# PROGRAMOWANIE W JĘZYKU JAVA

## MATERIAŁY SZKOLENIOWE


Kamil Perczyński
JSYSTEMS 2015

# Spis treści

# Wprowadzenie - czyli szybki rzut oka

## Ogólne omówienie platformy

Nie wierzę, że osoba czytająca to zdanie nigdy nie słyszała nazwy „JAVA". Jestem całkowicie przekonany, że nawet korzystałeś/aś już z programów napisanych w Javie nawet o tym nie wiedząc. Javę spotykamy bowiem wszędzie – w aplikacjach desktopowych, aplikacjach webowych, a nawet w mikrokontrolerach (dość świeża JAVA ME). Nas jednak obchodzi dużo bardziej wnikliwa odpowiedź na pytanie „Czym jest JAVA?". Otóż jest to platforma napisana w C++ udostępniająca nowy język programowania (którego składnia jest wzorowana na C++). Co więcej programy napisane w Javie są całkowicie niezależne od systemu operacyjnego. Ten sam program, który napiszemy korzystając z systemu Windows, tak samo uruchomimy na dowolnym Linuksie czy OS X. Warunek jest jeden – na danej maszynie musi być zainstalowana Java.

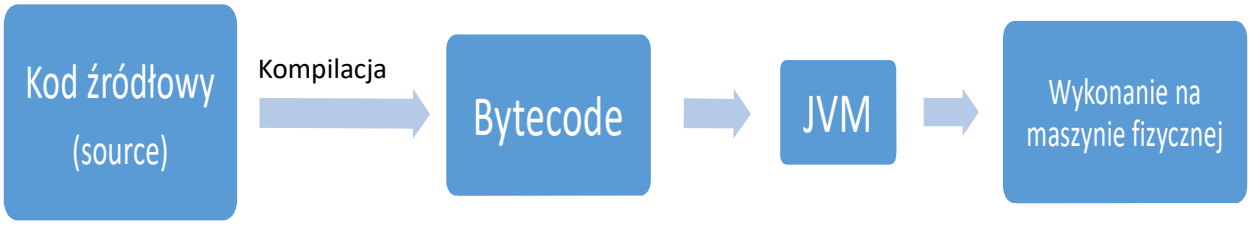
## Składowe JAVA – zagadkowe trójliterowce JDK, JVM, JRE

Podstawowy element pozwalający na uruchamianie programów Javy to JRE *(Java Runtime Enviroment*). Jeśli zainstalujemy JRE uruchomimy każdy program napisany w Javie przeznaczony do uruchamiania na desktopach. JDK (*Java Development Kit*) jest to pakiet składający się z JRE uzupełnionego o kompilator (*javac*). Mając zainstalowane JDK możemy zarówno uruchamiać programy Javowe, jak również pisać własne. Podsumujmy więc:

$$JAVA\ SE = \begin{cases} JRE \rightarrow Java\ Runtime\ Enviroment \\ JDK = JRE + javac\ (kompilator) \rightarrow Java\ Development\ Kit \end{cases}$$

OK. Wyjaśniliśmy już dwa skróty, pozostał ostatni – JVM (*Java Virtual Machine*). Pojęcie maszyny wirtualnej Javy wynika z jej podstawowego założenia – przenośności kodu (*Code Once Run Anywhere*) i jest środkiem do jego osiągnięcia. Standardowo proces przetworzenia kodu źródłowego do kodu wykonywalnego przebiega następująco (np. w przypadku C, C++):

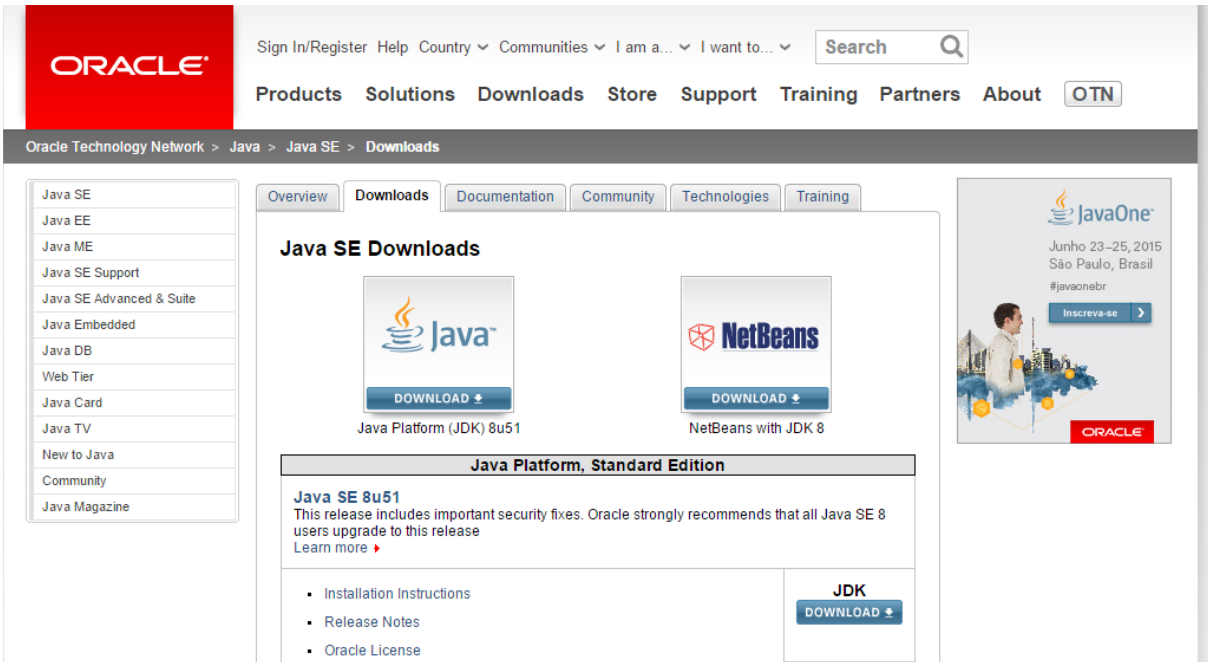

W ten sposób powstał kod maszynowy, który jest ściśle związany z systemem operacyjnym. W Javie powyższy proces przebiega inaczej – wprowadza się jeszcze jeden etap pośredni:




Już tłumaczę co się tu dzieje. Otóż w Javie, jeśli kompilujemy nasz kod źródłowy (kody źródłowe znajdują się w plikach `*.java`) – nie jest on przetwarzany bezpośrednio do kodu maszynowego, a do tzw. Bytecode (w wyniku kompilacji powstaje plik `*.class`). Bytecode jest kodem, który może być wykonywany na maszynie wirtualnej Javy i to ona zajmuje się przetłumaczeniem naszego Bytecode'u na komendy sprzętowe. Z powyższego wykresu bezpośrednio wynika, że Bytecode, jest uniwersalny (niezależny od platformy systemowej), wszystko jest kwestią odpowiedniego JVM. I tak jest w rzeczywistości. Utrzymaniem i rozwojem JVM zajmuje się firma Oracle, nam pozostaje tylko pisać, pisać, pisać i pisać ☺

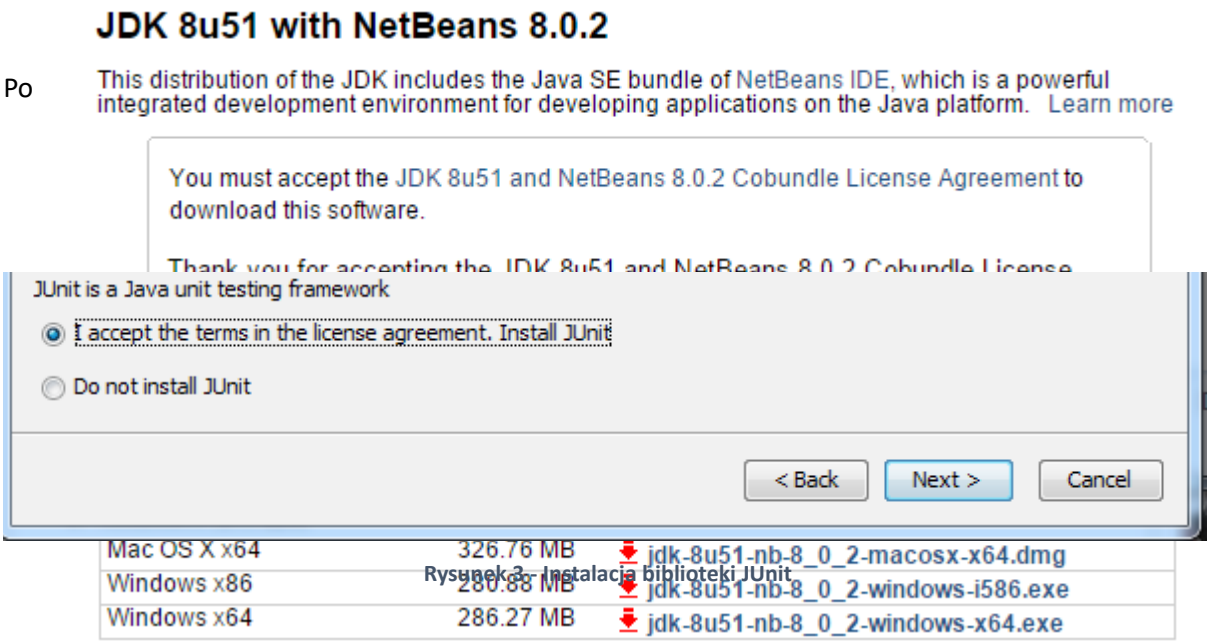
## Instalacja JDK 1.8 i środowiska programistycznego NetBeans IDE 8.0.2

Instalkę najnowszego JDK (wersja 1.8) pobieramy z oficjalnej strony Oracle'a. JDK dla różnych platform znajduje się pod adresem http://www.oracle.com/technetwork/java/javase/downloads/index.html.




**Rysunek 1 - Instalacje JDK 1.8**

Mamy tu do wyboru dwie opcje. Ta po lewej oznacza instalację samego JDK (czyli samej platformy), natomiast druga oznacza instalację zarówno JDK 1.8 jak również środowiska programistycznego NetBeans IDE 8.0.2 z modułami do programowania w Javie SE (Standard Edition). Nas interesuje da druga opcja.



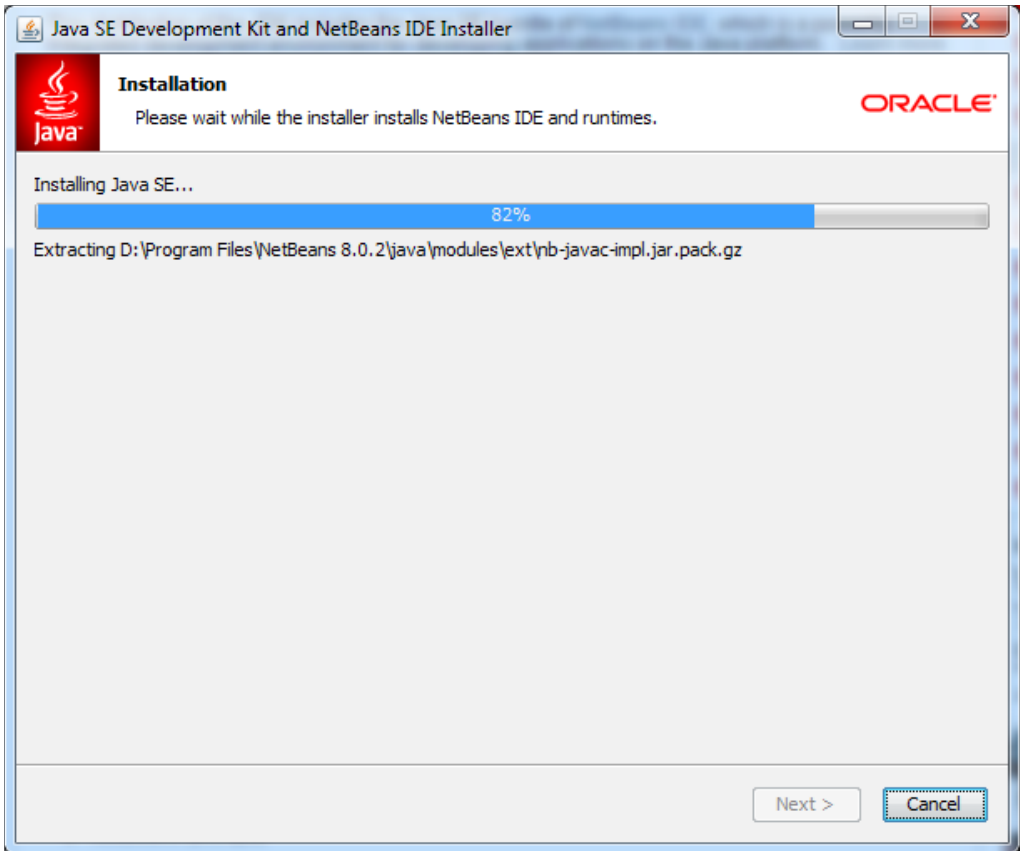

Rysunek 2- Dostępne instalacje najnowszej wersji JDK 1.8 wraz z NetBeans 8.0.2



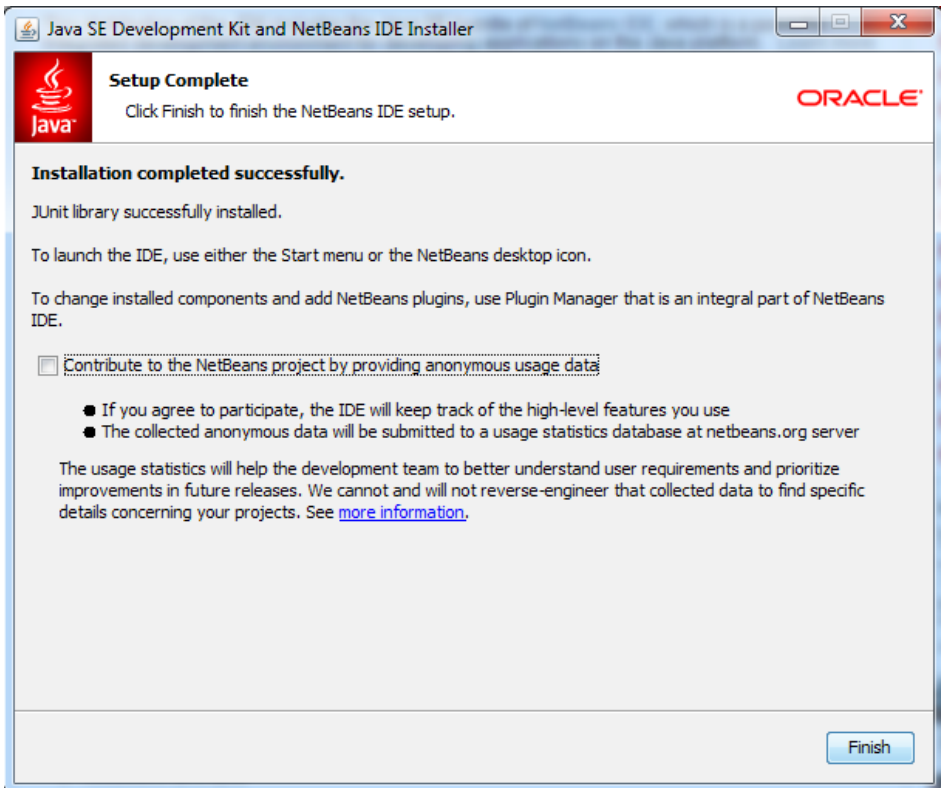

Rysunek 3- Instalacja biblioteki JUnit

Po ściągnięciu odpowiedniej dla siebie wersji (w moim przypadku Windows x64). Po prostu uruchamiamy instalator (Proces instalacji przebiega całkowicie standardowo). Pierwsze pytanie instalatora to instalacja biblioteki `JUnit` – dającej narzędzia do przeprowadzania testów jednostkowych i integracyjnych naszego oprogramowania. Zainstalujemy ją.

Następnie wybieramy ścieżkę pod którą zostanie zainstalowane nasze JDK. Najczęściej zostawia się domyślną. Kolejny ekran to wskazanie katalogu, gdzie zostanie zainstalowany NetBeans. Ostatni ekran to potwierdzenie danych, które już wprowadziliśmy i pytanie czy instalator ma sprawdzić aktualizacje dla NetBeans. Po kliknięciu `Next` nastąpi instalacja – uprzedzam – trwa dość długo.




Ostatni ekran instalacji to pytanie o przekazywanie informacji dla twórców NetBeans IDE w celu poprawiania aplikacji.

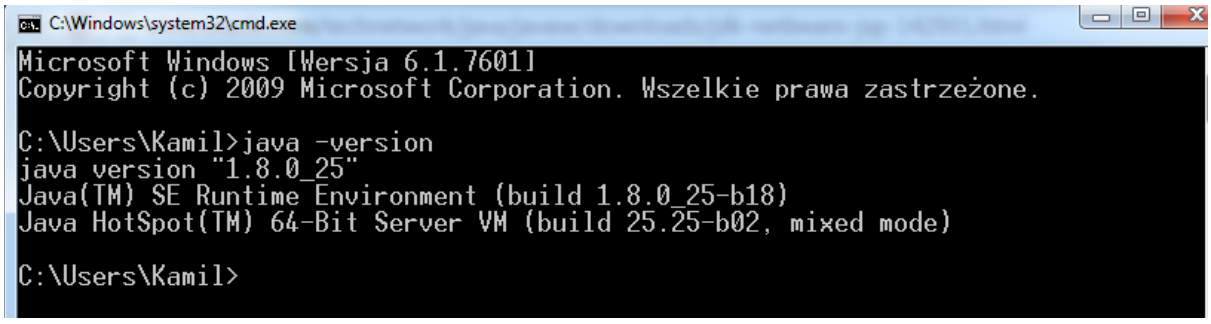



**Rysunek 4 - Końcowy etap instalacji JDK 1.8 i NetBeans**

Po kliknięciu `Finish` nastąpi wyjście z instalatora. W tym momencie posiadamy już zainstalowane JDK 1.8 oraz środowisko programistyczne. Możemy sprawdzić poprawność instalacji za pomocą komendy:

```
java -version
```

odpalonej w Windows'owym `Command Line`:




**Rysunek 5 - sprawdzenie poprawności instalacji JDK 1.8**

Jeżeli w konsoli uzyskasz podobny Output oznacza to poprawność instalacji – Gratulacje! ☺

# Zaczynamy pisać pierwszy program

### Kompilacja i uruchomienie programu z konsoli

W tym momencie możemy już napisać każdy, nawet najbardziej rozbudowany program – oczywiście byłoby to strasznie niewygodne, ale wszystkie narzędzia wspomagające proces kompilacji (`Ant`, `Maven`) i budowania programu wykonują operacje, które zaraz wykonamy my. Napiszemy bowiem najprostszy program w notatniku, skompilujemy go i uruchomimy ręcznie. Omówię najpierw podstawową strukturę programu w Javie:

```
1.  public class Test {
2.
3.      public static void main (String[] args) {
4.
5.              System.out.println( "HELLO WORLD!" );
6.
7.      }
8.
9.  }
```

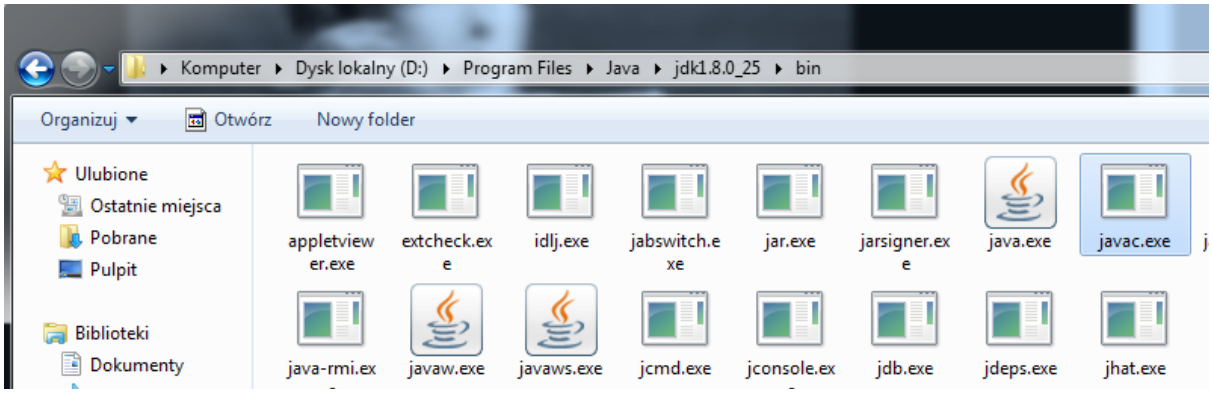
Strasznie to skomplikowane…, co tu się w ogóle dzieje? Spokojnie ☺ jest to bardzo proste. Zacznijmy od początku. W Javie wszystko znajduje się w klasach (nie jest to do końca prawda, ale na początek możemy tak założyć), w każdym razie nasz plik musi zaczynać się od definicji klasy publicznej.

**WAŻNE!**
W jednym pliku może być tylko jedna klasa publiczna i jej nazwa musi dokładnie odpowiadać nazwie pliku (Java jest `Case Sensitive` – wielkość liter ma znaczenie). Zatem w pliku `Test.java` będzie znajdować się publiczna klasa `Test`.

Obiecuję, że działanie słów kluczowych `public` oraz `static` bardzo dokładnie omówimy przy programowaniu obiektowym ☺.

Następnie w klasie `Test`, musi znaleźć się publiczna, statyczna, nic nie zwracająca metoda `main` przyjmująca jako argument tablicę elementów typu `String`. Wiem, że nic Ci to nie mówi, na razie przyjmij, że oznacza to, że metoda ma uruchomić się wszędzie (`public`), zawsze (`static`), nic nie zwracać (`void`), nazywać się `main` itd. Ciało tej metody jest wykonywane w momencie uruchomienia pliku.

Aby skompilować mój plik `Test.java` muszę wykonać komendę kompilatora Javy (`javac`) w konsoli. Kompilator znajduje się w folderze `bin\` katalogu z instalacją JDK (domyślnie `C:\Program Files\Java\jdk1.8.0[wersja aktualizacji]\`). Ja akurat zainstalowałem JDK na dysku `D:\`.

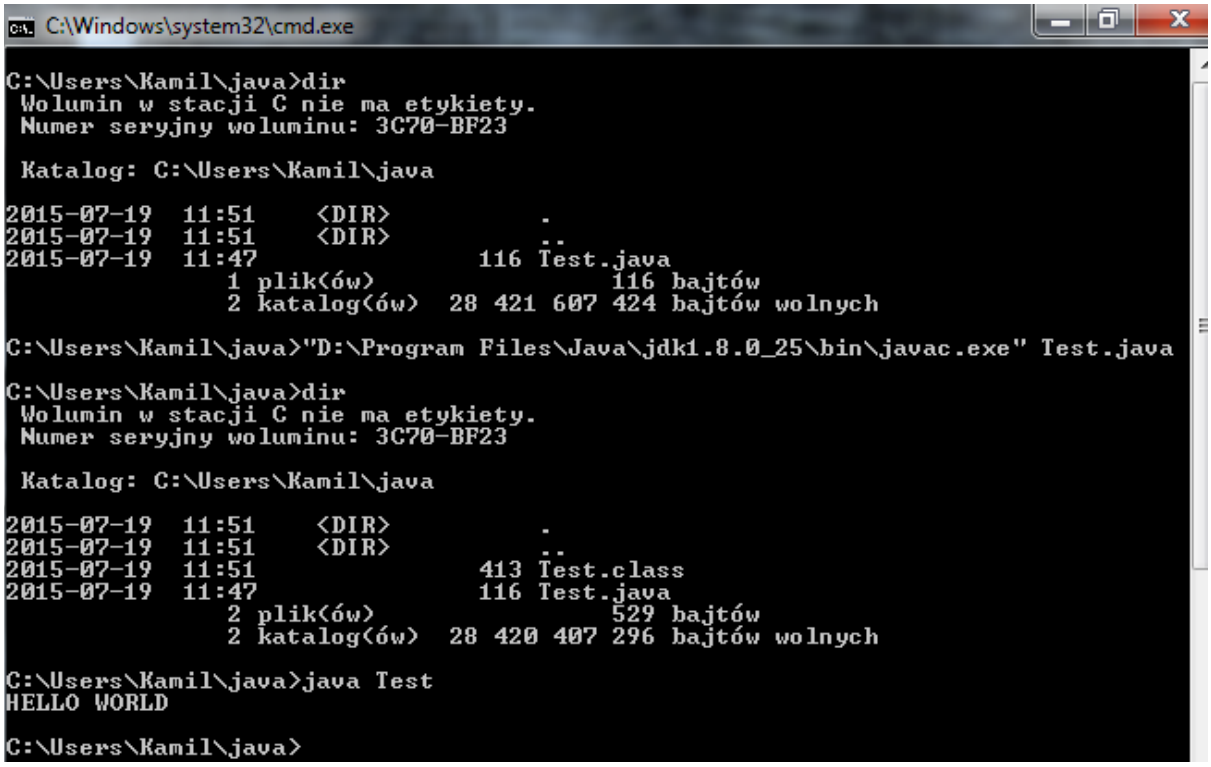



Rysunek 6 - Lokalizacja kompilatora Javy (javac)

Uruchamiam `Command Line` i przechodzę do folderu z moim plikiem `Test.java`. Następnie przeciągam plik `javac.exe` do `Command Line` – zostanie wstawiona pełna ścieżka do `javac`'a. Pozostaje mi już tylko dopisać pełną nazwę pliku do skompilowania. Jeśli kompilacja przebiegła prawidłowo uruchamiam mój program komendą

```
java Test
```

Tym razem nie piszemy już rozszerzenia pliku. Powinieneś uzyskać następujący Output:




Rysunek 7 - Kompilacja pliku Test.java z konsoli

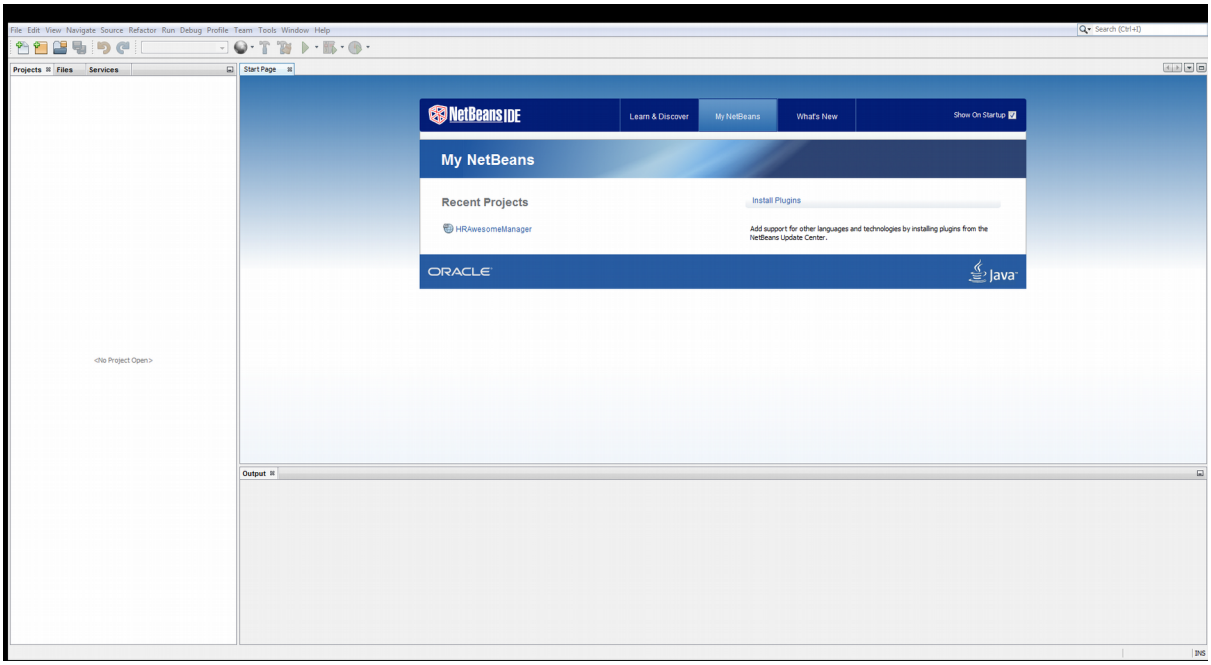
Jeżeli używasz systemu Linux nie musisz podawać ścieżki do pliku `javac.exe`. Wystarczy, że w terminalu wydasz komendę **`javac Test.java`**. Cała reszta nie różni się niczym od Windows'a.

Jak widać w wyniku kompilacji z pliku źródłowego `Test.java` powstał plik `Test.class`.

# Programowanie na poważnie - NetBeans

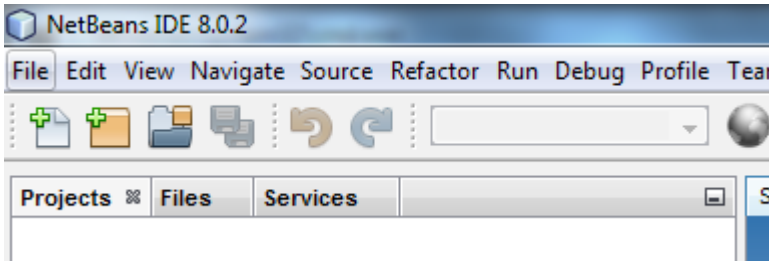
## Pierwsze uruchomienie NetBeans IDE i stworzenie nowego projektu

Po uruchomieniu środowiska programistycznego NetBeans IDE 8.0.2 zobaczysz poniższy obszar roboczy:




Rysunek 8 - Obszar roboczy NetBeans IDE

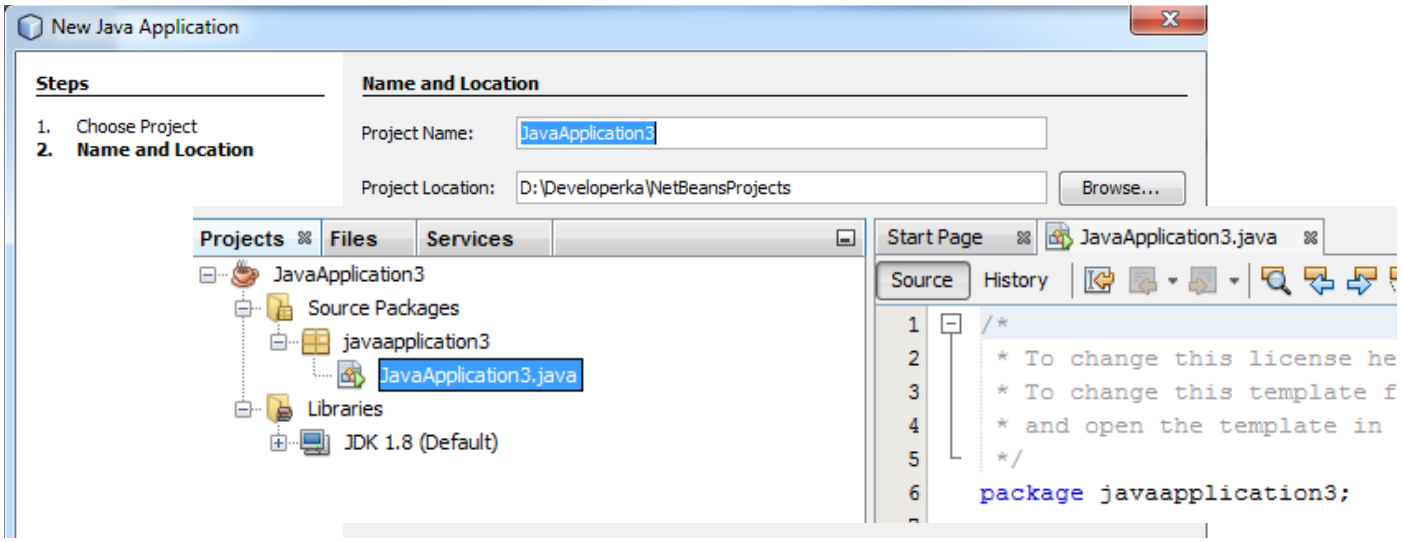
Stworzymy teraz nową aplikację. Klikamy na ikonę folderu w lewym górnym rogu:




Na następnym ekranie z katalogu `Java` wybieram `Java Application`. Uzyskuję poniższy ekran:



Rysunek 9 - Konfiguracja nowego projektu




`Project Name` to oczywiście nazwa naszej aplikacji. Pole `Create Main Class` to nazwa klasy, której metoda `main` zostanie wykonana w przypadku uruchomienia całego projektu, a nie konkretnego pliku. Ja pozostawię domyślną konfigurację. Po kliknięciu `Finish` do naszej zakładki `Projects` zostanie dodany nowy projekt z nazwą (w moim przypadku) `JavaApplication3`.

Struktura projektu jest dość prosta:

- W zakładce `Source Packages` znajdują się pakiety, a w nich nasze pliki `*.java`. Czym i po co są pakiety omówimy zaczynając programowanie obiektowe.
- W zakładce `Libraries` znajdują się biblioteki, które dołączymy do naszego projektu – np. bibliotekę `ojdbc6.jar` czyli sterownik `JDBC` dla Oracle Database pozwalający łączyć się z bazą danych.

**Rysunek 10 - Struktura nowego projektu Aplikacji Java SE**

# Podstawy podstaw - typy nieobiektowe

## Pojęcie zmiennej i typy nie obiektowe

Zmienną nazywamy wartość którą widzimy w programie pod określoną nazwą. Jest to najbardziej elementarna struktura niemal każdego języka programowania. Możemy sobie ją wyobrażać jako pudełko na jakąś wartość. Definicja zmiennej w Javie wygląda tak samo jak w C, C++:

```
1. [typ zmiennej] [nazwa zmiennej] = [wartość];
```

Np.:

```
1. int liczbaCalkowita = 5;

2. double liczbaZmiennoprzecinkowa = 3.14;
```

Od tej pory możemy używać wartość 5 podając nazwę zmiennej. Jest to zadeklarowanie zmiennej połączone z jej inicjalizacją. W Javie (w przeciwieństwie do np. C, C++) nie musimy inicjować każdej zmiennej, wygląda to np. tak:

```
1. int liczbaCalkowita;
```

Jednak próba użycia takiej zmiennej (poza instrukcją przypisania wartości) będzie powodowała błąd kompilacji:

```
public class JavaApplication3 {

    /**
     * @param args the command line arguments
     */
    public static void main(String[] args) {

        int liczbaCalkowita;
        System.out.println( liczbaCalkowita );
    }

}
```

variable liczbaCalkowita might not have been initialized
---
(Alt-Enter shows hints)

**Rysunek 11 - Błąd! Próba wypisania na ekran niezainicjowanej zmiennej**

Instrukcję wypisania na ekran ( `System.out.println()` ) już poznaliśmy więc powyższy kod powinien być jasny.

**WAŻNE!**

W Javie nie ma czegoś takiego jak wartość domyślna zmiennej! Zaobserwujemy jednak pewne wyjątki od tej reguły.

Użyliśmy do tej pory tylko dwóch typów zmiennych `int` oraz `double`, jest ich oczywiście troszkę więcej.

Typy zmiennych **nieobiektowych** w Javie:

- **`int`** – liczba całkowita w zakresie od -2147483648 do 2147483647
- **`long`** – liczba całkowita w zakresie od -9223372036854775808 do 9223372036854775807
- **`double`** – liczba zmiennoprzecinkowa w zakresie od $4.9^{-324}$ do $1.79769313486231570^{308}$
- **`float`** - typ reprezentujący te same dane do `double` tylko w mniejszym zakresie
- **`char`** – pojedynczy znak. Jest to zwykły `int`, ale inaczej interpretowany. Operacja `'a' + 5` jest jak najbardziej prawidłowa ☺
- **`byte`** – pojedynczy bajt. Będziemy używać tego typu danych przy obsłudze plików
- **`boolean`** – typ logiczny. Może przyjąć dwie wartości: `true` lub `false`

Wprowadzimy sobie jeszcze jeden typ zmiennych – **`String`**, który przechowuje tekst. Jest to jednak typ **obiektowy**! O różnicy między obiektem i nieobiektem dowiesz się później. Poniżej przedstawię zadeklarowanie, zainicjowanie i wypisanie zmiennej każdego omówionego typu (oprócz `byte`):

```
12  public class JavaApplication3 {
13
14      /**
15       * @param args the command line arguments
16       */
17      public static void main(String[] args) {
18
19          int i = 321476554;
20          long l = 999999999999999999L;
21          double d = 3.3245239459324579;
22          float f = 3.13f;
23          boolean b = true;
24          char c = 'a';
25          String tekst = "Ala ma kota";
26
27          System.out.println("i = " + i);
28          System.out.println("l = " + l);
29          System.out.println("c = " + c);
30          System.out.println("d = " + d);
31          System.out.println("b = " + b);
32          System.out.println("f = " + f);
33          System.out.println("tekst = " + tekst);
34      }
35
36  }
```

javaapplication3.JavaApplication3

utput - JavaApplication3 (run)

```
run:
i = 321476554
l = 999999999999999999
c = a
d = 3.324523945932458
b = true
f = 3.13
tekst = Ala ma kota
BUILD SUCCESSFUL (total time: 0 seconds)
```

**Rysunek 12 - typy zmiennych nieobiektowych oraz typ `String`**

Zapewne zainteresowały Cię literki (`L`, `f`) na końcu zmiennych typu `long`, `float`. Są potrzebne, ponieważ literały (czyli wartości podane przez programistę bezpośrednio i „z palca”) w Javie również są typowane. Wartość 999999999999999999 wykracza poza zakres zmiennej `int` (a pisząc liczbę „zwyczajnie” domyślnym typem literału jest właśnie `int`), dlatego muszę dookreślić typ literału dopisując na jego koniec `L`. W przypadku literału 3.13 domyślny typ literału zmiennoprzecinkowego to `double`, który ma większy zakres niż `float`, dlatego muszę określić wartość literału jako `float`.

Plik uruchamiamy w NetBeans'ie skrótem klawiszowym `Shift + F6`.

### Odbiór danych z konsoli i konwersje między typami

Napiszemy sobie prosty kalkulator. Program ma przyjąć od użytkownika (z konsoli) dwie liczby typu `double` i wyświetlić wynik dodawania, odejmowania, mnożenia, dzielenia w postaci:

**`[liczba 1] + [liczba 2] = [wynik]`**

Zaczniemy od odbioru danych z konsoli, potrzebny będzie nam tzw. `Scanner`, który deklarujemy w następujący sposób:

```
1.  Scanner s = new Scanner( System.in );

2.  double liczba1 = Double.parseDouble( s.nextLine() );

3.  s.close();
```

Nie będę tłumaczył co oznacza `System.in`, komenda `new` – wszystko wyjaśni się przy omawianiu struktur obiektowych. Na razie wystarczy Ci informacja, że od momentu zadeklarowania `Scannera`, do wywołania `s.close()`, możemy odczytywać dane z konsoli metodą `s.nextLine()`. Zwraca ona jednak `String` – czyli tekst, który musimy przekonwertować na liczbę typu `double`. Robimy to metodą `Double.parseDouble(String argument)`. Kod takiego programu wygląda następująco:

Zwróć uwagę, że wartość wyniku zamknąłem do dodatkowe nawiasy. Jest to konieczne ze względu na to, że np. w przypadku dodawania operator `+` nie może zostać prawidłowo zinterpretowany (kompilator nie wie czy potraktować go jako konkatenację tekstów, czy dodanie liczb).



**Rysunek 13 - Prosty kalkulator**

```java
public class Kalkulator {

    public static void main(String[] args) {

        double liczba1 = 0;
        double liczba2 = 0;

        Scanner s = new Scanner( System.in );
        liczba1 = Double.parseDouble( s.nextLine() );
        liczba2 = Double.parseDouble( s.nextLine() );
        s.close();

        System.out.println("Dodawanie");
        System.out.println(liczba1 + " + " + liczba2 + " = " + (liczba1+liczba2));
        System.out.println("Odejmowanie");
        System.out.println(liczba1 + " - " + liczba2 + " = " + (liczba1-liczba2));
        System.out.println("Mnożenie");
        System.out.println(liczba1 + " * " + liczba2 + " = " + (liczba1*liczba2));
        System.out.println("Dzielenie");
        System.out.println(liczba1 + " / " + liczba2 + " = " + (liczba1/liczba2));


    }
}
```

cwiczenia.dzien1.podstawy.Kalkulator

Output - SzkolenieJava14 (run)

```
run:
12.1
4
Dodawanie
12.1 + 4.0 = 16.1
Odejmowanie
12.1 - 4.0 = 8.1
Mnożenie
12.1 * 4.0 = 48.4
Dzielenie
12.1 / 4.0 = 3.025
BUILD SUCCESSFUL (total time: 2 seconds)
```

Nie stało się nic nieprzewidzianego, dużo ciekawsze jest jednak operowanie na typie `int`:

∫ +∫ ¿∫ ¿∫ −∫ ¿∫ ¿∫ ¿∫ ¿∫ ¿∫ :∫ ¿∫ !!!

Na poparcie swoich słów dorzucę screen'a:



**Rysunek 14 - Wynik dzielenia int przez int**

```
12  public class JavaApplication3 {
13
14      /**
15       * @param args the command line arguments
16       */
17      public static void main(String[] args) {
18
19          int dzielna = 4;
20          int dzielnik = 8;
21
22          System.out.println( dzielna/dzielnik );
23      }
24  }
25
```

javaapplication3.JavaApplication3 > main

Output - JavaApplication3 (run)

```
run:
0
BUILD SUCCESSFUL (total time: 0 seconds)
```

## Konwersje między typami podstawowymi nazywane też rzutowaniem

A co jeśli chciałbym wpisać `double`'a do `inta`? W przypadku odwrotnym nie nastąpi błąd, ponieważ `double` ma zakres większy niż `int`.

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

    int liczba1 = 4;
    double liczba2 = 8.0;

    double liczba3 = liczba1;

                     incompatible types: possible lossy conversion from double to int
                     ----
                     (Alt-Enter shows hints)
    int liczba4 = liczba2;

}
```

**Rysunek 15 - Próba wpisanie double'a do inta**

Co wtedy? – wtedy muszę użyć konwertowania, nazywanego też rzutowaniem zmiennych – ma ono następującą składnię.

```
1. [typ docelowy] [nazwa] = ([typ docelowy]) [zmienna innego typu]
```

Poprawiam linijkę informującą o błędzie w następujący sposób:

```
27          int liczba4 = (int) liczba2;
```

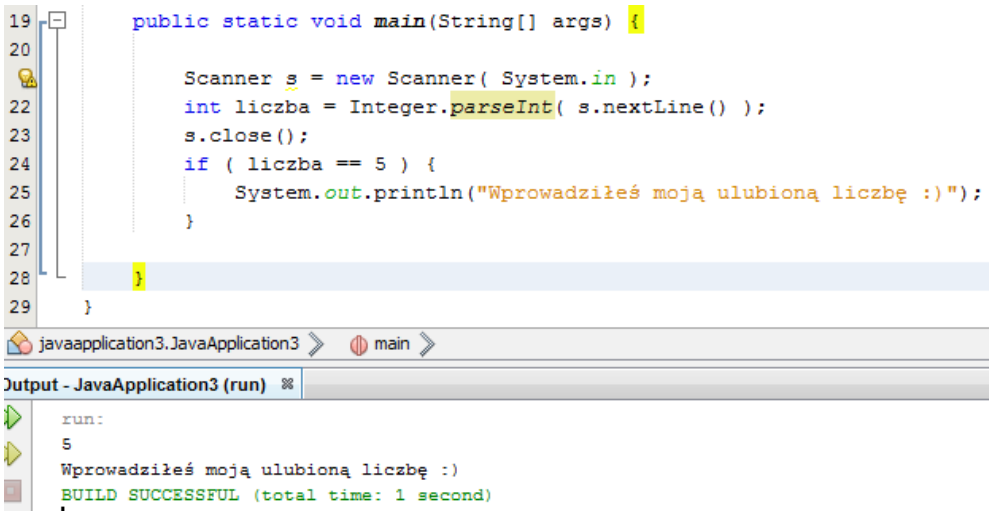
**Rysunek 16 - konwersja double do int**

# Sterowanie programem - warunki

## Podstawowa składnia prostej instrukcji warunkowej

```
1. if ( [wyrażenie typu boolean] ) {
2.     .
3.     . Operacje do wykonania
4.     .
5. }
```

Np. poniższy program przyjmuje od użytkownika jedną liczbę i sprawdza czy jest nią 5, jeśli tak – wyświetla komunikat. Operator porównania w Javie to == (podwójne *równa się*)




Rysunek 17 - Najprostsza instrukcja warunkowa

## Blokowość kodu i zakres widoczności zmiennych

Jeśli napisałeś powyższy kod - zadeklarowałeś blok kodu nawet o nim nie wiedząc. Przypuśćmy, że chcę zmodyfikować powyższy program tak, aby zamiast komunikatu deklarował nową zmienną. Następnie program będzie wypisywał tą nową zmienną na ekran. Otrzymam błąd kompilacji:

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

    Scanner s = new Scanner( System.in );
    int liczba = Integer.parseInt( s.nextLine() );
    s.close();
    if ( liczba == 5 ) {
        int nowaZmienna = 5;
    }

    System.out.println( nowaZmienna );

}
```

cannot find symbol
symbol: variable nowaZmienna
location: class JavaApplication3
---
(Alt-Enter shows hints)

Rysunek 18 - zakres widoczności zmiennych

Powyższy błąd wynika z faktu, że zmienna `nowaZmienna` została zadeklarowana w *bloku-dziecku* bloku metody `main`, wtedy blok nadrzędny nie widzi takiej zmiennej. Blok to po prostu dwie klamry `{ }` np.:

```
public static void main(String[] args) {
    {
        int i = 0;
    }
    System.out.println( i );
}
```

Te klamry nie są kwiatkiem na kożuchu, lecz mają określone działanie. Tak naprawdę po np. warunku wykonywana jest wyłącznie jedna instrukcja (czyli wykonywane jest wszystko do pierwszego średnika ☺).

Blok (czyli te klamry) agreguje kilka instrukcji w pojedynczą komendę (technicznie `Statement`). Dlatego poprawną jest również poniższa składnia (działa tylko przy instrukcjach warunkowych i pętlach):

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

    Scanner s = new Scanner( System.in );
    int liczba = Integer.parseInt( s.nextLine() );
    s.close();
    if ( liczba == 5 )
        System.out.println("Wprowadziłeś moją ulubioną liczbę :)");

}
```

Jeśli jednak po spełnieniu danego warunku chcemy wykonać więcej niż jedną instrukcję, wtedy bezwzględnie musimy stworzyć nowy blok `{ }`.

### Bardziej pokręcone instrukcje warunkowe

W poprzednim przykładzie sprawdzaliśmy tylko jeden warunek pojedynczym `if`'em. Jeśli chcemy sprawdzić warunek przeciwny korzystamy ze słowa kluczowego `else`:



Rysunek 19 - konstrukcja if-else

```
if ( liczba == 5 ) {
    System.out.println("Wprowadziłeś moją ulubioną liczbę :)");
}
else {
    System.out.println("Nie wprowadziłeś mojej ulubionej liczby :(");
}
```

Chcąc sprawdzić kilka warunków po kolei – jeśli liczba = 5 to coś, jeśli liczba = 4 to coś innego, w przeciwnym wypadku jeszcze coś innego nie mamy do dyspozycji dodatkowego słowa kluczowego typu PL/SQL'owego `elsif`, PHP'owego `elseif` czy Bash'owego `elif`. Rozwiązujemy to w inny sposób:

```
if ( liczba == 5 ) {
    System.out.println("Wprowadziłeś moją ulubioną liczbę :)");
}
else if (liczba == 4 ) {
    System.out.println("Wprowadziłeś liczbę mniejszą od mojej"
            + "ulubionej :)");
}
else {
    System.out.println("Nie wprowadziłeś mojej ulubionej liczby :(");
}
```

**Rysunek 20 - rozszerzona konstrukcja if-else**

Tak naprawdę nie zrobiliśmy nic nowego ☺. Blok `if-else` jest w rzeczywistości pojedynczą instrukcją. Program najpierw sprawdza czy `liczba = 5`, jeśli nie wykonuje się blok `else` – czyli kolejny blok `if-else`. Sprytne użycie struktury blokowej programu pozwoliło zachować pełną funkcjonalność bez wprowadzania dodatkowych słów kluczowych.

## Operatory logiczne

**Operatory porównania:**

¿=−równe
¿−większe
¿−mniejsze
¿−większe lub równe
¿−mniejsze lub równe
!=−nierówne

**Operatory łączenia warunków:**

$$¿\wedge - koniunkcja(spójnik\,'i')$$
$$¿\vee - alternatywa(spójnik\,'lub')$$
$$! - negacja('\ni e')$$

Przykład użycia operatorów z pierwszej grupy:

```
if ( liczba != 5 ) {
    System.out.println("Wprowadziłeś moją ulubioną liczbę :)");
}
else if (liczba <= 4 ) {
    System.out.println("Wprowadziłeś liczbę mniejszą od mojej"
            + "ulubionej :)");
}
else {
    System.out.println("Nie wprowadziłeś mojej ulubionej liczby :(");
}
```

Oraz z pierwszej i drugiej grupy:

```
if ( liczba >= 5 && liczba <= 10 ) {
    System.out.println("Wprowadziłeś moją ulubioną liczbę :)");
}
else if ( !( liczba != 4 && liczba != 11 ) ) {
    System.out.println("Wprowadziłeś liczbę mniejszą od mojej"
            + "ulubionej :)");
}
else {
    System.out.println("Nie wprowadziłeś mojej ulubionej liczby :(");
}
```

**Rysunek 21 - łączenie warunków za pomocą koniunkcji, alternatywy i negacji**

Negacja w drugim warunku odwraca wartość warunku wewnętrznego (`!false = true` analogicznie `!true = false`), dlatego można ten warunek przekształcić do postaci:

```
1.  else if (liczba == 4 || liczba == 11)
```

(Prawa de Morgana).

**BADZO WAŻNE!**

Za pomocą zwykłego operatora porównania `==` nie można porównywać tekstów (`String`). Dlaczego? Bo `String` jest **obiektem**! Służy do tego metoda `equals()` która wykonuje porównanie obiektowe.

Każdy blok `if – else if – else` wykonuje się liniowo. Ponadto jego wykonanie zostanie przerwane po znalezieniu pierwszego pasującego warunku. Jeśli nie chcemy aby tak się stało używamy instrukcji `switch`, która ma następującą składnię:

```
1.  switch ( [zmienna do sprawdzenia] ) {
2.
3.      case [wartość]:
4.          .
5.          . OPERACJE
6.          .
7.      case [inna wartość]:
8.          .
9.          . OPERACJE
10.         .
11.         break; // przerwanie sprawdzania
12.     default:
13.         .
14.         . Operacje wykonane jeśli
15.         . żaden warunek nie został spełniony
16.         .
17. }
```

Np.:

```
switch (liczba) {
    case 1:
        System.out.println("");
        System.out.println("HEYYYA HOO :)");
        System.out.println("");
    case 2:
        break;
    default:
        System.out.println("Nic nie pasuje...");
}
```

**Rysunek 22 - instrukcja switch**

Jest jednak stosowana bardzo rzadko, ze względu na swoją nieczytelność, ale także poważne ograniczenia. Za pomocą `switch`'a można sprawdzać wyłącznie wartości jakiejś zmiennej i tylko typu `String`, `int` lub `Integer`.

# Pętle - podstawa algorytmiki

## Czym jest pętla?

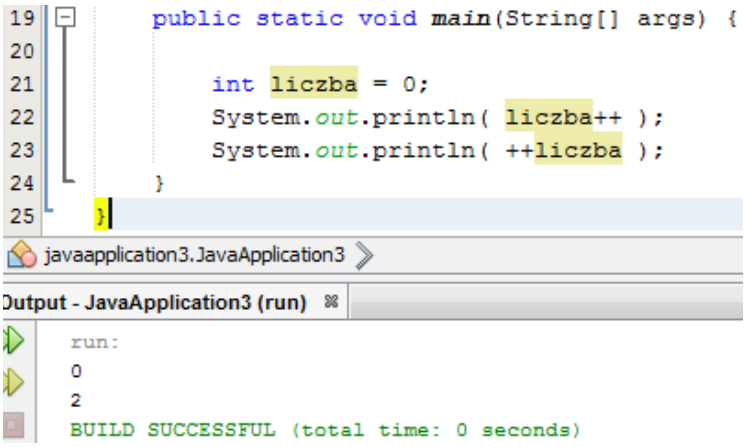
Pętla jest konstrukcją algorytmiczną i oznacza powtarzanie jakiś akcji dopóki spełniony jest jakiś warunek, lub po prostu określoną ilość razy. Są potężnym i podstawowym narzędziem algorytmicznym. W Javie stworzone zostały trzy rodzaje pętle `do-while`, `while` oraz `for`.

## Operatory inkrementacji i dekrementacji

W kontekście pętli nie można nie wspomnieć o dwóch najczęściej wykonywanych operacjach na liczbach przez programistów. Są to: zwiększenie zmiennej o 1 i zwrócenie wartości (inkrementacja) i zmniejszenie zmiennej o 1 i zwrócenie wartości (dekrementacja). Wykonujemy je tak często, że dostały własne operatory:

```
1.  int liczba = 1;
2.
3.  liczba++; // postinkrementacja ---> i = i + 1;
4.  ++liczba; // preinkrementacja
5.
6.  liczba--; // postdekrementacja  ---> i = i - 1;
7.  --liczba; // predekrementacja
```

Różnicę między `++liczba` i `liczba++` obrazuje poniższy przykład:




**Rysunek 23 - postinkrementacja i preinkrementacja**

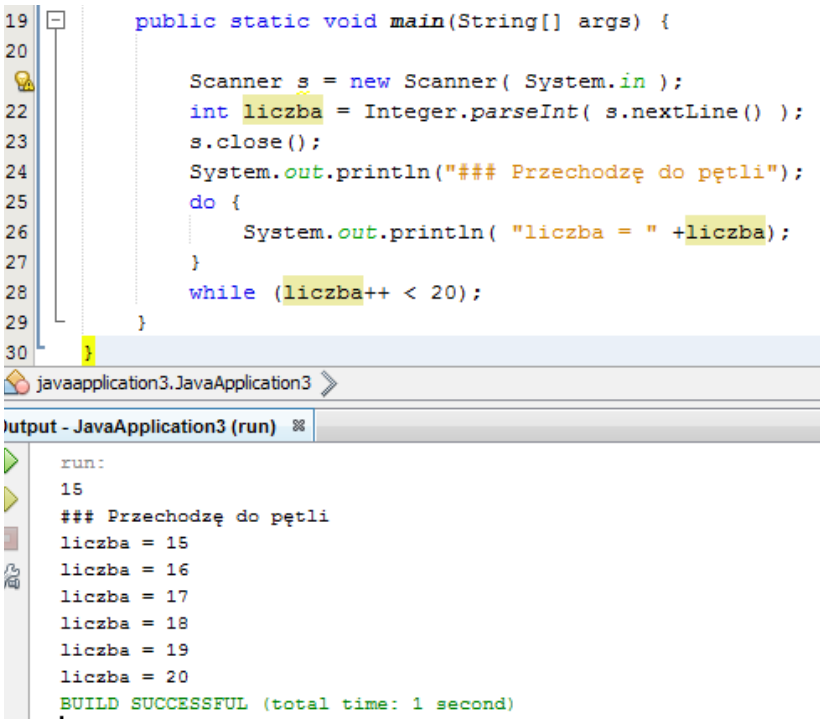
Różnica polega na tym, że postinkrementacja zwraca wartość zmiennej przed zwiększeniem, a preinkrementacja zwraca wartość już zwiększoną. Dla dekrementacji wygląda to analogicznie.

### Rzadka pętla do-while

Pętla `do-while` ma następującą składnię:

```
1.  do {
2.        .
3.        .  OPERACJE
4.        .
5.  }
6.  while ( [wyrażenie typu boolean] );
```

Np.:




**Rysunek 24 - przykład pętli do-while**

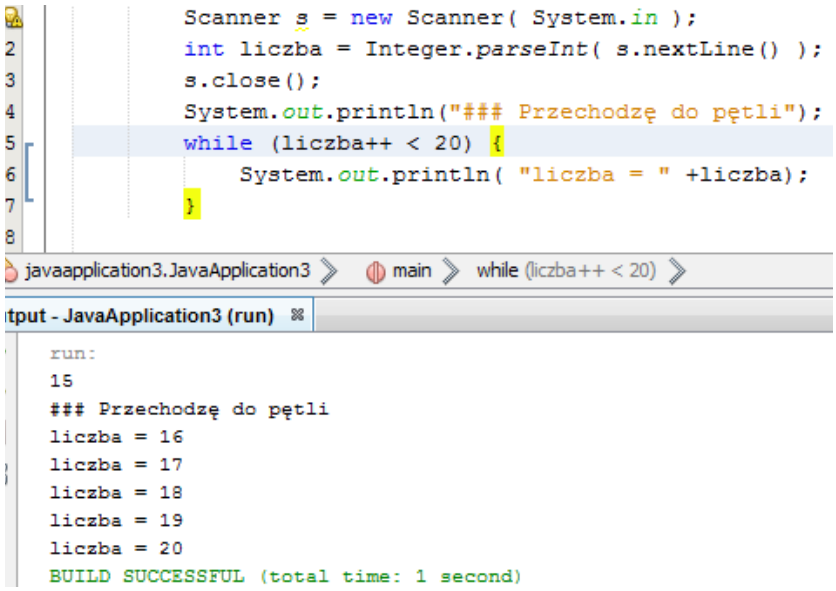
Jej działanie polega na tym, że wykonuje ona swoje ciało – czyli zawartość między klamrami po słowie `do`, a później sprawdza warunek określny w nawiasach po słowie `while`. Jeśli warunek jest spełniony, ciało zostanie wykonane ponownie, ponownie zostanie sprawdzony warunek, jeśli warunek jest spełnio… ☺. Ważne w pętli `do-while` jest to, że jej ciało jest wykonywane co najmniej raz.

### Bardzo ważna pętla while

Pętla `while` działa prawie tak samo jak pętla `do-while`, różnica polega na tym, że najpierw sprawdzany jest warunek, a dopiero potem wykonywane jest ciało (w pętli `do-while` jest odwrotnie). Jej składnia jest następująca:

```
1.  while ( [wyrażenie typu boolean] ) {
2.      .
3.      .  OPERACJE
4.      .
5.  }
```

Powyższy przykład przekształcony do pętli `while`:




**Rysunek 25 - przykład z pętlą while**

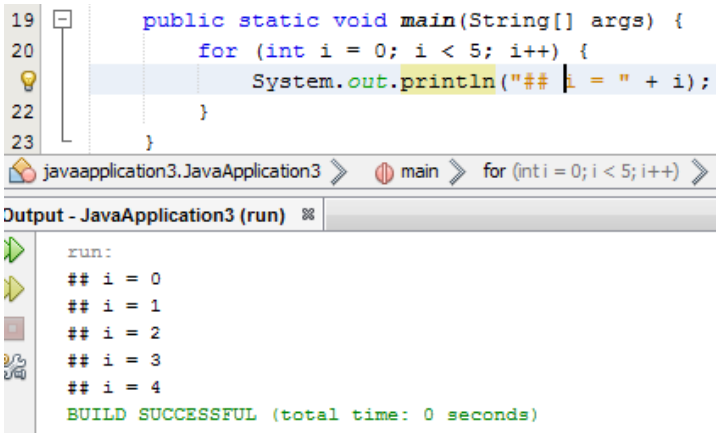
### Najczęściej używana pętla for

Pętla `for` jest najczęściej używaną pętlą ze względu na swoją elastyczność. Jeżeli programowałeś/aś w PL/SQL wiesz, że pętla `for` wykonuje się od jakiejś wartości do jakiejś innej wartości. Pętla for w Javie (podobnie zresztą jak w C i C++) jest inteligentniejsza. Javowy `for` to tak naprawdę `while`, ale tak zdefiniowany, że potrafi stworzyć warunek zakończenia pętli już na etapie jej deklarowania. Składnia `for`'a wygląda następująco:

```
1.  for ( [wyrażenie typu void]; [wyrażenie typu boolean]; [wyrażenie typu void] ) {
2.      .
3.      .  OPERACJE
4.      .
```

```
5.  }
```

Np.:




Rysunek 26 - Przykład pętli for

Działa to w ten sposób, że pierwszy element `for`'a (`int i = 0`), jest deklarowany przed wywołaniem pętli. Następnie sprawdzany jest warunek (`i < 5`), jeśli jest spełniony to wykonywane jest ciało pętli. Po każdym obrocie wywoływana jest trzecia instrukcja `for`'a (`i++`)

**WAŻNE!**

Żaden z elementów znajdujących się w nawiasach po słowie `for` nie jest obowiązkowy, pętla z powyższego przykładu mogłaby wyglądać równie dobrze tak:

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

Albo nawet tak:

```
1.  int i = 0;
2.  for ( ; i < 5 ; ) {
3.
4.      System.out.println( "## i = " + i);
5.      i++;
6.  }
```

Teraz już widać, że pętla `for` to tak naprawdę pętla `while`, ale z możliwością stworzenia warunku zakończenia pętli na etapie jej deklarowania. Dopuszczalna jest nawet taka składnia:

```
1.  for ( ;; ) {
2.      System.out.println( "## Pętla nieskończona! " );
3.  }
```

### Przykłady zastosowania pętli while i ich transformacje do pętli for

Tak jak powiedziałem przy poprzednim przykładzie pętla for to taki rozszerzony `while`. Poniżej przedstawię dwa przykłady z pętlą `while` i ich transformacje do pętli `for`:

Poniższy kod przyjmuje liczbę z konsoli i rozkłada ją na czynniki pierwsze:

```
1.  Scanner s = new Scanner( System.in );
2.  int liczbaDoRozlozenia = Integer.parseInt( s.nextLine() );
3.  s.close();
4.
5.  int dzielnik = 2;
6.  while (dzielnik <= liczbaDoRozlozenia) {
7.      if ( liczbaDoRozlozenia % dzielnik == 0 ) {
8.          System.out.println( dzielnik );
9.          liczbaDoRozlozenia /= dzielnik;
10.     }
11.     else {
12.        dzielnik++;
13.     }
14. }
```

Są tu w zasadzie tylko dwie rzeczy do wyjaśnienia. Operator `%` oznacza tzw. dzielenie modulo czyli obliczenie reszty z dzielenia. Natomiast operator `/=` oznacza jednoczesne podzielenie i przypisanie wartości czyli `liczba /= dzielnik <==> liczba = liczba / dzielnik`. Oczywiście analogicznie wyglądają inne operatory tego typu (+=, −=, *=).

Teraz to samo za pomocą pętli `for`:

```
1.  Scanner s = new Scanner( System.in );
```

```
2.  int liczbaDoRozlozenia = Integer.parseInt( s.nextLine() );
3.  s.close();
4.
5.
6.  for (int dzielnik = 2; dzielnik <= liczbaDoRozlozenia; ) {
7.      if ( liczbaDoRozlozenia % dzielnik == 0 ) {
8.          System.out.println( dzielnik );
9.          liczbaDoRozlozenia /= dzielnik;
10.     }
11.     else {
12.        dzielnik++;
13.     }
14. }
```

Kolejny przykład:

Poniższy kod oblicza ilość lat potrzebnych do osiągnięcia określonego kapitału na lokacie:

```
1.  double kapitalPoczatkowy = 1000;
2.  double kapitalDocelowy = 10000;
3.  double oprocentowanie = 0.04;
4.
5.  int iloscLat = 0;
6.  while ( kapitalPoczatkowy < kapitalDocelowy ) {
7.      kapitalPoczatkowy *= (oprocentowanie+1);
8.      iloscLat++;
9.  }
10. System.out.println( "Potrzebne było " + iloscLat + " lat." );
```

Teraz to samo `for`'em:

```
1.  double kapitalPoczatkowy = 1000;
2.  double kapitalDocelowy = 10000;
```

```
3.  double oprocentowanie = 0.04;
4.
5.
6.
7.  int iloscLat = 0;
8.  for (; kapitalPoczatkowy < kapitalDocelowy; iloscLat++ ) {
9.      kapitalPoczatkowy *= (oprocentowanie+1);
10. }
11. System.out.println( "Potrzebne było " + iloscLat + " lat." );
```

# Tablice - najprostsze zbiory zmiennych

## Czym jest tablica?

Tablicą nazywamy zbiór o określonej długości oraz, zdolny do przechowywania zmiennych określonego typu. O ile zmienną możemy wyobrażać sobie jako zwyczajne pudełko na jakąś wartość, to tablica jest pudełkiem z przegródkami (każda przegródka ma swój numer, czyli indeks), ale na samym początku musimy określić ilość tych przegródek.

Rysunek 27 - Zmienna

Rysunek 28 - Tablica

## Różne sposoby deklarowania tablic jednowymiarowych

Zaczniemy od wprowadzenia podstawowego sposobu deklarowania tablicy:

```
1.  [typ przechowywany] [] [nazwa tablicy] = new [typ przechowywany][ilość elementow]
```

Np.:

```
21          int[] tablicaIntow = new int[10];
```

Dopuszczalna w większości przypadków jest również składnia charakterystyczna dla C, C++:

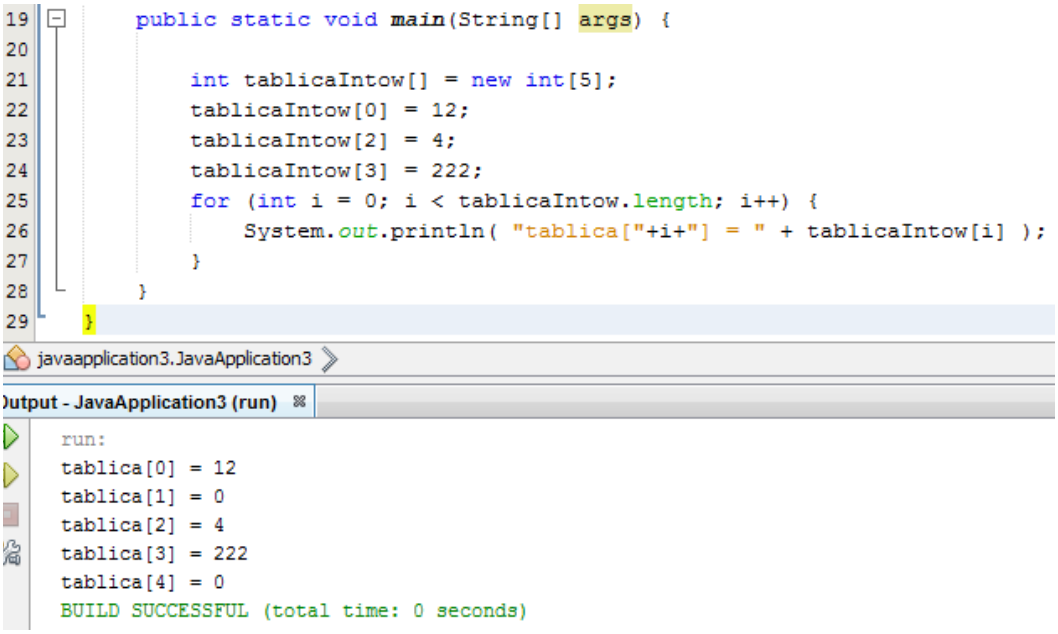
```
21          int tablicaIntow[] = new int[10];
```

**WAŻNE!**
Tablice w Javie są obiektami! Wrócimy do tego później.

Wpiszemy teraz coś do naszej tablicy. W przeciwieństwie do np. PL/SQL czy PHP, w Javie indeksem tablicy może być wyłącznie liczba.



Rysunek 29 - Modyfikujemy tablicę i wypisujemy jej zawartość



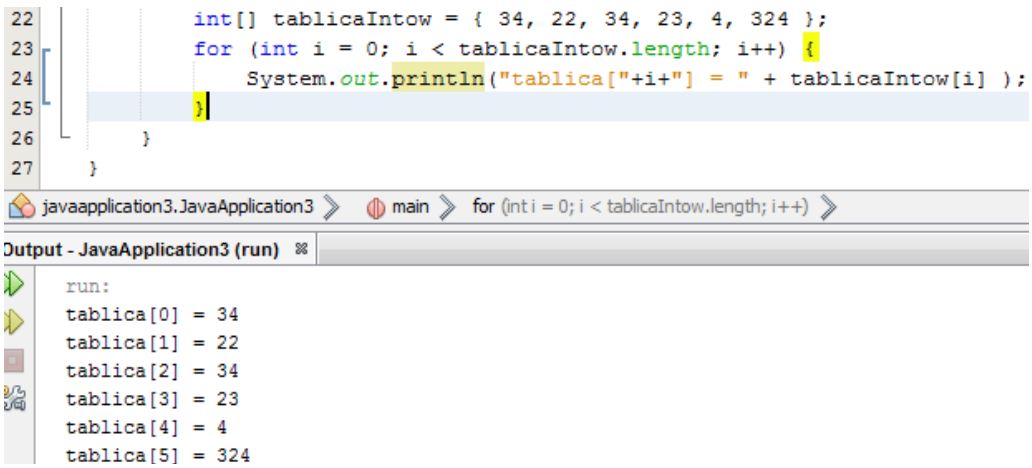

Pierwsza rzecz – elementy tablicy (w tym przypadku jednowymiarowej) indeksujemy od zera więc dla tablicy o długości 5 indeksy są liczbami całkowitymi z przedziału `[0, 4]`.

Po drugie – tablice w Javie są nieco inteligentniejsze od tych znanych z C, C++, ponieważ znają swój rozmiar.

I na koniec, powiedziałem, że w Javie nie ma czegoś takiego jak wartości domyślna zmiennej. Natomiast tworząc 5 – elementową tablicę `int`'ów zadeklarowaliśmy 5 zmiennych typu `int`. Skąd więc na indeksach `1` i `4` wartość `0` ?!. Odpowiedź wynika z C++. Po alokacji tablicy na stercie, w C++ należy zainicjować wszystkie elementy tablicy, aby móc później jej bezpiecznie używać. Typ `int` w C++ (podobnie jak w Javie zresztą) nie może przyjąć wartości `NULL` (bo nie przechowuje wskaźnika tylko wartość). Stąd biorą się te dziwaczne `0` ☺.

### Deklarowanie tablicy i jednoczesne wpisywanie wartości do elementów

Istnieje jeszcze jeden sposób deklarowania tablicy, który od razu tworzy tablicę odpowiedniego rozmiaru i jednocześnie wpisuje wartości do jej elementów. Wygląda to następująco:




### Tablice wielowymiarowe i ich inicjowanie, tablice szarpane

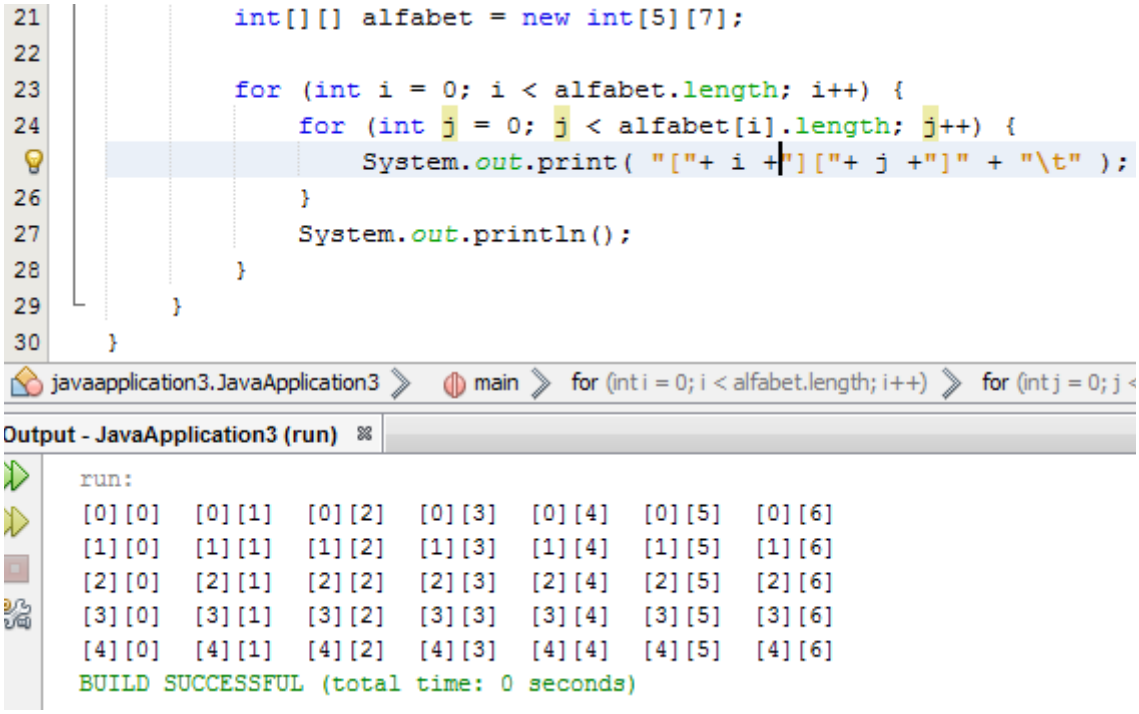
To tej pory używaliśmy tylko tablic jednowymiarowych, czyli posiadających tylko długość. Wprowadzimy sobie teraz tablice wielowymiarowe. Np. tablica dwuwymiarowa (używamy ich np. do tworzenia `JTable` w Swingu) typu `char`:

```
21 |       char[][] alfabet = new char[7][12];
```

OK. Czyli moja tablica ma 7 wierszy i 12 kolumn – powstała mi tabelka znaków. Co w ogóle technicznie oznacza, że tablica jest dwuwymiarowa? Technicznie nie ma czegoś takiego jak tablica wielowymiarowa, ale tworząc tablicę tablic uzyskujemy taki efekt. Na poziomie **C++** wyglądałoby to tak (wskaźnik na wskaźnik ☺):

```
1.  char** tablicaDwuwymiarowa = NULL;
```

Indeksy tablicy dwuwymiarowej przedstawiają się jak poniżej:




**Rysunek 30 - indeksy tablicy dwuwymiarowej**

Posiadając wiedzę z poprzedniego akapitu jasna powinna być już poniższa podwójna pętla:

```
for (int i = 0; i < alfabet.length; i++) {
    for (int j = 0; j < alfabet[i].length; j++) {
        
    }
}
```

**Rysunek 31 - podwójna pętla iterująca po tablicy dwuwymiarowej**

Teraz, żeby wpisać coś do mojej tablicy muszę odwołać się do obu „współrzędnych”:

```
21          char[][] alfabet = new char[7][12];
22
23          for (int i = 0; i < alfabet.length; i++) {
24              for (int j = 0; j < alfabet[i].length; j++) {
25                  char znak = (char)(alfabet.length*i + 'A' + j);
26                  alfabet[i][j] = znak;
27                  System.out.print( znak + " " );
28              }
29              System.out.println();
30          }
31      }
32  }
```

javaapplication3.JavaApplication3 > main >

Output - JavaApplication3 (run)

```
run:
A B C D E F G H I J K L
H I J K L M N O P Q R S
O P Q R S T U V W X Y Z
V W X Y Z [ \ ] ^ _ ` a
] ^ _ ` a b c d e f g h
d e f g h i j k l m n o
k l m n o p q r s t u v
BUILD SUCCESSFUL (total time: 0 seconds)
```

Oczywiście operacje z linijki `25` powinny być jasne (dlaczego do znaku można dodać liczbę).

W tej chwili nasza tablica jest tabelką 7x12. Nie chciałbym, abyś miał/a mylne przeświadczenie, że tablice mogą być wyłącznie prostokątami. Po pierwsze, mogą mieć dowolną skończoną ilość wymiarów. Po drugie, nie muszą mieć stałego wymiaru.

Spójrz na poniższy przykład:

```
21          int[][] alfabet = {
22                                { 12, 436, 643, 3, 46, 456, 54, 65, 46},
23                                { 3534, 34, 457, 5 },
24                                { 234, 5, 435, 53, 3, 45 }
25                              };
26
27          for (int i = 0; i < alfabet.length; i++) {
28              for (int j = 0; j < alfabet[i].length; j++) {
29                  System.out.print( alfabet[i][j] + "\t" );
30              }
31              System.out.println();
32          }
33      }
34  }
```

javaapplication3.JavaApplication3 >

Output - JavaApplication3 (run)

```
run:
12      436     643     3       46      456     54      65      46
3534    34      457     5
234     5       435     53      3       45
BUILD SUCCESSFUL (total time: 0 seconds)
```

**Rysunek 32 - Tablica szarpana**

Tablice, które nie mają stałego wymiaru nazywamy tablicami szarpanymi.

### Pętla for-each

Zaczynając zabawę ze zbiorami (na razie tylko w postaci tablic – podobnych struktur jest sporo więcej), nie sposób nie wspomnieć o pętli `for-each`. Jest to skrócona składnia `for`'a, który iteruje po wszystkich elementach tablicy. Czyli zamiast:

```
1.  for (int i = 0; i < tablica.length; i++) {
2.
3.      System.out.println( tablica[i] );
4.
5.  }
```

napiszemy:

```
1.  for (int element: tablica) {
2.
3.      System.out.println( element );
4.
5.  }
```

Działa to w ten sposób, że nie musimy już za każdym razem odwoływać się do elementu tablicy po indeksie, tylko dostajemy za darmo zmienną element, która reprezentuje każdy kolejny element tablicy. Niestety `for-each` ma jedno **ograniczenie**. Nie zmodyfikujemy tablicy pisząc następującą komendę:

```
1.  element = 12;
```

# Metody w Javie

## Pojęcie metody i po co są? Deklarowanie metod

Metodą nazywamy zamknięty i samodzielny fragment kody zdolny do zwracania wartości i przyjmowania argumentów. Poznaliśmy już jedną metodę – `main(String[] args)` i nie poruszaliśmy się poza nią. Zmienimy zaraz ten stan rzeczy. Zadeklarujemy bowiem własną metodę, ale najpierw przedstawię składnię takiej deklaracji (na razie będziemy używać tylko metod publicznych statycznych - co to znaczy wyjaśnię później):

```
1. public static [typ zwracany] [nazwa metody]([zestaw argumentów]) {
2.      .
3.      .  OPERACJE
4.      .
5. }
```

Np.:

```
16      public static int zwrocLiczbe() {
17          System.out.println("## Jestem w metodzie zwrocLiczbe");
18          return 21;
19      }
```

Rysunek 33 - deklaracja pierwszej metody

**WAŻNE!**

Metody deklarujemy zawsze bezpośrednio w klasach! Nigdy nie znajdziesz ich w innych miejscach.

Przykład innej deklaracji i wywołania:

```
14  public class JavaApplication3 {
15
16      public static int zwrocLiczbe( int argument) {
17          System.out.println("## Jestem w metodzie zwrocLiczbe");
18          return argument;
19      }
20
21      public static void main(String[] args) {
22          System.out.println("## Jestem w metodzie main");
23          System.out.println( zwrocLiczbe( 21 ) );
24
25      }
26  }
```

javaapplication3.JavaApplication3

Output - JavaApplication3 (run)

```
run:
## Jestem w metodzie main
## Jestem w metodzie zwrocLiczbe
21
BUILD SUCCESSFUL (total time: 0 seconds)
```

Rysunek 34 - deklaracja i wywołanie metody

Porównajmy same deklaracje metod. Na rysunku 33 metoda `zwrocLiczbe` nie przyjmuje żadnych argumentów (wtedy jej wywołaniem byłoby po prostu `zwrocLiczbe()`), natomiast na rysunku 34 przyjmuje jeden argument typu `int`, który w ciele mojej metody widzę w postaci zmiennej o nazwie `argument`. Zwrócenie wartości przez metodę wstawia zwracaną wartość w miejsce wywołania.

**WAŻNE!**

Zwrócenie wartości przez metodę jest równoznaczne z natychmiastowym przerwaniem jej wykonywania. Ponadto, jeśli zadeklarujemy, że metoda będzie zwracała wartość danego typu to musi taki typ zwrócić – jeśli tego nie robi program się nie skompiluje.

Wartości jakiego typu może zwracać metoda? Każdego ☺ Co więcej, metoda może nie zwracać nic i wtedy jej typ oznaczamy jako `void`. Np.:

```
21      public static void fun() {
22          System.out.println("## Jestem w metodzie fun");
23          for (int i = 0; i < 10; i++) {
24              System.out.print( i + "\t");
25          }
26      }
27      public static void main(String[] args) {
28          System.out.println("## Jestem w metodzie main");
29          fun();
30      }
31  }
```

javaapplication3.JavaApplication3

utput - JavaApplication3 (run)

```
run:
## Jestem w metodzie main
## Jestem w metodzie fun
0       1       2       3       4       5       6       7       8
```

**Rysunek 35 - metodo nie zwracająca nic - typ void**

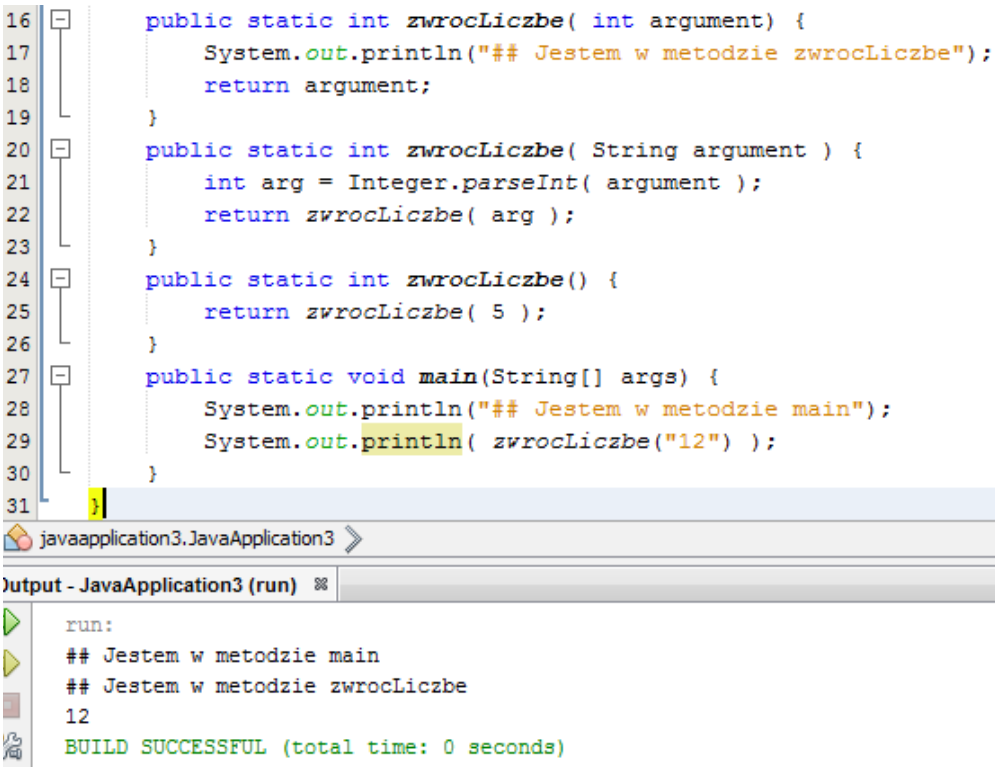
W metodach typu `void` również możemy używać komendy `return` ale nie podajemy zmiennej, lub literału do zwrócenia, piszemy po prostu `return;` - co oznacza przerwanie metody w tej linijce.

## Przeciążanie metod

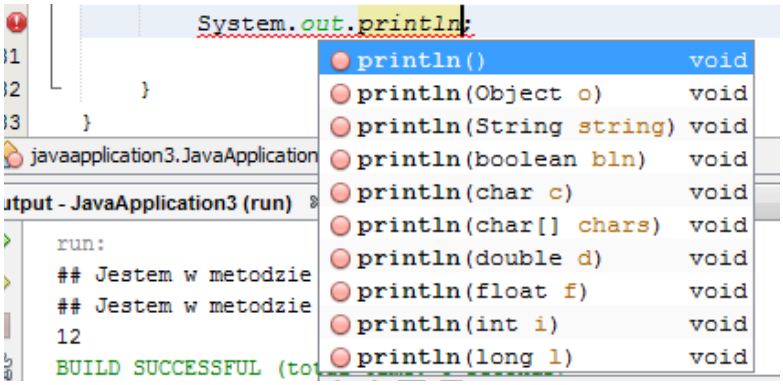
Czasem chcemy, żeby nasza metoda przyjmowała, powiedzmy, dziesięć argumentów, ale siedem z nich ma przyjmować wartość domyślną, tak, abym nie musiał zawsze powtarzać pełnej konfiguracji. Albo chcę, żeby moja metoda była w stanie przyjąć jako argument jednego `int`'a, i tablicę `int`'ów. Mogę oczywiście stworzyć podobnie nazywające się metody, ale nie byłoby to eleganckie rozwiązanie, ponieważ mogę dwie metody nazwać tak samo!



**Rysunek 36 - Przeciążanie metod**




Jest to konstrukcja bardzo charakterystyczna dla Javy, ponieważ w Javie metody nie muszą mieć unikalnych nazw. Jednoznacznym identyfikatorem dla metody jest dopiero nazwa metody i zestaw argumentów. Zadeklarowanie metody o tej samej nazwie, a innym zestawie argumentów, nazywamy przeciążaniem metod i bardzo często to wykorzystujemy. Jeśli chcesz zobaczyć jak wiele jest przeciążeń metody `println()`, którą wykorzystujemy bez przerwy, wpisz `System.out.println` i naciśnij `Ctrl + Spacja`.



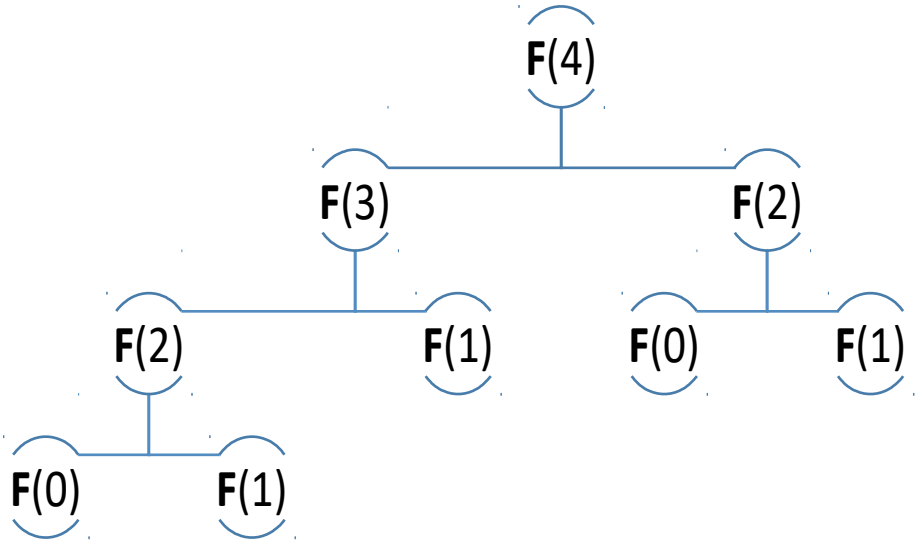
# Rekurencja czyli problem – matrioszka

## Pojęcie rekurencji (rekursji)

Może pamiętasz z matematyki coś takiego jak indukcja matematyczna (metoda udowadniania twierdzeń). Chodziło w niej o to, że posiadając udowodnione twierdzenie dla kilku wartości można było wykazać prawdziwość twierdzenia dla każdej kolejnej wykorzystując prawdziwość dla wartości poprzednich. Jest to właśnie istota rekurencji. Troszkę bardziej obrazowym przykładem jest np. ciąg Fibonacciego który definiujemy następująco:

$$F_n = \begin{cases} F_0 = 0 \\ F_1 = 1 \\ F_n = F_{n-1} + F_{n-2} \end{cases}$$

Zauważ jak




wyglądałoby obliczenie `F(4)`:

Tak musiałbyś/musiałabyś to rozpisać, żeby obliczyć wynik. Komputer liczy takie rzeczy dokładnie tak samo. Zwróć jednak uwagę, że wartość `F(2)` liczymy dwa razy. Człowiek zapamięta wynik i kolejne obliczenie `F(2)` nie sprawi mu problemu. Komputer jednak tego nie potrafi i ponowi obliczanie tego

co liczył przed kilkoma ułamkami sekundy. Nie w tym jednak tkwi problem. Pomyśl jak wielkie byłoby drzewko dla `F(100)`. Każde kółeczko w powyższym grafie to wartość przechowywana na stosie (a dokładniej mówiąc odwołanie), który jak wiemy jest ograniczony – dlatego dla rozbudowanych rekursji może dojść do przepełnienia stosu.

Wnioski nasuwają się same:

- Rekurencja bardzo mocno obciąża pamięć
- Jest niebezpieczna ze względu na zapychanie stosu (`StackOverFlowError`)
- Zwykle jest wolna, komputer musi rozwiązywać mnóstwo odwołań.

Ale:

- Wszystko to, co da się napisać pętlą, da się też załatwić rekursją – w drugą stronę to nie działa
- Często pozwala łatwo rozwiązywać skomplikowane problemy
- Najszybszy algorytm sortowania tablic jest rekurencyjny ☺ (*Algorytm QuickSort z 1962r.*)

### Implementowanie wzorów rekursywnych

Zaimplementujemy teraz opisany wyżej ciąg Fibonacciego w metodzie o nazwie `fib( int n )`.

```
16      public static int fib( int n ) {
17          if ( n == 0 ) {
18              return 0;
19          }
20          if ( n == 1 ) {
21              return 1;
22          }
23          return fib( n-1 ) + fib( n-2 );
24      }
```

**Rysunek 37 - pierwsza metoda wykorzystująca rekursję**

Jak widzisz nie ma błędu. Metoda w swoim ciele widzi samą siebie. Czyli dwa wywołania `fib(4)` zostanie wywołana metoda `fib(3)` i `fib(2)` – dokładnie tak jak na rysunku.

Analogicznie mógłbym opisać obliczanie silni:

$$n!=\begin{cases}0!=1\\1!=1\\n!=n*(n-1)*\ldots*2*1\end{cases}$$

Czyli:

$$5!=5*4*3*2*1$$
$$4!=4*3*2*1$$
$$3!=3*2*1$$

A z tego łatwo zauważyć, że:

$$5!=5*4!$$
$$4!=4*3!$$
$$n!=n*(n-1)!$$

Zatem moja metoda będzie wyglądać następująco:

```
1.  public static int factorial( int n ) {
2.      if ( n == 0 ) {
3.          return 1;
4.      }
5.      return n * factorial( n-1 );
6.  }
```

Oczywiście takie przykłady są dość wymuszone ☺ Do obliczania wyrazów ciągu Fibonacciego mamy wzór Eulera – Bineta, a do silni wzory Stirlinga, oddają one jednak doskonale istotę rzeczy. Jeśli chodzi zaś o niematematyczny przykład zastosowania rekursji to zastanów się jak wyglądałby algorytm do wypisywania zawartości drzewa katalogów i plików z dysku twardego. Takiego problemu nie rozwiązalibyśmy stosując pętle. Konieczne byłoby wykorzystanie rekurencji.

# Obiektowość w Javie

## Krótki wstęp do programowania obiektowego

Do tej pory nie używaliśmy obiektów (oprócz tablic, tak naprawdę nawet nie bardzo zdając sobie z tego sprawę, że są obiektami), a więc nie korzystaliśmy (świadomie ☺) z elementów posiadających jakąś wewnętrzną strukturę. Czas przejść na wyższy poziom abstrakcji – zacząć programować obiektowo. Java jest językiem niemal całkowicie obiektowym i prawie wszystko w Javie jest obiektem (z wyjątkiem tych kilku typów, o których powiedziałem, że nimi nie są).

OK, wszystko fajnie, ale do tej pory programowałem/am proceduralnie i jestem bardzo zadowolony/a.

Tak, ale żaden z Twoich programów nie był hermetyczny. Wszystko dało się zmodyfikować z każdego miejsca w programie. Brak było wydzielenia odpowiedzialności (dostęp do bazy danych, odbiór danych od użytkownika) poszczególnych fragmentów aplikacji. To wszystko daje programowanie obiektowe (*OOP – Object Oriented Programming*).

## Na czym polega różnica między obiektem, a nieobiektem

Tak jak wspomniałem typ nieobiektowy jest tak prosty, że nie posiada, żadnej wewnętrznej struktury (bo i jaką strukturę mogłoby mieć `5` albo `'a'`). Obiekt tablicy dowolnego typu posiada w sobie zmienne oraz informacje o długości tablicy (i to jest ta jego wewnętrzna struktura). Zabawmy się w zgadywankę ☺ Ja zadeklaruję i zainicjuję zmienną a Ty odpowiedz jaka jest jej wartość. Zaczynamy!

**Zagadka nr 1:**

```
1.  int liczba = 5;
```

Odpowiedź jest oczywista – 5.

**Zagadka nr 2:**

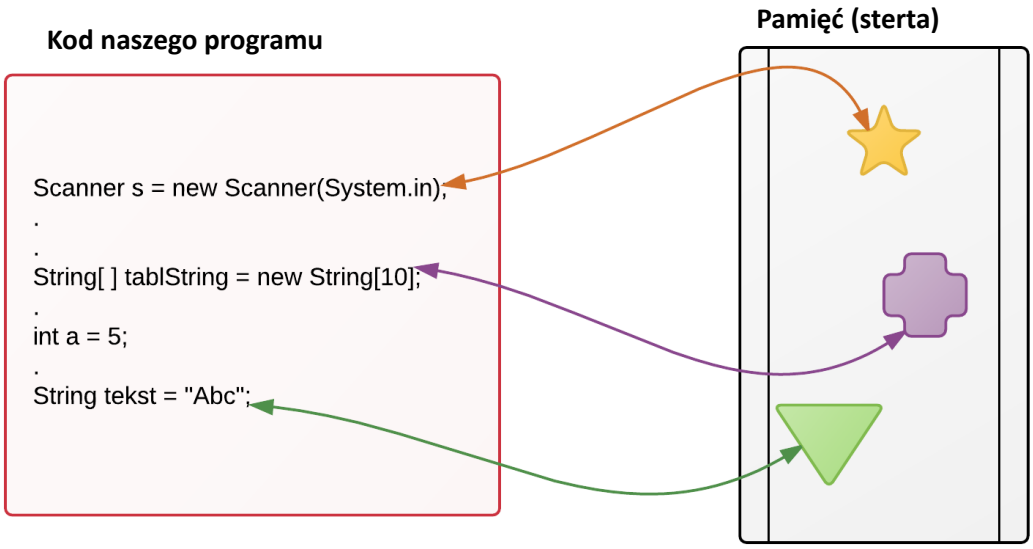
```
1.  String tekst = "abc";
```

Zapewne odpowiedziałeś/aś – tekst abc. Guzik prawda! Ale niech by nawet tak było…

**Zagadka nr 3:**

```
1.  Scanner s = new Scanner( System.in ); // yyyyyyyyyy... :)
```

Noooooo właśnie… jaka wartość znajduje się w zmiennej `s`? Nawet jeśli wypiszesz ją na ekran nie zobaczysz jej prawdziwej zawartości ☺. Otóż znajduje się w niej referencja do pamięci (do sterty będąc precyzyjnym), do miejsca, gdzie znajduje się obiekt `Scanner`. I to jest techniczna różnica między obiektem a typem prostym. Wartością zmiennej typu prostego jest po prostu jego wartością, natomiast wartością zmiennej typu obiektowego jest referencja w pamięć.




Rysunek 38 - Istota pojęcia referencji

Zapoznaj się teraz z poniższym przykładem, ponieważ jest to test na sprawdzenie obiektowości jakiegoś typu, a przy okazji ujawnia pewien szczegół związany z wywoływaniem metod.

```
16      public static void modyfikujObiekt( int[] tab ) {
17          tab[0] = 12;
18      }
19      public static void modyfikujNieobiekt( int liczba ) {
20          liczba = 4;
21      }
22      public static void main(String[] args) {
23          int[] tablica = { -2 };
24          int liczba = 0;
25          modyfikujObiekt( tablica );
26          modyfikujNieobiekt( liczba );
27          System.out.println("Tablica: " + tablica[0]);
28          System.out.println("Liczba: " + liczba);
29      }
30  }
```

```
javaapplication3.JavaApplication3
utput - JavaApplication3 (run)
run:
Tablica: 12
Liczba: 0
BUILD SUCCESSFUL (total time: 0 seconds)
```

Rysunek 39 - Programowa różnica między referencją, a typem prostym

Zauważ, że wartość zmiennej typu `int` nie uległa zmianie, natomiast tablica jak najbardziej. Dlaczego? Bo tablica jest obiektem, a `int` nie! Warto w tym miejscu powiedzieć o drobnym szczególe związanym z wywoływaniem metod.

Gdy wywołujesz metodę, a jako argument podasz jakąś wartość, albo zmienną, do metody nie trafi wartość jako ona sama, tylko jej kopia. Dlatego metoda `modyfikujNieobiekt()` pracuje na zupełnie innej zmiennej. A metoda `modyfikujObiekt()`? Też! Ale co z tego, że skopiuję referencję skoro kopia pokazuje dokładnie w to samo miejsce w pamięci – czyli pracuję na tym samym **obiekcie**!

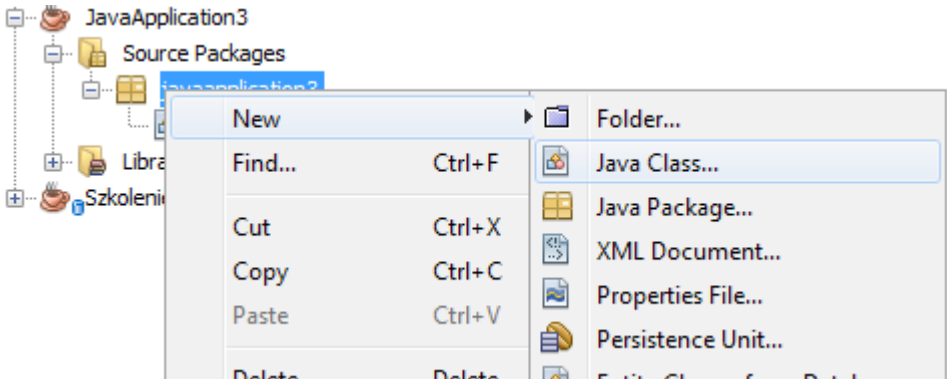
# Klasa i obiekt danej klasy

## Ogólne rozeznanie

Załóżmy, że chciałbym uruchomić produkcję samochodów. Od czego bym zaczął? Od zrobienia dokładnego projektu, opisującego wszystkie właściwości mojego samochodu, powiązania między częściami, a także funkcjonalności auta. OK. Projekt stworzony, uruchamiam produkcję – powstają kolejne samochody wykonane wg mojego projektu. Posiadają pewne właściwości wspólne dla wszystkich samochodów z mojej produkcji (moc silnika), ale też elementy charakterystyczne tylko dla konkretnego egzemplarza (kolor, numer silnika). W świecie programowania taki super-dokładny projekt nazywamy klasą, a na jego podstawie (czyli klasy) tworzone są kolejne obiekty (mówimy też *instancje klasy*), czyli egzemplarze.

## Piszemy pierwszą klasę

Aby utworzyć nową klasę klikam prawym przyciskiem myszy na pakiecie wybieram `New » Java Class`.




**Rysunek 40 - Tworzenie nowej klasy**

Na kolejnym ekranie podajemy nazwę naszej nowej klasy. Chcę stworzyć klasę, która będzie reprezentowała punkt na płaszczyźnie kartezjańskiej. Konwencja nazewnicza w Javie każe nazywać wszystkie klasy z dużej litery, więc moja klasa będzie nazywać się `Punkt`. Wychodzę na coś takiego:

```
10      * @author Kamil
11      */
12     public class Punkt {
13
14     }
15
```

**Rysunek 41 - Pusta klasa Punkt**

Najpierw zdefiniuję właściwości mojego punktu. Będą to współrzędne x, y – obie typu `double`.

Żeby obiekt klasy Punkt miał właściwości x, y muszę dodać tzw. pola. Robimy to w następujący sposób:

```
12  public class Punkt {
13
14      public double x;
15      public double y;
16
17  }
```

Rysunek 42 - Pola klasy

Zwróć uwagę na różnice między polami a zwykłymi zmiennymi. Po pierwsze, pola deklarujemy w klasach i tylko w nich, nigdy w metodach. Po drugie, pola mają przed sobą `public` czyli specyfikator dostępu. Po trzecie, są widoczne z każdego miejsca w klasie.

Aby stworzyć obiekt `Punkt` w jakiejś klasie z metodą `main` muszę napisać poniższą linijkę.

```
14      public static void main(String[] args) {
15          Punkt p = new Punkt();
16      }
17  }
```

Rysunek 43 - Tworzymy obiekt

Zauważ, że tworząc klasę, definiuję nowy typ zmiennych. Jest to dość intuicyjne `Punkt p` jest nowym punktem. Słowo kluczowe `new` tworzy nowy obiekt. W następnym rozdziale wyjaśnimy znaczenie tych tajemniczych nawiasów. Żeby dobrać się do naszych pól, używam operatora `'.'`.

```
12  public class JavaApplication3 {
13
14      public static void main(String[] args) {
15          Punkt p = new Punkt();
16          p.x = 5;
17          p.y = 3.13;
18          System.out.println( "["+ p.x +", "+ p.y +"]" );
19      }
20  }
```

Rysunek 44 - dostęp do pól obiektu

Oczywiście jeśli chciałbym dodać kolejną właściwość mojego Punktu, wystarczy, że dopiszę kolejne pole.

### Metody obiektów

Napiszę teraz program obliczający odległość między dwoma punktami. Potrzebne mi będzie jakaś metoda, umiejąca liczyć odległość. Mógłbym zawrzeć wszystkie obliczenia w metodzie `main`, ewentualnie stworzyć oddzielną metodę (taką jak do tej pory robiliśmy) przyjmującą dwa argumenty typu `Punkt`.

Dzięki programowaniu obiektowym możemy zrobić to znacznie bardziej elegancko.

W klasie `Punkt` deklaruję metodę `obliczOdległosc( Punkt p )`. (metodę niestatyczną!)

```
12    public class Punkt {
13
14        public double x;
15        public double y;
16
17        public double obliczOdleglosc( Punkt p ) {
18            return 0;
19        }
20
21    }
```

Rysunek 45 - metody niestatyczne

Od teraz na rzecz obiektu typu Punkt mogę wywołać metodę `obliczOdleglosc( Punkt p )`:

```
12    public class JavaApplication3 {
13
14        public static void main(String[] args) {
15            Punkt p = new Punkt();
16            p.x = 5;
17            p.y = 3.13;
18
19            p.obliczOdleglosc( null ); // <----
20
21            System.out.println( "["+ p.x +", "+ p.y +"]" );
22        }
23    }
```

Rysunek 46 - wywołanie metody na rzecz obiektu

Pewnie dziwi Cię, dlaczego metoda `obliczOdleglosc` przyjmuje tylko jeden argument – przecież potrzebne są dwa!. Nie sposób się nie zgodzić, ale mam dwa argumenty ☺ – jeden jawnie zadeklarowany, a drugim „argumentem" będzie obiekt na rzecz którego wywołano metodę. Odwołuję się do niego za pomocą słowa kluczowego `this`. (a tak naprawdę nawet nie muszę pisać `this.x`, robię to tylko po to aby odwołanie było całkowicie jawne) Wróćmy do klasy `Punkt` i zaimplementujmy metodę `obliczOdleglosc`.

```
    public class Punkt {
13
14      public double x;
15      public double y;
16
17      public double obliczOdleglosc( Punkt p ) {
18          double odlegloscX = this.x - p.x;
19          double odlegloscY = this.y - p.y;
20
21          return Math.sqrt( odlegloscX*odlegloscX + odlegloscY*odlegloscY );
22      }
23
24  }
```

Następnie przechodzę do klasy z metodą `main`, stworzę tam nowy `Punkt` i wywołam metodę `obliczOdleglosc` już z prawidłowym argumentem.

```
12  public class JavaApplication3 {
13
14      public static void main(String[] args) {
15          Punkt p = new Punkt();
16          p.x = 5;
17          p.y = 3.13;
18
19          Punkt p2 = new Punkt();
20          p2.x = 12;
21          p2.y = -2;
22
23          double odleglosc = p.obliczOdleglosc( p2 ); // <----
24          System.out.println("Odleglosc miedzy p i p2 wynosi: " + odleglosc);
25      }
26  }
```

javaapplication3.JavaApplication3

Output - JavaApplication3 (run)

```
run:
Odleglosc miedzy p i p2 wynosi: 8.678530981681174
BUILD SUCCESSFUL (total time: 0 seconds)
```

Spójrz jak pięknie upraszcza się kod naszej metody `main`. Co więcej, jest o wiele czytelniejszy cała magia dzieje się w linijce `23` – po prostu mówię punktowi `p`, żeby obliczył jak daleko jest od punktu `p2`. Dzięki klasom każdy `Punkt` będzie wiedział jak ma obliczać odległość od siebie do jakiegoś innego `Punkt`u.

### Porównanie obiektowe

Tak jak już wspomniałem, za pomocą zwykłego operatora == nie można porównywać tekstów, służy do tego metoda `equals()`. Już pokazuję dlaczego tak nie można:

```
3   public class JavaApplication3 {
4
5       public static void main(String[] args) {
6           String t = "abc";
7           String t2 = new String("abc");
8           System.out.println( t == t2 );
9       }
10  }
```

javaapplication3.JavaApplication3

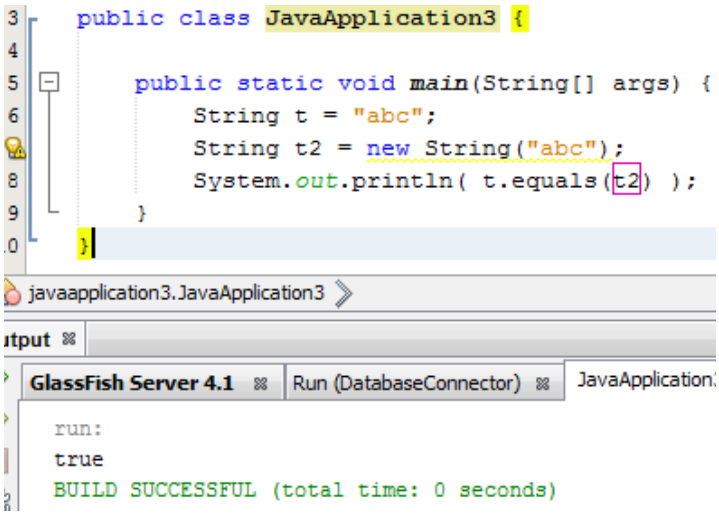
Output

GlassFish Server 4.1 | Run (DatabaseConnector) | JavaApplication3

```
run:
false
BUILD SUCCESSFUL (total time: 0 seconds)
```

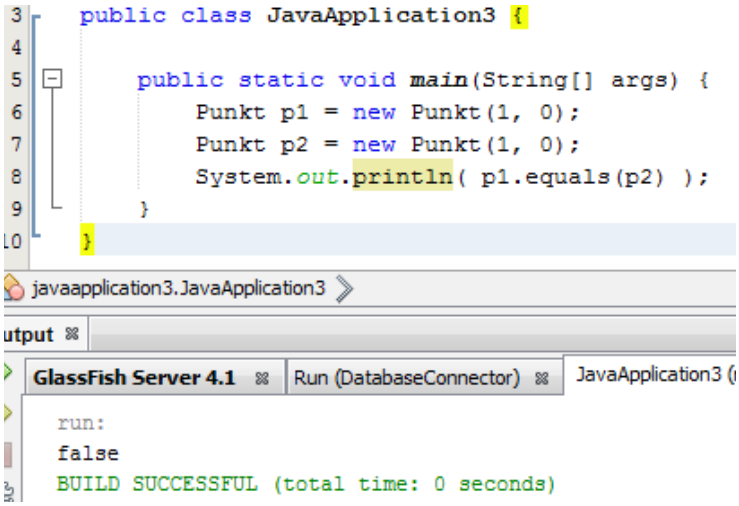
Czy dzieje się tu coś dziwnego? abc ≠ abc? Nie o to chodzi. Operator porównania służy do porównywania wartości zmiennych. A co jest wartością zmiennej obiektowej? Referencja! Mogę mieć dwa różne obiekty (referencje), ale w obu obiektach może znajdować się ta sama treść. Krótko mówiąc operatorem == zadaję pytanie czy `t` i `t2` to **ten sam obiekt**, a nie taki sam.

Żeby wykonać takie porównanie poprawnie należy użyć metody `equals()` tak jak na poniższym przykładzie:




**Rysunek 47 - użycie metody equals()**

A co z naszym `Punkt`em?




Czyżby porównanie obiektowe nie działało? Działa, ale Java nie wie jak ma stwierdzić tożsamość dwóch obiektów typu `Punkt`. Porównanie obiektowe polega na przyrównaniu do siebie wszystkich pól dwóch obiektów. Niestety JVM nie jest w stanie dynamicznie odwołać się do tych pól, dlatego żeby móc używać metody `equals()` na obiektach typu `Punkt` musielibyśmy przesłonić metodę `equals()` z klasy `Object`. Do tego jednak potrzebna nam będzie znajomość takiego zjawiska jak polimorfizm, które omówimy później.

# Konstruktory – po co były te nawiasy?

### Jednoczesne tworzenie obiektu i inicjowanie pól

Dotychczas, chcąc stworzyć obiekt Punkt i wpisać w niego informacje, musiałem pisać aż trzy linijki kodu:

```
15          Punkt p = new Punkt();
16          p.x = 5;
17          p.y = 3.13;
```

Czy nie da się tego załatwić prościej i wygodniej, tak abym mógł od razu podać wartości, które zostaną wpisane w obiekt? Mogę ☺ Po to jest właśnie konstruktor, czyli twór metodopodobny, który umie przyjmować argumenty i jest wywoływany **PO** stworzeniu obiektu w celu zainicjowania pól.

Konstruktory piszemy w klasach, których obiektów będą dotyczyć – czyli konstruktor zadeklaruję w klasie `Punkt`. Mają one następującą składnię:

```
1.  public class [nazwa klasy] {
2.
3.      public [nazwa klasy] ([zestaw argumentów]) { // <--- KONSTRUKTOR
4.          .
5.          .  OPERACJE
6.          .
7.      }
8.  }
```

Czyli konstruktor `Punkt`u będzie wyglądał tak:

```
12  public class Punkt {
13
14      public double x;
15      public double y;
16
17      public Punkt(double x, double y) { // <--- KONSTRUKTOR
18          this.x = x;
19          this.y = y;
20      }
21
22      public double obliczOdleglosc( Punkt p ) {
23          double odlegloscX = this.x - p.x;
24          double odlegloscY = this.y - p.y;
25
26          return Math.sqrt( odlegloscX*odlegloscX + odlegloscY*odlegloscY );
27      }
28
29  }
```

Zapewne zauważyłeś/aś już, że w momencie zapisania klasy `Punkt` w obecnej formie pojawiły się błędy w klasie z metodą `main`:

```
public static void main(S
    Punkt p = new Punkt()
    p.x = 5;
    p.y = 3.13;

    Punkt p2 = new Punkt();
    p2.x = 12;
    p2.y = -2;
```

```
constructor Punkt in class Punkt cannot be applied to given types;
  required: double,double
  found: no arguments
  reason: actual and formal argument lists differ in length
----
(Alt-Enter shows hints)
```

Powyższy błąd sygnalizuje, że do konstruktora podaliśmy nieprawidłowy zestaw argumentów, bo nie dostała żadnego, spodziewał się otrzymać dwa argumenty typu `double`. Poprawiam więc, mój kod do poniższej postaci:

```
    public static void main(String[] args) {
        Punkt p = new Punkt( 5, 3.13 );

        Punkt p2 = new Punkt( 12, -2 );
```

Teraz, nie muszę ręcznie wpisywać wartości w pola – robi to za mnie konstruktor. Te tajemnicze nawiasy po `new Punkt` są po prostu wywołaniem odpowiedniego konstruktora.

Chwila chwila… Przecież wcześniej nie napisaliśmy żadnego konstruktora i mogliśmy napisać `new Punkt()`. Teraz, gdy mamy już nowy konstruktor tamten (tzw. konstruktor pusty) gdzieś zniknął.

Dokładnie tak. Zniknął. W Javie każda klasa musi mieć konstruktor, dlatego kompilator jeśli nie wykryje żadnego konstruktora w klasie, dokleja własny – pusty, czyli nie robiący nic i nie przyjmujący

żadnych argumentów. **ALE!** Tylko w przypadku, gdy w klasie nie będzie żadnego konstruktora. Jeśli napiszemy własny oznacza to wzięcie na siebie odpowiedzialności za określenie inicjowania obiektu.

### Przeciążanie konstruktorów

Na pewno zauważyłeś olbrzymie podobieństwo składniowe między konstruktorem, a metodą. Przy konstruktorze po prostu nie piszemy zwracanego typu. OK. Pisaliśmy już metody prawidłowo reagujące na dane różnego typu (przeciążenia). Podobnie możemy przeciążać konstruktory, nie użyjemy jednak słowa kluczowego `return` (bo konstruktor nie zwraca nic, nie jest `void`'em – i żeby było jasne – konstruktor nie tworzy obiektu!). Wykorzystamy słowo kluczowe `this`, ale w inny, mniej konwencjonalny sposób.

```
19      public Punkt(double x, double y) { // <--- KONSTRUKTOR
20          this.x = x;
21          this.y = y;
22      }
23
24      public Punkt(String x, String y) { // <--- KONSTRUKTOR
25          this ( Double.parseDouble( x ), Double.parseDouble( y ) );
26      }
```

Rysunek 48 - Przeciążanie konstruktora

`this()` oznacza wywołanie innego konstruktora tej samej klasy. Jest to konstrukcja charakterystyczna dla konstruktorów i dostępna tylko w nich. Dodatkowo musi to być pierwszą jego instrukcją.

# Cykl życia obiektu

### Gdzie obiekt się zaczyna, a gdzie kończy?

Odpowiedzmy najpierw na to pierwsze pytanie – powołujemy obiekt do życia pisząc `new [nazwa klasy]()`. W tym momencie następuje alokacja pamięci, którą zajmie obiekt. Tworzyliśmy już obiekty, ale nie usuwaliśmy ich z pamięci. Czyli albo spowodowaliśmy tzw. wyciek pamięci (programiści C, C++ doskonale wiedzą o czym mowa ☺), albo robi się to jeszcze inaczej. Obiekt żyje, krótko mówiąc, od momentu kiedy zostanie stworzony do momentu gdy przestaje być potrzebny. Decyzji o tym, czy obiekt jest jeszcze potrzebny nie podejmuje programista, a *Javowa Odśmieciarka* ☺, programista może co najwyżej pomóc jej podjąć decyzję.

W Javie programista panuje wyłącznie nad tworzeniem obiektów, nie przejmuje się ich kasowaniem (w rozbudowanych aplikacjach naprawdę ciężko jest określić dobry moment na skasowanie obiektu), robi to za niego `Garbage Collector`. Jest to twór, który skanuje pulę wszystkich obiektów i szuka referencji do niego w uruchomionych programach Javy oraz w innych obiektach. jeśli znajdzie choć jedną – oznacza to, że obiekt jest wykorzystywany (*`alive-object`*), jeśli nie – obiekt został porzucony (*`dead-object`*), czyli istnieje, ale program stracił referencję do niego – idzie więc do niszczarki.

### Garbage Collector, wystawianie obiektów do skasowania

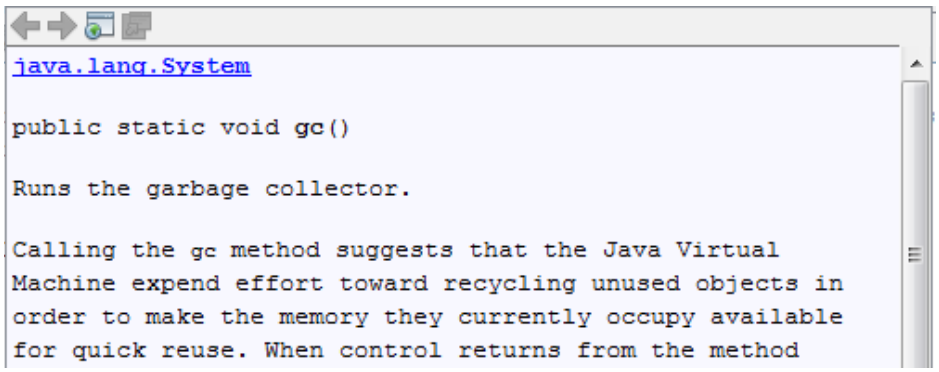
Zajmiemy się najpierw odznaczaniem obiektów do skasowania. Poprzez wprowadzenie pojęcia referencji rozumiesz, że mogę mieć w programie jedną zmienną, a stworzyć 10 obiektów i odwrotnie: mogę mieć 10 zmiennych, a tylko jeden obiekt (każda z tych 10 zmiennych to ten sam obiekt). Aby wystawić obiekt do `Garbage Collection` wystarczy że napiszę:

```
14      public static void main(String[] args) {
15          Punkt p = new Punkt( 5, 3.13 );
16
17          p = null;
18
19      }
```

Chodzi oczywiście o linijkę `17`. Wystarczy w zmienną przechowującą referencję wpisać wartość `null`, która oznacza pustą referencję (czyli przerwanie istniejącej). `null` oczywiście można wpisać w każdy obiekt. Jeśli była to jedyna zmienna, która posiadała referencję do `Punktu` – przy najbliższym uruchomieniu `Garbage Collector` punkt zostanie usunięty z pamięci. Kiedy nastąpi takie wywołanie? Tego wiemy nigdy. Możemy jednak zasugerować maszynie wirtualnej, aby przeprowadziła `Garbage Collection`.

Robimy to wywołaniem:

```
16          System.gc();
```



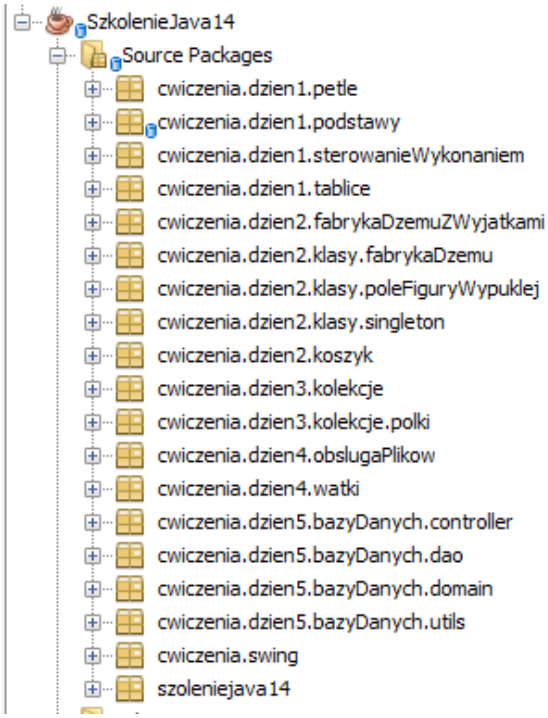

**Rysunek 49 - Dokumentacja metody gc() z klasy System**

Nie było złym pomysłem zabranie programiście możliwości uruchomienia `Garbage Collection` na żądanie, ponieważ `Garbage Collector` działa w obrębie wszystkich aplikacji, a programista porusza się zawsze w obrębie jednej. Ciągłe wywoływanie `Garbage Collectora` (który jest dość czasochłonnym procesem) paradoksalnie doprowadziłoby wyłącznie do spadku wydajności.

# Pakiety - jednostka organizacji klas

## Idea pakietu

Pakiety w Javie są jednostką agregującą wiele klas w grupy odpowiadające za pewną część aplikacji lub biblioteki (Przynajmniej tak powinno być). Zakładka `Source Packages` zawiera w sobie pakiety naszej aplikacji.




**Rysunek 50 - pakiety repozytorium szkoleniowego**

W każdym z tych pakietów znajduje się wiele klas, dzięki temu można łatwiej zachować porządek w projekcie.

## Instrukcja import

Przy pierwszych ćwiczeniach (tych w których korzystaliśmy z klasy `Scanner`) na pewno podświetlało Ci na czerwono deklarację zmiennej typu `Scanner`, musiałeś/aś więc wykonać import klasy. Dlaczego musieliśmy wykonać taką operację? Otóż `Scanner` jest zwykłą klasą (nie różni się niczym od naszej klasy `Punkt`), a klasy widzą się wzajemnie tylko w obrębie jednego pakietu. Moja klasa `Punkt` była w pakiecie `javaapplication3`, natomiast klasa `Scanner` w pakiecie `java.util`, dlatego musieliśmy dodać instrukcję `import`, przed deklaracją klasy publicznej:

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

Obejrzyjmy jej strukturę – jest to po prostu nazwa klasy poprzedzona pakietami. Po wpisaniu tej instrukcji został dodany „synonim” – typ `Scanner` oznacza `Scanner` z pakietu `java.util`. Tak

naprawdę nie musisz dodawać importów – wystarczy, że przed każdym typem napiszesz pełną ścieżkę pakietową:

```
1  package javaapplication3;
2
3  public class JavaApplication3 {
4
5      public static void main(String[] args) {
6          java.util.Scanner s = new java.util.Scanner(System.in);
7          s.close();
8      }
9  }
```

Rysunek 51 - używanie pełnych ścieżek pakietowych

### Konwencja nazewnicza pakietów

Programiści Javy nie mogą dopuścić żeby powstała sytuacja w której dwóch programistów identycznie nazwało swoje klasy i umieścili je w tak samo nazywających się pakietach (bo jak je wtedy odróżnić?). Przypuszczając, że trzeci programista musi wykorzystać obie z nich, w tym momencie ma związane ręce, ponieważ może zaimportować tylko jedna z nich. Dlatego powstała konwencja nazewnicza pakietów, której w aplikacjach komercyjnych aplikacjach lepiej przestrzegać.

Nazwa pakietu:

**[odwrócona domena w której pracujemy] . [nazwa aplikacji] . [część tej aplikacji]**

Np.:

**pl.jsystems.zlotaKaczkaNaLacuchu.controller**

Przestrzeganie tej reguły gwarantuje unikalność klas w pakietach.

# Specyfikatory dostępu – public i spółka

### Po co ograniczać dostęp do pól i metod?

Nie zawsze wystawianie wszystkich struktur naszych obiektów jest dobrym pomysłem. Zacznijmy od tego, że docelowo piszemy tylko klasy, które później są wykorzystywane w innych miejscach i oprogramowywane przez innych programistów, którzy mogą nie znać wszystkich powiązań i zależności. Np. blokując dostęp do pól możemy udostępnić metodę, która zrobi dokładnie to samo, ale w ramach metody mogę wykonać dodatkowy kod – chociażby zaokrąglenie kwot pieniężnych do drugiego miejsca po przecinku. Kolejną korzyścią jest to, że w moim obiekcie wykształci się swego rodzaju obudowa, nie pozwalająca dowolnie modyfikować mojego obiektu, a jeśli już, to tylko w określony przeze mnie-twórcę sposób.

### Jak działa specyfikator dostępu?

Ogranicza widoczność pól i metod. Pamiętasz pierwszego `main`'a którego pisaliśmy? Powiedziałem wtedy, że `public` oznacza, że metoda jest widoczna wszędzie. Zmienię teraz specyfikator na `private`. (W następnym rozdziale omówię zakresy każdego z nich) Klasa `Punkt`:

Tera

```
12  public class Punkt {
13
    priva  public static void main(String[] args) {
    priva
16              Punkt p = new Punkt(2234.34, 34324.4);
17  public
18      t
19      t
20  }           System.out.println( p.x );
21  public
22      d
23      d  }
24      return Math.sqrt( odlegloscX*odlegloscX + odlegloscY*odlegloscY );
25  }
26
27  }
```

x has private access in Punkt
---
(Alt-Enter shows hints)

Rysunek 53 - działanie specyfikatora dostępu

Analogicznie wyglądałoby określenie dostępu do jakiejś metody.

Rysunek 52 - specyfikator private

### Zakres widoczności każdego ze specyfikatorów

| | Klasa macierzysta | Pakiet | Podklasy poza pakietem | Reszta aplikacji |
|---|---|---|---|---|
| **public** | + | + | + | + |
| **protected** | + | + | + | - |
| **Bez specyfikatora** | + | + | - | - |
| **private** | + | - | - | - |

Klasa macierzysta to klasa w której zadeklarowane zostało pole lub metoda. Dlatego nie mogłem dostać się do pola `x` w obiekcie typu `Punkt`. Nastąpiło odwołanie spoza klasy macierzystej.

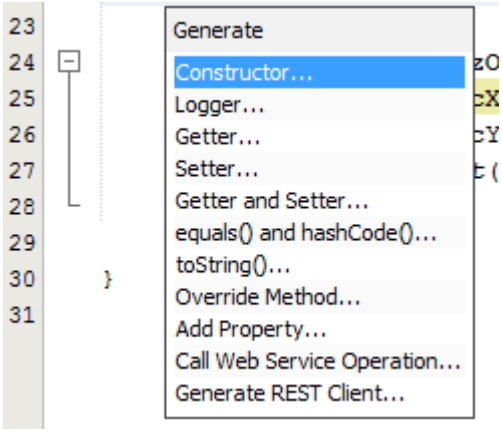
Wedle konwencji pola klas zawsze będą prywatne i dostawać się za pomocą metod `get[pole]()` i `set[pole]()`. Zatem klasa Punkt zostanie uzupełniona o metody:

```
17      public Punkt(double x, double y) {
18          this.x = x;
19          this.y = y;
20      }
21
22      public double getX() {
23          return x;
24      }
25
26      public void setX(double x) {
27          this.x = x;
28      }
29
30      public double getY() {
31          return y;
32      }
33
34      public void setY(double y) {
35          this.y = y;
36      }
```

Rysunek 54 - klasa encyjna wg konwencji

Wynikają z takiej architektury same korzyści. Po pierwsze, możemy zarządzać dostępem do pól – np. tworzyć pola tylko do odczytu, albo określać sposób wpisywania danych do obiektu. Po drugie, otrzymaliśmy miejsce na dodawanie reakcji na dane innego typu – np. typu `String`. Po trzecie, gdy na obiekcie będziemy mogli wywoływać 300 metod (a takie sytuacje nader często zdarzają się w programowaniu UI), możemy filtrować metody za pomocą prefiksów (wszystkie metody odczytujące zaczynają się od `get`, a wszystkie wpisujące od `set`).

Zestaw takich metod możemy wygenerować za pomocą środowiska NetBeans. Wystarczy, że będąc w klasie naciśniesz `alt + insert`:




Rysunek 55 - Autogeneracja metod za pomocą NetBeans

Wystarczy, że wybierzesz z powyższej listy pozycję `Getter and Setter` i zaznaczysz pola, dla których chcesz wygenerować akcesory (czyli metody `get` i `set`).

Najczęściej klasy pisze się w następującej konwencji:

- Prywatne pola
- Mocno sparametryzowane metody oznaczamy jako prywatne

- Resztę metod oznaczamy jako publiczne

Np.:

```
12  public class AnalizaFinansowa {
13
14      private double netto;
15      private double vat;
16
17      public double getNetto() {
18          return netto;
19      }
20
21      public void setNetto(double netto) {
22          this.netto = netto;
23      }
24
25      public double getVat() {
26          return vat;
27      }
28
29      public void setVat(double vat) {
30          this.vat = vat;
31      }
32
33      public AnalizaFinansowa getAnalizaZOstatniegoMiesiaca() {
34          return getAnalizaOkres( "miesiac" );
35      }
36      private AnalizaFinansowa getAnalizaOkres(String okres) {
37          this.netto = 234*okres.length();
38          this.vat = 32.2/okres.length();
39          return this;
40      }
```

# Tworzenie dokumentacji

## Komentarze

Komentarze są tekstem oznaczonym przez programistę, który ma zostać całkowicie zignorowany przez kompilator. Często wykorzystuje je się także do wyłączania krótkich fragmentów kodu. W Javie mamy komentarz jednoliniowy, który wykomentuje wszystko od miejsca w którym został wstawiony do końca obecnej linii, oraz komentarz wieloliniowy, który musimy zamykać po otwarciu:

Komentarz jednoliniowy to po prostu dwa slash'e:

```
5   public static void main(String[] args) {
6
7       // TO ZOSTANIE ZIGNOROWANE PRZEZ KOMPILATOR
8
9
10  }
```

**Rysunek 56 - Komentarz jednoliniowy**

Komentarz wieloliniowy zaczynamy od `/*`, a kończymy `*/`.

```
5   public static void main(String[] args) {
6
7       /*  TO ZOSTANIE ZIGNOROWANE PRZEZ KOMPILATOR
8           I TO
9           I TO TEŻ
0       */
1   }
```

**Rysunek 57 - Komentarz wieloliniowy**

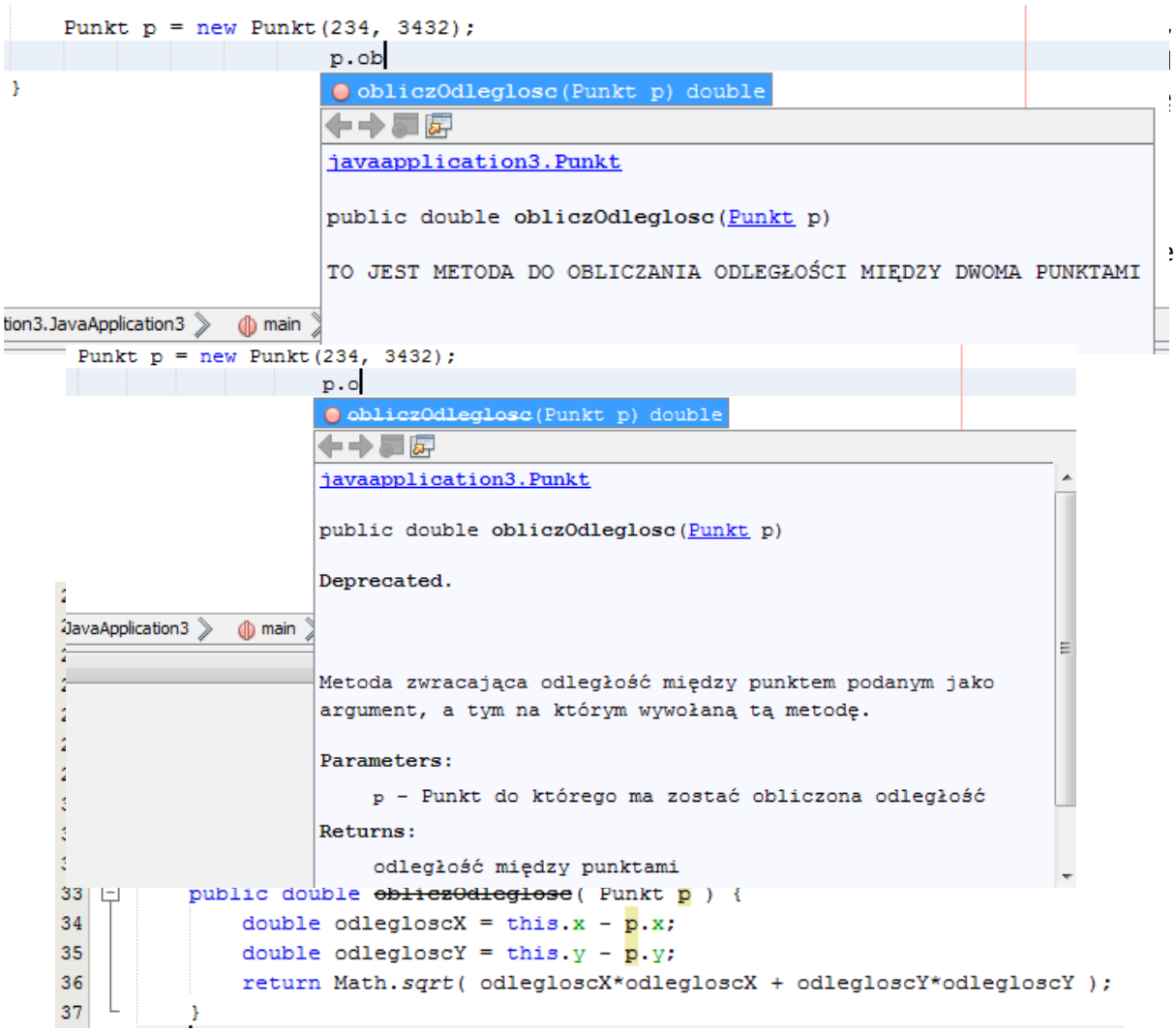
## Gdzie znajduje się dokumentacja kodu i jak ją tworzyć?

Dokumentacja znajduje się ponad metodami w postaci komentarzy wielolinowych o dość charakterystycznym wyglądzie, bo zaczynają się od `/**`, a każda ich linia zaczyna się od `*`:

```
23  /**
24   *
25   * TO JEST METODA DO OBLICZANIA
26   * ODLEGŁOŚCI MIĘDZY DWOMA PUNKTAMI
27   *
28   */
    public double obliczOdleglosc( Punkt p ) {
30      double odlegloscX = this.x - p.x;
31      double odlegloscY = this.y - p.y;
32      return Math.sqrt( odlegloscX*odlegloscX + odlegloscY*odlegloscY );
33  }
```

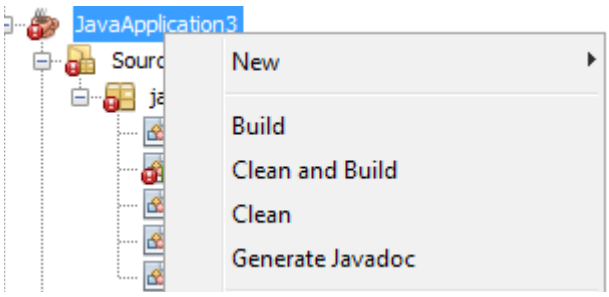
**Rysunek 58 - Dokumentacja metody obliczOdleglosc**

Żeby zobaczyć naszą dokumentację wystarczy, że rozwinięmy sobie listę dostępnych metod na obiekcie `Punkt` (`Ctrl + Spacja`) i wybierzemy z niej metodę `obliczOdleglosc`:



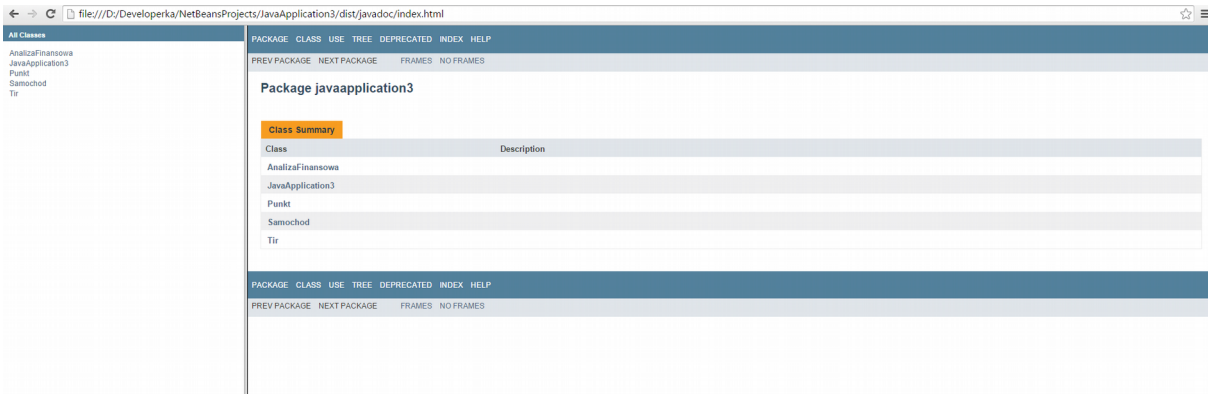

## Generowanie javadoc

Przeglądanie dokumentacji w tej formie nie jest zbyt wygodne, dlatego w środowisku NetBeans możemy wygenerować sobie stronę internetową (javadoc), która będzie ergonomicznie prezentować dokumentację każdej klasy w naszej aplikacji. Robimy to klikając prawym przyciskiem myszy na projekcie i wybieramy `Generate javadoc`.

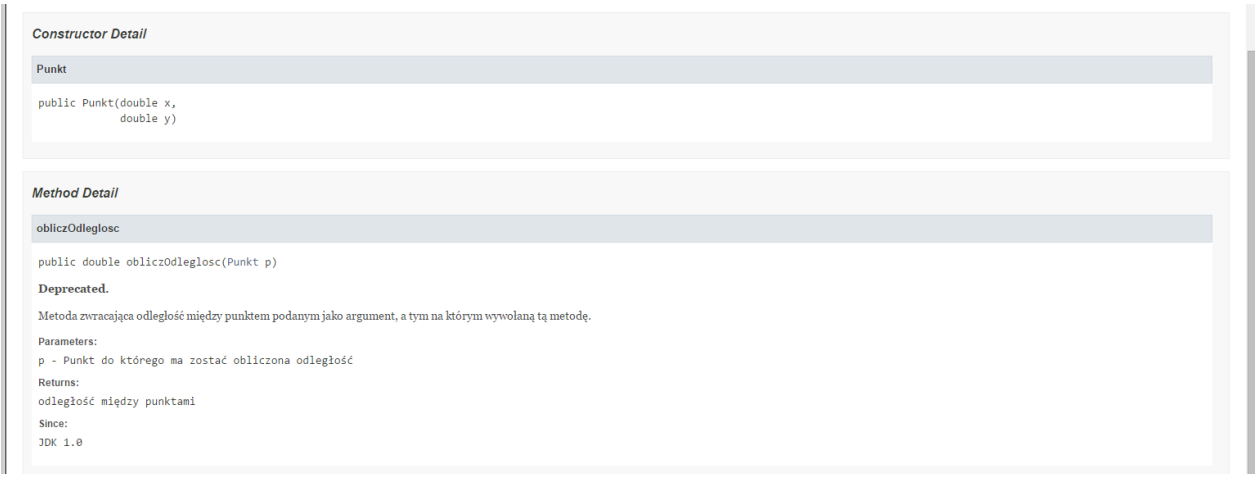



**Rysunek 59 - generowanie dokumentacji projektu**

Jeśli projekt się skompiluje zostanie wygenerowana i otwarta poniższa strona internetowa:




Obejrzyjmy dokumentację naszej metody `obliczOdleglosc`, wchodzę więc w klasę Punkt:




**Rysunek 60 - Korzystanie z javadoc**

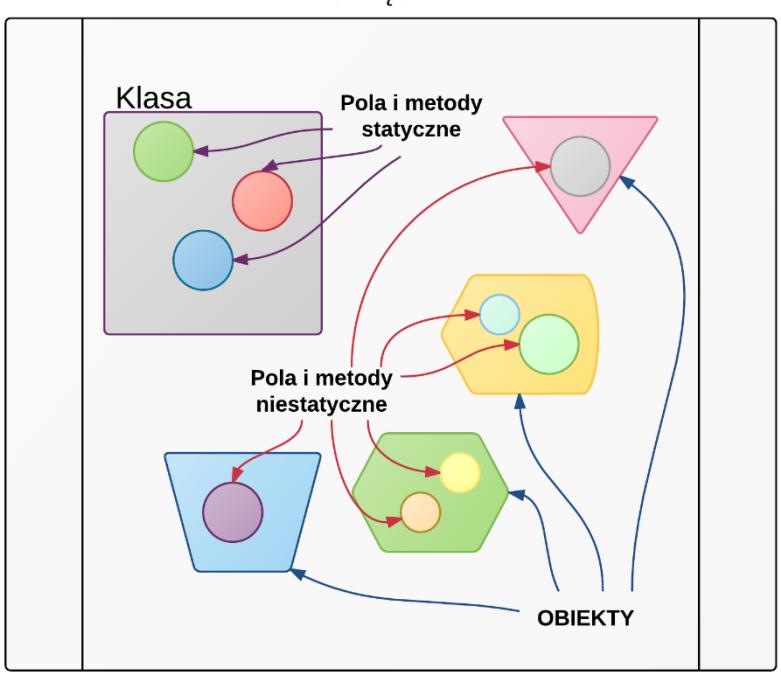
# Pola i metody statyczne

## Co robi słowo static?

Krótko mówiąc umieszcza metodę w klasie, a nie w obiekcie. Jeśli oznaczymy jakieś pole, albo metodę jako statyczną, przestają one być związane z jakimś obiektem – np. metoda `getX()` z klasy `Punkt` wypisywała współrzędną `x` z OBIEKTU `Punkt`. Po co używać elementów statycznych – żeby wydzielić jakieś informacje, albo funkcjonalności poza obiekty. Np. chcemy żeby nasze punkty były numerowane, będę więc potrzebował jakiegoś licznika. Intuicyjnie byłoby umieścić go w klasie `Punkt`, ale nie będzie on związany z żadnym obiektem. W takim przypadku klasa Punkt musiałaby wyglądać następująco:

```
12  public class Punkt {
13      private static int numeracja = 0;
        private double x;
        private double y;
        private int numerPunktu;
17
18      public Punkt(double x, double y) {
19          this.x = x;
20          this.y = y;
21          this.numerPunktu = ++numeracja;
22      }
23
24      public static int getNumeracja() {
25          return numeracja;
26      }
```

Pojawiły się dwa nowe pola – `numeracja`, `numerPunktu`. Pole numeracja jest moim licznikiem z którego konstruktor OBIEKTU `Punkt` pobiera kolejne wartości (ale numeracja jest związana z klasą, a nie z obiektem), natomiast pole `numerPunktu` to po prostu numer konkretnego OBIEKTU `Punkt`.




[

Rysunek 61 - Istota pól i metod statycznych

**WAŻNE!**

Jeśli przerwiemy związek metody z obiektem oznaczając ją słówkiem `static` nie możemy w jej ciele używać słówka kluczowego `this`. Co wynika z pierwszej części poprzedniego zdania. Z poziomu metod niestatycznych mamy jednak dostęp do wszystkich pól i metod zadeklarowanych jako statyczne.

OK. Wszystko bardzo ładnie, ale jak taką metodę wywołać? Trzeba najpierw wejść w element w którym ta metoda się znajduje, czyli do klasy.

```
3   public class JavaApplication3 {
4
5       public static void main(String[] args) {
6
7           Punkt p1 = new Punkt(12, 222.3);
8           Punkt p2 = new Punkt(12, 222.3);
9           Punkt p3 = new Punkt(12, 222.3);
10          Punkt p4 = new Punkt(12, 222.3);
11
12          System.out.println( Punkt.getNumeracja() );
13      }
14  }
```

javaapplication3.JavaApplication3

Output - JavaApplication3 (run)

```
run:
4
BUILD SUCCESSFUL (total time: 0 seconds)
```

### Wzorzec projektowy Singleton – zwieńczenie tego co statyczne

Bardzo często w aplikacjach musimy zapewnić wyłącznie jedną instancję klasy w obrębie całej aplikacji (np. klasy reprezentujące usługi, połączenie z bazą danych). Żeby to osiągnąć, musimy zaimplementować wzorzec projektowy nazwany `Singleton`. Nasza klasa ma zablokować dostęp do konstruktora i przy pierwszym wywołaniu pobrania obiektu stworzyć go, a każde kolejne wywołanie pobrania obiektu ma zwrócić już istniejący obiekt. Na początek zablokujemy dostęp do konstruktora spoza klasy rodzicielskiej – oznaczam go więc jako prywatny:

```
12  public class Singleton {
13
14      private Singleton() {}
15
16  }
```

W tej chwili nie da się już utworzyć obiektu klasy `Singleton`:

```
4
5       public static void main(Stri
6
7           Singleton s = new Singleton();
8
9       }
```

Singleton() has private access in Singleton
---
(Alt-Enter shows hints)

Skoro utraciłem już możliwość ręcznego tworzenia obiektu poza klasą `Singleton` (oznaczyłem kostruktor jako prywatny), takie wywołanie musi nastąpić z wewnątrz klasy, a jeśli nie mam obiektu to muszę odwołać się przez klasę jakąś metodą statyczną.

Następnie muszę gdzieś przechować informację o tym, czy utworzyłem już obiekt klasy `Singleton` czy nie. Potrzebuję więc pola statycznego przechowującego instancję mojej klasy. Będzie to wyglądać następująco:

```
12  public class Singleton {
13
14      private static Singleton instance;
15
16      private Singleton() {}
17      public static Singleton getInstance() {
18          if (instance == null) {
19              instance = new Singleton();
20          }
21          return instance;
22      }
23  }
```

Od teraz to klasa `Singleton` zarządza tworzeniem swoich obiektów. Jeszcze tylko screen dla dowodu:

```
5       public static void main(String[] args) {
6
7           Singleton s = Singleton.getInstance();
8           Singleton nibyDrugiObiekt = Singleton.getInstance();
9
10          System.out.println( s + " == " + nibyDrugiObiekt );
11
12      }
13  }
```

javaapplication3.JavaApplication3

utput - JavaApplication3 (run)

```
run:
javaapplication3.Singleton@15db9742 == javaapplication3.Singleton@15db9742
BUILD SUCCESSFUL (total time: 0 seconds)
```

### Stałe w Javie

W przeciwieństwie do wielu języków programowania stałe w Javie oznaczamy słówkiem kluczowym `final`. Ponadto, stała będzie najczęściej publicznym statycznym polem. Np.:

```
12  public class Punkt {
13
        public static final int STALA = 1;
```

**Rysunek 62 - deklaracja stałej w Javie**

Wg konwencji stałe we wszystkich językach programowania zapisujemy wielkimi literami.

Znacznie ciekawsze są stałe obiektowe np.:

```
4
5       public static final Punkt CONST_POINT = new Punkt(12, 12);
6
7       public static void main(String[] args) {
8
9           
10
11      }
```

Rysunek 63 - Stała obiektowa

Jak myślisz co się stanie gdy wywołam następującą linijkę

```
1.  CONST_POINT.setX( 14.4 );
```

Czy nastąpi błąd? Zobaczmy:

```
5       public static final Punkt CONST_POINT = new Punkt(12, 12);
6
7       public static void main(String[] args) {
8           CONST_POINT.setX( 14.4 );
9       }
10  }
```

Dlaczego nic się nie stało? Czyżby słówko `final` nie działało?

Działa, ale troszkę inaczej niż Ci się wydaje. Otóż `final` blokuje możliwość zmiany wartości jakiegoś pola lub zmiennej. Metoda `setX()` nie zmienia referencji do naszego `Punkt`u, tylko pole znajdujące się w nim. Dlatego wszystko się zgadza ☺. Dla dowodu:

```
4
5       public static final Punkt CON
6
7       public static void main(Strin
8           CONST_POINT = new Punkt(324, 234);
9       }
```

cannot assign a value to final variable CONST_POINT
---
(Alt-Enter shows hints)

# Dziedziczenie klas

### Czym jest dziedziczenie

Wyobraź sobie sytuację w której musisz stworzyć klasę, która posiada, powiedzmy, dwadzieścia pól. Co więcej każde z nich musi zostać oprogramowane. Następnie dostałeś/aś informację, że musisz stworzyć jeszcze dziesięć takich klas, które różnią się od tej pierwszej dwoma polami i mają o dwie metody więcej. Gdyby nie mechanizm dziedziczenia, musielibyśmy dosłownie przekopiować całość kodu z klasy bazowej i umieścić go w całości w innych klasach. Dzięki dziedziczeniu możemy zrobić dokładnie to o co nas proszą – napisać jedną klasę bazową i na jej podstawie kilka dziedziczących, które uzupełniają bazową o kilka elementów. To tylko taki prosty przykład, korzyści z używania dziedziczenia jest znacznie więcej.

### Składnia dziedziczenia, pierwsze użycie

Informację o tym, że jedna klasa dziedziczy po drugiej umieszczamy na etapie jej deklaracji w następującej formie:

```
1.  public [klasa dziedzicząca] extends [klasa bazowa] {
2.      .
3.      .   Dodatkowe pola i metody
4.      .
5.  }
```

Np.:

```
12  public class Tir extends Samochod {
13
14  }
```

Słowo kluczowe `extends` jest dość intuicyjne, ponieważ w klasach dziedziczących możemy wyłącznie dodawać pola i metody.

Dodam teraz kilka metod do klasy `Samochod` i do klasy `Tir`.

Klasa `Samochod`:

```
public class Samochod {
    private String marka;

    public String getMarka() {
        return marka;
    }

    public void setMarka(String marka) {
        this.marka = marka;
    }
}
```

Oraz klasa `Tir`:

```
public class Tir extends Samochod {
    private Double ladownosc;

    public Tir(Double ladownosc) {
        this.ladownosc = ladownosc;
    }

    public Double getLadownosc() {
        return ladownosc;
    }

    public void setLadownosc(Double ladownosc) {
        this.ladownosc = ladownosc;
    }
}
```

Przyjrzyjmy się teraz metodom obiektu klasy `Tir`:

```
public static void main(String[] args) {
    Tir t = new Tir(123.0);
    t.
}
}
```

```
equals(Object o)                boolean
getClass()                      Class<?>
getLadownosc()                  Double
getMarka()                      String
hashCode()                      int
notify()                        void
notifyAll()                     void
setLadownosc(Double ladownosc)  void
setMarka(String marka)          void
toString()                      String
wait()                          void
wait(long l)                    void
wait(long l, int i)             void
```

**Rysunek 64 - Metody obiektu dziedziczącego po klasie**

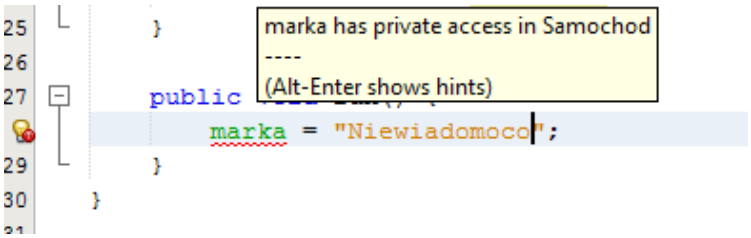
Zauważ, że w obiekcie klasy `Tir` mam zarówno metody klasy `Tir` jak i `Samochod`. To m.in. jest korzyść z używania dziedziczenia, możemy dzięki temu pisać bardziej generyczne elementy.

Z dziedziczeniem wiąże się jeszcze cała gama szczegółów, których powyższy przykład nie ujawnił, omówimy je w kolejnych rozdziałach.

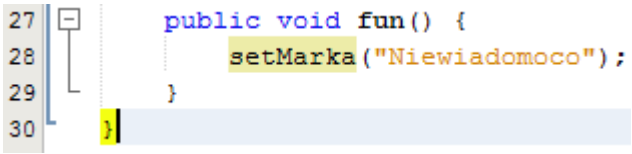
**WAŻNE!**
Struktura klas w Javie jest drzewem, a to oznacza że dziedziczyć możemy **wyłącznie** po jednej klasie!

### Działanie dziedziczenia, instrukcja super()

Zaczniemy od sprecyzowania, które elementy są dziedziczone. Publiczne, prywatne, chronione, a może wszystkie? Odpowiedź jest jedna – wszystkie, natomiast w dokumentacji Oracle'a napisane jest, że elementy prywatne nie są dziedziczone. Chodzi po prostu o to, że nie mamy do nich bezpośredniego dostępu. Zostały jednak stworzone:




Wiele osób twierdzi, że pola i metody prywatne nie są dziedziczone – chodzi po prostu interpretację dziedziczenia. Zostały stworzone (musiały być stworzone – zaraz wyjaśnimy dlaczego), ale po prostu nie mamy do nich bezpośredniego dostępu (elementy prywatne są widoczne tylko z poziomu klasy w której zostały zadeklarowane). Dlatego tutaj muszę skorzystać z setter'a.




Jeśli chcemy mieć bezpośredni dostęp do pól i metod w klasach dziedziczących, a jednocześnie nie chcemy wystawiać ich całkowicie na zewnątrz – korzystamy ze specyfikatora `protected` – po to został stworzony.

**WAŻNE!**
Każda klasa dziedziczy niejawnie po klasie `Object`. Jest to fundamentalna klasa Javy zaimplementowana natywnie (czyli w C++). Stąd właśnie biorą się metody `hashCode()`, `equals()`, `toString()`, `wait()`, `notify()` etc.

Odpowiemy teraz na pytanie jak działa dziedziczenie od spodu. Spójrz na poniższy przykład:

Klasa `Tir`:

```
12  public class Tir extends Samochod {
13
14      public Tir() {
15          System.out.println("## Tworze Tira");
16      }
17
18  }
```

Klasa `Samochod`:

```
    public class Samochod {
13
14      public Samochod() {
15          System.out.println("## Tworze samochod");
16      }
17
18  }
```

Klasa tworząca `Tira`:

```
3   public class JavaApplication3 {
4
5       public static void main(String[] args) {
6           Tir t = new Tir();
7       }
8   }
```

javaapplication3.JavaApplication3

Output

JavaApplication3 (run) | GlassFish Server 4.1 | Run (HRAwesomeMan

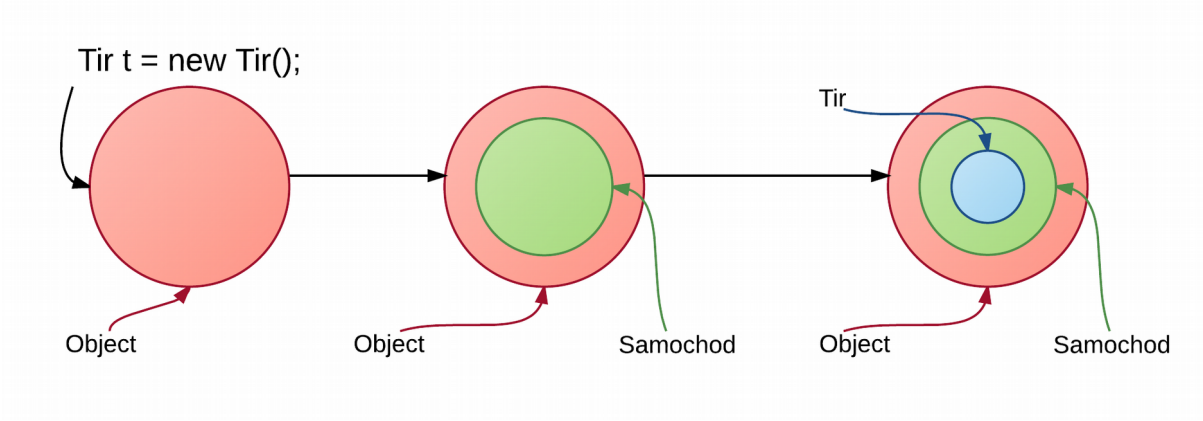
```
run:
## Tworze samochod
## Tworze Tira
BUILD SUCCESSFUL (total time: 0 seconds)
```

Spójrz na output. OK. Napisu `## Tworze Tira` się spodziewaliśmy, ale skąd wzięło się `## Tworze samochod`? Otóż, dziedziczenie działa w ten sposób, że żeby stworzyć obiekt `Tira`, Java musi dodać do niego całą strukturę klasy nadrzędnej, klasa `Samochod` dziedziczy po klasie `Object`, więc musi zostać stworzona również kolejna struktura. Te wywołania muszą nastąpić od typu najbardziej ogólnego do typu najbardziej szczególnego, czyli `Object >> Samochod >> Tir`. Z kolei po stworzeniu obiektu, wywoływany jest konstruktor, czyli po stworzeniu obiektu klasy `Object` został wywołany jego konstruktor, po stworzeniu obiektu klasy `Samochod` został wywołany jego konstruktor i w końcu po stworzeniu obiektu klasy `Tir` został wykonany jego konstruktor. Stąd wziął się powyższy output.

Może lepiej wyjaśni to poniższy rysunek:



**Rysunek 65 - Działanie JVM przy tworzeniu obiektu klasy Tir**




Na potrzeby przykładu zredukowałem klasy `Samochod` i `Tir` do samego konstruktora. Co się stanie, gdy konstruktor `Samochod`u będzie przyjmował jakieś parametry? (a przecież od tego jest).

Otóż dostanę błąd kompilacji:

Klasa `Samochod`:

```
13
15
16      public class Samochod {
17
18          private String marka;
19
20          public Samochod(String marka) {
21              this.marka = marka;
22              System.out.println("## Tworze samochod");
23          }
24
25      }
```

Klasa `Tir`:

```
8   /**
9    *
10   * @author Kam
11   */
12  public class T
13
15      public Tir() {
16          System.out.println("## Tworze Tira");
17      }
18
19  }
```

constructor Samochod in class Samochod cannot be applied to given types;
required: String
found: no arguments
reason: actual and formal argument lists differ in length
---
(Alt-Enter shows hints)

Co jest tu nie tak? Kompilator nie może wywołać konstruktora klasy nadrzędnej, ponieważ ten wymaga jednego argumentu typu `String`. Rozwiązaniem jest modyfikacja konstruktora klasy `Tir` do poniższej postaci:

```
12  public class Tir extends Samochod {
13
14      public Tir(String marka) {
15          super(marka);
16          System.out.println("## Tworze Tira");
17      }
18  }
```

Pojawiło nam się nowe słowo kluczowe `super()`, które oznacza wywołanie konstruktora klasy nadrzędnej, które jest **ZAWSZE** pierwszą instrukcją każdego konstruktora. Jeśli nie napiszemy jej sami, kompilator zrobi to za nas, ale wywoła zawsze konstruktor pezparametrowy. Jeśli takowego nie ma – otrzymamy powyższy błąd.

### Różnica między specyfikatorem protected, a brakiem specyfikatora

Różnica jest, prawie niezauważalna będąc szczerym, ale jest ☺. Ujawnia się dopiero gdy dziedziczymy po jakiejś klasie, przy czym klasa bazowa jest w innym pakiecie, a klasa dziedzicząca w innym.

Przyjrzyjmy się poniższemu przykładowi. Mam znaną Ci klasę `Samochod`, a teraz dodaję klasę `Autobus` dziedziczącą po `Samochodzie`, przy czym `Autobus` znajduje się w innym pakiecie niż `Samochod`.

Klasa `Samochod`:

```
   public class Samochod {
13
14      protected String marka;
15
16      public Samochod(String marka) {
17          this.marka = marka;
18          System.out.println("## Tworze samochod");
19      }
20
22      public String getMarka() {
23          return marka;
24      }
    }
```

Klasa `Autobus`:

```
    public class Autobus extends Samochod {
15      public Autobus(String marka) {
16          super(marka);
17      }
18
19      @Override
21      public String getMarka() {
22          return marka;
23      }
    }
```

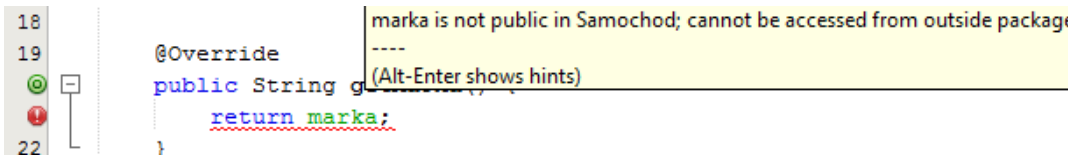
Zauważ, że pole `marka` jest dla `Autobusu` widoczne mimo, że **jest w innym pakiecie**. Po usunięciu

specyfikatora `protected` zostanie użyty domyślny, który zablokuje widoczność pola `marka` poza pakietem dla klas dziedziczących.

Klasa `Samochod`:

```
13
14      String marka;
15
16      public Samochod(String marka) {
```

Klasa `Autobus`:



# Polimorfizm – kwintesencja obiektowości

## Czym jest polimorfizm?

Pierwsze co przychodzi mi na myśl to polimorficzne stwierdzenie – `Tir` jest `Samochod`em. W polimorfizmie chodzi, krótko mówiąc, o traktowanie obiektów pewnych klas jako obiekty innych klas z zachowaniem pełnej funkcjonalności w obrębie wspólnego mianownika.

Wiem, że strasznie to niejasne, ale spójrz na poniższą linijkę:

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

    Samochod t = new Tir("TIRR: ");

}
```

**Rysunek 66 - Odwołanie polimorficzne**

Mogę potraktować obiekt `Tir` jako obiekt `Samochod` – `Tir` dziedziczy po `Samochod`, a więc nim jest! (przypomnij sobie rysunek jak powstaje nasz `Tir`).

## Przesłanianie metod – czyli co oferuje polimorfizm?

Przypuśćmy, że tworzymy aplikację odpowiadającą za system informatyczny w jakimś sklepie, będziemy więc z całą pewnością potrzebować klasy `Produkt` i `Koszyk`. `Koszyk` będzie przechowywał jakiś zbiór produktów (przyjmijmy na ten moment, że będzie nim tablica). Natomiast cena może być uzależniona od właściwości charakterystycznych dla bardziej szczegółowych klas reprezentujących Produkty – np. `Kosmetyk`. Gdyby nie polimorfizm i dziedziczenie nie mógłbym trzymać w jednej tablicy obiektów niejako różnych typów. Co więcej, zakładając, że `Produkt` ma w sobie metodę `getCena()`, nie mam właściwości pozwalających w miarę racjonalnie ową cenę obliczyć. Polimorfizm w dość prosty sposób pozwala rozwiązać ten problem.

Klasa `Produkt`:

```
public class Produkt {

    private String nazwa;

    public Produkt(String nazwa) {
        this.nazwa = nazwa;
    }

    public String getNazwa() {
        return nazwa;
    }

    public double getCena() {
        return 0;
    }
}
```

Klasa `Kosmetyk`:

```
public class Kosmetyk extends Produkt {

    private String marka;

    public Kosmetyk(String nazwa, String marka) {
        super(nazwa);
        this.marka = marka;
    }
}
```

Na razie nic specjalnie skomplikowanego ani ciekawego, prawdziwe czary zaczną się dziać gdy zmodyfikuję klasę `Kosmetyk` do poniżej postaci:

```
public class Kosmetyk extends Produkt {

    private String marka;

    public Kosmetyk(String nazwa, String marka) {
        super(nazwa);
        this.marka = marka;
    }

    @Override
    public double getCena() {
        double wspolczynnik = 1;
        if (marka.equals("Supermarka")) {
            wspolczynnik = 2;
        }
        return 211.32 * wspolczynnik;
    }
}
```

**Rysunek 67 - Przesłonięcie metody**

Co tu się stało? Podałem nową implementację metody `getCena()` z klasy `Produkt` w klasie `Kosmetyk`. W przypadku takiego zapisu, która z nich zostanie wykonana? Ta z klasy `Produkt` czy ta z klasy `Kosmetyk`?

```
14      public static void main(String[] args) {
15
16          Produkt p = new Kosmetyk("Krem", "Supermarka");
17
            System.out.println( "Cena: " + p.getCena() );
19
20      }
```

Odpowiedź jest jedna! Ta z klasy `Kosmetyk`:

```
run:
Cena: 422.64
BUILD SUCCESSFUL (total time: 0 seconds)
```

Teraz mogę już powiedzieć co oznacza to tajemnicze `@Override` nad metodą. Jest to tzw. adnotacja (wprowadzone zostały od JDK 1.5) i informuje ona programistę, że poniższa metoda przesłania tą z klasy bazowej. Adnotacja `@Override` nie ma żadnego skutku w czasie wykonania, ani kompilacji – jest po prostu informacją dla programisty (co nie znaczy, że wszystkie adnotacje działają w ten sposób).

### Przesłonięcie metody toString() z klasy Object

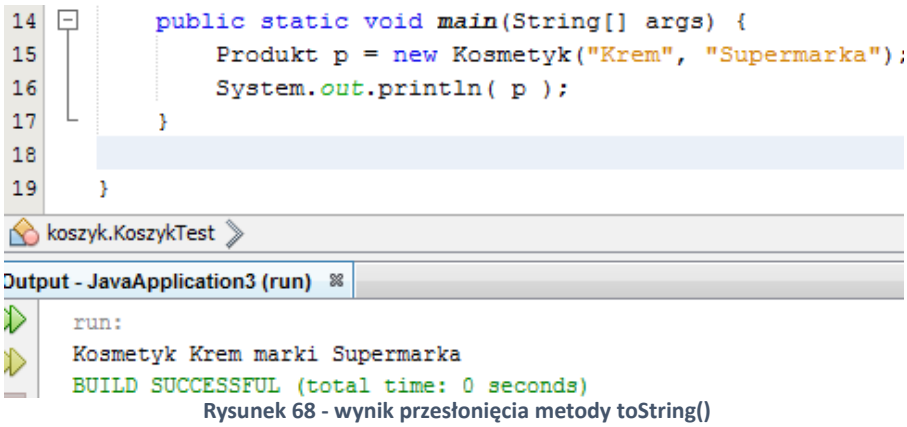
Jestem pewien, że próbowałeś/aś już wypisać na ekran jakiś własny obiekt i otrzymałeś/aś podobny output:

```
run:
koszyk.Kosmetyk@15db9742
BUILD SUCCESSFUL (total time: 0 seconds)
```

Nie jest to (boże broń!) referencja do naszego obiektu, a jedynie wynik działania metody `toString()` z klasy `Object`. Jak przed chwilą pokazaliśmy metody możemy bez przeszkód przesłaniać, więc do dzieła:

```
25              wspolczynnik = 2;
26          }
27          return 211.32 * wspolczynnik;
28      }
29
30      @Override
        public String toString() {
32          return "Kosmetyk "+ getNazwa() +" marki " + marka;
33      }
34  }
```

Analogicznie możemy przesłonić metodę `equals()` z klasy `Object`. Teraz po wypisaniu naszego obiektu na ekran zobaczymy już taki output:




Rysunek 68 - wynik przesłonięcia metody toString()

### Słowo kluczowe super, a polimorfizm

Wrócimy na chwilkę do przykładu z `Tir`em i `Samochod`em. Odkryjemy tam kolejną funkcjonalność związaną ze słowem kluczowym `super` w kontekście polimorfizmu. Przypuśćmy, że w klasie dziedziczącej chcę tak przesłonić metodę z klasy nadrzędnej, aby całe przesłonięcie polegało na poprzedzeniu wywołania tej metody jakimś innym kodem – za to odpowiada również słowo `super` – wywołuje metodę z klasy nadrzędnej.

Spójrz na poniższy przykład:

Klasa `Samochod`:

```
public class Samochod {

    private String marka;

    public Samochod(String marka) {
        this.marka = marka;
        System.out.println("## Tworze samochod");
    }
    public String getMarka() {
        return marka;
    }
}
```

Klasa `Tir`:

```
12  public class Tir extends Samochod {
13
14      public Tir(String marka) {
15          super(marka);
16          System.out.println("## Tworze Tira");
17      }
18
19      @Override
21      public String getMarka() {
22          return "## TIR " + super.getMarka();
23      }
}
```

Zauważ, że słowo `super` reprezentuje w tym kontekście stworzony obiekt niejako klasy nadrzędnej, dlatego w dalszym ciągu mam dostęp poprzedniej implementacji `getMarka()`:

Wywołanie i output:

```
3   public class JavaApplication3 {
4
5       public static void main(String[] args) {
6
7           Samochod t = new Tir("Ciezarowka: ");
8
9           System.out.println( t.getMarka() );
10
11      }
12  }
```

javaapplication3.JavaApplication3 > main

utput - JavaApplication3 (run)

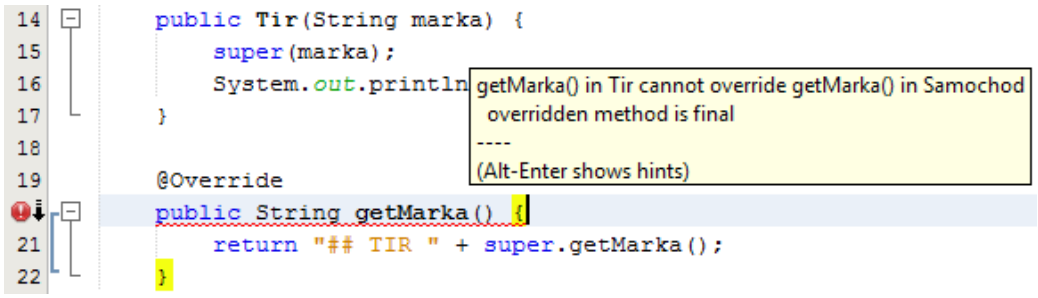
```
run:
## Tworze samochod
## Tworze Tira
## TIR Ciezarowka:
BUILD SUCCESSFUL (total time: 0 seconds)
```

### Blokowanie możliwości przesłaniania metod w klasach dziedziczących

Uzyskujemy to, podobnie jak w przypadku pól, słowem kluczowym `final`, które może zostać użyte zarówno przy deklaracji metody:

```
16      public Samochod(String marka) {
17          this.marka = marka;
18          System.out.println("## Tworze samochod");
19      }
        public final String getMarka() {
21          return marka;
22      }
23  }
```

Od teraz nie będzie dało się nadpisać implementacji tej metody w żadnej klasie dziedziczącej:


14 public Tir(String marka) {
15 super(marka);
16 System.out.println
17 }
18
19 @Override
21 return "## TIR " + super.getMarka();
22 }
getMarka() in Tir cannot override getMarka() in Samochod
overridden method is final
---
(Alt-Enter shows hints)

# Klasy abstrakcyjne i interfejsy

## Blokowanie tworzenia obiektu

Wrócimy do przykładu z Produktem i Klasami dziedziczącymi po nim. Metoda `getCena()` w klasie Produkt zwraca `0` mimo, że nie powinna nic zwracać, powiem więcej nie powinno się nawet dać utworzyć obiektu klasy `Produkt` tak bezpośrednio. W tym celu wprowadzono w Javie klasy abstrakcyjne – oznaczamy nimi klasy, których obiektów nie chcemy tworzyć bezpośrednio – wymuszamy na programistach używających takiej klasy dziedziczenie po niej i podanie implementacji pewnych metod (abstrakcyjnych) na poziomie klas dziedziczących.

## Słowo kluczowe abstract i konsekwencje jego użycia

Zaczniemy od oznaczenia klasy Produkt jako abstrakcyjną, robimy to dodając słowo kluczowe `abstract` na poziomie deklarowania klasy:

```
public abstract class Produkt {

    private String nazwa;

    public Produkt(String nazwa)
        this.nazwa = nazwa;
    }

    public String getNazwa() {
```

Rysunek 69 - Oznaczanie klas jako abstrakcyjnych

Oznaczenie klasy jako abstrakcyjnej uniemożliwia stworzenie obiektu tej klasy (ale tak całkowicie – to nie to samo co oznaczenie konstruktora słowem `private`).

```
6
7       public static void main(S
8
            Produkt p = new Produkt("PRODUKT");
10
11      }
```

Produkt is abstract; cannot be instantiated
---
(Alt-Enter shows hints)

Po oznaczeniu klasy jako abstrakcyjną możemy oznaczać również metody jako abstrakcyjne – oznacza to, że nie podaję wtedy ich implementacji – natomiast będzie trzeba ją podać na poziomie klas dziedziczących (o ile klasa dziedzicząca też nie jest abstrakcyjna).

```
public abstract class Produkt {

    private String nazwa;

    public Produkt(String nazwa) {
        this.nazwa = nazwa;
    }

    public String getNazwa() {
        return nazwa;
    }

    public abstract double getCena();
}
```

Rysunek 70 - Prosta klasa abstrakcyjna

Zauważ, że nie podaję implementacji metody `getCena()`. Po prostu oznaczam ją jako abstrakcyjną – czyli aby można było utworzyć obiekt `Produktu` – ta metoda musi zostać ukonkretniona.

Jeśli teraz skasuję implementację metody `getCena()` w klasie Kosmetyk otrzymam błąd kompilacji:

```
 *
 * @author Kamil
 */
public class Kosmetyk extends Produkt {

    private String marka;

    public Kosmetyk(String nazwa, String marka) {
        super(nazwa);
        this.marka = marka;
    }
```

Kosmetyk is not abstract and does not override abstract method getCena() in Produkt
---
(Alt-Enter shows hints)

To samo tyczy się bezpośredniego tworzenia obiektów klasy `Produkt`:

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

        Produkt p = new Produkt("Produkt") {

            @Override
            public double getCena() {
                return 100;
            }
        };

    }
}
```

Czyli na etapie tworzenia klasy `Produkt` muszę podać implementację metody `getCena()`.

## Interfejsy - czyli niby to samo, ale inaczej

Interfejsy to takie specjalne twory, dzięki którym możemy niejako dziedziczyć po kilku klasach naraz. W praktyce możemy potraktować interfejs jako klasę abstrakcyjną w której możemy deklarować tylko metody i zawsze wszystkie metody będą abstrakcyjne. Interfejs deklarujemy w następujący sposób:

```
12  public interface Wypisywalny {
13
14      public String wypisz();
15
16  }
```

Za pomocą interfejsów zwykle określamy metody, które musi zawierać jakaś klasa, aby mogła spełniać jakąś funkcjonalność. Np. chcąc zadeklarować interfejs `CRUD`, czyli zestaw metod do podstawowych operacji na encji bazodanowej (create, read, update, delete) zawierałby metody:

- `getAll()`
- `getOne()`
- `insert()`
- `update()`
- `delete()`

Mówimy, że klasy implementują pewne interfejsy i dokładnie tak samo dodajemy metody z interfejsu do klasy – używamy słowa kluczowego `implements`:

```
8   /**
9    *
10   * @author Kamil
11   */
     Koszyk is not abstract and does not override abstract method wypisz() in Wypisywalny
     ----
     (Alt-Enter shows hints)
public class Koszyk implements Wypisywalny {
13
14      private Produkt[] produkty;
15
16      public Koszyk() {
17          produkty = new Produkt[5];
18      }
19
20      public void dodajProdukt( Produkt p ) {
21          for (int i = 0; i < produkty.length; i++) {
```

Oczywiście otrzymuję dokładnie ten sam błąd, który poznaliśmy omawiając klasy abstrakcyjne – muszę podać implementację metod zadeklarowanych w interfejsie. Po zaimplementowaniu metody `wypisz()` wszystko jest już OK:

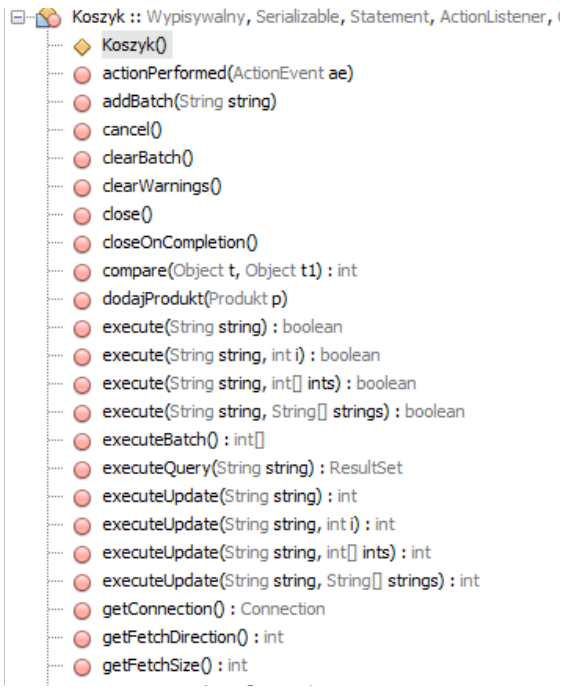
```
33      @Override
34      public String wypisz() {
35          String result = "";
36          for (Produkt p: produkty) {
37              if (p != null)
38                  result += p + " == " + p.getCena();
39          }
40          return result;
41      }
```

## Jedna klasa – wiele interfejsów

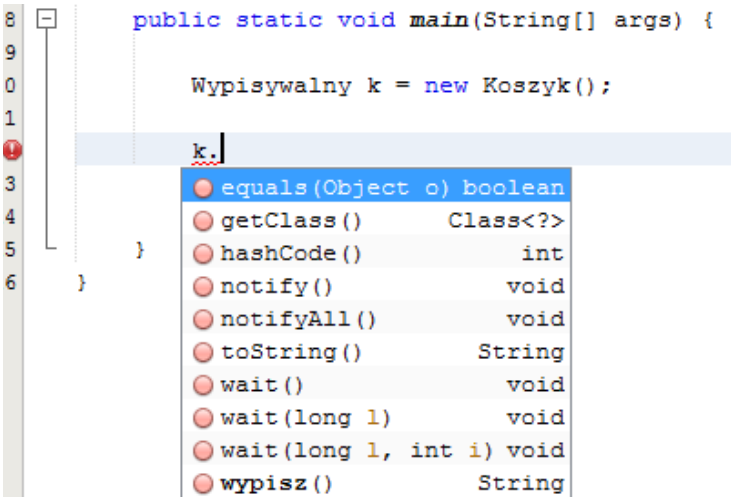
Tak jak powiedziałem w poprzednim rozdziale, interfejsy pozwalają obejść ograniczenie dziedziczenia po jednej klasie – oczywiście – ponieważ jedna klasa może implementować wiele interfejsów:

```
22  public class Koszyk implements Wypisywalny, Serializable, Statement, ActionListener, Comparator {
```

Spójrz teraz ile metod zostało dodanych do klasy Koszyk:




To tylko część z nich ☺. Na razie jednak nigdzie nie widać podobnych korzyści, które mieliśmy w przypadku dziedziczenia – na razie tylko deklaracje metod piszą się za nas ☺. Otóż używając interfejsów, również możemy korzystać z najważniejszego mechanizmu obiektowości - polimorfizmu:




Czyli mogę traktować obiekt jakiejś klasy jako coś, co spełnia pewne zadane przez interfejs funkcjonalności – jak? – nie obchodzi mnie to ani trochę ☺.

### Ważniejsze interfejsy

Na wyższym poziomie Javy nie korzysta się już praktycznie z klas abstrakcyjnych, tylko niemal wyłącznie z interfejsów, ponieważ są bardziej generyczne. Ponadto pozwalają osiągnąć funkcjonalnie

to samo co klasy abstrakcyjne (dopóki nie musimy dziedziczyć pól). Nie chciałbym abyś miał/a mylne przeświadczenie, że interfejsy nie mogą dziedziczyć – mogą jak najbardziej ☺.

Nie chcę się rozwodzić tu nad przydatnością interfejsów – są niezbędne. Omówimy pokrótce kilka ważniejszych interfejsów wykorzystywanych najczęściej:

Interfejs **Comparator** i **Comparable**

Służy do pokazania Javie jak ma porównywać obiekty, np. gdy sortujemy tablicę `int`ów wywołujemy po prostu statyczną metodę `sort()` z klasy `Arrays`:

```
5   public class JavaApplication3 {
6
7       public static void main(String[] args) {
8           int[] tablica = { 435, 345, 3, 5, 4325, 234, 5, 3245};
9           Arrays.sort(tablica);
10          System.out.println( Arrays.toString(tablica) );
11      }
12  }
```

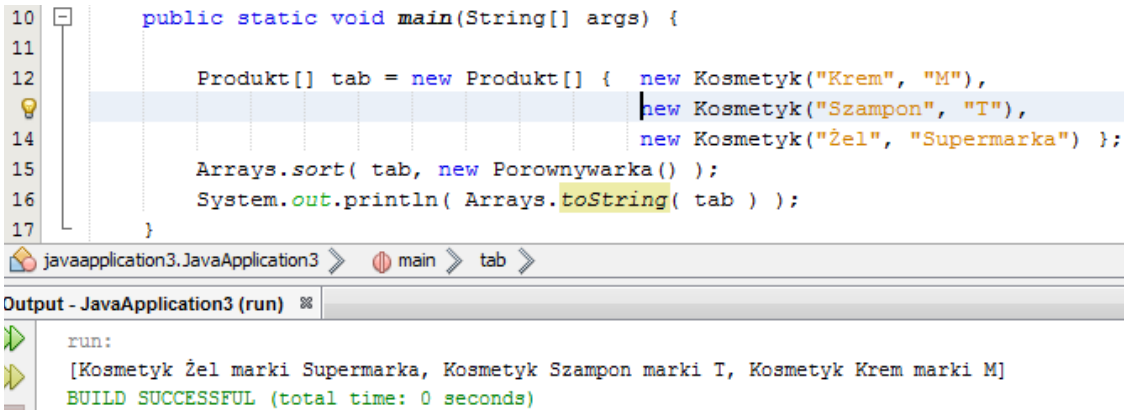
```
javaapplication3.JavaApplication3
Output - JavaApplication3 (run)
run:
[3, 5, 5, 234, 345, 435, 3245, 4325]
BUILD SUCCESSFUL (total time: 1 second)
```

A co jeśli chcielibyśmy posortować tablicę `Produkt`ów? Jak wtedy posortować taką tablicę? Mamy dwa wyjścia – albo klasa `Produkt` musi implementować interfejs `Comparable` (wtedy będziemy mogli sortować Produkt tylko na jeden sposób), albo stworzyć oddzielną klasę, która będzie implementowała interfejs `Comparator` – wtedy możemy stworzyć kilka `Comparator`ów, dla różnych sortowań – skorzystamy z tego drugiego rozwiązania.

```
14  public class Porownywarka implements Comparator<Produkt>{
15
16      @Override
17      public int compare(Produkt t, Produkt t1) {
18          if (t.getCena() < t1.getCena()) {
19              return 1;
20          }
21          return -1;
22      }
23
24  }
```

`Comparator` jest interfejsem tzw. generycznym czyli pomiędzy nawiasami ostrymi podaję typ, jaki będę porównywał (jeśli nie określę – zostanie użyty typ `Object`). Następnie do metody `Arrays.sort()` podaję obiekt tablicy i obiekt `Comparator`a:

Teraz tablica `Produktów` została posortowana malejąco po cenie.




Interfejs **Serializable**:

Omówimy go później – pozwala zapisywać obiekty do plików (ale nie w postaci a'la plik CSV).

Interfejs **Runnable**:

Pozwala uruchamiać wątki, czyli wydzielać część operacji do oddzielnych procesów.

Interfejs **ActionListener**:

Wykorzystywany w tworzeniu Graficznego Interfejsu Użytkownika – pozwala oprogramowywać reakcję na zdarzenia (np. kliknięcie na przycisk).

# Klasy wewnętrzne

## Straszny galimatias, klasy główne niepubliczne

Nie wyszliśmy jeszcze poza schemat – jedna klasa – jedna plik. Czas to zmienić. Otóż w jednym pliku może znajdować się tylko jedna główna klasa publiczna, ale nie ma żadnych ograniczeń co do klas nieoznaczonych żadnym specyfikatorem dostępu (specyfikator domyślny).

```
12    class Siatka {
13
14    }
15    class Reklamowka {
16
17    }
18    public class Koszyk implements Wypisywalny {
19
```

**Rysunek 71 - kilka klas w jednym pliku**

Do takich klas mamy dostęp wyłącznie w obrębie pakietu w którym stworzona została klasa `Koszyk`.

Kiedy używamy takich klas? Bardzo rzadko – niekiedy definiujemy w ten sposób klasy reprezentujące wątki, które chcemy wykorzystywać tylko w jednym module aplikacji. Można natomiast zapewnić dokładnie taką samą funkcjonalność dużo bardziej elegancko.

## Klasy wewnętrzne

Niekiedy istnienie jakiejś klasy ma sens wyłącznie w przypadku, gdy istnieje obiekt jakiejś innej klasy. Np. jakaś klasa potrzebuje do swojego działania jakiejś struktury wykorzystywanej tylko w niej. (np. interfejs `Entry` w interfejsie `Map` – interfejsy zachowują się dokładnie tak jak klasy). Za pomocą klas wewnętrznych zwykle definiuje się typy wykorzystywane tylko w jednej konkretnej klasie, albo definiuje się własne definicje błędów, żeby oznaczyć jedną klasę jako ściśle związaną z drugą.

Klasy wewnętrze udostępniają też coś co nazywamy rzutowaniem w górę (w konwencjonalny sposób rzutujemy w dół – z typu bardziej ogólnego do typu bardziej szczególnego). Jedna jest to już bardzo zaawansowana Java.

Klasy wewnętrzne możemy definiować jako publiczne, prywatne etc., a także jako statyczne.

Poniżej przedstawiam przykład klasy `MailUtil`, która wspomagałaby wysyłanie maili. Natomiast potrzebna jest mi również możliwość wysyłania nie jednego, a dwudziestu tysięcy maili – używał będę wtedy zupełnie innych struktur (miałem okazję pisać coś takiego – naprawdę tak jest ☺) – potrzebna będzie więc klasa, która jest ściśle związana z `MailUtil` (nazwijmy ją `BulkMail`) i tylko `MailUtil` będzie mógł z nią pracować.

Bardzo uproszczony schemacik. Np.:

```
12  public class MailUtil {
13
14      private class BulkMail {
15
16          public void runMailing() {
17              System.out.println("TA METODA WYSYŁA 20 000 maili "
18                      + " i wykorzystuje inne struktury");
19          }
20      }
21
22      public void wyslijMaila() {
23          System.out.println("Ta metoda wysyła 1 maila");
24      }
25
26      public void runMailing() {
27          new BulkMail().runMailing();
28      }
29
30  }
```

Klasa `BulkMail` jest jakby kolejnym elementem obiektu klasy `MailUtil` – czyli do stworzenia obiektu `BulkMail` potrzebny jest obiekt `MailUtil`. Mówimy wtedy, że obiekt `BulkMail` **opiera się** na obiekcie `MailUtil`.

**CIEKAWOSTKA**

Klasy wewnętrzne możemy definiować również w metodach – np. typy danych potrzebne do wykonania pojedynczego wykonania metody. ☺

### Wykorzystanie klas wewnętrznych statycznych

Jeśli nie chcemy żeby definicja jakiejś klasy była fabrykowana przy każdym utworzeniu obiektu jakiejś innej klasy – możemy jej definicję wyciągnąć poza obiekty i umieścić w innej klasie. Oczywiście używamy do tego słowa kluczowego `static` (tak – klasy również mogą być statyczne).

Pokażę teraz bardzo fajny (przynajmniej tak mi się wydaje ☺) przykład zastosowania klas wewnętrznych statycznych.

Gdy tworzymy obiekt jakiejś klasy za pomocą konstruktora, musimy podać pełny zestaw jego argumentów. Niekiedy encje mają znacznie więcej niż piętnaście pól – co więcej nie zawsze uzupełniamy wszystkie pola. Koniec końców, konstruktor – fajna sprawa, ale jakiś on taki strasznie mało elastyczny.

Za pomocą publicznej statycznej klasy wewnętrznej możemy w prosty sposób obejść ten problem:

```
14  public class Pracownik {
15
16      private String imie;
17      private String nazwisko;
18      private Date dataZatrudnienia;
19
20      public static class PracownikBuilder {
21
23          private Pracownik nowy = new Pracownik();
24
25          public PracownikBuilder withImie(String imie) {
26              nowy.imie = imie;
27              return this;
28          }
29          public PracownikBuilder withNazwisko(String nazwisko) {
30              nowy.nazwisko = nazwisko;
31              return this;
32          }
33          public PracownikBuilder withDataZatrudnienia(Date data) {
34              nowy.dataZatrudnienia = data;
35              return this;
36          }
37
38          public Pracownik compile() {
39              return nowy;
40          }
41      }
```

Rysunek 72 - Zastosowane klasy wewnętrznej statycznej

Zauważ, że z poziomu klasy wewnętrznej mam dostęp do prywatnych pól klasy zewnętrznej. Teraz nowego pracownika mogę utworzyć za pomocą poniższego łańcuszka metod:

(tak dla jasności `Pracownik.PracownikBuilder` to pełna nazwa nowopowstałego typu danych).

```
14  public class PracownikTest {
15      public static void main(String[] args) {
16
17          Pracownik p = new Pracownik.PracownikBuilder()
18                                          .withImie( "Jasio" )
19                                          .withNazwisko( "Kowalski" )
20                                          .withDataZatrudnienia( new Date() )
21                                          .compile();
22
23          System.out.println( p );
24      }
25  }
```

klasyWewnetrzne.PracownikTest > main

Output - JavaApplication3 (run)

```
run:
Pracownik{imie=Jasio, nazwisko=Kowalski, dataZatrudnienia=Thu Jul 23 15:29:21 CEST 2015}
BUILD SUCCESSFUL (total time: 1 second)
```

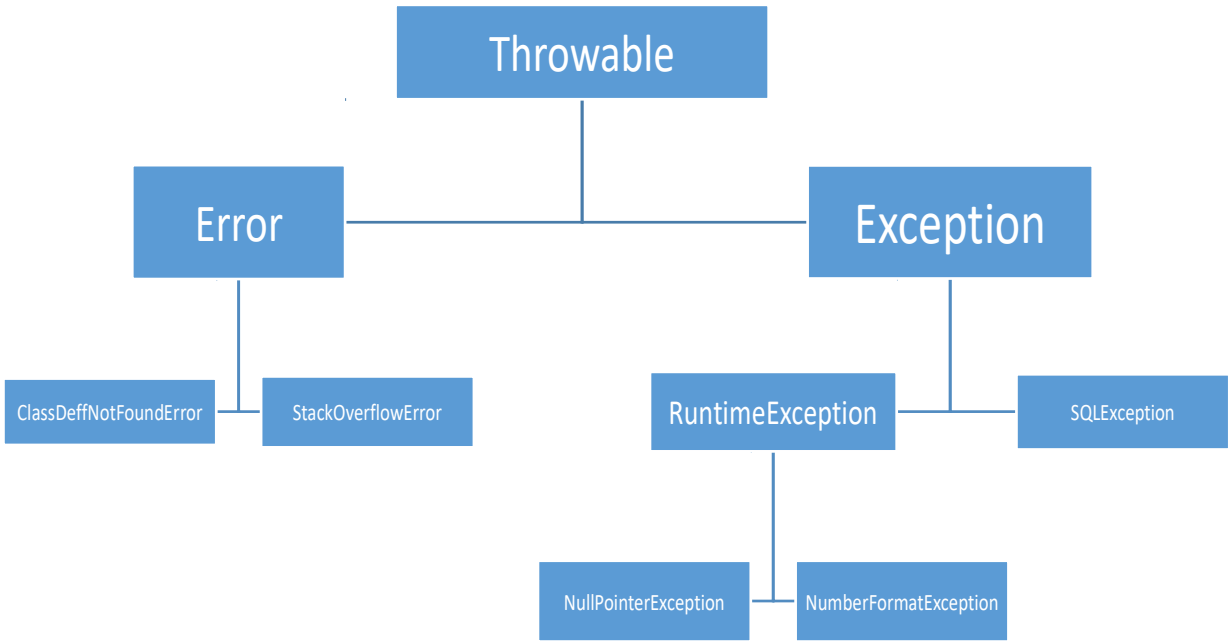
Zwróć uwagę, że wywołuję te metody z klasy `PracownikBuilder` które chcę – nie muszę wpisywać wszystkich wartości i nie muszę przejmować się już kolejnością, nazwijmy to, argumentów.

# Wyjątki i obsługa błędów

### Wyjątek, a błąd – jakieś różnice?

Żaden z naszych programów napisanych do tej pory nie został zabezpieczony przed niespodziankami, które robią nam – programistom, użytkownicy – czyli nieprawidłowymi danymi (np. podanie tekstu zamiast liczby). OK – czasem nawet programiści sami sobie zostawiają podobne niespodzianki. Oczywiście możemy takie rzeczy sprawdzać odpowiednią ilością `if`-ów, ale Java udostępnia gotowy mechanizm obsługi błędów.

W Javie wyróżniamy dwa rodzaje błędów (w rozumieniu nietechnicznym). Mamy bowiem wyjątki i błędy (to są terminy techniczne). Reprezentuje to poniższe drzewko klas:




**Rysunek 73 - Drzewo klas reprezentujących strukturę błędów i wyjątków**

Błąd (`Error`) różni się od wyjątku (`Exception`) tym, że wystąpienie wyjątku oznacza, że coś poszło nie tak w czasie wykonania programu i dotyczy tylko jego. Wystąpienie błędu (`Error`) oznacza błąd po stronie JVM na tyle poważny, że program zawsze zostanir wtedy przerwany. Możemy obsługiwać wyjątki – błędów (`Error`) nawet nie warto, ponieważ i tak ich zaistnienie następuje poza naszym kodem.

### Klauzula try-catch

Przechwycenie błędu w Javie ma następującą składnię:

```
1.  try {
2.      .
3.      .  Operacje, które mogą powodować wyjątki
4.      .
5.  }
6.  catch ([klasa wyjątku] e) {
7.      . Obsługa danego wyjątku
8.  }
```

Oczywiście klauzul `catch()` może być więcej, co zaraz pokażemy na przykładach.

Przypomnę teraz pierwszy program, który pisaliśmy – kalkulator jego kod wyglądał tak:

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

    double liczba1 = 0;
    double liczba2 = 0;

    Scanner s = new Scanner( System.in );
    liczba1 = Double.parseDouble( s.nextLine() );
    liczba2 = Double.parseDouble( s.nextLine() );
    s.close();
```

Nie jest to oczywiście jego pełny kod, ale interesuje nas wyłącznie poniższy kawałek. Popełniliśmy straszny błąd. Instrukcja Double.parseDouble() została pozostawiona sama sobie – nie ma nic, co pozwoliłoby obsłużyć sytuację w której użytkownik zamiast liczby poda tekst. Muszę więc to otrajkeczować ☺.

```
        Scanner s = new Scanner( System.in );
22      try {
23          liczba1 = Double.parseDouble( s.nextLine() );
24          liczba2 = Double.parseDouble( s.nextLine() );
25      }
26      catch (NumberFormatException e) {
            e.printStackTrace();
28      }
```

W klauzuli `try` umieszczam kod, który może rzucać wyjątkami, natomiast w klauzuli `catch()` muszę umieścić informację jakiego wyjątku dotyczy (kaluzul `catch()` może wyć więcej – możemy różnie reagować na różne błędy), a później podać nazwę obiektu wyjątku (tak tak… błędy też są obiektami) w obrębie bloku kodu będącego reakcją na wyjątek – coś jak argument w metodzie. Metoda `printStackTrace()` jest bardzo charakterystyczna dla wyjątków i służy do wypisania wszystkich klas, które wysypały się przez zaistnienie wyjątku na standardowy strumień dla błędów (`System.err`), a także klasy i wiadomości błędu.

Output po podaniu do programu wartości `Ala ma kota`:

```
run:
Ala ma kota
java.lang.NumberFormatException: For input string: "Ala ma kota"
        at sun.misc.FloatingDecimal.readJavaFormatString(FloatingDecimal.java:2043)
        at sun.misc.FloatingDecimal.parseDouble(FloatingDecimal.java:110)
        at java.lang.Double.parseDouble(Double.java:538)
        at klasyWewnetrzne.Kalkulator.main(Kalkulator.java:23)
Dodawanie
0.0 + 0.0 = 0.0
Odejmowanie
0.0 - 0.0 = 0.0
Mnożenie
0.0 * 0.0 = 0.0
Dzielenie
0.0 / 0.0 = NaN
BUILD SUCCESSFUL (total time: 4 seconds)
```

Warto powiedzieć, że gdybyśmy nie dodali klauzuli `try-catch`. Output byłby bardzo podobny, natomiast program zostałby przerwany w momencie zaistnienia wyjątku.

### Klasy wyjątków i deklarowanie własnych

Zajmiemy się przede wszystkim deklarowaniem własnych wyjątków, jednak będzie nam potrzebna pewna wiedza – czym one są fizycznie. Otóż wyjątkiem jest obiekt jakiejkolwiek klasy dziedziczącej po klasie `Exception`. Kolejną ważną rzeczą jest fakt, że tylko metody mogą zgłaszać wyjątki. I ostatnia sprawa – jeśli metoda może zgłosić jakikolwiek wyjątek – to musimy ten wyjątek obsłużyć. Np.:

```
18      public static void main(String[] args) {
19
20          SimpleDateFormat sdf = new SimpleDateFormat("dd-MM-yyyy");
21          String data = "12-12-2005";
22                                  unreported exception ParseException; must be caught or declared to be thrown
23                                  ----
24                                  (Alt-Enter shows hints)
            Date d = sdf.parse(data);
```

Klasa `SimpleDateFormat` odpowiada za konwersję dat z formatu `String` do formatu `java.util.Date`. Parsowanie może się nie udać (czyli może zostać rzucony wyjątek), a my **musimy** go obsłużyć – jeśli go nie obsłużymy – program się nie skompiluje.

Nie musimy obsługiwać każdego wyjątku, który może podnieść metoda **tylko w jednym przypadku** – kiedy klasa wyjątku dziedziczy po `RuntimeException` (oddzielnie zaznaczyłem ją na drzewku). Dziedziczą po niej takie klasy jak `NullPointerException`, `NumberFormatException` etc. Są to wyjątki, które mogą nastąpić dosłownie wszędzie (zwłaszcza `NullPointerException`) i dlatego nie warto ich `try-catch`'ować.

OK. Wiemy już wszystko co nam potrzebne do zadeklarowania własnego wyjątku. Ja stworzę sobie klasę `MechanicznyException`:

```
13
14      public class MechanicznyException extends Exception {
15
16          public MechanicznyException(String message) {
17              super(message);
18          }
19
20      }
```

Rysunek 74 - Deklaracja własnego wyjątku

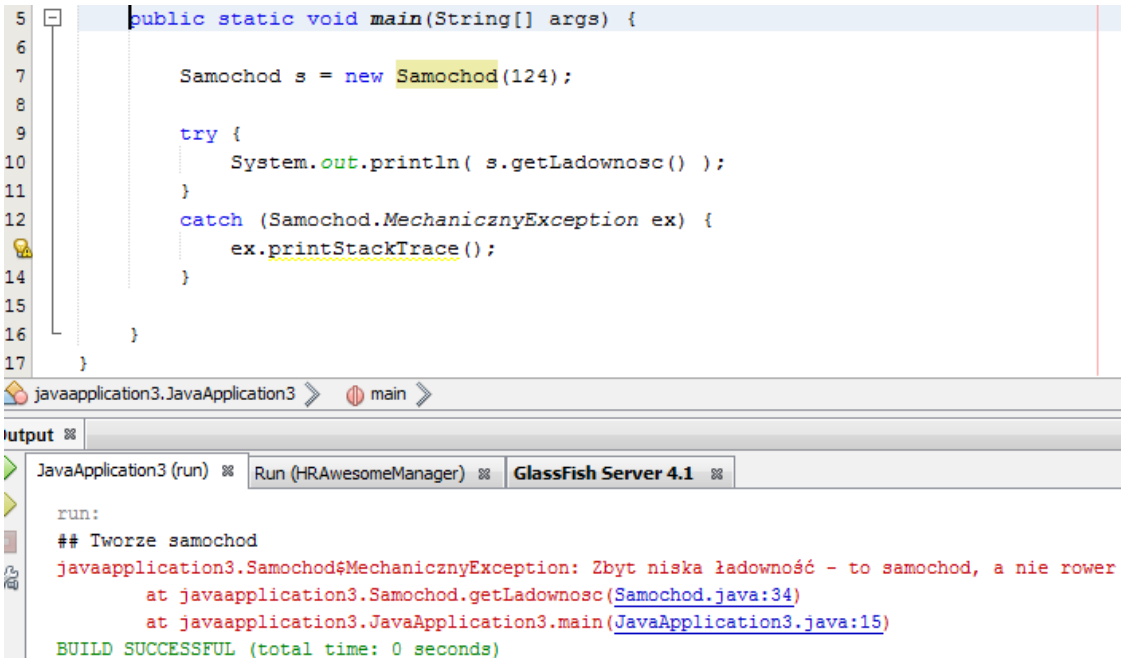
I to tyle ☺. Teraz mam już własny wyjątek i chcę, aby ewentualnie podnosiła go metoda `getLadownosc()` obiektu `Samochod`. Ponadto, MechanicznyException jest ściśle związany z klasą Samochod, więc wydzielę go jako klasę publiczną wewnętrzną statyczną.

Klasa `Samochod` teraz wygląda tak:

```
15  public class Samochod {
16
17      public static class MechanicznyException extends Exception {
18
19          public MechanicznyException(String message) {
20              super(message);
21          }
22      }
23      private double ladownosc;
24
25      public Samochod(double ladownosc) {
26          this.ladownosc = ladownosc;
27          System.out.println("## Tworze samochod");
28      }
29
30      public double getLadownosc() throws MechanicznyException {
31          if (ladownosc < 1000)
32              throw new MechanicznyException("Zbyt niska ładowność - to samochod, a nie rower");
33
34          return ladownosc;
35      }
36  }
```

Interesuje nas z tego wszystkiego przede wszystkim metoda `getLadownosc()`. Podniesienie wyjątku uruchamiamy komendą `throw` po której musimy podać obiekt wyjątku, który ma zostać rzucony. Jeśli rzucany wyjątek nie jest typu `RuntimeException`, musimy na etapie deklarowania metody dodać klauzulę `throws` i po przecinku podać klasę lub klasy wyjątków, które mogą zostać zgłoszone przez metodę.

Teraz każde wywołanie metody `getLadownosc()` będzie pociągać za sobą konieczność otoczenia go `try-catch`'em.




Klauzula finally

W Javie wszystkie operacje wejścia-wyjścia są obsługiwane za pomocą strumieni (omówimy je zresztą bardzo dokładnie). W przypadku strumieni obowiązkowe jest ich zamykanie – metodą `close()`. Pojawia się problem – operacje IO są generalnie bardzo błędogenne i każda komunikacja z jakimś odbiornikiem może podnosić wyjątek `IOException`. Żeby zachować nasz kod w czytelnej postaci warto jest użyć klauzuli `finally`, która jest wykonywana zawsze gdy program opuści bloki `try-catch`.

Przykład:

```
14          FileOutputStream fos = null;
15          try {
16              fos =  new FileOutputStream( new File("nowyplik") );
17              fos.write(null);
18          }
19          catch (FileNotFoundException e) {
20              e.printStackTrace();
21          }
22          catch (IOException e) {
23              e.printStackTrace();
24          }
25          catch (Exception e) {
26              e.printStackTrace();
27          }
28          finally {
29              try {
30                  fos.close();
31              }
32              catch (Exception e) {
33                  e.printStackTrace();
34              }
35          }
```

Spójrz jak straszny robi się bałagan. Klauzula `finally` nie jest tu konieczna (gdybym ją usunął, kod działałby tak samo), ale łączy cały ten rozbudowany łańcuch w jedną, w miarę spójną, całość.

### Try-catch with resources – nowa, lepsza metoda niż użycie finally

Powyższy przykład można zmodyfikować do postaci:

```
try (FileOutputStream fos =  new FileOutputStream( new File("nowyplik") );) {

    fos.write(null);
}
catch (FileNotFoundException e) {
    e.printStackTrace();
}
catch (IOException e) {
    e.printStackTrace();
}
```

**Rysunek 75 - try-catch with resources**

Jest to metoda `try-catch`owania dedykowana dla wszystkich klas (wprowadzona w JDK 1.7), dla których musi zostać wywołana metoda `close()`, ponieważ ta metoda zostanie wywołana zawsze po zakończeniu bloku `try-catch` (nie musimy używać już `finally`).

W tej chwili obiekt `FileOutputStream` staje się zasobem boku `try-catch` i jest widoczny zarówno w bloku `try` jak również w blokach `catch`.

Używając `try-catch with resources` musimy wiedzieć o kliku rzeczach, które regulują jego użycie. Po pierwsze wszystkie elementy które oznaczymy jako `resource` stają się finalne. Po drugie, aby oznaczyć obiekt naszej klasy jako `resource` – klasa musi implementować jeden z interfejsów:

- `Closeable`
- `AutoCloseable`

Zaimplementuję ten interfejs do klasy `Samochod` i oznaczę go jako `resource`:

Klasa `Samochod`:

```
15
16      @Override
17      public void close() {
18          System.out.println("## SAMOCHOD ZOSTAŁ POPRAWNIE ZAMKNIĘTY :)");
19      }
20
21      public static class MechanicznyException extends Exception {
22
```

A teraz wywołanie i output:

```
5      public static void main(String[] args) {
6
7          try ( Samochod s = new Samochod(400); ) {
8              s.getLadownosc();
9          }
10         catch (Samochod.MechanicznyException e) {
12             e.printStackTrace();
13         }
           catch (Exception e) {
15             e.printStackTrace();
16         }
17     }
```

javaapplication3.JavaApplication3

utput

JavaApplication3 (run) | Run (HRAwesomeManager) | GlassFish Server 4.1

```
run:
## Tworze samochod
## SAMOCHOD ZOSTAŁ POPRAWNIE ZAMKNIĘTY :)
javaapplication3.Samochod$MechanicznyException: Zbyt niska ładowność - to samochod, a nie rower
        at javaapplication3.Samochod.getLadownosc(Samochod.java:36)
        at javaapplication3.JavaApplication3.main(JavaApplication3.java:9)
BUILD SUCCESSFUL (total time: 0 seconds)
```

### Propagacja wyjątków, czyli co dzieje się gdy rzucany jest wyjątek

Dla bardziej skomplikowanych układów (a będziemy z takimi pracować omawiając strumienie), warto wiedzieć jak przebiega obsługa wyjątku w bloku `try-catch`.

Jeśli pojawia się wyjątek w sekcji, która nie ma `try-catcha`, następuje wyświetlenie `StackTrace` i przerwanie programu. A jak wygląda obsługa wyjątku w przypadku zagnieżdżenia klauzuli `try-catch`?

Otóż Java najpierw przerywa blok `try` w którym nastąpił wyjątek i szuka liniowo najbliższej sekcji `catch`, której klasa wyjątku odpowiada klasie wyjątku zgłoszonego (**polimorfizm cały czas działa!**) i jeśli znajdzie pasującą to po prostu wykona jej ciało i przejdzie dalej (pomijając kolejne `catch`'e). Jeśli nie znajdzie dalej przerwie kolejny blok `try` i znowu będzie szukać pasującego `catch`'a. Jeśli i tam nie znajdzie – wtedy przerwie wątek.

```
1.          try {
2.              try {
3.                  // Tu mamy wyjątek (ParseException) - przerwanie bloku try
4.                  System.out.println("Chyba wszystko OK"); //instrukcja się nie wykona
5.              }
6.              catch (IOException e) {  // Nie pasuje
7.                    e.printStackTrace();
8.              }
9.              catch (NumberFormatException e) {  // Nie pasuje - przerywa blok try!!!
10.                   e.printStackTrace();
11.             }
12.         }
13.         catch (NullPointerException e) {  // Nie pasuje
14.             e.printStackTrace();
15.         }
16.         catch (RuntimeException e) { // Nie pasuje
17.             e.printStackTrace();
18.         }
19.         catch (Exception e) {  // Pasuje - wykonanie bloku catch
20.             e.printStackTrace();
```

```
21.         }
```

A teraz bez ostatniej sekcji `catch(Exception e)`:

```
1.          try {
2.              try {
3.                  // Tu mamy wyjątek (ParseException) - przerwanie bloku try
4.                  System.out.println("Chyba wszystko OK"); //instrukcja się nie wykona
5.              }
6.              catch (IOException e) {  // Nie pasuje
7.                    e.printStackTrace();
8.              }
9.              catch (NumberFormatException e) {  // Nie pasuje - przerywa blok try!!!
10.                   e.printStackTrace();
11.             }
12.         }
13.         catch (NullPointerException e) {  // Nie pasuje
14.             e.printStackTrace();
15.         }
16.         catch (RuntimeException e) { // Nie pasuje - nastąpi przerwanie wątku!!!
17.             e.printStackTrace();
18.         }
```

## Debugowanie programu

Debugowanie programu polega głównie na podejrzeniu zawartości obiektów, albo zmiennych w pewnym momencie. Taki moment nazywamy `breakpoint`'em. Możemy uzupełnić nasz program o mnóstwo instrukcji `System.out.println()`, ale jest do tego dedykowane narzędzie - `Debugger`. `Debugger` dostępny w NetBeans działa w ten sposób, że będzie zatrzymywał się w każdym miejscu, które oznaczymy jako `breakpoint` i udostępni nam możliwość podejrzenia wszystkich zmiennych zadeklarowanych do tej linijki.

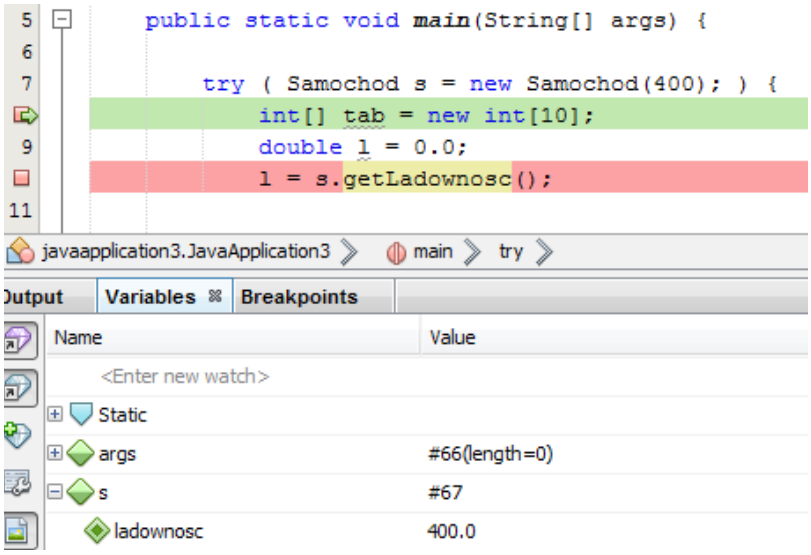
Aby oznaczyć jakąś linijkę jako `breakpoint` wystarczy, że kliknę w numer linijki:



Rysunek 76 - oznaczanie linijki jako breakpoint

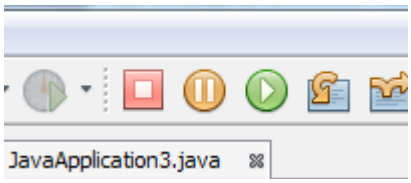
```
7           try ( Samochod s = new Samochod(400); ) {
            int[] tab = new int[10];
9           double l = 0.0;
            l = s.getLadownosc();
11
12          }
```

Aby rozpocząć proces debugowania wybieram `Debug >> Debug Poject`, albo używam skrótu klawiszowego `Ctrl + F5`.




Gdy `Debugger` dojdzie do linijki oznaczonej jako `breakpoint` wstrzymuje wykonanie programu. Możemy teraz podejrzeć wartość każdej zmiennej zadeklarowanej do linijki w której znajduje się `Debugger`.

Dodatkowo mamy możliwość kontynuowania wykonania programu, lub jego przerwania – pojawia się bowiem dodatkowe menu w pasku narzędziowym NetBeans'a:




**Rysunek 77 - Menu Debugger'a**

# Kolekcje – zbiory ciekawsze od tablic

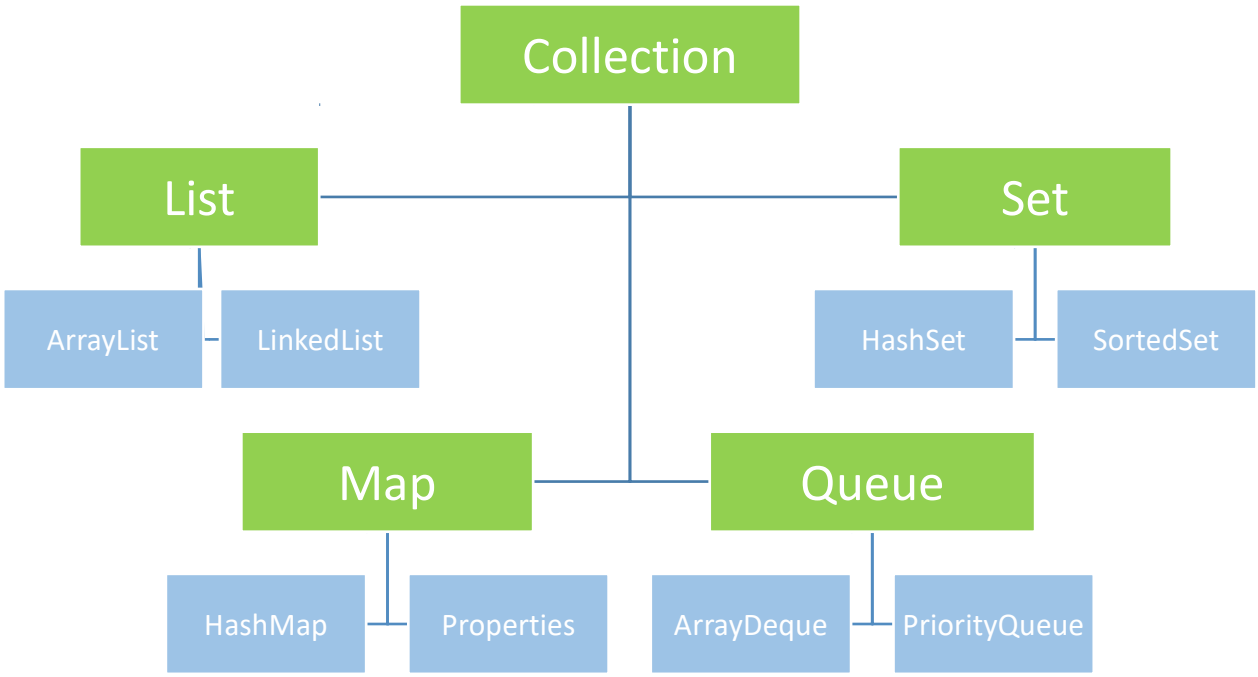
### Po co kolekcje?

Do reprezentacji jakichkolwiek zbiorów w naszych programach używaliśmy tablic. Są one jednak strasznie niewygodne i ograniczone funkcjonalnie Nie mamy możliwości automatycznego dodania obiektu na ostatni wolny indeks tablicy, nie możemy rozszerzyć tablicy etc. W pakiecie `java.util` znajdują się odpowiednie klasy i interfejsy odpowiedzialne za obsługę bardziej praktycznych zbiorów. Mamy tam m.in. klasy pozwalające na obsługę:

- List
- Zbiorów w sensie matematycznym (tzn. bez powtórek)
- Map – czyli reprezentacji zjawiska asocjacji w Javie
- Kolejek

### Kolekcje w pakiecie java.util

Zaczniemy od drzewka klas:




**Rysunek 78 - drzewko klas i interfejsów podstawowych kolekcji**

Oto podstawowe kolekcje języka Java (wszystkich jest ich znacznie, znacznie więcej). Kolorem zielonym oznaczyłem interfejsy, a kolorem niebieskim klasy je implementujące.

Aby jakaś klasa mogła stać się listą musi implementować interfejs list – dopiero zestaw konkretnych metod określa czy jakaś klasa może być uznana za listę czy kolejkę.

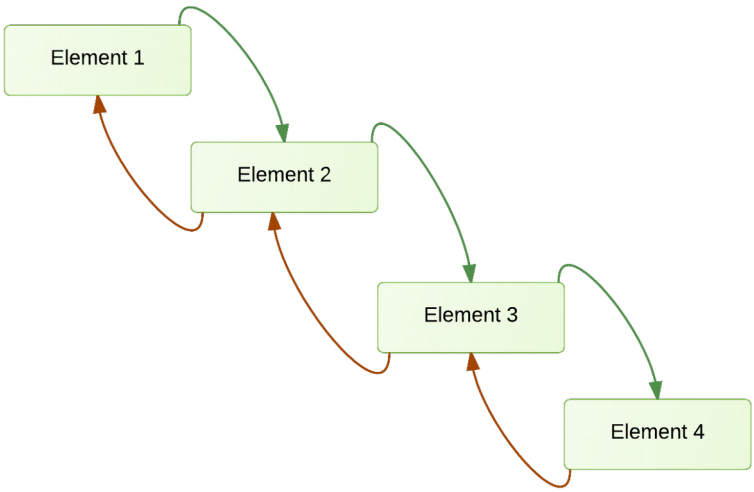
Omówimy sobie każdą z tych kolekcji w szczególności skupiając się na listach i mapach.

### Lista – najczęściej używana kolekcja

Lista jest najprostszą z kolekcji. Możemy ją traktować jako taką fajną tablicę, która dopasowuje się do ilości elementów, jednocześnie udostępniając nam wiele akcji wspomagających pracę z nią. Np.:

- Przeszukanie listy w celu odnalezienia konkretnego elementu
- Sprawdzenie czy konkretny element znajduje się na liście
- Skasowanie elementu na podanym indeksie lub podanie obiektu do wyrzucenia z listy
- Dodanie jednego elementu lub przekopiowanie elementów z innej kolekcji

Ale jaki pomysł leży u podstaw listy – jak to działa? Aby stworzyć bardzo prostą listę wystarczy do klasy dodać pola reprezentujące element poprzedni i następny - element pierwszy w polu `poprzedniObiekt` miałby `null`.




**Rysunek 79 - Powiązanie kolejnych elementów - istota listy**

Listę deklarujemy w następujący sposób:

```
1.  List<[typ elementów]> [nazwa] = new ArrayList<[typ elementów]>();
```

Np.:

```
List<String> lista = new ArrayList<String>();
```

Kolekcje są tzw. typami generycznymi i stąd bierze się określanie typu między nawiasami ostrymi – tak

naprawdę mamy z tego wyłącznie korzyści. Gdyby nie typy generyczne – elementy listy zawsze musiałyby być typu `Object`, a tak – od razu wyjmiemy z niej obiekt odpowiedniej klasy.

Co w ogóle oznacza `ArrayList`? Na drzewku kolekcji zaznaczyłem, że List to interfejs, a `ArrayList` i `LinkedList` to konkretne jego implementacje. OK. A na czym polega różnica pomiędzy `ArrayList` (lista tablicowa), a `LinkedList` (lista wiązana). Różnica jest jak widać na poziomie implementacji. Pod spodem `ArrayList` wykorzystuje zwykłe tablice – tak sprytnie natomiast oprogramowane, że spełniają wszystkie funkcjonalności listy. Lista tablicowa daje niemal zerowy czas odczytu danych z niej, ale wszelkie modyfikacje są dużo trudniejsze. Lista wiązana spełnia dokładnie schemat z powyższego rysunku – kolejne obiekty są uzupełniane o dwie referencje (element poprzedni i następny). Dzięki temu wszelkie modyfikacje listy ograniczają się do kilku bardzo prostych operacji. Dużo dłużej trwają natomiast odczyty.

Zastanawiające może być dla Ciebie również niby niepotrzebne używanie polimorfizmu. Przecież mogłem napisać „normalnie" – nie używając polimorfizmu. Z takiego (tzn. **tego polimorficznego**) zapisu mamy same korzyści. Po pierwsze, lista jest interfejsem, a `ArrayList` konkretną implementacją tegoż interfejsu – używając polimorfizmu możemy bezkolizyjnie zmienić tę implementację. Takie polimorficzne deklarowanie jest uniwersalne dla wszystkich kolekcji.

Przykład użycia listy:

```
20          List<String> lista = new ArrayList<String>();
21          String tekst = "ala";
22          lista.add( tekst );
23          lista.add( "ma" );
24          lista.add( "kota" );
25          lista.add( tekst );
26
27          for ( String s: lista) {
28              System.out.println( s );
29          }
30      }
```

klasyWewnetrzne.Kalkulator › main

utput

GlassFish Server 4.1 | Run (HRAwesomeManager) | JavaApplication3 (run)

```
run:
ala
ma
kota
ala
BUILD SUCCESSFUL (total time: 0 seconds)
```

Zauważ, że powtórki tego samego obiektu są traktowane w liście jako oddzielne elementy. Iteracja po listach pętlą `for-each` nie różni się niczym od iteracji po tablicy. Jednak o ile przy tablicach nie byliśmy w stanie w ogóle jej zmodyfikować, o tyle w przypadku list możemy wykonać modyfikacje na niej jeśli nie wpływają one na ilość elementów w liście – np. metoda `set()`.

Jeśli w trakcie iteracji po liście pętlą `for-each`, wywołamy na niej metody zmieniające ilość elementów znajdujących się w niej – otrzymamy `ConcurrentModificationException`.

```
            for ( String s: lista) {
                lista.remove(s);
            }
```

klasyWewnetrzne.Kalkulator > main > for (String s : lista) >

Output

GlassFish Server 4.1 | Run (HRAwesomeManager) | JavaApplication3 (run)

```
run:
Exception in thread "main" java.util.ConcurrentModificationException
        at java.util.ArrayList$Itr.checkForComodification(ArrayList.java:901)
        at java.util.ArrayList$Itr.next(ArrayList.java:851)
        at klasyWewnetrzne.Kalkulator.main(Kalkulator.java:26)
Java Result: 1
BUILD SUCCESSFUL (total time: 0 seconds)
```

Przyczyną tego wyjątku jest to, że w trakcie iteracji po jakimś elemencie pętlą `for-each` używany jest interfejs `Iterator`, a lista tablicowa ma własną implementację `Iterator`'a w postaci klasy wewnętrznej.

Rozwiązaniem tego problemu jest iteracja po kopii naszej listy. Klasa `ArrayList` ma zdefiniowany konstruktor w którym jako argument podajemy inną kolekcje której elementy są tego samego typu. Np.:

```
        List<String> lista = new ArrayList<String>();
        List<String> lista2 = new ArrayList<String>( lista );
```

**Rysunek 80 - kopiowanie zawartości listy**

Teraz możemy iterować po innej liście, a usuwać elementy z innej:

```
        List<String> lista2 = new ArrayList<String>( lista );
        System.out.println( lista );
        for ( String s: lista2) {
            lista.remove(s);
        }
        System.out.println( lista );
    }
```

klasyWewnetrzne.Kalkulator > main >

Output

GlassFish Server 4.1 | Run (HRAwesomeManager) | JavaApplication3 (run)

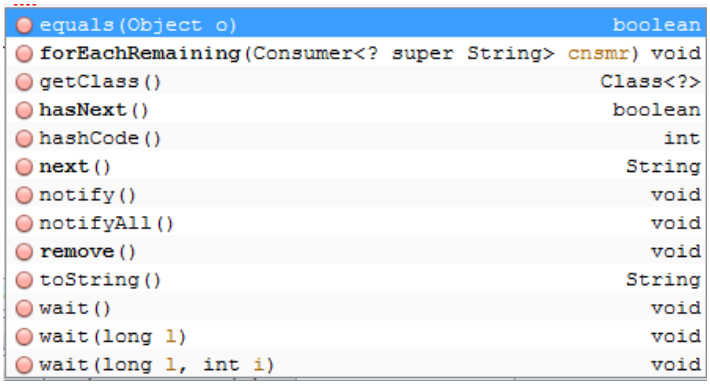
```
run:
[ala, ma, kota, ala]
[]
BUILD SUCCESSFUL (total time: 1 second)
```

Jeszcze inaczej można rozwiązać ten problem używając wprost `Iterator`'a. Obiekt typu `Iterator` możemy wyjąć z prawie każdej kolekcji:

```
        System.out.println( lista );
        for (Iterator<String> i = lista.iterator(); i.hasNext() ; ) {
            i.next();
            i.remove();
        }
        System.out.println( lista );
```

**Rysunek 81 - Iteracja po liście za pomocą Iterator'a**

Wywołanie `i.next()` oznacza zwrócenie kolejnego elementu z listy. Należy jednak pamiętać, że `Iterator` ma dość ograniczoną funkcjonalność:




Co można trzymać w kolekcjach, a czego nie – klasy opakowujące

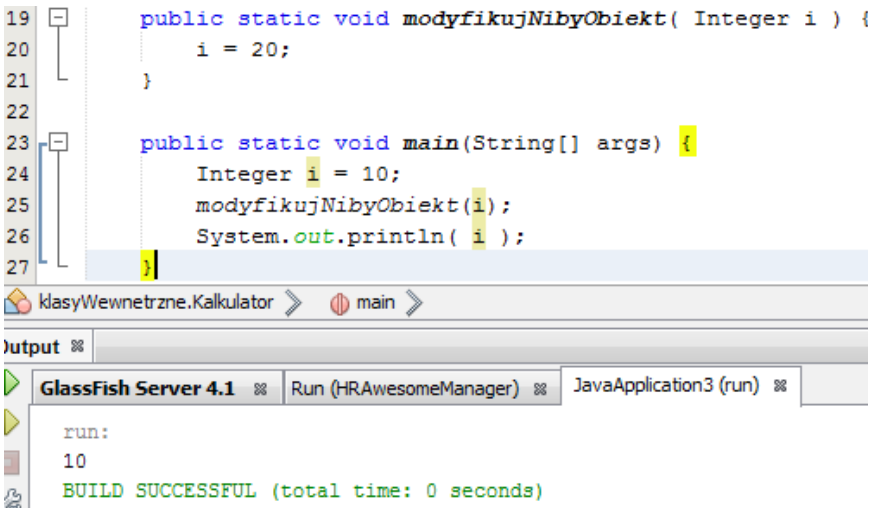
**WAŻNE!**

W kolekcjach składować możemy wyłącznie elementy typów obiektowych. Nie można stworzyć listy `int`ów.

Jak w takim razie stworzyć listę liczb całkowitych? Między innymi w tym celu prowadzono w Javie klasy opakowujące dla typów prostych, tak, aby posiadały klasy i dało się stworzyć kolekcje je przechowujące.

- Klasa **`Integer`** – dla typu `int`
- Klasa **`Double`** – dla typu `double`
- Klasa **`Character`** – dla typu `char`
- Klasa **`Boolean`** – dla typu `boolean`

Używamy ich dokładnie tak samo jak odpowiedników nieobiektowych. Należy jednak pamiętać, że nie są to takie „normalne” obiekty. Typy powyższych klas opakowujących zachowują się dokładnie tak samo jak ich nieobiektowe odpowiedniki:




Zadeklarowanie kolekcji np. liczb całkowitych wygląda tak:

```
        List<Integer> lista = new ArrayList<Integer>();
20      lista.add( 2 );
21      lista.add( 4 );
22      lista.add( 6 );
23      System.out.println( lista );
```

## Zbiory – listy bez powtórzeń

Zbiór (Set) w Javie to dość rzadko używana kolekcja, niemniej jest ważną składową kolekcji Map. Od listy odróżnia go to, że Set nie dopuszcza do zaistnienia powtórek.

Deklaracja zbioru wygląda tak:

```
        Set<String> zbior = new HashSet<String>();
```

Oczywiście to wszystko, co obowiązywało przy listach obowiązuje również przy zbiorach (np. `ConcurrentModificationException` przy pętli `for-each`).

Jak już wspomniałem, zbiór nie dopuszcza powtórek, sprawdźmy to:

```
        Set<Samochod> zbior = new HashSet<Samochod>();
23      zbior.add( new Samochod(100) );
24      zbior.add( new Samochod(100) );
25      System.out.println( zbior );
26  }
```

klasyWewnetrzne.Kalkulator

utput | GlassFish Server 4.1 | Run (HRAwesomeManager) | JavaApplication3 (run)

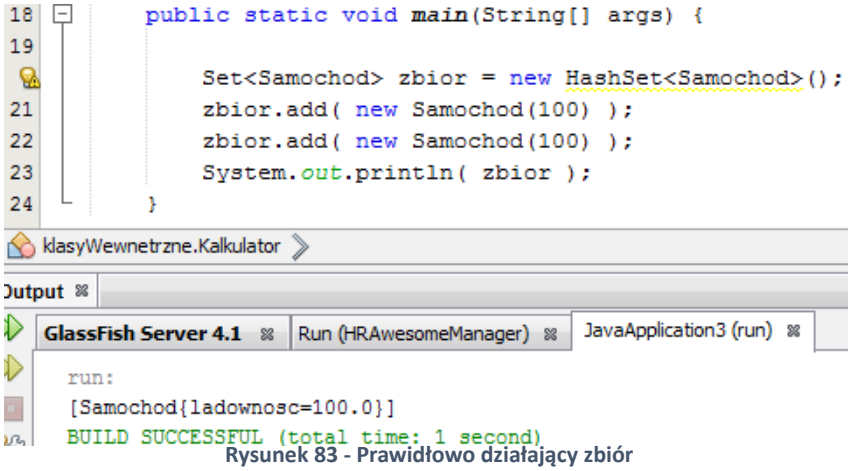
```
run:
[Samochod{ladownosc=100.0}, Samochod{ladownosc=100.0}]
BUILD SUCCESSFUL (total time: 0 seconds)
```

Czyżby zbór używać porównania referencyjnego? Nic podobnego! Używa normalnego `equals`'a. Niestety my nie pokazaliśmy Javie, jak porównuje się obiekty typu `Samochod`. Zrobimy to teraz przesłaniając metodę `equals( Object o )` w klasie `Samochod`.

```
33      @Override
        public boolean equals(Object obj) {
35          if (obj == null) {
36              return false;
37          }
38          if (getClass() != obj.getClass()) {
39              return false;
40          }
41          Samochod other = (Samochod) obj;
42          return this.ladownosc == other.ladownosc;
43      }
```

**Rysunek 82 - Przeciążenie metody equals()**

(Jak widać klasy w Javie też mogą stać się obiektami – można z nich wyciągnąć bardzo dużo informacji nawiasem mówiąc). Teraz kod z poprzedniego przykładu da już prawidłowy wynik:



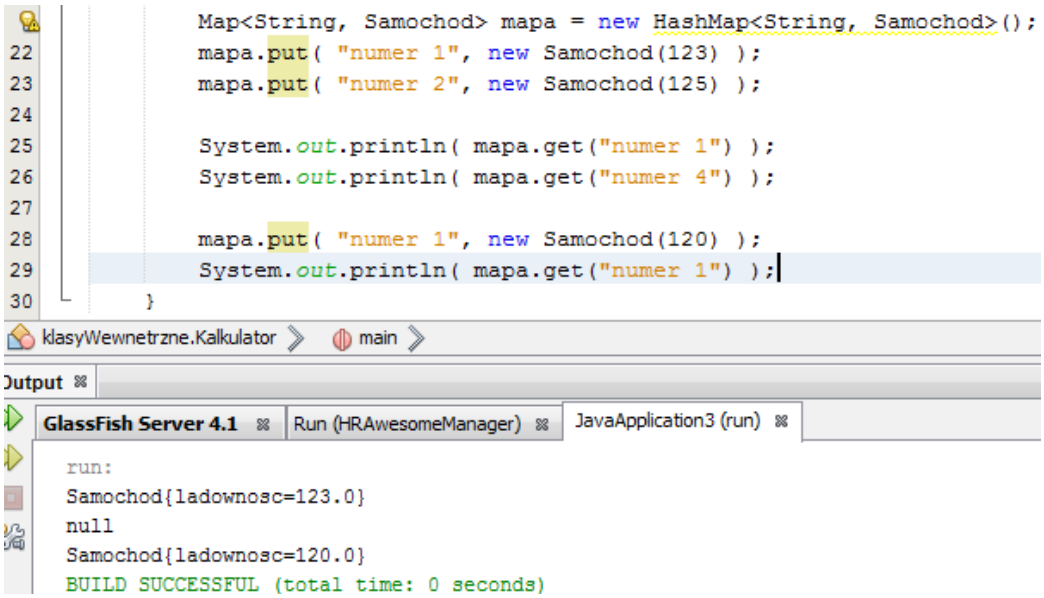

**Rysunek 83 - Prawidłowo działający zbiór**

## Mapy – asocjacja

W programowaniu bardzo często musimy powiązać pewną wartość z jakimś kluczem. Oczywiście zarówno klucz jak i wartość mogą być dowolnego typu obiektowego.

Mapa składa się ze zbioru kluczy i specjalnego typu `Values`, będącego klasą wewnętrzną implementacji mapy, reprezentującą wartości. Próba pobrania z mapy elementu od nieistniejącego klucza nie powoduje wyjątku, tylko zwrócenie wartości `null`.

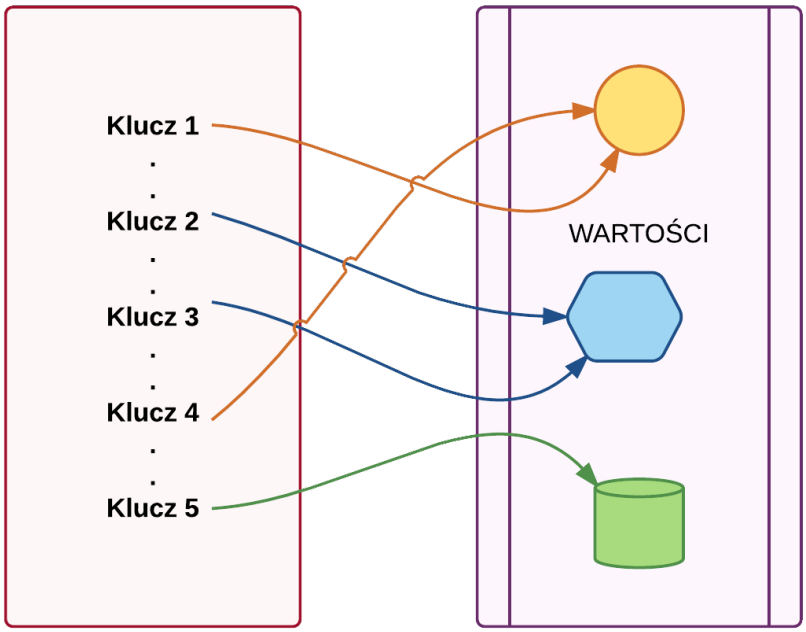
Deklaracja i użycie mapy wygląda np. tak:




Zauważ, że ponowne dodanie elementu na istniejący już klucz powoduje zaktualizowanie tegoż elementu.



**Rysunek 84 - Konstrukcja mapy**




OK, skoro mamy asocjację – jak w takim razie iterować po mapach? Służy do tego specjalny typ `Entry`, który jest interfejsem wewnętrznym w interfejsie `Map`. Reprezentuje on parę klucz-wartość. Używamy go w trochę nieintuicyjny sposób. (metoda `entrySet()` zwraca zbiór elementów typu

```
22          Map<String, Samochod> mapa = new HashMap<String, Samochod>();
23          mapa.put( "numer 1", new Samochod(123) );
24          mapa.put( "numer 2", new Samochod(125) );
25
26          for ( Map.Entry<String, Samochod> entry: mapa.entrySet() ) {
27              System.out.println( entry.getKey() + ": " + entry.getValue() );
            }
```

klasyWewnetrzne.Kalkulator › main

Output

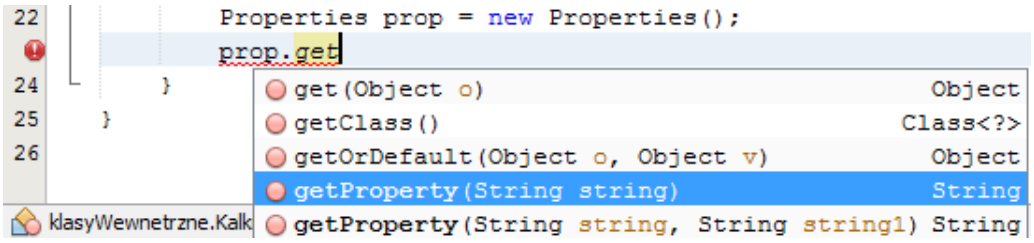
GlassFish Server 4.1 | Run (HRAwesomeManager) | JavaApplication3 (run)

```
run:
numer 1: Samochod{ladownosc=123.0}
numer 2: Samochod{ladownosc=125.0}
BUILD SUCCESSFUL (total time: 0 seconds)
```

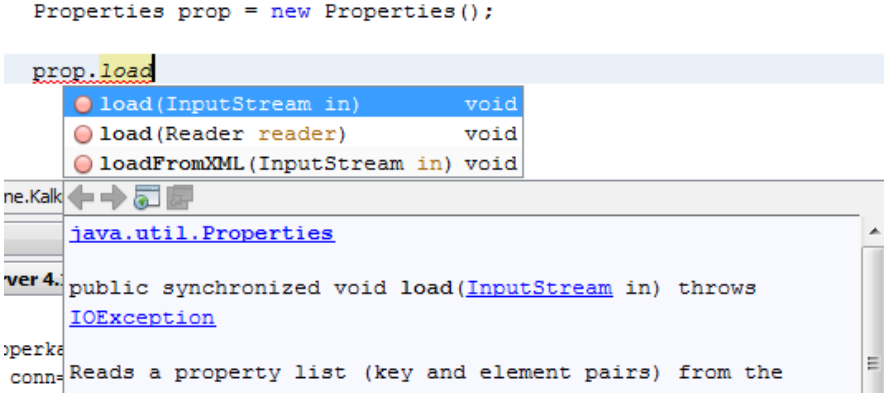
`Entry`):

Szczególnym typem mapy jest klasa Properties. Jest to kolekcje, która nie implementuje bezpośrednio interfejsu `Map` i jest zdolna do przechowywania par `Object-Object`. Została jednak stworzona przede wszystkim do przechowywania par `String-String`, dlatego dodano w niej metody wspomagające pracę z tzw. właściwościami – np. `getProperty()`, `setProperty()`, a także o metody wspomagające ładowanie właściwości z plików XML – metoda `load()`. Oczywiście dalej mamy dostęp do wszystkich metod z interfejsu `Map`.

Ta kolekcja jest intensywnie używana przy np. `JavaMail`.




Tu mamy wyjątek – kolekcji `Properties` nie deklarujemy polimorficznie. Jeszcze tylko metody pozwalające ładować dane do kolekcji:




**Rysunek 85 - Ładowanie danych z pliku *.properties**

## Kolejki – szeregowanie elementów, FIFO (First In First Out)

Kolejki pozwalają szeregować elementy (podobnie jak listy) w kolejności w jakiej zostały dodane. Oferują jednak ograniczenie rozmiaru kolejki, sprawdzanie, czy udało się dodać element do kolejki, a także znacznie bardziej rozbudowany zestaw środków do pobierania elementów z niej.

Dwie podstawowe implementacje interfejsu `Queue` to `ArrayDeque` oraz `PriorityQueue`.

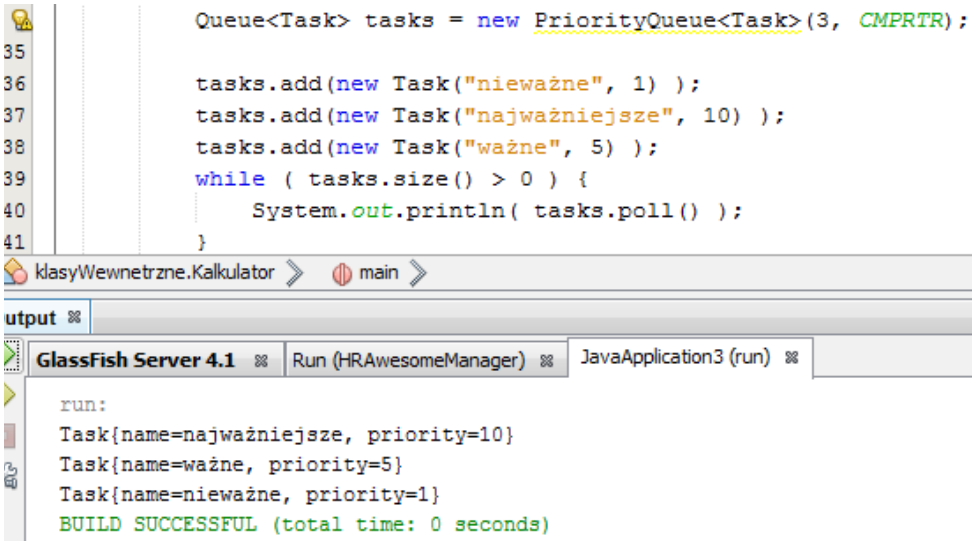
Kolejkę deklarujemy jak poniżej:

```
Queue<String> q = new ArrayDeque<String>();
```

`ArrayDeque` nie jest ograniczona przez rozmiar – dopasowuje się do ilości elementów. Jako kolejka ma metody dla nich charakterystyczne:

- `poll()` – zwraca i usuwa element znajdujący się na szczycie kolejki
- `peek()` – to samo co `poll()`, ale nie usuwa nic z kolekcji
- `remove()` – to samo co `poll()`, ale podnosi wyjątek jeśli kolejka jest pusta (`poll()` zwraca `null`)
- `offer()` – podejmuje próbę dodania elementu na koniec kolejki – zwraca `true` jeśli operacja się powiodła, `false`, jeśli przekroczymy ładowność kolejki

Ciekawszą implementacją kolejki jest kolejka priorytetowa (`PriorityQueue`). Działa w ten sposób, że dodając nowy element do kolekcji od razu wskakuje on na pozycję wynikającą z jakiegoś sortowania (interfejs `Comparator`). Deklaryjemy ją w następujący sposób:



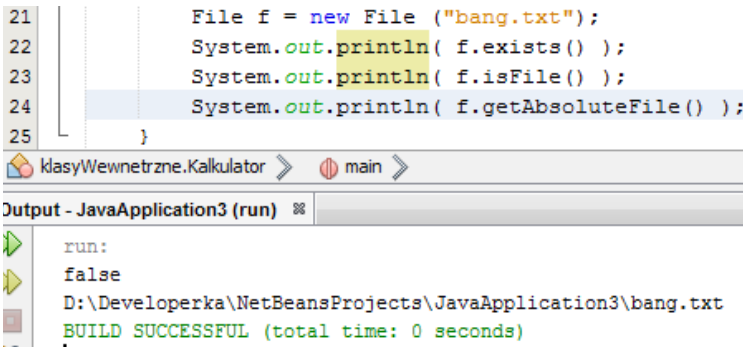

Zmienna `CMPRTR` to obiekt `Comparator` sortujący moją kolejkę malejąco po priorytecie. Wartość `3` w konstruktorze kolejki określa startową objętość kolejki (**nie oznacza ograniczenia ładowności!**)

Ważne jest, żeby nie używać iteracji za pomocą `for-each`'a – dostaniemy wtedy elementy w takiej kolejności w jakiej dodaliśmy je do kolejki (nieposortowane po priorytecie).

# Pierwsze operacje IO – obsługa plików

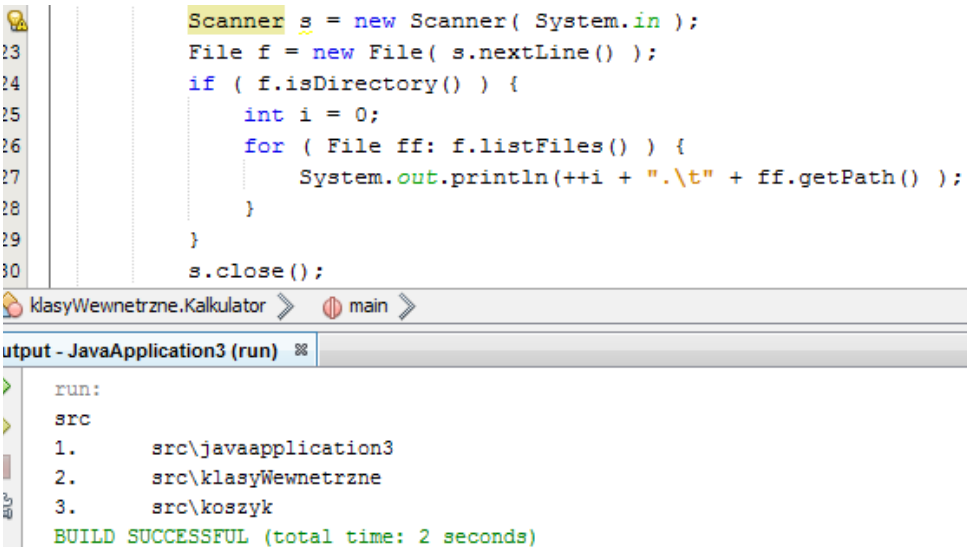
### Zbieranie podstawowych informacji o plikach

W Javie niemal wszystko jest obiektem, tak więc nietrudno się domyśleć, że za obsługę plików odpowiada klasa `File`. Samo stworzenie obiektu pliku w programie nie pociąga za sobą jeszcze żadnych konsekwencji trwałych – tzn. na dysku jeszcze nic się nie stanie. Warto jeszcze tylko powiedzieć, że foldery w Javie również obsługujemy za pomocą klasy `File`.

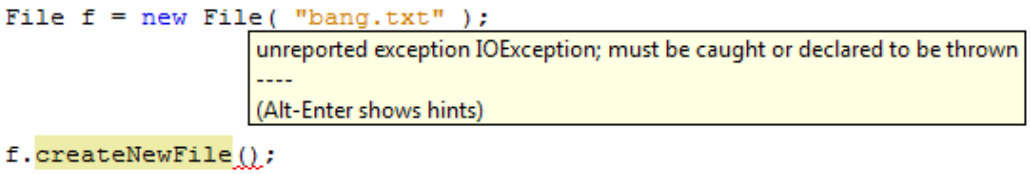



Wynik na wyjściu powinien być raczej jasny – zapoznaj się z metodami, które można wywołać na pliku

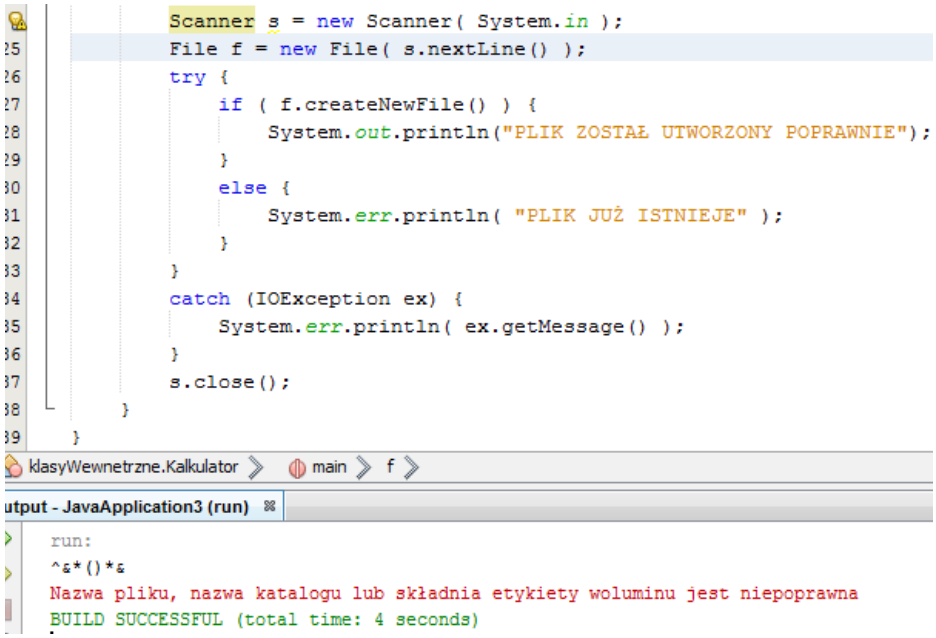
- naprawdę można z nich wyciągnąć mnóstwo informacji. Np. poniższy program przyjmuje ścieżkę do folderu z konsoli i wypisuje jego zawartość:




OK. Napiszemy prosty program, który sprawdzi czy podany plik istnieje i jeśli nie to go utworzy. W zasadzie interesuje nas tylko komenda stworzenia pliku – jest to metoda `createNewFile()` wywołana na obiekcie pliku.




Jest jednak mały problem – metoda `createNewFile()` może podnieść wyjątek wejścia – wyjścia. Za co on odpowiada? Np. za błąd pracy dysku, ale najczęściej za nieprawidłową nazwę pliku. Mój program wygląda tak:



Możliwe, że dziwi Cię brak jakiegokolwiek wywołania `f.exists()`. Nie jest ono potrzebne. Metoda `createNewFile()` zwraca `true` jeśli plik, który ma stworzyć nie istnieje, a `false` w przeciwnym wypadku.

### Czytanie i pisanie do pliku. Strumienie znakowe.

Zaczniemy od omówienia strumieni. Wszystkie operacje IO w Javie obsługuje się tak samo – za pomocą strumieni. Strumienie dzielimy na bajtowe i znakowe. Na razie zajmiemy się znakowymi. Są to klasy kończące się sufiksem `Reader` lub `Writer`. Do pisania i czytania z plików używamy klas `FileWriter`, `FileReader`, `BufferedReader`, `PrintWriter`.

Najprostszy zapis do pliku wygląda tak:

```
22          File f = new File( "bang.txt" );
23          FileWriter out = null;
24          try {
25              out = new FileWriter( f );
26              f.createNewFile();
27              out.write( "HELLO WORLD".toCharArray() );
28          }
29          catch (IOException ex) {
                ex.printStackTrace();
31          }
32          finally {
33              try {
                    out.close();
35              }
36              catch (IOException ex) {
                    ex.printStackTrace();
38              }
39          }
```

Chciałbym zwrócić uwagę na kilka rzeczy. Wywołanie `out.write()` nie powoduje zapisu do pliku, a jedynie zapisanie do cache'a w postaci strumienia. Dopiero wywołanie metody `flush()` na strumieniu pociąga za sobą fizyczne połączenie z odbiornikiem i opróżnienie strumienia. Warto wiedzieć że wywołanie `flush()` zawiera się w wywołaniu `close()`. Między innymi dlatego należy bezwzględnie zamykać strumienie.

**WAŻNE!**

**Każdy strumień musi zostać zamknięty** niezależnie od tego czy w trakcie pracy z nim wystąpiły wyjątki czy nie! Jeśli tego nie zrobimy najprawdopodobniej nic się nie stanie (na pewno strumień nie zostanie przez zamknięciem opróżniony), ale istnieje szansa że otrzymamy błąd `ResourceLeak`.

Zauważ, że metoda `write()` nie potrafi zapisywać całych linijek, a wyłącznie pojedyncze znaki lub tablice znaków.

Ulepszymy teraz nasz program tak, aby umiał zapisywać całe linijki i zrezygnujemy z klauzuli `finally` na rzecz `try-catch with resources`, wtedy strumień zostanie automatycznie zamknięty.

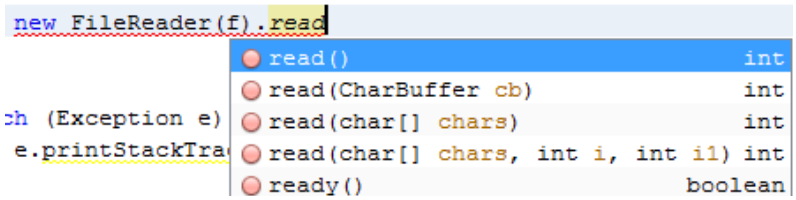


Rysunek 86 - Opakowanie klas strumieniowych

```
24          File f = new File( "bang.txt" );
25
26          try ( PrintWriter out = new PrintWriter( new FileWriter(f) ); ) {
27              f.createNewFile();
28              out.println( "HELLO WORLD" );
29          }
30          catch (IOException ex) {
32              ex.printStackTrace();
            }
```

Chciałbym zwrócić uwagę na linijkę `26`. Specjalnie w ten sposób zadeklarowałem obiekt `PrintWriter`, mimo, że konstruktor jest w stanie przyjąć również argument typu `File`. Chodzi o to, że takie opakowujące deklarowanie jest bardzo charakterystyczne dla strumieni. Dzięki temu opakowaniu dostajemy doskonale znaną nam metodę `println()`, którą teraz możemy zapisywać dane do pliku.

Odczytamy teraz całą zawartość jakiegoś pliku. Najprostszy odczyt wygląda następująco. Podstawową klasą do czytania z pliku jest oczywiście `FileReader`, niestety ma dość mocno ograniczoną funkcjonalność:




Użyjemy zatem charakterystycznego dla strumieni opakowania – do klasy `BufferedReader`. Klasa `BufferedReader` dodatkowo daje nam korzyść w postaci buforowania, dlatego nie każde wywołanie odczytu będzie wiązać się z fizycznym odczytem z dysku, a z bufora, który jest znacznie szybszy.

```
27          try ( BufferedReader in = new BufferedReader( new FileReader(f) ); ) {
```

Nasz odczyt będzie wyglądał tak jak poniżej:

```
25          File f = new File( "lorem.txt" );
26
27          try ( BufferedReader in = new BufferedReader( new FileReader(f) ); ) {
28              String line;
29              while ( (line = in.readLine()) != null ) {
30                  System.out.println( line );
31              }
32          }
33          catch (Exception e) {
35              e.printStackTrace();
            }
```

klasyWewnetrzne.Kalkulator > main > try

Output - JavaApplication3 (run)

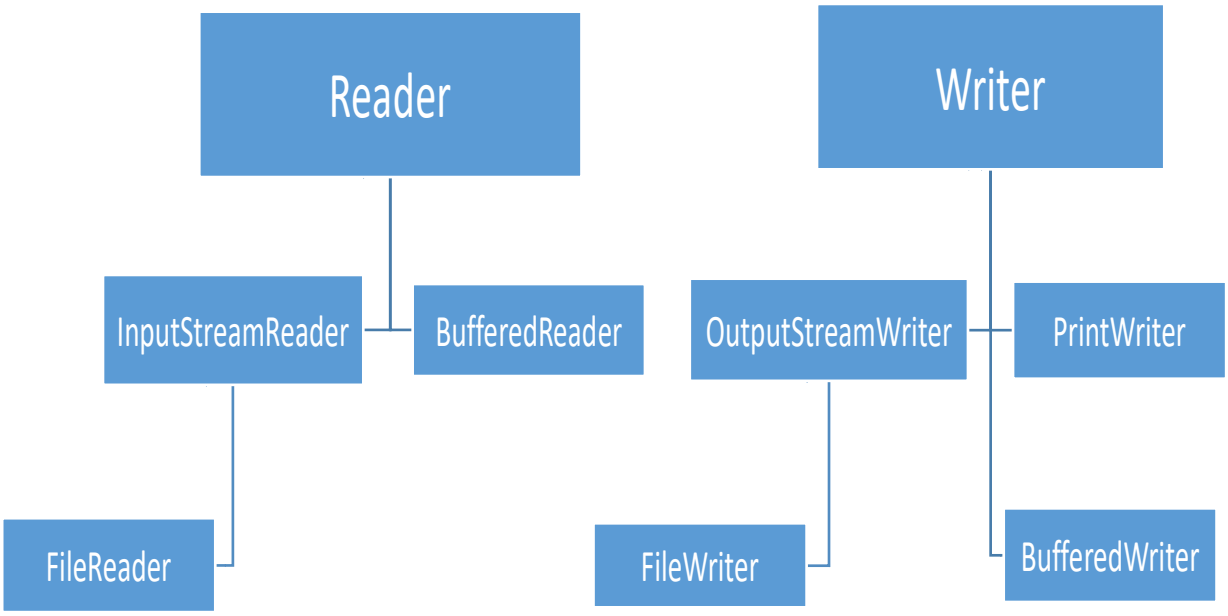
```
run:
Sed ut perspiciatis, unde omnis iste natus error sit voluptatem accusantium doloremque
laudantium, totam rem aperiam eaque ipsa, quae ab illo inventore veritatis et quasi
```

Chcę koniecznie pokazać Ci konstrukcję wewnątrz `while` w linijce `29`, chodzi o `(line = in.readLine())`. Taki konstrukcja jest bardzo sprytna i często wykorzystywana przy obsłudze odczytów ze strumieni. Chodzi o to, że instrukcja przypisania zwraca wartość wpisywaną, dlatego po

zamknięciu jej w nawiasy mogę użyć porównania, a w ciele pętli aktualna linia jest widoczna w zmiennej `line`.

### Czytanie danych w postaci bajtów – dokładniej o strumieniach

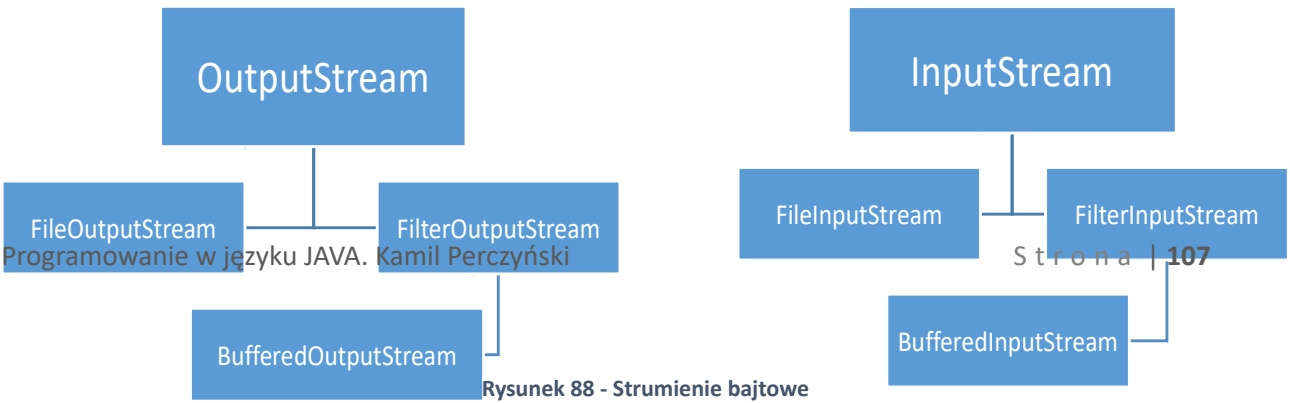
Zaczniemy od drzewka klas dla strumieni znakowych:




Wszystkie te klasy znajdują się w pakiecie `java.io`.

**Rysunek 87 - Strumienie znakowe**

Struktura klas dla strumieni bajtowych:




**Rysunek 88 - Strumienie bajtowe**

O ile do zapisywania treści znakowych używaliśmy `Reader`'ów i `Writer`'ów, o tyle do danych bajtowych użyjemy klas z powyższego drzewka. Ich obsługa znacząco różni się od klas, których używaliśmy ostatnio.

Przykładowy kod kopiujący plik instalacyjny NetBeans IDE ( 190 MB ):

```
28          File toRead = new File( "nb.exe" );
29          File toWrite = new File( "nbCopy.exe" );
30          try (   FileInputStream fis = new FileInputStream( toRead );
31                  FileOutputStream fos = new FileOutputStream( toWrite )
32              )
33          {
34              toWrite.createNewFile();
35              byte[] data = new byte[ fis.available() ];
36              fis.read( data );
37              fos.write(data);
38          }
39          catch (FileNotFoundException ex) {
                ex.printStackTrace();
41          }
42          catch (IOException e) {
                e.printStackTrace();
44          }
```

Do konstruktorów strumieni wyjścia i wejścia podaję odpowiednie pliki. Następnie tworzę tablicę bajtów w której przechowam dane z pliku `toRead`. Metoda `available()` zwraca ilość bajtów znajdujących się w strumieniu. Do linijki `35` nie wykonałem jeszcze żadnego odczytu, więc metoda `available()` zwróci rozmiar pliku. Tablica bajtów `data` to miejsce gdzie przechowam całą zawartość pliku `toRead` pomiędzy odczytem, a zapisaniem danych.

Metoda `read( byte[] )` wpisuje do podanej jako argument tablicy tyle bajtów ile pomieści tablica. Analogicznie metoda `write( byte[] )` strumienia wyjściowego zapisuje wszystkie bajty z podanej tablicy do pliku, podanego w konstruktorze strumienia. Dla dowodu, że powyższy kod działa ☺:

| | | | |
|---|---|---|---|
| manifest.mf | 2015-07-19 12:41 | Plik MF | 1 KB |
| nb.exe | 2015-07-14 22:26 | Aplikacja | 189 449 KB |
| nbCopy.exe | 2015-07-27 14:46 | Aplikacja | 189 449 KB |

Dla bardziej zaawansowanych operacji na większych plikach warto zapoznać się z metodami statycznymi z klasy `Files`.

### Serializacja czyli zapisywanie obiektów do plików

Java udostępnia mechanizm serializacji obiektów – czyli przekonwertowanie ich struktury do bajtów i zapisanie ich do pliku. Jest jeden warunek – aby móc serializować obiekty danej klasy, musi ona implementować interfejs `Serializable`. Jest on pusty – nie ma w nim żadnych metod.

```
    public class Samochod implements Serializable {
```

Teraz cały obiekt będzie mógł zostać zapisany do pliku. Przyjrzyjmy się poniższemu, niewielkiemu programowi:

```
List<Samochod> lista = new ArrayList<Samochod>();
File f = new File( "objects.o" );
try {
    f.createNewFile();
} catch (IOException ex) {
    return;
}
Scanner s = new Scanner( System.in );
String cmd ;
while ( !(cmd = s.nextLine()).equals("exit") ) {
    if (cmd.equals("ls")) {
        for ( Samochod i: lista ) {
            System.out.println( i );
        }
    }
    else if ( cmd.equals("read") ) {
        try (ObjectInputStream ois = new ObjectInputStream( new FileInputStream(f) ); ) {
            lista =  (List<Samochod>) ois.readObject();
        }
        catch (IOException ex) {
            ex.printStackTrace();
        } catch (ClassNotFoundException ex) {
            ex.printStackTrace();
        }
    }
    else if ( cmd.equals("write") ) {
        try (ObjectOutputStream oos = new ObjectOutputStream( new FileOutputStream(f) ); ) {
            oos.writeObject( lista );
        } catch (IOException ex) {
            ex.printStackTrace();
        }
    }
    else {
        try {
            lista.add( new Samochod( Double.parseDouble(cmd) ) );
        }
        catch (NumberFormatException e) {
            System.err.println("NIEPRAWIDŁOWE DANE");
        }
    }
}
s.close();
```

Rysunek 89 - Przykład serializacji obiektów

output – pierwsze wykonanie. Dodaję dwa samochody i zapisuję je do pliku:

```
run:
123
321
ls
Samochod{ladownosc=123.0}
Samochod{ladownosc=321.0}
write
exit
BUILD SUCCESSFUL (total time: 13 seconds)
```

Teraz drugie wykonanie. Odczyt z pliku i wyświetlenie wyników:

```
run:
ls
read
ls
Samochod{ladownosc=123.0}
Samochod{ladownosc=321.0}
exit
BUILD SUCCESSFUL (total time: 11 seconds)
```

Powyższe operacje są możliwe dzięki klasom `ObjectInputStream` i `ObjectOutputStream`, które są zwykłymi strumieniami uzupełnionymi o metody `readObject()` i `writeObject()`.

### Odczyt większych plików

Wg wielu ludzi Java jest wolna, ale najczęściej powodem wolnego jej działania jest problem na poziomie kodu, a nie samej Javy. Pokażę teraz program ujawniający pewien szczegół związany z używaniem klasy `String`, czytaniem plików i optymalizacją programów.

Poniższy, prosty program wczytuje całą zawartość pliku do pamięci, a następnie wyświetla ją na ekran. Jeden szczegół – plik ma 30 MB – to ponad 30 milionów znaków.

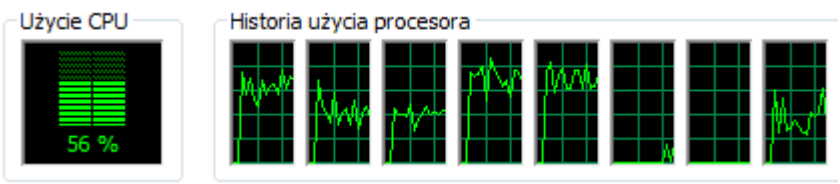
```
33          File f = new File( "bulk.txt" );
34          System.out.println( "Plik " + f.getPath() + " - " + (f.length()/(1024.0*1024)) + "MB" );
35          String zawartoscPliku = "";
36          try ( BufferedReader in = new BufferedReader( new FileReader(f) ); ) {
37              String line;
38              while ( (line = in.readLine()) != null ) {
39                  zawartoscPliku += line;
40              }
41              System.out.println( zawartoscPliku );
42          }
43          catch (Exception e) {
45              e.printStackTrace();
            }
```

klasyWewnetrzne.Kalkulator › main

```
Output - JavaApplication3 (run)
run:
Plik bulk.txt - 29.56680965423584MB
BUILD STOPPED (total time: 10 seconds)
```

Jak widać w ciągu dziesięciu sekund Java nie była w stanie wyświetlić zawartości pliku. Czy to znaczy, że nie jest w stanie poradzić sobie z załadowaniem 30 MB do pamięci?

Jest! Niestety w powyższym programie jest dość poważny błąd. Problem znajduje się w linijce `39`. Hmm… Przecież to zwykła konkatenacja tekstu – co może być w tym problematycznego? Otóż klasa `String` została tak napisana, że używa jej się tak samo jak typów nieobiektowych – tzn. każda operacja na obiekcie `String` tworzy obiekt. Plik bulk.txt ma ponad 390 000 linii, a więc w trakcie pętli dla każdej linii powstaje nowy obiekt. Potwierdza to obciążenie procesora:



Jak wiemy na poziomie C, C++ wszystkie napisy są reprezentowane przez tablice znaków. Rozwiązaniem naszego problemu jest dobranie się do tablicy znaków, którą zawiera w sobie `String` i modyfikację jej bez tworzenia nowego obiektu. Robimy to używając klasy `StringBuilder` (lub jej starszego brata – `StringBuffer`).

```
33          File f = new File( "bulk.txt" );
34          System.out.println( "Plik " + f.getPath() + " - " + (f.length()/(1024.0*1024)) + "MB" );
35          try ( BufferedReader in = new BufferedReader( new FileReader(f) ); ) {
36              String line;
37              StringBuilder sbs = new StringBuilder();
38              while ( (line = in.readLine()) != null ) {
39                  sbs.append(line).append("\n");
40              }
41              System.out.println( sbs.toString() );
42          }
43          catch (Exception e) {
45              e.printStackTrace();
            }
```

Rysunek 90 - Użycie klasy StringBuilder

W klasach `StringBuilder` i `StringBuffer` zdefiniowano metodę `append()`, która modyfikuje zawartość `String`'a bez tworzenia nowego obiektu. Po tej drobnej modyfikacji wyniki dostaję od razu:

```
Choro possit te usu, te vel nibh doctus persius, quis latine vim ut. Eu est iusto altera nominati, docendi ancillae et quo. Cum id splendide constituto, fac
constituto no nec, laboramus posidonium cu vix. Qui choro quando indoctum in. Elit detracto eum at.


Et errem perfecto pro, nulla platonem vim ex. Mel cu agam pericula, pri partem quaeque eu. Cum ad detracto quaerendum consequuntur, no recteque sapientem iu
 Ad reque solet referrentur mei, cu duo quem laboramus. Cu sea commodo apeirian honestatis, id duis everti ius.
 Te vim minim repudiandae, qui soleat labores ne. Putant philosophia vel ad, vim veniam corpora ex, eam homero dolorum legendos ut.
 Lorem ipsum dolor sit amet, vis meis dolor volutpat id. Augue omittam eos ne, usu impetus alienum definitiones eu, magna graeci mel ei. Ut sale dicunt phil
i, pro ad exerci altera moderatius. Eu cum suas vide. Est iudico invenire volutpat ut, cum cetero latine appellantur in.
i, pro ad exerci altera moderatius. Eu cum suas vide. Est iudico invenire volutpat ut, cum cetero latine appellantur in.BUILD STOPPED (total time: 2 seconds
```

## Foldery w Javie

Za obsługę folderów w Javie również odpowiada klasa `File`. Do tworzenia nowych folderów stworzono dwie metody:

- `mkdir()`
- `mdirs()`

Ich użycie jest bardzo proste, ponieważ te metody nie podnoszą żadnych wyjątków, a jedynie zwracają wartość `boolean`, która określa czy utworzono folder, czy nie.

Użycie `mkdir()` od `mkdirs()` różni się jedynie tym, że `mkdir()` może utworzyć tylko jeden folder. `mkdirs()` potrafi natomiast utworzyć całe drzewko folderów jednym wywołaniem.



Rysunek 91 - Tworzenie folderów z poziomu programu

```
49      public static void main(String[] args) {
50
51          File folder = new File( "nowyFolder" );
52
53          File foldery = new File( "1/2/3/4/5/6/6/7/8/" );
54
55          if ( folder.mkdir() ) {
56              System.out.println( "## UTWORZYŁEM FOLDER " + folder );
57          }
58
59          if ( !foldery.mkdir() ) {
60              System.out.println( "## NIE UTWORZYŁEM FOLDERÓW - nie ta metoda" );
61          }
62          if ( foldery.mkdirs() ) {
63              System.out.println( "## UTWORZYŁEM MNÓSTWO FOLDERÓW JEDNĄ KOMENDĄ" );
64          }
```

pl.jsystems.materialyJava.utils.DatabaseConnector > initConnection >

Output - MVCApp (run)

```
run:
## UTWORZYŁEM FOLDER nowyFolder
## NIE UTWORZYŁEM FOLDERÓW - nie ta metoda
## UTWORZYŁEM MNÓSTWO FOLDERÓW JEDNĄ KOMENDĄ
BUILD SUCCESSFUL (total time: 0 seconds)
```

# Wielowątkowość

## Pojęcie wątku

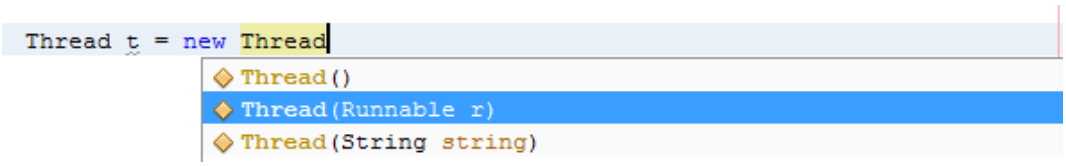
Wszystko co napisaliśmy teraz, wykonywało się liniowo – tzn. linijka po linijce. To z kolei znaczy, że nasze programy były w stanie robić tylko jedną rzecz jednocześnie. W rzeczywistości komputer potrafi robić sto, tysiąc rzeczy jednocześnie. Wątkiem nazywamy proces wydzielony przez program, który wykonuje się niezależnie od reszty programu (wątku głównego).


**Rysunek 92 - Ilość wątków - screen z monitora zasobów w Windows 7**

Czyli przy programowaniu współbieżnym (wielowątkowym) linijka `20` wcale nie musi wykonać się po linijce `19` ☺. W zamian oferują wielozadaniowość aplikacji. Pamiętasz przykład z kopiowaniem plików? Jeśli kopiowanie pliku trwałoby dziesięć minut, musielibyśmy poczekać na zakończenie kopiowania, aby wydać kolejną komendę. Wydzielenie samego przekopiowania do oddzielnego wątku pozwoliłoby użytkownikowi na ciągłe wydawanie komend – natomiast samo kopiowanie odbywałoby się w tle.

## Klasa Thread i interfejs Runnable

Wątki tworzymy tak jak obiekty innych klas – za pomocą słowa kluczowego `new`.

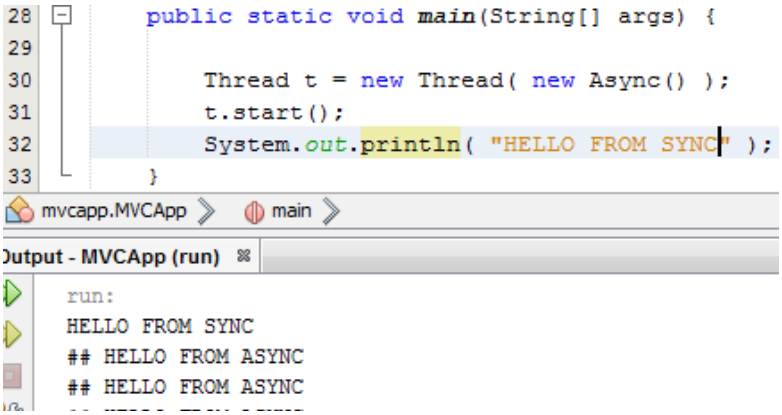



**Rysunek 93 - Tworzenie nowego wątku**

Klasa `Thread` odpowiada za asynchroniczne wykonywanie zadań i do swojego konstruktora przyjmuje obiekt typu `Runnable`. `Thread` można porównać do pracownika, któremu delegujemy zadanie – reprezentowane przez obiekt typu `Runnable`. Stworzę zatem zadanie polegające na 10-krotnym wyświetleniu napisu „Hello From Async". Wewnątrz klasy w której obecnie się znajduję tworzę prywatną statyczną klasę wewnętrzną implementującą interfejs `Runnable`.

```
16      private static class Async implements Runnable {
17
18          @Override
19          public void run() {
20              for (int i = 0; i < 10; i++) {
21                  System.out.println( "## HELLO FROM ASYNC" );
22              }
23          }
24      }
```

Kod metody `main` tworzącej i uruchamiającej wątek:



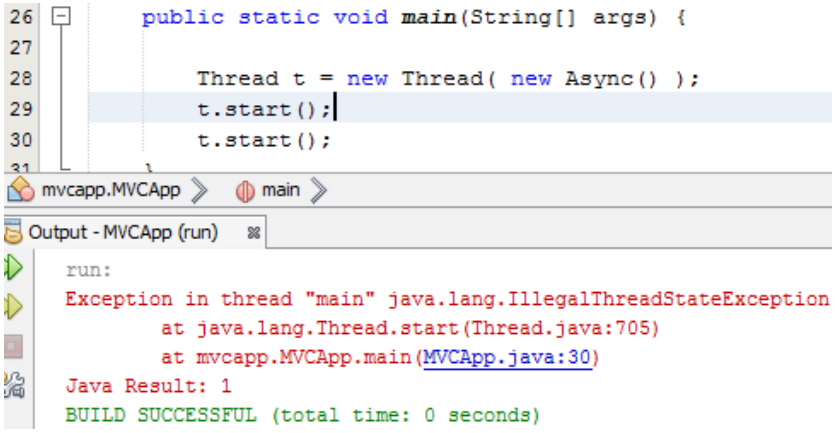

Rysunek 94 - Wykonanie asynchroniczne

Wątki uruchamiamy metodą **`start()`**. Jest też myląca metoda `run()`, która po prostu uruchamia ciało metody `run()` znajdującej się w obiekcie typu `Runnable`, który podawaliśmy jako argument.

Koniecznie przyjrzyj się wynikom wykonania programu. Czyżby linijka `32` wykonała się przed `31`? Poniekąd. Uruchomienie wątku nastąpiło oczywiście przed wypisaniem „HELLO FROM SYNC" na ekran, ale program przystąpił do wykonania ciała wątku później. Czy tak będzie zawsze? Zdecydowanie nie! Wypisanie „HELLO FROM SYNC" może nastąpić w dowolnym miejscu ponieważ pętla wykonuje się całkowicie niezależnie.

**WAŻNE!**

Jeden wątek może być wykonany tylko raz. Jeśli spróbujemy wywołać na jednym wątku `start()` drugi raz otrzymamy wyjątek `IllegalThreadStateException`.




### Przerywanie wykonania wątku

Niekiedy potrzebujemy możliwości przerwania jednego wątku z poziomu innego – np. wątku głównego. Możemy na wątku wywołać metodę `stop()` – oznaczoną jako `Deprecated`, oraz metodę `interrupt()`. Generalną zasadą jest, że unikamy przerywania jednego wątku z poziomu innego. Jeśli już musimy to zrobić używamy metody **`interrupt()`**, nigdy `stop()`.

Metoda stop() zabija wątek natychmiastowo – *by-force*, natomiast `interrupt()` powoduje, że w wątku pojawia się wyjątek `InterruptedException`, co z nim zrobimy – to już nasza sprawa.

Dlaczego `interrupt()`? Przypuśćmy że wydelegowaliśmy do oddzielnego wątku realizację jakiejś transakcji. Transakcja w rozumieniu informatycznym jest zestawem operacji gdzie dane są w stabilnym stanie przed i po jej wykonaniu – niekoniecznie w trakcie. Wątek może korzystać ze strumieni, które zawsze będzie musiał zamknąć etc. Chyba nie potrzeba więcej tłumaczenia do czego mogłoby doprowadzić przerwanie wykonania wątku w połowie pracy.

Jak zatem obsłużyć zakłócenie wykonania wątku metodą `interrupt()`? Możemy zrobić to na dwa sposoby. Tak jak powiedziałem w wątku pojawia się wyjątek – czyli wszystkie metody zdolne do podniesienia `InterruptedException` podniosą go:

Klasa `Async`:

```
16      private static class Async implements Runnable {
17
18          @Override
19          public void run() {
20              try {
21                  while (true) {
22                      Thread.sleep(0);
23                  }
24              }
25              catch (InterruptedException e) {
26                  e.printStackTrace();
27              }
28
29          }
30      }
```

Metoda `Thread.sleep()` wstrzymuje wykonanie wątku na określoną ilość milisekund i może podnosić `InterruptedException`.

Wywołanie i output:

```
36              Thread t = new Thread( new Async() );
37              t.start();
38              t.interrupt();
39          }
```

mvcapp.MVCApp › main

Output - MVCApp (run)

```
run:
java.lang.InterruptedException: sleep interrupted
        at java.lang.Thread.sleep(Native Method)
        at mvcapp.MVCApp$Async.run(MVCApp.java:22)
        at java.lang.Thread.run(Thread.java:745)
BUILD SUCCESSFUL (total time: 0 seconds)
```

Wszystko działa ☺. Nie jest to jednak dobra metoda. Problem stanowią dwie rzeczy. Po pierwsze, wyjątek `InterruptedException` podnoszą dwie metody – jedną już poznaliśmy, druga to metoda `wait()` wywoływana na dowolnym obiekcie (nie będziemy jej omawiać). Po drugie, w takim układzie wyjątek można przechwycić tylko w momencie wykonywania `Thread.sleep()`.

Często jednak wykonujemy pewne części transakcji i dopiero po ich zakończeniu będziemy sprawdzać czy możemy iść dalej, czy przerywamy wykonanie. Służy do tego metoda `isInterrupted()` wykonywana na wątku. Używamy jej w następujący sposób.

```
20          public void run() {
21              while (true) {
22                  if ( Thread.currentThread().isInterrupted() ) {
23                      System.out.println( "## Kończę wykonanie" );
24                      return;
25                  }
26              }
```

Metoda statyczna `currentThread()` zwraca obecnie wykonywany wątek. Następnie wystarczy sprawdzić czy inny wątek nie podjął decyzji o zakłóceniu tego, w którym wykonano `isInterrupted()`. Przy takim samym wywołaniu uzyskuję poniższy output:

```
mvcapp.MVCApp > Async >
Output - MVCApp (run)
run:
## Kończę wykonanie
BUILD SUCCESSFUL (total time: 0 seconds)
```

Należy uważać na bardzo podobną metodę `interrupted()`, która działa inaczej. Mianowicie jeśli wątek został zakłócony i dwa razy wywołamy metodę `interruped()` zwróci ona `true` tylko przy pierwszym wywołaniu. Możemy ją wywołać również bezpośrednio na klasie `Thread`.

Metoda `run()`:

```
@Override
public void run() {
    System.out.println( Thread.currentThread().interrupted() );
    System.out.println( Thread.currentThread().interrupted() );
    System.out.println( Thread.currentThread().interrupted() );
}
```

Oraz output przy takim samym wywołaniu jak poprzednio:

```
Output - MVCApp (run)
run:
true
false
false
BUILD SUCCESSFUL (total time: 0 seconds)
```

### Problem współdzielenia zasobów

Przyjrzyjmy się poniższemu prostemu przykładowi. Zadeklarowałem dwa wątki, które operują niezależnie od siebie na jednej zmiennej, czyli współdzielą zasób. Jeden z nich dziesięć tysięcy razy

dodaje do zmiennej jeden, a drugi dziesięć tysięcy razy odejmuje jeden. Ciekawe jaki będzie stan zmiennej po zakończeniu obu wątków.

```
16      private static class Incrementer implements Runnable{
17
18          @Override
19          public void run() {
20              for (int i = 0; i < 10000; i++) {
21                  var++;
22              }
23              System.out.println( "Kończę z wynikiem: " + var );
24          }
25      }
26
27      private static class Decrementer implements Runnable {
28
29          @Override
30          public void run() {
31              for (int i = 0; i < 10000; i++) {
32                  var--;
33              }
34              System.out.println( "Kończę z wynikiem: " + var );
35          }
36      }
37      private static Integer var = 0;
```

Wydawałoby się, że jeden wątek zawsze skończy z wynikiem `0`. Nic bardziej mylnego:

```
run:
Kończę z wynikiem: 4629
Kończę z wynikiem: -1694
BUILD SUCCESSFUL (total time: 0 seconds)
```

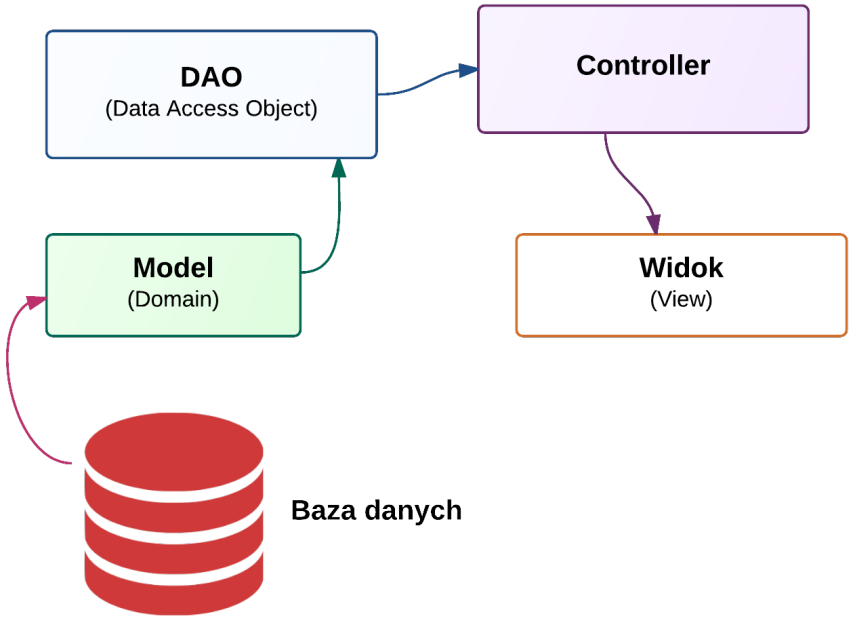
Z czego wynika powyższy output? Dlaczego $0+10000-10000\neq 0$ ? Wynika to m. in. z nieatomowości operacji realizowanych przez te wątki. Aby output był prawidłowy należy te wątki skoordynować (to jest termin techniczny) używając metod `wait()` i `notify()`. Jednak ich użycie znacząco wykracza poza zakres tych materiałów.

Jeśli jednak chcesz zgłębić temat współbieżności, zapoznaj się z pojęciem synchronizacji, zmiennymi współdzielonymi (ulotnymi), metodami `wait()`, `notify()`, zamkami obiektowymi, semaforami oraz kolekcjami z pakietu `java.concurrent` – np. `ArrayBlockingQueue`.

# Obsługa relacyjnych baz danych – JDBC

## Model warstwowy aplikacji

Wg wzorca projektowego MVC dzielimy części aplikacji na warstwy mające swoje własne odpowiedzialności (komunikacja z użytkownikiem, bazą danych etc.). Pojawiają nam się poniższe warstwy aplikacyjne:




Rysunek 95 - Wzorzec projektowy DAO

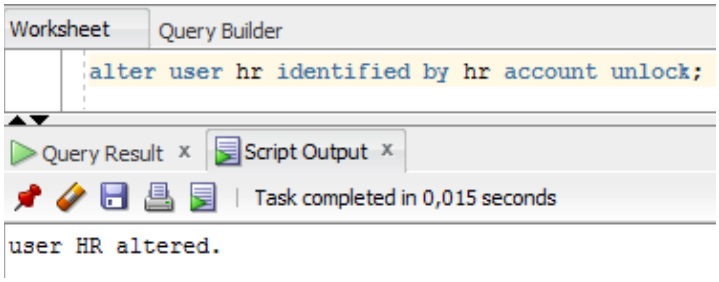
Warstwą najniższą jest oczywiście baza danych, po niej jest warstwa modelowa, czyli obiektowe odwzorowanie tabel bazodanowych. Nie zawieramy w modelu żadnej logiki – jest to tylko reklamówka na dane. Klasa modelu składa się z prywatnych pól, publicznych akcesorów i niczego poza tym! Nie piszemy w nich konstruktorów (Ewentualnie za pomocą klas wewnętrznych – nazywaliśmy je `Builder`'ami).

Kolejną warstwą jest **`DAO`** (Data Access Object). Jej odpowiedzialnością jest rozmowa z bazą danych i tworzenie obiektów modelu, które zostaną wykorzystane na wyższych warstwach aplikacji. Zawiera także metody do utrwalania obiektów w bazie danych.

**`Controller`** to warstwa odpowiedzialna za odebranie danych od użytkownika, parsowanie ich, oraz wprowadzenie informacji do bazy danych. W **`controller`**'ach znajduje się rzeczywista logika aplikacji.

Widok to klasa, która odpowiada wyłącznie za prezentację danych. W przypadku aplikacji desktop'owych `Controller` i `Widok` będzie jedną klasą.

Będę oczywiście potrzebował jakiejś bazy danych. Dla celów poniższych materiałów posłużę się bazą `Oracle 11g Express`. Wykorzystam wbudowany w niej schemat HR, który możemy odblokować uruchamiając na koncie `SYS` poniższą komendę:



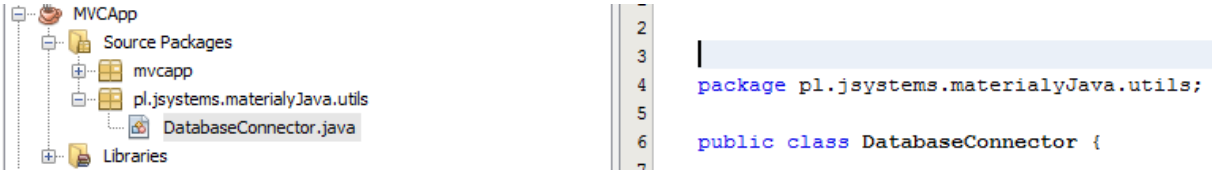

**Rysunek 96 - Odblokowanie konta HR w bazie Oracle**

OK. Teraz mogę łączyć się z bazą za pomocą SQLDeveloper'a, ale z poziomu programu jeszcze nie. Aby móc nawiązać połączenie z bazą danych w Javie, czyli uzyskać instancję interfejsu `java.sql.Connection`.

Zakładam sobie pakiet:

```
pl.jsystems.materialyJava.utils
```

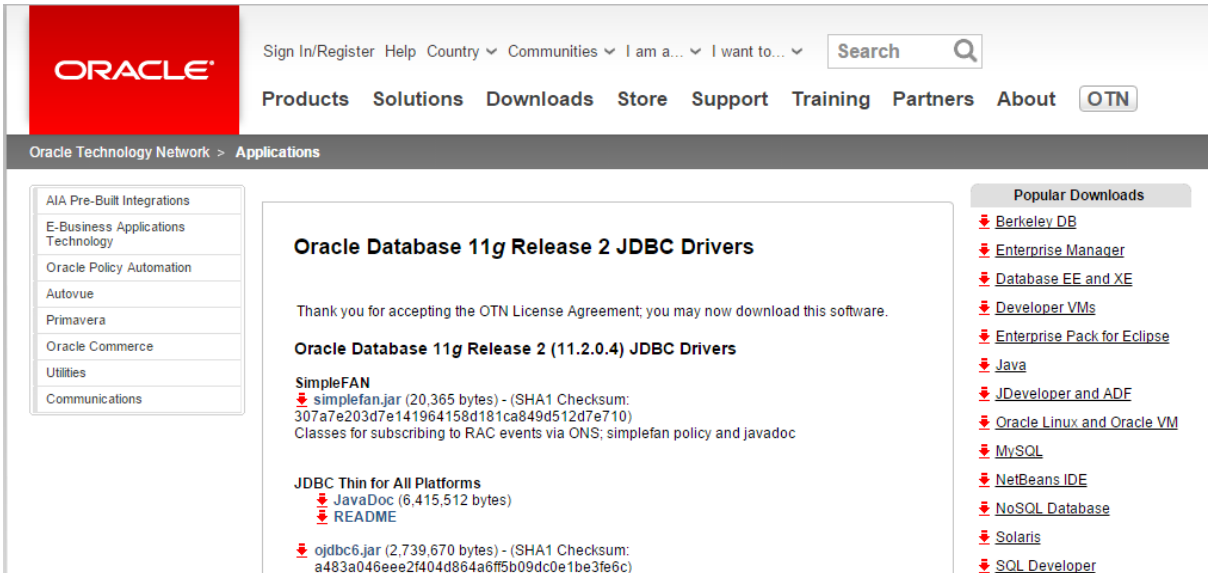
A w nim klasę `DatabaseConnector`.



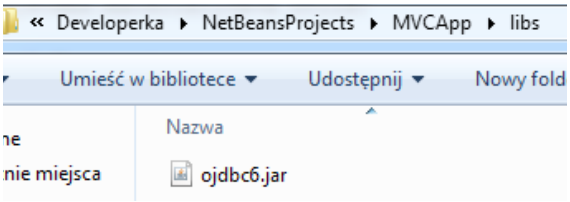

Obiekt połączenia z bazą danych będzie oczywiście `Singleton`'em. Dlatego doprowadzam klasę do poniższego stanu:

```
public class DatabaseConnector {

    private static Connection connection;

    public static Connection getConnection() {
        if ( connection == null ) {
            connection = initConnection();
        }
        return connection;
    }
    /**
     * Metoda nawiązująca połączenie
     * @return
     */
    private static Connection initConnection() {

        return null;
    }

}
```

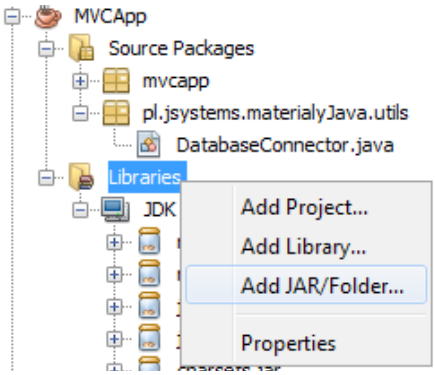
Zajmiemy się teraz ciałem metody `initConnection()`. Pierwsza rzecz, której będziemy potrzebować to `sterownik JDBC` dla bazy danych Oracle. Jest to zwykła biblioteka zewnętrzna umożliwiająca nam łączenie się z bazą danych. Plik nazywa się dokładnie `ojdbc6.jar` i można pobrać go z oficjalnej strony firmy Oracle.




Po ściągnięciu pliku, umieszczamy go w naszym projekcie w folderze `libs` (trzeba będzie go utworzyć).




Pozostaje już tylko dodać bibliotekę `ojdbc6` do projektu. Robimy to klikając na sekcji `Libraries` projektu w NetBeans.




Wystarczy, że wskażemy plik biblioteki, a zostanie dołączona do naszego projektu – będziemy zatem mieć dostęp do nowych klas odpowiedzialnych za obsługę baz danych.

Nawiązanie połączenia z bazą przebiega wg dość prostego schematu. Najpierw musimy sprawdzić czy posiadamy zaimportowany sterownik JDBC, później podajemy wszystkie dane do połączenia i jeśli w trakcie nawiązywania połączenia nie zostanie podniesiony żaden wyjątek – wtedy możemy bezproblemowo zwrócić obiekt typu `Connection`.

Na początek sprawdzam, czy w aplikacji jest sterownik Oracle JDBC:

```
29      private static Connection initConnection() {
30          try {
31              Class.forName("oracle.jdbc.driver.OracleDriver");
32          }
33          catch (ClassNotFoundException ex) {
34              System.err.println("## BRAK STEROWNIKA JDBC");
35          }
36
37          return null;
38      }
```

Następnie nawiązuje już połączenie z bazą danych. Potrzebne do tego są trzy informacje: URL bazy danych (charakterystyczny dla niej), nazwa użytkownika i hasło. Połączenie zwróci mi metoda statyczna z klasy `DriverManager`.

```
30      private static Connection initConnection() {
31          try {
32              Class.forName("oracle.jdbc.driver.OracleDriver");
33
34              String url = "jdbc:oracle:thin:@localhost:1521:xe";
35              String username = "hr";
36              String passwd = "hr";
37              return DriverManager.getConnection(url, username, passwd);
38          }
39          catch (ClassNotFoundException ex) {
40              System.err.println("## BRAK STEROWNIKA JDBC");
41          }
42          catch (SQLException ex) {
43              ex.printStackTrace();
44          }
45
46          return null;
47      }
```

Rysunek 97 - Standardowe łączenie się z bazą Oracle

Zauważ, że pojawiła kolejna klauzula `catch` obsługująca `SQLException`. Jest to uniwersalna pośród sterowników bazodanowych klasa reprezentująca wszystkie wyjątki związane z bazą danych – w tym np. błąd autoryzacji. Oczywiście istnieją też wyjątki bardziej szczegółowe np. `ConstraintIntegrityViolationException`, które jednak dziedziczą po klasie `SQLException`.

Istnieje też inna metoda łączenia się z bazą. Zamiast używać `DriverManager`'a możemy ręcznie utworzyć obiekt sterownika i podać mu URL bazy danych oraz kolekcję `Properties` z właściwościami `user` i `password`.



Rysunek 98 - Inna forma łączenia się z bazą danych

Przy tym podejściu nie musimy programowo sprawdzać, czy mamy sterownik JDBC, ponieważ jeśli go nie

```
30      private static Connection initConnection() {
31          Driver d = new oracle.jdbc.driver.OracleDriver();
32          Properties props = new Properties();
33
34          props.setProperty("user", "hr");
35          props.setProperty("password", "hr");
36
37          try {
38              return d.connect("jdbc:oracle:thin:@localhost:1521:xe", props);
39          }
40          catch (SQLException ex) {
41              ex.printStackTrace();
42          }
43          return null;
44      }
```

będzie powyższy kod się nie skompiluje.

## Obsługa SELECT'ów – dwie pierwsze warstwy aplikacji.

Napiszemy teraz kilka `SELECT`'ów wykonywanych z poziomu aplikacji. Zmapujemy encję `regions` ze schematu `HR`. Zaczniemy od klasy modelowej odwzorowującej pojedynczy region. Potrzebny będzie oddzielny pakiet:

`pl.jsystems.materialyJava.domain`

Tworzę w nim klasę `Region`.

```
6   package pl.jsystems.materialyJava.domain;
7
8   public class Region {
9
10      private Long regionID;
11      private String regionName;
12
13      public Long getRegionID() {
14          return regionID;
15      }
16
17      public void setRegionID(Long regionID) {
18          this.regionID = regionID;
19      }
20
21      public String getRegionName() {
```

**Rysunek 99 - Standardowa klasa modelowa**

Tak jak powiedziałem, przy bazach danych należy używać tylko typów obiektowych – po to aby móc wpisać w nie wartość `null`. Ponadto, jeśli używamy w tabeli klucza głównego będącego liczbą, lepiej nie używać typu `Integer`, ponieważ wartości sekwencji są w stanie przekroczyć jego zakres.

Następnie tworzę kolejny pakiet:

`pl.jsystems.materialyJava.dao`

A w nim klasę `RegionsDao`:

```
6   package pl.jsystems.materialyJava.dao;
7
8   public class RegionsDao {
9
0   }
```

Napiszemy w niej metodę zwracającą wszystkie regiony z bazy danych w postaci listy. Zaczniemy od prostego szkieletu:

```
14  public List<Region> getAll() {
        List<Region> result = new ArrayList<Region>();
16
17      return result;
18  }
```

Nie stało się jeszcze nic szczególnego. Musimy jeszcze wyciągnąć dane z bazy i przetłumaczyć je na obiekty klasy `Region`.

Schemat odczytu wygląda następująco. Musimy mieć obiekt typu `Statement` (interfejs `java.sql.Statement`). Dopiero on będzie w stanie wykonać `SELECT`'a na bazie danych zwracając nam wyniki jako obiekt typu `ResultSet` (interfejs `java.sql.ResultSet`). Obiekty klasy `Region` będziemy tworzyć iterując po zbiorze wyników.

```
18  public List<Region> getAll() {
        List<Region> result = new ArrayList<Region>();
20      try {
            Statement s = DatabaseConnector.getConnection().createStatement();
22          String sql = "select * from regions";
23          ResultSet rows = s.executeQuery(sql);
24          while ( rows.next() ) {
25              Region r = new Region();
26              r.setRegionID( rows.getLong("region_id") );
27              r.setRegionName( rows.getString("region_name") );
28              result.add( r );
29          }
30          rows.close();
31          s.close();
32      }
33      catch (SQLException ex) {
            ex.printStackTrace();
35      }
36      return result;
37  }
```

Jak widać, obiekt `Statement` uzyskujemy z połączenia z bazą danych. `ResultSet` to zbiór wyników. Wywołując na nim metodę `next()` przesuwamy się niejako po kolejnych wierszach. Jednocześnie metoda `next()` zwraca wartość `boolean`, która określa czy w zbiorze wyników znajdują się jeszcze jakieś wiersze.

Każdy kolejny wiersz ma w sobie dane na podstawie których mogę stworzyć obiekt. Wartości poszczególnych kolumn wyciągam wywołując metody `getString()`, `getInteger()` etc. Jako argument mogę podać nazwę lub indeks kolumny (**indeksy w bazach danych liczone są od 1!**). Warto wiedzieć, że typ bazodanowy nie ma nic wspólnego z typem programowym. Mogę w kolumnie typu `VARCHAR2` trzymać liczby i na poziomie aplikacji wyjmować je jako typ `Long`.

Teraz, gdy mam już stworzoną odpowiednią metodę w klasie `DAO`, odczyt danych z bazy sprowadza się do prostej pętli `for-each`:

```
19          RegionsDao rd = new RegionsDao();
20
            for ( Region r: rd.getAll() ) {
22              System.out.println( r.getRegionID() + ". " + r.getRegionName() );
23          }
```

mvcapp.MVCApp › main

Output - MVCApp (run)

```
run:
1. Europe
2. Americas
3. Asia
4. Middle East and Africa
BUILD SUCCESSFUL (total time: 0 seconds)
```

### Zapytania preparowane – PreparedStatement

Zajmiemy się teraz wprowadzaniem i modyfikowaniem danych w bazie – czyli komendami `DML`. Analogicznie przygotowujemy sobie metodę w klasie `DAO` – `dodajRegion()`. Zastanów się jakie argumenty przyjmować będzie powyższa metoda – nazwę regionu? Można i tak, ale po co? Mamy przecież klasę modelową `Region`, która potrafi przechować w sobie wszystkie informacje o `Region`ie.

Deklaracja metody będzie wyglądać zatem tak:

```
42      public void dodajRegion( Region r ) {
43
44      }
```

Wpierw utworzę w schemacie sekwencje o nazwie `universal` tak aby zaczęła liczyć od `300`, a jej wartości wzrastały od `1`.

```
1.  create sequence universal start with 300 increment by 1;
```

Pozostaje mi tylko na poziomie programu napisać odpowiednią komendę `SQL` i wstrzyknąć do niej wartości.

```
44          String sql = "insert into regions (region_id, region_name) "
45                  + "values (universal.nextval, '" + r.getRegionName() + "')";
```

Aby uruchomić tę metodę wystarczy, że stworzę obiekt typu `Statement` i wywołam na nim metodę `executeUpdate()`.

Od razu mówię, że nie jest to prawidłowy sposób wprowadzania danych do bazy! ☺

```
42      public void dodajRegion( Region r ) {
43
44          String sql = "insert into regions (region_id, region_name) "
45                  + "values (universal.nextval, '" + r.getRegionName() + "')";
46          System.out.println( sql );
47          try {
49              Statement s = DatabaseConnector.getConnection().createStatement();
50              s.executeUpdate(sql);
51              s.close();
52          }
54          catch (Exception e) {
55              e.printStackTrace();
56          }
57
58      }
```

Powyższy kod nie jest jednak poprawny. Wszystko będzie działać dopóki nazwa regionu nie będzie zawierała apostrofu.

Spójrzmy na poniższy przykład:

```
50          RegionsDao rd = new RegionsDao();
51
52          Region r = new Region();
53          r.setRegionName( "Russia" );
54
55          Region r2 = new Region();
56          r2.setRegionName( "C'thulu" );
57
58          rd.dodajRegion(r);
59          rd.dodajRegion(r2);
60
61      }
```

pl.jsystems.materialyJava.utils.DatabaseConnector > main >

Output - MVCApp (run)

```
run:
insert into regions (region_id, region_name) values (universal.nextval, 'Russia')
insert into regions (region_id, region_name) values (universal.nextval, 'C'thulu')
java.sql.SQLSyntaxErrorException: ORA-00917: missing comma

        at oracle.jdbc.driver.T4CTTIoer.processError(T4CTTIoer.java:447)
        at oracle.jdbc.driver.T4CTTIoer.processError(T4CTTIoer.java:396)
```

Oczywiście możemy zaszyć gdzieś w kodzie zamianę apostrofu na dwa apostrofy i wtedy znów wszystko zacznie działać, ale nie tędy droga.

Rozwiązaniem są tzw. zapytania preparowane czyli typ `PreparedStatement`. Jest to interfejs dziedziczący po `Statement`.

```
9   public interface PreparedStatement extends Statement {
10
11      public ResultSet executeQuery() throws SQLException;
```

Jego użycie pozwala nam na wcześniejsze przygotowanie zapytania do bazy i późniejsze wstrzyknięcie wartości pomijające składnię `SQL`. Dzięki temu mój kod stanie się mniej podatny na błędy, a samo zapytanie `SQL` zyska na czytelności.

Wygląda to w następujący sposób. Modyfikuję komendę do poniższej postaci:

```
44          String sql = "insert into regions (region_id, region_name) "
45                  + "values ( universal.nextval, ? )";
```

przygotowuję zapytanie przed jego wykonaniem.

Metoda `setString()` wstawia w pierwsze miejsce oznaczone znakiem zapytania podaną wartość

```
try {
    PreparedStatement ps = DatabaseConnector.getConnection().prepareStatement(sql);
    ps.setString(1, r.getRegionName());

    ps.executeUpdate();
    ps.close();
}
catch (Exception e) {
    e.printStackTrace();
}
```

Rysunek 100 - Użycie PreparedStatement do operacji DML

jako `String`. Jednak nie jest to fizyczne wstrzyknięcie wartości do składni `SQL`. Możemy porównać to do bindowania zmiennych w `PL/SQL`.

Z powyższej konstrukcji mamy same korzyści. Tak jak już powiedziałem: oddzielamy samą składnię zapytania od wartości, więc uodparniamy zapytanie na atak `SQL Injection`.

Możemy też wstawiać do kolumn dłuższe teksty (zapytanie `SQL` nie może mieć zwykle więcej niż około 3800 znaków, jeśli je przekroczymy JDBC zwróci wyjątek `String Literal Too Long`), a także strumienie które zapiszą w bazie dane do kolumn typu `BLOB`, albo `Readery` które wstawią dane do kolumn typu `CLOB`.

Oczywiście możemy też używać `PreparedStatement` do uruchamiania zwykłych zapytań `SELECT`, ponieważ `PreparedStatement` dziedziczy po `Statement`.

```
42      public Region getOneById( Long regionID ) {
43          Region r = null;
44          try {
45              String sql = "select * from regions where region_id = ?";
46              PreparedStatement ps = DatabaseConnector.getConnection().prepareStatement(sql);
47              ps.setLong(1, regionID);
48              ResultSet rows = ps.executeQuery();
49              if ( rows.next() ) {
50                  r = new Region();
51                  r.setRegionID( rows.getLong("region_id") );
52                  r.setRegionName( rows.getString("region_name") );
53              }
54              ps.close();
55          }
56          catch (Exception e) {
57              e.printStackTrace();
58          }
59          return r;
60      }
```

Rysunek 101 - Użycie PreparedStatement do kwerend

## Autogenerowanie kluczy

Jeśli programowałeś/aś w `PL/SQL`, prawdopodobnie znasz klauzulę `returning into`. Zwraca ona wartości, które w rzeczywistości trafiły do bazy – po „przepuszczeniu” komendy przez `trigger`'y, odwołania do sekwencji etc. JDBC również oferuje taką funkcjonalność, nazywamy to autogeneracją kluczy.

W aplikacjach bardzo często potrzebujemy wykonać `insert` do bazy danych i zwrócić z niego wartość klucza głównego. Aby móc w ogóle coś takiego wykonać musimy użyć `PreparedStatement`.

Przerobimy tak metodę `dodajRegion()` aby zwracała wartość klucza głównego pod jakim został umieszczony region w bazie. Do metody zwracającej obiekt `PreparedStatement` oprócz zapytania `SQL` muszę podać tablicę `String`'ów, której elementami są nazwy kolumn, których wartości będę chciał przechwycić.

```
String sql = "insert into regions (region_id, region_name) "
           + "values ( universal.nextval, ? )";
System.out.println( sql );
try {
    PreparedStatement ps = DatabaseConnector.
           getConnection().prepareStatement(sql, new String[] { "region_id" } );
    ps.setString(1, r.getRegionName());
```

Rysunek 102 - Autogeneracja kluczy w JDBC

Teraz po wykonaniu `insert`'a (wywołaniu `executeUpdate()`), mogę wywołać metodę `getGeneratedKeys()` zwracającą obiekt typu `ResultSet` w którym zawarte będą nasze wyniki.

Po wywołaniu `next()` na zbiorze wyników mogę już bez przeszkód wyciągać z niego informacje, tak samo jak ze zwykłego `SELECT`'a – **jeden warunek** – nie mogę już używać nazw kolumn, **muszę używać indeksów!**. Po zmianie zwracanego typu przez metodę `dodajRegion()` i dodaniu zmiennej
`Long result = null;` przystępuję do modyfikacji metody do poniższego stanu.

```
72              ps.executeUpdate();
73
74              ResultSet keys = ps.getGeneratedKeys();
75              if ( keys.next() ) {
76                  result = keys.getLong(1);
77              }
78              ps.close();
```

Jeśli użyłbym nazwy kolumny, JDBC podniosłoby wyjątek:

```
insert into regions (region_id, region_name) values ( universal.nextval, ? )
java.sql.SQLException: Funkcja nieobsługiwana
        at oracle.jdbc.driver.OracleReturnResultSet.findColumn(OracleReturnResultSet.java:1909)
        at oracle.jdbc.driver.OracleResultSet.getLong(OracleResultSet.java:444)
        at pl.jsystems.materialyJava.dao.RegionsDao.dodajRegion(RegionsDao.java:76)
        at pl.jsystems.materialyJava.utils.DatabaseConnector.main(DatabaseConnector.java:51)
BUILD SUCCESSFUL (total time: 1 second)
```

Teraz wywołanie `dodajRegion()` za każdym razem zwróci wartość typu `Long` będącą kluczem głównym pod jakim dodany został `Region`.

Wywołanie metody:

```
RegionsDao regionsDao = new RegionsDao();

Region r = new Region();
r.setRegionName( "Bździszewo" );

Long idRegionu = regionsDao.dodajRegion( r );

System.out.println( regionsDao.getOneById(idRegionu).getRegionName() );
```

Output:

```
run:
insert into regions (region_id, region_name) values ( universal.nextval, ? )
Bździszewo
BUILD SUCCESSFUL (total time: 0 seconds)
```

Oraz pełna zawartość metody `dodajRegion()`:

```
public Long dodajRegion( Region r ) {
    Long result = null;
    String sql = "insert into regions (region_id, region_name) "
            + "values ( universal.nextval, ? )";
    System.out.println( sql );
    try {
        PreparedStatement ps = DatabaseConnector.
                getConnection().prepareStatement(sql, new String[] { "region_id" } );

        ps.setString(1, r.getRegionName());
        ps.executeUpdate();

        ResultSet keys = ps.getGeneratedKeys();
        if ( keys.next() ) {
            result = keys.getLong( 1 );
        }
        ps.close();
    }
    catch (Exception e) {
        e.printStackTrace();
    }
    return result;
}
```

## CallableStatement - specjalny typ do obsługi procedur składowanych

Zajmiemy się teraz uruchamianiem procedur składowanych, a także wychwytywaniem parametrów `OUT`. Przygotujemy sobie najpierw w bazie jakiś prosty pakiet i dwie procedury.

Nagłówek:

```
create or replace package dummy is

  procedure without_out_params( nazwa varchar2 ) ;
  procedure with_out_params ( nazwa varchar2, ilosc out integer );

end;
```

I ciało pakietu:

```
create or replace package body dummy is

  procedure without_out_params( nazwa varchar2 ) is
    pragma autonomous_transaction;
    begin
      update employees set salary = salary *1.1
        where department_id in (
          select department_id from departments where department_name = nazwa
        );
      commit;
  end;
  procedure with_out_params ( nazwa varchar2, ilosc out integer ) is
  pragma autonomous_transaction;
    begin
      update employees set salary = salary *1.1
        where department_id in (
          select department_id from departments where department_name = nazwa
        );
      ilosc := sql%rowcount;
      commit;
  end;

end;
```

Są to dwie banalne procedury dające 10% podwyżki wszystkim pracującym w danym departamencie. Różnią się tylko tym, że druga przekazuje ilość zmodyfikowanych wierszy jako parametr `OUT`. Zaczniemy od wywołania tej pierwszej. Stworzę zatem oddzielną klasę EmployeesDao w której zawrę metodę `dajPodwyzke()`.

```
12    public class EmployeesDao {
13
14        public void dajPodwyzke( String departmentName ) {
15
16        }
17
18    }
```

Następnie muszę stworzyć obiekt zdolny do uruchomienia procedury czyli `CallableStatement`. Robię to dokładnie tak samo jak poprzednio:

```
17        public void dajPodwyzke( String departmentName ) {
18            try {
                  CallableStatement cs = DatabaseConnector.getConnection()
20                                .prepareCall("{call dummy.without_out_params( ? ) }");
21                cs.setString(1, departmentName);
22                cs.executeUpdate();
23                cs.close();
24            }
25            catch (Exception e) {
                  e.printStackTrace();
27            }
28        }
```

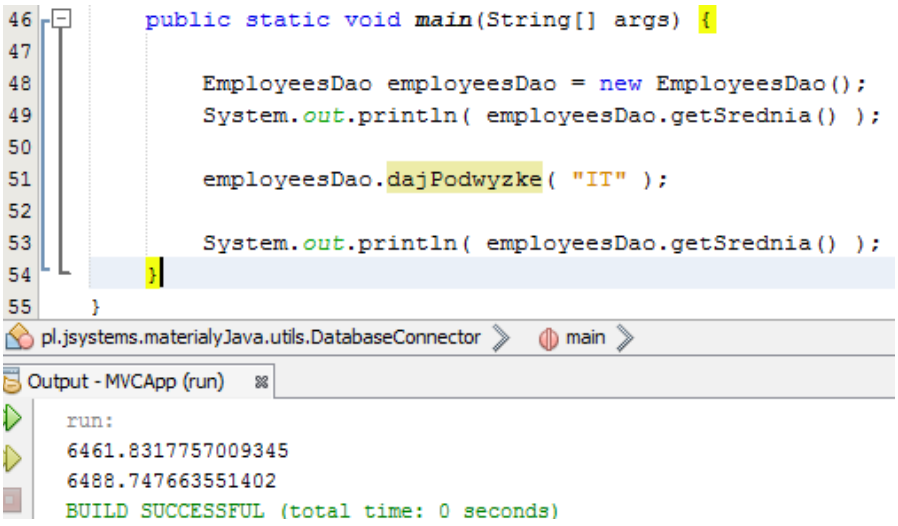
Wygląda to analogicznie do operacji `DML` wykonywanych za pomocą `PreparedStatement` (`CallableStatement` dziedziczy po `PreparedStatement`). Należy pamiętać, że wywołanie procedury przez JDBC ma kompletnie inną składnię niż w `SQL`.

Wywołanie procedury przez JDBC ma następującą strukturę:

```
1.  {call [nazwa procedury i parametry] }
```

Dalsza część metody powinna być już zrozumiała. Sprawdźmy czy procedura działa:

Przygotowałem sobie wcześniej metodę odczytującą średnią zarobków wśród pracowników.




Jak widać po daniu podwyżki całemu działowi IT średnia zarobków wzrosła czyli uruchomienie procedury działa prawidłowo.

### Odbieranie parametrów OUT procedur

Wszyscy wiemy jak często korzysta się z parametrów wyjściowych procedur. Ich odbiór na poziomie JDBC na szczęście nie jest trudny. Wrócimy do metody `dajPodwyzke()` z klasy `EmployeesDao`. Zmienię wywołanie tak, aby uruchamiało drugą procedurę:

```
40              CallableStatement cs = DatabaseConnector.getConnection()
                        .prepareCall("{call dummy.with_out_params( ?, ? ) }");
```

Oczywiście wprowadzam też zmienną i modyfikuję deklarację metody tak, aby zwracała wartość typu `Integer`. Jak widać procedura przyjmuje dwa argumenty z czego jeden jest typu `OUT`. Jeśli chcę wyjąć z niego wartość muszę poinformować JDBC o tym, który parametr jest typu `OUT` i jakiego jest typu. Robimy to metodą `registerOutParameter()`.

```
40                      .prepareCall("{call dummy.with_out_params( ?, ? ) }")
41
42              cs.registerOutParameter(2, java.sql.Types.INTEGER);
```

Po raz kolejny przypominam, że indeksy w JDBC liczymy od `1`! Drugi argument metody to numer typu którego jest parametr. Oczywiście numerków nikt nie pamięta, dlatego w klasie `Types` (z pakietu `java.sql`) zdefiniowano odpowiednie stałe. Po wywołaniu `cs.executeUpdate()`, mogę wyjąć wartość parametru wywołując odpowiednią metodę i podając indeks parametru.

```
            cs.executeUpdate();
45          result = cs.getInt(2);
46          cs.close();
```

Wywołanie i output:

```
46  public static void main(String[] args) {
47
48      EmployeesDao employeesDao = new EmployeesDao();
49      System.out.println( employeesDao.getSrednia() );
50
51      Integer ilosc = employeesDao.dajPodwyzke( "IT" );
52      System.out.println("Podwyzke dostalo " + ilosc + " osób" );
53
54      System.out.println( employeesDao.getSrednia() );
55  }
```

```
Output - MVCApp (run)
run:
6518.355140186916
Podwyzke dostalo 5 osób
6550.923364485981
BUILD SUCCESSFUL (total time: 0 seconds)
```

Oraz pełny kod metody `dajPodwyzke()`:

```
36  public Integer dajPodwyzke( String departmentName ) {
37      Integer result = null;
38      try {
            CallableStatement cs = DatabaseConnector.getConnection()
40                  .prepareCall("{call dummy.with_out_params( ?, ? ) }");
41
42          cs.registerOutParameter(2, java.sql.Types.INTEGER);
43          cs.setString(1, departmentName);
44          cs.executeUpdate();
45          result = cs.getInt(2);
46          cs.close();
47      }
48      catch (Exception e) {
            e.printStackTrace();
50      }
51      return result;
52  }
```

# Graficzny Interfejs Użytkownika – SWING

## Krótko o samym Swingu'u

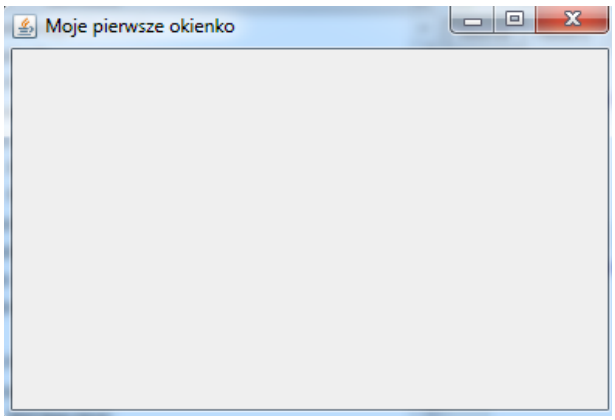
Swing jest biblioteką Javy za pomocą której możemy tworzyć bardziej wyrafinowany interfejs użytkownika niż tylko napisy na konsoli. Nie jest już najnowszą biblioteką – najnowszą jest JavaFX. Swng jest jednak najstabilniejszą i najszerzej używaną biblioteką do tworzenia okienek ☺.

## Pierwsze okienko

Aby utworzyć okienko w Javie. Nasza klasa musi dziedziczyć po klasie `JFrame` znajdującej się w pakiecie `javax.swing`. Całą inicjalizację interfejsu użytkownika umieścimy w konstruktorze.

```
18      public MVCApp() {
19          super( "Moje pierwsze okienko" );
20          setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE );
21          setBounds(100, 100, 400, 270);
22      }
23
24      public static void main(String[] args) {
25          new MVCApp().setVisible( true );
26      }
```

Wynik jest następujący:




Wywołanie `super()` z argumentem ustawia tytuł okienka, można też użyć w tym celu metody `setTitle()` – jak kto woli. Instrukcja `setDefaultCloseOperation()` określa jaka akcja zostanie wykonana po kliknięciu na przycisk zamknięcia okna. Mamy tu trzy możliwości:

- `EXIT_ON_CLOSE` – powoduje zamknięcie całej aplikacji
- `DISPOSE_ON_CLOSE` – powoduje zamknięcie pojedynczego okna
- `DO_NOTHING_ON_CLOSE` – nie pociąga za sobą żadnej akcji

Metoda setBounds() określa rozmiar i pozycję okienka. Pierwszy argument to przesunięcie w prawo od lewej krawędzi ekranu, drugi odległość od góry ekranu, trzeci określa szerokość okna, a czwarty

jego wysokość. Na koniec w metodzie `main` po prostu tworzę obiekt mojego okna i ustawiam je jako widoczne.

### JPanel i podstawowe kontrolki

W zasadzie moglibyśmy już zacząć nanosić jakieś kontrolki na nasze okienko, ale wg konwencji nie należy „pisać" bezpośrednio na `JFrame`. Wprowadzimy do naszego kodu obiekt `JPanel`, który jest po prostu kontenerem do którego dodajemy kontrolki.

Napiszemy prosty kalkulator, który będzie przeliczał stopnie Fahrenheita na stopnie Celsjusza. Zaczniemy od dodania do naszego okna `JPanel`'u.

```
    private JPanel panel;
18
19  public MVCApp() {
20      super( "Moje pierwsze okienko" );
21      setDefaultCloseOperation( JFrame.DO_NOTHING_ON_CLOSE );
22      setBounds(100, 100, 400, 270);
23
24      panel = new JPanel();
25      panel.setLayout(null);
26      add(panel);
27  }
```

Przyjęło się, że wszystkie kontrolki w ramach jednego `JFrame`, trzymamy w polach naszej klasy. Metoda `setLayout()` ustawia tzw. `LayoutManager` w obrębie `JPanel`. Będziemy jeszcze je omawiać, podanie do metody wartości `null` spowoduje, że ręcznie będziemy podawać rozmiar i pozycję każdej kontrolki.

Metoda `add()`, po prostu dodaje `JPanel` do `JFrame`'a.

Wygląd naszego okienka oczywiście się nie zmieni.

Dodam teraz do `JPanel`'a kilka podstawowych kontrolek:

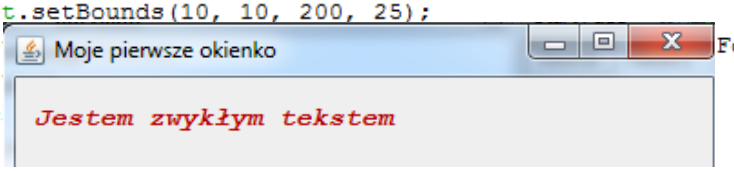
- `JLabel` – statyczny tekst, nieedytowalny przez użytkownika
- `JTextField` – podstawowe pole tekstowe
- `JButton` – Zwykły przycisk

Zacznę od `JLabel`. Dodaję pole do mojej klasy i umieszczam w konstruktorze poniższy wpis:

```
tekst = new JLabel("Jestem zwykłym tekstem");
tekst.setBounds(10, 10, 200, 25);
panel.add(tekst);
```

Konstruktor `JLabel` przyjmuje tekst, który będzie wyświetlał w okienku . Oczywiście możemy troszkę pozmieniać nasz tekst. Ograniczymy się do zmiany czcionki i koloru.

```
35          tekst = new JLabel("Jestem zwykłym tekstem");
36          tekst.setBounds(10, 10, 200, 25);
37          teks
38          teks
39          pane
```




Font.ITALIC), 15) );

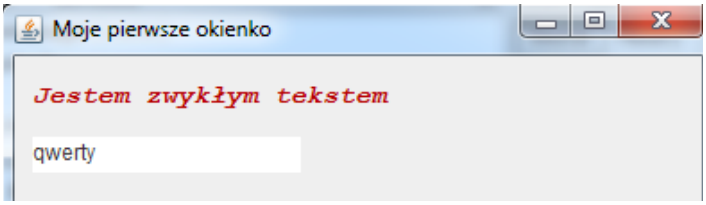
Teraz okienko wyglądać będzie tak jak na poniższym screen'ie:

Metoda `setFont()` zmienia czcionkę `JLabel`, a `setForeground()` zmienia kolor czcionki. Klasami `Color` i `Font` nie będziemy się zajmować. Ich używanie jest bardzo intuicyjne.

Za proste pola tekstowe odpowiada klasa `JTextArea`. Po dodaniu pola i poniższego wpisu w konstruktorze, w okienku powstanie pole tekstowe.

```
field = new JTextArea();
field.setBounds(10, 45, 150, 20);
panel.add(field);
```

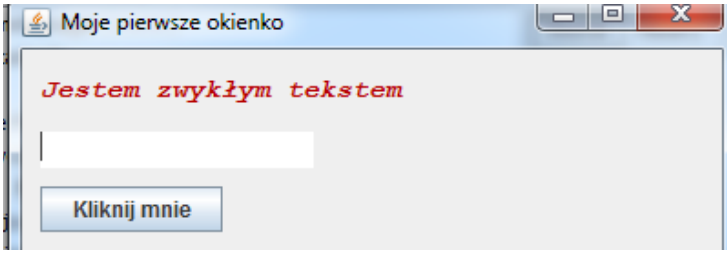
Teraz okienko wygląda jak poniżej:



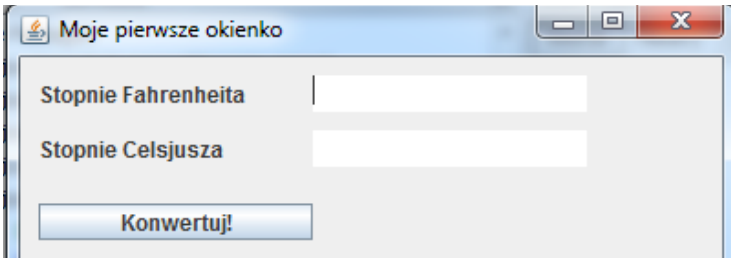

Zajmiemy się ostatnią z najbardziej podstawowych kontrolek – `JButton`. Dodamy teraz przycisk do naszego prostego okienka. Standardowo dodajemy pole i wpis do konstruktora.

```
button = new JButton( "Kliknij mnie" );
button.setBounds(10, 75, 100, 25);
panel.add( button );
```

Domyślnie przycisk wygląda tak jak teraz:




Oczywiście po jego kliknięciu nic jeszcze się nie dzieje. Niedługo to zmienimy, ale najpierw przygotuję sobie podstawowe GUI do kalkulatora, który będziemy pisać.




## Podpinanie zdarzeń do kontrolek – interfejs ActionListener

Aby nasze kontrolki były interaktywne podepniemy do nich słuchacza zdarzeń. Kontrolki będą nadajnikami zdarzeń, a `JFrame` odbiornikiem. Aby okienko mogło słuchać zdarzeń musi implementować interfejs `ActionListener`, zdefiniowano w nim jedną metodę – `actionPerformed()`, która przyjmuje jako argument obiekt zdarzenia.

```
public class MVCApp extends JFrame implements ActionListener {

    private JPanel panel;
```

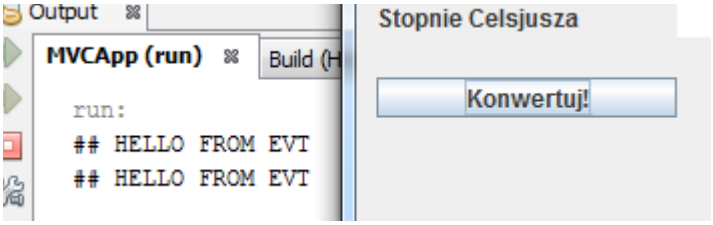
Oraz metoda `actionPerformed()`:

```
66      @Override
        public void actionPerformed(ActionEvent evt) {
68
69      }
```

Zostanie ona wywołana za każdym, gdy jakiś obiekt (nieważne który) stworzy zdarzenie. Zaimplementowałem tam po prostu wyświetlenie „HELLO FROM EVT” na konsolę. Następnie muszę poinformować `JButton` o tym, że okno będzie słuchało zdarzeń typu `ActionEvent`. Przechodzę do konstruktora i dodaję kolejną linijkę.

```
button = new JButton("Konwertuj!")
button.setBounds(10, 80, 150, 20);
button.addActionListener(this);
```

Teraz po kliknięciu na przycisk na konsoli widać kolejne komunikaty:




W obecnej formie, ta sama akcja jest wywoływana dla każdego obiektu. Aby móc odróżnić źródła zdarzeń wystarczy na obiekcie zdarzenia wywołać metodę `getSource()`, która zwraca obiekt źródła zdarzenia.

```
67      @Override
        public void actionPerformed(ActionEvent evt) {
69          if ( evt.getSource() == button ) {
70              System.out.println("## HELLO FROM EVT");
71          }
72      }
```

Użycie zwykłego porównania powinno być raczej oczywiste. Chodzi mi po zadanie pytanie czy obiekt generujący zdarzenie i przycisk to jeden i ten sam obiekt.

Zdarzenie typu `ActionEvent` jest generowane w momencie wystąpienia akcji charakterystycznej dla danej kontrolki – np. w przypadku `JButton` będzie to kliknięcie, ale w przypadku `JTextField` będzie to naciśnięcie `[ENTER]`.

Dla bardziej wyspecjalizowanych zdarzeń np. myszy używamy interfejsu `MouseListener`. Zaimplementujemy go teraz:

```
CApp extends JFrame implements ActionListener, MouseListener {

nel panel;
```

Pojawią się dodatkowe metody w naszej klasie:

```
77      @Override
        public void mouseClicked(MouseEvent me) {
79          System.out.println( "onClick" );
80      }
81
82      @Override
        public void mousePressed(MouseEvent me) {
84          System.out.println( "mouseDown" );
85      }
86
87      @Override
        public void mouseReleased(MouseEvent me) {
89          System.out.println( "mouseUp" );
90      }
91
92      @Override
        public void mouseEntered(MouseEvent me) {
94          System.out.println( "mouseOver" );
95      }
96
97      @Override
        public void mouseExited(MouseEvent me) {
99          System.out.println( "mouseOut" );
.00     }
```

Podpięcie `MouseListener`'a do `JButton` wygląda analogicznie jak w przypadku `ActionListener`'a.

```
button.addActionListener(this);
button.addMouseListener(this);
```

## LayoutManager – automatyczne skalowanie kontrolek do rozmiaru okna

Używaliśmy już `LayoutManager`'a nawet o tym nie wiedząc. Domyślnym layoutem dla każdego kontenera – `JFrame`, `JPanel` jest `FlowLayout`. Zajmiemy się najpierw `BorderLayout`. Reprezentuje on poniższy układ komponentów:

| PAGE_START | | |
|---|---|---|
| **LINE_START** | **CENTER** | **LINE_END** |
| **PAGE_END** | | |

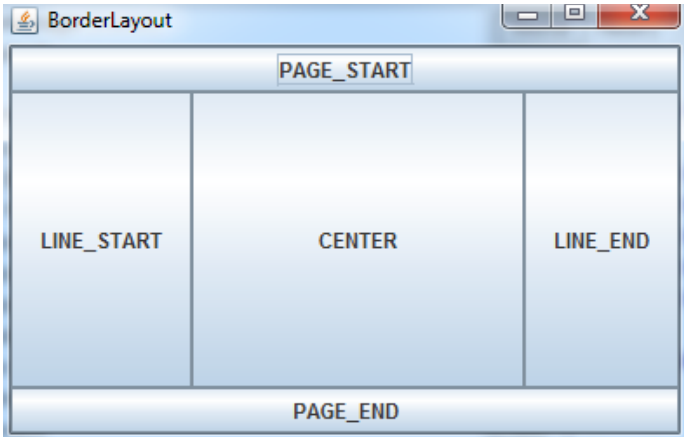
Omówimy kilka najważniejszych `LayoutManager`'ów:

- `BorderLayout`
- `FlowLayout`
- `GridLayout`
- `Null Layout`

Dodam teraz kilka `JButton`'ów do `JPanel` na którym ustawię `BorderLayout`. Pozycję `JButton`'a w `BorderLayout` określamy cyfrą w trakcie dodawania komponentów do `JPanel`.

```
panel = new JPanel( new BorderLayout() );

panel.add( buttonTop, BorderLayout.PAGE_START );
panel.add( buttonLeft, BorderLayout.LINE_START );
panel.add( buttonCenter, BorderLayout.CENTER );
panel.add( buttonRight, BorderLayout.LINE_END );
panel.add( buttonBottom, BorderLayout.PAGE_END );
```

Okienko będzie wyglądać dokładnie tak, jak na powyższym obrazku:



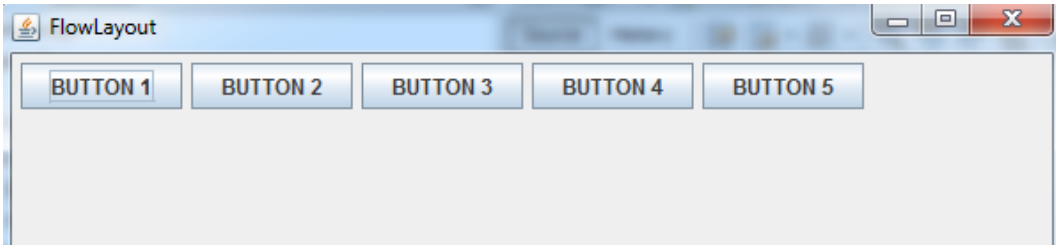

Jak widać, komponent wstawiony na konkretną pozycję w `BorderLayout` zostanie automatycznie doskalowany do rozmiaru swojej sekcji. Najczęściej używa się go w połączeniu z innymi layoutami. Jako komponent ustawiamy inny `JPanel`, a w nim inny layout.

Zajmiemy się teraz domyślnym layoutem – czyli `FlowLayout`. Ustawiamy go dokładnie tak samo jak `BorderLayout`. `FlowLayout` umieszcza kolejne komponenty liniowo w ustawieniu

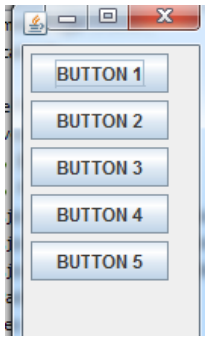
horyzontalnym, wertykalnym – zależnie od ustawionego wyrównania. Domyślnie `FlowLayout` centruje umieszczone w nim komponenty.

```
42              panel = new JPanel( new FlowLayout( FlowLayout.LEFT ) );
43
44              panel.add( button1 );
45              panel.add( button2 );
46              panel.add( button3 );
47              panel.add( button4 );
```

Do konstruktora `FlowLayout` podaję liczbę, która określa wyrównanie komponentów. Teraz dodaję już kolejne kontrolki normalnie. Kontrolki zostaną ułożone poziomo lub pionowo (zależnie od szerokości okna) i wyrównane do lewej:



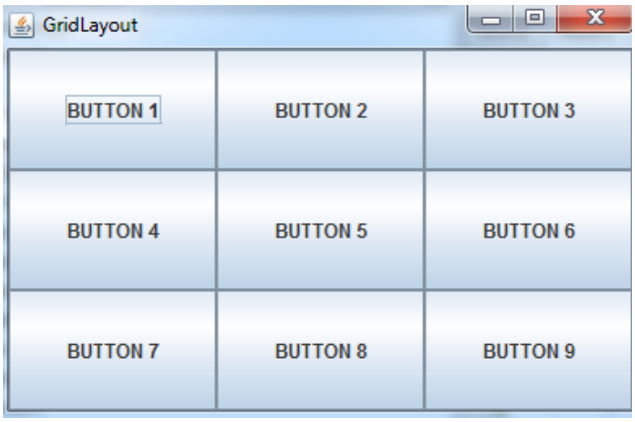

Jeśli zwężę okno kontrolki ułożą się jedna pod drugą:




GridLayout służy do ustawiania komponentów w kształt tabelki, ułożymy z niego teraz prostą klawiaturę.

```
panel = new JPanel( new GridLayout(3, 3) );
```

Konstruktor `GridLayout` przyjmuje dwa argumenty, pierwszym jest ilość wierszy, a drugim ilość kolumn naszej pseudo-tabelki. Po dodaniu dziewięciu `JButton`'ów zostaną one ułożone w kształt prostej tabelki w której każdy komponent ma taki sam rozmiar:




`NullLayout` jest dość często używany, ponieważ wymusza na programiście obowiązek podania rozmiaru i pozycji każdego komponentu. Wtedy programista ma pełną kontrolę nad układem komponentów. Używaliśmy go na samym początku pisząc:

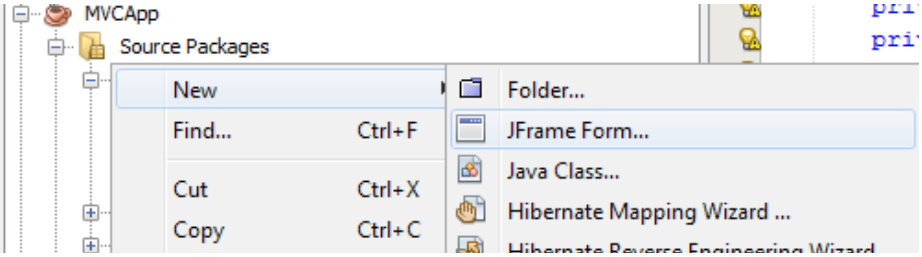
```
51          panel = new JPanel( null );
```

# Obsługa kolejnych kontrolek

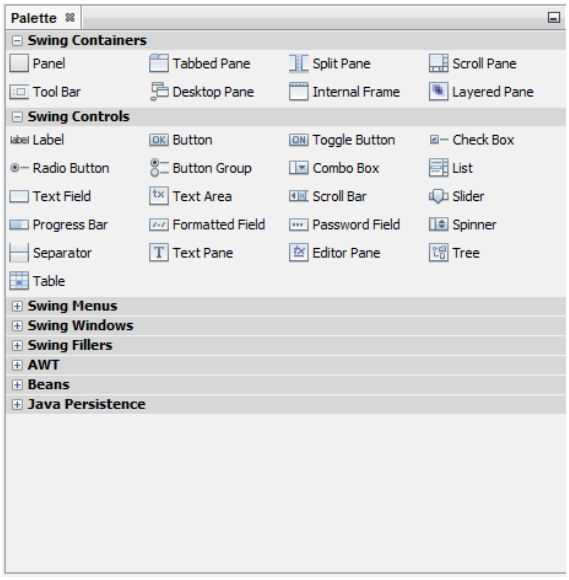
## Korzystanie z GUI Designer'a w NetBeans

Podczas tworzenia bardziej skomplikowanego interfejsu okienkowego powstaje niebotyczna ilość kody odpowiedzialna za samo inicjowanie sceny, podpięcie zdarzeń etc. Dlatego bardzo modne stały się GUI `Designer`'y. NetBeans IDE posiada wbudowany Designer, dla Eclipse trzeba ściągnąć wtyczkę – najpopularniejsza to `WindowDesigner`.

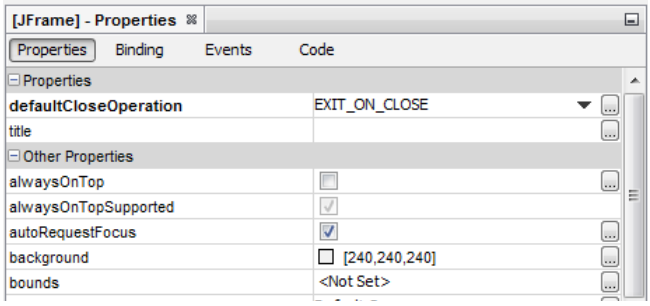
Aby móc skorzystać z GUI Designer'a w NetBeans musimy utworzyć nowy `JFrame Form`.



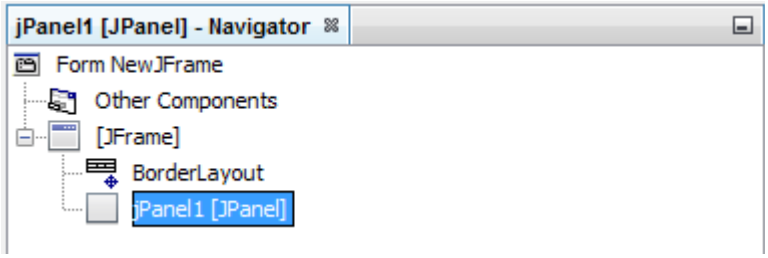

Kolejny widok to już specjalna zakładka dla JFrame pozwalająca projektować interfejs użytkownika metodą `Drag and Drop`. Przyjrzymy się teraz panelom narzędziowym. Po prawej stronie mamy paletę z wbudowanymi kontrolkami:



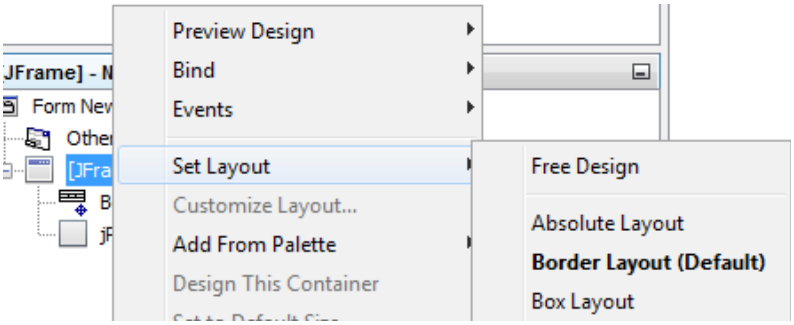

Pod nią znajduje się panel ze wszystkimi właściwościami danego komponentu:



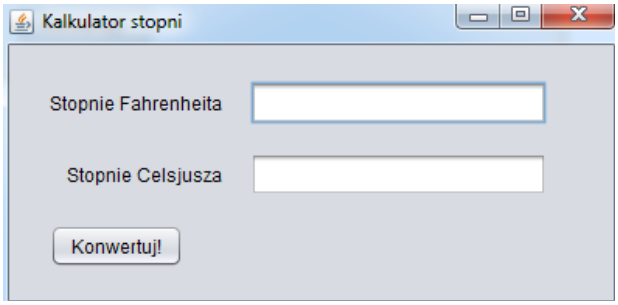

Po lewej stronie na dole znajduje się panel ze strukturą naszych komponentów pozwalający łatwo wybierać nawet te niewidoczne.




Aby zmienić podstawowe właściwości danego komponentu (np. layout w przypadku `JFrame`) wystarczy że kliknę prawym przyciskiem myszy na pozycję reprezentującą `JFrame` i z menu kontekstowego po prostu zmieniam layout.

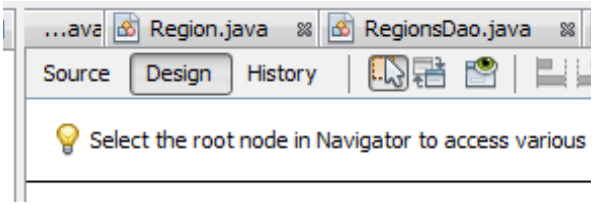


Teraz bezproblemowo przygotujemy takie samo GUI dla naszego kalkulatora po prostu przeciągając odpowiednie kontrolki do `JPanel`.




### Krótko o Pluggable Look and Feel

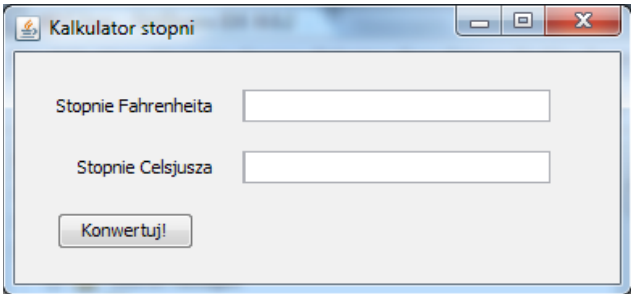
Jak już zauważyłeś/aś nasze kontrolki zaprojektowane za pomocą NetBeans'a różnią się wyglądem od tych, które pisaliśmy ręcznie. Wynika to z tego, że w Javie istnieje coś takiego jak `Pluggable Look and Feel` czyli wiele styli kontrolek. Domyślnym typem kontrolek jest styl `Metal` (słabo wyglądający na Windowsie, ale świetnie prezentuje się na Linuksach), natomiast NetBeans domyślnie ustawia styl `Nimbus` (ten który widać powyżej). Aby zmienić ustawienia styli, będziemy na chwilę wrócić do kodu. Wybieramy zakładkę `Source` z menu nad obszarem projektowania:




Po rozwinięciu zakładki `Look and Feel setting code` wystarczy, że zmienimy Nimbus na Windows, a kontrolki od razu zaczną wyglądać bardziej znajomo.

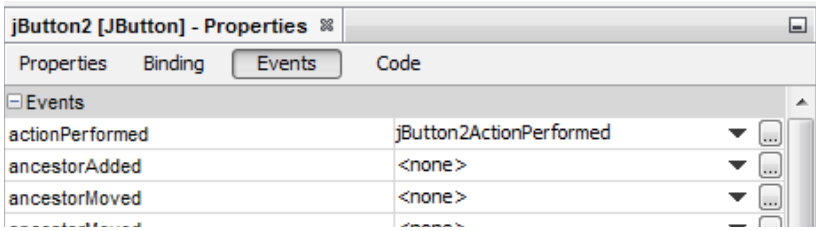
```
 */
try {
    for (javax.swing.UIManager.LookAndFeelInfo info : javax.swir
        if ("Windows".equals(info.getName())) {
            javax.swing.UIManager.setLookAndFeel(info.getClassNa
            break;
        }
    }
} catch (ClassNotFoundException ex) {
    java.util.logging.Logger.getLogger(NewJFrame.class.getName()
```

Zmianę widać od razu:



Napiszemy teraz kod, który spowoduje, że po kliknięciu na przycisk zostanie wykonana konwersja stopni Fahrenheita do stopni Celsjusza.

Używając Designer'a mogę po prostu dwa razy kliknąć na przycisk, a zostanie dodana specjalna metoda wywoływana w momencie kliknięcia na przycisk. Jeśli jednak chciałbym użyć bardziej wyspecjalizowanych zdarzeń, muszę przejść do zakładki `Events`, panelu z właściwościami mojego `JButton`.



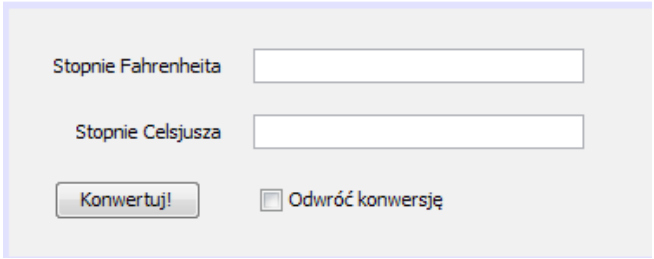

Implementacja nie jest zbyt skomplikowana:

```
private void jButton2ActionPerformed(java.awt.event.ActionEvent evt) {
    try {
        Double stopnieF = Double.parseDouble( jTextField1.getText() );
        Double stopnieC = (stopnieF - 32)* ( 5.0 / 9 );

        jTextField2.setText( String.valueOf(stopnieC) );
    }
    catch (NumberFormatException e) {
        jTextField2.setText( "Wprowadzono nieprawidłowe dane!" );
    }
}
```

## JCheckBox i JRadioButton – konfigurowanie działania programu

Umożliwimy teraz konwersję dwukierunkową - do tej pory dało się konwertować wyłącznie Fahrenheity na Celsjusze. Zrobimy to najpierw prostszą w obsłudze kontrolką – czyli `JCheckBox`, a później grupą `JRadioButton`'ów.

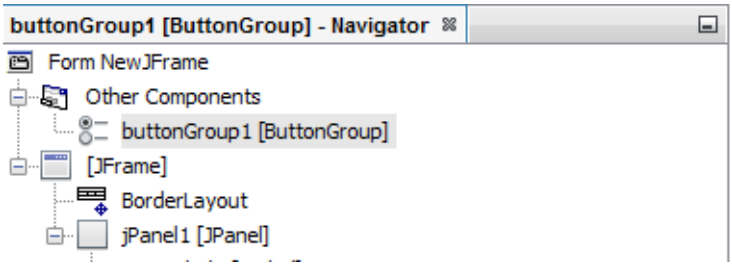
`JCheckBox` to bardzo prosta, doskonale znana kontrolka z poniższego screena. W zasadzie jedyne używane jej właściwości to `label` – czyli tekst obok samego `checkbox`'a oraz właściwość `selected` określająca czy `checkbox` ma domyślnie być zaznaczony.



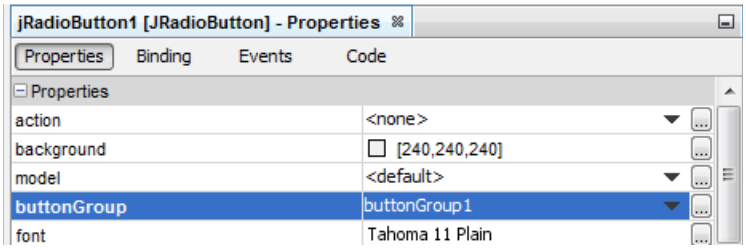

Jego obsługa programowa też jest bardzo prosta. Z całej gamy metod, które możemy wywołać na `JCheckBox` najbardziej będzie interesować nas prosta metoda `isSelected()` zwracająca wartość `boolean`.

```
if ( !jCheckBox1.isSelected() ) {
    try {
        Double stopnieF = Double.parseDouble( jTextFie
        Double stopnieC = (stopnieF - 32)* ( 5.0 / 9 )

        jTextField2.setText( String.valueOf(stopnieC)
```

Zajmiemy się teraz kontrolką `JRadioButton` jej wykorzystanie wymaga od nas wprowadzenia drugiego komponentu – `ButtonGroup`. Wystarczy ją przeciągnąć z palety do naszego okna:

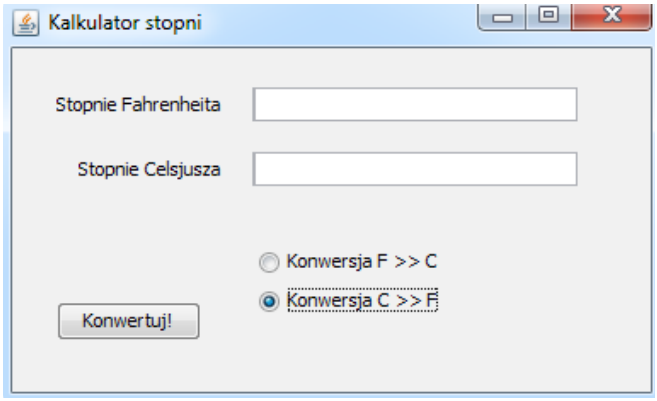



Następnie przeciągam dwa `JRadioButton` do okna, ale aby działały w połączeniu ze sobą, musimy podpiąć oba `JRadioButton` pod jedną `ButtonGroup`.




Okienko teraz wygląda tak.

Oczywiście `JRadioButton` również posiada właściwość `selected`, która działa dokładnie tak samo jak w przypadku `JCheckBox`.

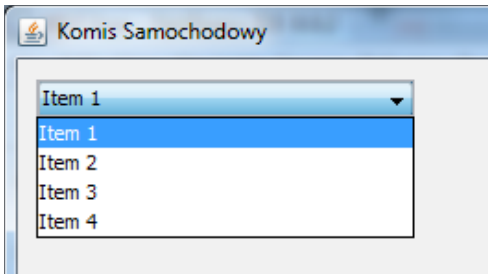



Obsługa programowa nie różni się niczym od `JCheckBox`.

```
if ( jRadioButton1.isSelected() ) {
    try {
        Double stopnieF = Double.parseDouble( j
        Double stopnieC = (stopnieF - 32)* ( 5.
```

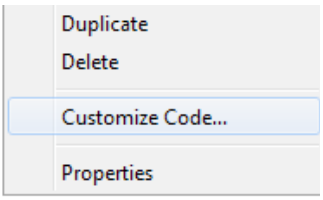
## Trudniejsze komponenty – JComboBox

`JComboBox` to po prostu rozwijana lista:

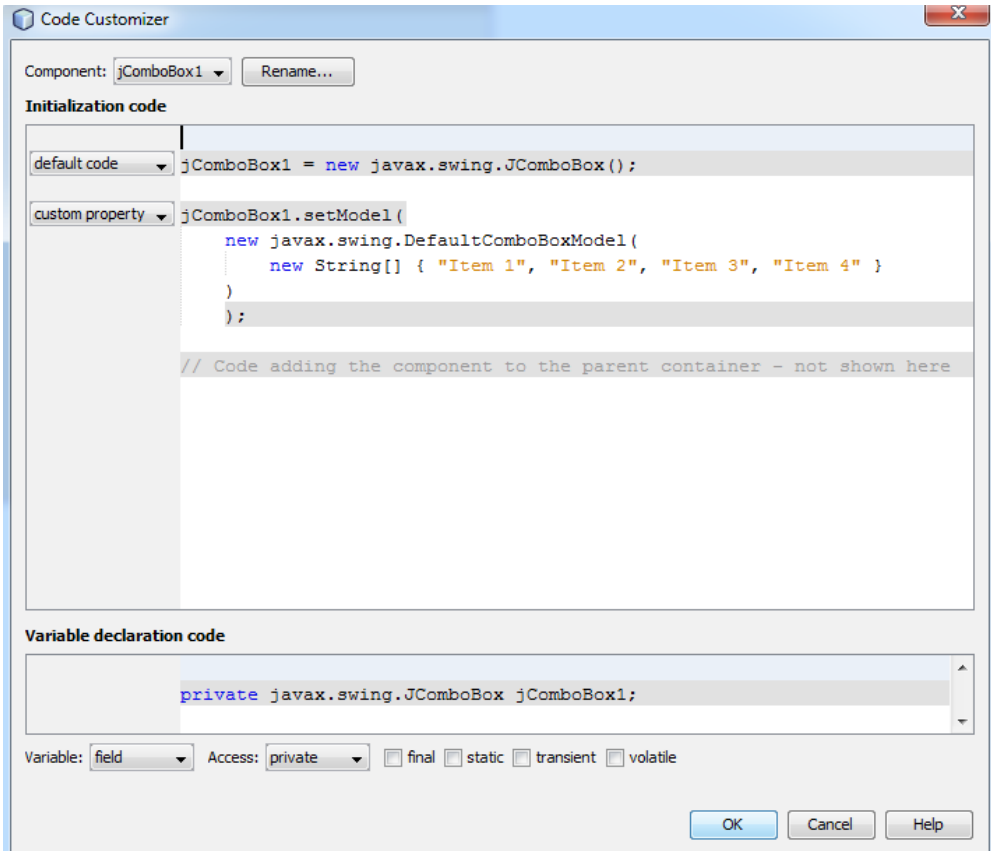



Od prostego HTML-owego selektora odróżnia go to, że w `JComboBox`, trzymamy całe obiekty, a napisy, które widać w liście to po prostu wynik wywołania metody `toString()` na obiektach trzymanych w `JComboBox`. Takie listy mają nieco bardziej skomplikowaną strukturę niż np. `JCheckBox`. Chodzi o to, że w Swingu rozdzielamy dane trzymane w `JComboBox` od samego selektora. Samo wprowadzenie `JComboBox` do okna nie jest trudne – wystarczy przeciągnąć pozycję `ComboBox` do okna. Uzyskasz wtedy dokładnie taki sam `ComboBox` jak na powyższym obrazku.

Zmiana danych w `ComboBox`'ie nie jest już taka prosta. Zobaczmy jak dokładnie wygląda stworzenie `ComboBox` od strony kodu. Aby od razu przejść do odpowiedniego miejsca kliknij w pozycję `Customize Code` w menu kontekstowym `ComboBox`.

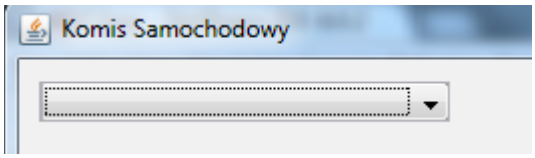



Otrzymasz poniższe okno. Aby móc zmienić jakąkolwiek linijkę z bocznego selektora wybierz pozycję `Custom Property`.




`ComboBox`'y są jednym z komponentów, które do swojego działania wykorzystują tzw. Model. Model reprezentuje dane przechowywane w danym selektorze. Jeśli chcielibyśmy mieć kilka `ComboBox`'ów, w których zawsze przechowywane byłyby te same obiekty, musielibyśmy wyeksportować model do jakiegoś pola i pracować tylko na nim. Mamy dwa typy reprezentujące model w ComboBox'ach – `ComboBoxModel` (interfejs), `DefaultComboBoxModel` (klasa), oba typy są generyczne dlatego mogę określić dokładnie jaki typ ma przechowywać mój selektor. `DefaultComboBox` model do `kostruktora` przyjmuje tablicę obiektów (lub innych typów), która zostanie wstawiona do selektora.

Po jej usunięciu lista zostanie wyczyszczona ( `DefaultComboBoxModel` posiada pusty konstruktor).



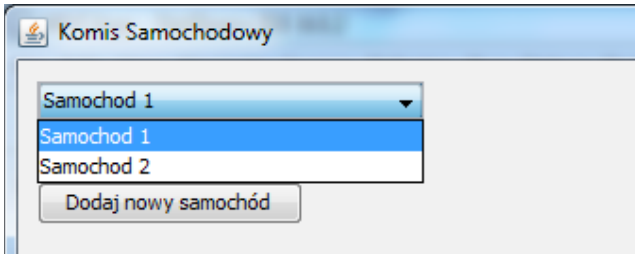

Aby teraz dodać cokolwiek do mojego `ComboBox` muszę wyjąć obiekt modelu. Dodam zatem jakiś `JButton`, którego akcję oprogramuję.

```
DefaultComboBoxModel<Samochod> model =
                    (DefaultComboBoxModel<Samochod>)jComboBox1.getModel();
model.addElement( new Samochod() );
```

Warto przyjrzeć jeszcze się klasie `Samochod`. Obiekt tej klasy ma kilka właściwości numer, markę oraz moc. Metoda `toString()` zwraca nawis informujący o numerze `Samochod`u.

```
        return moc;
    }

    public void setMoc(Double moc) {
        this.moc = moc;
    }

    @Override
    public String toString() {
        return "Samochod " + carID;
    }
```

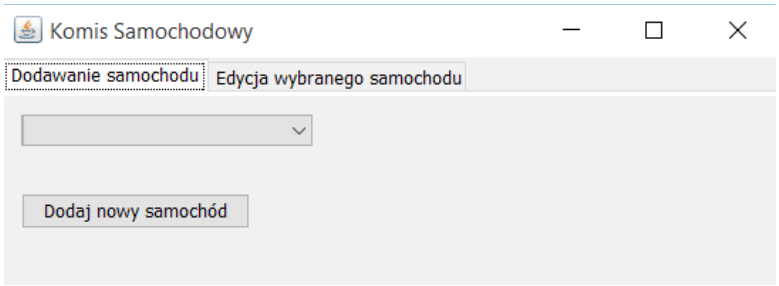
Po kilkukrotnym kliknięciu na oprogramowany przycisk wynik od razu rzuca się w oczy:



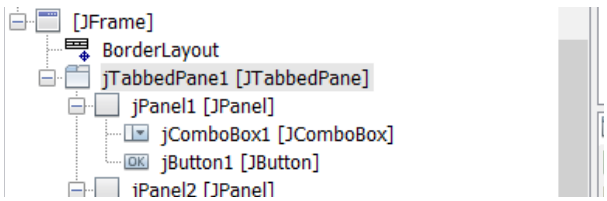

Stworzymy teraz prosty formularz do dodawania i edytowania `Samochod`ów umieszczonych w `ComboBox`. W tym celu wykorzystamy kolejny komponent czyli `TabbedPane`.

### TabbedPane – tworzenie zakładek z oddzielnymi JPanel'ami

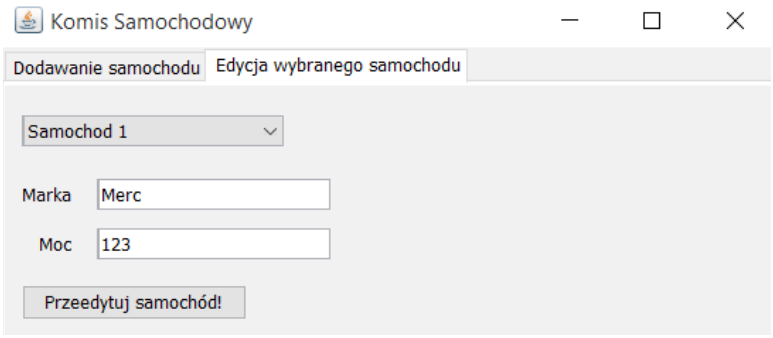
Wprowadzimy sobie bardziej złożony kontener czyli `TabbedPane`. Przechowuje on komponenty `JPanel` i tworzy zakładki pomiędzy którymi łatwo się przełączać:




Aby dodać `TabbedPane` do `JFrame` wystarczy po prostu przeciągnąć go do okna. Z kolei aby dodać zakładkę do `TabbedPane`, wystarczy przeciągnąć komponent `JPanel` do `TabbedPane` (paradoksalnie nie jest to takie łatwe ☺ powinny podświetlić się takie pomarańczowe linie na `TabbedPane`). W każdym razie struktura `JFrame` powinna wyglądać tak:



Przygotujemy sobie formularz edycji pojedynczego samochodu. W jednej zakładce będzie znajdować się tylko przycisk do dodawania samochodów, a w drugiej formularz edycji z `ComboBox` do wybrania odpowiedniego samochodu.




Dodanie nowego, pustego samochodu od strony programowej już pokazałem, zajmiemy się tylko obsługą zmiany samochodu z `ComboBox`. Takie zdarzenie to po prostu `ActionEvent`. Obsługa zmiany samochodu to tylko zaktualizowanie pól formularza:

```
private void jComboBox2ActionPerformed(java.awt.event.ActionEvent evt) {
    Samochod choise = (Samochod)jComboBox2.getSelectedItem();
    jTextField1.setText( choise.getMarka() );
    jTextField2.setText( String.valueOf( choise.getMoc() ) );

}
```

Tak jak powiedziałem, w `ComboBox` trzymamy obiekty w całości! Dlatego mogę wyciągnąć z niego obiekt, który wystarczy zrzutować na `Samochod`. Dalsze instrukcje są już trywialne.

Następnie obsługa kliknięcia na przycisk `Przeedytuj samochód!` sprowadza się do bardzo podobnych operacji:

```
151    private void jButton2ActionPerformed(java.awt.event.ActionEvent evt) {
152        Samochod choise = (Samochod)jComboBox2.getSelectedItem();
153        try {
154            choise.setMarka( jTextField1.getText() );
155            Double moc = Double.parseDouble( jTextField2.getText() );
156            choise.setMoc( moc );
157        }
158        catch (NumberFormatException e) {
159            jTextField2.setText( "Nieprawidłowe dane" );
160        }
161    }
```

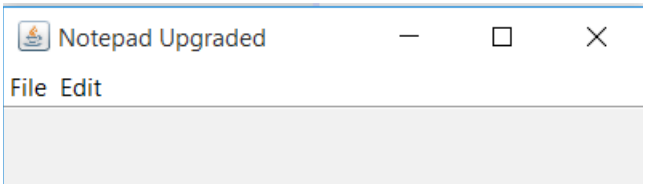
## Menu w Java Swing

W aplikacjach desktopowych rozwijane menu jest jednym z najważniejszych komponentów do nawigacji pomiędzy kolejnymi widokami w programie. Do budowy menu używamy elementów poniższych klas:

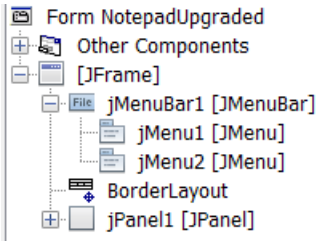
- `JMenuBar`
- `JMenuItem`
- `JMenu`
- `JCheckboxMenuItem`
- `JRadioButtonMenuItem`

Napiszemy prosty edytor tekstowy, do którego dodamy możliwość zapisywania pliku z menu. Ponadto udostępnimy dedykowane menu do wstawiania częstych wartości to pola tekstowego – np. Aktualną datę.

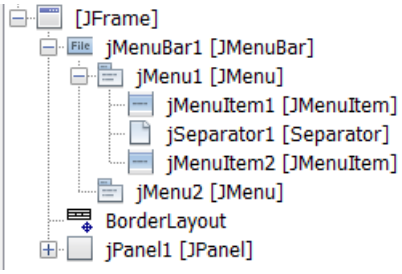
Zaczniemy od stworzenia `JFrame` (z ustawionym `BorderLayout`) i `JPanel`. Następnie musimy kliknąć prawym przyciskiem myszy kliknąć na `JFrame` i wybrać `Add From Palette` **>>** `Swing Menus` **>>** `Menu Bar`. Spowoduje to dodanie paska z menu i przypisanie do niego dwóch elementów typu `JMenu`.



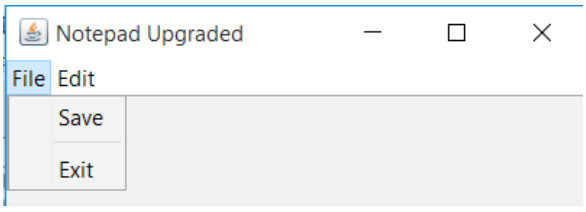

W nawigatorze powinna znajdować się poniższa struktura:




Następnie znowu używając opcji `Add From Palette` dodam do menu `File` dwie opcje: `Save` i `Exit`. Teraz wybieram `MenuItem` czyli pojedynczą pozycję w menu. Gdybym potrzebował kolejnego rozwijanego podmenu użyłbym komponentu `Menu`. Dodatkowo nowe dwie opcje rozdzielę `Separator`em, dodajemy go analogicznie do poprzednich komponentów:




Po uruchomieniu aplikacji mogę już rozwinąć moje menu:




Podpięcie zdarzenia pod `MenuItem` sprowadza się do dodania metody `ActionPerformed`.

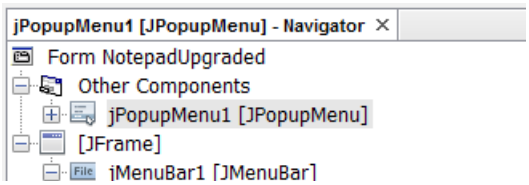
```
private void jMenuItem2ActionPerformed(java.awt.event.ActionEvent evt) {
    System.exit(0);
}
```

Komenda `System.exit(0)` oznacza zakończenie wykonywania całego programu.

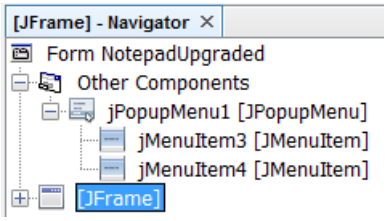
Dodawanie `JCheckBoxMenuItem` i `JRadioButtonMenuItem` wygląda tak samo jak dodawanie analogicznych kontrolek poza `MenuBar`. `JRadioButton` dalej potrzebuje przyporządkowania do `ButtonGroup`.

### Pop-up menu

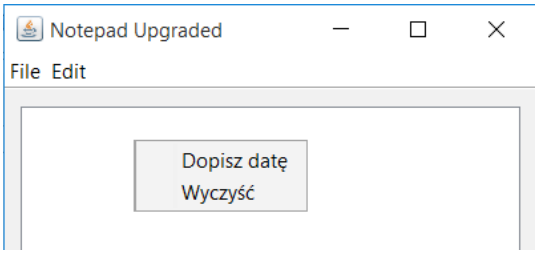
W Swingu `Pop-up menu` nazywamy po prostu menu kontekstowe wyświetlane po kliknięciu prawym przyciskiem myszy na danym komponencie. Dodajemy je przeciągając pozycję `Pop-up menu` z palety do `JFrame`. Dodałem też, `JTextArea` do okna, aby obsługiwać wieloliniowe teksty – obsługa `JTextField` i `JTextArea` nie różni się od niczym. W nawigatorze pojawi się odpowiedni wpis:




Następnie do `JPopupMenu` dodaję dwa `JMenuItem`:




Następnie w `Properties` mojego `JTextArea` odnajduję pozycję `componentPopUpMenu` i ustawiam je na dodane przed chwilą `JPopupMenu`. Po odpaleniu programu menu kontekstowe będzie już działać:




Oczywiście do takiego menu mogę dodać bardziej złożone podmenu, a nawet `CheckBox`'y czy `RadioButton`'y. Implementacja metody `ActionPerformed` dla obu `JMenuItem`:

```
private void jMenuItem3ActionPerformed(java.awt.event.ActionEvent evt) {
    jTextArea1.append( new Date().toString() );
}

private void jMenuItem4ActionPerformed(java.awt.event.ActionEvent evt) {
    jTextArea1.setText("");
}
```

## JOptionPane – okienka dialogowe

Omówimy teraz wyskakujące okienka w aplikacjach – czyli okienka dialogowe. Wszystkie dialogi w Swing'u reprezentowane są przez klasę `JOpionPane` i jej metody statyczne. Napiszemy niedługo formularz logowania oparty niemal wyłącznie o okna dialogowe – w przypadku nieprawidłowych danych również pokaże się stosowny dialog.

Najprostsze okno dialogowe uruchamiamy statyczną metodą `showConfirmDialog()`:

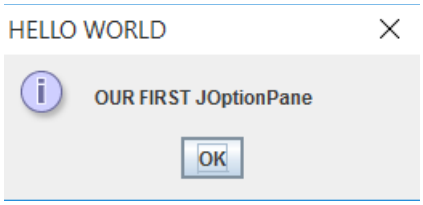
```
public static void main(String[] args) {

    JOptionPane.showMessageDialog(null, "OUR FIRST JOptionPane", "HELLO WORLD",
                                  JOptionPane.INFORMATION_MESSAGE);
}
```

**Rysunek 103 - Uruchomienie okna dialogowego**

Jak widzisz, do wyświetlenia dialogu nie potrzebuję nawet `JFrame` ☺. Omówię teraz po kolei każdy argument powyższej metody. Pierwszy to komponent rodzicielski – gdybym użył `JOptionPane` w obrębie `JFrame` podałbym tu odwołanie do niego. Wtedy, w momencie wyświetlenia dialogu `JFrame` zostałby zablokowany. Nic nie stoi na przeszkodzie, aby podać tu wartość `null`. Kolejny argument to treść wiadomości na okienku, po nim podajemy tytuł. Ostatni argument to liczba reprezentująca ikonę wyświetlaną na okienku. Mamy tu do wyboru poniższe opcje:

- `JOptionPane.ERROR_MESSAGE`
- `JOptionPane.INFORMATION_MESSAGE`
- `JOptionPane.WARNING_MESSAGE`
- `JOptionPane.QUESTION_MESSAGE`
- `JOptionPane.PLAIN_MESSAGE`

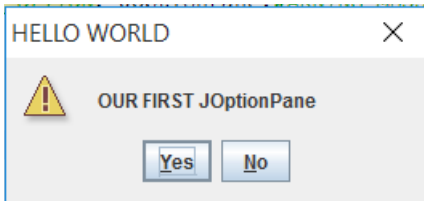
Uruchomienie powyższego programu da efekt jak na obrazku:




Metoda `showMessageDialog()` jest typu `void`, istnieje również jej bliźniacza wersja `showConfirmDialog()` zwracająca wartość `int`, która jest uzupełniona o jeszcze jeden argument – zestaw opcji do wyboru.

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

    int wynik = JOptionPane.showConfirmDialog(null, "OUR FIRST JOptionPane", "HELLO WORLD",
                                  JOptionPane.YES_NO_OPTION, JOptionPane.WARNING_MESSAGE);
    System.out.println( "## Wciśnięto opcję: " + wynik );
}
```

W wyniku wykonania programu wyświetlone zostanie okno z taką ikonką ostrzeżenia, ale będą w nim opcje `Tak`, `Nie` po kliknięciu których do zmiennej `wynik` zostanie wpisana odpowiednia wartość.



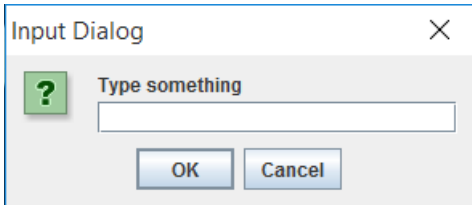

Oraz output po kliknięciu `Nie`.

```
Run (HRAwesomeManager) × GlassFish Server 4.1
run:
## Wciśnięto opcję: 1
BUILD SUCCESSFUL (total time: 2 seconds)
```

Jeśli chodzi o zestawy opcji w oknach dialogowych mamy do wyboru poniższe stałe:

- `JOptionPane.DEFAULT_OPTION`
- `JOptionPane.YES_NO_OPTION`
- `JOptionPane.YES_NO_CANCEL_OPTION`
- `JOptionPane.OK_CANCEL_OPTION`

Ostatnim z okienek dialogowym jest `InputDialog` wyświetlanym przez metodę `showInputDialog()`, metoda zwraca obiekt `String` dlatego, że w oknie wyświetlane jest pole tekstowe do wpisania wartości, którą zwróci metoda:




Powyższe okno jest wynikiem poniższego wywołania metody `showInputDialog()`.

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

    String wynik = JOptionPane.showInputDialog(null, "Type something",
                                "Input Dialog", JOptionPane.QUESTION_MESSAGE);
    System.out.println( "## wpisano: " + wynik );
}
```

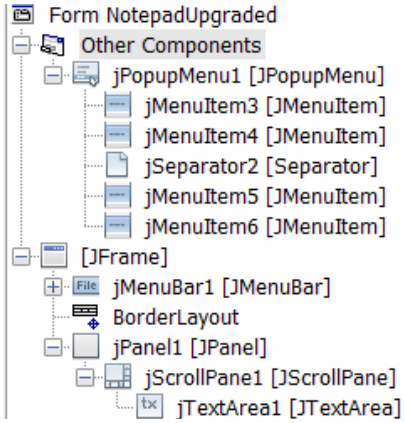
Po kliknięciu OK w oknie uzyskuję spodziewany output:

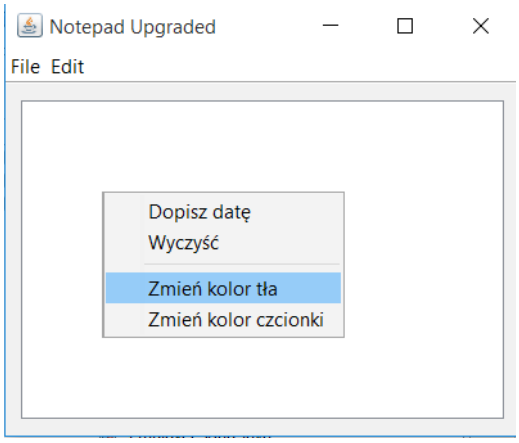
```
run:
## wpisano: Hello to you
BUILD SUCCESSFUL (total time: 6 seconds)
```

## Choosery konstruowanie obiektów przez okna dialogowe – ColorChooser i FileChooser

Z okien dialogowych często korzysta się jako `Chooserów` – np. do wybrania pliku, który chcemy edytować, albo wybranie rodzaju czcionki, którą chcemy zastosować do jakiegoś pola tekstowego. W Swingu istnieją gotowe komponenty tego rodzaju do wybierania koloru i przeglądania zawartości dysku. Wykorzystamy je w mikro-aplikacji, którą pisaliśmy niedawno czyli prostym edytorze tekstowym. Na początek zapewnimy możliwość zmiany koloru i tła `JTextArea` z poziomu menu kontekstowego (`JPopUpMenu`).

Wrócimy zatem do `JFrame` z edytorem tekstowym i dodamy do pop-up menu dwie nowe pozycje rozdzielając je separatorem.

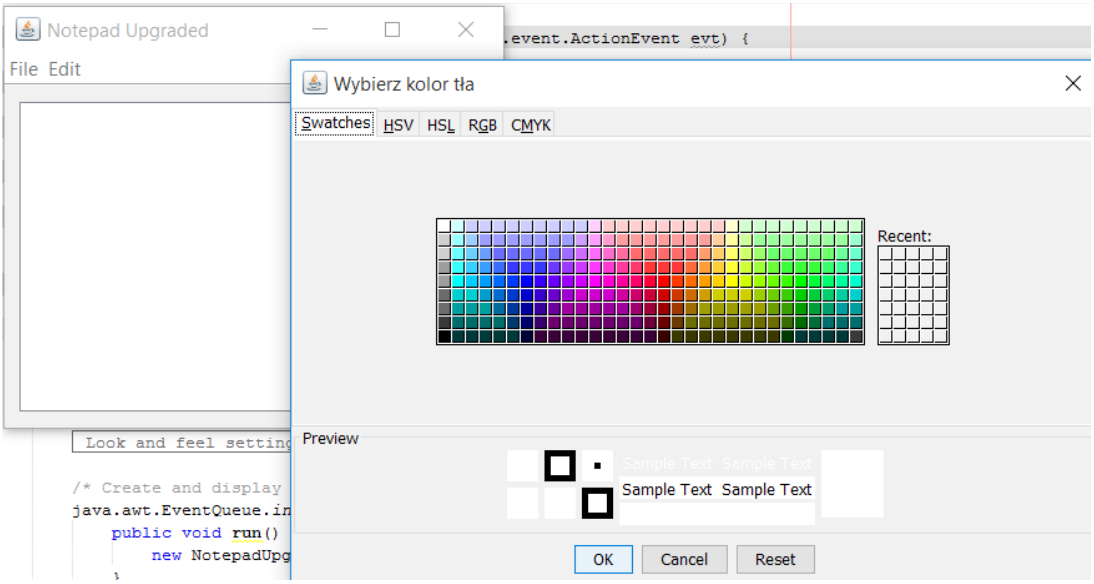


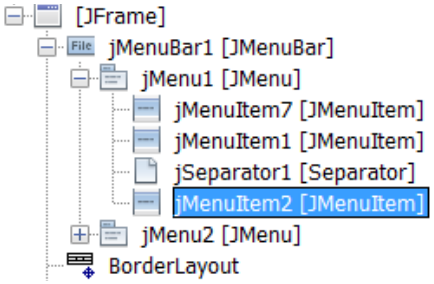

Metoda `ActionPerformed` do obsługi kliknięcia na `JMenuItem` jest bardzo prosta.

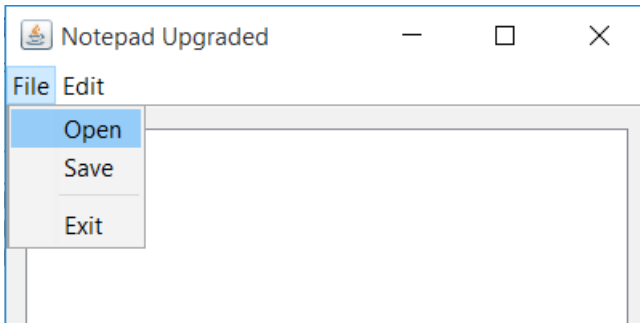
```
private void jMenuItem5ActionPerformed(java.awt.event.ActionEvent evt) {
    Color bg = JColorChooser.showDialog(this, "Wybierz kolor tła",  jTextArea1.getBackground() );
    jTextArea1.setBackground( bg );
}
```

Wyświetlenie metoda `showDialog()` przyjmuje trzy argumenty: pierwszy to standardowy dla okien dialogowych komponent nadrzędny (`this` reprezentuje tu obiekt `JFrame`), kolejny argument to tytuł okna, a trzeci to kolor, który ma zostać automatycznie wybrany po wyświetleniu okna. Poniżej efekt kliknięcia na oprogramowaną pozycję w menu kontekstowym:




Zapewnimy teraz możliwość otwierania plików w naszym edytorze. Dodamy pozycję do menu `File`, po kliknięciu której wyświetli się okno dialogowe pozwalające na wybranie pliku. Wybór plików obsługujemy za pomocą klasy `JFileChooser`. Zaczynamy oczywiście od dodania opcji w menu:




Obsługa metody `ActionPerformed` jest troszkę bardziej skomplikowana niż w przypadku `ColorChooser`'a:

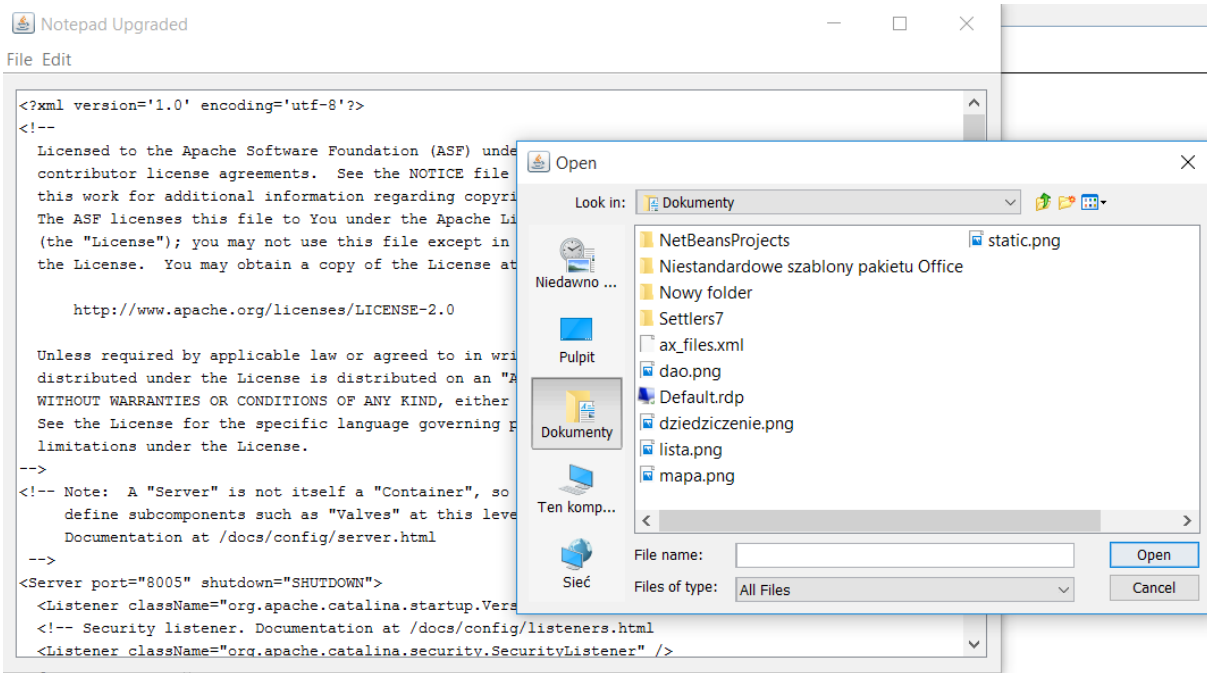
```
private void jMenuItem7ActionPerformed(java.awt.event.ActionEvent evt) {
    JFileChooser choose = new JFileChooser();
    choose.showOpenDialog(this);
    choose.setFileSelectionMode( JFileChooser.FILES_ONLY );

    File f = choose.getSelectedFile();

    StringBuilder b = new StringBuilder();
    try ( BufferedReader bis = new BufferedReader( new FileReader(f) ) )  {
        String line;
        while ( (line = bis.readLine()) != null ) {
            b.append(line).append("\n");
        }
        jTextArea1.setText( b.toString() );
    }
    catch (FileNotFoundException ex) {}
    catch (IOException e) {}

}
```

`JFileChooser` nie zawsze jest używany jako okno dialogowe, ale istnieje taka możliwość. Dlatego

taki dialog nie jest reprezentowany przez metodę statyczną (ponadto obiekt `JFileChooser` powinien być polem, a nie zmienną, ale chciałem pokazać cały kod w jednym miejscu). Metodą `setFileSelectionMode()` określam, które elementy mnie interesują – pliki, foldery albo jedno i drugie. Mamy tu do dyspozycji poniższe opcje:

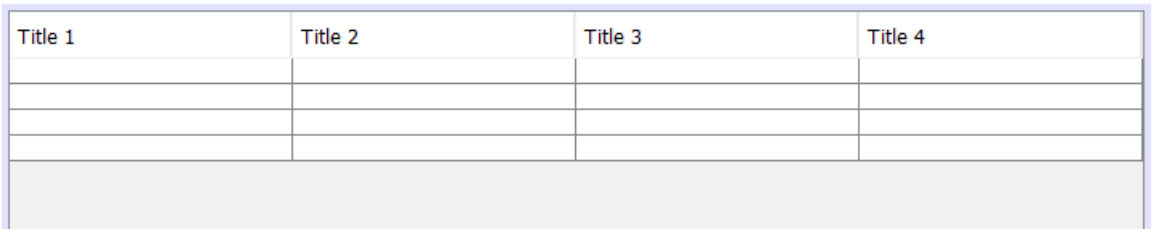
- `JFileChooser.FILES_ONLY`
- `JFileChooser.DIRECTORIES_ONLY`
- `JFileChooser.FILES_AND_DIRECTORIES`

Wybrany przez użytkownika plik wyjmuję z `Chooser`'a metodą `getSelectedFile()`. Dalsza część powinna być już jasna – odczyty zawartości plików robiliśmy już wielokrotnie ☺. Na koniec już tylko screen z wykonania powyższego programu.




## JTable – zwieńczenie interfejsu graficznego w Swing

`JTable` – czyli tabela, jest jednym z najbardziej złożonych bytów w całej bibliotece Swing. Podobnie jak `JComboBox` wykorzystuje obiekt modelu, jednak jej obsługa jest dużo bardziej złożona. Aby umieścić `JTable` w oknie po prostu standardowo przeciągamy tabelkę do okna:




Jeśli zastosowałeś w `JPanel BorderLayout` uzyskasz powyższy efekt. Aby wyczyścić tabelkę musimy wybrać opcję `Customize Code` z menu kontekstowego `JTable`.

```
default code  jTable1 = new javax.swing.JTable();

default code  jTable1.setModel(new javax.swing.table.DefaultTableModel(
                  new Object [][] {
                      {null, null, null, null},
                      {null, null, null, null},
                      {null, null, null, null},
                      {null, null, null, null}
                  },
                  new String [] {
                      "Title 1", "Title 2", "Title 3", "Title 4"
                  }
              ));

              jScrollPane1.setViewportView(jTable1);
```

Chyba widać już, gdzie znajdują się kolumny, a gdzie dane przechowywane w tabeli – wszystko znajduje się w modelu podawanym do konstruktora `JTable`. `DefaultTableModel` ma konstruktor pozwalający podać tylko kolumny tabeli i ilość wierszy:

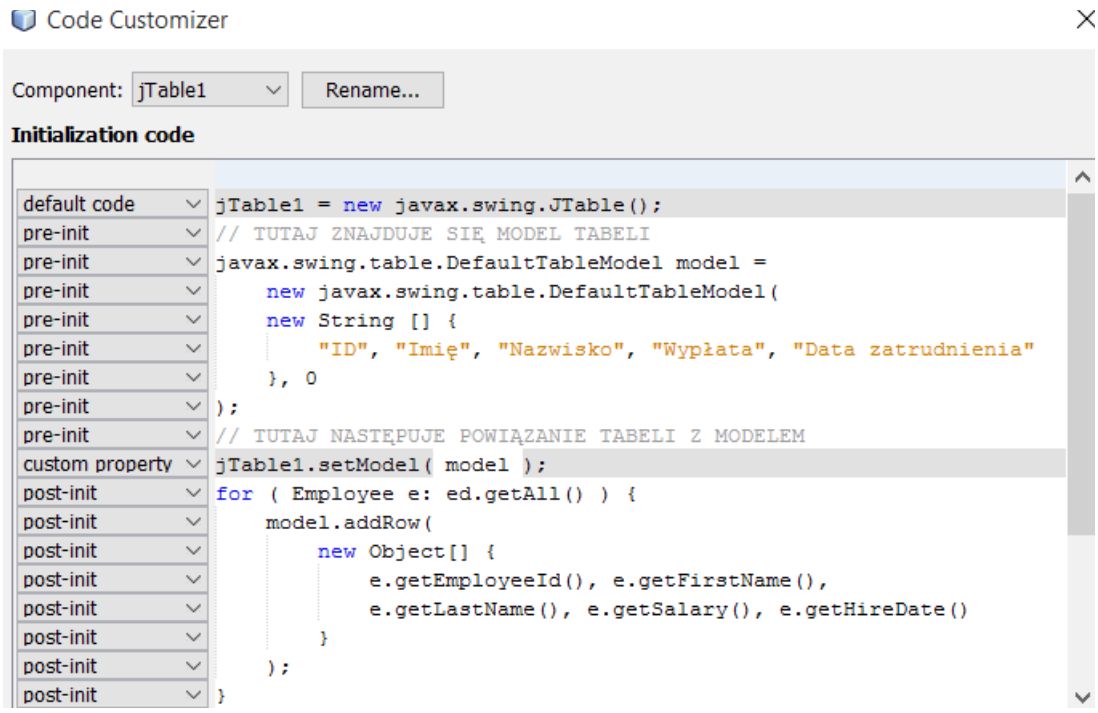
```
custom property  jTable1.setModel(new javax.swing.table.DefaultTableModel(
                     new String [] {
                         "ID", "Imię", "Nazwisko", "Wypłata", "Data zatrudnienia"
                     }, 0
                 ));

                 jScrollPane1.setViewportView(jTable1);
```

Pamiętaj o odwróceniu kolejności! Najpierw podaję nazwy kolumn jako tablicę, a później ilość wierszy.

Żeby wstawić do tabeli treść otrzymaną z bazy danych znów posłużymy się klasą dostępu do danych – `EmployeesDao`, którą wcześniej sobie przygotowałem – jej obiekt będzie oczywiście polem w mojej klasie okna:

```
    private EmployeesDao ed = new EmployeesDao();

92
93  // Variables declaration - do not modify
94  private javax.swing.JPanel jPanel1;
```

Ponownie wchodząc w opcję `Customize Code` dodaję pętlę `for-each`, która wstawi kolejne wiersze do `JTable`. Warto jednak wcześniej wyeksportować model do oddzielnej zmiennej (wiersze dodajemy do obiektu `DefaultTableModel`, a nie bezpośrednio do `JTable`):



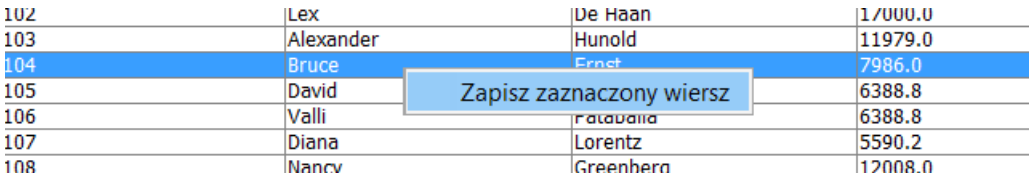

**WAŻNE!**

Tylko obiekty klasy `DefaultTableModel` mają metodę `addRow()`. Metoda `getModel()` wywołana na `JTable` zwraca obiekt `TableModel`, który jest tworem bardziej abstrakcyjnym. Żeby uzyskać dostęp do metody `addRow()` musimy użyć rzutowania.

Po uruchomieniu całego `JFrame` uzyskamy poniższy wynik:

| ID | Imię | Nazwisko | Wypłata | Data zatrudnienia |
|---|---|---|---|---|
| 100 | Steven | King | 24000.0 | 2003-06-17 |
| 101 | Neena | Kochhar | 17000.0 | 2005-09-21 |
| 102 | Lex | De Haan | 17000.0 | 2001-01-13 |
| 103 | Alexander | Hunold | 11979.0 | 2006-01-03 |
| 104 | Bruce | Ernst | 7986.0 | 2007-05-21 |
| 105 | David | Austin | 6388.8 | 2005-06-25 |
| 106 | Valli | Pataballa | 6388.8 | 2006-02-05 |
| 107 | Diana | Lorentz | 5590.2 | 2007-02-07 |
| 108 | Nancy | Greenberg | 12008.0 | 2002-08-17 |
| 109 | Daniel | Faviet | 9000.0 | 2002-08-16 |

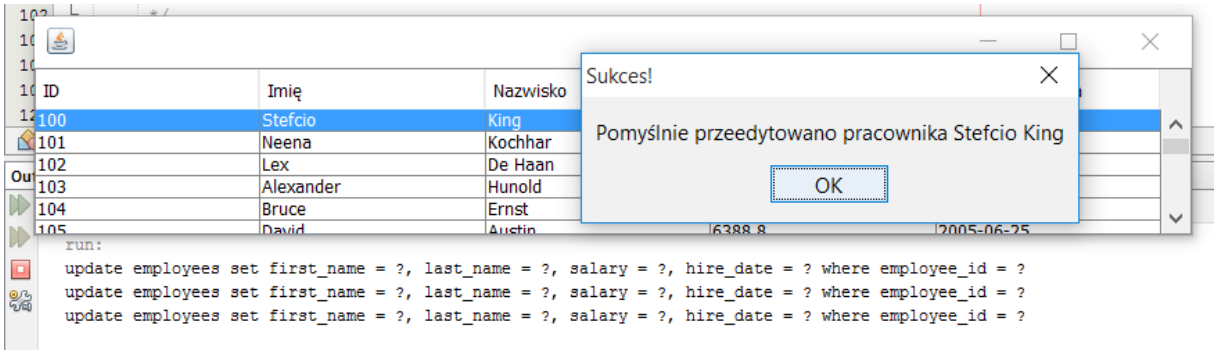
`JTable` w Swingu tworzy bardzo interaktywne tabelki, możemy bez przeszkód edytować każdą komórkę czy zaznaczać wiersze. Dodamy teraz możliwość zapisywania stanu zaznaczonego pracownika w bazie danych poprzez menu kontekstowe. Całość będzie działać tak, że użytkownik zmodyfikuje wiersz `JTable`, a do samej tabelki podepniemy menu kontekstowe. Stworzenie i podpięcie `JPopUpMenu` pomijam, omawialiśmy to wcześniej:



Obsługa `ActionPerformed` dla powyższego `JMenuItem` wyglądać będzie tak:

```
private void jMenuItem1ActionPerformed(java.awt.event.ActionEvent evt) {
    int zaznaczonyWiersz = jTable1.getSelectedRow();
    Long id = (Long)jTable1.getModel().getValueAt(zaznaczonyWiersz, 0);
    String imie = (String)jTable1.getModel().getValueAt(zaznaczonyWiersz, 1);
    String nazwisko = (String)jTable1.getModel().getValueAt(zaznaczonyWiersz, 2);
    Double wyplata = (Double)jTable1.getModel().getValueAt(zaznaczonyWiersz, 3);
    Date dataZatrudnienia = (Date)jTable1.getModel().getValueAt(zaznaczonyWiersz, 4);

    Employee e = new Employee();
    e.setEmployeeId( id );
    e.setFirstName( imie );
    e.setLastName( nazwisko );
    e.setSalary( wyplata );
    e.setHireDate( dataZatrudnienia );
    ed.updateEmployee( e );
    JOptionPane.showMessageDialog(this, "Pomyślnie przeedytowano pracownika "
                    + "" + e.getFirstName() + " " + e.getLastName(), "Sukces!", JOptionPane.PLAIN_MESSAGE);
}
```

Taaak ☺ Strasznie to skomplikowane. Pierwszą rzeczą, którą musimy zrobić to zdobyć informację o zaznaczonym wierszu metodą `getSelectedRow()` wywołaną na `JTable`. Później już tylko składam obiekt do zapisania w bazie odczytując konkretne komórki modelu `JTable` – pod spodem model wykorzystuje tablice, jak już widzieliśmy, więc czas odczytu jest w zasadzie sprowadzalny do 0. Jednak metoda `getValueAt()` zwraca `Object`, więc muszę dodatkowo zrzutować wynik na odpowiedni typ. Po złożeniu obiektu, wystarczy tylko zapisać go w bazie i wyświetlić jakieś potwierdzenie:




Dla porządku pasowałoby dodać już tylko prawidłowe parsowanie liczb i dat, a także obsługę wyjątków. Nie robiłem tu tego, aby nie zaburzać czytelności przykładu.

# Obsługa sieci w Javie

## Podstawowe klasy i pojęcia

Stworzymy prostą implementację architektury klient-serwer w Javie. Ponownie będziemy bawić się strumieniami oraz wykorzystamy klasy `Socket` oraz `ServerSocket`. Najpierw napiszemy prosty serwer, który będzie nadawał na porcie `221`, a potem dorobimy do niego odpowiedniego klienta.

## Najpierw serwer

Zacznijmy od tego, że każda operacja na klasach obsługujących gniazda sieciowe może podnieść `IOException`, więc zacząć trzeba od solidnego `try-catcha` ☺. Aby utworzyć obiekt serwerowego gniazda sieciowego, do konstruktora muszę podać numer portu, który chcę otworzyć:

```
try (
        ServerSocket sock = new ServerSocket(221);
```

Samo stworzenie obiektu typu ServerSocket nie pociąga za sobą oczekiwania na połączenie z jakimś klientem, robi to metoda `accept()` zwracająca obiekt typu `Socket`. Dopiero z tego obiektu mogę wyjąć strumienie wyjściowe i wejściowe, które posłużą mi do komunikacji z klientem:

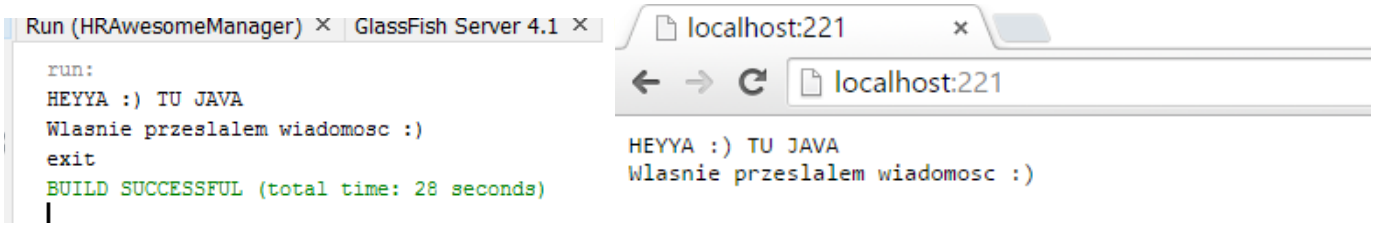
```
try (
        ServerSocket sock = new ServerSocket(221);
        Socket so = sock.accept();
        PrintWriter pw = new PrintWriter( so.getOutputStream() );
    )
{
```

Teraz mam już pełny arsenał środków do przesłania jakiegoś komunikatu. Nie robimy na razie odczytu ze strumienia wejściowego, ponieważ nie mamy jeszcze odpowiedniego klienta. Kolejne linijki to po prostu odczyt z konsoli i przesłanie wprowadzonych danych przez sieć:

```
try (
        ServerSocket sock = new ServerSocket(221);
        Socket so = sock.accept();
        PrintWriter pw = new PrintWriter( so.getOutputStream() );
    )
{
    Scanner s = new Scanner( System.in );
    String cmd;
    while ( !(cmd = s.nextLine()).equals("exit") ) {
        pw.println( cmd );
        pw.flush();
    }
    s.close();
}
catch (IOException ex) {
    ex.printStackTrace();
}
```

Nie można zapomnieć o wywołaniu `flush()` po każdym wpisaniu danych do strumienia – inaczej do klienta dane trafią dopiero w momencie zamykania strumienia. Przetestujmy nasz serwer – jako

klienta wykorzystamy zwykłą przeglądarkę internetową. Po wprowadzeniu adresu `localhost:221` będę mógł zacząć pisać kolejne linijki, które wyświetli przeglądarka:




### Potem klient

Napiszemy jakiś prosty komunikator – odpalimy dwie konsole w NetBeans i skomunikujemy dwa programy. Aby serwer mógł odczytywać komunikaty przesłane przez klienta potrzebujemy Reader'a, który będzie sprawdzał czy w strumieniu coś się znajduje i jeśli tak, to niech wyświetli na konsolę jego zawartość. Warto byłoby wydzielić odczytywanie do oddzielnego wątku:

```
public static class Odczyt implements Runnable {

    private BufferedReader in;
    public Odczyt(BufferedReader in) {
        this.in = in;
    }

    @Override
    public void run() {
        try {
            while ( true ) {
                if ( in.ready() ) {
                    System.err.println( in.readLine() );
                }
            }
        }
        catch (Exception e) {}
    }
}
```

Specjalnie do wyświetlania nie użyłem `System.out`, ponieważ wypisanie za pomocą `System.err` pokoloruje odebrane dane na czerwono. Cała obsługa odczytywania danych od klienta lub serwera sprowadza się teraz wyłącznie do dwóch linijek:

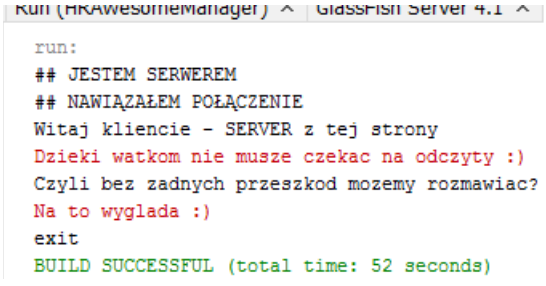
```
try (
    Socket so = new Socket("localhost", 221);
    PrintWriter out = new PrintWriter( so.getOutputStream() );
    BufferedReader in = new BufferedReader( new InputStreamReader( so.getInputStream() ) )
)
{
    System.out.println( "## NAWIĄZAŁEM POŁĄCZENIE" );
    new Thread( new Odczyt(in) ).start();
```

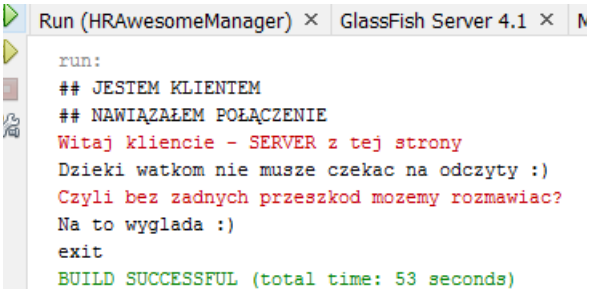
Na powyższym sample'u od razu widać jak połączenie z serwerem nawiązuje klient – tworzymy wtedy od razu obiekt `Socket` podając do konstruktora IP i numer portu. W `try-catch`'u pojawia się natomiast kolejny `resource` – `BufferedReader`, którym odczytamy dane z gniazda sieciowego.

Analogicznie wygląda modyfikacja strony serwera:

```
System.out.println( "## JESTEM SERWEREM" );
try (
        ServerSocket sock = new ServerSocket(221);
        Socket so = sock.accept();
        PrintWriter pw = new PrintWriter( so.getOutputStream() );
        BufferedReader in = new BufferedReader(
                new InputStreamReader( so.getInputStream() )
        )
    )
{
    System.out.println( "## NAWIĄZAŁEM POŁĄCZENIE" );
    new Thread( new Client.Odczyt(in) ).start();
```

Przetestujmy nasz program, uruchamiając najpierw serwer, a później klienta:



# Co dalej?

Poznałeś/aś już podstawy (no dobra, nie takie znowu podstawy ☺) programowania w Javie. Jeśli chciałbyś/chciałabyś dalej zgłębiać ten temat warto zapoznać się z poniższymi zagadnieniami:

- `Java Enterprise Edition` – w Javie można pisać również bardzo zaawansowane aplikacje i interfejsem webowym, oprogramowujemy za jej pomocą stronę serwera
- `Spring Framework` – wielowarstwowy framework wspomagający rozwój aplikacji na wszystkich warstwach, również aplikacji WEB'owych
- `Hibernate` – framework typu ORM wspomagający mapowanie obiektowo relacyjne. Pozwala programiście niemal całkowicie zapomnieć o bazie danych – cały SQL zostanie wygenerowany przez framework
- `Maven` – narzędzie do kompilacji projektów. Jego głównym atutem jest brak konieczności szukania w sieci odpowiednich bibliotek – wszystko sprowadza się do prostej edycji jednego pliku XML
- `JUnit` – biblioteka do testowania naszego oprogramowania