Java krok po kroku Documentation

Wydanie 1.0

Centrum Edukacji Obywatelskiej

June 02 2016

Spis treści

Kurs ten powstał w ramach projektu Koduj z Klasą prowadzonego przez Centrum Edukacji Obywatelskiej pod patronatem Ministerstwa Administracji i Cyfryzacji.

Materiały stanowią scenariusze 8 lekcji wprowadzających do programowania w języku Java i zakładają brak wcześniejszej znajomości tej technologii. Są one zbudowane w taki sposób, aby zajęcia miały charakter warsztatowy i w trakcie każdych z nich wykonać kilka praktycznych zadań poznając jednocześnie nowe elementy języka. **Ćwiczenia** służą do samodzielnego przećwiczenia zagadnienia wcześniej omówionego przez prowadzącego. W tematach związanych z tworzeniem graficznego interfejsu użytkownika w Javie FX w związku z mocno ograniczonym czasem lekcje zbudowane są w taki sposób, aby wraz z prowadzącym rozwijać wspólny kod, a następnie dodać samodzielnie dodatkowe funkcjonalności do programu.

Z powodu wielkości biblioteki standardowej Javy i relatywnie niewielkiej ilości czasu wiele elementów zostało celowo pominiętych.

ROZDZIAŁ 1

Język Java

Java jest technologią, która znajduje szerokie zastosowanie w informatyce. Tworzone są w niej zarówno aplikacje na komputery stacjonarne, starsze telefony komórkowe, smartfony z systemem Android, ale także dynamiczne strony WWW. Dzięki temu, że kod źródłowy kompilowany jest do kodu pośrednioego (byte-codu) a ten uruchamiany jest na wirtualnej maszynie (JVM), pozwala to uruchamiać aplikacje napisane w Javie praktycznie na dowolnym systemie operacyjnym.

Uniwersalność ta sprawia, że od wielu lat Java pozostaje w czołówce najpopularniejszych języków programowania zarówno w zastosowaniach komercyjnych jak i naukowych oraz hobbystycznych.

ROZDZIAŁ 2

Wprowadzenie i konfiguracja

Przed rozpoczęciem szkolenia zalecamy skonfigurować wszystkie komputery zgodnie z poniższą instrukcją.

2.1 Instalacja i wymagania systemowe

2.1.1 System operacyjny

Ponieważ programy w języku Java kompilowane są do kodu pośrednioego a następnie uruchamiane na wirtualnej maszynie, to programowanie w tej technologii jest tak samo wygodne na dowolnym systemie operacyjnym. Z racji swojej popularności przykłady będą jednak obrazowane na systemie operacyjnym Windows 7 (identyczne kroki należy podejmować w przypadku systemów Windows XP, czy Windows 8).

2.1.2 Java Development Kit

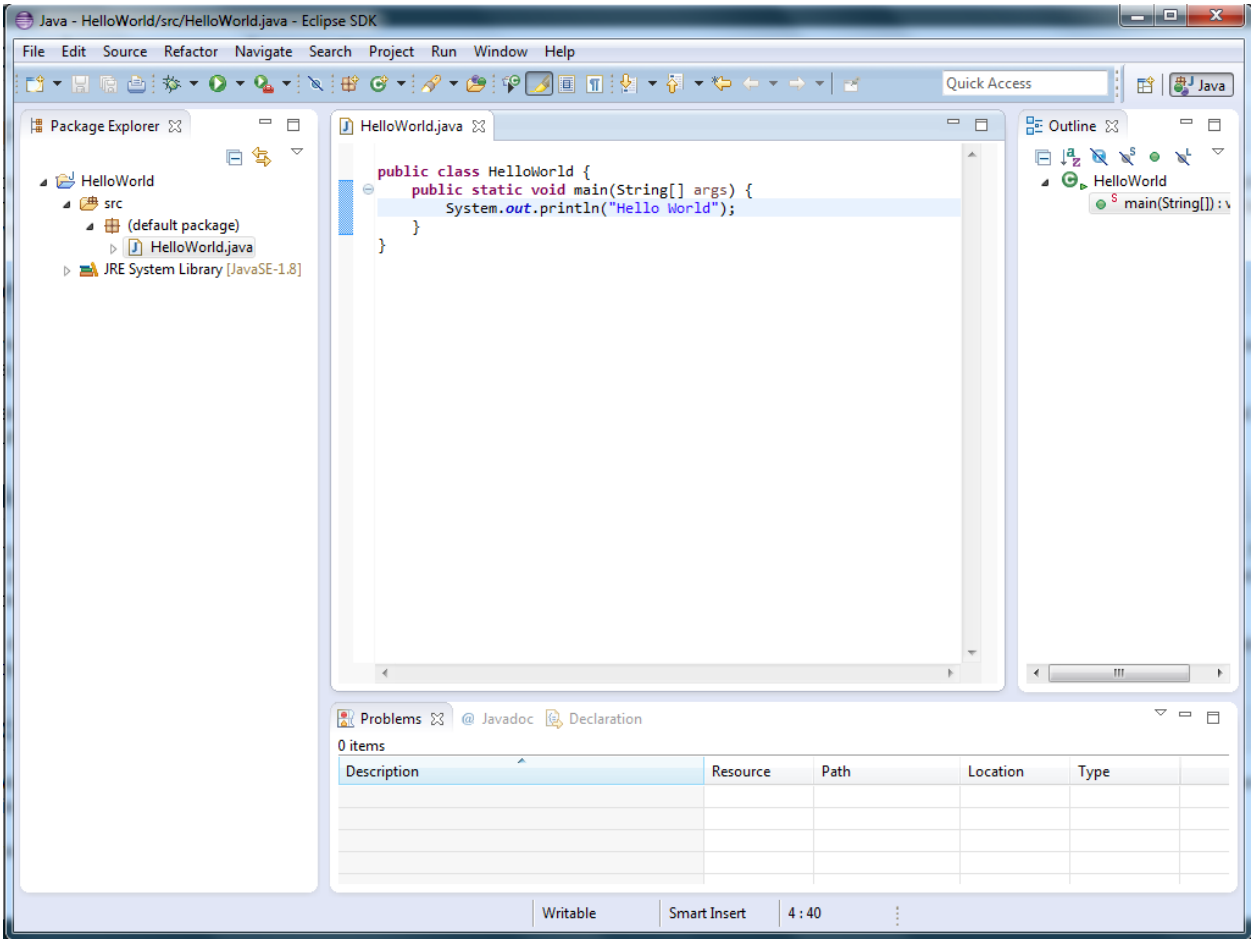
Elementem niezbędnym do kompilowania kodów źródłowych napisanych w języku Java jest zestaw narzędzi o nazwie Java Development Kit. Obecnie obowiązującą wersją konsumencką jest Java 7, jednak dla deweloperów dostępna jest już wersja 8, która wprowadza znaczące usprawnienia i to z niej będziemy korzystali.

Ze strony Oracle należy pobrać najnowszą dostępną wersję JDK w wersji odpowiedniej dla naszego systemu operacyjnego (32 lub 64 bit) oraz zainstalować ją w dowolnym miejscu na swoim komputerze.

2.1.3 Środowisko programistyczne (IDE)

Kolejnym narzędziem, które będzie wykorzystywane w trakcie kursu jest środowisko programistyczne eclipse. Na rynku istnieją także inne IDE takie jak Netbeans oraz Intellij IDEA (szczególnie popularne w zastosowaniach komercyjnych), jednak to eclipse jest najpopularniejsze, a jednocześnie proste w obsłudze i z dużymi możliwościami rozbudowy oraz co ważne - całkowicie bezpłatne.

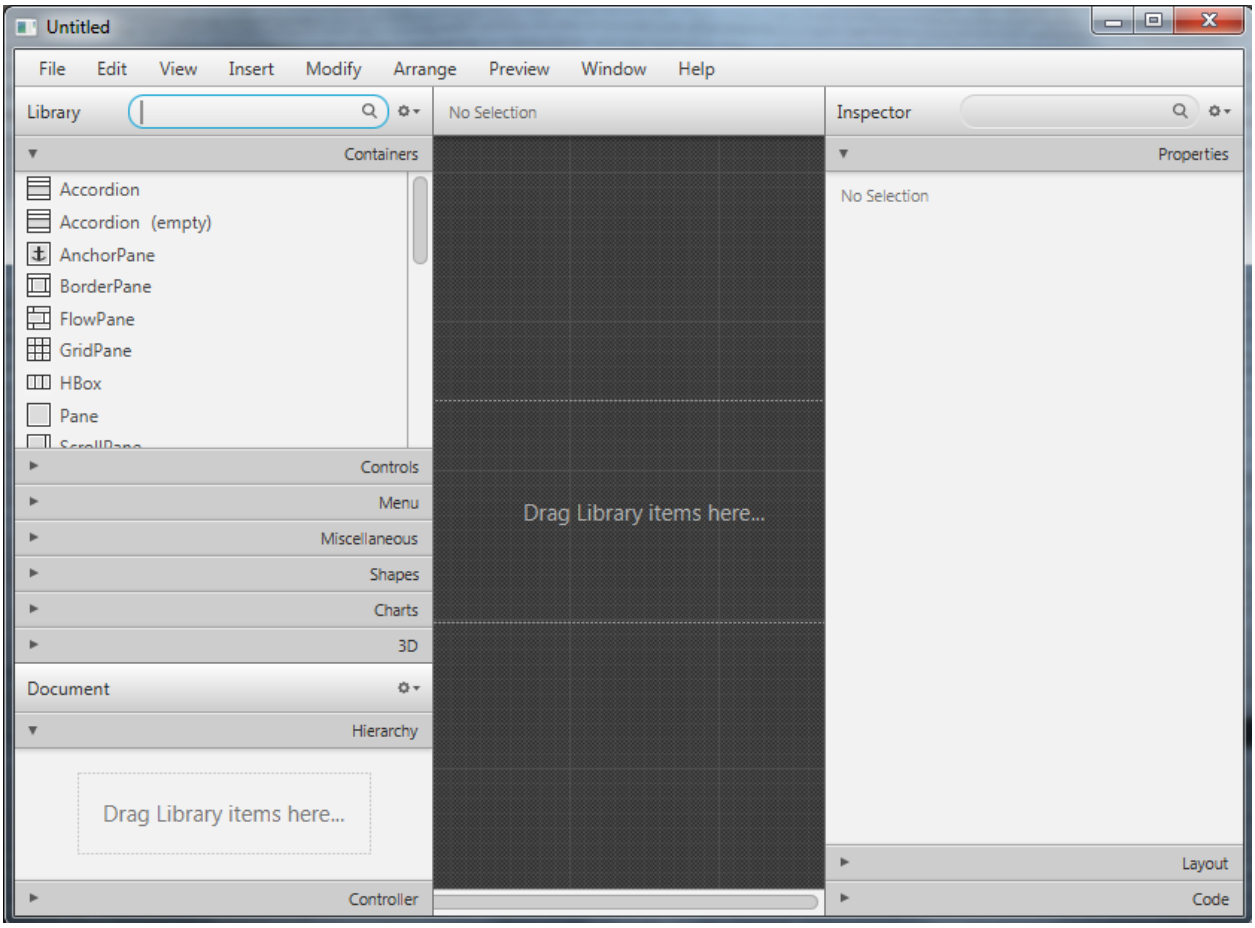
Ponieważ w dalszej części kursu będziemy tworzyli aplikacje z graficznym interfejsem użytkownika, zalecamy pobrać wcześniej przygotowane środowisko eclipse wyposażone w niezbędne wtyczki. Całość można pobrać ze strony eFXclipse w postaci archiwum zip, a następnie wypakować je w dowolnym miejscu na komputerze.



2.1.4 JavaFX Scene Builder

Rekomendowaną przez firmę Oracle technologią do tworzenia graficznego interfejsu użytkownika (GUI) aplikacji pisanych w Javie jest obecnie JavaFX. Powstało w tym celu narzędzie, które znacznie ułatwia cały proces i pozwala stworzyć interfejs za pomocą przyjaznego mechanizmu drag&drop.

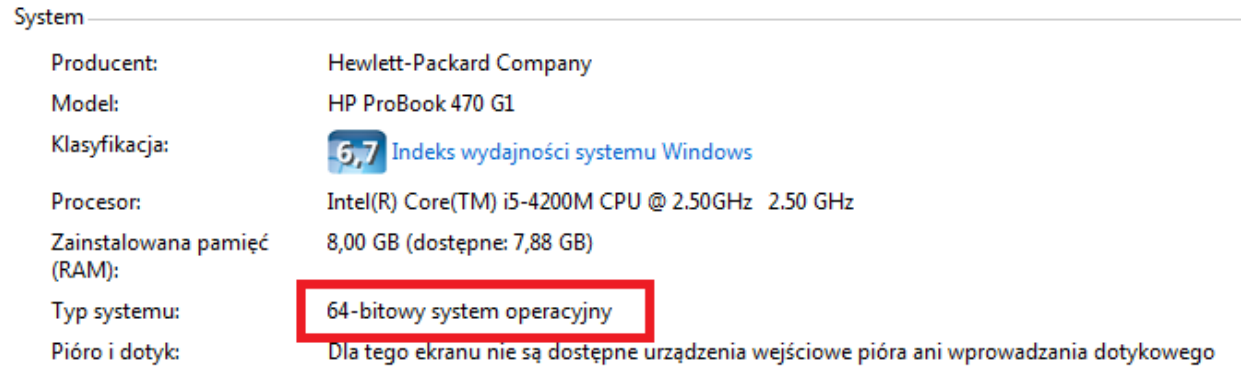
Narzędzie JavaFX Scene Builder w wersji 2.0 można pobrać ze strony Oracle.



2.1.5 Rozwiązywanie problemów

1. Sprawdzenie wersji systemu operacyjnego

W celu sprawdzenia wersji systemu operacyjnego kliknij prawym przyciskiem myszy na opcji “Komputer” i wybierz właściwości. Wśród informacji znajdziesz także wersję systemu operacyjnego (32 lub 64 bit)



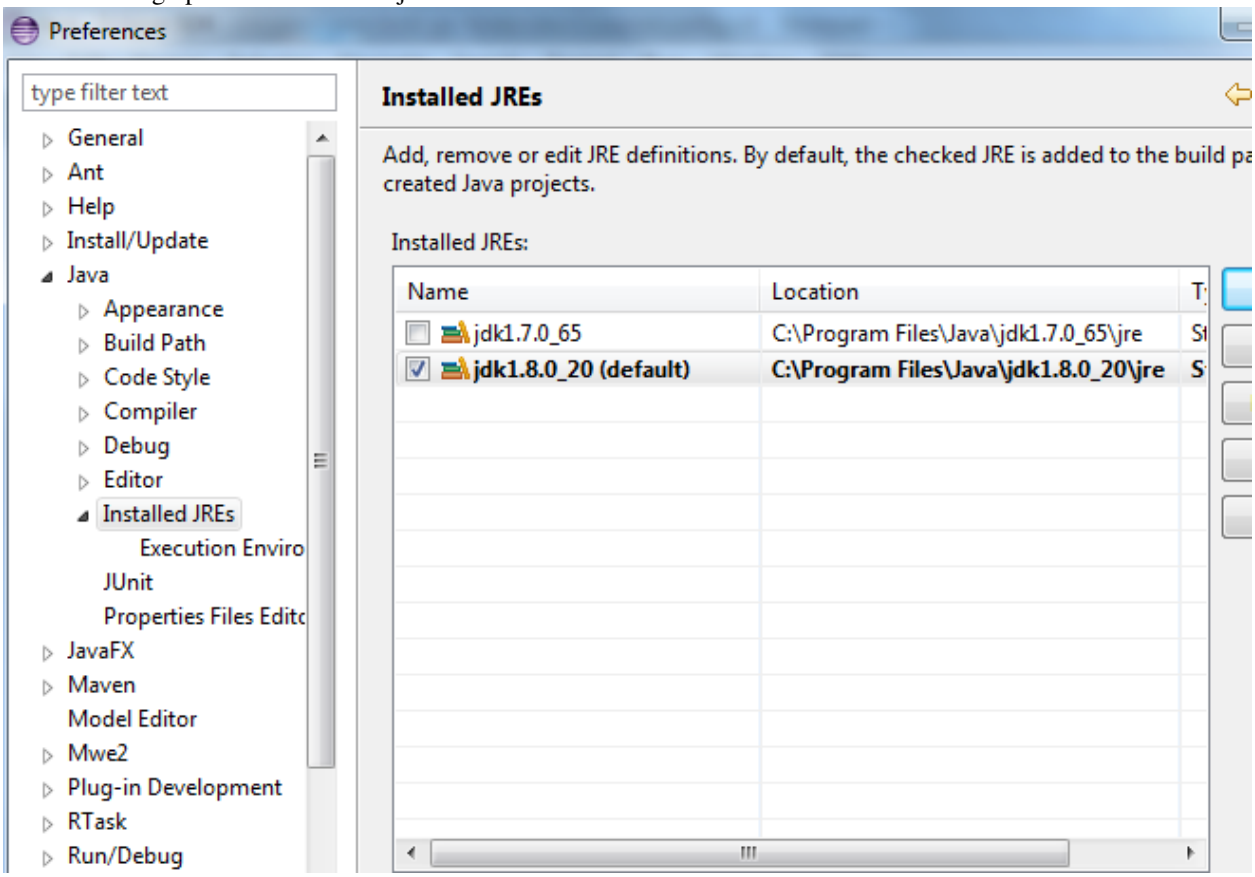
Wszystkie wcześniej wspomniane narzędzia należy zainstalować zgodnie z wersją swojego systemu operacyjnego.

2. Konfiguracja Javy 8 w przypadku kilku jej wersji na komputerze.

W przypadku, gdy na komputerze było wcześniej zainstalowanych kilka wersji Javy, eclipse może domyślnie korzystać

z jej starszej wersji (np. 7), co na późniejszym etapie uniemożliwi nam tworzenie aplikacji z graficznym interfejsem użytkownika w Javie FX. W celu sprawdzenia i skonfigurowania odpowiedniej wersji Javy w eclipse należy przejść do opcji Window -> Preferences -> Installed JREs.

Jako domyślne środowisko uruchomieniowe powinno być to oznaczone jako 1.8.0 (z ewentualnym dopiskiem o wersji aktualizacji). Jeżeli tak nie jest należy wybrać opcję Add -> Standard VM a następnie wskazać główny folder zainstalowanego pakietu JDK w sekcji JRE home.



Powyżej widać, że skonfigurowana jest zarówno Java 7 jak i 8, ale wersja 8 ustawiona jest jako domyślna

3. Eclipse po uruchomieniu pokazuje błąd związany z wtyczką Mercurial (system kontroli wersji).

Błąd należy zignorować, nie powinien pojawiać się przy kolejnych uruchomieniach środowiska.

ROZDZIAŁ 3

Kurs

Cały kurs podzielony jest na 8 lekcji.

3.1 Wprowadzenie do języka Java

W lekcji tej dowiesz się:

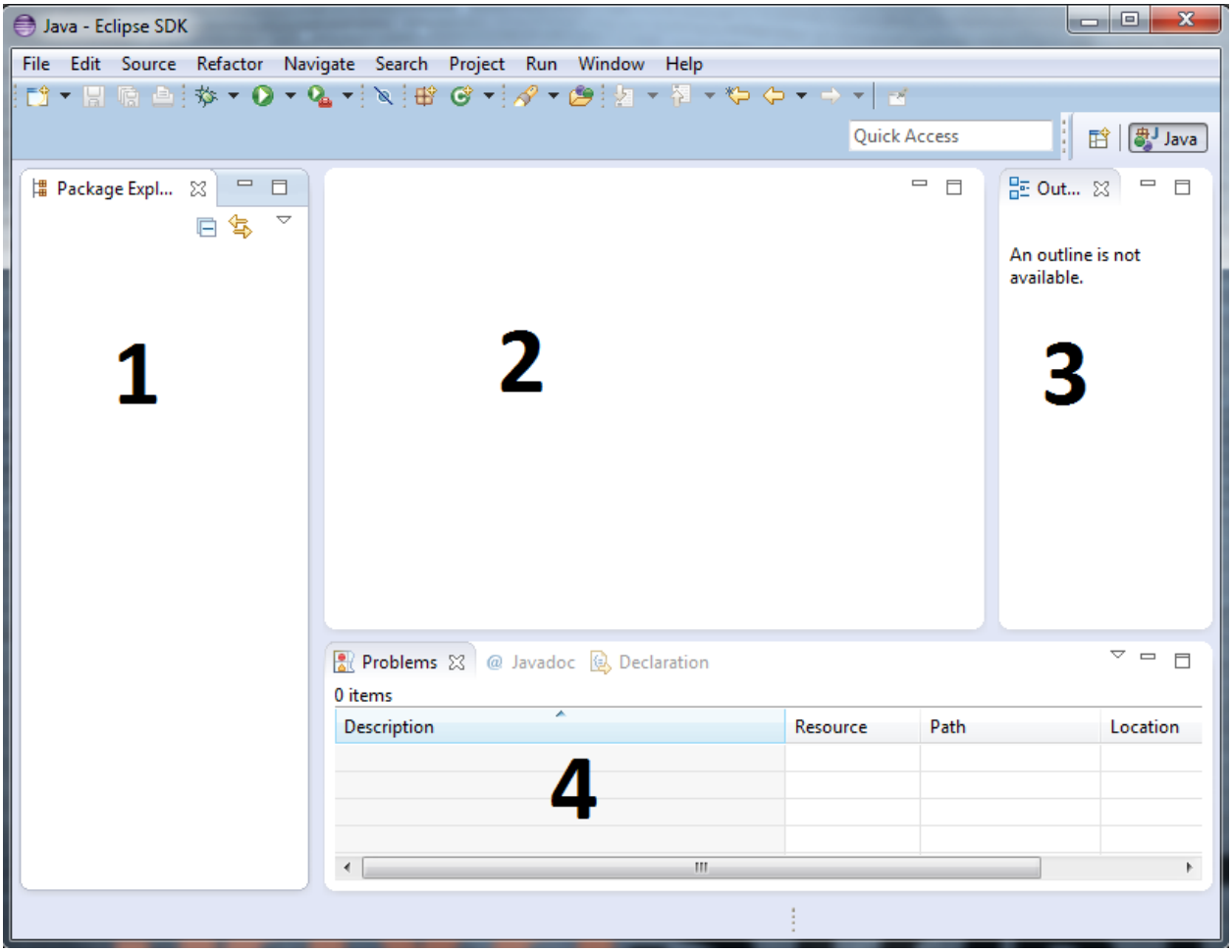
- jak używać środowiska eclipse
- co powstaje w wyniku kompilacji kodu źródłowego napisanego w języku Java
- jak uruchamiać swoje programy
- czym są zmienne
- jakie są podstawowe typy danych w Javie
- jak wykonywać podstawowe operacje arytmetyczne

Podczas lekcji nauczyciel powinien krótko omawiać daną sekcję pod względem teoretycznym, a następnie omawiać prezentowane przykłady, prosząc z wyprzedzeniem uczniów o sugestie tego jak zapisać dalszy fragment kodu źródłowego. Uczniowie powinni pisać kod jednocześnie na swoich komputerach, aby móc obserwować wynik działania aplikacji. Lekcja kończy się zadaniem do samodzielnego wykonania, które należy wykonać na zajęciach lub dokończyć samodzielnie w domu.

3.1.1 Eclipse IDE

Eclipse to najpopularniejsze środowisko programistyczne wśród programistów Java. Jego głównymi zaletami z punktu widzenia osoby początkującej jest wykrywanie błędów w trakcie pisania kodu a także podpowiadanie składni wraz z możliwością generowania i prostej modyfikacji (refaktoryzacji) kodu źródłowego. Wszystko to składa się przede wszystkim na znaczną oszczędność czasu.

Przy pierwszym uruchomieniu środowiska zostaniemy zapytani o wskazanie folderu “workspace” - jest to folder, w którym przechowywane będą tworzone przez nas projekty. Zalecamy zostawić tę lokalizację domyślną.

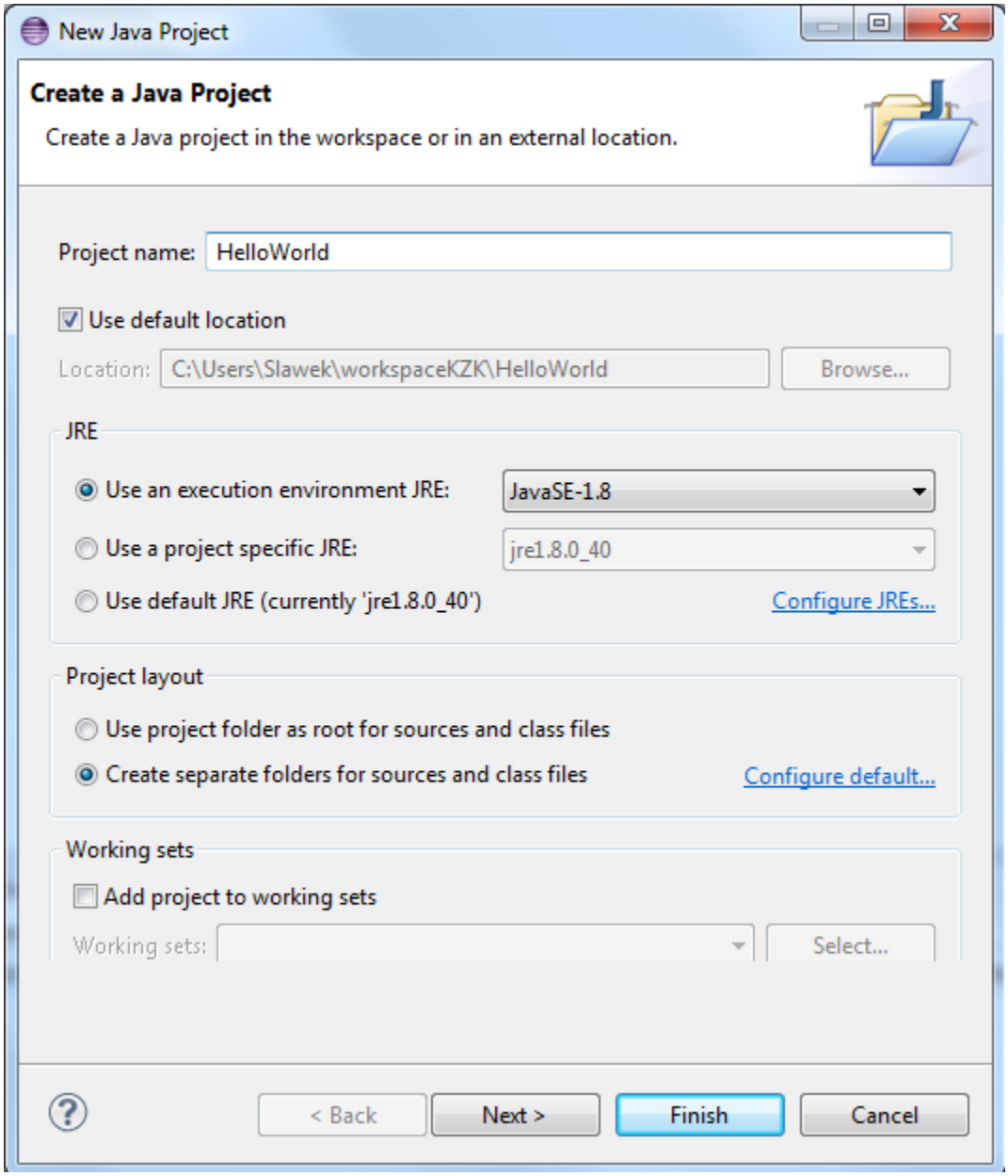


Na powyższym zrzucie ekranu widać domyślny widok, który zastaniemy po uruchomieniu środowiska eclipse. Można w nim wyróżnić 4 główne obszary:

1. Package Explorer - w tym miejscu będziemy widzieli wszystkie projekty, które zapisane są w folderze **workspace** oraz ich strukturę w postaci rozwijanego drzewa
2. Obszar oznaczony numerem 2 to główna część robocza - w tym miejscu będziemy edytowali kod źródłowy aplikacji
3. Outline - to skrótowy podgląd danego pliku i elementów w nim zawartych (zmienne, metody/funkcje)
4. W dolnej części ekranu znajduje się kilka zakładek. Najważniejsza z nich to **Problems**, która pokazuje wszelkie błędy i ostrzeżenia występujące w kodzie źródłowym. W tym miejscu zobaczymy także dodatkową zakładkę **Console** z wydrukami generowanymi przez nasze aplikacje.

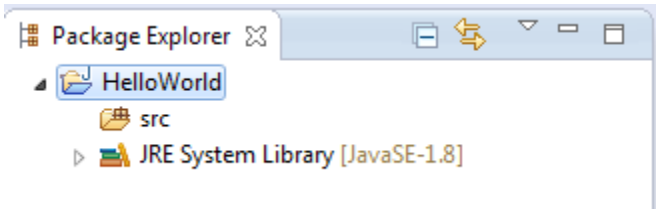
3.1.2 Pierwszy Projekt

W celu utworzenia nowego projektu wybieramy opcję **File -> New -> Java Project**



Wpisujemy dowolną nazwę projektu w polu **Project name** a także wybieramy wersję maszyny wirtualnej (JRE), na jakiej ma być uruchomiony nasz program (domyślnie JavaSE-1.8). Klikamy Finish.

W obszarze Project Explorer pojawi się nowo utworzony projekt:



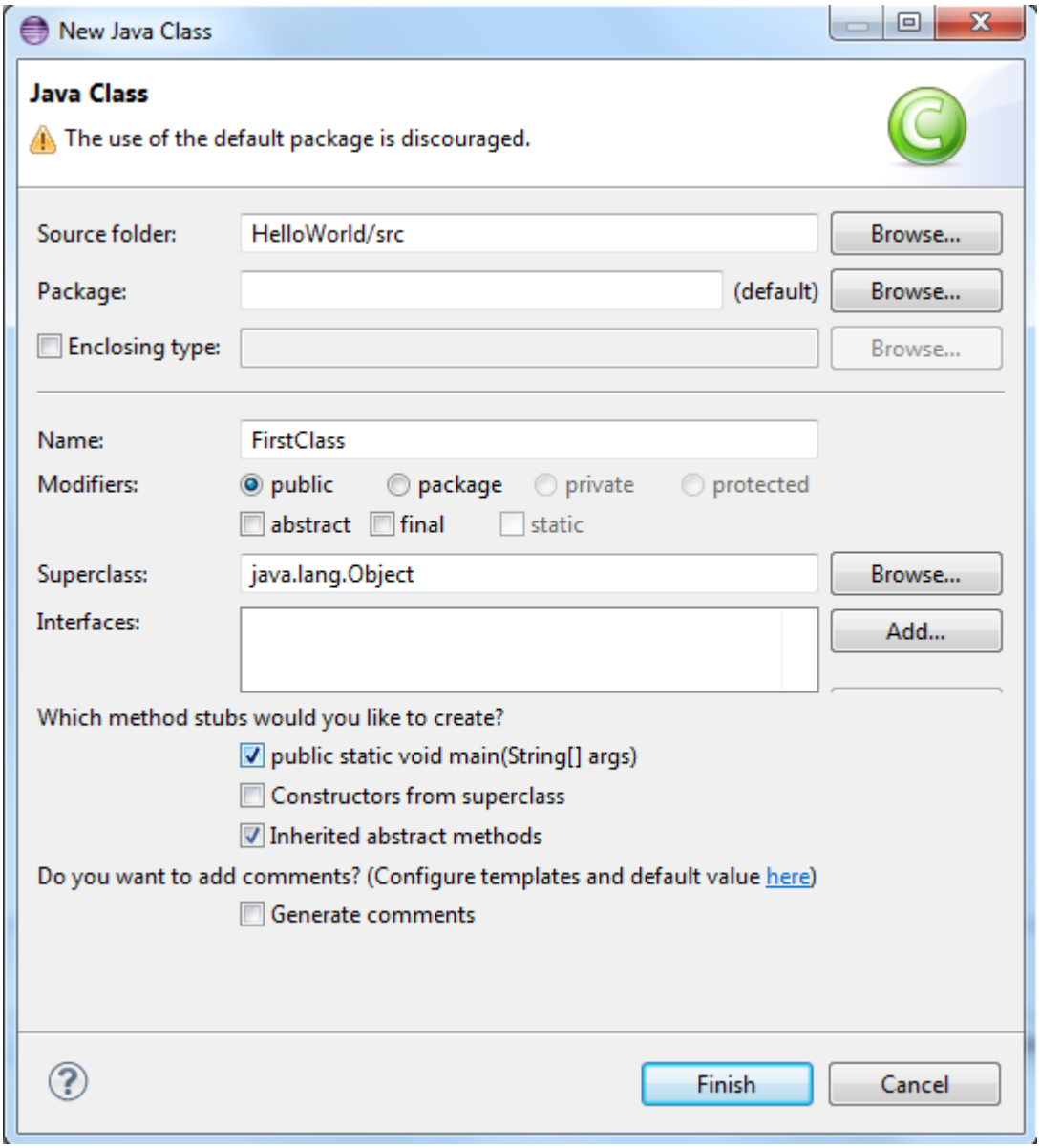
Widzimy tu folder **src**, w którym umieszczane będą pliki z kodem źródłowym, a także dołączoną wirtualną maszynę, na której nasz projekt będzie uruchamiany.

Teraz należy utworzyć plik, w którym będziemy edytowali kod źródłowy. W dalszej częściu kursu będziemy mówili

krótko o tworzeniu nowej **klasy**. Klasa jest pojęciem związanym z programowaniem obiektowym, które będzie głównym zagadnieniem kolejnej lekcji, na chwilę obecną przyjmijmy po prostu, że w klasie umieszczamy kod naszego programu.

Kliknij prawym przyciskiem na folderze src i wybierz opcję **New -> Class**. Można także posłużyć się wygodnym skrótem i skorzystać z przycisku z symbolem C, który znajdziemy na górnym pasku nawigacyjnym.

W kreatorze klasy wymagane jest podanie jedynie nazwy klasy. My jednak zaznaczymy także opcję przy **public static void main(String[] args)**.



Informacja: Nazwy klas rozpoczynaj zawsze wielką literą a jeżeli nazwa składa się z kilku wyrazów to je także rozpoczynaj wielką literą, np. NazwaTwojejKlasy albo ThisIsMyClass. Pamiętaj także, że nazwa klasy musi być identyczna z nazwą pliku, w którym ta klasa się znajduje (eclipse zadba o to automatycznie).

W naszym przypadku nazwą klasy jest **FirstClass**.

Uwaga: Zapamiętaj, że w Javie nazwy klas i wielkość używanych liter, mają znaczenie. "NazwaKlasy" i "nazwaKlasy" będą potraktowane jako dwa zupełnie różne elementy.

W utworzonej przez nas klasie został wygenerowany następujący kod źródłowy, do którego dopisaliśmy także dodatkowy wiersz *System.out.print("Witaj Świecie!");* :

plik FirstClass.java

```
1 public class FirstClass {
2
3   public static void main(String[] args) {
4     // TODO Auto-generated method stub
5     System.out.print("Witaj Świecie!");
6   }
7
8 }
```

W linii 1 widzimy definicję klasy. Klasę definiujemy za pomocą słowa kluczowego **class**, po którym następuje jej nazwa. Eclipse automatycznie oznacza ją jako publiczną za pomocą słowa kluczowego **public**, co najprościej można wytłumaczyć jako możliwość jej wykorzystania z dowolnego miejsca w naszym projekcie.

W wierszu 3 widzimy zapis *public static void main(String[] args)*, czyli publiczną, statyczną metodę o nazwie **main**, która nie zwraca żadnego wyniku (void). Element umieszczony w nawiasie, czyli *String[] args* to argument metody main w postaci tablicy - więcej na ten temat powiemy w dalszej części lekcji.

Informacja: Zapamiętaj, że od metody main rozpoczyna się działanie każdej aplikacji napisanej w języku Java. Jeżeli nie zdefiniujesz takiej metody w swojej klasie, to nie będziesz w stanie uruchomić programu.

Zauważ, że zarówno definicja klasy jak i metody rozpoczyna się i kończy nawiasami klamrowymi. Nawiasy klamrowe znacznie podnoszą czytelność kodu, szczególnie, gdy klasa składa się z kilkuset, czy nawet kilku tysięcy wierszy.

W metodzie main znajduje się jeden wiersz komentarza poprzedzony znakiem podwójnego ukośnika.

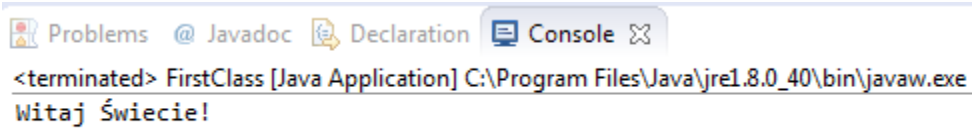
Informacja: Komentarze w języku Java można zapisywać na dwa sposoby. Jeżeli komentarz ma zajmować jedynie jeden wiersz, poprzedź go znakiem podwójnego ukośnika. Jeżeli będzie dłuższy umieść komentarz w bloku /* komentarz */ Komentarze są pomijane w trakcie kompilacji programu i nie mają wpływu na jego działanie.

Po komentarzu dopisaliśmy także linijkę o treści **System.out.print("Witaj Świecie");**, która wydrukuje na ekranie tekst podany w nawiasie, czyli Witaj Świecie.

Informacja: Zapamiętaj, że każda linijka z wyrażeniem (np. drukowaniem tekstu) w języku Java musi kończyć się średnikiem.

W celu uruchomienia programu wybierz z górnego paska nawigacyjnego charakterystyczny przycisk z symbolem strzałki (Run) lub po prostu użyj skrótu klawiaturowego **Ctrl+F11**.

W tym momencie w dolnej części eclipse powinno się pojawić okno konsoli (Console) z wydrukiem naszego tekstu.



3.1.3 Typy danych

W Javie tak jak w praktycznie każdym języku programowania istnieją różne typy danych. **Typ danych** to opis tego co reprezentuje dana wartość. Z powodu budowy komputera rozróżnia się liczby całkowite, liczby zmiennoprzecinkowe, znaki, ciągi znaków.

- boolean - typ logiczny. Może przyjmować jedynie dwie wartości - true (prawda) lub false (fałsz).
- byte, short, int, long - typy całkowitoliczbowe. Różnią się zakresem wielkości liczby jakie mogą przechowywać (od najmniejszego do największego) - np. 1, 5, 10, 3456.
- float, double - typy zmiennoprzecinkowe o różnym zakresie (double może przechowywać większe liczby). Separatorem dziesiętnym jest kropka, np. 3.14, 276.24563.
- char - typ znakowy, reprezentuje pojedyncze litery lub znaki. Wartości tego typu umieszczamy pomiędzy znakami pojedynczego cudzysłowu, np. ‘a’, ‘&’.
- String - (pisany z wielkie litery) - specjalny typ, który służy do przechowywania ciągów znaków. Ciągi znaków zapisujemy pomiędzy podwójnymi cudzysłowami, np. “Ania”, “Jakiś dowolny tekst”.

Ćwiczenie (5 min)

Napisz program, który wydrukuje na ekranie następujące wartości (wykorzystaj różne typy danych):

```
245
123.456
a
Java jest cool
```

plik DataTypes.java

```
public class FirstClass {
  public static void main(String[] args) {
    System.out.println(245);
    System.out.println(123.456);
    System.out.println('a');
    System.out.println("Java jest cool");
  }
}
```

3.1.4 Zmienne

Drukowanie danych, na których nie możemy wykonywać żadnych działań, czy po prostu zapamiętać w pamięci komputera nie byłoby zbyt użyteczne. Na szczęście w Javie możemy tworzyć tzw. **zmienne**, czyli takie elementy, które pozwalają na przechowywanie wartości różnych typów danych. Java w odróżnieniu od np. PHP jest językiem statycznie typowanym co oznacza, że zmienna musi mieć określony typ. Jeżeli chcesz przechowywać w niej wartość zmiennoprzecinkową, to nie możesz jej zadeklarować jako int, ponieważ spowoduje to błąd kompilacji.

Tworzenie zmiennej możemy podzielić na dwa etapy:

- deklarację - w tym momencie następuje zaalokowanie pamięci w komputerze
- inicjalizację (inicjację) - w tym momencie następuje przypisanie konkretnej wartości do zmiennej

Dwa wyżej wspomniane etapy mogą być od siebie oddzielone lub też można je połączyć ze sobą.

plik Variables.java

```
public class Variables {
  public static void main(String[] args) {
    // deklaracja zmiennych różnego typu
    int x;
    double num;
    char letter;
    String napis;

    // inicjalizacja zmiennych
    x = 5;
    num = 12.67;
    letter = 'b';
    napis = "To może być bardzo długie zdanie";

    // deklaracja połączona z inicjalizacją
    int y = 15;
    String zdanie = "To jest przykładowe zdanie";
  }
}
```

Ćwiczenie (5 min)

Napisz program podobny do tego z poprzedniego zadania - wydrukuj na ekranie kilka wartości różnego typu, ale tym razem skorzystaj także ze zmiennych.

plik DataTypesVars.java

```
public class DataTypesVars {
  public static void main(String[] args) {
    int num1 = 123;
    double num2 = 567.123;
    char b = 'b';
    String name = "Jan Kowalski";

    System.out.println(num1);
    System.out.println(num2);
    System.out.println(b);
    System.out.println(name);
  }
}
```

3.1.5 Operacje arytmetyczne i logiczne

Jednymi z najważniejszych elementów w jakich wykorzystujemy komputery są obliczenia. Nie zawsze muszą to być skomplikowane rachunki matematyczne - czasami chcemy coś po prostu przesunąć o 1 piksel w prawo na ekranie (np. w grach), a innym razem zwiększyć wiek użytkownika o 1, gdy ma urodziny.

W języku Java znajdziemy wszystkie najpopularniejsze operatory arytmetyczno logiczne:

- **+, -** - dodawanie i odejmowanie liczb
- ***, /** - mnożenie i dzielenie całkowite liczb

- **%** - dzielenie modulo (reszta z dzielenia)
- **&&** - koniunkcja logiczna. Tylko PRAWDA && PRAWDA da w wyniku PRAWDA
- || - alternatywa logiczna. Co najmniej jedna składowa musi być PRAWDĄ, aby wynik całego wyrażenia był prawdą. PRAWDA||PRAWDA lub PRAWDA||FAŁSZ lub FAŁSZ||PRAWDA ale nie FAŁSZ||FAŁSZ
- **>, >=, <, <=** - porównania. Większe, większe lub równe, mniejsze, mniejsze lub równe.
- **==** - porównanie równości

Wyniki wyrażeń arytmetyczno logicznych mogą być obliczane na podstawie zmiennych lub wartości, a także przypisywane do innych zmiennych.

plik Arithmetic.java

```
public class Arithmetic {
  public static void main(String[] args) {
    int num1 = 5;
    int num2 = 3;
    //jaki jest wynik działania num1*num2 ?
    int num3 = num1 * num2;
    System.out.println(num3);

    //czy zmienna num1 jest większa od 3 ?
    boolean validate = num1 > 3;
    System.out.println(validate);
  }
}
```

Znak dodawania (+) ma również specjalne zastosowanie w przypadku ciągów znaków (typ String). Powoduje on złączenie (konkatenację) dwóch ciągów znaków i utworzenie na ich podstawie nowego napisu.

```
String napis = "Jan" + "Kowalski";
```

Ćwiczenie (10 min) Napisz prosty kalkulator. Zadeklaruj i zainicjuj dwie liczby typu zmiennoprzecinkowego a następnie wyświetl na ekranie wynik ich dodawania, odejmowania, mnożenia i dzielenia. Dodatkowo wyświetl na ekranie, czy pierwsza z liczb jest większa od drugiej, a także, czy ich iloczyn jest większy od 100. Przykładowy wydruk programu:

```
a + b = 28.3
a - b = 18.7
a * b = 112.8
a / b = 4.895833333333334
A > B ?true
A * B > 100 ? true
```

plik SimpleCalculator.java

```
public class SimpleCalculator {
  public static void main(String[] args) {
    double a = 23.5;
    double b = 4.8;

    System.out.println("a + b = " + (a + b));
    System.out.println("a - b = " + (a - b));
    System.out.println("a * b = " + (a * b));
    System.out.println("a / b = " + (a / b));
    System.out.println("A > B ?" + (a > b));
    System.out.println("A * B > 100 ? " + (a*b > 100));
  }
}
```

Zauważ, że w powyższym przykładzie konkatenacji a nawet obliczeń dokonujemy bezpośrednio w metodzie drukującej wynik na ekranie. Warto zwrócić także uwagę na to, że wartość typu String możemy łączyć z wartościami innego typu i zostaną one automatycznie dołączone do naszego napisu. Jeżeli wykorzystałeś dodatkowe zmienne do przechowywania wyników poszczególnych działań - nie jest to błędem.

3.1.6 Tablice jednowymiarowe

Zmienne nadają się świetnie do przechowywania pojedynczych wartości, jednak jeżeli w swoim programie posiadasz pewien zbiór danych, niezbędne będzie zastosowanie czegoś bardziej wygodnego. W końcu zapisywanie 100 liczb w postaci:

```
int x1 = 1;
int x2 = 2;
int x3 = 3;
//itd.
```

nie byłoby zbyt wygodne, prawda? Podstawowym elementem, który pozwala rozwiązać ten problem w programowaniu są tablice.

Informacja: Tablica to specjalny typ danych, który pozwala przechowywać duże ilości wartości tego samego typu.

Deklaracja i inicjalizacja tablic jest bardzo podobna do zwykłych zmiennych:

```
1 int[] tab = new int[5];
2 String[] words = new String[10];
```

Powyżej zadeklarowano i utworzono tablicę 5 liczb całkowitych typu int, która może przechowywać 5 wartości oraz tablicę typu String, która może przechowywać 10 napisów.

W przypadku, gdy z góry znasz wartości, którymi chcesz uzupełnić tablicę, istnieje szybki sposób na jej inicjalizację poprzez wymienienie wszystkich wartości w trakcie tworzenia tablicy:

```
int[] numbers = new int[]{1, 2, 3, 4, 5};
```

Zauważ, że w takim przypadku nie jest konieczne określanie rozmiaru tablicy w nawiasach kwadratowych, ponieważ maszyna wirtualna wywnioskuje to sama na podstawie ilości podanych elementów.

W wielu przypadkach tablica będzie tworzona na podstawie rozmiaru, który użytkownik wprowadzi np. z klawiatury i nie będziemy go znali w dalszej części kodu. W takiej sytuacji możemy jednak skorzystać z wartości **length**, którą posiada każda tablica niezależnie od tego jakiego jest typu. Poniżej przedstawiony jest kod tablicy liczb całkowitych o rozmiarze 5, jednak w przyszłości nauczymy się odbierać dane od użytkownika z klawiatury i wartość ta może być wcześniej nieznana. W takiej sytuacji w 2 wierszu możemy pobrać rozmiar tablicy poprzez właściwość *tab.length*.

```
int[] tab = new int[5];
int size = tab.length;
System.out.print(size);
```

Informacja: Zapamiętaj jednak, że właściwość length zwraca całkowity rozmiar tablicy, a nie to ile rzeczywiście elementów jest do niej wpisanych.

W celu przypisania lub odwołania się do poszczególnych komórek takich tablic należy odwołać się do nich poprzez indeksy:

plik Tabs.java

```java
public class Tabs {
    public static void main(String[] args) {
        int[] tab = new int[5];
        String[] words = new String[10];

        // tablice posiadają indeksy numerowane od 0
        tab[0] = 1;
        tab[1] = 2;

        // ale elementów nie musimy uzupełniać w określonym porządku
        words[0] = "Ala";
        words[3] = "kot";

        System.out.println("Pierwszy element tablicy tab[] = " + tab[0]);
        System.out.println("Czwarty element tablicy words[] = " + words[3]);
    }
}
```

Uwaga: Zapamiętaj, że tablice tak jak i praktycznie wszystkie inne struktury danych w Javie są indeksowane zaczynając od 0, a nie od 1.

Rozmiaru tablic niestety nie da się zmienić, więc jeżeli uznasz, że zabrakło Ci w niej miejsca, będziesz musiał utworzyć nową, większą tablicę.

Ćwiczenie (10 min)

Napisz program, w którym utworzysz tablicę 10 losowo wybranych przez siebie liczb zmiennoprzecinkowych. Wydrukuj na ekranie:

- wszystkie wartości,
- sumę wartości zapisanych na pozycjach nieparzystych tablicy(pierwszy, trzeci, piąty ... element tablicy)
- ostatni element tablicy (wykorzystaj właściwość length)

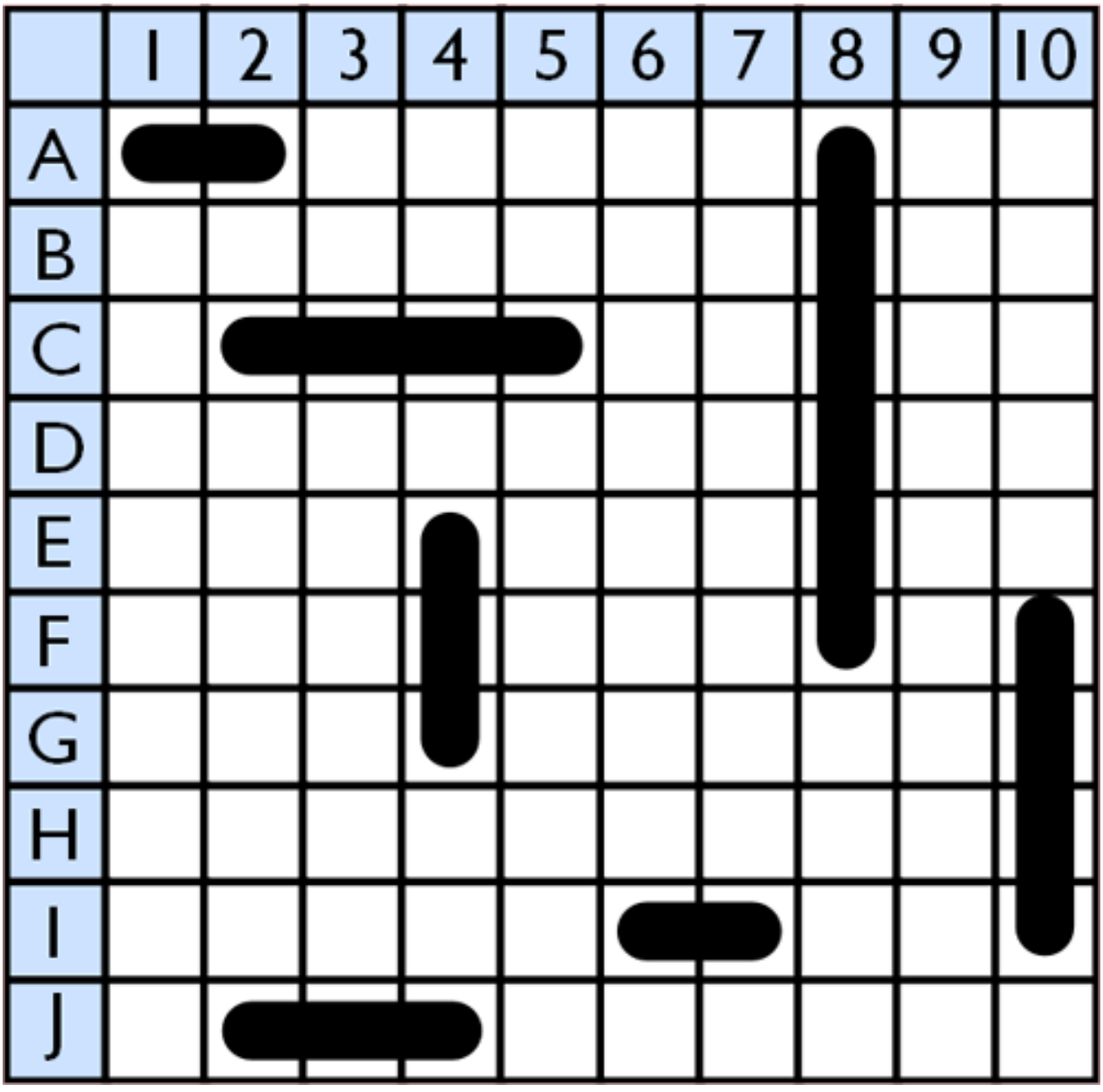
```java
public class TabCalculator {
    public static void main(String[] args) {
        double[] nums = new double[] { 2.5, 15.7, 1024.6, 33, 56.82, 1.1,
                23.90, 999.25, 550.6, 15.7 };

        System.out.println("Elementy tablicy: ");
        System.out.println(nums[0] + " " + nums[1] + " " + nums[2] + " "
                + nums[3] + " " + nums[4] + " " + nums[5] + " " + nums[6] + " "
                + nums[7] + " " + nums[8] + " " + nums[9]);

        double sum = nums[0] + nums[2] + nums[4] + nums[6] + nums[8];
        System.out.println("Suma elementów na indeksach nieparzystych: " + sum);

        System.out.println("Ostatni element tablicy: " + nums[nums.length-1]);
    }
}
```

W ćwiczeniu można było napotkać na kilka problemów. Zarówno w pierwszym jak i drugim podpunkcie należy pamiętać o indeksowaniu tablic zaczynając od 0. Do nieparzystych elementów tablicy odwołujemy się poprzez parzyste indeksy (bo zaczynają się od 0). Z kolei w trzecim punkcie należy pamiętać o tym, że właściwość length zwraca rzeczywisty rozmiar tablicy, a ponieważ indeksy numerowane są od 0, to ostatnim indeksem, do którego możemy się odwołać jest **length-1**.

Uwaga: W przypadku, gdy spróbujesz odwołać się do indeksu tablicy większego od **length-1** otrzymasz wyjątek ArrayIndexOutOfBoundsException. Jest to jeden z częściej popełnianych błędów przez młodych programistów. Może on sprawiać początkowo problemy, ponieważ jest błędem fazy wykonania aplikacji, a nie kompilacji - eclipse nie powiadomi nas więc o tym problemie w trakcie pisania kodu.

3.1.7 Tablice wielowymiarowe

Tablice jednowymiarowe znacząco usprawniają przechowywanie danych w naszej aplikacji, ponieważ nie musimy już deklarować dużej ilości zmiennych. Wyobraź sobie jednak sytuację, gdy tworzysz grę w okręty:



Możliwe, że przychodzi Ci teraz do głowy pomysł, aby wykorzystać w niej kilka tablic jednowymiarowych, które będą reprezentowały kolejne wiersze planszy. Słusznie, jednak w sytuacji, gdy będziemy chcieli utworzyć planszę o rozmiarze 20x20 komórek, niezbędne będzie zadeklarowanie 20 tablic, np.:

```
int[] w0 = new int[20];
int[] w1 = new int[20];
//...
int[] w19 = new int[20];
```

Przy tablicach jednowymiarowych stwierdziliśmy jednak, że tablica to taki typ danych, który pozwala przechowywać większe ilości wartości tego samego typu. Nic więc nie stoi na przeszkodzie, żeby w tablicy przechowywać inne tablice, a tym samym utworzyć **tablicę wielowymiarową**.

Tablicę taką najłatwiej wyobrazić sobie jako siatkę o rozmiarze x na y:

0,0	0,1	0,2				
1,0	1,1	1,2				
2,0	2,1	2,2				
						x,y

Zauważ kilka rzeczy:

- tablica wielowymiarowa nie musi mieć takiej samej liczby wierszy co kolumn
- poszczególne wiersze mogą przechowywać różne ilości elementów

Przykład:

plik MultiArray.java

```
1  public class MultiArray {
2      public static void main(String[] args) {
3          // tablica liczb całkowitych o rozmiarze 2x2
4          int[][] multiArray = new int[2][2];
5
6          // tablica liczb zmiennoprzecinkowych, która składa się z tablic o
7          // różnych rozmiarach
8          double[][] multiArray2 = new double[3][];
9          multiArray2[0] = new double[3];
10         multiArray2[1] = new double[2];
11         multiArray2[2] = new double[1];
12
13         // W wyniku tablica multiArray2 ma następującą strukturę:
14         /*
15          * XXX
16          * XX
17          * X
18          */
19
20         //lub na konkretnych liczbach:
21         int[][] multiArray3 = new int[3][];
22         multiArray[0] = new int[]{0, 1, 2};
23         multiArray[1] = new int[]{3, 4};
24         multiArray[2] = new int[]{5};
25
26         //co daje w wyniku:
27         /*
28          * 0 1 2
29          * 3 4
```

```
30            * 5
31            */
32      }
33  }
```

Ćwiczenie (10 min)

Napisz program, w którym utworzysz tablicę o rozmiarze NxN typu boolean (załóżmy maksymalny rozmiar jako 5x5). Wypełnij jej przekątną wartościami typu true a na końcu wyświetl wartość elementu przechowywanego w prawym dolnym wierzchołku tablicy. Przy wyświetlaniu wartości wykorzystaj właściwość length tak, aby po zmianie rozmiaru tablicy nie było konieczne modyfikowanie kodu wyświetlającego tę wartość.

plik Matrix.java

```
1  public class Matrix {
2      public static void main(String[] args) {
3
4          //rozmiar tablicy
5          int n = 3;
6
7          boolean[][] array = new boolean[n][n];
8
9          //uzupełniamy przekątną
10         array[0][0] = true;
11         array[1][1] = true;
12         array[2][2] = true;
13
14         /*
15          * Wyświetlamy element w prawym dolnym krańcu tablicy
16          * array[array.length-1] - indeks ostatniego wiersza
17          * array[array.length - 1].length - 1 - indeks ostatniej kolumny w ostatnim wierszu
18          */
19         System.out.println("Prawy dolny element: " + array[array.length - 1][array[array.length - 1]
20     }
21 }
```

3.1.8 Dodatek 1 - praca z eclipse

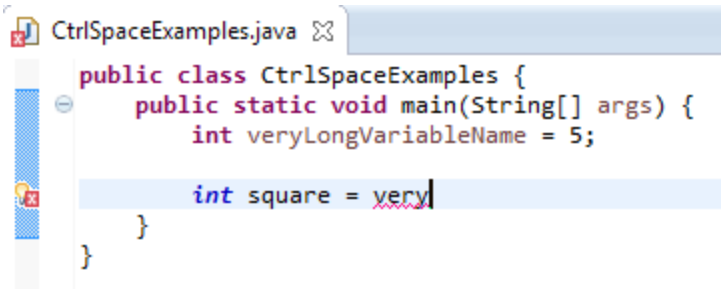
Całkiem możliwe, że już na tym etapie zacząłeś zauważać, że pisanie powtarzającego się kodu (np. nazw zmiennych, czy powtarzanie co chwilę System.out.print()) potrafi doprowadzić do lekkiej frustracji i odbiera chęci do pisania kodu "bo przecież wiadomo jak to ma wyglądać".

W tym miejscu pokażemy Ci kilka użytecznych skrótów, które w eclipse znacząco podnoszą efektywność pracy oraz oszczędzają Twój cenny czas.

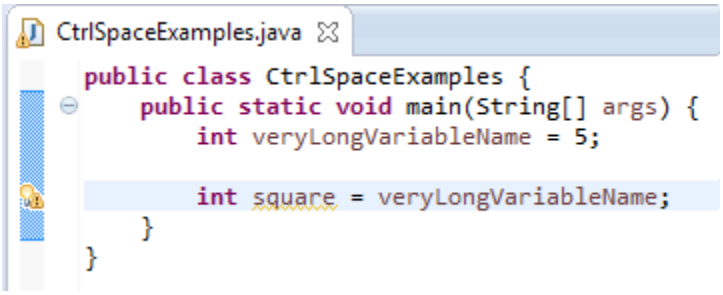
CTRL + SPACJA

Skrót, który wykorzystuje się zdecydowanie najczęściej. Pozwala na autouzupełnianie kodu i wystarczy, że wpiszesz jedynie kilka pierwszych liter zmiennej, a reszta zostanie uzupełniona automatycznie.

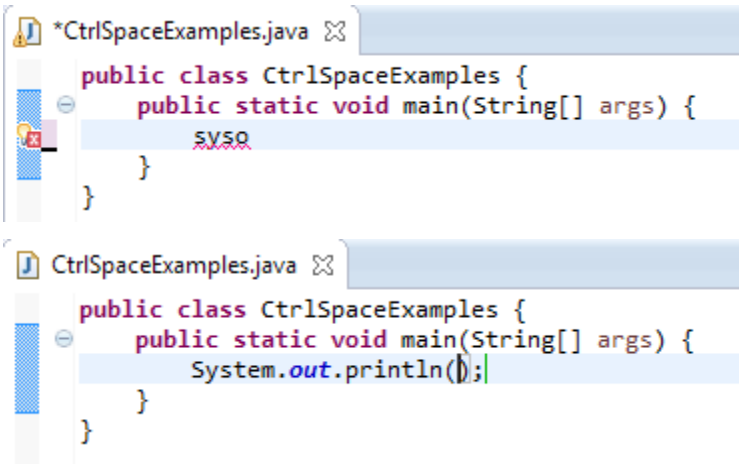
Mając zmienną o długiej nazwie:



Wystarczy, że przy kolejnym użyciu wpiszesz fragment nazwy i wciśniesz Ctrl+Spacja, a długa nazwa zostanie uzupełniona:

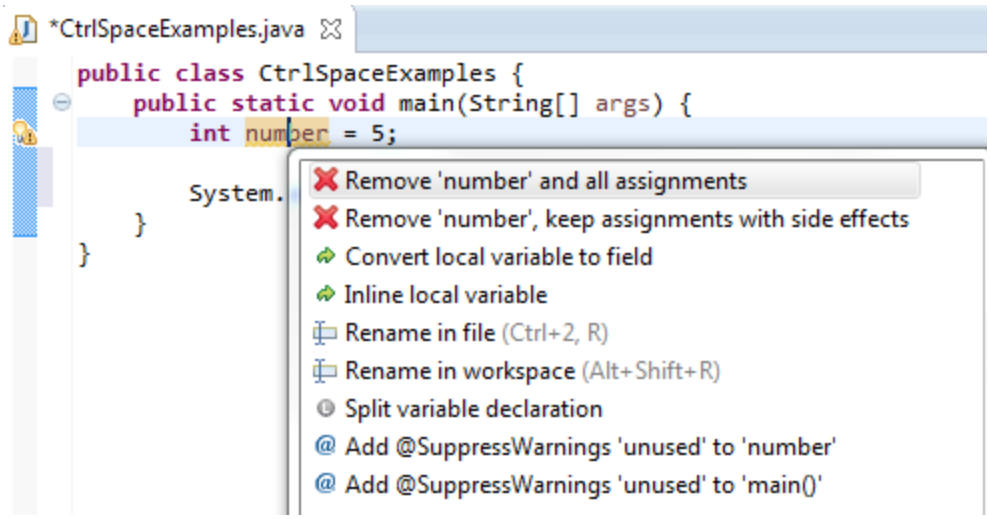


W początkowej fazie nauki Javy równie często wykorzystuje się instrukcję *System.out.println()* - jej wpisywanie również można uprościć. Wystarczy, że wpiszesz *syso* i wciśniesz Ctrl+Spację, a reszta zostanie uzupełniona.



CTRL + 1

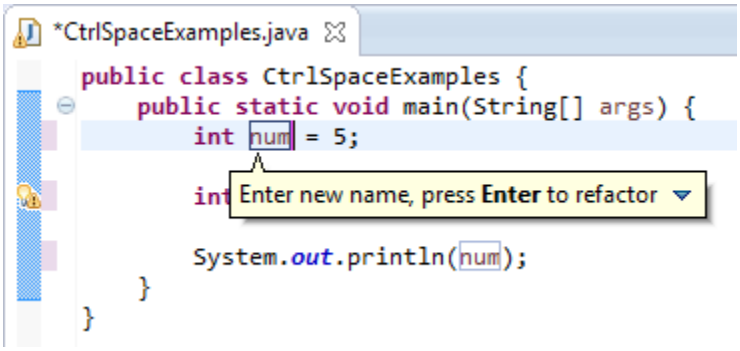
Skrót, który przydaje się szczególnie wtedy, gdy dużo pracujemy na klawiaturze i nie przepadamy za sięganiem po mysz. Jeżeli przykładowo widzisz ostrzeżenie (podkreślenie na żółto) wystarczy, że najedziesz w dany obszar kursorem i wciśniesz Ctrl+1 a eclipse podpowie Ci sugerowane rozwiązania.



W powyższym przykładzie widzimy ostrzeżenie, któe informuje nas o tym, że utworzyliśmy zmienną, której nigdzie nie wykorzystujemy - w podpowiedziach pojawia się m.in. możliwość jej usunięcia.

Alt + Shift + R

Jeżeli w swoim kodzie chcesz zmienić nazwę zmiennej lub zauważyłeś błąd typu literówka, to poprawienie tego może być problematyczne, ponieważ zmiennej tej prawdopodobnie używasz co najmniej w kilku innych miejscach. Wciskając skrót Ctrl+Shift+R na nazwie zmiennej, czy też nazwie klasy, możesz zmienić ich nazwę, a eclipse zadba o to, aby zaktualizować jej nazwę również we wszystkich innych jej wystąpieniach w kodzie źródłowym.

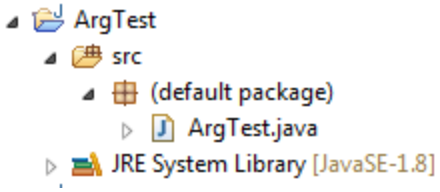


3.1.9 Dodatek 2 - wywołanie programu z argumentami

Wróćmy jeszcze na chwilę do metody main i jej argumentów. Programy napisane w Javie najczęściej są pakowane docelowo do jednego archiwum w postaci pliku **.jar** (Java ARchive). Program taki możemy uruchomić z wiersza poleceń z odpowiednimi argumentami w postaci listy oddzielonej spacjami. Argumenty te utworzą tablicę typu String, a następnie zostaną przekazane jako argument metody *main()* - to właśnie **String[] args**, który powtarza się za każdym razem.

Najczęściej argumenty wywołania są pomijane, jednak warto wiedzieć jak z nich korzystać. Utwórzmy zwykły pusty projekt z jedną klasą zawierającą metodę *main()*:

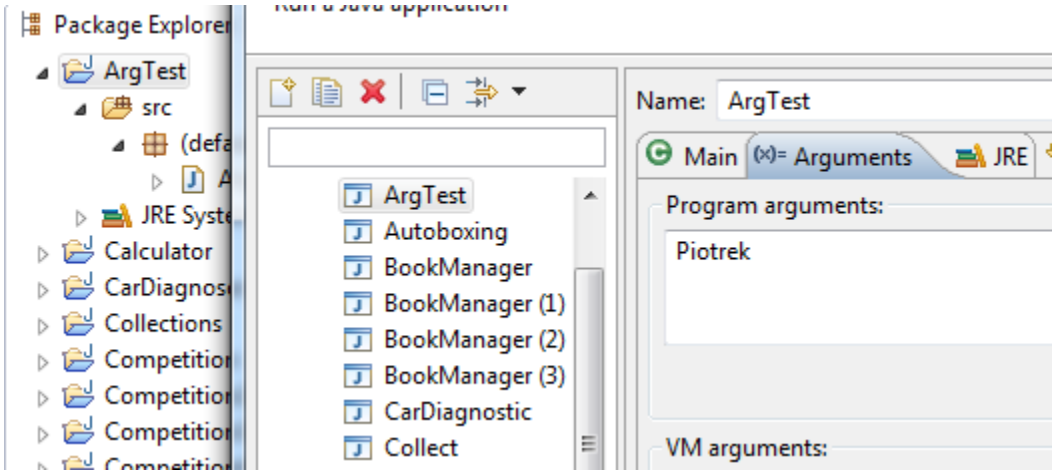
Struktura projektu



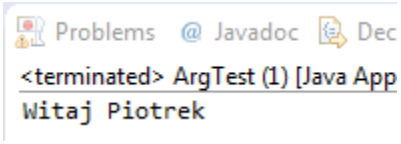
plik ArgTest.java

```
public class ArgTest {
    public static void main(String[] args) {
        System.out.println("Witaj " + args[0]);
    }
}
```

Widzimy tutaj prosty przykład bardzo podobny do naszego pierwszego "Hello World" jednak tym razem wyświetlamy tekst "Witaj" połączony z pierwszym elementem tablicy **args**. W jaki sposób przekazać ten argument do metody main? Kliknij prawym przyciskiem myszy na projekcie, wybierz opcję **Run As -> Run Configuration** a następnie w zakładce Arguments podaj np. swoje imię:



Teraz wystarczy wybrać Apply oraz Run, a na ekranie zobaczymy, że argument został poprawnie przekazany:



Jeżeli chcesz przekazać większą ilość argumentów, wystarczy, że wymienisz je po spacji, a następnie odwołasz się do nich po odpowiednich indeksach w kodzie Javy, tj. *args[0], args[1], args[2]* itd - tak jak w przypadku każdej innej tablicy.

3.2 Programowanie obiektowe

Czego się dowiesz

- Czym jest programowanie obiektowe
- Jaka jest różnica między klasą a obiektem
- W jaki sposób definiować własne typy danych

- Czym są metody i konstruktory
- Czym jest przeciążanie metod i konstruktorów
- Do czego służy słowo kluczowe
- Czym są pakiety i jakie są możliwe specyfikatory dostępu
- Jak porównywać obiekty

Czym jest programowanie obiektowe?

Czysto teoretycznie programowaniem obiektowym najłatwiej nazwać próbę odwzorowania bytów ze świata rzeczywistego w naszej aplikacji poprzez utworzenie nowych, bardziej złożonych typów danych.

Dużo prościej jest to jednak zrozumieć na konkretnych przykładach, do których za chwilę przejdziemy.

3.2.1 Klasy i obiekty

Ćwiczenie Wyobraź sobie, że tworzysz prostą aplikację do obsługi sklepu komputerowego. Najważniejsze będzie w niej zdecydowanie przechowywanie informacji o produktach. Wypisz kilka cech produktu, które Twoim zdaniem będą istotne z punktu widzenia użytkownika tej aplikacji(sprzedawcy).

Przykłady cech przydatnych

```
Nazwa produktu
Cena
Producent
Model / nazwa kodowa
Rok produkcji
```

Możliwe, że rzeczy, które wypisałeś są inne od pokazanych powyżej - to bardzo dobrze! Świadczyć to może o różnej wizji na aplikację. Na tę chwilę ograniczymy się do dwóch podstawowych informacji, czyli nazwy produktu oraz jego ceny. Te dwie podstawowe rzeczy na dobrą sprawę pozwolą nam znaleźć dowolny towar w bazie wszystkich produktów oraz dokonać sprzedaży na podstawie podanej ceny.

Ćwiczenie *(5 min)*

> Napisz prosty program, w którym w zmiennych przechowasz informacje o 2 różnych produktach (np. Monitor Samsung Syncmaster za 700zł oraz Laptop HP Probook 450 za 3000zł). Wydrukuj następnie informacje o tych produktach na ekranie. Wykorzystaj różne typy danych do przechowywania nazw produktów a inne do przechowywania ceny.

Przypomnijmy, że najpierw należy utworzyć nowy projekt poprzez menu File -> New -> Java Project, a następnie w folderze *src* dodać nową klasę np. korzystając z przycisku ze znakiem C na górnej belce nawigacyjnej. Nazwa klasy powinna być identyczna z nazwą pliku (eclipse zadba o to automatycznie) - w naszym przypadku będzie to plik Shop.java.

plik Shop.java

```
1  public class Shop {
2      public static void main(String[] args) {
3          String product1Name = "Samsung Syncmaster";
4          double product1Price = 700.0;
5
6          String product2Name = "HP Probook 450";
7          double product2Price = 3000.0;
8
9          System.out.println("Produkty w sklepie: ");
10         System.out.println(product1Name + ":" + product1Price);
11         System.out.println(product2Name + ": " + product2Price);
```

```
    }
}
```

Wszystko wygląda na pierwszy rzut oka w porządku, ale zastanów się teraz w jaki sposób zapisałbyś informację o kilkuset produktach w tym sklepie? Możliwe, że przychodzi Ci do głowy utworzenie tablic z imionami, nazwiskami itd. Nie jest to najgorszy pomysł, jednak ma jedną dużą wadę - dane nie są spójne i w żaden sposób ze sobą powiązane. Zmiana czegokolwiek, np dodanie 1 produktu, wiązałoby się z koniecznością aktualizacji innej tablicy z cenami, należałoby uważać na odwoływanie się do odpowiednich indeksów tablic itd.

Tutaj dochodzimy do sedna programowania obiektowego. Czy nie byłoby świetną sprawą możliwość utworzenia swojego typu danych "Product", który moglibyśmy wykorzystywać tak jak wartości typu int, czy double? W Javie możemy zdefiniować swój typ, zwyczajnie tworząc nową klasę, a w niej definiując zmienne typów prostych (lub wcześniej utworzonych typów obiektowych).

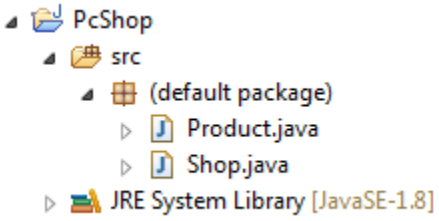
Informacja: Klasą nazywamy zbiór cech oraz funkcjonalności obiektu, który chcemy odwzorować w pamięci komputera.

---

plik Product.java

```
public class Product {
    String name;
    double price;
}
```

Struktura projektu eclipse:



Zauważ, że w klasie tej nie tworzymy po raz kolejny metody **main** - jest ona potrzebna tylko w jednej klasie w ramach całego projektu i to od niej rozpocznie się działanie programu. Pozostałe klasy będą miały natomiast za zadanie definicję nowego typu danych lub wydzielenie pewnych funkcjonalności (np. obliczeń, przetwarzania tekstu, pobierania danych z internetu itd.). Na podstawie dowolnej klasy możemy utworzyć **obiekt**. Obiekty zawsze będziemy tworzyli poprzez zapis **new NazwaKlasy();** przekazując w nawiasie ewentualne parametry (o tym za chwilę).

Informacja: Obiektem nazywamy konkretny egzemplarz danej klasy. Klasą nazwiemy "Produkt", ale obiektem " produkt laptop HP Probook 450 kosztujący 3000zł".

---

plik Shop.java

```
public class Shop {
    public static void main(String[] args) {
        Product product1 = new Product();
        product1.name = "Samsung Syncmaster";
        product1.price = 700.0;

        Product product2 = new Product();
        product2.name = "HP Probook 450";
        product2.price = 3000.0;
```

```
10
11          System.out.println("Produkty w sklepie: ");
12          System.out.println(product1.name + ":" + product1.price);
13          System.out.println(product2.name + ": " + product2.price);
14      }
15  }
```

Jak widzisz zmienne *name* i *price* z wcześniejszego kodu są teraz opakowane w **obiekty** typu Product. Do poszczególnych **pól klasy** odwołujemy się za pomocą operatora kropki, np. *"product1.name"*.

Informacja: Jeżeli nie zainicjujesz poszczególnych pól obiektu, przyjmą one wartości domyślne. Dla typów liczbowych jest to 0 lub 0.0, dla typu char specjalna wartość pusta, a dla typów obiektowych (w tym String) jest to wartość null.

Uwaga: Często powtarzającym się błędem w Javie jest **NullPointerException**. Oznacza on, że obiekt, do którego próbujesz się odwołać nie został utworzony, a jedynie zadeklarowany. Jeżeli zobaczysz go w eclipse sprawdź więc, czy przypisałeś do odpowiedniej zmiennej (referencji) obiekt utworzony za pomocą słowa new.

Na chwilę obecną może Ci się wydawać, że zrobiło się tylko więcej kodu, a program nadal robi to samo. Zwróć jednak uwagę, że dane są teraz bardziej spójne, a dzięki podejściu obiektowemu informacje o np. 100 produktach możemy przechowywać w tylko 1 tablicy typu *Product[]*.

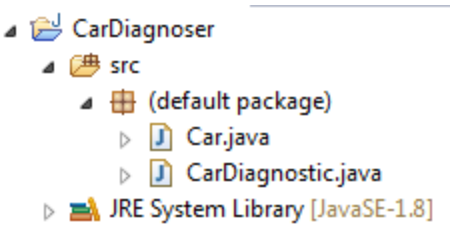
plik Shop.java

```
1  public class Shop {
2      public static void main(String[] args) {
3          Product[] products = new Product[2];
4
5          products[0] = new Product();
6          products[0].name = "Samsung Syncmaster";
7          products[0].price = 700.0;
8
9          products[1] = new Product();
10         products[1].name = "HP Probook 450";
11         products[1].price = 3000.0;
12
13         System.out.println("Produkty w sklepie: ");
14         System.out.println(products[0].name + ":" + products[0].price);
15         System.out.println(products[1].name + ": " + products[1].price);
16     }
17 }
```

Ćwiczenie *(10 min)*

> Wyobraź sobie, że tworzysz aplikację do diagnostyki komputerowej samochodów. Zacznij od utworzenia klasy Car przechowującej informacje o marce producenta, modelu, roku produkcji i mocy silnika. W drugiej klasie o nazwie CarDiagnostic utwórz dwa obiekty klasy Car i wyświetl informacje o samochodach na ekranie.

Struktura projektu:



plik Car.java

```
public class Car {
    String carBrand; // marka samochodu
    String model;
    int year; //rok produkcji
    int horsePower; // ilość koni mechanicznych
}
```

plik CarDiagnostic.java

```
public class CarDiagnostic {
    public static void main(String[] args) {
        Car audiA4 = new Car();
        audiA4.carBrand = "Audi";
        audiA4.model = "A4";
        audiA4.year = 2008;
        audiA4.horsePower = 170;

        Car vwGolf = new Car();
        vwGolf.carBrand = "Volkswagen";
        vwGolf.model = "Golf";
        vwGolf.year = 2010;
        vwGolf.horsePower = 130;

        System.out.println("Samochód 1: ");
        System.out.println(audiA4.carBrand + " " + audiA4.model
                + ", rok produkcji: " + audiA4.year + ", moc: "
                + audiA4.horsePower);

        System.out.println("Samochód 2: ");
        System.out.println(vwGolf.carBrand + " " + vwGolf.model
                + ", rok produkcji: " + vwGolf.year + ", moc: "
                + vwGolf.horsePower);
    }
}
```

3.2.2 Metody

Klasa Product potrafi już przechowywać informacje o nazwie i cenie produktu, jednak jak wspomnieliśmy w definicji klasy jest to także zbiór funkcjonalności. W programowaniu obiektowym funkcjonalności danej klasy realizuje się poprzez utworzenie **metod**. W naszym przykładzie funkcjonalnością może być na przykład zwrócenie przez obiekt klasy Product nazwy oraz ceny w czytelnej formie, dzięki czemu w metodzie println() nie będziemy musieli się odwoływać do poszczególnych pól.

plik Product.java

```
public class Product {
    String name;
```

```
    double price;

    String getProductInfo() {
        return name + ": " + price;
    }
}
```

W klasie Product utworzyliśmy metodę **getProductInfo**. Ponieważ zwraca ona opisową formę produktu musieliśmy zadeklarować jej typ jako String. Wynik metody należy zwrócić za pomocą słowa kluczowego **return**.

Informacja: Ogólna postać metody to:

```
typ_zwracany nazwaMetody(opcjonalne_argumenty_metody) {
  //ciało metody między nawiasami klamrowymi
}
```

Elementami opcjonalnymi są jeszcze specyfikatory dostępu (np. public) oraz oznaczenie metody jako statycznej (static) - do tego dojdziemy jednak niebawem. Metody mogą zwracać wynik (np. String) i wtedy musi w nich występować instrukcja **return**, ale mogą także nie zwracać żadnego wyniku - sytuacja taka będzie miała miejsce, gdy metoda ma za zadanie np. wydrukować coś na ekranie za pomocą System.out.print(). Jeżeli metoda nie zwraca żadnego wyniku należy jako jej typ zwracany podać słowo kluczowe **void**. Przykładem metody, która nie zwraca żadnego wyniku jest metoda **main**, którą poznałeś już na samym początku kursu.

---

plik Shop.java

```
public class Shop {
    public static void main(String[] args) {
        Product[] products = new Product[2];

        products[0] = new Product();
        products[0].name = "Samsung Syncmaster";
        products[0].price = 700.0;

        products[1] = new Product();
        products[1].name = "HP Probook 450";
        products[1].price = 3000.0;

        System.out.println("Produkty w sklepie: ");
        System.out.println(products[0].getProductInfo());
        System.out.println(products[1].getProductInfo());
    }
}
```

Do metod, podobnie jak do pól klasy odołujemy się za pomocą operatora kropki, jednak oprócz samej nazwy metody nie możemy zapomnieć o dodaniu na końcu okrągłych nawiasów.

3.2.3 Konstruktory

Ostatnią rzeczą, którą możemy uprościć w klasie Shop jest inicjalizacja zmiennych. W chwili obecnej w celu utworzenia jednego tylko obiektu potrzebujemy aż 3 linijek kodu - w przypadku tworzenia 100 obiektów, kod rozrasta się do 300 linii - jest to niedopuszczalne.

Tworzenie obiektów możemy jednak uprościć za pomocą specjalnych metod nazywanych **konstruktorami**.

Informacja: Konstruktor to specjalna metoda, która nie ma zadeklarowanego żadnego zwracanego typu (nawet

void), a jej nazwa jest identyczna z nazwą klasy, w której się znajduje (z uwzględnieniem wielkości liter). Podobnie jak każda inna metoda może przyjmować argumenty.

plik Product.java

```
public class Product {
    String name;
    double price;

    //konstruktor przyjmujący 2 argumenty
    Product(String n, double p) {
        name = n;
        price = p;
    }

    String getProductInfo() {
        return name + ": " + price;
    }
}
```

Nasz konstruktor przyjmuje dwa argumenty - jeden typu String, a drugi typu double. Wartości przekazane jako argumenty konstruktora przypisujemy następnie do pól klasy, czyli *name* oraz *price*. Teraz możemy uprościć tworzenie obiektów w klasie Shop do tylko jednej linijki dla każdego z nich:

plik Shop.java

```
public class Shop {
    public static void main(String[] args) {
        Product[] products = new Product[2];

        products[0] = new Product("Samsung Syncmaster", 700.0);

        products[1] = new Product("HP Probook 450", 3000.0);

        System.out.println("Produkty w sklepie: ");
        System.out.println(products[0].getProductInfo());
        System.out.println(products[1].getProductInfo());
    }
}
```

Uwaga: Każda klasa posiada domyślnie jeden niejawny konstruktor (bez parametrów). Jeżeli jednak zdefiniujesz w swojej klasie chociaż jeden konstruktor przyjmujący dowolne argumenty, to konstruktor domyślny przestaje istnieć.

Ćwiczenie *(10 min)*

Rozwiń aplikację z poprzedniego ćwiczenia (diagnostyka sanmochodu) o następujące informacje. W klasie Car dodaj konstruktor pozwalający zainicjować wszystkie pola klasy oraz dwie metody: getInfo(), która zwróci opisową formę danego samochodu, a także upgreade(), która zwiększa moc silnika o tyle koni mechanicznych ile przekażemy jako jej parametr. Przetestuj nowe funkcjonalności w klasie CarDiagnostic.

plik Car.java

```
public class Car {

    String carBrand; // marka samochodu
    String model;
```

```
    int year; //rok produkcji
    int horsePower; // ilość koni mechanicznych

    //konstruktor do zainicjowania wszystkich pól
    Car(String cb, String m, int y, int hp) {
        carBrand = cb;
        model = m;
        year = y;
        horsePower = hp;
    }

    //zwiększenie mocy silnika
    void upgreade(int hp) {
        horsePower = horsePower + hp;
    }

    //zwrócenie opisowej formy samochodu
    String getInfo() {
        return carBrand + " " + model + "; " + year + "; " + horsePower + "HP";
    }
}
```

plik CarDiagnostic.java

```
public class CarDiagnostic {
    public static void main(String[] args) {
        //utworzenie obiektów
        Car audiA4 = new Car("Audi", "A4", 2008, 170);
        Car vwGolf = new Car("Volkswagen", "Golf", 2010, 130);

        //tuning
        audiA4.upgreade(30);
        vwGolf.upgreade(20);

        //wydruk informacji
        System.out.println("Samochód 1: ");
        System.out.println(audiA4.getInfo());

        System.out.println("Samochód 2: ");
        System.out.println(vwGolf.getInfo());
    }
}
```

3.2.4 Przeciążanie metod i konstruktorów

Czasami może zdarzyć się sytuacja, w której nie będziemy mieli pełnych informacji o danym produkcie, który chcielibyśmy utworzyć. Przykładowo będziemy mieli jego nazwę, ale będziemy musieli jeszcze chwilę poczekać na ustalenie jej ceny w centrali. W takiej sytuacji możemy utworzyć kilka konstruktorów, które pozwolą nam zainicjować obiekt danej klasy. Gdy w klasie istnieje kilka konstruktorów lub metod o takiej samej nazwie, ale różniących się przyjmowanymi parametrami, to będziemy mówili o nich, że są dostępne w kilku **przeciążonych** wersjach.

plik Product.java

```
public class Product {
    String name;
    double price;

```

```
    //konstruktory
    Product(String n, double p) {
        name = n;
        price = p;
    }

    Product(String n) {
        name = n;
    }

    String getProductInfo() {
        return name + ": " + price;
    }
}
```

W powyższym przykładzie widzimy, że utworzyliśmy drugi konstruktor, który inicjuje jedynie nazwę produktu. Cena (price) przyjmie w takiej sytuacji wartość domyślną, którą dla typu double jest 0.0. W podobny sposób możemy przeciążać dowolne metody, które będą miały takie same nazwy, ale przyjmą różne parametry.

Ćwiczenie *(5 min)*

> W programie do diagnostyki samochodu dopisz dodatkowy konstruktor pozwoli zainicjować jedynie markę i model samochodu, pozostawiając rok produkcji i moc silnika wartościami domyślnymi. W klasie CarDiagnostic utwórz za pomocą tego konstruktora nowy obiekt i wyświetl informacje o nim na ekranie.

plik Car.java

```
public class Car {

    String carBrand; // marka samochodu
    String model;
    int year; //rok produkcji
    int horsePower; // ilość koni mechanicznych

    Car(String cb, String m) {
        carBrand = cb;
        model = m;
    }

    Car(String cb, String m, int y, int hp) {
        carBrand = cb;
        model = m;
        year = y;
        horsePower = hp;
    }

    void upgreade(int hp) {
        horsePower = horsePower + hp;
    }

    String getInfo() {
        return carBrand + " " + model + "; " + year + "; " + horsePower + "HP";
    }
}
```

plik CarDiagnostic.java

```
public class CarDiagnostic {
    public static void main(String[] args) {
        Car audiA4 = new Car("Audi", "A4", 2008, 170);
```

```
        Car vwGolf = new Car("Volkswagen", "Golf", 2010, 130);
        Car opelCorsa = new Car("Opel", "Corsa");

        //tuning
        audiA4.upgreade(30);
        vwGolf.upgreade(20);

        System.out.println("Samochód 1: ");
        System.out.println(audiA4.getInfo());

        System.out.println("Samochód 2: ");
        System.out.println(vwGolf.getInfo());

        System.out.println("Samochód 3: ");
        System.out.println(opelCorsa.getInfo());
    }
}
```

3.2.5 Słowo kluczowe this

Zarówno aplikacja symulująca sklep z częściami komputerowymi jak i program symulujący diagnostykę samochodową posiadają już pewną bazową funkcjonalność. Czytelność kodu w kilku miejscach można jednak poprawić.

Dobrą praktyką jest stosowanie wszędzie tam gdzie to możliwe opisowych nazw zmiennych, jednak w naszym przypadku ciężko wymyślić zamiennik dla "name", czy "year" (stosowanie polskich nazw również nie jest dobrą praktyką). Na szczęście przewidziano również taką funkcjonalność języka i nazwy argumentów metod, czy konstruktorów mogą być identyczne z nazwami pól klasy, te drugie należy jednak poprzedzić dodatkowo słowem kluczowym **this**.

Tym sposobem nasza poprawiona klasa Product prezentuje się jak poniżej.

plik Product.java

```
public class Product {
    String name;
    double price;

    //konstruktory
    Product(String name, double price) {
        this.name = name;
        this.price = price;
    }

    Product(String name) {
        this.name = name;
    }

    String getProductInfo() {
        return name + ": " + price;
    }
}
```

Zapis *this.name = name* należy rozumieć jako "przypisz do pola tej (this) klasy o nazwie *name* wartość argumentu konstruktora o nazwie *name*".

Słowo kluczowe this ma także drugie zastosowanie, które pozwala wywoływać przeciążoną wersję konstruktora w innym konstruktorze. Ma to takie zastosowanie, że pozwala zaoszczędzić powtarzalnego kodu w przypadku, gdy w klasie zdefiniujemy np. 4 podobne konstruktory, w których spora część kodu źródłowego się powtarza. U nas nie zaoszczędzimy specjalnie kodu, jednak wygląda to następująco:

plik Product.java

```
public class Product {
    String name;
    double price;

    //konstruktory
    Product(String name, double price) {
        this(name); //wywołanie innego konstruktora
        this.price = price;
    }

    Product(String name) {
        this.name = name;
    }

    String getProductInfo() {
        return name + ": " + price;
    }
}
```

Jak widzisz w konstruktorze przyjmującym dwa argumenty wywołujemy w pierwszej kolejności konstruktor z jednym argumentem poprzez this(name). W ten sposób zainicjowaliśmy pole name, a następnie możemy zainicjować cenę, czyli pole price.

Ćwiczenie *(5 min)*

> Popraw klasę Car z poprzedniego zadania w taki sposób, aby nazwy argumentów konstruktorów miały bardziej znaczące nazwy (najlepiej takie same jak nazwy pól klasy). Wykorzystaj także słowo kluczowe this w celu wywołania w jednym z konstruktorów drugiego konstruktora i zaoszczędzić tym samym powtarzalnego kodu.

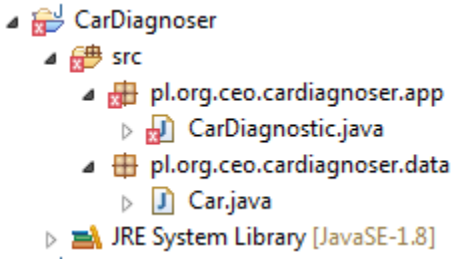
plik Car.java

```
public class Car {

    String carBrand; // marka samochodu
    String model;
    int year; //rok produkcji
    int horsePower; // ilość koni mechanicznych

    Car(String carBrand, String model) {
        this.carBrand = carBrand;
        this.model = model;
    }

    Car(String carBrand, String model, int year, int horsePower) {
        this(carBrand, model);
        this.year = year;
        this.horsePower = horsePower;
    }

    void upgreade(int hp) {
        horsePower = horsePower + hp;
    }

    String getInfo() {
        return carBrand + " " + model + "; " + year + "; " + horsePower + "HP";
    }
}
```

3.2.6 Pakiety i modyfikatory dostępu

Istotnym elementem, który pomaga w organizacji większych projektów jest podział klas i plików źródłowych na pakiety. Pakiety są niczym innym jak dodatkowymi folderami, które pozwalają grupować wspólnie klasy, które odpowiadają za podobne funkcjonalności. W celu utworzenia pakietu wybieramy po prostu New -> Package. Nazewnictwo pakietów zazwyczaj odzwierciedla nazwę domeny autorów, czyli np. pl.org.ceo.kursjava.pakiet1 - gdzie pakiet1 powinien być jakąś znaczącą nazwą, kursjava nazwą projektu, a pl.org.ceo to odwrócona nazwa domeny Centrum Edukacji Obywatelskiej.

Gdy w kodzie projektu CarDiagnoser podzielimy klasy na dwa pakiety (przeciągnij klasy do odpowiednich pakietów):



zauważamy błąd w klasie CarDiagnostic.

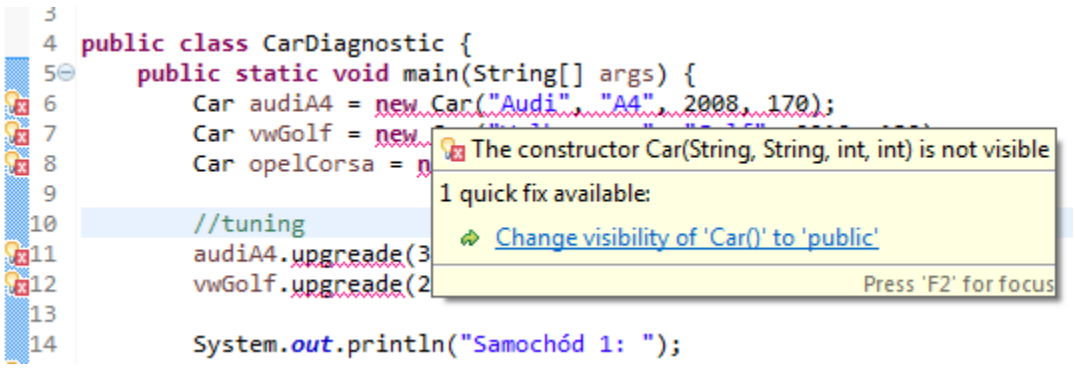
plik CarDiagnostic.java

```
1 package pl.org.ceo.cardiagnoser.app;
2 import pl.org.ceo.cardiagnoser.data.Car;
3
4 public class CarDiagnostic {
5 //kod bez zmian
6 }
```

Pierwszą rzeczą, na którą warto zwrócić uwagę są dwie linie kodu, które eclipse dodał automatycznie:

- package oznacza pakiet, w którym umieszczona jest dana klasa
- import jest dyrektywą niezbędną w przypadku, gdy korzystamy z klas umieszczonych w innych pakietach. W naszym przypadku musieliśmy zaimportować klasę Car.

Błąd w klasie polega na tym, że konstruktor jest niewidoczny:



Można sobie zadać pytanie, ale jak to niewidoczny, skoro przed chwilą z niego korzystaliśmy? Jest to spowodowane różnymi zasięgami widoczności pól metod i konstruktorów. W Javie istnieją cztery możliwe zasięgi:

- default - domyślny, czyli pakietowy zasięg dostępu
- public - publiczny, można się odwoływać z dowolnego miejsca w pakiecie i poza nim
- protected - zasięg ograniczony do danego pakietu

- private - zasięg tylko w ramach jednej klasy. Do tak oznaczonych pól, metod i konstruktorów nie można odwołać się nawet z klas w tym samym pakiecie

Ponieważ klasy Car i CarDiagnoser znajdują się w różnych pakietach odpowiednie konstruktory i pola musimy oznaczyć jako publiczne:

plik Car.java

```
package pl.org.ceo.cardiagnoser.data;

public class Car {

    public String carBrand; // marka samochodu
    public String model;
    public int year; //rok produkcji
    public int horsePower; // ilość koni mechanicznych

    public Car(String carBrand, String model) {
        this.carBrand = carBrand;
        this.model = model;
    }

    public Car(String carBrand, String model, int year, int horsePower) {
        this(carBrand, model);
        this.year = year;
        this.horsePower = horsePower;
    }

    public void upgreade(int hp) {
        horsePower = horsePower + hp;
    }

    public String getInfo() {
        return carBrand + " " + model + "; " + year + "; " + horsePower + "HP";
    }
}
```

Informacja: Nasze przykłady najczęściej będą na tyle proste, że dla uproszczenia wszystkie pliki będą znajdowały się w jednym pakiecie.

3.2.7 Ćwiczenie podsumowujące (20 minut)

Napisz prosty kalkulator, który będzie zgodny z podejściem programowania obiektowego. Niech składa się on z dwóch klas:

- Calculator - klasa, w której zdefiniujesz metody add(), subtract(), multiply() oraz divide() odpowiedzialne odpowiednio za dodawanie, odejmowanie, mnożenie i dzielenie. Każda z metod powinna przyjmować dwa argumenty typu double, na których wykonuje obliczenie i zwraca w wynik.
- Main - klasa z metodą main(), w której należy przetestować działanie poszczególnych metod. Argumentami metod powinny być dwie wcześniej zadeklarowane zmienne.

plik Calculator.java

```
public class Calculator {
    double add(double a, double b) {
        return a + b;
```

```
    }

    double subtract(double a, double b) {
        return a - b;
    }

    double multiply(double a, double b) {
        return a * b;
    }

    double divide(double a, double b) {
        return a / b;
    }
}
```

plik Calculator.java

```
public class Main {
    public static void main(String[] args) {
        double a = 25;
        double b = 5;
        Calculator calc = new Calculator();

        System.out.println(a + "+" + b + " = " + calc.add(a, b));
        System.out.println(a + "-" + b + " = " + calc.subtract(a, b));
        System.out.println(a + "*" + b + " = " + calc.multiply(a, b));
        System.out.println(a + "/" + b + " = " + calc.divide(a, b));
    }
}
```

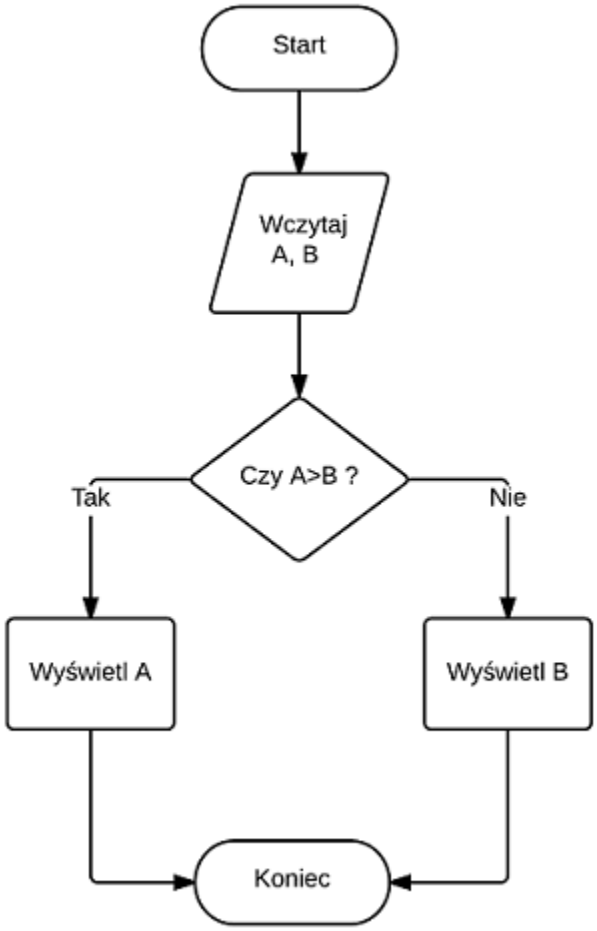
3.3 Struktury sterujące i pętle

W tej lekcji dowiesz się:

- W jaki sposób sterować wykonaniem programu
- Czym są pętle while, do while, for i for each oraz jak ich używać

3.3.1 Struktury sterujące

Struktury sterujące to specjalne elementy języka, które pozwalają na wybór odpowiedniej ścieżki w aplikacji. Przykładowo jeżeli chcemy porównać ze sobą dwie liczby całkowite, musimy sprawdzić warunek, a następnie podjąć decyzję, co w wyniku tego porównania wyświetlić na ekranie:



Co więcej warunki takie mogą być bardziej rozgałęzione, czyli np. możemy sprawdzić, czy wczytana liczba jest większa, mniejsza, czy równa 0 itd.

3.3.2 Instrukcja if else

Podstawową strukturą sterującą w większości języków programowania, w tym także w Javie jest instrukcja if. W swojej najprostszej formie odpowiada ona sytuacji, którą widzisz na wcześniejszym diagramie - pozwala sprawdzić prosty warunek, który w wyniku powinien zwrócić wartość typu boolean, czyli true lub false i na podstawie tego zadecydować, co nasz program zrobi dalej.

Ogólna konstrukcja bloku if wygląda następująco:

```
if(warunek) {
  //działania, gdy warunek zwraca true
} else {
  //działania, gdy warunek zwraca false
}
```

Ćwiczenie *(5 minut)*

> Zaimplementuj program, który służy do porównywania dwóch liczb całkowitych zgodny z przedstawionym wyżej diagramem. Wyświetl na ekranie komunikat o tym, która liczba jest większa od której. Liczby wczytaj do wcześniej zadeklarowanych zmiennych.

plik Main.java

```
public class Main {
    public static void main(String[] args) {
        int a = 10;
        int b = 11;

        if (a > b) {
            System.out.println("A jest większe od B " + a + " > " + b);
        } else {
            System.out.println("B jest większe od A " + b + " > " + a);
        }
    }
}
```

W nawiasie występującym po instrukcji if musi znajdować się wyrażenie, które w wyniku zwraca true lub false. Nie musi to być jedynie porównanie dwóch liczb tak jak w powyższym przykładzie, ale także bardziej złożony warunek np. wykorzystujący operatory logiczne && lub ||.

W przypadku kodu tak prostego jak ten powyżej, gdzie w bloku if oraz else pomiędzy nawiasami klamrowymi znajduje się jedynie jedna instrukcja (np. *System.out.print()*), nawiasy klamrowe można pominąć, czyli uprościć nasz kod do postaci:

```
public class Main {
    public static void main(String[] args) {
        int a = 10;
        int b = 11;

        if (a > b)
            System.out.println("A jest większe od B " + a + " > " + b);
        else
            System.out.println("B jest większe od A " + b + " > " + a);
    }
}
```

W praktyce jednak stosowanie klamer podnosi czytelność kodu, więc warto je stosować nawet w najbardziej trywialnych sytuacjach.

Możliwe, że zauważyłeś, że powyższy kod ma jedną poważną wadę. W przypadku, gdy liczby a i b będą identyczne, to ich porównanie za pomocą znaku ostrej nierówności zwróci false. W takiej sytuacji wyświetlony zostanie komunikat o tym, że liczba B jest większa od A, a to oczywiście nie jest prawdą.

W takim przypadku możemy zastosować np. zagnieżdżone instrukcje if.

Ćwiczenie *(5 minut)*

> Popraw wcześniejszy kod w taki sposób, aby najpierw sprawdzić, czy liczby są równe, a dopiero gdy nie są, porównaj je operatorem nierówności. Wykorzystaj zagnieżdżone warunki if i w każdej sytuacji wyświetl na ekranie stosowny komunikat.

```
public class Main {
    public static void main(String[] args) {
        int a = 10;
        int b = 11;

        if (a == b) {
            System.out.println("Liczby A i B są równe " + a + " = " + b);
        } else {
            if (a > b) {
                System.out.println("A jest większe od B " + a + " > " + b);
```

```
            } else {
                System.out.println("B jest większe od A " + b + " > " + a);
            }
        }
    }
}
```

Powyższy kod posiada jedną, aczkolwiek istotną wadę. Nawet pojedynczo zagnieżdżone bloki instrukcji if wpływają w znaczącym stopniu na zmniejszenie czytelności kodu i w ogólności dobrą praktyką jest unikanie takich sytuacji. Najprościej jest to zrobić korzystając z nieco bardziej złożonej instrukcji if else:

```
if (warunek1) {
    //instrukcje gdy warunek1 jest true
} else if (warunek2) {
    //instrukcje gdy warunek2 jest true
} else if (warunek3) {
    //instrukcje gdy warunek3 jest true
} else {
    //instrukcje gdy żadny z warunków nie był true
}
```

Korzystając z dodatkowych warunków w postaci *else if* możemy w dowolny sposób rozgałęzić działanie naszej aplikacji bez konieczności zagnieżdżania warunków if i obniżania czytelności kodu. Pamiętaj, że blok z warunkiem2 wykona się tylko wtedy, gdy warunek1 zwróci false, analogicznie warunek3, gdzy warunek2 oraz warunek1 będą nieprawdziwe.

Ćwiczenie *(5 minut)*

> Przerób wcześniejszy kod z porównywaniem liczb w taki sposób, aby wyeliminować zagnieżdżone warunki if.

```
public class Main {
    public static void main(String[] args) {
        int a = 12;
        int b = 11;

        if (a == b) {
            System.out.println("Liczby A i B są równe " + a + " = " + b);
        } else if (a > b) {
            System.out.println("A jest większe od B " + a + " > " + b);
        } else {
            System.out.println("B jest większe od A " + b + " > " + a);
        }
    }
}
```

Zauważ, że ponieważ wiemy, że istnieją tylko trzy możliwe wyniki porównania dwóch liczb, to w ostatnim bloku nie musimy zapisywać *else if(a<b)* a wystarczy jedynie samo *else* - jest to jedyny możliwy wynik jaki pozostał.

3.3.3 Struktura sterująca switch

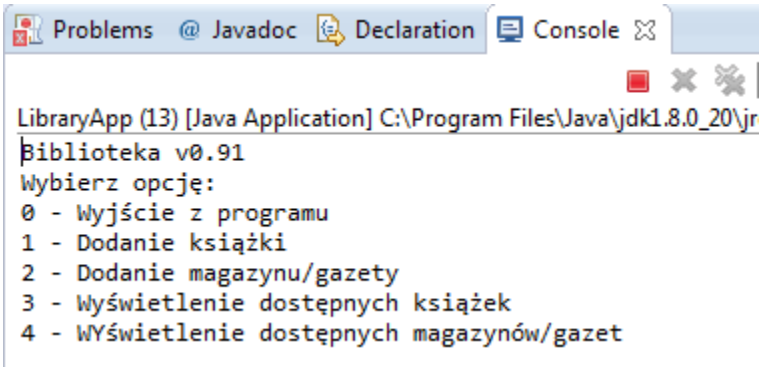
Powyżej pokazana struktura if-else-if pomimo iż bardziej czytelna od zagnieżdżonych warunków, to jednak w sytuacji, gdy mamy np. menu aplikacji składające się z 10 możliwych opcji, nie wydaje się najlepszym rozwiązaniem. W większości języków programowania rozwiązaniem tego problemu jest bardziej złożona struktura **switch**.

W odróżnieniu od instrukcji if, w strukturze switch operujemy nie na warunkach zwracających wartość true lub false, ale na liczbach całkowitych. Od Javy w wersji 7 możliwe jest także stosowanie w miejsce liczb napisów typu String.

Schematyczna budowa switch wygląda jak poniżej:

```
//wybrana opcja musi być typu całkowitoliczbowego lub być wartością typu String
switch (wybrana_opcja) {
    case wartość1:
        //instrukcje, gdy wartość1 jest równa wartości wybrana_opcja
        break;
    case wartość2:
         //instrukcje, gdy wartość2 jest równa wartości wybrana_opcja
        break;
    case ...
    default:
        //instrukcje, gdy żaden z wcześniejszych warunków nie pasuje do wybrana_opcja
        //odpowiednik ostatniego bloku else z instrukcji if-else-if
}
```

Istotne w powyższym kodzie jest zastosowanie instrukcji break. Jeżeli jej nie zastosujesz w danym bloku case, wtedy wykonane zostaną także instrukcje z innych bloków case znajdujących się poniżej (aż do napotkania break). W bloku default nie jest to wymagane, ponieważ jest on ostatnim w całej konstrukcji i nie musimy niczego przerywać. Dzięki instrukcji switch możemy zbudować w wygodny sposób proste menu w swojej aplikacji w stylu:



Ćwiczenie *(10 minut)*

> Napisz prostą aplikację, w której utworzysz jedną zmienną całkowitoliczbową i przypiszesz do niej wartość z zakresu od 1 do 10. Następnie stwórz blok switch, w którym na podstawie wybranej opcji wyświetlisz wcześniej zainicjowaną zmienną podniesioną do 1, 2, 3 lub 4 potęgi.

```
1  public class Exponential {
2      public static void main(String[] args) {
3          int number = 5;
4
5          int option = 2;
6
7          switch (option) {
8          case 1:
9              System.out.println(number + " do potęgi 1 = " + number);
10             break;
11         case 2:
12             System.out.println(number + " do potęgi 2 = " + number*number);
13             break;
14         case 3:
15             System.out.println(number + " do potęgi 3 = " + number*number*number);
16             break;
17         case 4:
18             System.out.println(number + " do potęgi 4 = " + number*number*number*number);
19             break;
```

```
        default:
            System.out.println("Wybrano niepoprawną opcję");
        }
    }
}
```

3.3.4 Pętle while i do while

W pierwszej części kursu dowiedziałeś się, że w Javie istnieją specjalne struktury danych, które pozwalają przechowywać wiele wartości tego samego typu, które nazwaliśmy tablicami. W tym miejscu powrócimy do nich na chwilę, aby pokazać, że ich przetwarzanie, czy wyświetlanie może być znacznie krótsze i do każdego elementu tablicy nie musimy odwoływać się osobno tak jak do zwykłych zmiennych.

Dwa pierwsze rodzaje pętli, które omówimy to **while** oraz **do while**. Różnica między nimi jest subtelna, aczkolwiek wpływa w znaczący sposób na to co dzieje się w naszym programie. Ogólna postać obu pętli wygląda jak poniżej:

```
//pętla while
while (warunek) {
    //instrukcje, które będą powtarzane tak długo, dopóki warunek zwraca true
}

//pętla do while
do{
    //instrukcje, które będą się wykonywały tak długo, dopóki warunek zwraca true
} while(warunek);
```

Różnica pomiędzy dwoma wyżej pokazanymi rodzajami pętli polega na tym, że w przypadku zwykłej pętli while warunek jest sprawdzany przed rozpoczęciem ciała pętli, więc jeżeli warunek nie będzie prawdziwy, to zawartość pętli nie wykona się ani razu. W przypadku pętli do while mamy pewność, że instrukcje w jej ciele wykonają się co najmniej raz, ponieważ warunek sprawdzany jest dopiero na końcu. W miejscu wyrażenia, które w powyższym kodzie nazwaliśmy jako *warunek* należy wstawić dowolną zmienną typu boolean lub wyrażenie logiczne zwracające true lub false.

Ćwiczenie *(10 minut)*

> Napisz program, w którym zadeklarujesz tablicę 50 liczb całkowitych. Wypełnij ją przy pomocy pętli while wartościami od 1 do 50, a następnie wyświetl jej kolejne elementy za pomocą pętli do while.

plik Loops1.java

```
public class Loops1 {
    public static void main(String[] args) {
        int[] array = new int[50];

        int i = 0; // licznik pętli

        // wypełniamy tablicę
        while (i < array.length) {
            array[i] = i + 1;
            i = i + 1;
        }

        // zerujemy licznik
        i = 0;

        //wyświetlamy wartości
        do {
            System.out.print(array[i] + "; ");
```

```
            i = i + 1;
        } while (i < array.length);
    }
}
```

Do rozwiązania należało zastosować scenariusz, który będzie się często powtarzał w niemal każdej aplikacji, która korzysta z tablic:

1. Utworzenie tablicy
2. Zainicjowanie licznika pętli
3. Wypełnienie kolejnych komórek tablicy wartościami (while)
4. Wykonanie operacji na danych w tablicy (pętla do while)

W praktyce dużo częściej stosuje się pętle while, ponieważ są po prostu bardziej intuicyjne

3.3.5 Inkrementacja i dekrementacja

Operacją, która w przypadku pętli będzie się bardzo często powtarzała jest zwiększanie lub zmniejszanie zmiennej reprezentującej licznik o 1. W programowaniu jest to tak często wykorzystywane, że powstały specjalne operatory, które skracają zapis, a po pewnym czasie stają się nawykiem przy pisaniu pętli.

Inkrementacja to zwiększenie wartości zmiennej o 1, natomiast **dekrementacja** to zmniejszenie o 1. Inkrementację oznaczamy znakiem podwójnego plusa (++), natomiast dekrementację podwójnego minusa (–).

W praktyce wygląda to następująco:

```
int a = 1;
a++;
//a ma teraz wartość 2
a--;
//a ma teraz znowu wartość 1
```

Dodatkowo istnieją dwa rodzaje powyższych operatorów. W formie przyrostkowej, czyli np. a++ oraz przedrostkowej, czyli ++a. Różnica polega na tym, kiedy wykonywane jest faktyczne zwiększenie, czy zmniejszenie wartości o 1. W przypadku inkrementacji przyrostkowej wartość zmiennej jest zwiększana dopiero po wykonaniu operacji, w której zmienna ta występuje.

Najłatwiej zobrazować to prostym przykładem:

```
int x = 1;
System.out.println(x); //wyświetla 1
System.out.println(x++); //również wyświetla 1, ale po wyświetleniu zwiększa wartość x do 2
System.out.println(x); //wyświetla 2
```

Analogicznie dla inkrementacji przedrostkowej:

```
int x = 1;
System.out.println(x); //wyświetla 1
System.out.println(++x); //najpierw zwiększa wartość x do 2 i wyświetla 2
System.out.println(x); //wyświetla 2
```

W pętlach znajduje to takie zastosowanie, że możemy pominąć ręczne zwiększanie licznika w postaci *i = i+1* i zapisać *i++*, dodatkowo w niektórych sytuacjach bezpośrednio przy sprawdzaniu warunku, np.:

```
int i=0;
while(i++ < 10) { ... }
```

Ćwiczenie *(5 minut)*

Przerób poprzedni przykład z wypełnianiem tablicy w taki sposób, aby wykorzystać operator inkrementacji. Dodatkowo wartości w tablicy wyświetl od końca wykorzystując dekrementację.

plik Loops1Increment.java

```
public class Loops1Increment {
    public static void main(String[] args) {
        int[] array = new int[50];

        int i = 0; // licznik pętli

        // wypełniamy tablicę
        while (i < array.length) {
            array[i] = i + 1;
            i++;
        }

        // przypisujemy do licznika ostatni indeks tablicy
        i = array.length-1;

        //wyświetlamy wartości
        do {
            System.out.print(array[i] + "; ");
        } while (i-- > 0);
    }
}
```

Pętla while nie zmieniła się znacząco, jedynie zamieniliśmy sposób zwiększania zmiennej i. W pętli do while zastosowaliśmy dekrementacją, a dodatkowo zmieniliśmy warunek, na taki, który pozwala nam wyświetlać wartości od ostatniego indeksu tablicy, do pierwszego (czyli 0 zgodnie z indeksowaniem tablic).

3.3.6 Pętle for i for each

Pętle for i for each to dwa kolejne rodzaje pętli, które idealnie znajdują zastosowanie w przypadku kolekcji danych o znanych z góry rozmiarach, czyli np. tablicach. Ich specyficzna konstrukcja sprawia, że powinniśmy je stosować przede wszystkim w sytuacjach, gdy chcemy przeglądnąć całą kolekcję elementów.

Schematyczna budowa pętli for:

```
for(inicjalizacja_licznika; warunek_stopu; zmiana_licznika) {
    //operacje
}
```

Zwróć uwagę, że w nawiasach okrągłych znajdują się trzy elementy, które są oddzielone od siebie średnikami. W sytuacji, gdy chcielibyśmy uzupełnić tablicę o rozmiarze 10 liczbami 10, 20, 30, ..., 100, można to zrobić w następujący sposób:

```
public class ForLoop {
    public static void main(String[] args) {
        int[] array = new int[10];

        for(int i=0; i < array.length; i++) {
            array[i] = (i+1)*10;
        }
    }
}
```

W porównaniu do pętli while i do while zapis taki staje się bardziej czytelny, ponieważ wszystkie elementy bezpośrednio związane z pętlą, czyli zmienna licznika i warunek końca pętli są zawarte bezpośrednio w deklaracji pętli obok

siebie. Będzie to przydatne, gdy kod w pętli będzie nieco bardziej rozbudowany.

Istnieje także odmiana pętli for, która przeznaczona jest wyłącznie do operacji na kolekcjach, gdzie znacząco uproszczono kwestie związane z licznikiem, czy jakimikolwiek warunkami stopu. Jedynym zadaniem pętli for each, o której mowa, jest przejście po wszystkich elementach kolekcji.

Schematyczna budowa pętli for each:

```
for(typ_zmiennej zmienna: kolekcja) {
    operacje na zmiennej
}
```

Typ zmiennej musi być zgodny z typem kolekcji (czyli np. tablicy), ponieważ w kolejnych iteracjach przypisywana będzie do niej kolejna jej wartość.

Przykładowo dodajmy do poprzedniego przykładu opcję wydruku danych na ekranie z użyciem pętli for each:

```
1  public class ForLoop {
2      public static void main(String[] args) {
3          int[] array = new int[10];
4
5          for(int i=0; i < array.length; i++) {
6              array[i] = (i+1)*10;
7          }
8
9          for(int number: array) {
10             System.out.print(number + "; ");
11         }
12     }
13 }
```

Ponieważ nasza tablica była typu int[] to zmienna *number* została zadeklarowana jako int. Możemy ją wykorzystać następnie w naszej pętli.

Uwaga: Pamiętaj, że pętlę for each najlepiej jest stosować wyłącznie do operacji odczytu kolekcji, a nie jej modyfikacji. Do zmiennej deklarowanej w pętli przypisywana jest w rzeczywistości kopia wartości (lub referencji obiektu) przechowywanej w kolekcji, więc modyfikując ją nie modyfikujesz oryginalnej wartości.

3.3.7 Ćwiczenie podsumowujące

Napisz program w którym wyświetlisz na ekranie zawartość tablicy dwuwymiarowej o rozmiarach NxN (gdzie rozmiar N jest wartością przechowywaną w zmiennej, którą można modyfikować). Tablica powinna być wypełniona znakami "0" (zero), jedynie na krawędziach oraz przekątnych powinny znajdować się znaki "X". Do wypełnienia tablicy zastosuj pętle for, natomiast do wyświetlenia jej zawartości pętle while. Używaj w kodzie tam gdzie to możliwe operatorów inkrementacji. Przykładowy wydruk programu:

```
X X X X X X X X
X X 0 0 0 0 X X
X 0 X 0 0 X 0 X
X 0 0 X X 0 0 X
X 0 0 X X 0 0 X
X 0 X 0 0 X 0 X
X X 0 0 0 0 X X
X X X X X X X X
```

plik Zad1.java

```java
public class Zad1 {
    public static void main(String[] args) {
        int n = 8;
        char[][] array = new char[n][n];

        //wypełniamy tablicę
        for (int i = 0; i < n; i++) {
            for (int j = 0; j < n; j++) {
                //wypełnienie X na krawędziach tablicy
                if (i == 0 || j == 0 || i == n - 1 || j == n - 1) {
                    array[i][j] = 'X';
                //wypełnienie X na przekątnych
                } else if (i == j || i == n - j - 1) {
                    array[i][j] = 'X';
                //wypełnienie 0 w pozostałych miejscach
                } else {
                    array[i][j] = '0';
                }
            }
        }

        //liczniki pętli
        int i = 0, j = 0;
        //wyświetlanie tablicy
        while (i < n) {
            while (j < n) {
                System.out.print(array[i][j] + " ");
                j++;
            }
            j = 0;
            System.out.println(); //nowa linia na końcu wiersza
            i++;
        }
    }
}
```

3.4 Programowanie obiektowe 2

W tej lekcji dowiesz się:

- W jaki sposób odczytywać dane od użytkownika
- Na czym polega dziedziczenie
- Do czego wykorzystujemy słowo kluczowe super
- Czym jest przesłanianie metod
- Co to jest polimorfizm
- Do czego służy operator instanceof
- Czym są klasy abstrakcyjne i interfejsy
- Co oznacza słowo static

3.4.1 Odbieranie danych od użytkownika

W naszych dotychczasowych programach skupialiśmy się na poznawaniu podstawowych mechanizmów języka Java, jednak brakowało w nich interakcji z użytkownikiem. Biblioteka wejścia/wyjścia w Javie jest dosyć rozbudowana jednak w tej części kursu poznamy podstawowe sposoby interakcji człowieka z komputerem.

Najprostszą klasą, która pozwoli odczytywać dane od użytkownika jest **Scanner**. Posiada ona zestaw użytecznych metod, które pozwolą nam wczytać zarówno liczby jak i napisy. Klasa ta jest dosyć uniwersalna, więc w zależności od tego co przekażemy w konstruktorze podczas inicjalizacji możemy za pomocą jej obiektu zarówno odczytywać dane od użytkownika, które będą wprowadzane z klawiatury, ale także odczytywać informacje zawarte w plikach.

Inicjalizacja obiektu Scanner w przypadku, gdy chcemy odczytać dane od użytkownika wygląda następująco:

```
import java.util.Scanner;

public class Example {
    Scanner sc = new Scanner(System.in);
}
```

Przede wszystkim, aby korzystać z klasy Scanner w swoim kodzie, musisz ją najpierw zaimportować z pakietu java.util (np. wykorzystując skrót Ctrl+Shift+O w eclipse). Obiekt tworzony jest przez przekazanie w konstruktorze standardowego strumienia wejścia, którym jest *System.in*.

Następnie do odczytu danych należy skorzystać z jednej z metod zawartych w klasie Scanner:

- nextInt() - odczytanie kolejnej liczby typu int
- nextDouble() - odczytanie kolejnej liczby typu double (uwaga, domyślny separator dziesiętny jest zależny od ustawień maszyny wirtualnej, w Polsce domyślnie liczby należy wprowadzać rozdzielone przecinkiem)
- nextLine() - wczytanie kolejnego wiersza tekstu (String) zakończonego znakiem nowej linii ‘n’
- analogicznie dla innych typów danych istnieją metody takie jak nextBoolean(), nextLong() itp.

Ważne jest to, że w przypadku, gdy odczytywać będziemy liczby, w buforze nadal pozostanie znak nowej linii, który należy wczytać za pomocą metody *nextLine()*. Przykład:

```
import java.util.Scanner;

public class SimpleInput {
    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);

        System.out.println("Podaj pierwszą liczbę: ");
        //wczytujemy wartość zmiennej z klawiatury
        int num1 = sc.nextInt();
        //usuwamy znak nowej linii z bufora
        sc.nextLine();

        System.out.println("Podaj drugą liczbę: ");
        //wczytujemy drugą liczbę
        int num2 = sc.nextInt();
        //usuwamy znak nowej linii z bufora
        sc.nextLine();

        System.out.println("Suma podanych liczb to: " + (num1 + num2));

        System.out.println("Jak masz na imię? ");
        //wczytujemy napis
        String name = sc.nextLine();

```

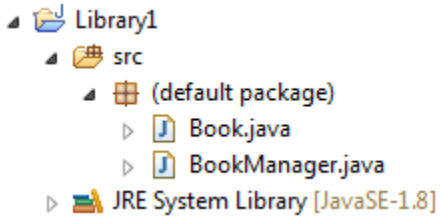
```
        System.out.println("Witaj " + name + "!");

        //po zakończonym odczycie zamykamy strumień wejścia
        sc.close();
    }
}
```

Uwaga: Pamiętaj, że po zakończeniu pracy ze strumieniami wejścia lub wyjścia, szczególnie odczytu plików należy je zamykać za pomocą metody close(). Analogicznie będzie należało postępować z operacjami wyjścia, czyli np. zapisem danych do plików.

Ćwiczenie *(10 min)*

Napisz prosty program do zarządzania książkami biblioteki. Powinien się on składać z dwóch klas:

- Book - klasa reprezentująca pojedynczą książkę. Powinna posiadać pola reprezentujące numer ISBN, tytuł oraz autora
- BookManager - główna klasa aplikacji, w której wczytasz od użytkownika dotyczące 1 książki, utworzysz na ich podstawie obiekt klasy Book i wyświetlisz odpowiednie informacje na ekranie. Zadbaj o utworzenie odpowiednich konstruktorów.



plik Book.java

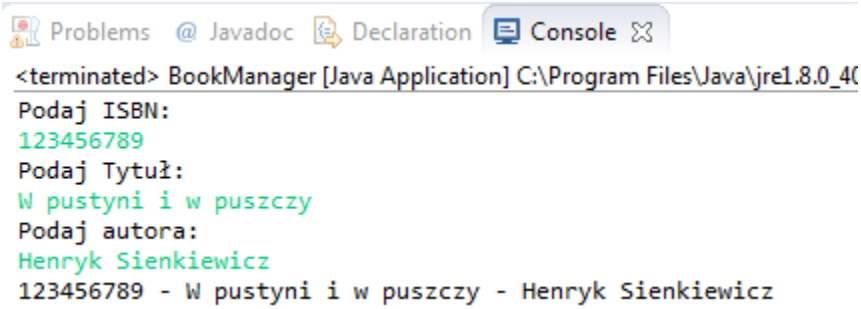
```
public class Book {
    String isbn;
    String title;
    String author;

    Book(String isbn, String title, String author) {
        this.isbn = isbn;
        this.title = title;
        this.author = author;
    }

    String getBookInfo() {
        return isbn + " - " + title + " - " + author;
    }
}
```

plik BookManager.java

```
import java.util.Scanner;

public class BookManager {
    public static void main(String[] args) {
        //tworzymy obiekt do odczytu danych
        Scanner sc = new Scanner(System.in);

        //prosimy użytkownika o podanie odpowiednich danych
        System.out.println("Podaj ISBN: ");
```

```
10          String isbn = sc.nextLine();
11          System.out.println("Podaj Tytuł: ");
12          String title = sc.nextLine();
13          System.out.println("Podaj autora: ");
14          String author = sc.nextLine();
15
16          //zamykamy strumień wejścia
17          sc.close();
18
19          //tworzymy obiekt Book i wyświetlamy informacje na ekranie
20          Book book = new Book(isbn, title, author);
21          System.out.println(book.getBookInfo());
22      }
23  }
```

Przykładowy wynik działania programu:



3.4.2 Dziedziczenie

W naszej aplikacji możemy teraz przechowywać i wczytywać od użytkownika informacje o książkach, jednak warto zauważyć, że przecież w bibliotece oprócz książek są także komiksy, gazety, ogólnie rzecz ujmując magazyny. W tym momencie należałoby stworzyć osobną klasę Magazine, która przechowywać będzie również numer ISBN, tytuł, oraz np. wydawnictwo. Problemem jest to, że pewne dane zaczynają się tutaj powielać, a tego w programowaniu zdecydowanie powinniśmy unikać.

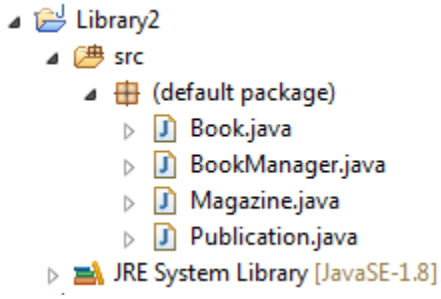
W celu rozwiązania m.in. tego problemu powstał paradygmat programowania obiektowego, które pozwala budować hierarchię klas, dającą możliwość tworzenia kodu, który może być wykorzystywany wielokrotnie i być bazą do tworzenia jeszcze kolejnych klas.

Na powyższym diagramie widać bazową klasę **Publication**, po której dziedziczą dwie kolejne klasy: **Book** oraz

Magazine. Różnią się one tym, że książka posiada najczęściej jednego autora (np. Henryk Sienkiewicz), natomiast w przypadku gazety w tym miejscu pojawi się wydawnictwo (np. Ringier Axel Springer).

Informacja: Jeżeli istnieje pewna klasa **A**, a poniej dziedziczy pewna klasa **B**, to klasa **B** przejmuje wszystkie widoczne cechy klasy **A**.

Powyższa regułka oznacza, że w przypadku, gdy spojrzymy na powyższy diagram, klasa **Book**, czy **Magazine** będą posiadały także pola z klasy **Publication**, czyli *isbn* oraz *title*. W Javie dziedziczenie można osiągnąć stosując słowo kluczowe **extends** w definicji klasy, np.:



plik Publication.java

```
public class Publication {
    String isbn;
    String title;

    Publication(String isbn, String title) {
        this.isbn = isbn;
        this.title = title;
    }

    String getInfo() {
        return title + " - " + isbn;
    }
}
```

plik Book.java

```
public class Book extends Publication {
    String author;

    Book(String isbn, String title, String author) {
        super(isbn, title);
        this.author = author;
    }

    String getInfo() {
        return super.getInfo() + " - " + author;
    }
}
```

plik Magazine.java

```
public class Magazine extends Publication {
    String publisher;

    Magazine(String isbn, String title, String publisher) {
```

```
        super(isbn, title);
        this.publisher = publisher;
    }

    String getInfo() {
        return super.getInfo() + " - " + publisher;
    }
}
```

Zwróć uwagę, że pomimo iż w klasach *Book* oraz *Magazine* nie zadeklarowaliśmy pól *isbn* oraz *title* to mamy do nich dostęp w konstruktorze, czy metodach *getBookInfo()* i *getMagazineInfo()*, ponieważ dziedziczą one te cechy z klasy Publication.

Kolejną nowością jest zastosowanie specjalnej konstrukcji **super()**. Działa ona w sposób podobny do słowa kluczowego *this*, którego używaliśmy w przypadku, gdy posiadaliśmy kilka przeciążonych wersji konstruktora z tą różnicą, że wywołuje konstruktor nadklasy z odpowiednimi parametrami.

Ostatnia rzecz dotyczy **przesłaniania** metod. W klasie Publication zdefiniowaliśmy metodę *getInfo()*. Jeżeli w klasie dziedziczącej (Book lub Magazine) zdefiniujemy metodę o takiej samej sygnaturze, to powiemy, że przesłania ona oryginalną metodę *getInfo()*. W celu wywołania metody nadklasy należy posłużyć się w takiej sytuacji zapisem **super.getInfo()**.

Informacja: Pierwszą, niejawną instrukcją jaka jest wykonywana w podklasie pewnej klasy bazowej jest wywołanie konstruktora nadklasy poprzez **super()**. Jeżeli w klasie bazowej nie jest zdefiniowany konstruktor bezparametrowy to należy jawnie wywołać konstruktor z odpowiednimi argumentami poprzez zapis **super(lista_parametrow)**. Jeżeli chcesz natomiast wywołać w którejś z metod metodę z nadklasy wykorzystaj zapis **super.nazwaMetody(parametry)**.

Informacja: Jeżeli chcesz zaznaczyć, że jakaś metoda jest przesłonięta możesz zastosować dodatkową adnotację *@Override*. W wielu sytuacjach będzie ona automatycznie wygenerowana przez eclipse, warto więc rozumieć co oznacza.

```
@Override
String getInfo() {
    return super.getInfo() + " - " + author;
}
```

Ćwiczenie *(10 min)* W klasie BookManager utwórz tablice do przechowywania książek oraz magazynów. W każdej tablicy utwórz przynajmniej po jednym obiekcie danego typu, a następnie wyświetl je na ekranie.

plik BookManager.java

```
import java.util.Scanner;

public class BookManager {
    public static void main(String[] args) {
        // tworzymy obiekt do odczytu danych
        Scanner sc = new Scanner(System.in);

        // tworzymy tablice
        Book[] books = new Book[10];
        Magazine[] magazines = new Magazine[10];

        // prosimy użytkownika o podanie informacji o książce
        System.out.println("Podaj ISBN książki: ");
        String isbn = sc.nextLine();
```

```
15          System.out.println("Podaj Tytuł książi: ");
16          String title = sc.nextLine();
17          System.out.println("Podaj autora książki: ");
18          String author = sc.nextLine();
19
20          books[0] = new Book(isbn, title, author);
21
22          // prosimy użytkownika o podanie informacji o magazynie
23          System.out.println("Podaj ISBN magazynu: ");
24          isbn = sc.nextLine();
25          System.out.println("Podaj Tytuł magazynu: ");
26          title = sc.nextLine();
27          System.out.println("Podaj wydawcę magazynu: ");
28          String publisher = sc.nextLine();
29
30          magazines[0] = new Magazine(isbn, title, publisher);
31
32          // zamykamy strumień wejścia
33          sc.close();
34
35          // tworzymy obiekt Book i wyświetlamy informacje na ekranie
36          System.out.println(books[0].getInfo());
37          System.out.println(magazines[0].getInfo());
38      }
39  }
```

W porównaniu do wcześniejszego kodu zyskaliśmy teraz możliwość przechowywania informacji zarówno o książkach jak i innego rodzaju publikacjach. Ponieważ przechowujemy je w tablicach to dodatkowo dane są teraz bardziej spójne.

3.4.3 Polimorfizm

Wcześniej wspomnieliśmy, że dziedziczenia warto używać między innymi po to, żeby zaoszczędzić konieczności powielania tego samego kodu. Problem w tym, że gdy tak jak w powyższym przykładzie stosujemy tablice oddzielnych typów to operacje na nich i tak będą powielane, np:

```
System.out.println("Książki: ");
for(int i=0; i < books.length; i++) {
    if(books[1] != null)
        System.out.println(books[i].getInfo());
}

System.out.println("Magazyny: ");
for(int i=0; i < magazines.length; i++) {
    if(magazines[1] != null)
        System.out.println(magazines[i].getInfo());
}
```

Lepiej by było tak naprawdę przechowywać wszystkie te dane w jednej wspólnej tablicy typu *Publication*, a następnie ewentualnie rozpoznać, czy dany element tablicy jest typu Book, czy Magazine.

Efekt taki można osiągnąć dzięki zastosowaniu **polimorfizmu** lub inaczej mówiąc wielopostaciowości. Polega to na tym, że do ogólnej, wspólnej referencji można przypisać obiekty różnych typów, które po typie tej referencji dziedziczą.

W naszym przypadku klasy Book i Magazine dziedziczą po klasie Publication, więc jak najbardziej możliwe jest zapisanie:

```
Publication pub1 = new Book(argumenty_konstruktora);
Publication pub2 = new Magazine(argumenty_konstruktora);
```

W podobny sposób możemy także utworzyć tablicę typu Publication, która będzie przechowywała zarówno książki jak i magazyny. Pamiętać należy jednak o dwóch rzeczach:

1. Zawsze mamy dostęp jedynie do metod z typu referencji. Jeżeli w klasie Book zdefiniujemy dodatkową metodę changeBookTitle(), a referencja będzie typu Publication, to nie będziemy mieli dostępu do takiej metody. Jeśżeli chcesz uzyskać dostęp do wszystkich metod (również tych nie zdefiniowanych w typie nadrzędnym) musisz skorzystać z rzutowania typu. Rzutowanie polega na wykorzystaniu nawiasu, np.:

```
Publication pub = new Magazine(...);
((Magazine)pub).metodaZKlasyMagazine();
```

2. Metody będą wywoływane na rzecz typu obiektu, a nie typu referencji. Jeżeli zapisaliśmy *Publication pub1 = new Book(argumenty_konstruktora);* to wywołując metodę pub1.getInfo() wywołamy jej wersję z klasy Book, a nie Publication.

Ćwiczenie *(5 min)*

Przerób klasę BookManager w taki sposób, aby przechowywać książki oraz magazyny w jednej wspólnej tablicy typu Publication[].

plik BookManager.java

```
import java.util.Scanner;

import java.util.Scanner;

public class BookManager {
    public static void main(String[] args) {
        // tworzymy obiekt do odczytu danych
        Scanner sc = new Scanner(System.in);

        // tworzymy tablice
        Publication[] publications = new Publication[10];

        // prosimy użytkownika o podanie informacji o książce
        System.out.println("Podaj ISBN książki: ");
        String isbn = sc.nextLine();
        System.out.println("Podaj Tytuł książi: ");
        String title = sc.nextLine();
        System.out.println("Podaj autora książki: ");
        String author = sc.nextLine();

        publications[0] = new Book(isbn, title, author);

        // prosimy użytkownika o podanie informacji o magazynie
        System.out.println("Podaj ISBN magazynu: ");
        isbn = sc.nextLine();
        System.out.println("Podaj Tytuł magazynu: ");
        title = sc.nextLine();
        System.out.println("Podaj wydawcę magazynu: ");
        String publisher = sc.nextLine();

        publications[1] = new Magazine(isbn, title, publisher);

        // zamykamy strumień wejścia
        sc.close();

```

```
        // wyświetlamy informacje na ekranie
        System.out.println("Książki i magazyny: ");
        for (Publication p: publications) {
            if (p != null)
                System.out.println(p.getInfo());
        }
    }
}
```

Dzięki uproszczeniu sposobu przechowywania danych wyświetlanie danych odbywa się w jednej tylko pętli, dane są bardziej spójne, a kod bardziej przejrzysty.

3.4.4 Operator instanceof

W poprzednim przykładzie brakuje w tej chwili trochę informacji o tym, czy dana pozycja w tablicy *publications* jest typu Book, czy Magazine. W niektórych sytuacjach chcielibyśmy mieć taką informację w celu zwiększenia przejrzystości. W Javie możemy sprawdzić rzeczywisty typ obiektu (nie typ referencji) za pomocą operatora **instanceof**. Jego składnia jest następująca:

```
obiekt instanceof NazwaKlasy
```

np.

```
Publication book = new Book("123456789", "W pustyni i w puszczy", "Henryk Sienkiewicz");
if(book instanceof Book) {
    System.out.println("To jest książka");
}
```

Na ekranie zobaczymy tekst “To jest ksiażka”.

Sprawdzenie obiektu za pomocą operatora instanceof zwraca w wyniku true lub false, więc jak widać w powyższym przykładzie, może on być zastosowany jako warunek w instrukcji if.

Ćwiczenie *(5 min)*

Przerób kod klasy BookManager w taki sposób, aby dane były wyświetlane w formie:

```
Książka: 123456789 - W pustyni i w puszczy - Henryk Sienkiewicz
Magazyn: 987654321 - Wprost - Wydawnictwo Wprost
```

plik BookManager.java

```
import java.util.Scanner;

public class BookManager {
    public static void main(String[] args) {
        // tworzymy obiekt do odczytu danych
        Scanner sc = new Scanner(System.in);

        // tworzymy tablice
        Publication[] publications = new Publication[10];

        // prosimy użytkownika o podanie informacji o książce
        System.out.println("Podaj ISBN książki: ");
        String isbn = sc.nextLine();
        System.out.println("Podaj Tytuł książi: ");
        String title = sc.nextLine();
        System.out.println("Podaj autora książki: ");
        String author = sc.nextLine();
```

```
18
19          publications[0] = new Book(isbn, title, author);
20
21          // prosimy użytkownika o podanie informacji o magazynie
22          System.out.println("Podaj ISBN magazynu: ");
23          isbn = sc.nextLine();
24          System.out.println("Podaj Tytuł magazynu: ");
25          title = sc.nextLine();
26          System.out.println("Podaj wydawcę magazynu: ");
27          String publisher = sc.nextLine();
28
29          publications[1] = new Magazine(isbn, title, publisher);
30
31          // zamykamy strumień wejścia
32          sc.close();
33
34          // wyświetlamy informacje na ekranie
35          System.out.println("Książki i magazyny: ");
36          for (Publication p: publications) {
37              if (p != null)
38                  if(p instanceof Book) {
39                      System.out.println("Książka: " + p.getInfo());
40                  } else if(p instanceof Magazine) {
41                      System.out.println("Magazyn: " + p.getInfo());
42                  }
43          }
44      }
45  }
```

3.4.5 Klasy abstrakcyjne i interfejsy

Nasza aplikacja jest już prawie gotowa, jednak warto się zastanowić nad jeszcze jedną rzeczą. Stworzyliśmy klasę Publication, która jest klasą bazową dla dwóch konkretnych typów danych, które wykorzystujemy. Ponieważ nie chcemy używać obiektów klasy Publication bezpośrednio, powinniśmy z niej zrobić jedynie pewną warstwę abstrakcji, czyli klasę, która jest jedynie klasą bazową dla innych typów, ale nie można tworzyć jej obiektów bezpośrednio.

W Javie można to osiągnąć wykorzystując **klasę abstrakcyjną**. Jeżeli jakaś klasa ma być abstrakcyjna, należy w jej sygnaturze dodać o tym informację za pomocą słowa kluczowego **abstract**.

plik Publication.java

```
1   public abstract class Publication {
2       String isbn;
3       String title;
4
5       Publication(String isbn, String title) {
6           this.isbn = isbn;
7           this.title = title;
8       }
9
10      String getInfo() {
11          return title + " - " + isbn;
12      }
13  }
```

Klasy abstrakcyjne mają dwie ważne właściwości, o których musisz pamiętać:

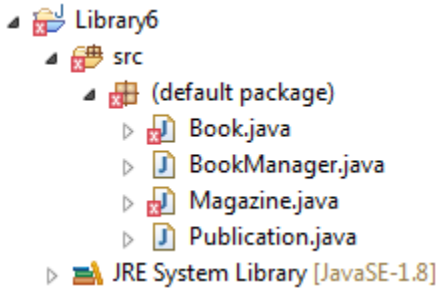
1. Nie można utworzyć obiektu klasy abstrakcyjnej za pomocą operatora new. Typ abstrakcyjny może być jedynie typem referencji dla typów bardziej sprecyzowanych (dziedziczących po tej klasie abstrakcyjnej).
2. W klasie abstrakcyjnej mogą być zdefiniowane **metody abstrakcyjne**, czyli metody, które określają jedynie sygnaturę metody, ale nie posiadają implementacji. Jeżeli jakaś klasa posiada chociaż jedną metodę abstrakcyjną, to musi być oznaczona jako klasa abstrakcyjna. Każda klasa, która dziedziczy po klasie abstrakcyjnej musi posiadać konkretną implementację każdej metody abstrakcyjnej.

Przykład metody abstrakcyjnej:

plik Publication.java

```
public abstract class Publication {
    String isbn;
    String title;

    Publication(String isbn, String title) {
        this.isbn = isbn;
        this.title = title;
    }

    String getInfo() {
        return title + " - " + isbn;
    }

    abstract void printInfo();
}
```

Jak widzisz metoda *printInfo()* oznaczona jest słowek abstract. Nie posiada ona żadnej implementacji - nie posiada nawet nawiasów klamrowych i zakończona jest średnikiem.



Jak widzisz dodanie metody abstrakcyjnej powoduje błędy w naszym projekcie w klasach dziedziczących po Publication. W celu ich wyeliminowania powinniśmy zaimplementować metodę *printInfo()* w tych klasach, np.:

plik Book.java

```
public class Book extends Publication {
    String author;

    Book(String isbn, String title, String author) {
        super(isbn, title);
        this.author = author;
    }

    String getInfo() {
        return super.getInfo() + " - " + author;
    }

    @Override
    void printInfo() {
```

```
        System.out.println(getInfo());
    }
}
```

analogicznie w klasie Magazine:

plik Magazine.java

```
public class Magazine extends Publication {
    String publisher;

    Magazine(String isbn, String title, String publisher) {
        super(isbn, title);
        this.publisher = publisher;
    }

    String getInfo() {
        return super.getInfo() + " - " + publisher;
    }

    @Override
    void printInfo() {
        System.out.println(getInfo());
    }
}
```

W Javie istnieją także klasy, które są w pełni abstrakcyjne nazywane **interfejsami**, czyli posiadają jedynie metody abstrakcyjne oraz ewentualnie zdefiniowane stałe wartości (oznaczone jako public static). Interfejsy definiujemy za pomocą słowa kluczowego **interface**, a implementujemy je (analogia do dziedziczenia) za pomocą słowa kluczowego **implements**. Na tym etapie nie będziemy się zagłębiali w stosowanie interfejsów w swoich programach, spójrz jedynie na przykład i zapamiętaj kilka rzeczy zapisanych poniżej.

Przykład interfejsu:

```
public interface Moveable {
    void move();

    void stop();
}
```

Klasa implementująca interfejs Moveable:

```
public class Car implements Moveable {

    public void move() {
        System.out.println("Samochód start");
    }

    public void stop() {
        System.out.println("Samochód stop");
    }
}
```

Ponieważ wszystkie metody interfejsu muszą być abstrakcyjne, to nie jest konieczne jawne używanie słowa kluczowego abstract - jest ono niejawnie dodawane automatycznie.

Zapamiętaj

1. W Javie można dziedziczyć tylko po jednej klasie (extends), nawet jeśli są to klasy abstrakcyjne.
2. Możesz implementować dowolną liczbę interfejsów (implements).

3.4.6 Dodatek - składowe statyczne

Jako dodatek tej lekcji wytłumaczymy jescze krótko co oznacza słowo kluczowe **static**. W niektorych sytuacjach chcielibyśmy mieć dostęp do metody, czy pola klasy bez konieczności tworzenia obiektu. Przykładem takiej metody jest dobrze nam już znana metoda *main()*.

Jeżeli zdefiniujemy jakieś pole lub metodę jako statyczne, to możemy się do nich odwoływać bez tworzenia obiektu takiej klasy poprzez konstrukcję NazwaKlasy.nazwaPolaStatycznego lub NazwaKlasy.metodaStatyczna().

Najważniejszą rzeczą dotyczącą składowych statycznych klasy jest to, że w metodach statycznych możemy odwoływać się jedynie do innych elementów statycznych.

3.5 Wyjątki i kolekcje

W tej lekcji dowiesz się:

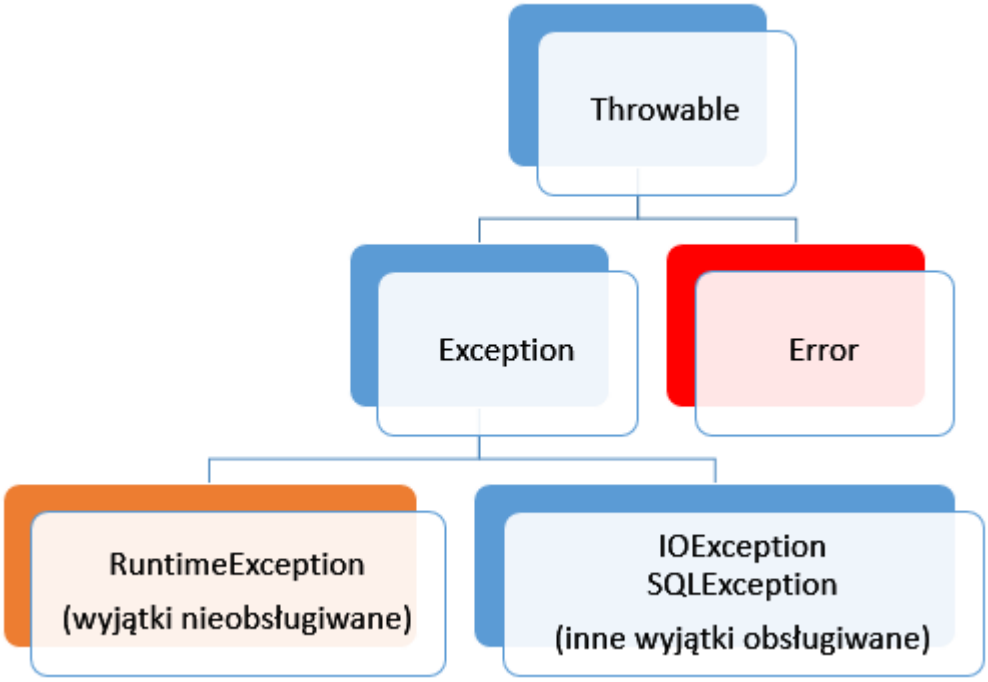
- Czym są wyjątki i jak je obsługiwać
- Czym są kolekcje (listy, zbiory i mapy)
- Co to oznacza, że kolekcje są typem generycznym
- Jakie są typy opakowujące
- Co oznaczają pojęcia autoboxing i unboxing

3.5.1 Wyjątki

W każdym programie występują pewne sytuacje wyjątkowe, które jednak można przewidzieć i w odpowiedni sposób obsłużyć. Nasza aplikacja powinna być odporna przede wszystkim na błędy, które mogą wyniknąć nie z naszej programistycznej winy, czyli np. zamknięcie połączenia sieciowego w trakcie komunikacji z innym komputerem podłączonym do sieci, błąd odczytu pliku, albo zwyczajnie wprowadzenie przez użytkownika danych w niepoprawnym formacie (napis zamiast liczby).

We wszystkich takich przypadkach zostają wygenerowane wyjątki, czyli specjalne obiekty, które mówią o tym co poszło nie tak jak powinno. W Javie istnieją dwa sposoby na obsługę wyjątków, które w tym miejscu krótko omówimy.

Hierarchia dziedziczenia klas wyjątków wygląda tak jak na poniższym schemacie.

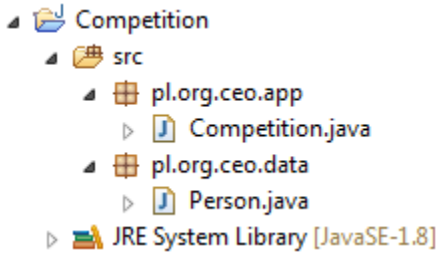


Nie musisz tego zapamiętywać, ponieważ najważniejszą różnicą pomiędzy poszczególnymi typami wyjątków jest to, czy musimy je obsługiwać, czy też nie. Obsługę wyjątków w niektórych sytuacjach wymusi na Tobie eclipse. W innych sytuacjach warto spojrzeć na sygnatury metod w dokumentacji, ponieważ to w nich znajdziesz informację, czy może ona generować jakiś wyjątek.

Zacznijmy od prostej aplikacji, która posłuży nam do omówienia zagadnień tej lekcji.

Ćwiczenie *(10 minut)*

> Przeanalizuj kod poniższego programu, który służy do zbierania danych o uczestnikach dowolnego konkursu, uruchom go oraz przetestuj dostępne opcje.



plik Person.java

```
package pl.org.ceo.data;
public class Person {
    private String firstName;
    private String lastName;
    private String pesel;
    private int age;

    public String getFirstName() {
        return firstName;
```

```
    }
    public void setFirstName(String firstName) {
        this.firstName = firstName;
    }
    public String getLastName() {
        return lastName;
    }
    public void setLastName(String lastName) {
        this.lastName = lastName;
    }
    public String getPesel() {
        return pesel;
    }
    public void setPesel(String pesel) {
        this.pesel = pesel;
    }
    public int getAge() {
        return age;
    }
    public void setAge(int age) {
        this.age = age;
    }

    @Override
    public String toString() {
        return firstName + " " + lastName + " - " + pesel + ", " + age + " lat";
    }
}
```

Zwróć uwagę na to, że pola tej klasy oznaczyliśmy jako prywatne oraz wygenerowaliśmy dla nich zestaw dwóch metod - tzw. getterów i setterów, które pozwalają je odczytać poza tą klasą. Jest to ogólnie przyjęta konwencja, do której należy się przyzwyczaić, ponieważ spotkamy się z nią w Javie na każdym kroku.

Druga nowość to przesłonięcie metody *toString()*. Oznaczona jest jako Override, czyli przesłania metodę *toString()* z klasy nadrzędnej. Możliwe, że myślisz - ale jak to, przecież klasa Person nie dziedziczy po żadnej klasie (brak extends). Otóż w Javie niejawnie każda klasa dziedziczy po specjalnej klasie Object. Metoda toString() to ogólnie przyjęta metoda, która zwraca opisową formę obiektu.

plik Competition.java

```
package pl.org.ceo.app;
import java.util.Scanner;

import pl.org.ceo.data.Person;

public class Competition {

    public static final int ADD_COMPETITOR = 0;
    public static final int PRINT_ALL = 1;
    public static final int EXIT = 2;

    private static Person[] competitors;
    private static int competitorsNumber;

    public static void main(String[] args) {
        competitors = new Person[100];
        competitorsNumber = 0;
        Scanner sc = new Scanner(System.in);
        int option = 0;
```

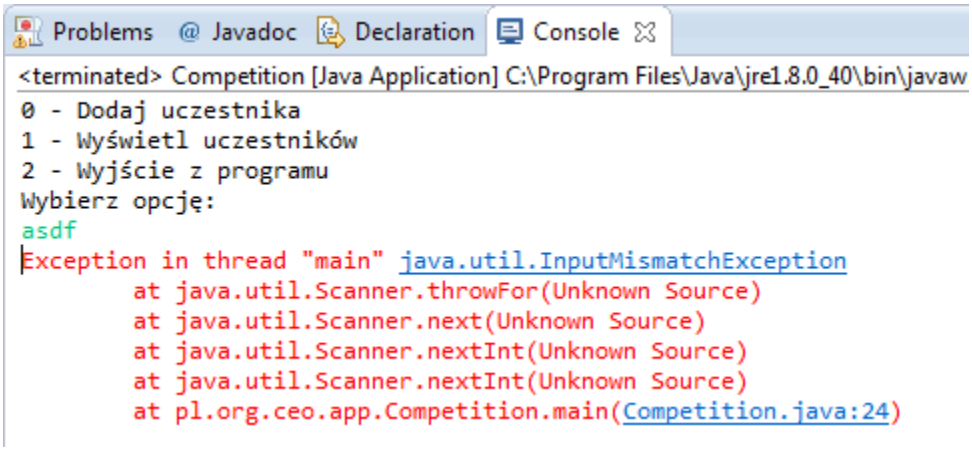
```
20
21          do {
22              printOptions();
23              option = sc.nextInt();
24              sc.nextLine();
25
26              switch (option) {
27              case ADD_COMPETITOR:
28                  addCompetitor(sc);
29                  break;
30              case PRINT_ALL:
31                  printCompetitors();
32                  break;
33              case EXIT:
34                  break;
35              }
36          } while (option != EXIT);
37
38      }
39
40      private static void printCompetitors() {
41          System.out.println("--------------------");
42          System.out.println("Lista uczestników:");
43          for(int i=0; i < competitorsNumber; i++) {
44              System.out.println(competitors[i].toString());
45          }
46      }
47
48      private static void addCompetitor(Scanner sc) {
49          if(competitorsNumber < competitors.length) {
50              Person person = new Person();
51              System.out.println("--------------------");
52              System.out.println("Dodawanie nowego uczestnika: ");
53              System.out.println("Imię: ");
54              person.setFirstName(sc.nextLine());
55              System.out.println("Nazwisko");
56              person.setLastName(sc.nextLine());
57              System.out.println("PESEL:");
58              person.setPesel(sc.nextLine());
59              System.out.println("Wiek:");
60              person.setAge(sc.nextInt());
61              sc.nextLine();
62
63              competitors[competitorsNumber] = person;
64              competitorsNumber++;
65          } else {
66              System.out.println("Osiągnięto maksymalną liczbę uczestników");
67          }
68      }
69
70      private static void printOptions() {
71          System.out.println("--------------------");
72          System.out.println("Dostępne opcje: ");
73          System.out.println(ADD_COMPETITOR + " - Dodaj uczestnika");
74          System.out.println(PRINT_ALL + " - Wyświetl uczestników");
75          System.out.println(EXIT + " - Wyjście z programu");
76          System.out.println("Wybierz opcję: ");
77      }
```

```
78 }
```

Klasa **Person** to nasz nośnik danych. Przechowuje ona informacje dotyczące imienia, nazwiska, nr. PESEL oraz wieku uczestnika. W klasie Competition znajduje się główna logika aplikacji, w której dajemy użytkownikowi jedną z trzech opcji, czyli dodanie nowego uczestnika, wyświetlenie wszystkich uczestników lub wyjście z programu. Po wybraniu opcji wywoływana jest odpowiednia metoda, w której wyświetlamy odpowiednie komunikaty, odbieramy dane od użytkownika i na ich podstawie tworzymy kolejne obiekty Person lub wyświetlamy już te dodane. Wszystkie składowe klasy zostały oznaczone jako statyczne, więc nie jest wymagane tworzenie obiektu klasy Competition w celu wywoływania metod, czy odwoływania się do poszczególnych pól z metody *main()*. Elementy oznaczone jako *public static final* nazywać będziemy stałymi.

3.5.2 Wyjątki - blok try catch

Miejscem, w którym w naszym programie mogą pojawić się problemy, są związane głównie z odbiorem danych od użytkownika - w końcu nie jesteśmy w stanie przewidzieć, czy zamiast konkretnej liczby nie wprowadzi on dla żartu napisu "asdf". Jeżeli coś takiego się wydarzy, zostanie wtedy wygenerowany wyjątek fazy wykonania o nazwie *InputMismatchException*, który jest spowodowany tym, że metoda *nextInt()* nie jest przygotowana na odbiór danych typu String.



Aplikacja w tej sytuacji przestaje działać, a dane zostają utracone. W Javie istnieje jednak dosyć prosty mechanizm obsługi sytuacji wyjątkowych za pomocą bloku try catch. Jego ogólna konstrukcja wygląda następująco:

```
try {
  //instrukcje mogące wygenerować wyjątek
} catch(typ_wyjatku nazwa_zmiennej) {
  //instrukcje, które zostaną wykonane po wygenerowaniu wyjątku
} finally {
  //instrukcje, które wykonają się zawsze, niezależnie, czy wyjątek wystąpi, czy też nie (blok opcjo
}
```

W naszym kodzie w bloku try można oczywiście umieścić odczyt danych, czyli wywołanie metody *nextInt()*. Można także w nim umieścić dużo większy fragment kodu, jednak warto się zastanowić, czy na pewno się to opłaca i czy pomoże nam to w identyfikacji konkretnego problemu.

plik Competition.java

```
1 public class Competition {
2
3     //reszta kodu bez zmian
4
5     public static void main(String[] args) {
```

```
        competitors = new Person[100];
        competitorsNumber = 0;
        Scanner sc = new Scanner(System.in);
        int option = 0;

        do {
            printOptions();
            try {
                option = sc.nextInt();
                sc.nextLine();
            } catch(InputMismatchException exc) {
                sc.nextLine(); //"zjadamy" znak nowej linii z bufora
                System.out.println("--------------------");
                System.out.println("Dane w nieprawidłowym formacie ");
                continue; //przejście do kolejnej iteracji pętli
            }

            switch (option) {
            case ADD_COMPETITOR:
                addCompetitor(sc);
                break;
            case PRINT_ALL:
                printCompetitors();
                break;
            case EXIT:
                break;
            }
        } while (option != EXIT);

    }

    //reszta kodu bez zmian
}
```

Jeżeli użytkownik wprowadzi teraz niepoprawne dane podczas przypisania *option = sc.nextInt();* wygenerowany zostanie wyjątek, który jednak obsługujemy w bloku try-catch, a tym samym możemy zapobiec zakończeniu programu. Ponieważ po wygenerowaniu wyjątku sterowanie programu jest przekazywane natychmiast do bloku catch, musimy w pierwszej kolejności pozbyć się z bufora znaku nowej linii, który pozostaje po wywołaniu metody nextInt(). Następnie wyświetlamy komunikat o błędzie i przechodzimy do kolejnej iteracji pętli dzięki instrukcji continue. Będzie się tak działo za każdym razem, gdy użytkownik wprowadzi wartość niezgodną z typem int.

Blok finally jest w tym przypadku zbędny. Przydatny będzie natomiast, gdy będziemy chcieli zamknąć strumień, czy plik niezależnie od tego, czy błąd wystąpił, czy nie.

3.5.3 Wyjątki - deklaracja throws

Istnieją takie sytuacje, w których nie chcemy obsługiwać wyjątków za pomocą bloku try-catch, bo zwyczajnie nie będziemy w stanie nic z tym problemem zrobić. W takiej sytuacji możemy przekazać wyjątek wyżej i dać osobie korzystającej z naszego kodu możliwość zadecydowania, czy chce obsłużyć dany wyjątek, czy też również nic z nim nie robić. Przykładem takiego działania jest metoda *nextInt()* klasy Scanner - może ona generować trzy różne wyjątki, które możemy obsłużyć tak jak w powyższym kodzie, albo je pominąć, tak jak robiliśmy to wcześniej.

plik Competition.java

```
package pl.org.ceo.app;

import java.util.InputMismatchException;
```

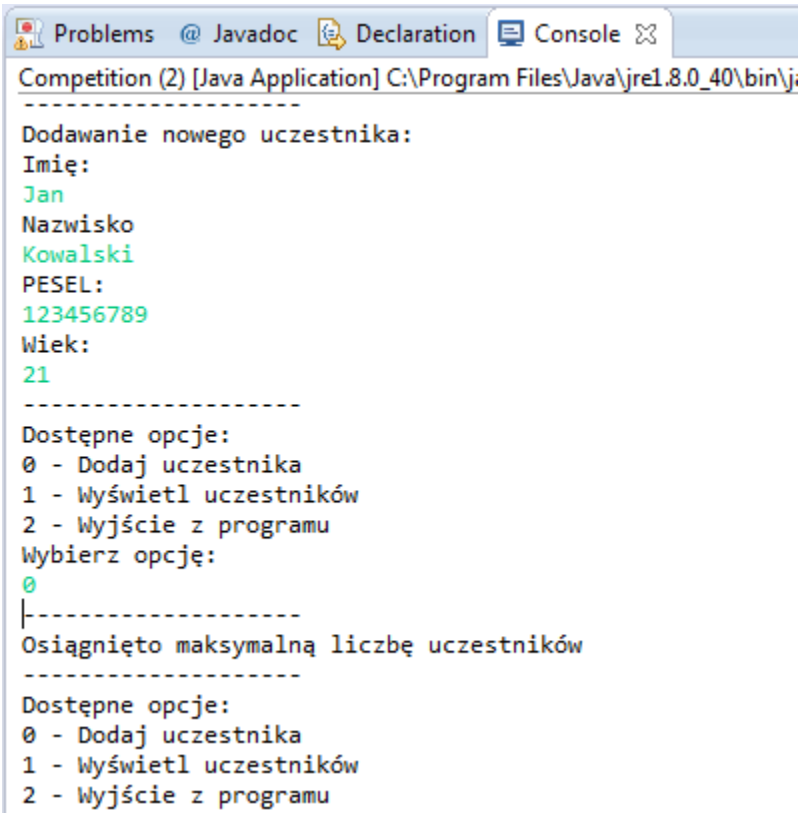
```
import java.util.Scanner;

import pl.org.ceo.data.Person;

public class Competition {

    public static final int ADD_COMPETITOR = 0;
    public static final int PRINT_ALL = 1;
    public static final int EXIT = 2;

    private static Person[] competitors;
    private static int competitorsNumber;

    public static void main(String[] args) {
        competitors = new Person[1];
        competitorsNumber = 0;
        Scanner sc = new Scanner(System.in);
        int option = 0;

        do {
            printOptions();
            try {
                option = sc.nextInt();
                sc.nextLine();
            } catch(InputMismatchException exc) {
                sc.nextLine();
                System.out.println("--------------------");
                System.out.println("Dane w nieprawidłowym formacie ");
                continue;
            }

            switch (option) {
            case ADD_COMPETITOR:
                try {
                    addCompetitor(sc);
                } catch(InputMismatchException e) {
                    sc.nextLine();
                    System.out.println("--------------------");
                    System.out.println("Błąd odczytu danych");
                } catch (ArrayIndexOutOfBoundsException e) {
                    System.out.println("--------------------");
                    System.out.println(e.getMessage());
                }
                break;
            case PRINT_ALL:
                printCompetitors();
                break;
            case EXIT:
                break;
            }
        } while (option != EXIT);

        sc.close();

    }

    private static void printCompetitors() {
        System.out.println("--------------------");
```

```
62          System.out.println("Lista uczestników:");
63          for(int i=0; i < competitorsNumber; i++) {
64              System.out.println(competitors[i].toString());
65          }
66      }
67
68      private static void addCompetitor(Scanner sc) throws InputMismatchException, ArrayIndexOutOfBound
69          if(competitorsNumber < competitors.length) {
70              Person person = new Person();
71              System.out.println("--------------------");
72              System.out.println("Dodawanie nowego uczestnika: ");
73              System.out.println("Imię: ");
74              person.setFirstName(sc.nextLine());
75              System.out.println("Nazwisko");
76              person.setLastName(sc.nextLine());
77              System.out.println("PESEL:");
78              person.setPesel(sc.nextLine());
79              System.out.println("Wiek:");
80              person.setAge(sc.nextInt());
81              sc.nextLine();
82
83              competitors[competitorsNumber] = person;
84              competitorsNumber++;
85          } else {
86              //jeżeli tablica jest pełna, tworzymy wyjątek, który o tym informuje
87              throw new ArrayIndexOutOfBoundsException("Osiągnięto maksymalną liczbę uczestników");
88          }
89      }
90
91      private static void printOptions() {
92          System.out.println("--------------------");
93          System.out.println("Dostępne opcje: ");
94          System.out.println(ADD_COMPETITOR + " - Dodaj uczestnika");
95          System.out.println(PRINT_ALL + " - Wyświetl uczestników");
96          System.out.println(EXIT + " - Wyjście z programu");
97          System.out.println("Wybierz opcję: ");
98      }
99  }
```

W naszym programie sensownym miejscem, w którym możemy stworzyć i rzucić wyjątek jest metoda *addCompetitor()*. Jeżeli tablica, którą utworzyliśmy będzie już pełna, wygenerujemy wyjątek, który pojawia się, gdy próbujemy odwoływać się do indeksu tablicy wykraczającego poza zakres, czyli *ArrayIndexOutOfBoundsException*. W kodzie używamy także metody *getInt()* klasy Scanner, jednak tym razem nie obsługujemy tu wyjątku, a jedynie dodajemy o nim informację w sygnaturze metody (lepiej jest go obsłużyć, ale na potrzeby ćwiczenia zróbmy to w ten sposób).

Oba wyjątki, czyli ArrayIndexOutOfBoundsException i InputMismatchException nie muszą być obsługiwane, więc informacja w sygnaturze metody jest bardziej komunikatem dla programisty, czego może się spodziewać. Oba wyjątki obsługujemy w osobnych blokach catch już bezpośrednio w bloku konstrukcji switch umieszczonej w metodzie *main()*. Jak widzisz do jednego bloku try możemy podpiąć kilka bloków catch do obsługi różnych wyjątków - działa to podobnie do instrukcji warunkowej if else.

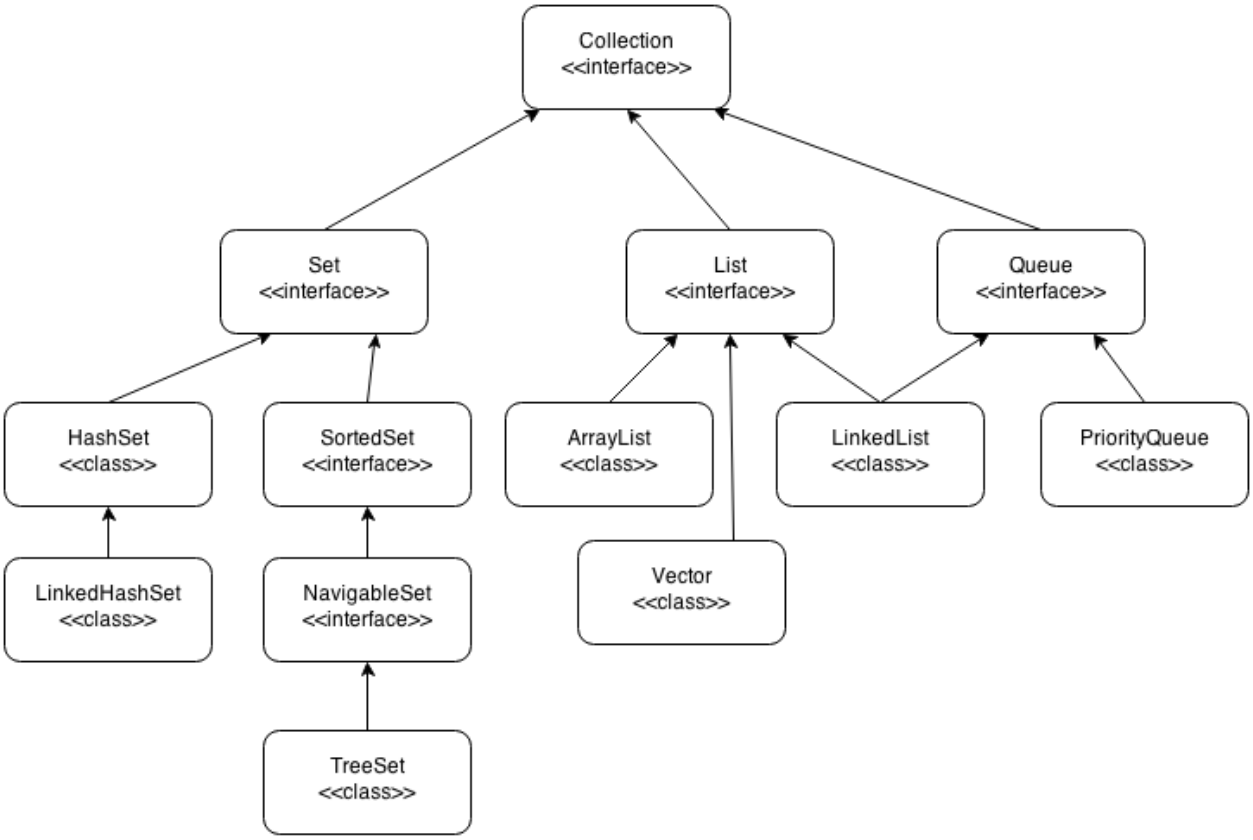
Rozmiar tablicy w powyższym przykładzie zmieniliśmy na 1, abyś mógł przetestować działanie wyjątku. Przy próbie dodania drugiego uczestnika do tablicy, generujemy wyjątek ArrayIndexOutOfBoundsException, z którego następnie już w bloku catch() pobieramy informację przekazaną w konstruktorze za pomocą metody *e.getMessage()*.



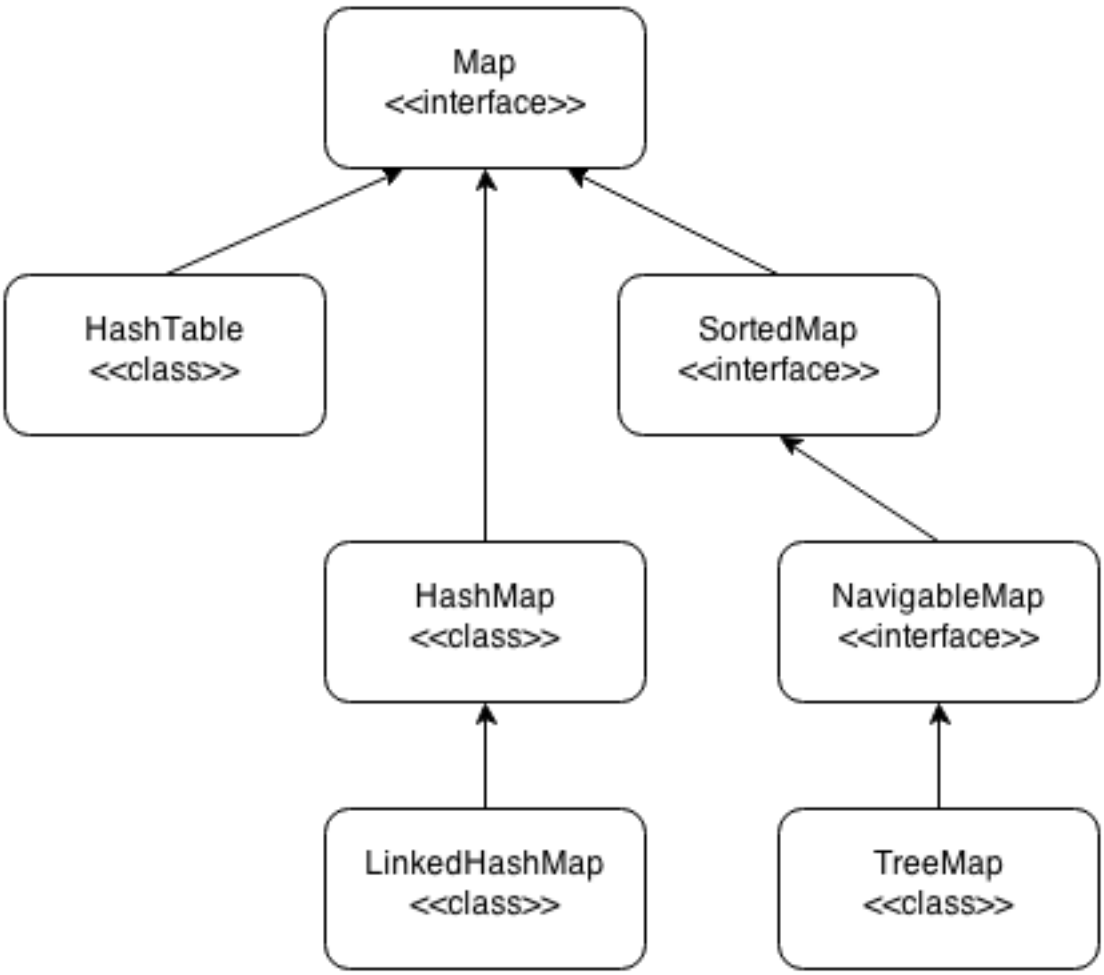
3.5.4 Kolekcje

Mówiąc o kolekcjach w Javie będziemy mieli na myśli Collections framework, czyli specjalny zestaw interfejsów i klas, które są przeznaczone do przechowywania różnych kolekcji obiektów. Hierarchię kolekcji w języku Java przedstawiono na poniższym diagramie, my omówimy najczęściej wykorzystywane listy.

Zbiory, listy i kolejki:


Collection
<<interface>>
Set
<<interface>>
List
<<interface>>
Queue
<<interface>>
HashSet
<<class>>
SortedSet
<<interface>>
ArrayList
<<class>>
LinkedList
<<class>>
PriorityQueue
<<class>>
LinkedHashSet
<<class>>
NavigableSet
<<interface>>
Vector
<<class>>
TreeSet
<<class>>

oraz mapy:



3.5.5 Typy opakowujące

Pierwszą rzeczą, którą musimy wyjaśnić przed przejściem do omówienia list są typy opakowujące typów prostych. Jest to spowodowane tym, że jak wspomnieliśmy na wstępie, kolekcje służą do przechowywania obiektów, a typy proste takie jak int, czy double typami obiektowymi nie są. Typem obiektowym jest natomiast String, w którego przypadku nie napotkamy na większe problemy.

Każdy z typów prostych ma swój odpowiednik obiektowy. Lista klas reprezentujących poszczególne z nich przedstawia się następująco:

- byte - Byte
- short - Short
- int - Integer
- long - Long
- float - Float
- double - Double
- boolean - Boolean
- char - Character

Jak widzisz w większości przypadków zmianie ulega jedynie litera z małej na wielką. W celu zamiany wartości typu prostego, np. liczby 15, na obiekt typu Integer reprezentujący wartość 15 należy wykorzystać statyczną metodę *valueOf()*, którą posiada każdy z wyżej wymienionych typów. Metoda ta jest dostępna w kilku przeciążonych wersjach, więc jako jej argument możemy podać zarówno liczbę w formie typu prostego lub jako String. Innym sposobem jest po prostu skorzystanie z konstruktora danej klasy - oba podejścia działają w praktyce tak samo.

plik Wrappers.java

```
public class Wrappers {
    public static void main(String[] args) {
        //konstruktor na podstawie liczby
        Integer num1 = new Integer(15);
        //konstruktor na podstawie Stringa
        Integer num2 = new Integer("15");

        //metody valueOf()
        Integer num3 = Integer.valueOf(15);
        Integer num4 = Integer.valueOf("15");

        //Porównanie obiektów nie działa tak jak porównywanie typów prostych!
        //do porównywanie obiektów zawsze wykorzystuj metodę equals()
        System.out.println("num1 == num2 = " + (num1==num2));
        System.out.println("num1.equals(num2) = " + (num1.equals(num2)));
    }
}
```

Ponieważ korzystając z kolekcji byłoby straszną męczarnią konwertowanie typów z prostych na obiektowe i z obiektowych na proste, to w Javie 1.5 wprowadzono mechanizm autoboxingu i unboxingu, czyli automatycznej konwersji pomiędzy typami. Można więc bez problemu przypisać do referencji typu Integer wartość 5, a wirtualna maszyna automatycznie zamieni ją na obiekt.

plik Autoboxing.java

```
public class Autoboxing {
    public static void main(String[] args) {
        //autoboxing
        Double num1 = 5.54;
        Integer num2 = 200;

        //unboxing
        double num3 = num1;
        int num4 = num2;

        System.out.println("num1.equals(num2) = " + (num1 == num3)); //true
    }
}
```

Typ String jest typem obiektowym (dlatego też pisany jest z wielkiej litery), więc nie wymaga dodatkowych zabiegów.

3.5.6 Listy

Listy to najprostsze struktury danych. Ich głównym zadaniem jest przechowywanie obiektów w uporządkowanej, indeksowanej formie - czyli podobnie jak w przypadku tablic. Istnieją dwa główne typy list:

- lista tablicowa (**ArrayList**) - jej wewnętrzna struktura opiera się dokładnie na tablicy
- lista wiązana (**LinkedList**) - obiekty w niej stanowią skończony ciąg połączonych ze sobą węzłów

Na poziomie tego szkolenia nie będziemy zgłębiali różnić pomiędzy tymi strukturami i skupimy się na pierwszej z nich, jednak warto wiedzieć, że z powodu odmiennej reprezentacji wewnętrznej wydajnosć operacji takich jak usuwanie, czy wyszukiwanie w obu z nich może się znacząco różnić w przypadku dużych grup danych.

Listę tworzymy w następujący sposób:

```
ArrayList<Typ_Danych> nazwaListy = new ArrayList<>();
//np.
ArrayList<String> names = new ArrayList<>();
```

Ponieważ typy kolekcyjne stanowią jednak pewną hierarchię dziedziczenia, warto również korzystać z zalet polimorfizmu i stosować ogólniejszy typ referencji, do którego można przypisać zarówno LinkedListę jak i ArrayListę.

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

Mamy jednak wtedy dostęp jedynie do metod z typu referencji, czyli interfejsu List (chyba, że zastosujemy rzutowanie na typ ArrayList). Pomiędzy ostrymi nawiasami określamy typ danych jaki będzie przechowywała dana lista. Powiemy dzięki temu, że kolekcja jest **typem generycznym**.

Na listach możemy wykonywać podstawowe operacje takie jak:

- dodawanie - metoda **add(Object obiekt)**
- usuwanie - metoda **remove(int index)** jeśli chcemy usunąć element o wskazanym indeksie lub **remove(Object o)** jeżeli chcemy usunąć obiekt, dla którego porównanie za pomocą metody equals() z przekazanym parametrem zwróci true
- pobranie elementu z listy - metoda **get()** - analogicznie jak w przypadku tablic listy indeksowane są od 0
- sprawdzenie rozmiaru listy - metoda **size()**
- sprawdzenie, czy dany obiekt znajduje się w liście - metoda **contains(Object obiekt)**

plik Collect.java

```
1  package pl.org.ceo.main;
2
3  import java.util.ArrayList;
4  import java.util.List;
5
6  public class Collect {
7      public static void main(String[] args) {
8          List<String> names = new ArrayList<>();
9
10         names.add("Jan");
11         names.add("Wojtek");
12         names.add("Kasia");
13
14         System.out.println("Czy Jan jest na liście: " + names.contains("Jan"));
15         System.out.println("Ile jest osób na liście: " + names.size());
16         System.out.println("A po usunięciu Wojtka ");
17         names.remove("Wojtek");
18         System.out.println("rozmiar to " + names.size());
19     }
20 }
```

Ćwiczenie ***(15 minut)*** Przerób program z wcześniejszej części lekcji (zapisy na konkurs) w taki sposób, aby uczestnicy byli dopisywani do listy, a nie tablicy.

plik Competition.java

```java
package pl.org.ceo.app;

import java.util.ArrayList;
import java.util.InputMismatchException;
import java.util.List;
import java.util.Scanner;

import pl.org.ceo.data.Person;

public class Competition {

    public static final int ADD_COMPETITOR = 0;
    public static final int PRINT_ALL = 1;
    public static final int EXIT = 2;

    // zastępujemy tablicę listą
    private static List<Person> competitors;

    public static void main(String[] args) {
        // inicjalizujemy listę
        competitors = new ArrayList<>();
        Scanner sc = new Scanner(System.in);
        int option = 0;

        do {
            printOptions();
            try {
                option = sc.nextInt();
                sc.nextLine();
            } catch (InputMismatchException exc) {
                sc.nextLine();
                System.out.println("--------------------");
                System.out.println("Dane w nieprawidłowym formacie ");
                continue;
            }

            switch (option) {
            case ADD_COMPETITOR:
                try {
                    addCompetitor(sc);
                } catch (InputMismatchException e) {
                    sc.nextLine();
                    System.out.println("--------------------");
                    System.out.println("Błąd odczytu danych");
                }
                break;
            case PRINT_ALL:
                printCompetitors();
                break;
            case EXIT:
                break;
            }
        } while (option != EXIT);

        sc.close();

    }

```

```
59      private static void printCompetitors() {
60          System.out.println("---------------------");
61          System.out.println("Lista uczestników:");
62          //W listach możemy wykorzystywać pętle for-each jak przy tablicach
63          for (Person p: competitors) {
64              System.out.println(p);
65          }
66      }
67
68      private static void addCompetitor(Scanner sc) throws InputMismatchException {
69          // sprawdzanie ilośći elementów jest zbędne, ponieważ zajmuje się tym
70          // lista
71          Person person = new Person();
72          System.out.println("---------------------");
73          System.out.println("Dodawanie nowego uczestnika: ");
74          System.out.println("Imię: ");
75          person.setFirstName(sc.nextLine());
76          System.out.println("Nazwisko");
77          person.setLastName(sc.nextLine());
78          System.out.println("PESEL:");
79          person.setPesel(sc.nextLine());
80          System.out.println("Wiek:");
81          person.setAge(sc.nextInt());
82          sc.nextLine();
83
84          competitors.add(person);
85      }
86
87      private static void printOptions() {
88          System.out.println("---------------------");
89          System.out.println("Dostępne opcje: ");
90          System.out.println(ADD_COMPETITOR + " - Dodaj uczestnika");
91          System.out.println(PRINT_ALL + " - Wyświetl uczestników");
92          System.out.println(EXIT + " - Wyjście z programu");
93          System.out.println("Wybierz opcję: ");
94      }
95  }
```

3.5.7 Porównywanie obiektów

Ważnym zagadnieniem, które może sprawiać pewne problemy jest porównywanie obiektów w języku Java. Najważniejszym zagadnieniem jest tutaj zrozumienie tego, że obiekt i referencja wskazująca na ten obiekt (uproszczając - zmienna) to dwie różne rzeczy, które porównujemy w różny sposób.

Obiekty zawsze powinniśmy porównywać za pomocą metody **equals()**. Jest to specjalna metoda, która jest dziedziczona przez każdą klasę Javy z klasy Object, która jest nadrzędną klasą dla wszystkich innych. Warto przesłonić metodę *equals()* i zdefiniować w niej porównanie wszystkich pól klasy. Metodę *equals()* możesz wygenerować w eclipse korzystając z opcji Source -> Generate hashCode() and equals(). Możemy następnie wybrać, które pola muszą być równe, aby uznać, że dwa obiekty danego typu będą równe. Np. dla naszej wcześniejszej klasy Car wyglądałoby to następująco:

plik Car.java

```
1  package pl.org.ceo.cardiagnoser.data;
2
3  public class Car {
4
```

```
5       public String carBrand; // marka samochodu
6       public String model;
7       public int year; //rok produkcji
8       public int horsePower; // ilość koni mechanicznych
9
10      public Car(String carBrand, String model) {
11          this.carBrand = carBrand;
12          this.model = model;
13      }
14
15      public Car(String carBrand, String model, int year, int horsePower) {
16          this(carBrand, model);
17          this.year = year;
18          this.horsePower = horsePower;
19      }
20
21      public void upgreade(int hp) {
22          horsePower = horsePower + hp;
23      }
24
25      @Override
26      public int hashCode() {
27          final int prime = 31;
28          int result = 1;
29          result = prime * result
30                  + ((carBrand == null) ? 0 : carBrand.hashCode());
31          result = prime * result + horsePower;
32          result = prime * result + ((model == null) ? 0 : model.hashCode());
33          result = prime * result + year;
34          return result;
35      }
36
37      @Override
38      public boolean equals(Object obj) {
39          if (this == obj)
40              return true;
41          if (obj == null)
42              return false;
43          if (getClass() != obj.getClass())
44              return false;
45          Car other = (Car) obj;
46          if (carBrand == null) {
47              if (other.carBrand != null)
48                  return false;
49          } else if (!carBrand.equals(other.carBrand))
50              return false;
51          if (horsePower != other.horsePower)
52              return false;
53          if (model == null) {
54              if (other.model != null)
55                  return false;
56          } else if (!model.equals(other.model))
57              return false;
58          if (year != other.year)
59              return false;
60          return true;
61      }
62
```

```
63      public String getInfo() {
64          return carBrand + " " + model + "; " + year + "; " + horsePower + "HP";
65      }
66  }
```

Metoda hashCode() generowane wspólnie z equals() jest tzw. funkcją mieszającą i zwraca unikalny identyfikator obiektu obliczony na podstawie jego pól. Nie będziemy się w tym miejscu zagłębiali w jej zastosowanie, jednak należy pamiętać, że jeżeli metoda equals (porównanie dwóch obiektów) zwraca true, to wynik metody hashCode() dla tych dwóch obiektów także powinien być równy.

Teraz jeszcze w celu uzupełnienia wróćmy do kwestii porównania referencji vs porównania faktycznych obiektów. Jeżeli zapiszemy np.:

```
Car car1 = new Car("VW", "Polo");
Car car2 = car1;
```

to w takiej sytuacji widzimy dwie referencje *car1* i *car2* wskazujące na dokładnie ten sam obiekt. Jeżeli zapiszemy teraz:

```
car2.year = 2000;
```

to odwołując się do pola *year* poprzez referencję *car1* również mamy dostęp do podanej wartości 2000. Zupełnie inna sytuacja będzie w sytuacji, gdy utworzymy osobne obiekty i przypiszemy je do dwóch różnych referencji:

```
Car car1 = new Car("VW", "Polo");
Car car2 = new Car("VW", "Polo");
```

Teraz istnieją dwa różne obiekty, ale o takiej samej strukturze. Jeżeli porównamy referencje operatorem == otrzymamy wartość false, ale jeśli sprawdzimy równość obiektów metodą equals() otrzymamy true.

```
Car car1 = new Car("VW", "Polo");
Car car2 = new Car("VW", "Polo");
boolean eq = car1.equals(car2); //true, bo obiekty są równe (równość "zawartości obiektów")
boolean eqRef = car1==car2; //false, bo car1 i car2 wskazują na różne obiekty
```

3.6 Graficzny interfejs użytkownika

Z powodu ilości nowych elementów oraz skomplikowania tej lekcji uczniowie powinni rozwijać kod i aplikację wraz z prowadzącym.

W tej lekcji dowiesz się:

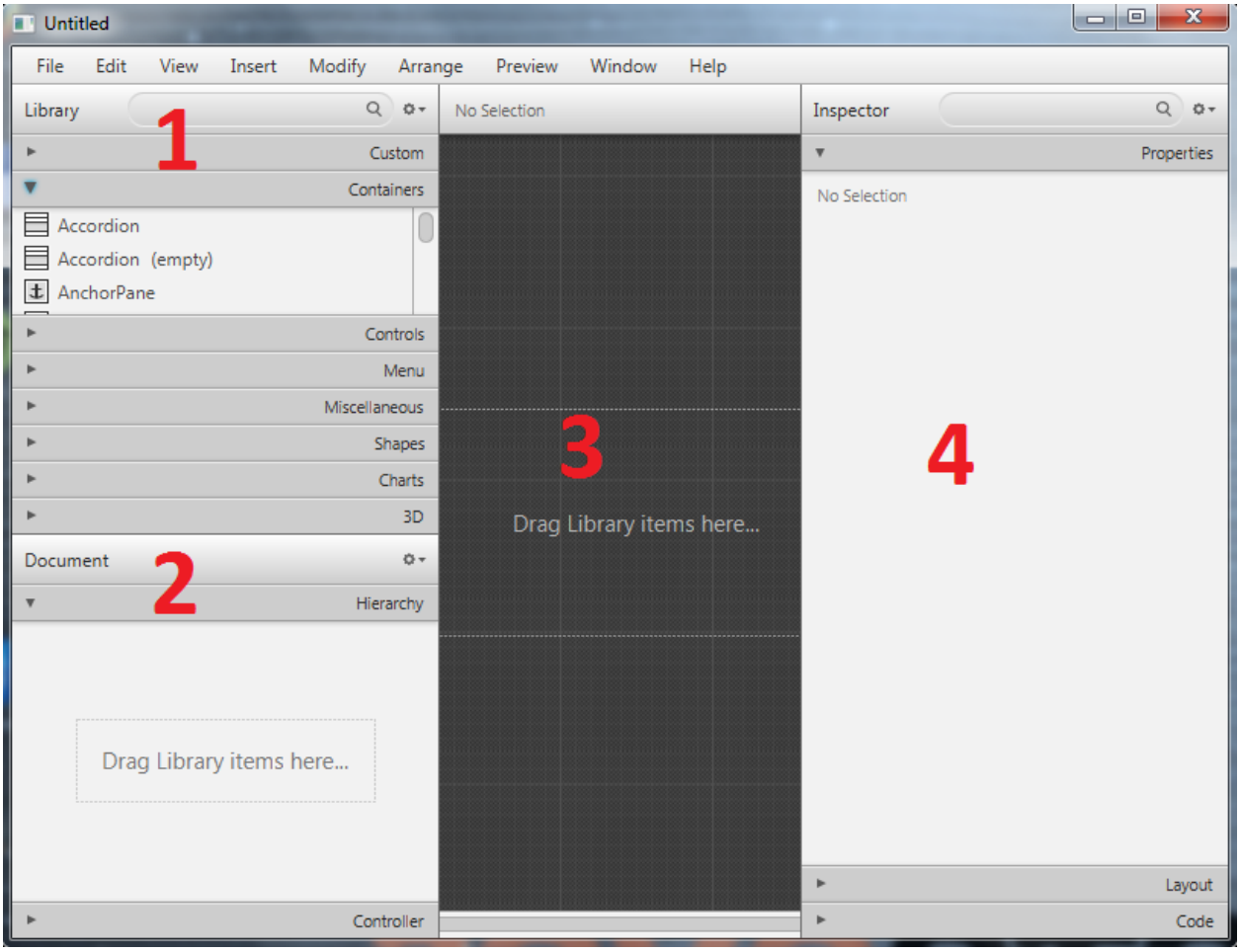
- Czym jest JavaFX
- Jak stworzyć graficzny interfejs użytkownika wykorzystując Scene Buildera
- Co oznacza wzorzec MVC
- Jak wygląda język FXML
- Jak stworzyć prosty projekt w JavieFX

3.6.1 JavaFX

JavaFX jest technologią, która pozwala na tworzenie bogatych graficznie interfejsów użytkownika do aplikacji napisanych w Javie. W początkowym zamyśle miała to być technologia przeznaczona głównie do tworzenia tzw. Rich Internet Application, które miałyby rywalizować z Adobe Flash, czy Microsoft Silverlight. Ostatecznie oprócz wspomnianej funkcji stała się także rekomendowaną biblioteką do tworzenia graficznego interfejsu użytkownika aplikacji desktopowych napisanych w języku Java. Trwają także prace nad tym, aby aplikacje napisane w JavieFX można portować na platformy Android oraz iOS (zasada write once run everywhere).

Głównym narzędziem, który posłuży nam do budowania GUI (graphical user interface) będzie Scene Builder - oficjalne narzędzie rozwijane przez Oracle, które pozwala tworzyć interfejs użytkownika za pomocą przyjaznego edytora WYSIWYG.

3.6.2 Scene Builder 2.0 - pierwszy rzut oka



W Scene Builderze możemy wyróżnić kilka głównych obszarów roboczych, z których będziemy korzystali. Zgodnie z oznaczeniami na powyższym zrzucie ekranu są to:

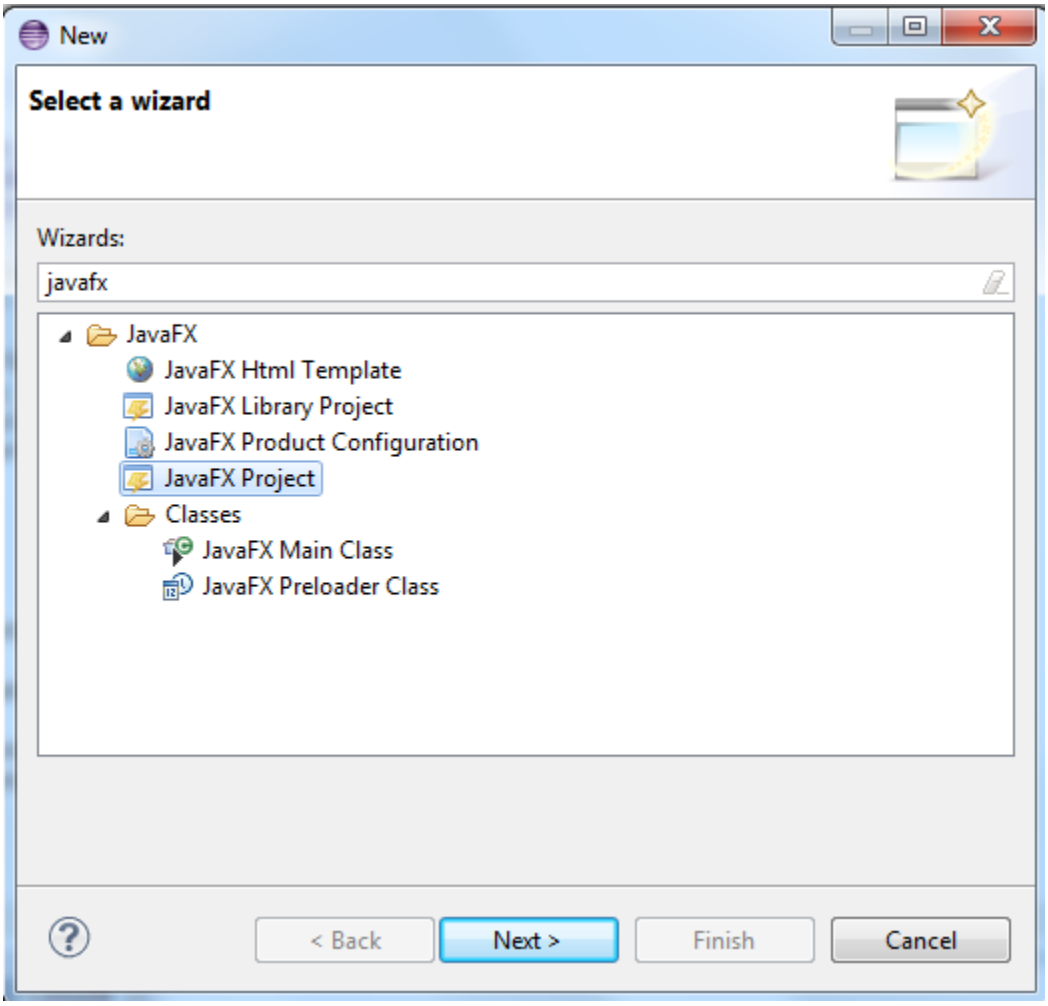
1. **Library** - zestaw możliwych do wykorzystania kontrolek i layoutów
2. **Document** - podgląd pliku w postaci drzewa węzłów
3. **Podgląd** - główny obszar roboczy
4. **Inspector** - ustawianie właściwości poszczególnych kontrolek

Ćwiczenie *(10 minut)*

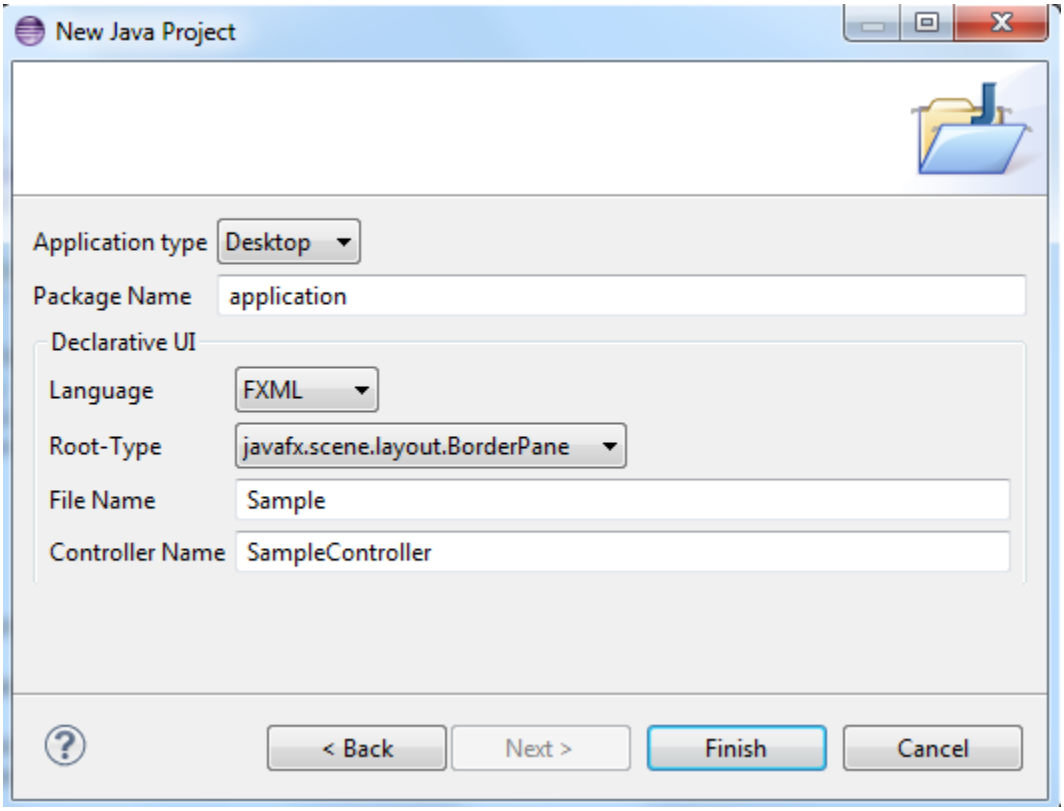
> Zapoznaj się przez chwilę z narzędziem Scene Builder. Spróbuj przeciągnąć różne elementy z sekcji Library do obszaru roboczego lub bezpośrednio do sekcji Document. Kliknij na wybrane z przeciągniętych elementów i spróbuj pozmieniać ustawienia w sekcji Inspector po prawej stronie.

3.6.3 Pierwszy projekt JavaFX

W celu utworzenia swojego pierwszego projektu w JavieFX przejdź do eclipse i utwórz nowy projekt JavaFX (PPM -> New -> Other -> JavaFX Project).



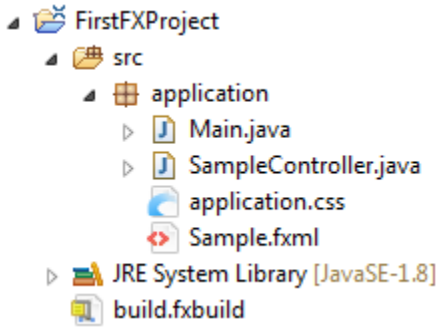
Następnie wpisz dowolną nazwę projektu i klikając przycisk Next przejdź do ostatniej zakładki kreatora projektu, gdzie wybierz Language jako **FXML**.



Wybierz Finish, po czym projekt zostanie utworzony.

3.6.4 Omówienie domyślnego projektu

Eclipse domyślnie wygeneruje dla nas prosty szablon projektu JavaFX, który już na pierwszy rzut oka różni się od tego co znaliśmy do tej pory.



- Main.java to główna klasa, od której rozpoczyna się działanie aplikacji.
- Sample.fxml zawiera definicję tego z jakich layoutów oraz kontrolek składa się widok naszej aplikacji - w skrócie jak wygląda
- application.css to definicja stylów CSS dla naszej aplikacji. Z pewnością słyszałeś już o nich w przypadku stron internetowych
- SampleController.java to dodatkowa klasa, która występuje w architekturze MVC (Model View Controller)

Model View Controller jest najpopularniejszym wzorcem architektonicznym wykorzystywanym w tworzeniu aplikacji. Jego głównym zamysłem jest to, żeby oddzielić od siebie definicję tego jak aplikacja wygląda od tego co robi.

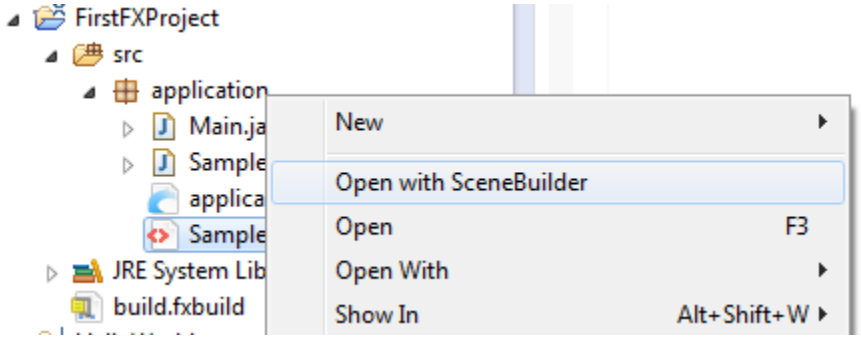
W komunikacji pomiędzy tymi dwoma elementami występuje kontroler, który odpowiednio przekazuje informacje od użytkownika (GUI aplikacji) do modelu danych i w drugą stronę.

Zaletą takiego podejścia jest to, że aplikacje można rozwijać równolegle i lepiej dzielić obowiązki w zespole programistów. Jeżeli ktoś jest silniejszy tworzeniu tzw. backendu może zająć się programowaniem logiki biznesowej, natomiast osoby o lepszym zmyśle estetycznym mogą się poświęcić stworzeniu frontendu aplikacji.

3.6.5 Definiowanie widoku

Widok naszej aplikacji będziemy definiowali w języku XML i formie plików o rozszerzniu .fxml. Na szczęście nie musimy wszystkiego pisać ręcznie, bo kod zostanie wygenerowany na podstawie tego co zrobimy w Scene Builderze.

Żeby przejść do edycji pliku fxml kliknij na niego prawym przyciskiem myszy w eclipse i wybierz opcję Open with SceneBuilder.



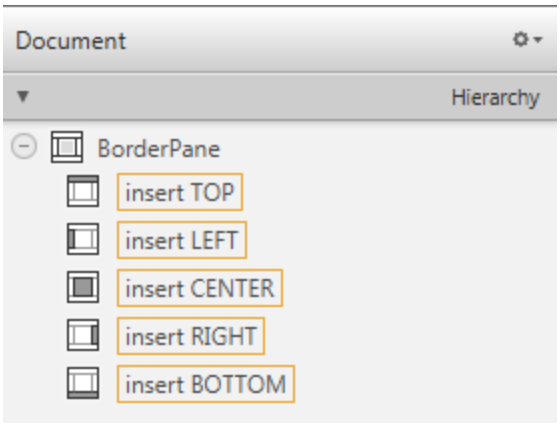
Jeżeli nic nie zmieniałeś w ustawieniach projektu, to plik Sample.fxml powinien mieć następującą postać:

plik Sample.fxml

```
<?xml version="1.0" encoding="UTF-8"?>

<?import javafx.scene.layout.BorderPane?>

<BorderPane xmlns:fx="http://javafx.com/fxml" fx:controller="application.SampleController">
    <!-- TODO Add Nodes -->
</BorderPane>
```

BorderPane to główny węzeł naszego widoku. Jest to zwyczajna klasa, którą możesz znaleźć w dokumentacji Javy i której obiekt możesz utworzyć także bezpośrednio w kodzie napisanym w Javie. Jak widzisz klasa ta jest najpierw zaimportowana z pakietu *javafx.scene.layout*. Warto zwrócić uwagę także na atrybut xmlns (skrót od XML Namespace), czyli przestrzeni nazw, z której korzystamy - jeżeli jej nie zdefiniujemy, nasza aplikacja się nie uruchomi, ponieważ kod zostanie uznany za niezgodny ze standardem. atrybut *fx:controller="application.SampleController"* wskazuje klasę kontrolera powiązaną z tym konkretnym plikiem widoku - w tym przypadku jest to nasza klasa *SampleController*.

W Scene builderze zauważysz, że kod XML ma odzwierciedlenie w sekcji Document Hierarchy, gdzie znajduje się struktura naszego widoku w postaci drzewa węzłów.



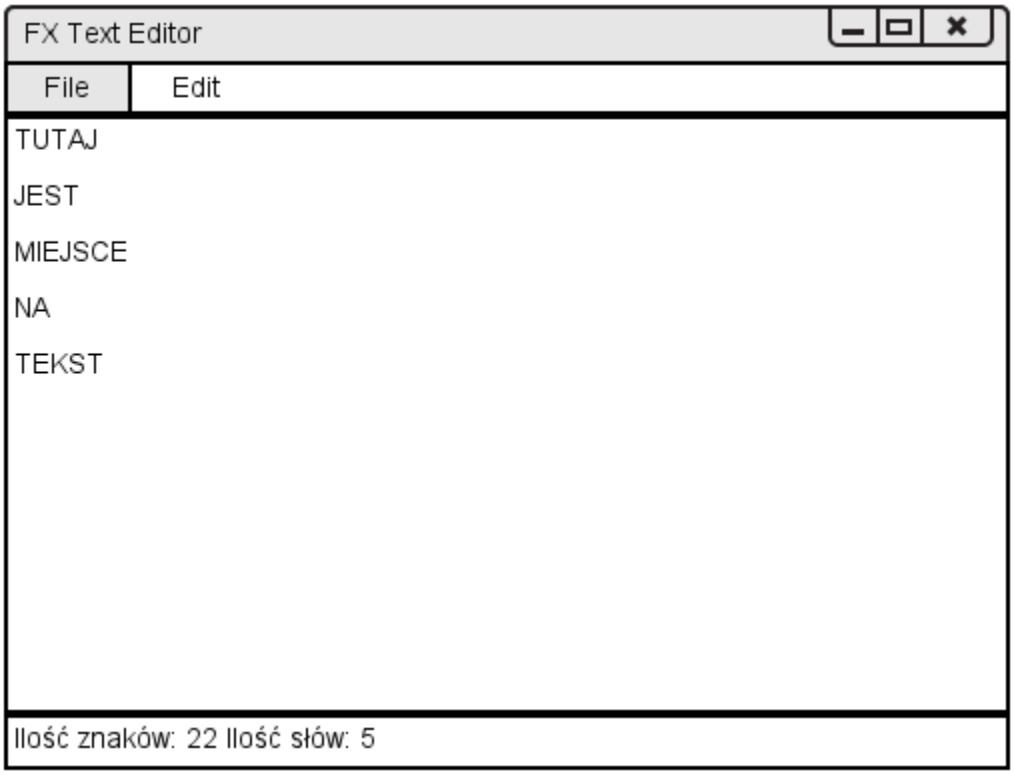
Edytor WYSIWYG

Scene Builder to wygodny wdytor w stylu "przeciągnij i upuść". Możesz w nim przeciągnąć kilka kontrolek na nasz główny layout, czyli BorderPane (bezpośrednio do sekcji Document Hierarchy) lub na obszar roboczy. Wszelkie zmiany będą aktualizowane automatycznie.

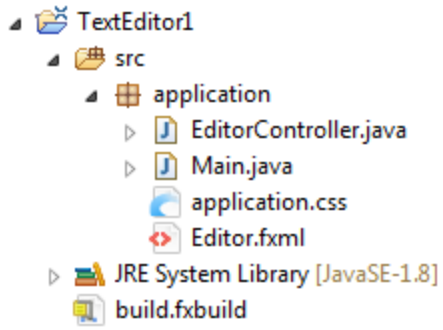
Spróbujmy teraz zaprojektować bardzo prosty edytor tekstowy, w którym muszą znaleźć się takie elementy jak:

- pasek menu kontekstowego
- główne pole tekstowe
- pasek podliczający ilość wprowadzonych znaków oraz ilość słów w tekście

Makieta aplikacji:



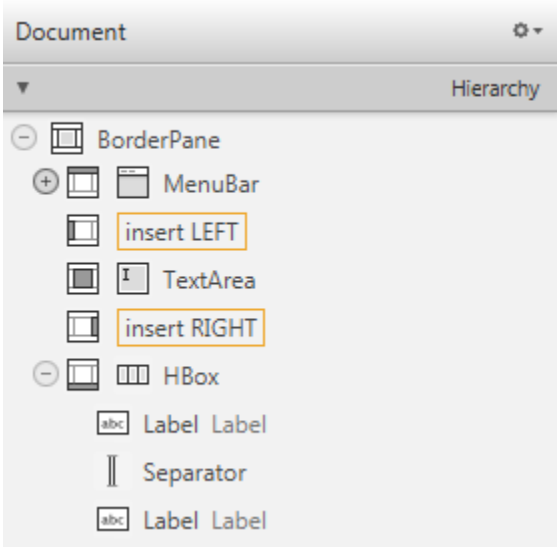
Struktura projektu eclipse:



Otwórz plik fxml w Scene Builderze i dobierz odpowiednie kontrolki, które umieścisz na głównym layoucie aplikacji (BorderPane). Kontrolki, które należy wykorzystać to:

- MenuBar - pasek nawigacyjny, na którym można umieszczać elementy typu MenuItem
- TextArea - pole tekstowe, w którym można wprowadzać wiele wierzy tekstu
- Label - etykiety tekstowe
- Separator - pozwala oddzielić kontrolki od siebie

Hierarchia dokumentu FXML w Scene Builderze:



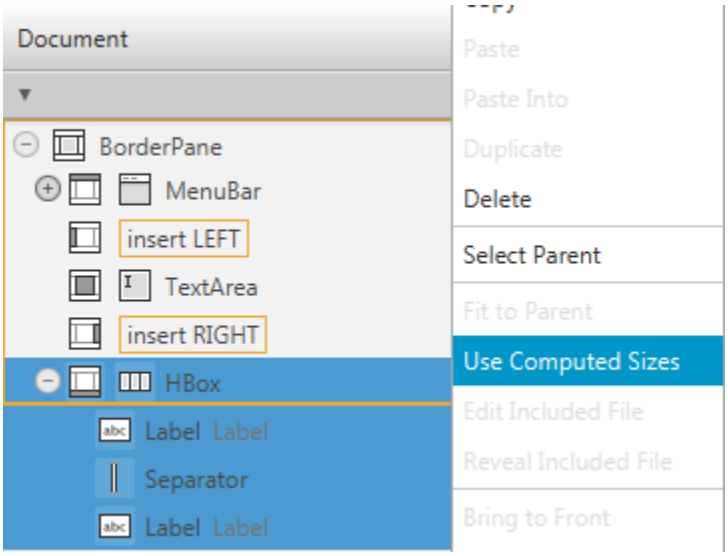
Jak widzisz etykiety w dolnej części aplikacji opakowane zostały w dodatkowy layout typu HBox.

Informacja: Layouty przy tworzeniu graficznego interfejsu użytkownika służą do ustalenia pewnego porządku dodawanych do nich kontrolek. BorderPane pozwala ustawić elementy na krawędziach (góra, dół, lewo, prawo lub środek) natomiast HBox ustawia kontrolki w jednym wierszu jeden obok drugiego. Istnieje dużo więcej layoutów, które możesz przejrzeć w sekcji containers Scene Buildera. Ikonki umieszczone przy poszczególnych z nich oraz nazwy bardzo dobrze opisują to w jaki sposób możemy je wykorzystać. Layouty mogą być także zagnieżdżane jeden w drugim tak jak w naszym przykładzie, gdzie HBox jest węzłem w layoucie typu BorderPane.

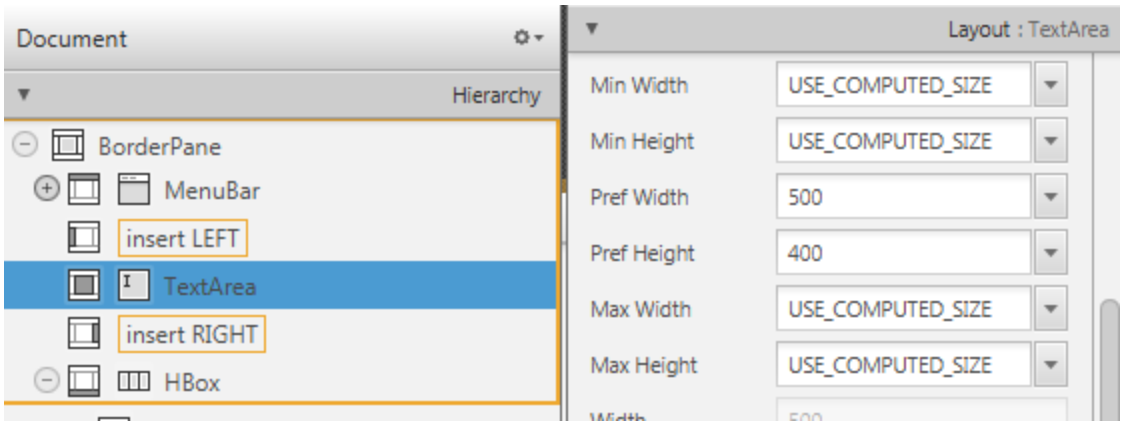
Problem jaki pojawia się w tym momencie to domyślne parametry kontrolek, które sprawiają, że aplikacja nie wygląda najlepiej:

Możemy to na szczęście dosyć łatwo zmodyfikować w sekcji Layout scene buildera (po prawej stronie po kliknięciu na dowolny element), a także wykorzystując opcję dostosowania do domyślnego rozmiaru węzłów potomnych.

Po wcześniejszym zaznaczeniu HBoxa oraz elementów do niego dodanych wybierz opcję Use Computed Sizes, co pozwoli na pozbycie się zbędnej przestrzeni przy naszych etykietach:



Teraz kliknij na obiekt TextArea i w ustawieniach Layout po prawej stronie ustaw jego wysokość i szerokość (Pref Width i Pref Height):



W tym momencie po zapisaniu naszego pliku fxml jego kod powinien wyglądać następująco:

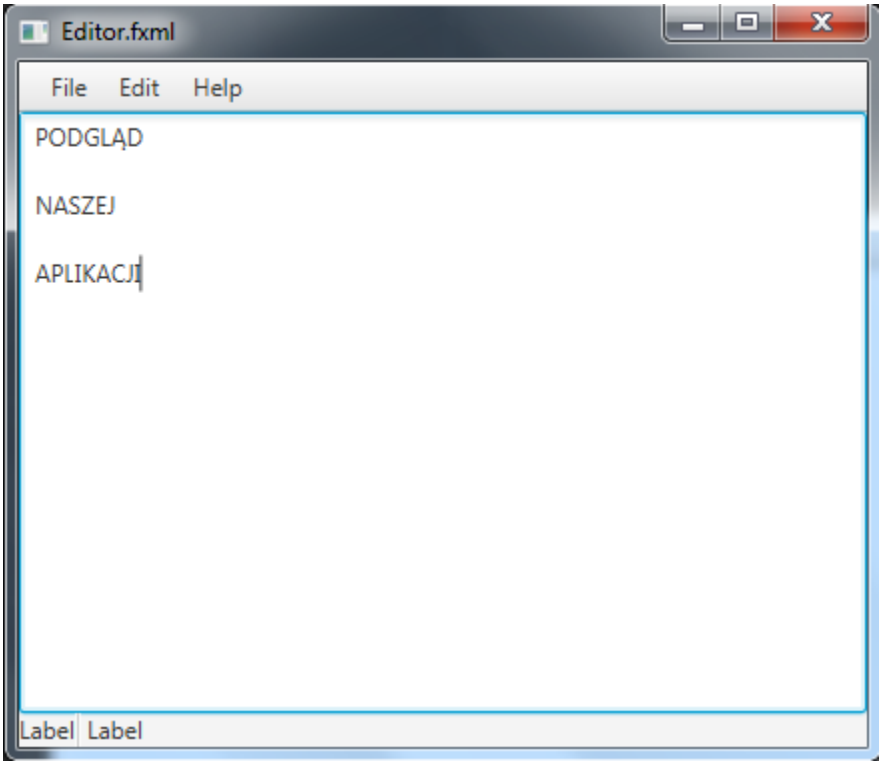
plik Editor.fxml

```
<?xml version="1.0" encoding="UTF-8"?>

<?import javafx.scene.control.*?>
<?import java.lang.*?>
<?import javafx.scene.layout.*?>
<?import javafx.scene.layout.BorderPane?>

<BorderPane xmlns:fx="http://javafx.com/fxml/1" xmlns="http://javafx.com/javafx/8"
    fx:controller="application.SampleController">
    <top>
        <MenuBar BorderPane.alignment="CENTER">
            <menus>
                <Menu mnemonicParsing="false" text="File">
                    <items>
                        <MenuItem mnemonicParsing="false" text="Close" />
                    </items>
                </Menu>
                <Menu mnemonicParsing="false" text="Edit">
                    <items>
                        <MenuItem mnemonicParsing="false" text="Delete" />
                    </items>
                </Menu>
                <Menu mnemonicParsing="false" text="Help">
                    <items>
                        <MenuItem mnemonicParsing="false" text="About" />
                    </items>
                </Menu>
            </menus>
        </MenuBar>
    </top>
    <center>
        <TextArea prefHeight="400.0" prefWidth="500.0"
            BorderPane.alignment="CENTER" />
    </center>
    <bottom>
        <HBox BorderPane.alignment="CENTER">
            <children>
                <Label text="Label" />
                <Separator orientation="VERTICAL" />
                <Label text="Label" />
```

```
            </children>
        </HBox>
    </bottom>
</BorderPane>
```

Jak widać każdy element, który dodaliśmy w Scene Builderze ma tutaj swoje odzwierciedlenie w postaci węzła XML. Widoczne są także ustawienia poszczególnych elementów, np. wysokość i szerokość TextArea postaci *TextArea prefHeight="400.0" prefWidth="500.0"*.

W tym momencie warto zobaczyć jak nasza aplikacja będzie wyglądała po uruchomieniu. Co ciekawe jeżeli chcemy podejrzeć tylko wygląd bez funkcjonalności, możemy to zrobić bezpośrednio z poziomu Scene Buildera korzystając z opcji Preview -> Show Preview in Window (Ctrl + P).



Jak widzisz widok aplikacji można więc definiować nawet nie znając języka Java.

3.6.6 Architektura aplikacji

Czas wrócić jednak do tego, aby nasza aplikacja dała się uruchomić z poziomu eclipse jako aplikacja Javy. Zacznijmy od poprawki w nazwie pliku fxml wczytywanym w klasie Main:

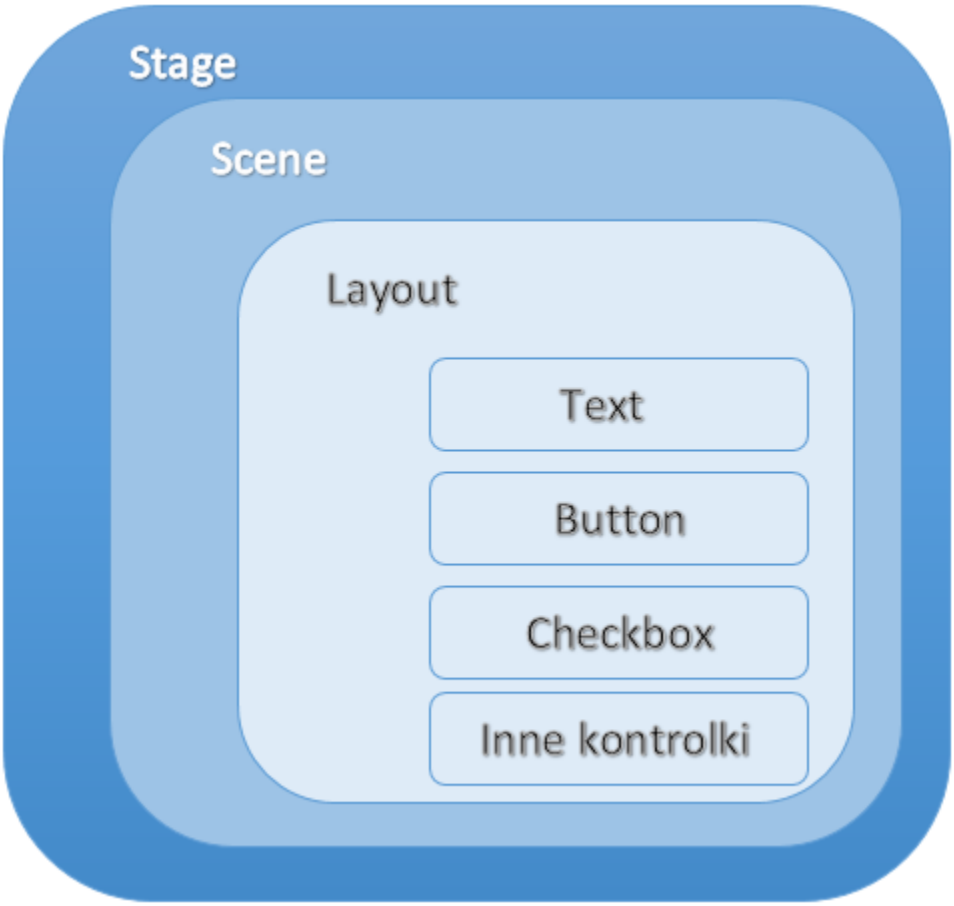
```
package application;

import javafx.application.Application;
import javafx.fxml.FXMLLoader;
import javafx.scene.Scene;
import javafx.scene.layout.BorderPane;
import javafx.stage.Stage;

public class Main extends Application {
    @Override
    public void start(Stage primaryStage) {
```

```
12          try {
13              BorderPane root = (BorderPane) FXMLLoader.load(getClass()
14                      .getResource("Editor.fxml"));
15              Scene scene = new Scene(root, 400, 400);
16              scene.getStylesheets().add(
17                      getClass().getResource("application.css").toExternalForm());
18              primaryStage.setScene(scene);
19              primaryStage.show();
20          } catch (Exception e) {
21              e.printStackTrace();
22          }
23      }
24
25      public static void main(String[] args) {
26          launch(args);
27      }
28  }
```

Omówmy ten kod linijka po linijce:

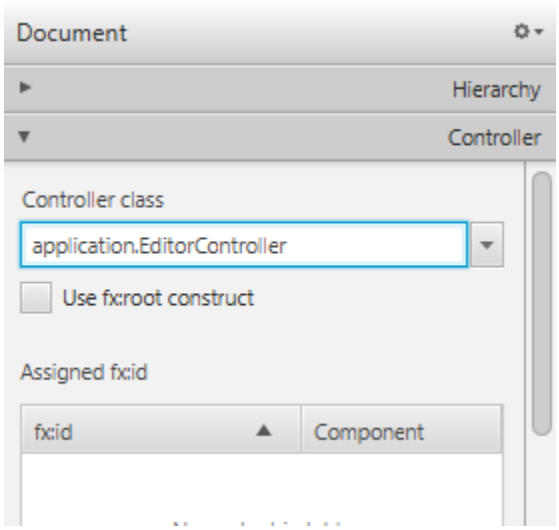
1. Wiersze 1-8 to deklaracja pakietu oraz import bibliotek, które później wykorzystujemy
2. W linijce 9 widzimy, że nasza klasa Main rozszerza klasę Application. Oznacza to, że jest to główna klasa aplikacji napisanej w JavieFX i to od niej rozpocznie się działanie naszej aplikacji.
3. Klasa Application posiada jedną abstrakcyjną metodę *start()*, którą musimy przesłonić (Override). Jako jej argument przekazany zostanie obiekt Stage, który zostanie utworzony przez wirtualną maszynę. **Stage** to okno naszej aplikacji.
4. W wierszach 13-14 wczytujemy nasz widok za pomocą specjalnej klasy FXMLLoader i metody *load()*. Przetwarza ona plik XML i na podstawie zawartych w nim definicji tworzy obiekty, które będą odzwierciedlone w kodzie Javy.
5. W 15 wierszu tworzymy obiekt **Scene**, który dodamy do naszego okna (Stage) w wierszu 18. Scene to klasa reprezentująca główny kontener z widokiem aplikacji, do którego możemy dodawać inne elementy takie jak layouty, czy konkretne kontrolki.
6. W 16 i 17 wierzu wczytujemy style CSS, które aplikujemy do naszej sceny. Ponieważ na tę chwilę plik *application.css* jest pusty, nie będzie to miało wpływu na wygląd naszej aplikacji.
7. W metodzie *main()* wywołujemy metodę *launch()* ta z kolei odpowiada za cykl życia aplikacji JavaFX, czyli m.in. wywołanie metody *start()*.

Ostateczna hierarchia widoku w JavieFX wygląda więc następująco:



Pozostaje nam jeszcze jedna rzecz do poprawy. Przy próbie uruchomienia programu otrzymujemy błąd *Caused by: java.lang.ClassNotFoundException: application.SampleController* - jest on spowodowany tym, że w pliku fxml nie zmieniliśmy klasy kontrolera (fx:controller) po zmianie nazwy pliku z klasą (EditorController).

W Scene Builderze można to zrobić także w sekcji Document rozwijając zakładkę Controller. Nazwę klasy kontrolera należy podać również ze ścieżką uwzględniającą pakiet (tzw. fully qualified name).



Definicja BorderPane w pliku fxml powinna więc wyglądać teraz następująco:

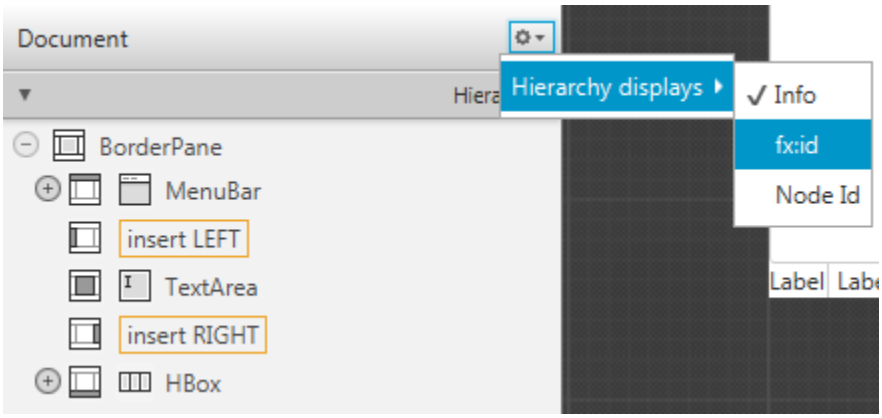
```
<BorderPane xmlns="http://javafx.com/javafx/8" xmlns:fx="http://javafx.com/fxml/1"
    fx:controller="application.EditorController">
```

W tym momencie powinniśmy mieć już możliwość uruchomienia naszej aplikacji z poziomu eclipse a naszym oczom powinien ukazać się widok analogiczny do podglądu, który widzieliśmy wcześniej w Scene Builderze.

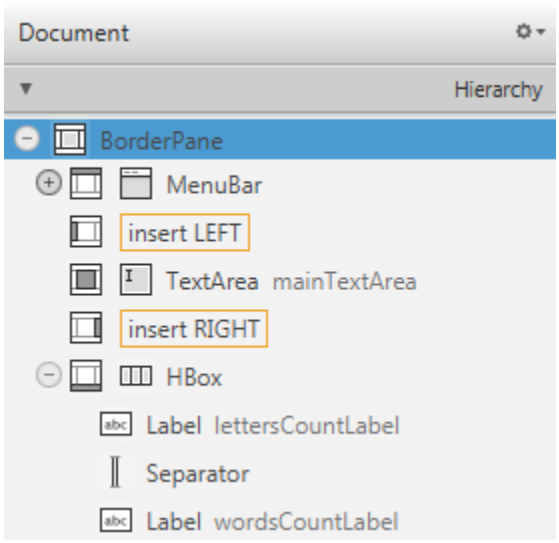
Uwaga: Jeżeli przy próbie uruchomienia aplikacji nadal pojawia się jakiś błąd związany z dokumentem fxml, a jesteś pewny, że zapisałeś go w Scene Builderze, otwórz go w eclipse, a to pomoże go odświeżyć. Jeżeli nie otworzysz pliku, który przed chwilą edytowałeś w innym edytorze, eclipse może korzystać ze starszej wersji dokumentu, który wczytał już do pamięci.

3.6.7 FXML a klasa kontrolera

Nasza aplikacja daje się już w tym momencie uruchomić, jednak w żaden sposób nie możemy odwołać się do pola tekstowego w naszym kodzie Javy. Zgodnie z architekturą MVC powinniśmy móc pobierać i wysyłać informacje do kontrolek zdefiniowanych w widoku poprzez klasę kontrolera. W tym celu musimy zdefiniować w dokumencie fxml dodatkowe atrybuty **fx:id** dla każdej z kontrolek, a najłatwiej będzie to zrobić przełączając najpierw widok w sekcji Document Hierarchy na fx:id właśnie.



Nadając fx:id pamiętaj, żeby były to znaczące nazwy, ponieważ będą to jednocześnie nazwy zmiennych w kodzie Javy. fx:id ustawić klikając dwukrotnie obok danej kontrolki w sekcji Document Hierarchy lub wprowadzając ją w sekcji Code (prawe menu).



Teraz w klasie ustawionej jako fx:controller należy utworzyć zmienne odpowiadające odpowiednim typom kontrolek i nadać im nazwy zgodne z ustalonymi przed chwilą fx:id. Na szczęście nie trzeba tego robić ręcznie. Przejdź w Scene Builderze do sekcji **View -> Show sample controller skeleton** skopiuj przykładowy kod i wklej go do pliku EditorController w eclipse.

W JavieFX ogólnie przyjętą praktyką jest także implementowanie interfejsu **Initializable** przez klasę kontrolera. Interfejs ten wymusi zaimplementowanie metody initialize(), która zostanie wywołana w momencie uruchamiania aplikacji przez FXMLLoadera.

plik EditorController.java

```
1  package application;
2
3  import java.net.URL;
4  import java.util.ResourceBundle;
5
6  import javafx.fxml.FXML;
7  import javafx.fxml.Initializable;
8  import javafx.scene.control.Label;
9  import javafx.scene.control.TextArea;
10
11 public class EditorController implements Initializable {
12
13     @FXML
14     private TextArea mainTextArea;
15
16     @FXML
17     private Label wordsCountLabel;
18
19     @FXML
20     private Label lettersCountLabel;
21
22     @Override
23     public void initialize(URL arg0, ResourceBundle arg1) {
24         // TODO Auto-generated method stub
25
26     }
27 }
```

3.6.8 Odwołanie do kontrolek z kodu Javy

Ostatni etap w tej lekcji to odwoływanie się do kontrolek zdefiniowanych w FXMLu z poziomu klasy kontrolera w kodzie Javy. Tak jak wspomnieliśmy przy wprowadzeniu do Javy FX, kontrolki to tak naprawdę nic innego niż zwykłe klasy Javy. Stworzyliśmy je w sposób deklaratywny w kodzie XML, jednak następnie obiekty zostały wstrzyknięte do klasy kontrolera i tam mamy już do nich dostęp, więc możemy na nich wywoływać odpowiednie metody.

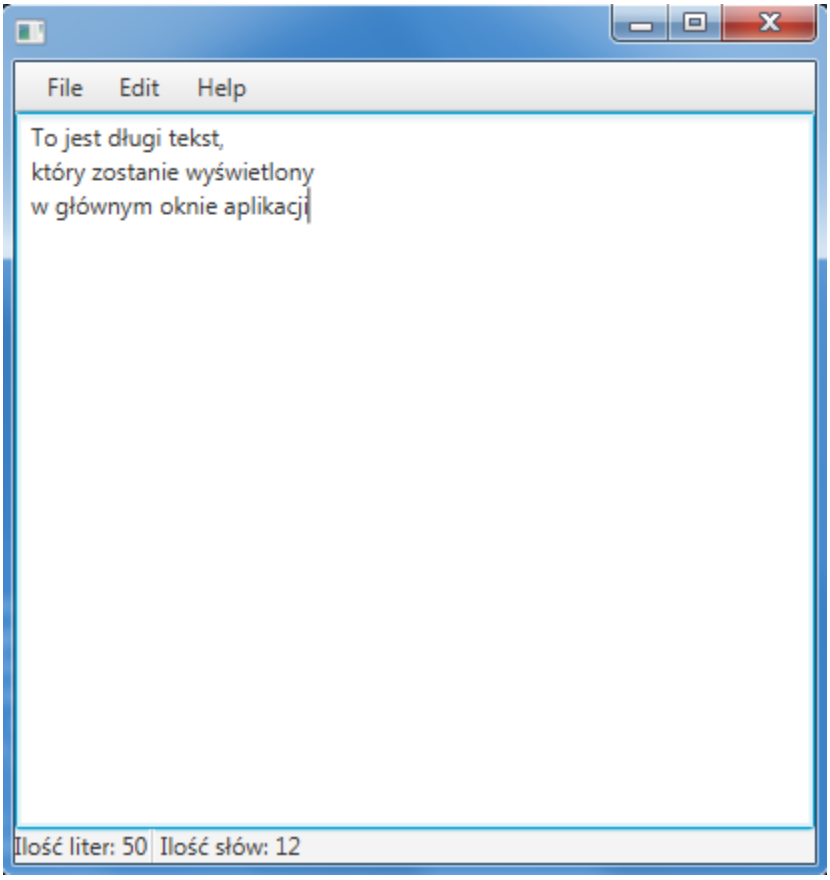
W metodzie initialize() ustawmy tekst naszego głównego pola tekstowego oraz etykiet, aby zweryfikować, że ustawione przez nas fx:id zostały poprawnie powiązane ze zmiennymi.

plik EditorController.java

```
1  package application;
2
3  import java.net.URL;
4  import java.util.ResourceBundle;
5
6  import javafx.fxml.FXML;
7  import javafx.fxml.Initializable;
8  import javafx.scene.control.Label;
9  import javafx.scene.control.TextArea;
10
11 public class EditorController implements Initializable {
12
13     @FXML
14     private TextArea mainTextArea;
15
16     @FXML
17     private Label wordsCountLabel;
18
19     @FXML
20     private Label lettersCountLabel;
21
22     @Override
23     public void initialize(URL arg0, ResourceBundle arg1) {
24         String mainText = "To jest długi tekst,\n"
25                 + "który zostanie wyświetlony\n"
26                 + "w głównym oknie aplikacji";
27         String letters = "Ilość liter: 50";
28         String words = "Ilość słów: 12";
29
30         mainTextArea.setText(mainText);
31         lettersCountLabel.setText(letters);
32         wordsCountLabel.setText(words);
33     }
34 }
```

W wierszach 30-32 ustawiamy teksty odpowiednich kontrolek za pomocą metod *setText()*. Analogicznie w celu odczytania tekstu, który wprowadzi użytkownik będziemy używali metody *getText()*, jednak tego nauczymy się już w kolejnej lekcji omawiając obsługę zdarzeń.

Adnotacja @FXML



3.7 Graficzny interfejs użytkownika cz.2

W tej lekcji dowiesz się:

- w jaki sposó obsługiwać zdarzenia przycisków
- jak dodać dodać obsługę zdarzeń myszy lub klawiatury
- czym są klasy anonimowe

3.7.1 Obsługa zdarzeń

W poprzedniej lekcji dowiedziałeś się w jaki sposób stworzyć prosty graficzny interfejs użytkownika korzystając z Javy FX, na czym polega architektura MVC i jak powiązać kontrolki zdefiniowane w pliku fxml ze zmiennymi w klasie kontrolera, a także jak z poziomu kodu Javy ustawiać ich właściwości na przykładzie metody *setText()*.

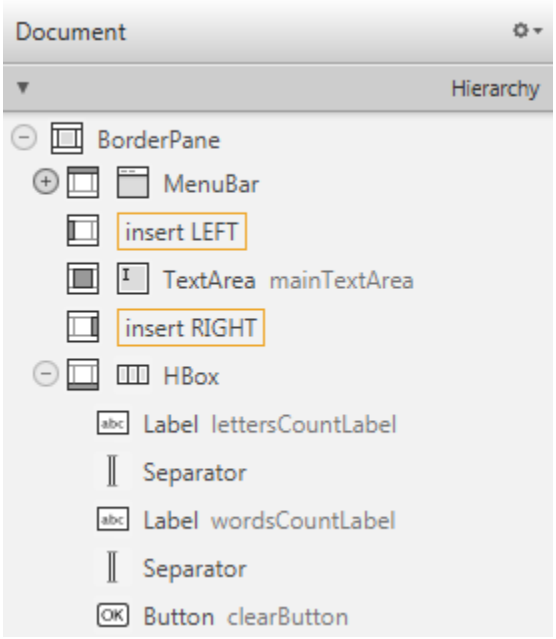
Problemem jest jednak to, że zdefiniowane menu nawigacyjne nie reaguje w żaden sposób na akcje wykonywane przez użytkownika, zliczanie słów oraz wyrazów także jest zakodowane na sztywno i nie reaguje na to co użytkownik wprowadza do pola tekstowego.

Rozwiązaniem tych problemów jest podpięcie pod poszczególne z kontrolek odpowiednich zdarzeń, które w odpowiedzi wykonają pewną akcję. Istnieje kilka metod na realizację tego zadania, jednak my skupimy się na dwóch.

3.7.2 Podpięcie zdarzeń przycisków

Jednymi z najczęściej powtarzających się opcji w niemal każdej aplikacji jest odpowiedź aplikacji na akcję użytkownika polegającą na wciśnięcie zwykłego przycisku (w JavieFX reprezentowany przez klasę Button). W naszym przykładowym programie nie posiadamy na tę chwilę żadnego przycisku, jednak nic nie stoi na przeszkodzie, żeby go dodać.

hierarchia Editor.fxml



plik Editor.fxml

```
<?xml version="1.0" encoding="UTF-8"?>

<?import javafx.scene.control.*?>
<?import java.lang.*?>
<?import javafx.scene.layout.*?>
<?import javafx.scene.layout.BorderPane?>

<BorderPane xmlns="http://javafx.com/javafx/8" xmlns:fx="http://javafx.com/fxml/1"
    fx:controller="application.EditorController">
    <top>
        <MenuBar BorderPane.alignment="CENTER">
            <menus>
                <Menu mnemonicParsing="false" text="File">
                    <items>
                        <MenuItem mnemonicParsing="false" text="Close" />
                    </items>
                </Menu>
                <Menu mnemonicParsing="false" text="Edit">
                    <items>
                        <MenuItem mnemonicParsing="false" text="Delete" />
                    </items>
                </Menu>
                <Menu mnemonicParsing="false" text="Help">
                    <items>
                        <MenuItem mnemonicParsing="false" text="About" />
```

```
                    </items>
                </Menu>
            </menus>
        </MenuBar>
    </top>
    <center>
        <TextArea fx:id="mainTextArea" prefHeight="400.0" prefWidth="500.0"
            BorderPane.alignment="CENTER" />
    </center>
    <bottom>
        <HBox alignment="CENTER_LEFT" BorderPane.alignment="CENTER">
            <children>
                <Label fx:id="lettersCountLabel" text="Label" />
                <Separator orientation="VERTICAL" />
                <Label fx:id="wordsCountLabel" text="Label" />
                <Separator orientation="VERTICAL" />
                <Button fx:id="clearButton" mnemonicParsing="false" text="Clear" />
            </children>
        </HBox>
    </bottom>
</BorderPane>
```

Przycisk dodaliśmy w dolnej części aplikacji ustawiając na nim domyślny tekst “Clear” oraz nadając fx:id *clearButton* - posłuży on nam do wyczyszczenia zawartości pola tekstowego.

Musimy także zdefiniować zmienną odpowiadającą fx:id przycisku w klasie kontrolera, abyśmy mogli dodać do niego obsługę zdarzenia. Robimy to używając adnotacji @FXML i nazwy zmiennej zgodnej z fx:id.

plik EditorController.java

```
package application;

import java.net.URL;
import java.util.ResourceBundle;

import javafx.fxml.FXML;
import javafx.fxml.Initializable;
import javafx.scene.control.Button;
import javafx.scene.control.Label;
import javafx.scene.control.TextArea;

public class EditorController implements Initializable {

    @FXML
    private TextArea mainTextArea;

    @FXML
    private Label wordsCountLabel;

    @FXML
    private Label lettersCountLabel;

    @FXML
    private Button clearButton;

    @Override
    public void initialize(URL arg0, ResourceBundle arg1) {
        String mainText = "To jest długi tekst,\n"
                + "który zostanie wyświetlony\n"
```

```
                + "w głównym oknie aplikacji";
        String letters = "Ilość liter: 50";
        String words = "Ilość słów: 12";

        mainTextArea.setText(mainText);
        lettersCountLabel.setText(letters);
        wordsCountLabel.setText(words);
    }
}
```

Jak widzisz wszystko robimy analogicznie jak w poprzedniej lekcji, czyli:

1. Dodajemy kontrolkę do pliku fxml w Scene Builderze.
2. Nadajemy jej odpowiednie fx:id.
3. Tworzymy zmienną odpowiadającą fx:id w klasie kontrolera oznaczając ją jednocześnie adnotacją @FXML.

3.7.3 Obsługa zdarzenia przycisku

Obsługa zdarzenia przycisku jest najłatwiejszą z tych, które poznamy. Jest to czynność tak często powtarzana, że przycisk (klasa Button) posiada specjalną metodę dedykowaną do tego celu o nazwie **setOnAction()**. Podpięcie akcji przycisku najlepiej jest zrealizować w metodzie initialize(), która będzie wywołana przy uruchamianiu aplikacji.

plik EditorController.java (metoda initialize())

```
@Override
public void initialize(URL arg0, ResourceBundle arg1) {
    //reszta kodu bez zmian
    clearButton.setOnAction(new EventHandler<ActionEvent>() {
        @Override
        public void handle(ActionEvent arg0) {
            //kod obsługi zdarzenia
            mainTextArea.setText("");
        }
    });
}
```

Jak widzisz na przycisku wywołaliśmy metodę **setOnAction()**, w której utworzyliśmy obiekt **EventHandler** z parametrem **ActionEvent**. Problem polega na tym, że jeśli pzyjrzysz się temu kodowi bliżej, to zauważysz, że występuje tutaj dziwne zagnieżdżenie nawiasów, a dodatkowo gdzieś podczas tworzenia obiektu implementujemy jeszcze metodę **handle()**. Jest to spowodowane tym, że EventHandler jest tak naprawdę interfejsem, a to oznacza, że nie możemy utworzyć obiektu na jego podstawie. Rozwiązaniem jest utworzenie **anonimowej klasy wewnętrznej**, czyli takiej klasy, która implementuje interfejs EventHandler i nie posiada swojej nazwy. ActionEvent jest typem zdarzenia jakie chcemy obsłużyć - w tym przypadku odpowiada ono wciśnięciu przycisku.

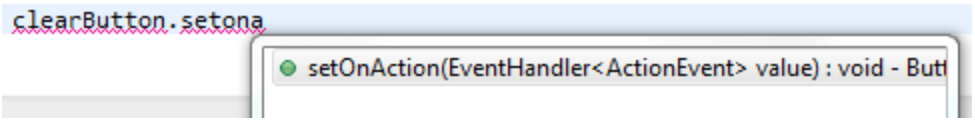
W metodzie *handle()* możemy zamieścić kod obsługi zdarzenia, czyli wywołać odpowiednie metody, ustawić tekst w innych kontrolkach itp. W naszym przypadku przycisk ma wyczyścić pole tekstowe - wywołujemy więc metodę *mainTextArea.setText("");* co oznacza ustawienie pustego Stringa jako jej zawartości.

Uruchom aplikację i przetestuj działanie zaimplementowanego zdarzenia przycisku.

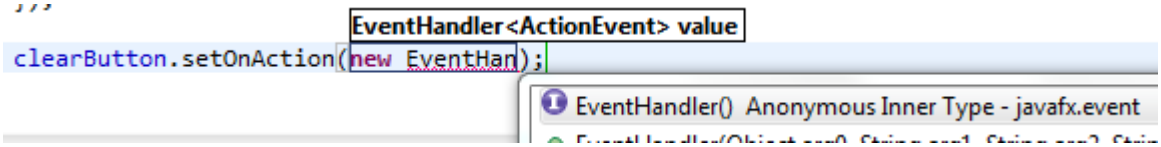
Porada

Pisanie kodu klasy anonimowej byłoby dosyć męczące, szczególnie biorąc pod uwagę nagromadzenie nawiasów i zagnieżdżeń kodu. Na szczęście większość kodu można wygenerować półautomatycznie.

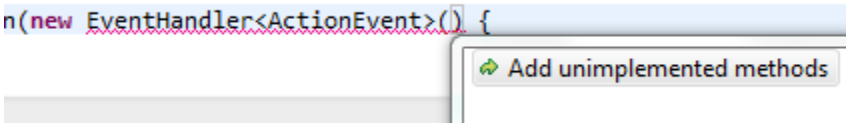
Krok 1 - wpisz po prostu "clearButton.setOna..." i Ctrl+Spacja



Krok 2 - zacznij wpisywać “new EventHandler” i Ctrl+Spacja



Krok 3 - wciśnij Ctrl+1 i wybierz opcję “add unimplemented methods”



3.7.4 Obsługa zdarzeń klawiatury i myszy

Obsługa zdarzeń przycisków jest prosta - wystarczy zapamiętać jedną metodę *setOnAction()* oraz sposób tworzenia klasy anonimowej - praktycznie nic się tam nie zmienia. Jeżeli jednak chcemy obsłużyć zdarzenia klawiatury, czy myszy (wciskanie przycisków, ruch kursora) musimy skorzystać z innej metody. W naszym edytorze tekstu chcemy mieć funkcjonalność zliczania znaków oraz słów. W tym celu zdarzenie należy podpiąć pod obiekt typu **TextArea** jednak tym razem posłużymy się metodą **addEventFilter()**.

plik EditorController.java (metoda initialize())

```
@Override
public void initialize(URL arg0, ResourceBundle arg1) {

    final String lettersCount = "Ilość znaków: ";
    final String wordsCount = "Ilość słów: ";

    //domyślne etykiety ustawione na 0
    lettersCountLabel.setText(lettersCount + 0);
    wordsCountLabel.setText(wordsCount + 0);

    clearButton.setOnAction(new EventHandler<ActionEvent>() {
        @Override
        public void handle(ActionEvent arg0) {
            //kod obsługi zdarzenia
            mainTextArea.setText("");

            //po wciśnięciu przycisku zmieniamy tekst etykiet na domyślny
            lettersCountLabel.setText(lettersCount + 0);
            wordsCountLabel.setText(wordsCount + 0);
        }
    });

    mainTextArea.addEventFilter(KeyEvent.KEY_TYPED, new EventHandler<KeyEvent>() {
        @Override
        public void handle(KeyEvent event) {
            //licznik znaków z pominięciem spacji
            int letters = mainTextArea.getText().trim().length();
            lettersCountLabel.setText(lettersCount + letters);
```

```
29
30              //licznik słów
31              int words = mainTextArea.getText().split(" ").length;
32              wordsCountLabel.setText(wordsCount + words);
33          }
34      });
35  }
```

W wierszu 23 dodaliśmy zdarzenie do obiektu mainTextArea. Metodzie addEventFilter należy przekazać dwa argumenty:

- typ zdarzenia jakie chcemy obsłużyć - u nas jest to KeyEvent.KEY_TYPED co odpowiada wciśnięciu i zwolnieniu dowolnego klawisza klawiatury
- obiekt EventHandler (podobnie jak w metodzie setOnAction), jednak z parametrem generycznym zgodnym z typem zdarzenia - w naszym przypadku typem tym jest KeyEvent.

W wierszach 26-32 definiujemy co ma się stać po wciśnięciu dowolnego klawisza. Liczbę znaków zliczamy usuwając najpierw białe znaki metodą *trim()* z klasy String, a następnie pobierając długość takiego napisu metodą *length()*. Ilość słów zliczamy podobnie, jednak najpierw pobrany tekst dzielimy na jednowymiarową tablicę słów, które były rozdzielone znakiem spacji " ". Właściwość length tablicy jest więc równa ilości słów w naszym tekście.

Teksty etykiet (Label) podobnie jak to było z obiektem TextArea ustawiamy za pomocą metody *setText()*.

Dostępne zdarzenia klawiatury, które możemy obsługiwać to:

- KeyEvent.KEY_PRESSED - wciśnięcie klawisza
- KeyEvent.KEY_RELEASED - zwolnienie klawisza
- KeyEvent.KEY_TYPED - wciśnięcie i zwolnienie klawisza

Jeżeli chcielibyśmy obsłużyć zdarzenia myszy, zrobimy to bardzo podobnie, jednak wykorzystamy klasę MouseEvent. Wybrane typy zdarzeń:

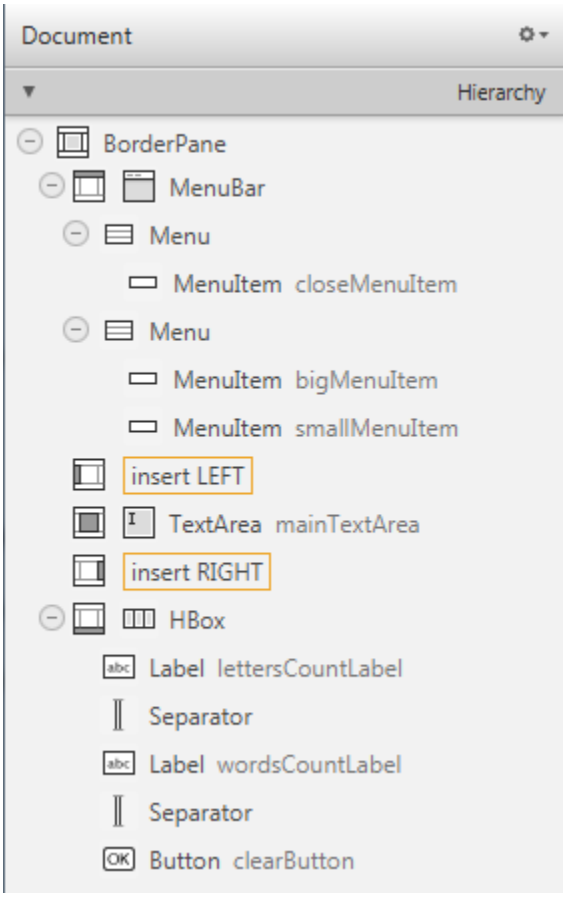
- MouseEvent.MOUSE_CLICKED - wciśnięcie i zwolnienie przycisku myszy
- MouseEvent.MOUSE_ENTERED - najechanie kursorem nad daną kontrolkę, do którego podpięte jest zdarzenie
- MouseEvent.MOUSE_DRAGGED - przeciągnięcie myszy z wciśniętym przyciskiem myszy

Metodę *addEventFilter()* można wywołać na niemal dowolnym obiekcie JavyFX, więc w identyczny sposób można obsługiwać zdarzenia pól tekstowych, etykie, buttonów, czy też menu.

3.7.5 Ćwiczenie

Dodaj do programu następujące opcje:

- Exit z menu File, które powoduje zamknięcie aplikacji. Aplikację JavyFX można zakończyć wywołując metodę *Platform.exit()*
- Big Letters / Small Letters w menu Edit - opcje powodujące podmianę tekstu na wielkie lub małe litery



plik Editor.fxml

```
<?xml version="1.0" encoding="UTF-8"?>

<?import javafx.scene.control.*?>
<?import java.lang.*?>
<?import javafx.scene.layout.*?>
<?import javafx.scene.layout.BorderPane?>

<BorderPane xmlns="http://javafx.com/javafx/8" xmlns:fx="http://javafx.com/fxml/1"
    fx:controller="application.EditorController">
    <top>
        <MenuBar BorderPane.alignment="CENTER">
            <menus>
                <Menu mnemonicParsing="false" text="File">
                    <items>
                        <MenuItem fx:id="closeMenuItem" mnemonicParsing="false"
                            text="Close" />
                    </items>
                </Menu>
                <Menu mnemonicParsing="false" text="Edit">
                    <items>
                        <MenuItem fx:id="bigMenuItem" mnemonicParsing="false"
                            text="BIG LETTERS" />
                        <MenuItem fx:id="smallMenuItem" mnemonicParsing="false"
                            text="small letters" />
                    </items>
                </Menu>
```

```
            </menus>
        </MenuBar>
    </top>
    <center>
        <TextArea fx:id="mainTextArea" prefHeight="400.0" prefWidth="500.0"
            BorderPane.alignment="CENTER" />
    </center>
    <bottom>
        <HBox alignment="CENTER_LEFT" BorderPane.alignment="CENTER">
            <children>
                <Label fx:id="lettersCountLabel" text="Label" />
                <Separator orientation="VERTICAL" />
                <Label fx:id="wordsCountLabel" text="Label" />
                <Separator orientation="VERTICAL" />
                <Button fx:id="clearButton" mnemonicParsing="false" text="Clear" />
            </children>
        </HBox>
    </bottom>
</BorderPane>
```

plik EditorCntroller.java

```
package application;

import java.net.URL;
import java.util.ResourceBundle;

import javafx.application.Platform;
import javafx.event.ActionEvent;
import javafx.event.EventHandler;
import javafx.fxml.FXML;
import javafx.fxml.Initializable;
import javafx.scene.control.Button;
import javafx.scene.control.Label;
import javafx.scene.control.MenuItem;
import javafx.scene.control.TextArea;
import javafx.scene.input.KeyEvent;

public class EditorController implements Initializable {

    @FXML
    private TextArea mainTextArea;

    @FXML
    private Label wordsCountLabel;

    @FXML
    private Button clearButton;

    @FXML
    private Label lettersCountLabel;

    @FXML
    private MenuItem closeMenuItem;

    @FXML
    private MenuItem bigMenuItem;

    @FXML
```

```
38      private MenuItem smallMenuItem;
39
40      @Override
41      public void initialize(URL arg0, ResourceBundle arg1) {
42
43          final String lettersCount = "Ilość znaków: ";
44          final String wordsCount = "Ilość słów: ";
45
46          //domyślne etykiety ustawione na 0
47          lettersCountLabel.setText(lettersCount + 0);
48          wordsCountLabel.setText(wordsCount + 0);
49
50          clearButton.setOnAction(new EventHandler<ActionEvent>() {
51              @Override
52              public void handle(ActionEvent arg0) {
53                  //kod obsługi zdarzenia
54                  mainTextArea.setText("");
55
56                  //po wciśnięciu przycisku zmieniamy tekst etykiet na domyślny
57                  lettersCountLabel.setText(lettersCount + 0);
58                  wordsCountLabel.setText(wordsCount + 0);
59
60              }
61          });
62
63          mainTextArea.addEventFilter(KeyEvent.KEY_TYPED, new EventHandler<KeyEvent>() {
64              @Override
65              public void handle(KeyEvent event) {
66                  //licznik znaków z pominięciem spacji
67                  int letters = mainTextArea.getText().trim().length();
68                  lettersCountLabel.setText(lettersCount + letters);
69
70                  //licznik słów
71                  int words = mainTextArea.getText().split(" ").length;
72                  wordsCountLabel.setText(wordsCount + words);
73              }
74          });
75
76          closeMenuItem.addEventHandler(ActionEvent.ACTION, new EventHandler<ActionEvent>() {
77              @Override
78              public void handle(ActionEvent event) {
79                  //zamknięcie aplikacji
80                  Platform.exit();
81              }
82          });
83
84          bigMenuItem.addEventHandler(ActionEvent.ACTION, new EventHandler<ActionEvent>() {
85              @Override
86              public void handle(ActionEvent event) {
87                  String text = mainTextArea.getText();
88                  text = text.toUpperCase();
89                  mainTextArea.setText(text);
90              }
91          });
92
93          smallMenuItem.addEventHandler(ActionEvent.ACTION, new EventHandler<ActionEvent>() {
94              @Override
95              public void handle(ActionEvent event) {
```

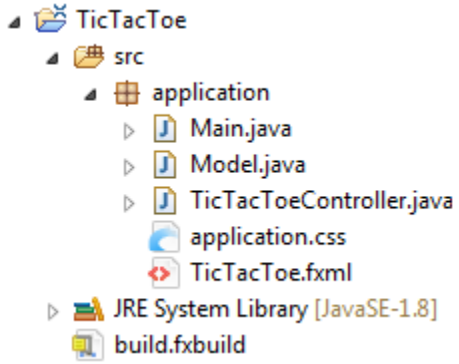
```
96                  String text = mainTextArea.getText();
97                  text = text.toLowerCase();
98                  mainTextArea.setText(text);
99              }
100         });
101
102     }
103 }
```

3.8 Praktyczna aplikacja

Jako podsumowanie naszej nauki Javy napiszemy prostą grę - kółki i krzyżyk wykorzystując wcześniej poznane elementy. Zaczniemy od zdefiniowania widoku, następnie utworzymy prosty model aplikacji a na końcu połączymy całość w klasie kontrolera.

3.8.1 Projekt

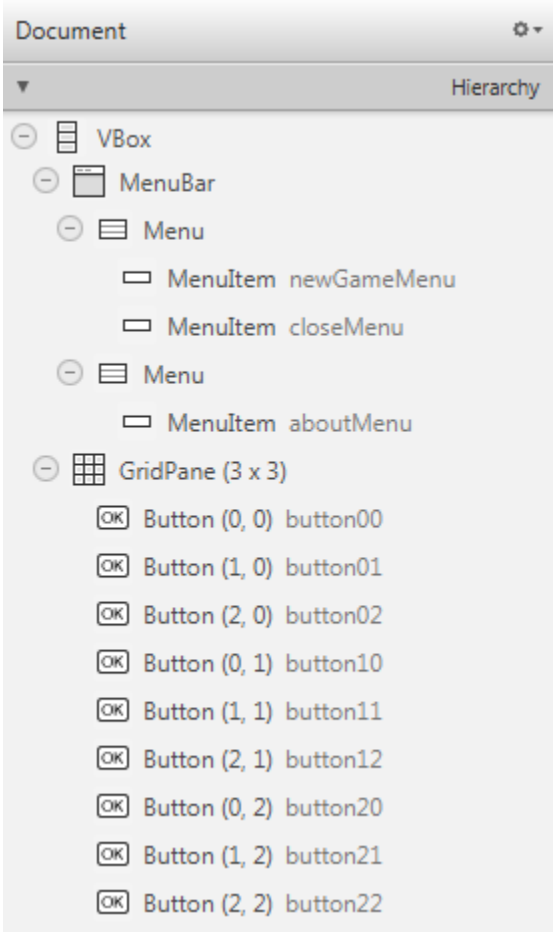
Zacznijmy od utworzenia nowego projektu wykorzystującego FXML w eclipse. Jako główny layout aplikacji (ostatnia zakładka tworzenia projektu) wybierz VBox.



- Main.java - główna klasa aplikacji
- Model.java - model danych
- TicTacToeController - klasa kontrolera
- TicTacToe.fxml - widok aplikacji
- application.css - definicja css dla widoku - pokażemy jak w prosty sposób wykorzystać style css do poprawy wyglądu aplikacji

3.8.2 Widok

Otwórz plik fxml w Scene builderze i odtwórz poniższą hierarchię layoutów i kontrolek uwzględniając także odpowiednie atrybuty fx:id.

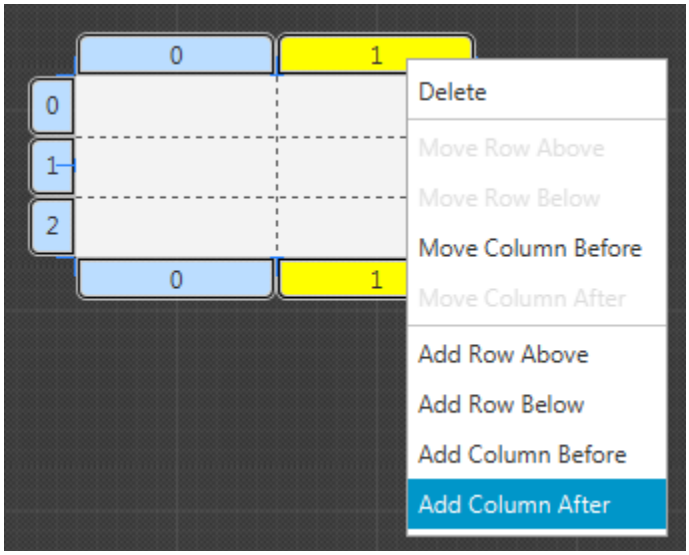


Jako nadrzędny layout wykorzystujemy VBox, ponieważ pozwala on ustawiać kontrolki i layouty w pojedynczej kolumnie. My potrzebujemy jedną kolumnę z dwoma wierszami - w jednym z nich umieszczamy podstawowe menu, a w drugim layout typu GridLayout.

W Menu dostępne będą trzy opcje:

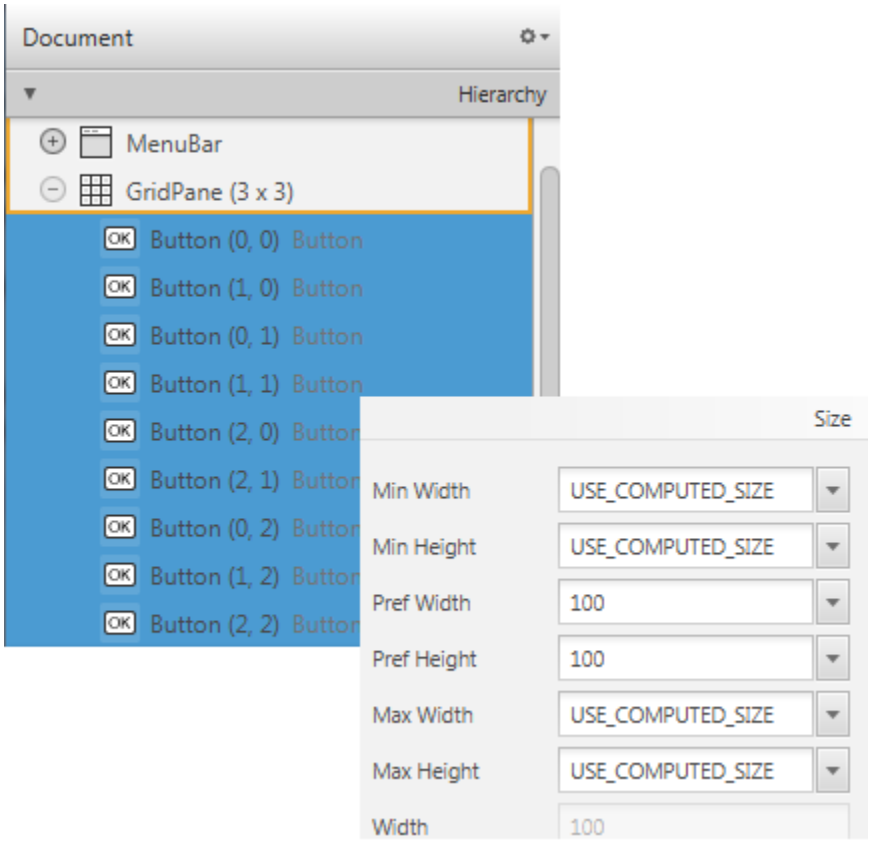
- New Game (nowa gra) - w menu File
- Close (wyjście z programu) - w menu File
- About (o programie) - w menu Help

Layout znajdujący się w drugim wierszu naszego VBoxa pozwala na utworzenie siatki o dowolnym rozmiarze - my potrzebujemy oczywiście siatkę 3x3, w której następnie umieścimy Buttony. Kolejne kolumny do GridPane można dodać po jego wcześniejszym zaznaczeniu i wybraniu opcji Add Row lub Add Column.

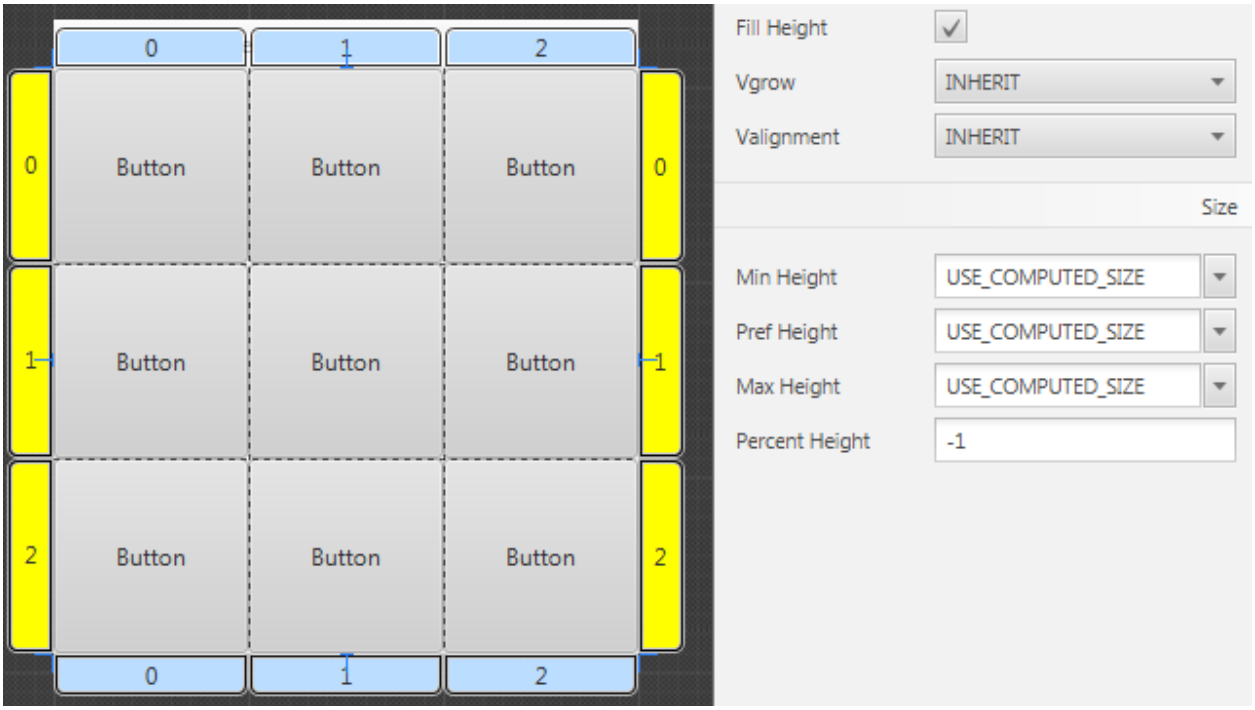


Problem jaki teraz się pojawia polega na tym, że przyciski są prostokątne, nie wykorzystują całej przestrzeni w GridPane a my chcielibyśmy, żeby były kwadratowe i nie było między nimi przerw.

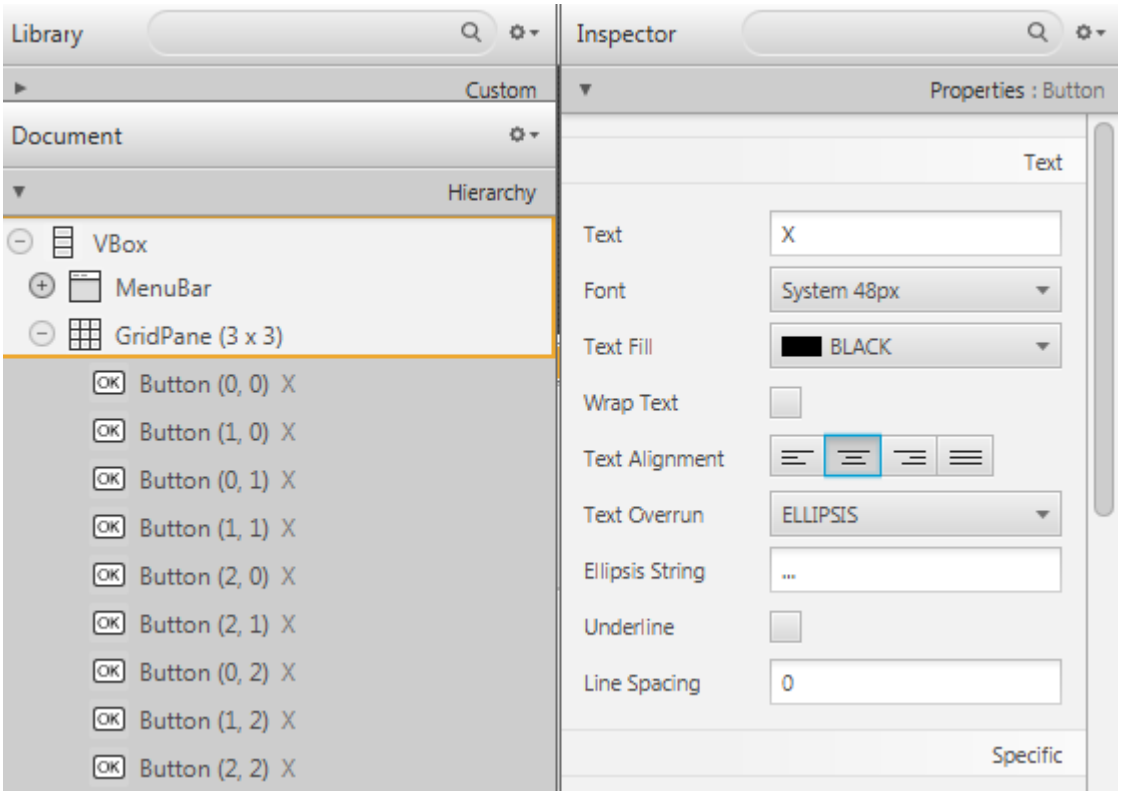
W tym celu zaznacz wszystkie przyciski w sekcji Document Hierarchy (np. z wciśniętym Shiftem) i przejdź do sekcji Layout po prawej stronie Scene Buildera. Tam ustaw porządany rozmiar przycisku wprowadzając opcję Pref Width oraz Pref Height (np. na 100 px).



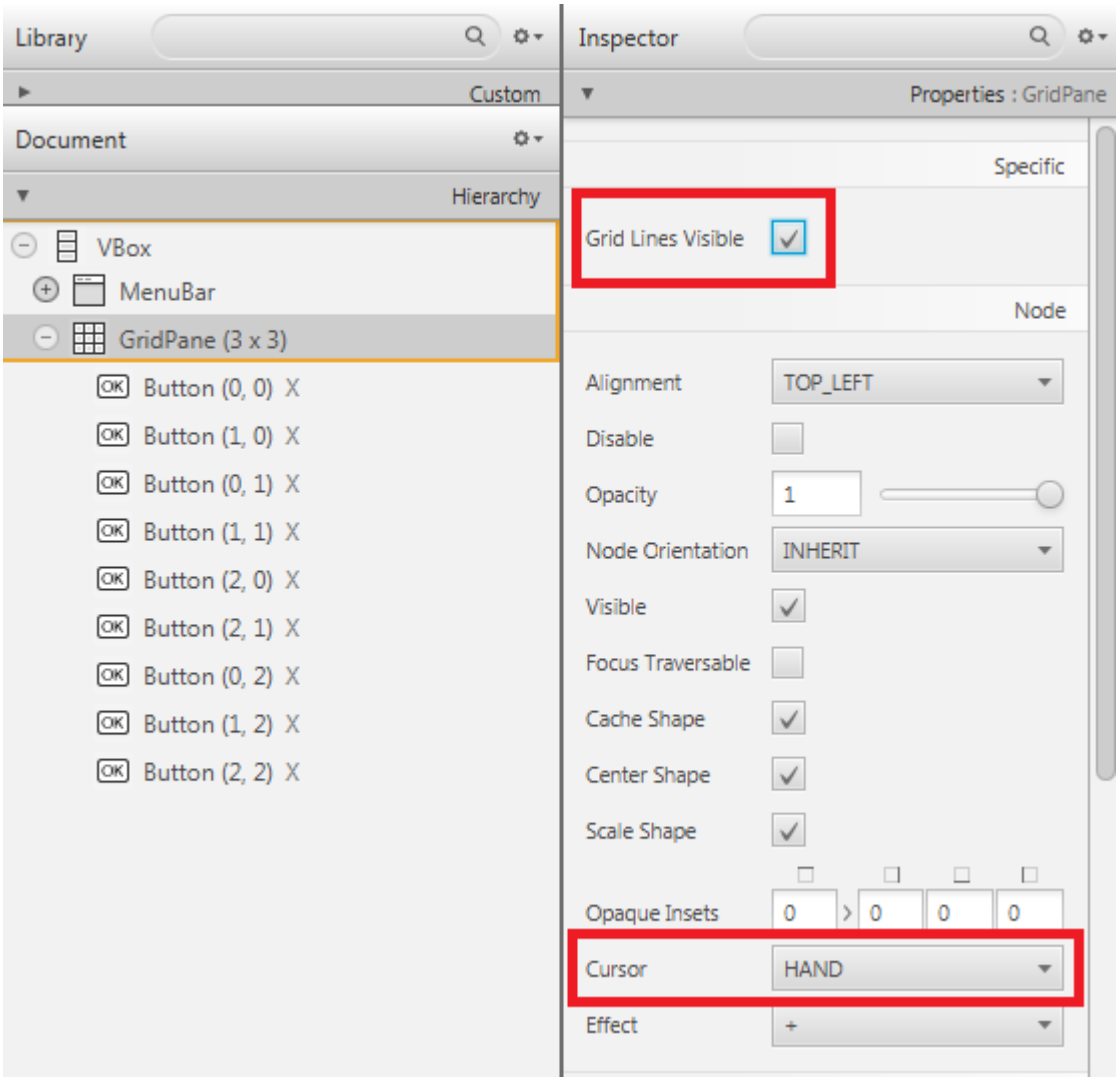
Przyciski zajmują już całą przestrzeń, ale nadal nie mają ustalonego przez nas rozmiaru 100x100. Jest to spowodowane tym, że kolumny i wiersze w GridPane mają zdefiniowane różne rozmiary i mają one wyższy priorytet niż rozmiar przycisków. Kliknij w każdy wiersz i kolumnę oraz wiersz GridPane i ustaw ich rozmiar (Min i Pref Width/Height) na *USE_COMPUTED_SIZES*. W tej sytuacji wykorzystane zostaną wymiary węzłów potomnych.



Zaznacz ponownie wszystkie przyciski i przejdźmy jeszcze na chwilę do ich sekcji Properties, gdzie ustawimy większą czcionkę (będzie na nich wyświetlany tylko X lub 0).



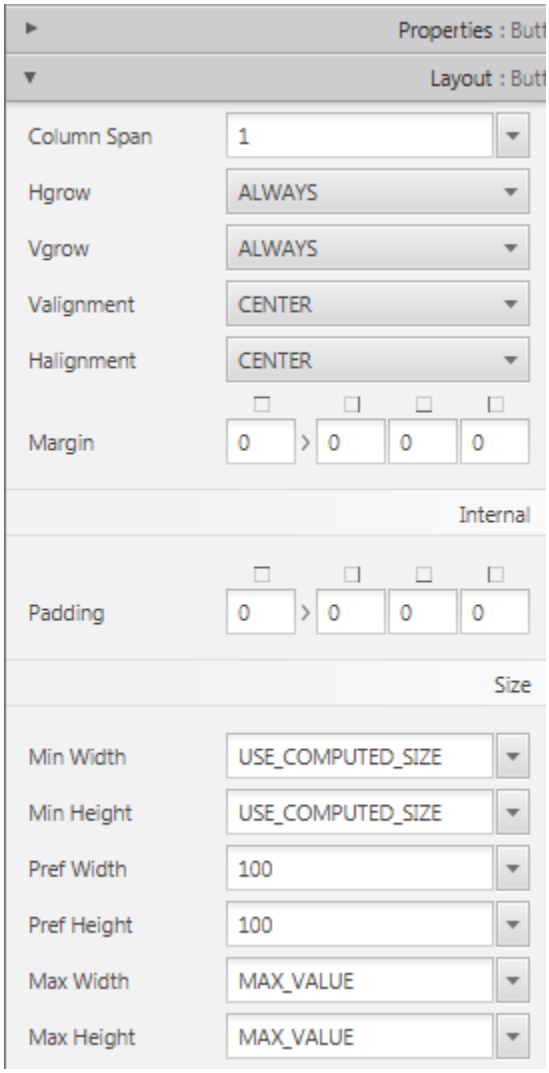
Dla lepszego ostatecznego wyglądu dodajmy jeszcze linie siatki w celu wyraźniejszego oddzielenia przycisków, a także zmieńmy wygląd kursora:



Z przycisków możesz usunąć domyślny tekst "Button" klikając na każdy z nich dwukrotnie. Jeżeli nie zrobiłeś tego na etapie tworzenia projektu, to dodatkowo ustaw odpowiednią ścieżkę do klasy kontrolera, czyli application.TicTacToeController.

Jeżeli chcesz, aby przyciski zawsze wypełniały całą dostępną przestrzeń, czyli np. powiększały się przy rozciąganiu okna musisz ustawić kilka rzeczy:

- właściwości VGrow i HGrow dla kolumn lub wierszy GridPane na wartość *ALWAYS* - w tym celu kliknij na daną kolumnę lub wiersz i w sekcji Layout ustaw odpowiednią wartość,
- właściwości VGrow i HGrow dla każdego przycisku ustawione na *ALWAYS* - zaznacz wszystkie przyciski i w sekcji Layout ustaw wartości HGrow i VGrow,
- ustawienie maxymalnego rozmiaru przycisków na *MAX_VALUE* - możesz je ustawić zaznaczając wszystkie przyciski i ustawiając w sekcji Layout właściwości Max Width i Max Height



Ostatecznie nasz plik fxml powinien mieć następującą postać:

plik TicTacToe.fxml

```
<?xml version="1.0" encoding="UTF-8"?>

<?import javafx.scene.text.*?>
<?import javafx.scene.control.*?>
<?import java.lang.*?>
<?import javafx.scene.layout.*?>
<?import javafx.scene.layout.GridPane?>

<VBox xmlns="http://javafx.com/javafx/8" xmlns:fx="http://javafx.com/fxml/1"
    fx:controller="application.TicTacToeController">
    <children>
        <MenuBar>
            <menus>
                <Menu mnemonicParsing="false" text="File">
                    <items>
                        <MenuItem fx:id="newGameMenu" mnemonicParsing="false"
                            text="New Game" />
                        <MenuItem fx:id="closeMenu" mnemonicParsing="false" text="Close" />
```

```
                    </items>
                </Menu>
                <Menu mnemonicParsing="false" text="Help">
                    <items>
                        <MenuItem fx:id="aboutMenu" mnemonicParsing="false" text="About" />
                    </items>
                </Menu>
            </menus>
        </MenuBar>
        <GridPane gridLinesVisible="true" VBox.vgrow="ALWAYS">
            <columnConstraints>
                <ColumnConstraints hgrow="ALWAYS" />
                <ColumnConstraints hgrow="ALWAYS" />
                <ColumnConstraints hgrow="ALWAYS" />
            </columnConstraints>
            <rowConstraints>
                <RowConstraints vgrow="ALWAYS" />
                <RowConstraints vgrow="ALWAYS" />
                <RowConstraints vgrow="ALWAYS" />
            </rowConstraints>
            <children>
                <Button fx:id="button00" contentDisplay="CENTER"
                    maxHeight="1.7976931348623157E308" maxWidth="1.7976931348623157E308"
                    mnemonicParsing="false" prefHeight="100.0" prefWidth="100.0"
                    GridPane.halignment="CENTER" GridPane.hgrow="ALWAYS"
                    GridPane.valignment="CENTER" GridPane.vgrow="ALWAYS">
                    <font>
                        <Font size="36.0" />
                    </font>
                </Button>
                <Button fx:id="button01" contentDisplay="CENTER"
                    maxHeight="1.7976931348623157E308" maxWidth="1.7976931348623157E308"
                    mnemonicParsing="false" prefHeight="100.0" prefWidth="100.0"
                    GridPane.columnIndex="1" GridPane.halignment="CENTER"
                    GridPane.hgrow="ALWAYS" GridPane.valignment="CENTER"
                    GridPane.vgrow="ALWAYS">
                    <font>
                        <Font size="36.0" />
                    </font>
                </Button>
                <Button fx:id="button02" contentDisplay="CENTER"
                    maxHeight="1.7976931348623157E308" maxWidth="1.7976931348623157E308"
                    mnemonicParsing="false" prefHeight="100.0" prefWidth="100.0"
                    GridPane.columnIndex="2" GridPane.halignment="CENTER"
                    GridPane.hgrow="ALWAYS" GridPane.valignment="CENTER"
                    GridPane.vgrow="ALWAYS">
                    <font>
                        <Font size="36.0" />
                    </font>
                </Button>
                <Button fx:id="button10" contentDisplay="CENTER"
                    maxHeight="1.7976931348623157E308" maxWidth="1.7976931348623157E308"
                    mnemonicParsing="false" prefHeight="100.0" prefWidth="100.0"
                    GridPane.halignment="CENTER" GridPane.hgrow="ALWAYS"
                    GridPane.rowIndex="1" GridPane.valignment="CENTER" GridPane.vgrow="ALWAYS">
                    <font>
                        <Font size="36.0" />
                    </font>
```

```
                </Button>
                <Button fx:id="button11" contentDisplay="CENTER"
                    maxHeight="1.7976931348623157E308" maxWidth="1.7976931348623157E308"
                    mnemonicParsing="false" prefHeight="100.0" prefWidth="100.0"
                    GridPane.columnIndex="1" GridPane.halignment="CENTER"
                    GridPane.hgrow="ALWAYS" GridPane.rowIndex="1" GridPane.valignment="CENTER"
                    GridPane.vgrow="ALWAYS">
                    <font>
                        <Font size="36.0" />
                    </font>
                </Button>
                <Button fx:id="button12" contentDisplay="CENTER"
                    maxHeight="1.7976931348623157E308" maxWidth="1.7976931348623157E308"
                    mnemonicParsing="false" prefHeight="100.0" prefWidth="100.0"
                    GridPane.columnIndex="2" GridPane.halignment="CENTER"
                    GridPane.hgrow="ALWAYS" GridPane.rowIndex="1" GridPane.valignment="CENTER"
                    GridPane.vgrow="ALWAYS">
                    <font>
                        <Font size="36.0" />
                    </font>
                </Button>
                <Button fx:id="button20" contentDisplay="CENTER"
                    maxHeight="1.7976931348623157E308" maxWidth="1.7976931348623157E308"
                    mnemonicParsing="false" prefHeight="100.0" prefWidth="100.0"
                    GridPane.halignment="CENTER" GridPane.hgrow="ALWAYS"
                    GridPane.rowIndex="2" GridPane.valignment="CENTER" GridPane.vgrow="ALWAYS">
                    <font>
                        <Font size="36.0" />
                    </font>
                </Button>
                <Button fx:id="button21" contentDisplay="CENTER"
                    maxHeight="1.7976931348623157E308" maxWidth="1.7976931348623157E308"
                    mnemonicParsing="false" prefHeight="100.0" prefWidth="100.0"
                    GridPane.columnIndex="1" GridPane.halignment="CENTER"
                    GridPane.hgrow="ALWAYS" GridPane.rowIndex="2" GridPane.valignment="CENTER"
                    GridPane.vgrow="ALWAYS">
                    <font>
                        <Font size="36.0" />
                    </font>
                </Button>
                <Button fx:id="button22" contentDisplay="CENTER"
                    maxHeight="1.7976931348623157E308" maxWidth="1.7976931348623157E308"
                    mnemonicParsing="false" prefHeight="100.0" prefWidth="100.0"
                    GridPane.columnIndex="2" GridPane.halignment="CENTER"
                    GridPane.hgrow="ALWAYS" GridPane.rowIndex="2" GridPane.valignment="CENTER"
                    GridPane.vgrow="ALWAYS">
                    <font>
                        <Font size="36.0" />
                    </font>
                </Button>
            </children>
        </GridPane>
    </children>
</VBox>
```

W pliku fxml widzimy wynik wcześniejszych operacji, nie powinno być w nim nic czego wcześniej nie poznaliśmy - konfiguracja layoutu oraz kilku przycisków z odpowiednio nadanymi atrybutami fx:id.

3.8.3 Model

Klasa modelu w naszym przypadku będzie stanowiła główną logikę aplikacji. Zamieścimy w niej zarówno model danych, czyli odwzorowanie planszy jak i logikę biznesową, czyli metody, które sprawdzą , czy ktoś już przypadkiem nie wygrał.

- -1 - kółko
- 0 - puste pole
- 1 - krzyżyk

plik Model.java

```
package application;

public class Model {
    public static final int X = 1;
    public static final int O = -1;
    public static final int BLANK = 0;

    public static final int SIZE = 3;
    private int[][] table;
    private int activePlayer;

    public void setValue(int x, int y, int value) {
        table[x][y] = value;
    }

    public int getValue(int x, int y) {
        return table[x][y];
    }

    public int getActivePlayer() {
        return activePlayer;
    }

    public void switchPlayer() {
        activePlayer = -activePlayer;
    }

    /*
     * Konstruktor
     */
    public Model() {
        table = new int[SIZE][SIZE];
        activePlayer = O;
    }

    /*
     * Metoda sprawdzająca kto wygrał
     */
    public int getWinner() {
        int winner = BLANK;

        //sprawdzamy wiersze
        for(int row=0; row < SIZE; row++) {
            for(int col=1; col < SIZE; col++) {
                if(table[row][col] !=  table[row][col-1]) {
                    //przerywamy sprawdzanie tego wiersza
```

```
47                      break;
48                  } else if(col == SIZE-1) {
49                      winner = table[row][col];
50                      return winner;
51                  }
52              }
53          }
54
55          //sprawdzamy kolumny
56          for(int row=0; row < SIZE; row++) {
57              for(int col=1; col < SIZE; col++) {
58                  if(table[col][row] !=  table[col-1][row]) {
59                      //przerywamy sprawdzanie tej kolumny
60                      break;
61                  } else if(col == SIZE-1) {
62                      winner = table[col][row];
63                      return winner;
64                  }
65              }
66          }
67
68          //sprawdzamy pierwszą przekątną
69          for(int i=1; i<SIZE; i++) {
70              if(table[i][i] != table[i-1][i-1]) {
71                  break;
72              } else if(i == SIZE-1) {
73                  winner = table[i][i];
74                  return winner;
75              }
76          }
77
78          //sprawdzamy drugą przekątną
79          for(int i=0; i < SIZE-1; i++) {
80              if(table[i][SIZE-1 - i] != table[i+1][SIZE-2 - i]) {
81                  break;
82              } else if(i == SIZE-2) {
83                  winner = table[i][i];
84                  return winner;
85              }
86          }
87
88          return winner;
89      }
90  }
```

Powyższy kod może wydawać Ci się nieco skomplikowany jak na nasze potrzeby, ale jego zaletą jest to, że dzięki wykorzystaniu pętli możemy go wykorzystać zarówno w planszy 3x3, 4x4 jak i większych. W praktyce w takie gry raczej się nie gra, bo prawie zawsze padałby remis, jednak rozwiązanie takie i tak wydaje się lepsze niż kilkadziesiąt warunków if sprawdzających wszystkie wiersze i kolumny osobno.

Bardziej szczegółowy opis klasy poniżej.

Wiersze 4-6 to deklaracja stałych, które posłużą nam do wypełniania i sprawdzania stanów danego pola na planszy. X to krzyżyk O to kółko, a BLANK oznacza pole puste. W wierzu 8 deklarujemy stałą definiującą rozmiar planszy - w tym przypadku 3x3 W 9 wierszu widzimy tablicę dwuwymiarową, która będzie reprezentowała naszą planszę Wiersz 10 to zmienna *activePlayer*, w której przechowywany będzie aktualny gracz - zmieniany co rundę.

W Javie zwykło się oznaczać pola, które nie są stałymi za pomocą prywatnego specyfikatora dostępu. Ponieważ są one prywatne potrzebujemy utworzyć metody dostępowe do tych pól.

getValue() zwraca pole w tablicy o współrzędnych x, y, *setValue()* odpowiednio je ustawia, *getActivePlayer()* zwraca aktualnie ustawionego gracza, *switchPlayer()* ustawia zmienną *activePlayer* na wartość ze zmienionym znakiem, czyli z 1 na -1 lub z -1 na 1.

W konstruktorze inicjalizujemy tablicę o zadanym rozmiarze oraz ustawiamy gracza rozpoczynającego grę na O (kółko zaczyna).

Metoda *getWinner()* jest najbardziej skomplikowana i służy do sprawdzenia warunków, czy w którymś wierszu, kolumnie, lub dowolnej z przekątnych znajdują się wartości jednego typu (kółko lub krzyżyk, reprezentowane w programie przez wartości 1 lub -1). Jeżeli napotkamy na pierwszy przypadek, w którym warunek zwycięstwa jest prawdziwy kończymy działanie pętli i zwracamy w wyniku wartość zgodną ze zwycięzkim graczem (kółko lub krzyżyk). Jeżeli wszystkie pętle zostaną ukończone oznacza to, że nie ma jeszcze zwycięzcy i zwracana jest wartość BLANK, którą zainicjowaliśmy zmienną winner.

3.8.4 Kontroler

Klasa kontrolera pozwoli nam połączyć klasę Modelu z widokiem wcześniej zdefiniowanym w pliku fxml.

plik TicTacToeController.java

```
1  package application;
2
3  import java.net.URL;
4  import java.util.ResourceBundle;
5
6  import javafx.application.Platform;
7  import javafx.event.ActionEvent;
8  import javafx.event.EventHandler;
9  import javafx.fxml.FXML;
10 import javafx.fxml.Initializable;
11 import javafx.scene.control.Button;
12 import javafx.scene.control.MenuItem;
13 import javafx.scene.control.TextArea;
14 import javafx.scene.input.MouseEvent;
15 import javafx.stage.Popup;
16
17 public class TicTacToeController implements Initializable {
18
19     @FXML
20     private MenuItem newGameMenu;
21
22     @FXML
23     private Button button02;
24
25     @FXML
26     private Button button10;
27
28     @FXML
29     private Button button21;
30
31     @FXML
32     private Button button20;
33
34     @FXML
35     private Button button01;
36
37     @FXML
38     private Button button12;
```

```
39
40      @FXML
41      private Button button00;
42
43      @FXML
44      private Button button11;
45
46      @FXML
47      private Button button22;
48
49      @FXML
50      private MenuItem closeMenu;
51
52      @FXML
53      private MenuItem aboutMenu;
54
55      private Model model;
56      private Button[][] buttons;
57
58      @Override
59      public void initialize(URL arg0, ResourceBundle arg1) {
60          model = new Model();
61          buttons = new Button[][] { { button00, button01, button02 },
62                  { button10, button11, button12 },
63                  { button20, button21, button22 } };
64          addButtonAction();
65          addMenuAction();
66      }
67
68      /*
69       * metoda resetująca grę - tworzy nowy model i ustawia pusty tekst
70       * przycisków
71       */
72      private void newGame() {
73          model = new Model();
74          for (int i = 0; i < buttons.length; i++) {
75              for (int j = 0; j < buttons[i].length; j++) {
76                  buttons[i][j].setText("");
77              }
78          }
79      }
80
81      /*
82       * Dodanie akcji do przycisków na planszy
83       */
84      private void addButtonAction() {
85          for (int i = 0; i < buttons.length; i++) {
86              for (int j = 0; j < buttons[i].length; j++) {
87                  final int x = i, y = j;
88                  buttons[i][j].setOnAction(new EventHandler<ActionEvent>() {
89
90                      @Override
91                      public void handle(ActionEvent event) {
92                          String buttonText = buttons[x][y].getText();
93                          if ("".equals(buttonText)) {
94                              model.setValue(x, y, model.getActivePlayer());
95                              updateView();
96                              model.switchPlayer();
```

```
                        }

                        if (model.getWinner() != Model.BLANK) {
                            showWinnerPopup();
                        }
                    }
                });
            }
        }
    }

    /*
     * Metoda dodająca akcje pod poszczególne elementy menu
     */
    private void addMenuAction() {
        newGameMenu.setOnAction(new EventHandler<ActionEvent>() {
            @Override
            public void handle(ActionEvent event) {
                newGame();
            }
        });

        closeMenu.setOnAction(new EventHandler<ActionEvent>() {
            @Override
            public void handle(ActionEvent event) {
                Platform.exit();
            }
        });

        aboutMenu.setOnAction(new EventHandler<ActionEvent>() {
            @Override
            public void handle(ActionEvent event) {
                String aboutText = "Gra w kółko i krzyżyk\n"
                        + "Koduj z klasą 2014";
                createPopup(aboutText);
            }
        });
    }

    /**
     * Aktualizujemy wygląd przycisków na podstawie modelu
     */
    private void updateView() {
        for (int i = 0; i < Model.SIZE; i++) {
            for (int j = 0; j < Model.SIZE; j++) {
                if (model.getValue(i, j) == Model.X) {
                    buttons[i][j].setText("X");
                } else if (model.getValue(i, j) == Model.O) {
                    buttons[i][j].setText("O");
                }
            }
        }
    }

    /*
     * wyświetlenie popupu z informacją o zwycięzcy
     */
    private void showWinnerPopup() {
```

```
155         String winner = null;
156         if (model.getWinner() == Model.X) {
157             winner = "X";
158         } else if (model.getWinner() == Model.O) {
159             winner = "O";
160         }
161 
162         String winnerText = "Wygrywa: " + winner;
163 
164         Popup popup = createPopup(winnerText);
165         popup.addEventFilter(MouseEvent.MOUSE_CLICKED,
166                 new EventHandler<MouseEvent>() {
167                     @Override
168                     public void handle(MouseEvent event) {
169                         newGame();
170                     }
171                 });
172     }
173 
174     /*
175      * Metoda odpowiedzialna za utworzenie popupu z tekstem przekazanym jako
176      * argument
177      */
178     private Popup createPopup(String text) {
179         TextArea popupText = new TextArea(text);
180         popupText.setPrefWidth(200);
181         popupText.setPrefHeight(100);
182         popupText.setEditable(false);
183 
184         Popup popup = new Popup();
185         popup.setAutoFix(true);
186         popup.getContent().addAll(popupText);
187 
188         popup.show(button00.getScene().getWindow());
189         popup.addEventFilter(MouseEvent.MOUSE_CLICKED,
190                 new EventHandler<MouseEvent>() {
191                     @Override
192                     public void handle(MouseEvent event) {
193                         newGame();
194                         popup.hide();
195                     }
196                 });
197 
198         return popup;
199     }
200 }
```

Wszystkie pola klasy oznaczone adnotacją @FXML to przyciski zdefiniowane w pliku fxml. Oprócz tego utworzyliśmy zmienną reprezentującą nasz Model a także tablicę, do której przypiszemy wszystkie przyciski w celu ich wygodniejszym zarządzaniem i możliwością robienia tego w pętli. Praktycznie cały pozostały kod moglibyśmy umieścić w metodzie *initialize()*, jednak dobrą praktyką jest rozbijanie go na mniejsze i bardziej czytelne elementy. Dobrą zasadą, którą należy się kierować jest to, żeby dana metoda była odpowiedzialna tylko za jedną rzecz. Takim sposobem posiadamy zestaw metod:

- *newGame()* - resetuje grę, czyli tworzy nowy model oraz czyści tekst na każdym przycisku
- *addButtonAction()* - dodaje do przycisków na planszy obsługę zdarzeń. Dzięki temu, że buttony przypisaliśmy do dwuwymiarowej tablicy możemy to zrobić w wygodny sposób w pętli. Akcje polegają na tym, że po wci-

śnięciu dowolnego przycisku ustawiamy na nim tekst zgodny z aktualnym graczem (chyba, że dany przycisk był już wcześniej wciśnięty), a następnie sprawdzeniu, czy aktualny stan gry doprowadził do sytuacji, gdzie znamy już zwycięzcę.

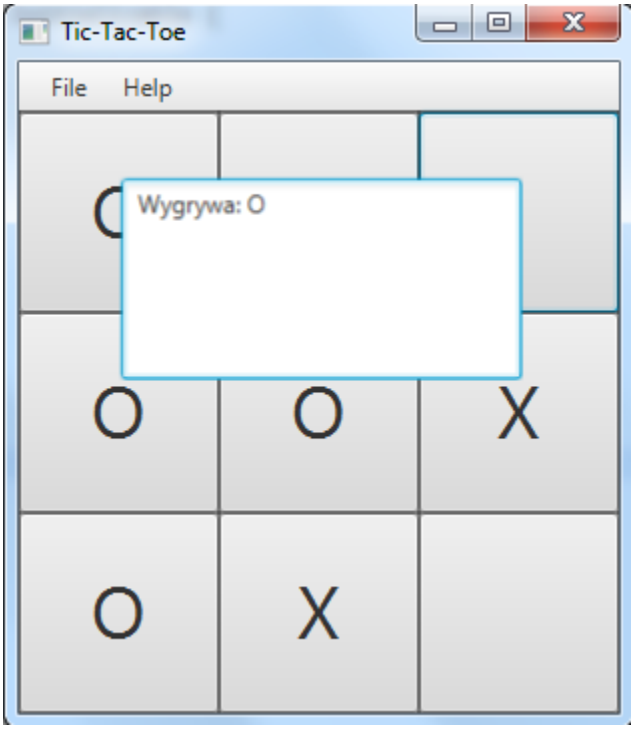
- *addMenuAction()* - podpięcie obsługi zdarzeń do 3 elementów menu. W przypadku opcji *New Game* po prostu wywołujemy metodę *newGame()*, dla opcji *Close* zamykamy program poprzez wywołanie metody *Platform.exit()*, a przy opcji *About* tworzymy nowy popup, z krótką informacją o programie.
- *updateView()* jest metodą odpowiedzialną za aktualizację tego co jest wyświetlane na przyciskach na podstawie stanu gry zapisanego w obiekcie modelu.
- *showWinnerPopup()* jest wywoływana w przypadku, gdy w grze znaleziono zwycięzcę i w takiej sytuacji wyświetlany jest popup z odpowiednią informacją
- *createPopup()* odpowiada za tworzenie popupów z tekstem przekazanym jako argument. Wykorzystujemy do tego celu obiekt klasy Popup z odpowiednio ustawionym tekstem w obiekcie TextArea, w którym wyłączamy możliwość edycji dzięki metodzie setEditable(false). Istotnym elementem jest podpięcie zdarzenia kliknięcia na dany popup, który spowoduje jego zamknięcie oraz zresetowanie gry.

3.8.5 Gotowa aplikacja

Na koniec dobrze nam już znana klasa uruchomieniowa programu, w której tworzymy widok na podstawie definicji fxml, ustawiamy odpowiedni tytuł okna i wyświetlamy aplikację użytkownikowi.

plik Main.java

```
package application;

import javafx.application.Application;
import javafx.fxml.FXMLLoader;
import javafx.scene.Scene;
import javafx.scene.layout.VBox;
import javafx.stage.Stage;

public class Main extends Application {
    @Override
    public void start(Stage primaryStage) {
        try {
            VBox root = (VBox) FXMLLoader.load(getClass().getResource(
                    "TicTacToe.fxml"));
            Scene scene = new Scene(root);
            scene.getStylesheets().add(
                    getClass().getResource("application.css").toExternalForm());
            primaryStage.setTitle("Tic-Tac-Toe");
            primaryStage.setScene(scene);
            primaryStage.show();
        } catch (Exception e) {
            e.printStackTrace();
        }
    }

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


Tic-Tac-Toe
File
Help
Wygrywa: O
O
O
X
O
X