dodanie słówka `abstract` w deklaracji klasy. Kolejna to linie 3 i 4 – polom `pojemnosc` i `masa` zmieniliśmy kwalifikatory dostępu na `protected`, ponieważ potrzebne będą nam do definicji metody abstrakcyjnej liczącej prędkość. Dalsze zmiany to linia 10, czyli podstawienie pod prędkość wartości zwracanej przez metodę abstrakcyjną `obliczPredkosc()` i linia 13, w której deklarujemy metodę abstrakcyjną `obliczPredkosc()`. Zmiany te sprawiają, że w klasie, która dziedziczy po klasie `Auto`, czyli w klasie `Ford`, musimy zdefiniować metodę `obliczPredkosc()`.

Poniżej mamy definicję metody abstrakcyjnej z klasy `Auto`, którą musimy dodać:

```
1  public class Ford extends Auto {
2      …
3     protected int obliczPredkosc() {
4         return (pojemnosc / masa) * 100;
5     }
6      …
7  }
```

Teraz każdy, kto stworzy klasę dziedziczącą po klasie `Auto`, będzie musiał określić sposób wyliczania prędkości, tak jak zrobiliśmy to w klasie `Ford`.

3.4 Interfejsy i polimorfizm

3.4.1 Interfejsy

Interfejsy są kolejnym sposobem umożliwiającym rozszerzenie funkcjonalności obiektów. Przez implementację interfejsu wymusza się na twórcy klasy definicję metod zadeklarowanych w interfejsie, podobnie jak to jest z metodami abstrakcyjnymi. Metody deklarowane w interfejsach są zawsze publiczne.

W interfejsie możemy dodać tylko pola i metody statyczne.

Dodatkowo, mamy możliwość zdefiniowania domyślnych definicji metod zadeklarowanych w interfejsie. W naszym przypadku stworzymy interfejs `Combi`, który będzie zmieni samochód osobowy w kombi i umożliwi podanie ładowności auta:

```
1  public interface Combi {

2      public final static int MAX_LADOWNOSC = 100;
3
4      public default void zaladuj(int masa) {
5          System.out.println("załadowałem " + masa + " bagaży");
6    }
7      public void wyladuj();
8
9  }
```

Powyżej mamy przykład interfejsu, który posiada pole statyczne (linia 2), deklarację metody z domyślną definicją (linia 4 słówko `default`).

Interfejsy mogą dziedziczyć po wielu interfejsach i tylko po interfejsach. Używa się do tego słówka `extends`, jak w przypadku klas dziedziczących po innych klasach.

```
public interface Interface1 extends Interface2, Interface3, … { … }
```

Stworzymy teraz nową klasę `FordFocusCombi`, która będzie miała zaimplementowany interfejs `Combi`. Musimy w niej zdefiniować wymagane metody. Metoda `załaduj()` ma domyślną definicję, więc musimy zdefiniować jedynie metodę `wyładuj()`:

```
1  public class FordFocusCombi extends Ford implements Combi {
2
3      public FordFocusCombi() {
4          super(1800, 1700);
5      }
6
7      @Override
8      public void wyladuj() {
9          System.out.println("wyładowałem bagaże");
10     }
11
12 }
```

3.4.2 Polimorfizm

Wszystkie ostatnio poznane elementy służące rozszerzeniu funkcjonalności klas, jak dziedziczenie, klasy abstrakcyjne i interfejsy, są związane z ostatnim bardzo ważnym zagadnieniem w programowaniu obiektowym jakim jest polimorfizm, inaczej wielopostaciowość. Przez dziedziczenie i implementację interfejsów zmienna obiektowa stworzona na podstawie klasy jest typu tej klasy, w której została stworzona, ale również wszystkich klas, które biorą udział w łańcuchu dziedziczenia, jak i wszystkich zaimplementowanych interfejsów. Co za tym idzie, zmienna obiektowa zbudowana na bazie klasy `FordFocusCombi` jest oczywiście typu `FordFockusCombi`, ale również `Ford` i `Auto` oraz `Combi`.

3.5 Użycia słówka `final`

Słówko `final` ma wiele zastosowań. W poniższym kodzie zostały zawarte wszystkie użycia słówka `final`:

```
1  public final class FordFocusCombi extends Ford implements Combi {
2
3      public final static String MARKS = "Ford";
4
5      private final Klucz kluczyk;
6
7      public FordFocusCombi() {
8          super(1800, 1700);
9          kluczyk = new Klucz();
10     }
11
12     public final uruchom(final Klucz kluczyk) {
13         if(kluczyk.equals(this.kluczyk)) {
14             jedzie();
15         }
16     }
17
18     @Override
19     public void wyladuj() {
20         System.out.println("wyładowałem bagaże");
21     }
22
23 }
```

— W definicji klasy (linia 1): jeżeli dodamy słówko `final` przy definicji klasy, to blokujemy możliwość dziedziczenia po takiej klasie,
— Przy deklaracji pól obiektowych i statycznych (linia 3 i 5): jeśli pod takie pole podstawimy wartość to nie można już jej zmienić,
— Przy definicji metody (linia 12): jeżeli metoda również statyczna zostaje poprzedzona słowem `final`, to blokujemy możliwość nadpisania takiej metody,
— Deklaracja parametru metody (linia 12): jeżeli parametr metody jest poprzedzony słowem `final`, to pod taki parametr nie można nic podstawić wewnątrz metody.

Rozdział 4.

Typy danych – ciąg dalszy

4.1 Tablice

Tablica to typ służący do przechowywania kolekcji (zbioru) danych. Tablica może przechowywać kolekcję składającą się z jednego typu, np. `int`, `String` lub `Object`. Elementy w tablicy są indeksowane, czyli każdy element tablicy otrzymuje swój unikatowy numer identyfikacyjny. Indeksy nadawane są do zera i dzięki nim możemy się odwoływać do konkretnego elementu w tablicy. Tablicę możemy inicjalizować na trzy sposoby:

```
1  // inicjalizacja tablicy pięcioelementowej
2  int[] tab1 = new int[5];
3  int tab2[] = new int[5];
4  // inicjalizacja i definicja tablicy pięcioelementowej
5  int[] tab3 = { 1, 2, 3, 4, 5 };
```

Linie 2 i 3 to dwa sposoby deklaracji pustych tabel pięcioelementowych.

Linia 5 do deklaracja wraz z definicją – zadeklarowaną tablicę od razu wypełniamy wartościami.

Tablice są obiektami i mają jedno publiczne pole – `lenght`, które przechowuje wielkość tablicy. Od zwykłego obiektu tablice różnią się pobieraniem i ustawianiem wartości elementów tablicy. By ustawić lub pobrać wartość w tablicy dla danego indeksu, używamy nawiasów kwadratowych ([]), w których podajemy indeks (miejsce w tablicy). W nawiasach kwadratowych możemy również podać zmienną, co ilustruje poniższy kod:

```
1 tab1[0] = 10; // ustawienie wartości na pozycji 0
2 System.out.println(tab1[0]); // pobranie i wypisanie wartości
3 int index = 3;
4 tab1[index] = 14; // ustawienie wartości na pozycji 0
5 System.out.println(tab1[index]); // pobranie i wypisanie wartości
6 System.out.println(tab1.length); // wypisanie rozmiaru tablicy
```

Elementy tablicy traktujemy jak zwykłe zmienne i podlegają tym samym zależnościom i operacjom, co zwykłe zmienne. Tablice można iterować, to znaczy przechodzić po jej elementach specjalną postacią pętli `for`:

```
1 for(int element : tab3) {
2     System.out.println(element);
3 }
```

W linii 1 zmienna `element` reprezentuje wartość z tablicy, a każda iteracja pętli przesuwa się po kolejnych elementach tablicy i zmienia wartość zmiennej `element`.

W linii 2 wypisywane są poszczególne wartości z tablicy tab3 na ekranie.

Dotychczas zajmowaliśmy się tablicami jednowymiarowymi, ale tablice mogą być również wielowymiarowe. Wymiary w tablicach należy interpretować jako zagłębianie kolejnych tablic w tablicach, czyli tablica 2-wymiarowa to tablica, której elementami są tablice jednowymiarowe. W tablicach wielowymiarowych każdy wymiar może przechowywać jako elementy tablice o różnych rozmiarach, ale zawsze tego samego typu. Tablicę wielowymiarową deklaruje się przez dodanie kolejnych nawiasów kwadratowych, np. `double[][] tab = new double[2][];` lub `Object[][][] tab = new Object[3][2][];`.

Poniżej przykłady użycia tablic wielowymiarowych:

```
1  String strTab[][][] = new String[4][2][2];
2  strTab[0][0][0] = "ala";
3  strTab[0][0][1] = "ola";
4  System.out.println(strTab[0][0][0] + "," + strTab[0][0][1]);
5  int iTab[][] = new int[3][];
6  iTab[0] = new int[3];
7  iTab[1] = new int[2];
8  int[] iTabSub = { 1, 3, 5, 7, 9 };
9  iTab[2] = iTabSub;
10 System.out.println(iTab[2][3]);
11 float[][] fTab = { {3F, 2.2F, (float)4.1}, {2.4F, 9F, 1.0F} };
12 System.out.println(fTab[1][2]);
```

4.2 Operacje na łańcuchach znaków

Łańcuchy znakowe (elementy typu `String`) można potraktować jako tablicę znaków (`char`) indeksowaną od zera. Można odnieść się do poszczególnego znaku w łańcuchu dzięki metodzie `charAt(int index)`, wyciąć fragment tekstu z użyciem funkcji `substring(int beginIndex, int endIndex)` czy zamienić tekst na tablicę słów dzieląc po spacji używając metody `split(String regexp)`. Można też wyszukiwać i zamieniać fragmenty tekstu wykorzystując wyrażenia regularne. Poniżej przykłady użycia metod z łańcucha znaków:

```
1  String str = "Ala ma kota, a Ola ma psa";
2  // pobieranie poszczególnych znaków z łancucha
3  System.out.println(str.charAt(5));
4  // wyciecie fragmentu tekstu "ma kota"
5  String subStr = str.substring(4, 11);
6  System.out.println(subStr);
7  // pobranie długości łańcuchów znaków
8  System.out.println(subStr.length() + " - " + str.length());
9  // wyszukanie indeksu wystąpienia od poczatku i od końca
10 System.out.println(
11    str.indexOf("ma") + " - " +
12    str.lastIndexOf("ma")
13 );
14 // zamiana tekstu
15 System.out.println(str.replace("ma", "da"));
16 // zamiana regexp (wyrazenia regularne)
17 System.out.println(str.replaceAll("ma\\s[A-z]+", "regexp"));
18 // przekształcenie w tablicę znaków i podzielenie
19 System.out.println(str.toCharArray().length); // tablica znaków
20 System.out.println(str.split(" ").length); // podzielenie
21 // zamiana wielkie i małe litery
22 System.out.println(str.toLowerCase());
23 System.out.println(str.toUpperCase());
```

4.3 Typ wyliczeniowy

Typ wyliczeniowy właściwie jest tablicą statycznych obiektów, które umożliwiają użycie ich np. jako identyfikatory zdarzeń, elementów, itp. Możliwości użycia typów wyliczeniowych są zwiększone dzięki temu, że mamy wpływ na postać statycznych obiektów w takiej tablicy – możemy zdefiniować klasy na bazie których te elementy będą stworzone.

Typ wyliczeniowy może być deklarowany wewnątrz klasy, jak i w osobnym pliku jako niezależny byt.

Poniżej przykład deklaracji klasy wewnątrz klasy:

```
1 public class Kalkulator {
2
3     private enum Operacje {
4           SUMA, ROZNICA, ILOCZYN, ILORAZ
5     }
6
7 }
```

Poniżej przykład deklaracji `enum` jako osobny plik:

```
1 public enum Operacje {
2
3     SUMA, ROZNICA, ILOCZYN, ILORAZ;
4
5 }
```

Elementy typu wyliczeniowego można porównywać znakiem == i idealnie nadają się do wykorzystania w instrukcji `switch`:

```
1  private double oblicz(double val1, double val2, Operacje oper) {
2      switch(oper) {
3          case ILOCZYN:
4              return val1 * val2;
5          case ILORAZ:
6              return val1 / val2;
7          case ROZNICA:
8              return val1 - val2;
9          case SUMA:
10             return val1 + val2;
11     }
12
13     return 0;
14 }
```

Elementy `enum` mogą być definiowane jako normalne klasy. Ciało klasy piszemy pod listą elementów. Konstruktor klasy elementu `enum` nie może być publiczny, nie może również dziedziczyć, natomiast może mieć zaimplementowane interfejsy.

Poniżej przykład silnika kalkulatora oparty o typ wyliczeniowy:

```java
public enum Operacje {

    SUMA('+'), ROZNICA('-'), ILOCZYN('*'), ILORAZ(':');

    private char znak;

    private Operacje(char znak) {
        this.znak = znak;
    }

    public double oblicz(double val1, double val2) {
        double wynik = 0;

        switch(this) {
            case ILOCZYN:
                wynik = val1 * val2;
                break;
            case ILORAZ:
                wynik = val1 / val2;
                break;
            case ROZNICA:
                wynik = val1 - val2;
                break;
            case SUMA:
                wynik = val1 + val2;
                break;
        }
        System.out.println(val1+" "+znak+" "+val2+" = "+wynik);

        return wynik;
    }

    public static Operacje pobierzOperacjęPoZnaku(char znak) {
        for(Operacje item : Operacje.values()) {
            if(item.znak == znak) {
                return item;
            }
        }

        return null;
    }

}
```

Rozdział 5.

Wyjątki

Błędy występujące podczas wykonywania programu mogą być spowodowane różnymi przyczynami – od błędnej implementacji kodu, czyli błędy w logice programu, po takie związane z czynnikami zewnętrznymi, jak brak pliku czy ograniczony dostęp do serwera. W tym rozdziale omówimy rozwiązanie, które umożliwi zapobieganie przerwania działania aplikacji przez ten drugi rodzaj błędów.

5.1 Czym jest wyjątek?

Wyjątek jest informacją o nieprawidłowości, którą zgłasza („rzuca") JVM w postaci obiektu `Exception` lub obiektu dziedziczącego po nim. Oprócz standardowych wyjątków, możemy tworzyć własne. Warunek jest jeden – nasze wyjątki muszą być pochodnymi klasy Exception. Klasy wyjątków są zwykłymi klasami i mają takie same właściwości:

```
1 public class MyException extends Exception {
2
3     private static final long serialVersionUID = 1L;
4
5     public MyException() {
6         super("Mój wyjątek!");
7     }
8
9 }
```

Powyższy kod przedstawia najprostszą własną klasę wyjątku o nazwie `MyException`.

W linii 1 mamy dziedziczenie po klasie `Exception` (jest to niezbędne, aby wyjątek mógł być „rzucony").

W liniach 5-7 mamy konstruktor klasy, a w nim (linia 6) wywołanie jednego z kilku konstruktorów klasy `Exception`, który ustawia komunikat naszego wyjątku, jaki zostanie zwrócony przy jego wystąpieniu.

Najważniejsze metody dostępne w klasie `Exception` to `getMessage()`, która zwraca ustawiony komunikat, `getStackTrace` oraz rodzina metod `printStackTrace`, które umożliwiają pobrać bądź wypisać zrzut pamięci, co w niej się kryło w trakcie wystąpienia błędu. Do „rzucania" wyjątku służy słówko `throw` np.

```
1 throw new MyException();
```

5.2 Konstrukcja `try...catch`

„Rzucenie" wyjątku zmusza nas do jego obsługi, to znaczy jeżeli w jakiejś metodzie rzucamy wyjątek, to musimy przy deklaracji tej metody poinformować o możliwości wystąpienia wyjątku. By to zrobić, używamy `throws` i wypisujemy nazwy możliwych wyjątków, rozdzielając je przecinkami.

Poniżej przykład takiej metody:

```
1 public static void metoda() throws MyException, IOException {
2     if(error1) {
3         throw new MyException();
4     } else if(error2) {
5         throw new IOException();
6     }
7 }
```

Deklaracja rzucanych wyjątków w nagłówku metody powoduje, że obowiązek obsłużenia wyjątku, który wystąpił w metodzie, spada na metodę, która ją wywołała. Możemy przekazywać obsługę wyjątków do góry, aż do metody `main`, w której też mamy możliwość przekazania jej wyżej za pomocą `throws`. Niestety, dla metody `main` elementem nadrzędnym jest JVM, a ona, otrzymując wyjątek do obsłużenia, przerwie działanie programu i wypisze w konsoli informację o błędzie.

Mamy w języku Java mechanizm, który umożliwia nam obsłużenie wyjątków – jest to konstrukcja `try...catch`. W jej pierwszym bloku `try` umieszczamy kod, w którym potencjalnie może wystąpić wyjątek, natomiast w bloku `catch` kod, który ma się

wykonać w przypadku wystąpienia wybranego wyjątku. Bloków `catch` może być tyle, ile typów wyjątków się spodziewamy.

```
1 public static void main(String[] args) {
2     try {
3         metoda();
4     } catch (MyException e) {
5         System.out.println(e.getMessage());
6     } catch(IOException ioe) {
7         ioe.printStackTrace();
8     }
9 }
```

Dodatkowo mamy możliwość użycia rozwiązania multi-catch, czyli jeden blok `catch` dla wielu wyjątków, jednak należy pamiętać, że przy tym rozwiązaniu zmienna reprezentująca wyjątki w bloku `catch` jest typu Exception, a nie poszczególnych wyjątków.

Poniżej przykład użycia multi-catch:

```
1 public static void main(String[] args) {
2     try {
3         metoda();
4     } catch (MyException | IOException e) {
5         System.out.println(e.getMessage());
6     }
7 }
```

Partnerzy


HORDE
TECHNOLOGY


STARTER


POLSKA
PRESS
INTERNET


nokaut.pl


SPEED NET


FUTURE 3


Trójmiasto JUG


recruitcoders
reach for competence


NET
VISION


nstytut
kultury
miejskiej


Komijo


KANCELARIA PRAWNA LT
Łukasz Tyszkowski


programista


STUDENCKIE KOŁO
INŻYNIERII POMIAROWEJ
SKIP
WYDZIAŁ ELEKTROTECHNIKI
I AUTOMATYKI